Vorlesungsbegleiter

Allgemeine Informatik III (2/2)
(Systemnahe Software I)

Prof. Dr. F. Schweiggert

Abt. Angewandte Informationsverarbeitung

Universität Ulm

WS 1999/2000 und folgende

Literatur

jedes "gute" Buch über C (unter Unix)

z.B.:

Kernighan, B.W.; Ritchie, D.M.: *The C Programming Language (second edition)*. Prentice Hall

Schreiner,A.: *Systemprogrammierung in UNIX. Teil I: Werkzeuge*. B.G.Teubner

zusätzlich:

Kernighan, B.W.; Pike, R.: *The UNIX Programming Environment*. Prentice Hall

Inhalt

C

1. Vorbemerkungen

- entwickelt 1972-73 von Dennis Ritchie bei den Bell Laboratories von AT&T
- 1978 erfolgten einige Erweiterungen von C (*enum, void, structure assignment,* ...), die den sog. Kernighan/Ritchie-Standard (K&R) bilden
- Der ANSI-Standard beinhaltet eine Reihe von Erweiterungen und Aufräumarbeiten
- *C++* und *Objective C* sind modulare / objektorientierte Erweiterungen von C.
- UNIX ist zum größten Teil in C geschrieben, ein UNIX-Kern besteht nur aus 5-10% Assemblertext
- C ist eine maschinennahe Sprache. Array's sind Speicherflächen, Array-Namen werden als Zeiger auf das erste Element aufgefaßt, Zugriff auf Elemente erfolgt via Zeigerarithmetik (Adreß-Rechnung)
- C wurde mit UNIX verbreitet und ist damit eine der "portabelsten" Plattformen (Bibliotheken?)

Aufbau eines C-Programms

```
$ C_Program  = { Definition }.
$ Definition = FunctionDefinition | DataDefinition
$              | ExternDeclaration | TypeDefinition.
```

- ein C-Programm ist eine beliebige Folge von Vereinbarungen

- Zu den Vereinbarungen gehören Funktionsdefinitionen, Typnamen-Vereinbarungen und Variablenvereinbarungen. Jede Funktionsdefinition hat einen Vereinbarungs- und Aktionsteil.
- Funktionen sind immer global, d.h. sie dürfen nicht geschachtelt werden
- Die Parameterübergabe ist *Call by Value*
- Bei Prozeduren (Funktionen ohne Resultat) wird *void* als *FunctionType* angegeben. Ist kein Ergebnistyp angegeben, so wird **implizit** *int* (Integer) angenommen!

```
$ FunctionDefinition = FunctionType FunctionHeader
$                      CompoundStatement.
$ CompoundStatement  = "{" { Declaration }
$                          { statement } "}".
$ Statement          = [ Expression ] ";"
$                      | CompoundStatement | ... .
```

Blockstruktur (anders als in Modula-2 oder Oberon):
Zu Beginn eines jeden *CompoundStatement* dürfen Variablen-Vereinbarungen getroffen werden; damit können mehrere Anweisungen zusammengefaßt werden und Sichtbarkeits- / Lebensdauerbereiche definiert werden:

```
if ( x > y ) /* x und y vertauschen */
  {
      int tmp;                          Declaration

      tmp = x; x = y; y = tmp;          Statements
  }
```

CompoundStatement

- Die Gültigkeit von *tmp* erstreckt sich auf das *CompoundStatement.*
- Mit “`int tmp;`” wird eine (lokale) Variablen mit Datentyp *int* deklariert. *int* ist ein Schlüsselwort und steht für *integer.*
- “=” ist der "Zuweisungsoperator" (Expression mit Nebeneffekt).
  “`tmp = x;`” gilt als Ausdruck und hat einen Wert, nämlich den zugewiesenen Wert, also den von x.
  Eine Zuweisung der Form “
  `x = y = 1;`”
  ist also möglich, die Bewertung des Ausdrucks erfolgt von von rechts nach links:
  zuerst: “`y=1`”, d.h. y bekommt den Wert 1; der Wert des Ausdrucks ist der zugewiesene Wert (also 1);
  dieser Wert wird dann `x` zugewiesen; der Wert des Ausdrucks “`x=...`” ist wiederum der zugewiesene Wert!

2. Beispiel-Programme

2.1 Wie immer am Anfang

```
/* Ein erstes Beispiel: hallo.c */

main() {
/* printf: formattierte Ausgabe nach stdout */
   printf("Hallo!!!\n");
}
```

```
byron$ gcc -Wall hallo.c # give all warnings
```

```
hallo.c:3: warning: return-type defaults to `int'
hallo.c: In function `main':
hallo.c:5: warning: implicit declaration of function `printf'
hallo.c:6: warning: control reaches end of non-void function
byron$
```

Verbessert:

```
/* hallo1.c */

#include <stdio.h>
/* in stdio.h ist u.a. die Funktion
 * printf vereinbart!
 */

int main() {
   printf("Hallo!!!\n");
   /* terminieren und Exit-Status 0 liefern! */
   exit(0);
}
```

```
byron$ gcc -Wall hallo1.c
byron$ a.out
Hallo!!!
byron$ echo $?
0
byron$
```

Anm.: Der Exit-Status eines Programmes - Funktion **exit(x)** - kann auf Shell-Ebene unmittelbar nach Beendigung des Programms durch **echo $?** abgefragt werden. Üblicherweise wird bei Erfolg **0** zurückgegeben.

Achtung: Bei C ist der Wert **0 FALSE**!!!

2.2 Fahrenheit nach Celsius

mit einer while-Schleife

```
/*      fahren.c: - mit while-Schleife
 *      Umwandlung von Fahrenheit in Celsius
 *      fuer f = 0 (20) 300
 */

#include <stdio.h>

/* globale 'Konstanten'-Vereinbarungen: */

const unsigned int OG = 300;
const unsigned int UG = 0;

int main() {
   int step = 20;
   float fahr, celsius;

   fahr = UG;
   while (fahr <= OG) {
      celsius = (fahr - 32.0) * (5.0/9.0);
        /* 5/9 waere ganzzahlige Division
           und somit 0, deshalb 5.0/9.0
         */
      printf("%4.0f Fahr. : %6.1f Celsius\n",fahr,celsius);
      fahr = fahr + step;
   }
   exit(0);
}
```

mit einer for-Schleife

```
/*      fahren1.c: mit for-Schleife
        Umwandlung von Fahrenheit in Celsius
        fuer f = 0 (20) 300
*/

#include <stdio.h>

const unsigned int OG = 300;
const unsigned int UG = 0;
const unsigned int STEP = 20;

int main() {
   int fahr;
   printf("%4s     :       %7s\n\n", "FAHR","CELSIUS");

   for (fahr=UG; fahr <= OG; fahr=fahr+STEP)
      printf("%4d     :       %6.1f\n",fahr,
                   (5.0/9.0)*(fahr-32));
   exit(0);
}
```

2.3 Euklid's Algorithmus

```
/*       euklid.c:      */
#include <stdio.h>

int main() {
   int x,y, x0,y0;
   printf("Geben Sie zwei pos. ganze Zahlen ein:");
   if ( scanf("%d %d", &x, &y) != 2)
     exit(1);
   x0 = x; y0 = y;
   while (x != y)
      if (x > y)
         x = x-y;
      else
         y = y-x;
   printf("ggT von %d und %d ist %d\n",x0,y0,x);
   exit(0);
}
```

In der Funktion *scanf* wird deutlich, daß C nur Wertparameterübergabe kennt. Damit die Variablen x und y auch außerhalb von *scanf* einen Wert bekommen, wird an *scanf* ein Zeiger auf diese Variable übergeben, so daß der Zeigerwert zwar lokal zu *scanf* ist, das Objekt, auf das gezeigt wird, ist aber "global". Mit dem Operator **&** wird ein Zeiger "konstruiert".

3. Etwas vorab zum Preprozessor

3.1 Textersatz

Eine andere Technik zur Realisierung von Unterprogrammen (bei Aufruf des Unterprogramms durch Nennung des Namens wird der zugehörige Anweisungsteil angesprungen) ist der **offene Einbau**:

- Das Unterprogramm hat einen Namen
- Der "Aufruf" erfolgt wieder durch Nennung des Namens (ggf. mit Parametern)
- an die Aufrufstelle wird der zugehörige Text "**kopiert**" - dies geschieht beim "Compilieren", also vor der Ausführung

<u>Vorteil:</u> Abarbeitung des Aufrufs ist schneller (kein Anspringen, keine Stack-Verarbeitung)

<u>Nachteil:</u> Programmtext wird mit jeder Aufrufstelle größer

Diese Technik heißt auch **Makrotechnik**. Das Programm, das diesen Textersatz durchführt, heißt **Makro-Prozessor** und ist bei Programm-Quelltexten dem eigentlichen Compiler vorgeschaltet; in diesem Zusammenhang spricht man dann vom **Preprocessor** ("Präprozessor").

3.2 cpp - C-Preprocessor

3.2.1 Makro-Definitionen

Dem eigentlichen C-Compiler ist ein Makroprozessor vorgeschaltet, der automatisch mit dem Aufruf von **gcc** aktiviert wird. Er kann auch direkt aktiviert werden (man gcc, man g++). Die von ihm erkannten Makros beginnen (bei vielen C-Compilern als erstes Zeichen in der Zeile) mit einem **#**, dem (meist unmittelbar danach) ein Makroname mit dem zugehörigen Ersatztext folgt. Der Ersatztext wird in die Eingabe zurückgestellt, also weiter auf Makroaufrufe untersucht.

Will man nur den Preprocessor aktivieren, so kann dies z.B. über die Option **-E** erfolgen.

3.2.2 define

define ist ein Makroname, dessen Ersatztext "Nichts" ist, der aber gleichzeitig eine Makrodefinition erledigt.

Beispiel:

```
/*      fahren2.c: mit Makro's statt const-Vereinbarung
        Umwandlung von Fahrenheit in Celsius
        fuer f = 0 (20) 300
*/

#define OG 300
#define UG 0

int main() {
   const int step = 20;
   float fahr, celsius;

   fahr = UG;
   while (fahr <= OG) {
       celsius = (fahr - 32.0) * (5.0/9.0);
       /* 5/9 waere ganzzahlige Division
          und somit 0, deshalb 5.0/9.0
        */
       printf("%4.0f Fahr. : %6.1f Celsius\n",fahr,celsius);
       fahr = fahr + step;
   }
   exit(0);
}
```

3.2.3 cpp - Output

```
thales$ gcc -E fahren2.c
# 1 "fahren2.c"









void main()
{
   const int step = 20;
   float fahr, celsius;

   fahr = 0 ;
   while (fahr <= 300 ) {
       celsius = (fahr - 32.0) * (5.0/9.0);



       printf("%4.0f Fahr. : %6.1f Celsius\n",fahr,celsius);
       fahr = fahr + step;
   }
   exit(0);
}
```

Will man die Kommentare erhalten, so sieht das wie folgt aus:

```
thales$ gcc -E -C fahr.c
# 1 "fahren2.c"
/*      fahren2.c: mit Makro’s statt const-Vereinbarung
        Umwandlung von Fahrenheit in Celsius
        fuer f = 0 (20) 300
*/




void main()
{
   const int step = 20;
   float fahr, celsius;

   fahr = 0 ;
   while (fahr <= 300 ) {
       celsius = (fahr - 32.0) * (5.0/9.0);
       /* 5/9 waere ganzzahlige Division
          und somit 0, deshalb 5.0/9.0
        */
```

```
        printf("%4.0f Fahr. : %6.1f Celsius\n",fahr,celsius);
        fahr = fahr + step;
    }
    exit(0);
}
```

3.2.4 include

include ist ein Makro, dessen Ersatzttext der Inhalt der folgenden Datei ist. Damit werden i.a. Vereinbarungen oder andere Makro's "hereinkopiert". Diese Dateien heißen **Header-Files** und haben üblicherweise die Endung **.h**.

Es gibt eine ganze Reihe solcher "Header-Files" im Katalog **/usr/include** - diese stellen die C-Bibliothek dar und entsprechen im Prinzip den *DEFINITION MODULE's* von Modula-2. Dies ist, wie später gezeigt wird, auch der Weg zur Modularisierung in C.

Da Makro's zu einem *inline* -Text führen, kann man damit auch Prozeduren realisieren, bei denen "kein Ansprung" erfolgt, die somit i.a. schneller sind. Dazu können Makro's auch parametrisiert werden (dazu und zu den weiteren Inhalten der *Header-Files* später mehr).

4. Quellformat

4.1 Kommentare

Kommentare beginnen mit “/*”, enden mit “*/”, und können (im ANSI-Standard) nicht geschachtelt werden.

4.2 Namen

Namen bestehen aus Buchstaben und Ziffern, wobei das erste Zeichen ein Buchstabe sein muß. Zu den Buchstaben wird der *Underscore* (also "_") gezählt.

4.3 Reservierte Worte

Die folgende Tabelle enthält die wichtigsten Schlüsselworte von C - einige wie *auto* oder *register* sind heute i.a. ohne Bedeutung.

break	case	char	const	continue
default	do	double	else	enum
extern	float	for	goto	if
int	long	return	short	signed
sizeof	static	struct	switch	typedef
union	unsigned	void	while	

5. Operatoren - Übersicht

<table>
<tr><td>1</td><td>primary</td><td>() [] . -></td><td>von links</td></tr>
<tr><td>2</td><td>unary</td><td>! ~ - ++ -- & *
(type) sizeof</td><td>von rechts</td></tr>
<tr><td>3</td><td>multiplicative</td><td>* / %</td><td>von links</td></tr>
<tr><td>4</td><td>additive</td><td>+ -</td><td>von links</td></tr>
<tr><td>5</td><td>shift</td><td><< >></td><td>von links</td></tr>
<tr><td>6</td><td>relational</td><td>< <= > >=</td><td>von links</td></tr>
<tr><td>7</td><td>equality</td><td>== !=</td><td>von links</td></tr>
<tr><td>8</td><td>bitwise AND</td><td>&</td><td>von links</td></tr>
<tr><td>9</td><td>bitwise OR (excl.)</td><td>^</td><td>von links</td></tr>
<tr><td>10</td><td>bitwise OR (incl.)</td><td>|</td><td>von links</td></tr>
<tr><td>11</td><td>log. AND</td><td>&&</td><td>von links</td></tr>
<tr><td>12</td><td>log. OR</td><td>||</td><td>von links</td></tr>
<tr><td>13</td><td>conditional</td><td>? :</td><td>von rechts</td></tr>
<tr><td>14</td><td>assignment</td><td>= += -= *= /= %=
>>= <<= &= ^=</td><td>von rechts</td></tr>
<tr><td>15</td><td>comma</td><td>,</td><td>von links</td></tr>
</table>

6. Ein-/Ausgabe (stdin, stdout, stderr)

6.1 Ausgabe nach stdin - printf()

printf(formatstring, wert1, wert2, ...)

- formatstring:
  In Doppelapostrophen eingeschlossene Zeichenfolge (konstanter String) oder Variable mit einem Zeichenkettenwert, bestehend aus freiem Text und eingestreuten Formatierungsangaben für die nach dem *formatstring* folgenden Wertangaben (Variable, Ausdrücke).
- Formatierungsangaben beginnen mit einem **%** gefolgt von einem Zeichen zur Format-Modifikation und einem Zeichen zur Konvertierung.
- Wert (Variable oder Ausdruck)

Zeichen f. Konvert.	Effekt
d,i,o,u,x,X	für Integer-Ausgabe; d und i für Dezimaldarstellung; u für unsigned; x,X für Hexadezimal, wobei x die Kleinbuchstaben a-f, X die Großbuchstaben A-F benutzt; unmittelbar davor kann ein h (für short int) oder l (für long int) stehen
f	Reelle Zahlen in Dezimalform ([-]mmm.nnnnnnn); Anzahl der Nachkommastellen durch Genauigkeitsangabe; default: 6
e,E	Reelle Zahlen in Exponentialform
c	gibt ein Zeichen aus; das zugehörige Datenargument wird als int übergeben, das letzte Byte wird ausgegeben
s	für Strings; Argument muß Zeiger auf Char-Vektor mit abschließendem Null-Byte sein; Genauigkeitsangabe wird als maximale auszugebende Zeichenzahl interpretiert
%	die Folge %% gibt ein % aus

Flags (opt.)	Bedeutung
-	linksbündig
+	auch pos. Vorzeichen wird ausgegeben
Blank	statt pos. Vorzeichen ein Blank

Minimale Feldgröße (optional):

Dezimalkonstante, die die minimale Anzahl auszugebender Stellen bestimmt; mit einem ***** wird das nächste Datenargument als minimale Feldgröße benutzt:

Beispiel:

```
thales$ cat feldgr.c
# include <stdio.h>

void main() {
   printf("%%5d\t%5d\n",25);
   printf("%%5.3d\t%5.3d\n",25);
   printf("%%.3d\t%.3d\n",25);
   printf("%%4.3f\t%4.3f\n",33.1);
   printf("%%4.3f\t%4.3f\n",33.123456789);
   printf("%%4.3e\t%4.3e\n",33.123456789);
   printf("%%.15s\t%.15s\n","Gib nur die\
   ersten 10 Zeichen aus.");
   printf("%%*d\t%*d\n",5,25);
}
thales$ gcc feldgr.c
thales$ a.out
%5d        25
%5.3d     025
%.3d    025
%4.3f   33.100
%4.3f   33.123
%4.3e   3.312e+01
%.15s   Gib nur die
%*d        25
thales$ a.out | od -bc > feldgr.dump
thales$ cat feldgr.dump # etwas vereinfacht
045 065 144 011 040 040 040 062 065 012 045 065 056 063 144 011
  %   5   d  \t               2   5  \n   %   5   .   3   d  \t
040 040 060 062 065 012 045 056 063 144 011 060 062 065 012 045
          0   2   5  \n   %   .   3   d  \t   0   2   5  \n   %
064 056 063 146 011 063 063 056 061 060 060 012 045 064 056 063
  4   .   3   f  \t   3   3   .   1   0   0  \n   %   4   .   3
```

```
146 011 063 063 056 061 062 063 012 045 064 056 063 145 011 063
  f  \t   3   3   .   1   2   3  \n   %   4   .   3   e  \t   3
056 063 061 062 145 053 060 061 012 045 056 061 065 163 011 107
  .   3   1   2   e   +   0   1  \n   %   .   1   5   s  \t   G
151 142 040 156 165 162 040 144 151 145 040 040 040 040 012 045
  i   b       n   u   r       d   i   e                  \n   %
052 144 011 040 040 040 062 065 012
  *   d  \t               2   5  \n

thales$
```

6.2 Ausgabe nach stderr - fprintf()

fprintf(stream, formatstring, wert1, wert2, ...)

Im Prinzip wie *printf* - zusätzlich als erstes Argument die Angabe der Ausgabeverbindung (File Pointer - für Diagnosemeldungen ist in *stdio.h* die Variable / das Makro *stderr* definiert) !

Beispiel:

```
hypatia$ cat diagnose.c

/* diagnose.c */

# include <stdio.h>

void main() {
  fprintf(stderr, "Fehlermeldungen wie Diagnosehinweise dahin\n");
  printf("Die ermittelten Ergebnisse dorthin\n");
}

hypatia$ gcc -Wall diagnose.c

hypatia$ a.out >res 2>err

hypatia$ cat res
Die ermittelten Ergebnisse dorthin
hypatia$ cat err
Fehlermeldungen wie Diagnosehinweise dahin
hypatia$
```

6.3 Eingabe von stdin - scanf()

anzahl = scanf(formatstring, Adresse1, Adresse2, ...)

- Format-String:
  Zeichen für die Konvertierung und weitere Zeichen
- Konvertierung:
  % gefolgt von optionalen Zeichen zur Modifikation gefolgt von Konvertierungszeichen

Andere Zeichen (außer Konvertierungszeichen und Leeraum) müssen mit Zeichen im Eingabestrom übereinstimmen. Leerzeichen, Tab's (\t) und Newline (\n) veranlassen *scanf()* zum nächsten Nicht-Leerzeichen weiterzugehen.

Beachte : C kennt nur Wertparameterübergabe. Aus diesem Grund muß speziell bei *scanf()* der Adreßoperator vor den aktuellen Parameter gestellt werden (Ausnahme: Vektoren - dazu später mehr). Damit wird ein Zeiger auf die Variable übergeben und *scanf()* greift über diesen Zeiger auf das "Orginal" durch!

Bsp.: *scanf("%d", &n);*

Damit wird in die Integer-Variable n ein Wert von *stdio* eingelesen.

scanf() hat einen ganzzahligen *return*-Wert, der die Zahl der gelesenen Werte wiedergibt, also die Anzahl der tatsächlich erfolgten Zuweisungen.

Konv.Zeichen	Wirkung
d	Dezimale Integer
o	Oktal-Konstante
x,X	Hexadezimalkonstante
f	Floating-Point
s	String; Zeichenfolge bis zum nächsten Leerraum (Blank, \t, \n); Null-Byte ('\0') wird angefügt
c	Nächstes Zeichen; um das nächste Nicht-Leerraum-Zeichen zu lesen: %1c
%	Liest %-Zeichen ohne Zuweisung

Maximale Feldlänge: Dezimal-Zahl

Flag: * liest, unterdrückt aber Zuweisung

```
thales$ cat scan.c
/*
 * scan.c
 */

#include <stdio.h>

void main()
{
  int i, digits, anzahl;
  float pi, x;
  char name[50];

  anzahl = scanf("Der Wert von pi auf %d Stellen ist %f",\
                      &digits, &pi);

  printf("%d Arg., pi: %f\n", anzahl, pi);

  anzahl = scanf("%2d %f %*d %2s", &i, &x, name);

  printf("%d Args, i: %d x: %f name: %s\n", \
                      anzahl, i, x, name);

}
thales$ gcc scan.c
thales$ a.out
Der Wert von pi auf 2 Stellen ist 2.17
2 Arg., pi: 2.170000
12345 0123 12a34
3 Args, i: 12 x: 345.000000 name: 12
thales$ a.out
Der WWWert von pi auf 2 Stellen ist 2.17
0 Arg., pi: 0.000000
0 Args, i: -268436676 x: 0.000000 name:
thales$ a.out
123 Der Wert von pi auf 2 Stellen ist 2.17
0 Arg., pi: 0.000000
2 Args, i: 12 x: 3.000000 name:
thales$
```

6.4 Weitere Ein-/Ausgabe-Funktionen

6.4.1 fgetc, fgets, getc, getchar, gets, ungetc

```
SYNOPSIS
       #include <stdio.h>

       int fgetc(FILE *stream);
       char *fgets(char *s, int size, FILE *stream);
       int getc(FILE *stream);
       int getchar(void);
       char *gets(char *s);
       int ungetc(int c, FILE *stream);

DESCRIPTION
       fgetc() reads the next character from stream and returns
       it as an unsigned char cast to an int, or EOF on end of
       file or error.

       getc() is equivalent to fgetc()  except  that  it  may  be
       implemented as a macro which evaluates stream more than
       once.

       getchar() is equivalent to getc(stdin).

       gets() reads a line from stdin into the buffer pointed to
       by s until either a terminating newline or EOF, which it
       replaces with '\0'.  No check for buffer overrun is
       performed.

       fgets() reads in at most one less than size characters
       from stream and stores them into the buffer pointed to by
       s.  Reading stops after an EOF or a newline. If a newline
       is read, it is stored into the buffer.  A '\0' is stored
       after the last character in the buffer.

       ungetc() pushes c back to stream, cast to unsigned char,
       where it is  available  for  subsequent  read  operations.
       Pushed-back characters will  be  returned in reverse
       order; only one pushback is guaranteed.

       Calls to the functions described here can  be  mixed  with
       each  other and with calls to other input functions from
       the stdio library for the same input stream.

RETURN VALUES
       fgetc(), getc() and getchar() return the character read as
       an unsigned char cast to an int or EOF on end of file or
       error.

       gets() and fgets() return s on success, and NULL on  error
       or  when  end of file occurs while no characters have been
       read.

       ungetc() returns c on success, or EOF on error.
```

```
CONFORMING TO
       ANSI - C, POSIX.1

BUGS
       Because it is impossible to tell without knowing the  data
       in  advance  how  many  characters  gets() will read, and
       because gets() will continue to store characters past the
       end  of  the buffer, it is extremely dangerous to use. It
       has been used to break computer security. Use fgets()
       instead.

       It  is  not advisable to mix calls to input functions from
       the stdio library with low - level calls to read() for the
       file  descriptor  associated  with  the  input stream; the
       results will be undefined and very probably not what  you
       want.

SEE ALSO
       read(2),  write(2), fopen(3), fread(3), scanf(3), puts(3),
       fseek(3), ferror(3)
```

6.4.2 fputc, fputs, putc, putchar, puts

```
SYNOPSIS
       #include <stdio.h>

       int fputc(int c, FILE *stream);
       int fputs(const char *s, FILE *stream);
       int putc((int c, FILE *stream);
       int putchar(int c);
       int puts((ccoonnsstt char **_s);

DESCRIPTION
       fputc() writes the character c, cast to an unsigned char,
       to stream.

       fputs() writes the string s to stream, without its trailing '\0'.

       putc() is equivalent to fputc()  except that it may be
       implemented as a macro which evaluates stream more than
       once.

       putchar(c) is equivalent to putc(c,stdout).

       puts() writes the string s and a trailing newline to stdout.

       Calls to the functions described here can be mixed with
       each other and with calls to other output functions from
       the stdio library for the same output stream.

RETURN VALUES
       fputc(), putc() and putchar() return the character written
       as an unsigned char cast to an int or EOF on error.

       puts() and fputs() return a non-negative number on success,
       or EOF on error.
```

```
CONFORMING TO
       ANSI - C, POSIX.1

BUGS
       It is not advisable to mix calls to output functions from
       the stdio library with low-level calls to write() for
       the file descriptor associated with the same output
       stream; the results will be undefined and very probably
       not what you want.

SEE ALSO
       write(2), fopen(3), fwrite(3), scanf(3), gets(3),
       fseek(3), ferror(3)
```

7. Objekte und L-Werte

Objekte sind allgemein modifizierbare Speicherflächen. L-Werte sind Ausdrücke, die ein Objekt bezeichnen (z.B. Variablenname). Da die Wertzuweisung in C ein Ausdruck ist, steht der Begriff **L-**Wert für die **linke** Seite in diesem Ausdruck, in dem als Nebeneffekt eben diese linke Seite verändert wird.

L-Werte können Operanden und Resultatwerte von unitären Operatoren (Auswahl eines Vektorelements oder einer Strukturkomponente) sein
- diese können somit auch links vom "Zuweisungsoperator" stehen. Dies gilt besonders für den Dereferenzierungsoperator ***** : ist p ein Zeigerwert, so ist ***p** das Objekt, auf das **p** zeigt.

8. Operanden

```
$ operand = name | konstante | zeichenkette
$           | "(" ausdruck ")" | operand"("[argumentliste]")"
$           | operand"["ausdruck"]" | operand"."name
$           | operand"->"name.
$ argumentliste = zuweiseung {"," zuweisung }.
```

name:

- ein Operand, meist L-Wert, wenn geeignet vereinbart
- Typ folgt aus der Namensvereinbarung
- ist er in einer Aufzählung eingeführt (Aufzähltyp), so ist es eine Konstante u. somit kein L-Wert.
- bezeichnet er einen Vektor, so gilt er als Adresse des ersten Elements (konstanter Zeigerwert, kein L-Wert)
- bezeichnet er eine Strukturvariable, so ist er ein L-Wert
- bezeichnet er eine Funktion (nicht beim Aufruf), so ist er eine Adresse (konstanter Zeigerwert)

konstante

- ihr Typ ergibt sich aus der Definition, kein L-Wert;
- Zeichen (Characters) sind vom Typ **int**

zeichenkette

- repräsentiert einen konstanten Zeigerwert auf das erste Zeichen, kein L-Wert

argumentliste

- kann fehlen, runde Klammern aber dennoch notwendig;
- Wertparameter
- Reihenfolge der Bewertung **nicht** definiert

Vektorindizierung

- ganzzahlig, unterster Index ist **0**

9. Operatoren im Einzelnen

9.1 unitäre Operatoren

```
$ unitaer = operand | operand"++" | operand"--"
$           | "*"unitaer | "&"unitaer | "-"unitaer
$           | "!"unitaer | "˜"unitaer | "++"unitaer
$           | "--"unitaer | "("typangabe")" unitaer
$           | "sizeof" unitaer | "sizeof""("typangabe")".
```

- Inkrement ++, Dekrement --;
  Argument muß L-Wert sein
- **(a)++ (a)--** liefern als Ergebnis den Wert **vor** Inkrementierung / Dekrementierung
- **++(a) --(a)** liefern als Ergebnis den Wert **nach** Inkr. / Dekr.
- ***** Derefenzierungsoperator
- **&** Adreßoperator, Argument ist L-Wert
- **!** logisches Komplement ("Negation"); Ergebnis ist 1, falls Operand den Wert 0 hatte, 0 sonst
- **sizeof** liefert den Speicherbedarf in *Bytes,* bei einem Vektor die Größe des gesamten Vektors
- **˜** Bitweises Komplement (1-er Komplement)

Beispiel zum bitweisen 1-er Komplement:

```
thales$ cat komplement.c
/*
 * komplement.c
 */

void main() {
   long int i;
   long int j;
   i=0;
   printf("i= %o (dez.: %d) und 1-Komplement: %o\n", i,i,˜i);
   i=1;
   printf("i= %o (dez.: %d) und 1-Komplement: %o\n", i,i,˜i);
   i = 2;
   printf("i= %o (dez.: %d) und 1-Komplement: %o\n", i,i,˜i);
   i = 1024;
   printf("i= %o (dez.: %d) und 1-Komplement: %o\n", i,i,˜i);
```

```
    j=1;
    for(i=1;i<31;i++)
         j = j*2;
    printf("j= %o (dez.: %d) und 1-Komplement: %o\n", j,j,~j);
}

thales$ gcc komplement.c
thales$ a.out
i= 0 (dez.: 0) und 1-Komplement: 37777777777
i= 1 (dez.: 1) und 1-Komplement: 37777777776
i= 2 (dez.: 2) und 1-Komplement: 37777777775
i= 2000 (dez.: 1024) und 1-Komplement: 37777775777
j= 10000000000 (dez.: 1073741824) und 1-Komplement: 27777777777
thales$
```

9.2 Binäre Operatoren

```
$ binaer = unitaer | binaer "||" binaer
$          | binaer "&&" binaer | binaer "|" binaer
$          | binaer "^" binaer | binaer "&" binaer
$          | binaer "==" binaer | binaer "!=" binaer
$          | binaer "<" binaer | binaer "<=" binaer
$          | binaer ">" binaer | binaer ">=" binaer
$          | binaer "<<" binaer | binaer ">>" binaer
$          | binaer "+" binaer | binaer "-" binaer
$          | binaer "*" binaer | binaer "/" binaer
$          | binaer "%" binaer.
```

- **&&** logisches UND, **||** logisches ODER.
  Operanden müssen mit Null vergleichbar sein; Resultat ist 0 oder 1; 0 entspricht FALSE; 1 (ungleich 0) entspricht TRUE
- **&** bitweises UND,
  | bitweises inklusives ODER,
  ^ bitweises exklusives ODER

  Operanden müssen Integer sein, Ergebnis ist Integer
- **Vergleiche**
  Resultat ist Integer, **1** wenn Vergleich zutrifft, **0** sonst

Beispiel:

```
byron$ cat bits.c
# include <stdio.h>

void main() {
   unsigned long int z1, z2;
   z1 = 5; z2 = 6;
   /*Bitmuster fuer z1: 00...000101
    *Bitmuster fuer z2: 00...000110
    */

    printf("z1=%ld (0101), z2=%ld(0110)\n", z1,z2);
    printf("z1 | z2: %ld\n", z1 | z2);
    printf("z1 & z2: %ld\n", z1 & z2);
    printf("z1 ^ z2: %ld\n", z1 ^ z2);
    printf("z1 << 2: %ld\n", z1 << 2);
    printf("z2 >> 1: %ld\n", z2 >> 1);
}
byron$ gcc -Wall bits.c
byron$ a.out
z1=5 (0101), z2=6(0110)
z1 | z2: 7
z1 & z2: 4
z1 ^ z2: 3
z1 << 2: 20
z2 >> 1: 3
byron$
```

9.3 Auswahl

```
$ auswahl = binaer | binaer "?" auswahl ":" auswahl.
```

Die **auswahl** besteht aus einer Bedingung, der zwei Ausdrücke folgen. Die Bedingung wird bewertet: ist der Wert ungleich Null, so wird der erste Ausdruck als Ergebnis genommen, ist er gleich Null, so der zweite.

Beispiel:

```
byron$ cat auswahl.c
# include <stdio.h>

void main() {
  printf("%s\n", 2 > 3 ? "2 > 3" : "NOT 2 > 3");
}
byron$ gcc -Wall auswahl.c
byron$ a.out
NOT 2 > 3
byron$
```

9.4 Zuweisungen (Expression's)

```
$ zuweisung = auswahl
$             | unitaer "=" zuweisung
$             | unitaer "| =" zuweisung
$             | unitaer "^ =" zuweisung
$             | unitaer "& =" zuweisung
$             | unitaer "<< =" zuweisung
$             | unitaer ">> =" zuweisung
$             | unitaer "+ =" zuweisung
$             | unitaer "- =" zuweisung
$             | unitaer "* =" zuweisung
$             | unitaer "/ =" zuweisung
$             | unitaer "% =" zuweisung.
```

a = a *op* **(b)**

kann effizienter als

a *op* **= b**

angegeben werden, da **a** nur einmal bewertet werden muß

Beispiele:

```
byron$ cat zuweisung.c
# include <stdio.h>
void main() {
   int a, b;
   a=7; b=15;

   printf("==========================================\n");
   printf("a= %2d; b= %2d\n",a,b);
   printf("------------------------------------------\n");
   printf("a= %%2d : a = %2d\n", a);
   printf("a |= 0xF  -> a= %%2d: a=%2d\n", a |= 0xF);
   /* Reihenfolge der Parameterbewertung??? */
   printf("b= %%2d : b |= 0x3F -> b= %%2d: %d / %d\n",b, b |= 0x3F);
   a=5; b=17;
   printf("==========================================\n");
   printf("a= %2d; b= %2d\n",a,b);
   printf("------------------------------------------\n");
   printf("a= %2d : ", a);
   printf("a &= 0x3  -> a= %2d\n", a &= 0x3);
   printf("b= %2d : b &= 0xF -> b= %2d\n",b, b |= 0xF);
}

byron$ gcc zuweisung.c
byron$ a.out
==========================================
a=  7; b= 15
------------------------------------------
a= %2d : a =  7
a |= 0xF  -> a= %2d: a=15
b= %2d : b |= 0x3F -> b= %2d: 63 / 63
==========================================
a=  5; b= 17
------------------------------------------
a=  5 : a &= 0x3  -> a=  1
b= 31 : b &= 0xF -> b= 31
byron$
```

Also Vorsicht!!!

9.5 Kontrollstrukturen

9.5.1 Übersicht

```
$ anweisung = prefix anweisung
$             | [ ausdruck ] ";" | "break;"
$             | "continue;" | "return" [ ausdruck ] ";"
$             | "goto" name ";" | block
$             | "if" "(" ausdruck ")" anweisung
$             | "if" "(" ausdruck ")" anweisung "else" anweisung
$             | "do" anweisung "while" "(" ausdruck ")" ";" .
$ prefix    = name ":"
$             | "while" "(" ausdruck ")"
$             | "for" "(" [ausdruck] ";" [ausdruck] ";" [ausdruck] ")"
$             | "switch" "(" ausdruck ")"
$             | "case" konstante ":"
$             | "default" ":" .
$ block     = "{" {lokal} {anweisung} "}" .
```

9.5.2 leere Anweisung

";" (Semikolon)

9.5.3 Bewertung eines Ausdrucks

ausdruck;
i.a. Zuweisungen oder Funktionsaufrufe

9.5.4 break

innerste Schleife vorzeitig verlassen, Abbrechen der **switch** -Anweisungen (in Modula-2 die **case** -Anweisung) - Verlassen eines "Falles"

9.5.5 continue

Vorzeitiger Neubeginn der umgebenden Schleife

9.5.6 return

i.P. wie in Modula-2 / Oberon: beendet Funktion und liefert Wert des danach stehenden Ausdrucks, falls vorhanden

9.5.7 if

Wahrheitswerte und Werte von Ausdrücken:

BOOL	int-Wert
TRUE	1 bzw. ungleich 0
FALSE	0

Beispiel:

```
if (j=5) tu_etwas();
```

Dies ist syntaktisch korrekt; *tu_etwas()* wird immer aufgerufen; die Bedingung *(j=5)* hat als Nebeneffekt die Wertzuweisung von 5 an die Variable j, der Wert der Bedingung ist der zugewiesene Wert, also 5, d.h. ungleich 0, also TRUE

Gemeint war wohl:

```
if (j == 5) tu_etwas();
```

9.5.8 if - else

das **else** wird jeweils dem "nächsten" **if** zugeordnet

9.5.9 switch (Mehrfachverzweigung)

```
/* switch.c */
#include <stdio.h>

int main() {
   int i;
   printf("Geben Sie eine Ganze Zahl: \n");
   while( scanf("%d", &i) > 0 ) {
      switch (i) {
      case 0:
              printf("0 eingegeben\n");
              break;
      case 1:
              printf("1 eingegeben\n");
              break;
      default:
              printf("weder 0 noch 1\n");
      }
      printf("Noch eine Zahl? \n");
   }
   exit(0);
}
```

Zur Semantik und Verwendung

- Der Ausdruck nach *switch* muß von integralem Typ sein (*char, short, int, long, enum* **nicht jedoch** *float, double, long double)*
  nach K&R war nur *int* erlaubt
- Der Ausdruck nach *case* muß ein konstanter integraler Ausdruck sein
- Der *switch-* Ausdruck wird ausgewertet, das *case-* Label, das zum Ergebnis paßt, wird angesprungen, der Kontrollfluß fährt beim zugehörigen Statement fort; d.h. insbesondere, daß auch die folgenden "Fälle" abgearbeitet werden, falls dies nicht durch ein explizites **break** verhindert wird.

- Die *case*- Labels müssen verschieden sein
- Trifft keiner der Fälle zu, geht es bei *default* weiter (muß zwar nicht, sollte aber grundsätzlich der letzte Fall sein).
- Der *default*- Fall sollte immer vorhanden sein.
- Auch der *default*- Fall sollte mit *break* verlassen werden (spätere Änderungen!)

```
/*   punct.c */
# include <stdio.h>

int is_punc(char arg) {
/*   This function returns 1 if the argument is
 *   a punctuation character, otherwise zero
 */

   switch (arg) {
      case '.':
      case ',':
      case ':':
      case ';':
      case '!':  return 1;
      default:   return 0; break;
   }
}

void main() {
   char z;

   printf("Give one character:");
   if ( scanf("%c",&z) > 0) {
      if ( is_punc(z) )
         printf("punctuation character!\n");
      else
         printf("no punct. char.!\n");
      exit(0);
   } else {
      printf("\nNichts eingegeben!\n");
      exit(1);
   }
}
```

9.5.10 while

Beispiel: Zeichenweises Lesen von *stdio* und Zählen der Blanks

Die Bibliotheks-Funktion *getchar()* aus *stdio.h* liest ein Zeichen von *stdio* und liefert es als *int* -Wert. Ende des Eingabestroms über Makro *EOF* (in *stdio.h* als -1 definiert.

```
/*
 * getchar.c
 */

#include <stdio.h>

int main() {
   int ch, num_of_spaces = 0;

   while ( (ch = getchar()) != EOF)
      if (ch == ' ')
          num_of_spaces++;
   printf("Number of Blanks: %d\n", num_of_spaces);
   exit(0);
}
```

9.5.11 do-while

```
thales$ cat do_while.c
/*
 * do_while.c
 */

#include <stdio.h>

void main()
{
        int ch, num_of_spaces = 0;
        printf("Give one sentence:\n");

        do
                if ( (ch = getchar()) == ' ')
                        num_of_spaces++;
        while ( ch != '\n');

        printf("Number of spaces: %d\n\n", num_of_spaces);
```

```
        exit(0);
}
thales$ gcc do_while.c
thales$ a.out
Give one sentence:
>>>Dies ist ein kleiner Satz mit 7 Blanks.
Number of spaces: 7

thales$
```

9.5.12 for

```
for(init-ausdruck;term-ausdruck;inkr-ausdruck)
        anweisung;
```

ist äquivalent zu

```
init-ausdruck; /* Initialisierung */
while (term-ausdruck) { /* Test */
   anweisung;
   inkr-ausdruck;  /* Inkrementierung */
}
```

Jeder der drei Ausdrücke kann leer sein; ein "leerer" Test setzt eine stets erfüllte Bedingung (Endlos-Schleife) - mit **break** oder **return** zu verlassen.

Beispiel: Ziffernfolge einlesen und Zahlenwert ausgeben

```
/*
 *   for1.c
 */

#include <stdio.h>
#include <ctype.h> /* wegen Funktion isdigit */

int make_int() {
   int num = 0, d;

   for (d=getchar(); isdigit(d);d=getchar()) {
      num *=10; num += d - '0';
   }
   return num;
}
```

```
int main() {
   printf("Give a sequence of digits:\n");
   printf("The number is: %d\n", make_int());
   exit(0);
}
```

Oder so:

```
/*
 * for2.c
 */

#include <stdio.h>
#include <ctype.h> /* wegen isdigit */

int make_int() {
   int num = 0, d;
   while ( isdigit( d = getchar() ) ) {
      num *=10; num += d - '0';
   }
   return num;
}

int main() {
   printf("Give a sequence of digits:\n");
   printf("The number is: %d\n", make_int());
   exit(0);
}
```

Beispiel: Blanks ignorieren:

```
/*
 * ignore.c
 */

#include <stdio.h>
#include <ctype.h>

void skip_spaces() {
   int c;
```

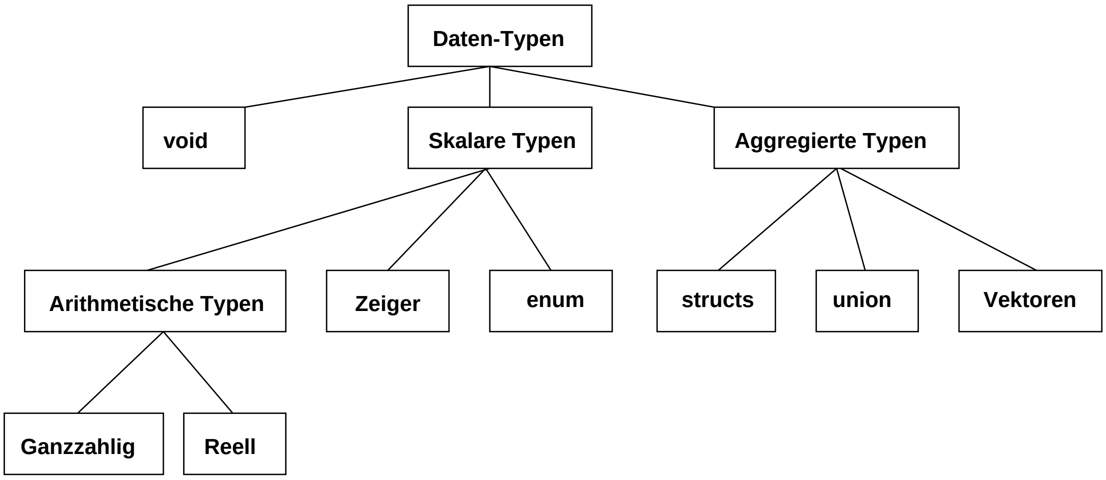
```
   for ( c=getchar(); isspace(c); c=getchar() )
      ;       /*leere Anweisung*/

   ungetc(c,stdin);
   /* Stellt das Nicht-Blankzeichen
    * in den Eingabepuffer zurueck
    */
}

int main() {
   int c;
   while ( (c = getchar()) != EOF)
      if ( c == ' ') {
         putchar(c);
         skip_spaces();
      } else
         putchar(c);
   exit(0);
}
```

10. Datentypen und Konstante

10.1 Skalare Datentypen

- Speicherbedarf
- Interpretation der Bitbelegung
- erlaubte Operationen

Übersicht:



Skalare Typen:

- **Grundtypen** ("Hauptworte"): *char, int, float, double, enum*
- **modifizierte Grundtypen** ("Eigenschaftsworte"): *long, short, signed, unsigned*

Konstante: Voranstellen von *const* (?!)

```
byron$ cat const.c
/* const.c */
# include <stdio.h>

void main() {
   const int i = 1;
   i++;
   printf("i = %d\n", i);
}
byron$ gcc -Wall const.c
const.c: In function ‘main’:
```

```
const.c:6: warning: increment of read-only variable `i'
byron$ a.out
i = 2
byron$
```

Type	Größe (Bytes)	Wertebereich
int	4 (?)	-2^{31} .. 2^{31}-1
short int	2	-2^{15} .. 2^{15}-1
long int	4	-2^{31} .. 2^{31}-1
unsigned short int	2	0 .. 2^{16}-1
unsigned long int	4	0 .. 2^{32}-1
signed char	1	-2^{7} .. 2^{7}-1
unsigned char	1	0 .. 2^{8}-1

10.2 Characters und Integers

Characters: 1 Byte Integers

Beispiel:

```
byron$ cat char_int.c
/* char_int.c */

#include <stdio.h>

void main() {
   char a,b; int i;
   a = 'A'; printf("a = %d\n", a);
   b = 65; printf("b = %c\n", b);
   i = 'A' + 1; printf("i = %c\n", i);
}
byron$ gcc -Wall char_int.c
byron$ a.out
a = 65
b = A
i = B
byron$
```

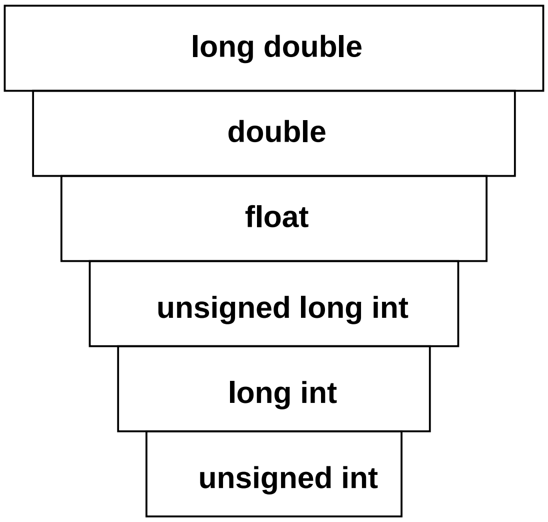
10.3 Mischen von Typen

Arithmetische Typen können in Ausdrücken mit wenigen Ausnahmen gemischt werden.

Explizite Konvertierung *(cast)*

Implizite (automatische) Konvertierung

Hierarchie Skalarer Typen



- Zuweisungs-Konvertierung
  Der Wert auf der rechten Seite wird konvertiert in den Typ auf der linken.
- Ganzzahlige Erweiterung
  char und *short int* in einem Ausdruck werden in *int* konvertiert.
  unsigned char und *unsigned int* werden in *int* konvertiert (falls nicht zu groß, sonst *unsigned int)*
- In arithmetischen Ausdrücken erfolgt die Konvertierung so, daß die Konvertierungsregeln des Operators beachtet werden.

  Enthält ein Operandenpaar eine *long double,* so wird der andere ebenfalls in *long double* konvertiert - ansonsten

  ist einer von beiden *double,* so wird der andere ebenfalls in *double* konvertiert - ansonsten

  ... entsprechend obiger Hierarchie.

Beispiel:

```
byron$ cat char_int2.c
/* char_int2.c */

#include <stdio.h>

void main() {
   char c = 5;
   short j = 6;
   int k;

   k = c + j + 8; /* c und j werden zu
                   * int konvertiert
                   */
   printf("k = %d\n", k);

   {
      unsigned short l = 71124;
      printf("l = %d\n", l);
   }

   c = 882;
   printf("c = %c\n", c);
   c = 114;
   printf("c = %c\n", c);
}
byron$ gcc -Wall char_int2.c
char_int2.c: In function ‘main’:
char_int2.c:16: warning: large integer
                implicitly truncated to
                unsigned type
char_int2.c:20: warning: overflow in
                implicit constant conversion
byron$ a.out
k = 19
l = 5588
c = r
c = r
byron$
```

Anm.:
- Die Binärdarstellung von 882 ist: 00000011 01110010
  Zugewiesen wird das "hintere" Byte, also 114!
- Bei der Zuweisung von 71124 an j wird der Rest von 71124/(65535+1) genommen, also 5588!

10.4 char

- Einzelne Zeichen (Konstanten) werden in einfache Hochkommas eingeschlossen
- Ersatzdarstellungen:

\b	BS	backspace
\f	FF	formfeed
\n	LF	newline, Zeilentrenner
\r	CR	carriage return, “Wagenrücklauf”
\t	HT	Horizontaler Tab
\\	\	’Fluchtsymbol’
\’	’	einfaches Hochkomma
\0	NUL	Null-Byte
\ddd		Bit-Muster (oktal)

10.5 Gleitkommawerte (float, double)

Beispiel (zur Sensibilisierung bzgl. Genauigkeit von Gleitkomma-Rechnungen):

```
byron$ cat genau.c
/* genau.c */
# include <stdio.h>

void main() {
   int x = 10864, y = 18817, z;
   float x1 = x, y1 = y, z1;
   double x2 = x1, y2 = y1, z2, z2a;
   double x3 = x2, y3 = y2, z3, u3, v3,w3;
```

```
   /* Formel:
        9*x*x*x*x-y*y*y*y+2*y*y
    */

   /* als int: */
   z = 9*x*x*x*x-y*y*y*y+2*y*y ;
   printf("ganzzahlig: %d\n",z);

   /* als float: */
   z1= 9*x1*x1*x1*x1-y1*y*y*y+2*y*y ;
   printf("als float: %e\n",z1);

   /* als double, mit anderer Auswertung */
   z2 = 9*x2*x2*x2*x2 + 2*y2*y2;
   z2a = z2 - y2*y2*y2*y2;
   printf("als double, Formel umgestellt: %.12f\n",z2a);

   /* double, schrittweise */
   u3 = 9*x3*x3*x3*x3;
   v3 = 2*y3*y3;
   w3 = u3+v3;
   z3 = y3*y3*y3*y3;
   printf("u3 = %g\n", u3);
   printf("v3 = %g\n", v3);
   printf("w3 = %g\n", w3);
   printf("z3 = %g\n", z3);
   z3 = w3 -z3;
   printf("und jetzt: %g\n", z3);
}

byron$ gcc genau.c
byron$ a.out
ganzzahlig: 1
als float: 1.000000e+00
als double, Formel umgestellt: -1.000000000000
u3 = 1.25372e+17
v3 = 7.08159e+08
w3 = 1.25372e+17
z3 = 1.25372e+17
und jetzt: 0
byron$
```

Beispiel:

```
hypatia$ cat genau1.c
/* genau1.c */

#include <stdio.h>

void main() {
        double x,y;

        x = 2.0;
        y = (2.0/0.77777777777777);
        y *=     0.77777777777777;

        if (x==y)
                printf("gleich\n");
        else
                printf("ungleich\n");
        if (abs(x-y) < 1.0E-9)
                printf("gleich\n");
        else
                printf("ungleich\n");

}
hypatia$ gcc -Wall genau1.c
hypatia$ a.out
ungleich
gleich
hypatia$
```

10.6 Aufzählungen - enum

```
$ enumtype = "enum" [ name ] { enumerator {"," enumerator}}
$            | "enum" name .
$ enumerator = name [ "=" konstante ] .
```

- Steht zwischen *enum* und der Aufzählung ein *name,* so kann anschließend damit die Aufzählung bezeichnet werden.
- Der Wert der Konstanten beginnt von links nach rechts folgend mit 0
- Wird beim *enumerator* mit "=" eine Konstante angegeben, so wird obige Angabe (0, 1, ...) überschrieben
- Verwendet man einen Aufzählungswert als Index in einem Vektor, so muß dieser explizit (*cast*) in *int* gewandelt werden.

```
/* enum.c */
#include <stdio.h>

void main() {
   /* Variable: */
   enum { rot, gelb, gruen } ampel;
   /* Typname: */
   enum farbe { blau, lila, pink };

   enum farbe wand;

   ampel = rot;
   if (ampel == rot )
      printf("Die Ampel ist rot!\n");

   wand = pink;
   if (wand == pink)
      printf("Die Wand ist pink-farben!\n");
}
```

10.7 Vektoren (Array's)

```
$ vector = type_spec vec_name "["vec_size"]"
$              { "["vec_size"]" } [ initializer ] .
```

- Der Name des Vektors ist ein **konstanter Zeiger** auf das erste Element und wird so auch an Funktionen übergeben!!!

- Der Typ ist also ein Zeiger auf den Elementtyp, unabhängig von der Dimensionierung
- Die Indizierung beginnt bei **0** und geht bis **vec_size - 1**
- Problem: Indizierungsfehler werden u.U. erst zur Laufzeit erkannt *(segmentation violation)!!!*

Beispiel: Summe über Vektorelemente

```
byron$ cat vektor1.c
/* vektor1.c */
#include <stdio.h>

const int Length = 10;

void main() {
        int vektor[Length];
        int i, sum;

        for (i=0; i < Length; i++)
                vektor[i] = i;

        sum = 0;
        for (i=Length - 1; i >= 0; i--)
                sum += vektor[i];
        printf("Summe: %d\n", sum);

       /* Indizierungsfehler: */
        sum = 0;
        for (i=0; i <= Length+2; i++)
                sum += vektor[i];
        printf("Summe (falsch): %d\n", sum);
}
byron$ gcc -Wall vektor1.c
byron$ a.out
Summe: 45
Summe (falsch): 1073766905
byron$
```

Noch ein Beispiel:

```
byron$ cat vektor2.c
/* vektor2.c */
#include <stdio.h>


void main() {
        int i, sum, length;
        int vektor[] = { 1, 2, 3, 4, 5 };
        length = sizeof(vektor) / sizeof(vektor[0]);
        printf("Length = %d\n", length);

        sum = 0;
        for (i=length-1; i >= 0; i--)
                sum += vektor[i];
        printf("Summe: %d\n", sum);

        /* etwas Bloedsinn: */
        vektor[5] = 6;

        sum = 0;
        for (i=length-1; i >= 0; i--)
                sum += vektor[i];
        printf("Summe: %d\n", sum);
}
byron$ gcc -Wall vektor2.c
byron$ a.out
Length = 5
Summe: 15
Summe: 21
byron$
```

Also aufgepaßt!!!

10.8 Zeiger

10.8.1 Grundlegendes

Vereinbarung:

```
$ zeigervariable = el_type "*" var_name .
```

Durchgriff auf referenziertes Objekt: durch Voranstellen des ***** Operators vor die Zeigervariable!

- Zeigerwerte können als *integer* interpretiert werden (Achtung: systemabhängig)
- Null-Zeiger (NIL): 0 oder als Makro: NULL (in stdlib.h)
- Der Typ ist an den Typ gebunden, auf den gezeigt wird; kann aber explizit auf andere Typen gewandelt werden *(cast)*
- Der Wert einer Zeigervarablen ist die Adresse einer Speicherfläche, deren Inhalt von dem Typ sein muß (sollte), auf den der Zeiger zeigt.
- Entstehung des Wertes einer Speichervariablen:
  Zuweisung der Adresse einer bestehenden Speicherfläche vom entsprechenden Typ
  Dynamische Speicher-Allokation (siehe Funktionen aus *stdlib.h: calloc, malloc, realloc)*

Beispiel:

```
byron$ cat zeiger.c
/* zeiger.c */
# include <stdio.h>

/* wegen calloc */
# include <stdlib.h>

void main() {
   int i = 10;
   int v[] = { 1,2,3,4,5 };
   int * p;

   p = &i;
   printf("*p = %d\n", *p);
```

```
   p = v;
   printf("*p = %d\n", *p);

   p = &v[0];
   printf("*p = %d\n", *p);

   p = (int *) calloc(1,sizeof(int));
   if (p) {
        *p = 100;
        printf("*p = %d\n", *p);
   }
}
byron$ gcc -Wall zeiger.c
byron$ a.out
*p = 10
*p = 1
*p = 1
*p = 100
byron$
```

10.8.2 Zeiger-Arithmetik

- Addition

  Sei *ptr* eine Zeigervariable: der Wert von *ptr+3* zeigt auf ein Objekt, das 3 "Plätze" hinter **ptr* liegt; d.h., es wird um *3*sizeof(*ptr)* weitergegangen.
- Subtraktion

  nur Zeiger vom selben Typ
  liefert *int* Wert (Anzahl der Objekte dazwischen)

Beispiele:

```
long *p1, *p2;
int j;
char *p_ch;

p2 = p1+4;       /*zulaessig; sinnvoll?*/
j = p2-p1;       /*zulaessig; j == 4*/
```

```
j = p1-p2;       /*zulaessig; j == -4*/
p1 = p2 - 2;     /*zulaessig, gleicher Typ; sinnvoll?*/
p_ch = p1 - 1;   /*NICHT zulaessig, versch. Typ*/
j = p1 - p_ch;   /*NICHT zulaessig, versch. Typ*/
```

10.8.3 Vektoren als Parameter

Der Name eines Vektors wird als Zeiger auf das erste Element interpretiert. Übergibt man also den Namen eines Vektors als Parameter, so wird der Zeigerwert via Wertparameter übergeben; greift man über die lokale Kopie des Zeigertwerts auf den Vektor durch, ist man beim **Orginal-Vektor.**

```
/* vekt_zeiger.c */

#include <stdio.h>

void init(int ar[], int length) {
   int i;
   for(i=0; i < length; i++)
   ar[i] = i;
}

int summe1(int ar[], int length) {
   int i, sum = 0;
   for (i=0; i < length; i++)
      sum += ar[i];
   return sum;
}

int summe2(int * ar, int length) {
   int i, sum = 0;
   for (i=0; i < length; i++,ar++)
      sum += *ar;
   return sum;
}

void main() {
   const int vekt_len = 10;
   int vektor[vekt_len];
```

```
   init(vektor, vekt_len);

   printf("Summe: %d\n", summe1(vektor, vekt_len));
   printf("Summe: %d\n", summe2(vektor, vekt_len));
}
```

10.8.4 sizeof()

liefert die Zahl der Bytes, die eine Datentyp T (*sizeof(T)*) oder eine Ausdruck x *sizeof x* oder *sizeof(x)* belegt!

10.9 Zeichenketten, Strings

- String-Konstante: eine in Doppelapostroph eingeschlossene Zeichenfolge mit abschließendem Null-Byte (**\0**), wird i.a. vom Compiler angefügt
- String-Konstante werden i.a. im Read-Only-Speicherbereich abgelegt!!!

<u>Beispiel:</u>

```
byron$ cat strings.c
/* strings.c */

#include <stdio.h>

char * str = "ein Text";

void main() {
        char array[10];
        char string[10] = "Hallo!";
        char *ptr1 = "10 char's";
        char *ptr2;

/*      array = "not OK";*/
        /* nicht zulaessig, array ist
         * konstante Adresse
         */

        array[5]='A';           /*zulaessig*/
```

```
/*      ptr1[5]='B';    */
        /* i.a. nicht zulaessig, da die
         * Speicherflaeche, auf die ptr1
         * hier zeigt, READ ONLY ist!
         */

        ptr1="ok";
        printf("ptr1: %s\n", ptr1);

        printf("ptr1: %s\n", ptr1);

        ptr2 = str;
        printf("ptr2: %s\n", ptr2);

        string[0]='X';
        printf("string=%s\n", string);
}
byron$ gcc -Wall strings.c
byron$ a.out
ptr1: ok
ptr1: ok
ptr2: ein Text
string=Xallo!
byron$
```

10.10 Einige Beispiele

```
/* string_func.c */

#include <stdio.h>

void strcpy1(char s[], char t[]) {
/*t nach s kopieren */
        int i;
        i = 0;
        while ( (s[i] = t[i]) != '\0')
        {
                printf(" i: %d\n", i);
                i++;
        }
```

```
}

void strcpy2(char *s, char *t) {
        while ( (*s = *t) != '\0' )
        {
                s++;
                t++;
        }
}

void strcpy3( char *s, char *t ) {
        while( (*s++ = *t++) != '\0')
                ;
}

/* Noch kuerzer moeglich, da bei Zuweisung
 * des Null-Bytes der Wert des Ausdrucks 0 ist
 * und damit die Bedingung "FALSE"
*/
strcpy4( char *s, char *t ) {
        while (*s++ = *t++)
                ;
}

int strcmp1(char s[], char t[]) {
/*<0, wenn s<t, 0, wenn s=t, >0, wenn s>t */
        int i = 0;
        while ( s[i] == t[i] )
                if ( s[i++] == '\0')
                        return (0);
        return (s[i]-t[i]);
}

/* Zeigerfassung */
int strcmp2(char *s, char *t) {
        for ( ; *s == *t; s++, t++)
                if (*s == '\0')
                        return (0);
        return (*s - *t);
}
```

```
char str[10] = "Hallo";
char * p_str = "Hallo";

void main() {
        char ziel[20];

        printf("String: %s\n", str);

        printf("strcpy1:  \n");
        strcpy1(ziel, str);
        printf(" %s\n", ziel);

        printf("strcpy2:  ");
        strcpy2(ziel, str);
        printf(" %s\n", ziel);

        printf("strcpy3:  ");
        strcpy3(ziel, str);
        printf(" %s\n", ziel);

        printf("strcpy4:  ");
        strcpy4(ziel, str);
        printf(" %s\n", ziel);
        printf("strcpy1:  \n");
        strcpy1(ziel, str);
        printf(" %s\n", ziel);

        if (strcmp(p_str, str) == 0)
                printf("\n %s und %s sind gleich\n", \
                        p_str, str);
}
```

10.11 Dynamische Speicherverwaltung

- *char * calloc(unsigned nelem, unsigned elsize)*

  liefert einen Zeiger auf mit Null initialisierte Speicherfläche für **nelem** Objekte der angegebenen Größe. Ist nicht genügend Speicher verfügbar, wird der Null-Zeiger zurückgegeben. Mit einer entsprechenden *cast-* Operation kann der Zeiger auf den entsprechenden Typ gewandelt werden:

```
char * calloc();
int * i_p;

i_p = (int *) calloc(n, sizeof(int));
```

Mehr dazu über *man calloc*

- *char * malloc(unsigned size)*

  liefert einen Zeiger auf eine Speicherfläche mit wenigstens *size* Bytes, nicht initialisiert.

- *char * realloc(char *ptr, unsigned size)*

  ändert die Größe der bislang über *ptr* definierten Speicherfläche auf die angegebene Byte-Zahl und liefert einen Zeiger auf eine (möglicherweise "bewegte") Speicherfläche zurück; der Inhalt bleibt soweit unverändert.

- *void free(char *ptr)*

  gibt eine zuvor durch *calloc(), malloc(), realloc()* besorgte Speicherfläche frei. In vielen Büchereien gibt es daneben die Funktion *int cfree(char *ptr);*

  Meist ist *free* vom Ergebnistyp *int* und signalisiert mit 1 Erfolg und mit 0 Mißerfolg.

Beispiel: Namen von *stdin* in Tabelle einlesen und wieder ausgeben

```
/* table.c */

# include <stdio.h>

/* storage allocation: */
# include <stdlib.h>

/* string copy */
# include <string.h>

const int TableLength = 30;
const int NameLength = 20;

int readline(char buffer[], int length) {
/* reads not more than length char's form stdin
 * appends Null-Byte and returns number of char's read
 * if more than length char's on input-line
 * rest of line is skipped
 */

   int i=0, ch;
   while( (i < length-1) && ( (ch = getchar()) != EOF) ){
        if (ch != '\n') {
           buffer[i] = (char) ch;
           i++;
        } else break;
   }
   buffer[i] = '\0';
   /*skip rest of line:*/
   while( (ch != '\n') && (ch != EOF))
     ch=getchar();
   return i;
}

void print_table(char ** table) {
   while (*table) {
      printf("%s\n", *table);
      table++;
   }
}

void main() {
   char * table[TableLength];
   char name[NameLength+1];
   int i;

   /* init table: */
   for(i=0; i< TableLength; i++)
        table[i] = NULL;

   /* read input into table" */
   i = -1;
   while (i < TableLength-1) {
      i++;
```

```
      if (readline(name, NameLength+1) > 0) {
          table[i] = (char *) calloc(NameLength+1, sizeof(char));
          strcpy(table[i], name);
      } else {
          i--;
          break;
      }
   }

  /* mit print_table ausgeben: */
   print_table(table);

   i=0;
   while( (i < TableLength) && table[i] ) {
      printf("%s\n", table[i]);
      free(table[i]);
      table[i] = NULL;
      i++;
   }

}
```

10.12 Mehrdimensionale Felder

Beispiel:

int matrix[2][3];

matrix ist ein 2-elementiger Vektor, dessen Elemente 3-elementige Vektoren sind, dessen Elemente integer-Zahlen sind.

Zugriff auf ein "Matrix"-Element:

matrix[1][0]

Initialisierung:

```
int matrix[2][3] = {    {0,1,2}, {3,4,5}
                     }
```

Element	Adresse	Inhalt
matrix[0][0]	1000	0
matrix[0][1]	1004	1
matrix[0][2]	1008	2
matrix[1][0]	100C	3
matrix[1][1]	1010	4
matrix[1][2]	1014	5

Der Zugriff *matrix[1][2]* wird interpretiert als

**(matrix[1]+2)*,

dies wiederum als

**(*(matrix + 1) + 2).*

10.13 Mehrdimensionale Felder als Parameter

```
f1() {
        int ar[5][6][7];
        ...
        f2(ar);
        ...
}

f2( int erh_arg[][6][7]) {
        ...
}
```

erh_arg wird interpretiert als *int (*erh_arg) [6][7]; erh_arg* ist also ein Zeiger auf einen 6-dimensionalen Vektor, dessen Elemente 7-dimensionale Vektoren sind, deren Elemente wiederum integer's sind.

Das Ganze mit Zeigern:

```
f1() {
        int ar[4][5][6];
        ...
        f2(ar,4,5,6);
        ...
}

f2( int *** erh_arg, int dim1, int dim2, int dim3) {
        ...
}
```

Zugriff auf *ar[x][y][z]:*

** ((int *) erh_arg + x*dim3*dim2 + y*dim2 + z)*

Der **cast** *(int *)* ist notwendig, da *erh_arg* Zeiger auf Zeiger ist und die Skalierung über Zeigergröße falsch wäre.

11. Strukturen (Records): struct's

11.1 Vereinbarungen

```
struct pers_dat {
   char * p_name, * p_nr ;
   short geb_monat, geb_tag, geb_jahr;
};

/* Variable pers: */
struct pers_dat pers;
```

Die Variable *pers* steht für die gesamte Struktur (Speicherfläche). *pers_dat* ist ein Name für die Struktur (keine Typ-Vereinbarung).

Ohne Struktur-Name:

```
struct {
    char * p_name, * p_nr;
    short geb_monat, geb_tag, geb_jahr;
} person;
```

Mit Typvereinbarung:

```
/* Typ-Name fuer Struktur: */
typedef struct {
   char * p_name, * p_nr;
   short geb_monat, geb_tag, geb_jahr;
} Pers_dat;

/*Variable person vom Type pers_dat*/
Pers_dat person;
```

11.2 Initialisieren bei Definition

```
Pers_dat person = {"Otto Huber","SAI",3,5,1950};
```

11.3 Zugriff auf Komponenten

```
if (person.geb_tag == 28) ...
```

11.4 Zeiger auf Strukturen

```
Pers-dat * p_to_person;
...
if( (p_to_pers -> geb_tag > 28) || ((*p_to_pers).geb_monat == 2 )){
        ...
}
```

11.5 Geschachtelte Strukturen

```
typedef struct {
        char * name, * vname;
        struct {
                short tag, monat, jahr;
        } geb_datum;
} Person;

Person p1;
 ...

p1.geb_datum.tag = 21;
```

oder so:

```
typedef struct {
        short tag, monat, jahr;
} Datum;

typedef struct {
        char * name, * vname;
        Datum geb_datum;
```

```
} Person;
```

11.6 Rekursive Strukturen

Beispiel:

```
struct s {
        int a,b;
        float c;
        struct s * p_to_s;
};

struct s1 {
        int a;
        struct s2 * b;
}

struct s2 {
        int v;
        struct s1 * w;
}
```

Eine der wenigen Ausnahmen von der Regel "declare before use" !!!

ABER:

```
typedef struct {
  int a;
  fehler * p; /*Fehler, da fehler noch nicht def.*/
} fehler;
```

11.7 Strukturen als Funktionsargumente

```
/* struct1.c */

#include <stdio.h>

typedef struct {
        short tag, monat, jahr;
} Datum;
```

```
typedef struct {
        char name[21], vname[11];
        Datum geb_datum;
} Person;

typedef struct {
        Datum d;
        char ereignis[31];
} Ereignis;

Ereignis fest = {
                  {1,1,2000}, {"Neujahr"}
};

void ausgabe(Ereignis arg) {
        printf("Am %d.%d.%d ist ein besonderes %s\n", \
                arg.d.tag, arg.d.monat, arg.d.jahr,
                arg.ereignis);
}

void main() {
        ausgabe(fest);
}
```

Mit Zeigern:

```
/* struct2.c */

#include <stdio.h>

typedef struct {
        short tag, monat, jahr;
} Datum;

typedef struct {
        char name[21], vname[11];
        Datum geb_datum;
} Person;

typedef struct {
        Datum d;
        char ereignis[31];
```

```
} Ereignis;

Ereignis fest = {
                  {1,1,2000}, {"Neujahr"}
};

void ausgabe(Ereignis * arg) {
        printf("Am %d.%d.%d ist ein besonderes %s\n", \
                arg->d.tag, arg->d.monat, arg->d.jahr,
                arg->ereignis);
}

void main() {
        ausgabe(&fest);
}
```

11.8 Strukturen als Ergebnis von Funktionen

```
/* struct3.c */
#include <stdio.h>

typedef struct {
        short tag, monat, jahr;
} Datum;

typedef struct {
        Datum d;
        char ereignis[31];
} Ereignis;

Ereignis bestimme() {
   Ereignis er = {
            {1,1,2000},
            {"Neujahr"}
   };
   return er;
}

void ausgabe(Ereignis arg) {
   printf("Am %d.%d.%d ist / war ein spezielles %s\n",
      arg.d.tag, arg.d.monat, arg.d.jahr, arg.ereignis);
}
```

```
void main() {
   Ereignis fest = bestimme();
   ausgabe(fest);
}
```

Achtung bei der Rückgabe von Zeiger auf Strukturen:

```
hypatia$ cat struct4.c

/* struct4.c */
#include <stdio.h>

typedef struct {
        short tag, monat, jahr;
} Datum;

typedef struct {
        Datum d;
        char ereignis[31];
} Ereignis;

Ereignis * bestimme() {
   Ereignis er = {
            {1,1,2000},
            {"Neujahr"}
   };
   return & er;
}

void ausgabe(Ereignis arg) {
   printf("Am %d.%d.%d ist / war ein spezielles %s\n",
      arg.d.tag, arg.d.monat, arg.d.jahr, arg.ereignis);
}
void main() {
   Ereignis * fest = bestimme();
   ausgabe(* fest);
}
hypatia$ gcc -Wall struct4.c
struct4.c: In function ‘bestimme’:
struct4.c:18: warning: function returns address of local variable
hypatia$ a.out
Am 1.1.2000 ist / war ein spezielles Ne
hypatia$
```

11.9 Eine kleines Listen-Programm

```
/* ein kleines Listenpogramm */

#include <stdio.h>
#include <stdlib.h>

#define FALSE 0
#define TRUE 1

typedef int BOOL;

typedef struct node *ListPtr;
typedef struct node { /*Listenelement */
        int count;
        ListPtr next;
} listNode;

ListPtr listalloc(void) {
        return (ListPtr) calloc(1, sizeof(listNode));
}

BOOL buildList(ListPtr * firstEl, int n) {
        int i = 1;
        ListPtr lastEl;

        * firstEl = listalloc();

        if (!(*firstEl)) {
                return FALSE;
        }
        (*firstEl)->count = i;

        lastEl = *firstEl;

        i++;
        while  (i <= n) {
                lastEl->next = listalloc();
                if (!lastEl)
                        return FALSE;
                lastEl = lastEl->next;
```

```
                lastEl->count = i;
                i++;
        }
        return TRUE;
}

void printListAsc(ListPtr p) {
        while (p) {
                printf("%d ", p->count);
                printf(" | ");
                p=p->next;
        }
}
void printListDesc(ListPtr p) {
        if (p != NULL) {
                printListDesc(p->next);
                printf("%d ", p->count);
                printf(" | ");
        }
        else {
                return;
        }
}

int main() {
        ListPtr  p;
        if (! buildList(&p,10))
                exit(1);

        printf("Listenelemente (von vorne):\n");
        printListAsc(p);
        printf("\n");
        printf("Listenelemente (von hinten):\n");
        printListDesc(p);
        printf("\n");

        exit(0);
}
```

12. typedef

Vereinbarung neuer Datentyp-Namen

Beispiel: *typedef int LENGTH;*

Damit ist *LENGTH* synonym zu *int.*

Also kann statt *int i,j;* auch *LENGTH i,j;* verwendet werden.

Die Vereinbarung ist wie bei Variablenvereinbarungen: der neue Typ-Name steht an der Stelle des Variablennamens.

Mit typedef wird **kein** neuer Datentyp (wie in Modula-2 oder Oberon mit TYPE) konstruiert. Es wird nur ein Name für einen anderweitig auch existierenden Datentyp eingeführt.

typedef ist nützlich gegen Portabilitätsprobleme: führt man Datentypen mittels *typedef* ein, so muß ggf. nur die *typedef*- Vereinbarung geändert werden

Mittels *typedef* kann ein Programm auch besser dokumentiert werden (sprechende Typ-Namen)

typedef verhält sich ähnlich wie *define* - nur wird *typedef* vom Übersetzer verarbeitet, während *define* vom Preprozessor verarbeitet wird. So ist

#define USHORT unsigned int

äquivalent mit

typedef unsigned int USHORT;

Aber:

```
#define PT_TO_INT int *

PT_TO_INT p1, p2;
```

wird expandiert zu `int * p1, p2;`

Damit ist nur *p1* ein *int* -Pointer, p2 ist eine *int.* Mit

```
typedef int * PT_TO_INT;

PT_TO_INT p1,p2;
```

werden zwei *int*- Pointer eingeführt.

13. Komplexe Deklaration

Bsp.: Ein Zeiger auf eine Funktion, die einen Zeiger auf einen 5-elementigen Vektor liefert, dessen Elemente Zeiger auf *int's* sind.

*int *(*(*x)())[5];*

entziffert von innen nach außen:

**x*	Zeiger auf
((*x)())*	() bindet stärker als *, also ist (*x)() eine Funktion, die einen Zeiger liefert. Also: x Zeiger auf Funktion, die einen Zeiger liefert
*int *(*(*x)())[5]*	[] bindet stärker als *, also ist x Zeiger auf eine Funktion, die einen Zeiger auf einen Vektor liefert, dessen 5 Elemente Zeiger auf int's sind.

Transparenter wird dies durch stufenweisen Aufbau mit *typedef:*

*typedef int *AP[5];*

AP ist damit Name für einen 5-elementigen Vektor mit Zeigern auf *int*

*typedef AP *FP();*

FP ist damit Funktion, die Zeiger auf *AP* liefert.

*FP * x;*

Beachte:

- Der Vektor-Operator [] und der Funktionsoperator () haben höheren Vorrang als der Zeigeroperator *
- [] und () werden von links nach rechts gruppiert, * von rechts nach links

Beispiele:

*char *x[];*

- *x[]* ist Vektor
- **x[]* ist Vektor von Zeigern
- *char *x[]* ist Vektor von Zeigern auf int

Klammern ändern den Vorrang:

*int (*x[])();*

- *x[]* ist Vektor
- *(*x[])* ist Vektor mit Zeiger-Elementen
- *(*x[])()* ist Vektor mit Zeigern auf Funktionen
- *int (*x[])()* ist Vektor von Zeigern auf Funktionen, die *int* liefern.

ohne Klammern:

*int*x[]();*

- ein Vektor von Funktionen, die Zeiger auf int's liefern
- **unzulässig,** da Vektoren mit Funktionen als Elemente nicht erlaubt!

Beispiel: Ein Zeiger auf einen Vektor, dessen Elemente Zeiger auf Funktionen sind, die Zeiger auf Vektoren liefern, deren Elemente Strukturen mit *tag*- Name S sind:

1.	*(*x)*	x ist Zeiger
2.	*(*x)[]*	x ist Zeiger auf Vektor
3.	*(*(*x)[])*	x ist Zeiger auf Vektor, dessen Elemente Zeiger sind
4.	*(*(*x)[])()*	x ist Zeiger auf Vektor, dessen Elemente Zeiger auf Funktionen sind
5.	*(*(*(*X)[])())*	x ist Zeiger auf Vektor, dessen Elemente Zeiger auf Funktionen sind, die Zeiger liefern
6.	*(*(*(*x)[])())[]*	x ist Zeiger auf Vektor, dessen Elemente Zeiger auf Funktionen sind, die Zeiger auf Vektoren liefern
7.	*struct S (*(*(*x)[])())[]*	x ist Zeiger auf Vektor, dessen Elemente Zeiger auf Funktionen sind, die Zeiger auf Vektoren liefern, deren Elemente Strukturen mit *tag*-Name S sind

Beispiele für unzulässige Deklarationen:

int af[]()	Vektor von Funktionen, die int's liefern
int fa()[]	Eine Funktion, die einen Vektor mit int's liefert
int ff()()	Eine Funktion, die eine Funktion liefert, die int's liefert
*int (*paf)[]()*	Ein Zeiger auf einen Vektor von Funktionen, die int's liefern
*int * ffp()()*	Eine Funktion, die eine Funktion liefert, die einen Zeiger auf int's liefert

14. Argumente aus der Kommandozeile

Die Funktion *main hat zwei Argumente (eigentlich drei): argc* und *argv:*

- *argc*
  Diese int-Größe gibt die Anzahl der Argumente auf der Kommandozeile (incl. Kommandoname)
- *argv*
  Dies ist ein Zeiger auf einen Vektor mit Zeichenketten, die die einzelnen Argumente enthalten, ein Argument pro Zeichenkette.

Nach Konvention ist *argv[0]* der Kommando-/Programmname, *argc* also mindestens 1.

Beispiel: alle Argumente inkl. Programmname ausgeben

```
/* arg1.c */

#include <stdio.h>

void main(int argc,char *argv[]) {
  int i;
  for (i = 0; i<argc; i++)
     printf("%s%c", argv[i], (i < argc-1) ? ' ':'\n');
}
```

Da *argv* ein Zeiger auf einen Vektor von Zeigern ist, gibt es mehrere Möglichkeiten, dieses Programm zu realisieren (jetzt ohne Programmname):

```
/* arg2.c */

#include <stdio.h>

void main(int argc, char *argv[]) {
  while (--argc > 0 )
     printf("%s%c", *++argv, (argc > 1) ? ' ':'\n');
```

```
}
```

argv ist ein Zeiger auf den Anfang des Vektors mit den Kommando-Argumenten; somit sorgt die Inkrementierung ++*argv* dafür, daß dieser Zeiger auf das Element *argv[1]* zeigt und nicht mehr auf *argv[0]*.

Oder:

```
/* arg3.c */

#include <stdio.h>

void main(int argc, char *argv[]) {
   while( --argc > 0)
       printf((argc > 1) ? "%s" : "%s\n", *++argv);
}
```

Ein Pattern-Suchprogramm a la grep:

```
/* mygrep.c */

#include <stdio.h>

const int line_length = 1000;

/* statt fgets: */
int getline(char s[], int lim) {
   int c,i=0;
   while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
          s[i++] = c;
   if ( c == '\n')
          s[i++] = c;
   s[i] = '\0';
   return (i);
}

int index(char s[], char t[]) {
/*
 * liefert Indexposition, ab der t in s vorkommt
 */
   int i,j,k;

   for (i = 0; s[i] != '\0'; i++) {
       for (j=i,k=0; t[k] != '\0' && s[j]==t[k]; j++,k++)
                            ;
```

```
        if (t[k] == '\0')
             return (i);
    }
    return (-1);
}

int main(int argc, char *argv[]) {
  /* Suchmuster als erstes Argument */
   char line[line_length];

   if (argc != 2) {
       fprintf(stderr, "Usage: %s pattern\n", argv[0]);
       exit(1);
   } else
       while (getline(line,line_length) > 0)
           if (index(line,argv[1]) >=0)
              printf("%s",line);
  exit(0);
}
```

Modifikation dahingehend, daß <u>zwei</u> optionale Argumente erlaubt sein sollen: Das eine Argument verlangt "Gib alle Zeilen aus, mit Ausnahme derer, die das Suchmuster enthalten", das zweite "Gib vor jeder Ausgabezeile die Zeilennummer aus".

```
/*  mygrep1.c:
 * find -x -n pattern
 * find -nx pattern
*/

#include <stdio.h>
const unsigned int line_length = 1000;

int getline(); int index();

int main(int argc, char *argv[]) {
    char line[line_length], *s;
    long lineno=0;
    int except = 0, number = 0;
    char * p_name = argv[0];

    while (--argc > 0 && (*++argv)[0] =='-')
      for (s = argv[0] + 1; *s != '\0'; s++)
         switch (*s) {
             case 'x':
                except = 1;
                break;
             case 'n':
```

```
                number = 1;
                break;
             default:
                fprintf(stderr, "%s: illegal option >%c<\n",
                        p_name, *s);
                argc = 0;
                break;
          }
    if (argc != 1) {
         fprintf(stderr, "Usage: %s -x -n pattern \n",
                 p_name);
         exit(1);
    } else
         while (getline(line, line_length) > 0) {
            lineno++;
            if ((index(line,*argv) >=0) != except) {
               if (number)
                    printf("%ld: ", lineno);
               printf("%s", line);
            }
         }
   exit(0);
}

int getline(char s[], int lim) {
    int c,i;
    i=0;
    while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
            s[i++] = c;
    if ( c == '\n')
            s[i++] = c;
    s[i] = '\0';
    return (i);
}

int index(char s[], char t[]) {
    int i,j,k;

    for (i = 0; s[i] != '\0'; i++)
    {
       for (j=i,k=0; t[k] != '\0' && s[j]==t[k]; j++,k++)
               ;
       if (t[k] == '\0')
              return (i);
    }
    return (-1);
}
```

Anm.: Statt *(*++argv)[0]* wäre auch ***++argv* möglich!

Man beachte: *(*++argv)[0]* und **++argv[0]* sind verschieden: letzteres ist als **++(argv[0])* zu lesen.

15. Mehr zum Preprozessor

15.1 Übersicht

- Makro-Verarbeitung
- Einfügen von Quelldateien ("Modularisierung")
- bedingte Übersetzung

Preprozessor-Anweisungen sind zeilenorientiert, beginnen mit **#** (das erste Nicht-Leerzeichen in der Zeile), können an beliebigen Stellen im C-Programm stehen (meist jedoch am Anfang) und enden mit *newline.* Falls mehr als eine Zeile benötigt wird, steht vor dem *newline* ein Backslash:

```
thales$ cat bsp1.c

# define LONG_MACRO "Dies ist ein langes Makro, \
das deutlich ueber eine \
Zeile hinausgeht."

: LONG_MACRO :

thales$ gcc -E bsp1.c
# 1 "bsp1.c"




: "Dies ist ein langes Makro, das deutlich ueber eine Zeile hinausgeht."  :

thales$ gcc -E -P bsp1.c


: "Dies ist ein langes Makro, das deutlich ueber eine Zeile hinausgeht."  :

thales$
```

Makro-Substitution

Makro: Ein Name, dem ein Text-String zugeordnet ist (Makro-Rumpf). Nach Konvention werden Makro-Namen, die Konstanten repräsentieren, groß geschrieben (Unterscheidung von Makro- und Variablen-Namen).

Textersatz: jedes Vorkommen eines Makro-Namens außerhalb seiner Definition (Makro-Aufruf) wird vom Preprozessor durch den Makro-Rumpf (Ersatztext) ersetzt, in die Eingabe für den Preprozessor zurückgestellt und ggf. erneut bearbeitet.

15.2 define

Syntax:

 # define *macro_name* [**(** *macro_argument* { **,** *macro_argument* } **)**] *macro_body*

Fehlerquellen:

```
thales$ cat bsp2.c
# define SIZE 10;
# define OKAY (var == 1);
# define MAX = 100
# define neg_a_plus_f(a) -(a) + f
# define neg_b_plus_f (b)  -(b) + f
# define min(a,b) ((a) < (b) ? (a) : (b))

a1) x = SIZE;

a2) int array[SIZE];

b) while OKAY
        tue();

c) for (j=MAX; j>0; j--)

d) j = neg_a_plus_f(x);

e) j = neg_b_plus_f(x);

f) a = min(b++,c);
```

```
thales$ gcc -E -P bsp2.c









a1) x = 10; ;

a2) int array[10; ];

b) while (var == 1);
        tue();

c) for (j== 100 ; j>0; j--)

d) j = -( x ) + f ;

e) j = (b)  -(b) + f (x);

f) a = (( b++ ) < ( c ) ? ( b++ ) : ( c )) ;
thales$
```

Anm.:

- a1) kein Problem
- a2) wird vom Compiler entdeckt
- b) hat die Variable *var* tatsächlich den Wert 1, so Endlosschleife
- c) wohl nicht gewollt!
- d) okay
- e) was, wenn *b* eine vereinbarte Variable ist?
- e) Seiteneffekt: b wird u.U. zweimal inkrementiert!

Beispiel:

```
thales$ cat bsp3.c
# define square(a) a*a
# define quadrat(a) (a)*(a)

j = 2 * square(3+4)
i = 2 * quadrat(3+4)

thales$ gcc -E -P bsp3.c




j = 2 *  3+4 * 3+4
i = 2 * ( 3+4 )*( 3+4 )
thales$
```

- Makro-Argumente haben keinen Typ, also keine Typprüfung
- Makro’s sind “schneller” als Funktionen (wirklich ein Argument für Makro’s?)

15.3 Entfernen von Makro-Definitionen

 # undef macro_name

Beispiel:

```
thales$ cat bsp4.c
# define NAME Schweiggert
Herr NAME ist Leiter der SAI.
# undef NAME
# define NAME Borchert
Herr NAME ist Mitarbeiter der SAI
thales$ gcc -E -P bsp4.c

Herr Schweiggert  ist Leiter der SAI.



Herr Borchert ist Mitarbeiter der SAI
thales$
```

15.4 built-in-Makro’s

```
thales$ cat bsp5.c
# include <stdio.h>

# define CHECK(a,b) \
  if ((a) != (b))  \
        fail(a,b, __FILE__ , __LINE__ )

void fail(int a, int b, char *p, int line)
{
        printf("Check failed in file %s at line %d:\n\
        received %d, exspected %d\n", p, line, a, b);
}

main()
{
        int x = 1;
        CHECK(x,0);
}
thales$ gcc bsp5.c
thales$ a.out
Check failed in file bsp5.c at line 16:
        received 1, exspected 0
thales$
```

15.5 Bedingte Übersetzung

Syntax:

 # if *cond_expr*
C-source-code
{ **# elif** *cond_expr*
C-source-code
}
[**# else**
C-Source-code
]
 # endif

Anm.: das **elif** -Konstrukt ist ANSI (also nicht in allen K&R-Compilern verfügbar)!

Beispiel:

```
thales$ cat bsp6.c
# define x 2
# if x == 1
# undef x
# define x 0
# elif x == 2
# undef x
# define x 3
# else
# define y 4
# endif

a) x
b) y
thales$ gcc -E -P bsp6.c







a) 3
b) y
thales$
```

Anm.: der *cond_expr* **muß** ein konstanter Ausdruck sein, d.h. in obigem Beispiel muß `x` ein Makro sein. Außerdem werden dort Makros erst expandiert und dann wird der Ausdruck bewertet. Nicht definierte Namen werden meist durch 0 ersetzt (oder Fehler).

Beispiel:

```
thales$ cat bsp7.c
# include <stdio.h>

main()
{
        int i = 9;
```

```
        int j = 10;

# if DEBUG
        printf(" Wert von i / j: %d / %d\n", i,j);
# endif

        printf("Ergebnis: %d\n", i*j);
}
thales$ gcc bsp7.c
thales$ a.out
Ergebnis: 90
thales$ gcc -DDEBUG=1 bsp7.c
thales$ a.out
 Wert von i / j: 9 / 10
Ergebnis: 90
thales$
```

Anm.: die Option
-Dname=wert
ist äquivalent zur Makro-Definition
#define name wert

Bei der Übergabe von Doppelanführungszeichen auf Kommandoebene muß wegen derer Interpretation durch die Shell eine Klammerung in einfache Hochkommas erfolgen:

$cc -DZAHL=float -D'INFMT="%f"' -D'OUTFMT="%g"' ...

15.6 Test auf Makro-Existenz:

ifdef

Beispiel:

```
thales$ cat bsp8.c
# include <stdio.h>

# ifndef ZAHL
#       define ZAHL int
#endif

#ifndef OUTFMT
```

```
#       define OUTFMT "%d"
#endif

main()
{
        ZAHL i = 9;
        ZAHL j = 10;

# ifdef DEBUG
        printf(" Wert von i / j:");
        printf(OUTFMT,i); printf(OUTFMT, j);
        printf"\n");
# endif

        printf("Ergebnis:"); printf(OUTFMT, i*j);
        printf("\n");
}
thales$ gcc -DZAHL=float -DOUTFMT='"%f"' bsp8.c
thales$ a.out
Ergebnis:90.000000
thales$ gcc bsp8.c
thales$ a.out
Ergebnis:90
thales$
```

15.7 include

 # include <filename>

 # include filename

Im ersten Fall sucht der Preprozessor an bestimmten Stellen (i.a. */usr/include* nach der Datei *filename* und setzt deren Inhalt an die Stelle von *include.*

Im zweiten Fall wird im eingestellten Pfad gesucht; wenn dort nicht enthalten, so Suche wie bei <...>.

Verwendung: Modularisierung (s.u.)

16. Speicherklassen

C-Programme können aus verschiedenen Quelldateien bestehen, die nacheinander übersetzt werden. Ebenso können Funktionen aus Büchereien geladen werden.

- Korrekte Vereinbarung von Variablen (Geltungsbereich, Lebensdauer)
- Anordnung der Vereinbarung zum korrekten Binden beim Laden

16.1 Geltungsbereich

Der Geltungsbereich eines Namens ist der Teil des Programms, für den er vereinbart wurde. Variable, die am Anfang einer Funktion vereinbart wurden (Speicherklasse **auto**), haben den Geltungsbereich der Funktion.

Der Geltungsbereich von Variablen, die außerhalb einer Funktion, also **extern** vereinbart wurden, ist von der Stelle der Vereinbarung bis zum Ende dieser Datei. Soll eine externe Variable verwendet werden, bevor sie in einer Quelldatei definiert wird oder wird eine Variable verwendet, die in einer anderen Quelldatei definiert ist, so ist eine **extern** -Deklaration notwendig (und aus Lesbarkeitsgründen sinnvoll)!

16.2 Deklaration vs. Definition

- Deklaration:
  legt Eigenschaften einer Variable (Typ, Speicherbedarf, u.a.) fest (Bekanntmachen).
- Definition:
  sorgt zusätzlich dafür, daß Speicherplatz bereitgestellt wird.

Beispiel:

```
# include <stdio.h>

int vekt[10];
int i=0;

main() {
        ...
}
```

Hier werden zwei Variablen *vekt, i* definiert, es wird Speicherplatz bereitgestellt und sie sind deklariert.

```
# include <stdio.h>

extern int vekt[];
extern int i;

main() {
        ...
}
```

Hier werden zwei Variablen *vekt, i* für den Rest der Datei deklariert, die Variablen werden nicht "erzeugt", es wird kein Speicherplatz angelegt, sie sind an anderer Stelle definiert.

- es darf nur **eine** Definition in allen Quelldateien zu einem Programm geben!
- andere Dateien, die diese Variable benötigen, **müssen** diese mit einer **extern** -Deklaration bekannt machen!
- Initialisierung einer **extern** -Variable kann nur bei der Definition erfolgen!

Beispiele:

```
thales$ cat file.c
float c = 4.5;

thales$ cat main0.c

main() {
        printf("Wert von c: %f\n", c);
}

thales$ gcc file.c main0.c
main0.c: In function ‘main’:
main0.c:3: ‘c’ undeclared (first use this function)
main0.c:3: (Each undeclared identifier is reported only once
main0.c:3: for each function it appears in.)
```

Oder besser so?

```
thales$ cat file.c
float c = 4.5; /*Definition*/

titania$ cat main1.c
extern c;

main() {
        printf("Wert von c: %f\n", c);
}

thales$ gcc file.c main1.c
thales$ a.out
Wert von c: 1024.000000
```

Oder besser so?

```
thales$ cat file.c
float c = 4.5; /*Definition*/

thales$ cat main2.c

float c;

main() {
        printf("Wert von c: %f\n", c);
}

thales$ gcc file.c main2.c
```

```
thales$ a.out
Wert von c: 4.500000
```

Oder besser so?

```
thales$ cat file.c
float c = 4.5; /*Definition*/

thales$ cat main3.c
extern float c;

main() {
        printf("Wert von c: %f\n", c);
}

thales$ gcc file.c main3.c
thales$ a.out
Wert von c: 4.500000
```

16.3 static

Variablen, die innerhalb einer Funktion mit vorangestelltem **static** definiert werden, haben eine Lebensdauer, die über die der Funktion hinausgeht, sie werden beim ersten Aufruf erzeugt und behalten ihren Wert auch nach Bendigung der Funktion.

Variable, die außerhalb von Funktionen definiert werden, können mit dem vorangestellten **static** in ihrem Gültigkeitsbereich auf die Datei beschränkt werden. Auch Funktionen kann ein **static** vorangestellt werden, mit derselben Wirkung.

17. Modularisierung

17.1 Generelles

Entsprechend dem Prinzip in Modula-2 / Oberon werden auch in C größere Programme in "Teile" (Dateien!) zerlegt, wobei das Prinzip *Information Hiding* (in Modula-2 mit **DEFINITION MODULE - IMPLEMENTATION MODULE,** in Oberon mit **DEFINITION - MODULE**) zugrundegelegt werden sollte.

Zur Definition der Schnittstelle (DEFINITION) werden in C *header files (name.h)* verwendet, die per *include* "importiert" (besser: bekannt gemacht) werden (man beachte die Wirkung von *include).* In der Datei *name.c* erfolgt die zugehörige Implementierung. Hierbei kann es allerdings zu sehr unliebsamen (teilweise compilerabhängigen) Effekten kommen.

Beispiel (in Modula-2:)

```
DEFINITION Defs;
(* Defs.od *)
  CONST
        Factor = 1331;
        Upper = 100;
        Lower = 1;
  TYPE
        RandomNumber = INTEGER;
  VAR
                start: RandomNumber;

  PROCEDURE Rand(VAR s: RandomNumber):RandomNumber;
END Defs.
```

```
MODULE Defs;
(* Defs.om *)

  PROCEDURE Rand(VAR s: RandomNumber):RandomNumber;
  BEGIN
        s := Factor*start MOD Upper + Lower;
        start := s;
        RETURN s
```

```
  END Rand;
BEGIN
  start := 1
END Defs.
```

```
MODULE Main;
(* Main.om *)
        IMPORT Defs, Write;

        VAR s: Defs.RandomNumber;
BEGIN
        Defs.start := 2;
        Write.Int(Defs.Rand(s),0);
        Write.Ln
END Main.
```

Die erste Umsetzung in C könnte wie folgt aussehen:

```
/* defs.h */

# define Factor 1331
# define Upper 100
# define Lower 1

typedef int RandomNumber;

RandomNumber start = 1;
RandomNumber Rand(RandomNumber * s);
```

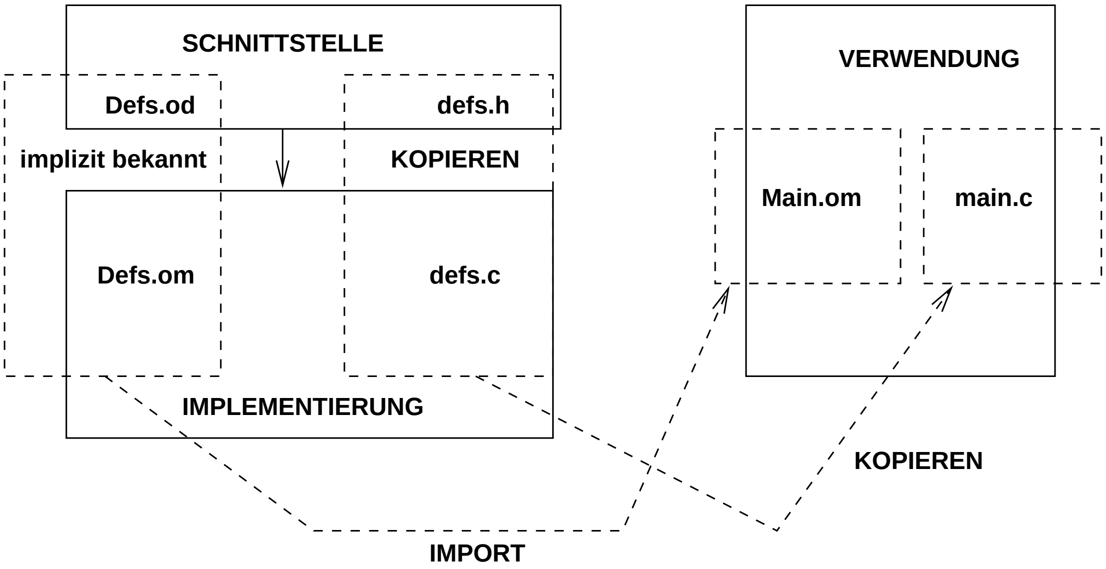
```
/* defs.c */

# include "defs.h"

RandomNumber Rand(RandomNumber * s) {
        *s = ((Factor*start) % Upper) + Lower;
                start = *s;
                return *s;
}
```

```
/* main.c */
# include <stdio.h>
# include "defs.h"

void main() {
        RandomNumber s;
        printf("Zufallszahl: %d\n", Rand(&s));
}
```

Einfach übersetzt:

```
hypatia$ gcc -Wall defs.c main.c
/tmp/cca003702.o(.data+0x0): multiple definition of `start'
/tmp/cca003701.o(.data+0x0): first defined here
hypatia$
```

So also nicht!



Das "Kopieren" des Header-Files muß unterschiedlich erfolgen:

```
/* defs1.h */

# ifdef defs_h
# define EXTERN
# define INIT = 2
# else
# define EXTERN extern
# define INIT
#endif
```

```
# define Factor 1331
# define Upper 100
# define Lower 1

typedef int RandomNumber;

EXTERN RandomNumber start INIT;

EXTERN RandomNumber Rand(RandomNumber * s);
```

```
/* defs1.c */
#define defs_h

# include "defs1.h"


RandomNumber Rand(RandomNumber * s) {
        *s = ((Factor*start) % Upper) + Lower;
                start = *s;
                return *s;
}
```

```
/* main1.c */

# include <stdio.h>
# include "defs1.h"

void main() {
        RandomNumber s = 2;
        start = 10;
        printf("Zufallszahl: %d\n", Rand(&s));
}
```

```
hypatia$ gcc -Wall defs1.c main1.c
hypatia$ a.out
Zufallszahl: 11
hypatia$
```

17.2 Abstrakte Datentypen (ADTs)

Prinzip: Nur der Typ-Name sowie die Operationen (Prozedur-Kopf) darauf werden an der Schnittstelle bekanntgegeben, die Implementierung (Typ-Struktur wie Prozedur-Rümpfe) selbst ist verborgen!

```
/* ---------- adt.h ------------*/

typedef struct _object * Object; /*ADT*/

typedef int BOOL;

/* Operationen: */
BOOL init(Object * x);
BOOL square(Object * x);

void printObjValue(Object x);

/* ------------- adt.c ----------------------------- */
# include <stdio.h>
# include <stdlib.h>
# include "adt.h"

struct _object {
  int cont;
  float value;
} ;


BOOL init(Object * x) {
  return (*x = (Object) calloc(1,sizeof(struct _object)))
                && ( (*x)->cont = 1 ) && ( (*x)->value=100);
}

BOOL square(Object * x) {
        (*x)->value *= (*x)->value;
        return 1;
}


void printObjValue(Object x) {
  printf("%f\n", x->value);
}
```

```
/* ------------------ mainAdt.c ------------- */
# include "adt.h"

Object x;
void main() {
        if ( init(&x) )
           printObjValue(x);
        if (square(&x) )
           printObjValue(x);
}
```

17.3 Abstrakter Datentyp “Stack”

```
/*
 *    stack.h - module interface for stack.c
 */

# ifndef STACK_H
# define EXTERN_H extern
# else
# define EXTERN_H
#endif

# ifdef ANSI_C
#       define VOID void
# else
#       define VOID char
# endif

typedef VOID *ELEMENT;
        /* type of data to be stored    */
typedef struct _stack * STACK;
        /* ADT STACK (hidden type)      */
typedef int BOOL;
        /* to indicate boolean results  */
typedef ELEMENT (*COPYPROC)(/*ELEMENT elem */);
        /* copy procedures              */

# ifdef ANSI_C
BOOL create (STACK *sp,  COPYPROC copy);
```

```
BOOL delete (STACK stack);
BOOL push   (STACK stack, ELEMENT elem);
BOOL pop    (STACK stack, ELEMENT *ep);
# else
EXTERN_H BOOL create ( /* STACK *sp,  COPYPROC copy */ );
EXTERN_H BOOL delete ( /* STACK stack */ );
EXTERN_H BOOL push   ( /* STACK stack, ELEMENT elem */ );
EXTERN_H BOOL pop    ( /* STACK stack, ELEMENT *ep  */ );
# endif
```

```
/* stack.c */

# include <stdlib.h>

#define STACK_H
#include "stack.h"

typedef struct _node *NODE;

#define true 1
#define false 0

struct _stack {
        COPYPROC copy;
        NODE top;
};

struct _node {
        ELEMENT elem;
        NODE next;
};

BOOL create (STACK *sp, COPYPROC copy) {
       return (*sp = (STACK) calloc(1,sizeof(struct _stack)))
                 && ((*sp)->copy = copy);
}
BOOL delete (STACK stack) {
       if (!stack || stack->top)
        /* stack == NULL || stack->top != NULL */
                return false;
        else {
        /* stack != NULL && stack->top == NULL */
                free(stack);
                return true;
        }
}
BOOL push(STACK stack, ELEMENT elem) {
       NODE node;
        if (stack && (node = (NODE) calloc(1,sizeof(struct _node)))
            && (node->elem = (*stack->copy)(elem))) {
                node->next = stack->top;
                stack->top = node;
                return true;
        }
```

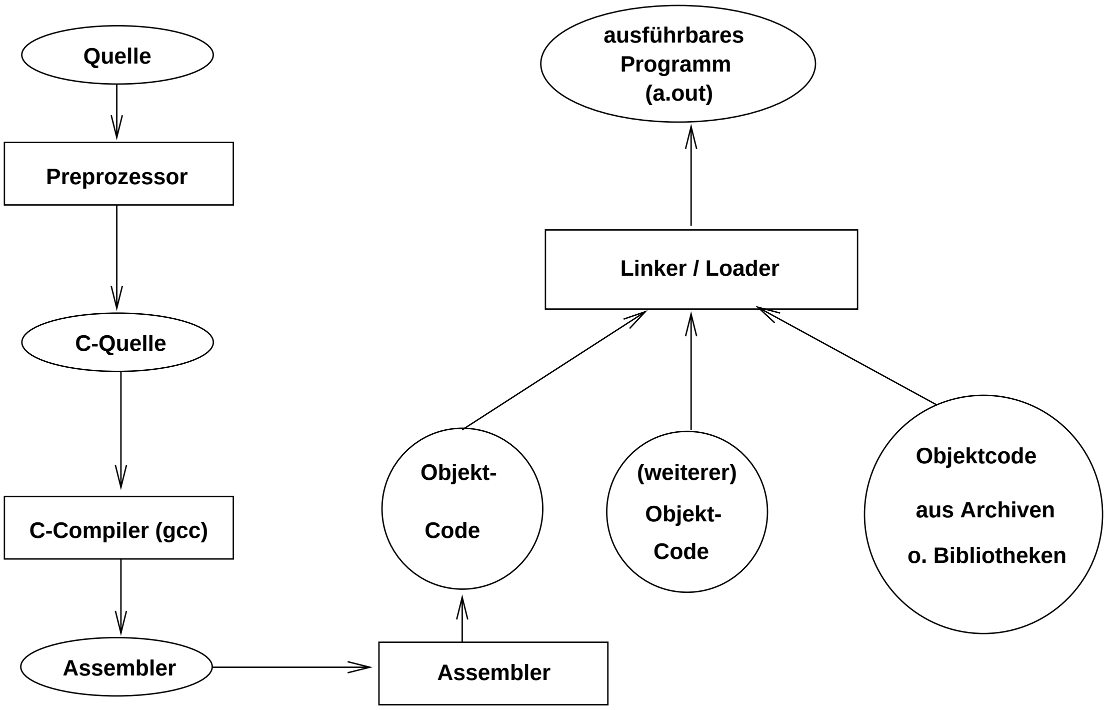
```
        else
                return false;
}
BOOL pop(STACK stack, ELEMENT *elem) {
        if (stack && stack->top) {
                NODE node = stack->top;
                stack->top = node->next;
                *elem = node->elem;
                free(node);
                return true;
        }
        else
                return false;
}
```

```
/* stack - main.c */
# include <string.h>
# include <stdio.h>
# include "stack.h"

void main() {
  char line[1024];
  char *out;
  STACK stack;

  printf("Ein Stack fuer Zeichenketten!\n");
  printf("Erstes Zeichen gibt Operation:\n");
  printf("+       : Push\n");
  printf("-       : Pop (incl. Top\n");
  printf("*       : Empty\n");

  if(create(&stack,strdup)) {
     while (gets(line)) {
       switch(line[0]) {
          case '+':
                  printf("pushed (%d): >%s<\n",
                          push(stack,line+1), line+1);
                  continue;
          case '-':
                  if (pop(stack,&out)) {
                     printf("popped: >%s<\n", out);
                  }
                  else
                      printf ("pop failed (stack empty?)\n");
                  continue;
          case '*':
                  while (pop(stack,&out)) {
                      printf("popped: >%s<\n",out);
                  }
                  printf("pop failed (stack empty?)\n");
                  continue;
          default:
                  printf("expected [-+*] as first char\n");
                  continue;
```

```
        }
      }
      printf("deleting stack:%d \n", delete(stack));
  }
}
```

18. Einige Tools

18.1 Compiler



C-Compiler

mehrere Durchgänge; z.B.:

- im ersten Durchgang:
  Zwischenräume, Kommentare entfernen und TOKEN bilden
- Assembler-Text erzeugen

```
hypatia$ cat trivial.c /* Intel-Prozessor */
/*trivial.c - ein trivial Programm*/

#include <stdio.h>
#include "mul.h"

int main() {
```

```
        int i, j;
        i= 2; j=4;
        i = mul(i,j);
        printf("Ergebnis: \%d\n", i);
        exit(0);
}
hypatia$ gcc -S trivial.c
hypatia$ cat trivial.s
        .file   "trivial.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "Ergebnis: %d\n"
.text
        .align 16
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        movl $2,-4(%ebp)
        movl $4,-8(%ebp)
        movl -8(%ebp),%eax
        pushl %eax
        movl -4(%ebp),%eax
        pushl %eax
        call mul
        addl $8,%esp
        movl %eax,%eax
        movl %eax,-4(%ebp)
        movl -4(%ebp),%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $8,%esp
        pushl $0
        call exit
        addl $4,%esp
        .align 16
```

```
.L1:
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.7.2.1"
hypatia$
```

Assembler: Erzeugt den Object-Code (Maschinensprache)

Linker/Loader (*ld*)

- sucht in seinen Argumenten nach Objektdateien und Bibliotheksdateien
- wenn gefunden, so liest er diese Datei, verschiebt den Programmcode und fügt dieses Programm am Ende der binären ausführbaren Datei an
- mischt die Symboltabelle des Objektprogramms mit der des binären ausführbaren Programms

18.2 Archivdateien

(→ **/lib** resp. **/usr/lib**)

Endung .a

- Linker/Loader durchsucht die Symboltabelle des Archivs, um Namen zu finden, die undefinierte Namen in der Symboltabelle der binären ausführbaren Datei auflösen
- diejenigen Objektprogramme des Archivs, die Namen "auflösen", werden zur ausführbaren Datei hinzugefügt Der *Linking/Loader* (**ld**) wird typischerweise indirekt über den C-Compiler aufgerufen:

```
hypatia$ cat math.c
# include <stdio.h>

extern double sin();

void main() {
        double x;

        printf("sin(%f) = %f\n",x, sin(x));
```

```
}
hypatia$ gcc -Wall math.c

/tmp/cca011561.o: In function ‘main’:
/tmp/cca011561.o(.text+0xd): undefined reference to ‘sin’

hypatia$ gcc -Wall math.c -lm # libm.a

hypatia$ a.out
sin(0.000000) = 0.000000
hypatia$
```

Häufig verwendete Bibliotheken (Archivdateien) (→ /usr/lib):

- libc.a: C-Standardbibliothek
- libm.a: mathematische Funktionsbibliothek
- libcurses.a: Bildschirmsteuerung

Eigene Bibliotheken

Kommando **ar** (→ *man ar*)

18.3 make, [Mm]akefile

- *make* ist ein Werkzeug zur Verwaltung mehrerer, miteinander in Beziehung stehender Dateien
- in Zusammenhang mit Modula-2 / Oberon und dem Kommando **mmm** (*make modula makefile*) / **mmo** (*make makefile for oberon*) wurde *make* zur effizienten Übersetzung von Modula-2- / Oberon-Modulen benutzt
- die von *make* interpretierte Steuerdatei *makefile* (Name kann auch anders gewählt werden und muß dann bei *make* als Argument angegeben werden (*make -f my_makefile*) (siehe *man make*) muß bei C-Programmen selbst erstellt werden

Aufbau der Steuerdatei makefile

makefile enthält Einträge der Form

```
Target: <TAB> {<TAB>} {Dependant}
<TAB> {<TAB>} Command
<TAB> {<TAB>} Command
                .
                .
                .
```

Target und *Dependant* sind in der Regel Dateinamen, *Command* ein beliebiges UNIX-Kommando oder ausführbares Programm.

Das Zeichen "#" leitet einen Kommentar bis zum Zeilenende ein.

Beispiel

```
/* main.h */

#define LINE_SIZE       80
/*---------------------------------
 *  main.c
 *---------------------------------
 */
#include <stdio.h>
#include "main.h"
#include "eingabe.h"
#include "ausgabe.h"

int main() {      /* Hauptprogramm */
char line[LINE_SIZE];

   while (eingabe(line)) {
      ausgabe(line);
   };
   printf("\n");
   exit(0);
}
```

```
/* eingabe.h */

extern int eingabe();
/*--------------------------------------
 * eingabe.c
 *
 *  Liest eine Zeile von stdin und schreibt
 *  sie in den uebergebenen String.
 *
 *  Returnwert : die Laenge des eingegebenen
 *               Strings.
 *--------------------------------------
 */

#include <stdio.h>

int eingabe(char *buffer) {
char * get_buf;

        printf("\t Eingabe : ");
        get_buf = gets(buffer);
        return ( get_buf == NULL ? 0 : strlen(get_buf));
}
```

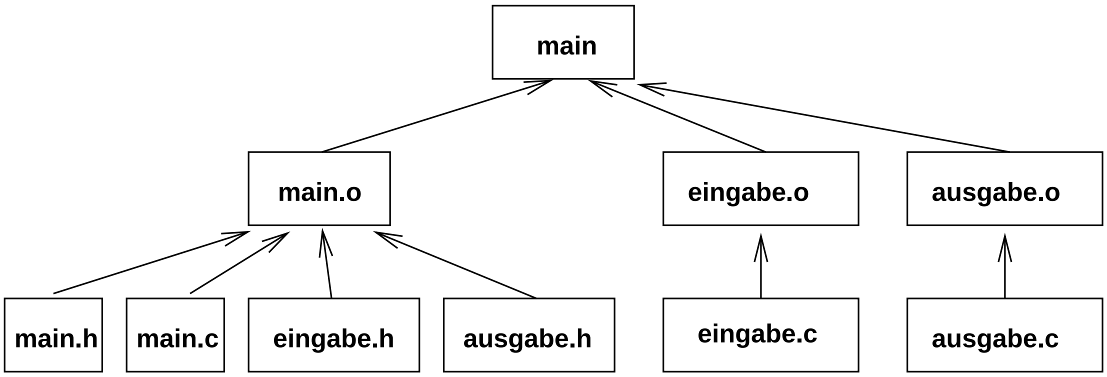
```
/* ausgabe.h */

extern void ausgabe();
/*------------------------------------
 * ausgabe..c
 *
 *  Gibt eine Zeile in Grossbuchstaben am
 *  Bildschirm aus.
 *--------------------------------------
 */
#include <stdio.h>

char * upper();
```

```
void ausgabe(char *buffer) {
   printf("\t Ausgabe : %s \n\n", upper(buffer));
}


/* String-Konvertierung in Grossbuchstaben */
char *upper(char *string) {
char *pos;
int offset;

   offset = 'a' - 'A';
   for (pos = string; *pos != '\0'; pos++) {
      if (*pos >= 'a' && *pos <= 'z')
         *pos = *pos - offset;
   };
   return string;
}
```

Zur effizienten Übersetzung dieser drei C-Programme kann die folgende Steuerdatei verwendet werden:

```
# MAKEFILE
# Version 1

main:   main.o eingabe.o ausgabe.o
        gcc -Wall -o main main.o eingabe.o ausgabe.o

main.o:         main.c main.h eingabe.h ausgabe.h
                gcc -Wall -c main.c

eingabe.o:      eingabe.c
                gcc -Wall -c eingabe.c

ausgabe.o:      ausgabe.c
                gcc -Wall -c ausgabe.c

clean:
                rm -f main.o ausgabe.o eingabe.o core
realclean:
                rm -f main.o ausgabe.o eingabe.o main core
```

make erzeugt aus dieser Beschreibung von Abhängigkeiten folgende Baumstruktur:



Dieser Baum wird in *postorder* traversiert. Wird dabei festgestellt, daß mindestens eine der Nachfolger-Dateien jünger ist als ihre Vaterknoten-Datei oder daß die Vaterknoten-Datei nicht existiert, so werden die dem Vaterknoten zugeordneten Kommandos ausgeführt.

```
hypatia$ ls
ausgabe.c   eingabe.c   main.c      makefile
ausgabe.h   eingabe.h   main.h
hypatia$ make
gcc -Wall -c main.c
gcc -Wall -c eingabe.c
gcc -Wall -c ausgabe.c
gcc -Wall -o main main.o eingabe.o ausgabe.o
hypatia$ ls
ausgabe.c   ausgabe.o   eingabe.h   main        main.h      makefile
ausgabe.h   eingabe.c   eingabe.o   main.c      main.o
hypatia$ make
make: `main' is up to date.
hypatia$ touch main.h
hypatia$ make
gcc -Wall -c main.c
gcc -Wall -o main main.o eingabe.o ausgabe.o
hypatia$ ls
ausgabe.c   ausgabe.o   eingabe.h   main        main.h      makefile
ausgabe.h   eingabe.c   eingabe.o   main.c      main.o
hypatia$ make clean
rm -f main.o ausgabe.o eingabe.o core
hypatia$ ls
ausgabe.c   eingabe.c   main        main.h
ausgabe.h   eingabe.h   main.c      makefile
hypatia$ make
gcc -Wall -c main.c
gcc -Wall -c eingabe.c
gcc -Wall -c ausgabe.c
gcc -Wall -o main main.o eingabe.o ausgabe.o
hypatia$ make realclean
rm -f main.o ausgabe.o eingabe.o main core
hypatia$ ls
```

```
ausgabe.c   eingabe.c   main.c      makefile
ausgabe.h   eingabe.h   main.h
hypatia$
```

Anmerkungen

- Wird *make* ohne Parameter aufgerufen, so beginnt die Abarbeitung der Steuerdatei mit dem ersten *target*; soll die Abarbeitung mit einem anderen *target* beginnen, so muß dieses als Parameter mitgegeben werden (*make clean* oder *make realclean* in obigem Beispiel).
- Heißt die Steuerdatei nicht *makefile* bzw. *Makefile*, so muß sie mit der Option **-f** an *make* gegeben werden.
- Die Option **-n** veranlaßt *make*, die resultierenden Befehle am Bildschirm auszugeben, ohne sie jedoch auszuführen
- Man kann im *makefile* Makros definieren und bereits vordefinierte Makros benutzen, um das *makefile* leichter modifizierbar und kompatibler zu gestalten (s.u.).
- *make* geht davon aus, daß eine **.o**-Datei von der zugehörigen **.c**-Datei abhängt und mit dem C-Compiler aus ihr erzeugt wird; diese Abhängigkeiten müssen also nicht explizit im *makefile* aufgeführt werden.
- Ein *target* muß nicht unbedingt ein Dateiname sein, und die Liste der *dependants* kann auch leer sein. Dann werden die zugehörigen Kommandos beim Erreichen des *targets* ausgeführt (siehe im Beispiel *clean*, *realclean*).

```
# MAKEFILE
# Version 2
MAIN= main
EIN=eingabe
AUS=ausgabe
OBJ=$(MAIN).o $(EIN).o $(AUS).o

main:           $(OBJ)
                $(CC) -o $(MAIN) $(OBJ)
```

```
main.o:          $(MAIN).h $(EIN).h $(AUS).h
                 $(CC) -c $(MAIN).c

clean:
                 rm -f $(OBJ) core

realclean:
                 rm -f $(OBJ) $(MAIN) core
```

Betriebssysteme

1. Literatur

/Bach86/

M. Bach: The Design of the UNIX Operating System. Prentice Hall 1986

/Comer84/

D. Comer: Operating System Design: The XINU Approach. Prentice Hall 1984

/Darnell88/

P.A. Darnell, P.E. Margolis: Software Engineering in C. Springer 1988

/Herold92/

H. Herold: UNIX-Shells. Addison-Wesley 1992

/Herold95/

H. Herold, M. Mayer: SCCS und RCS - Versionsverwaltung unter UNIX. Addison-Wesley 1995

/Kernighan87/

B.W. Kernighan, R. Pike: Der UNIX-Werkzeugkasten. Hanser 1987

/Rochkind88/

M.J. Rochkind: UNIX-Programmierung für Fortgeschrittene. Hanser 1988

/Schreiner84/

A.-T. Schreiner: System-Programmierung in UNIX - Teil1: Werkzeuge. Teubner 1984

/Stevens90/

W.R. Stevens: UNIX Network Programming. Prentice Hall 1990

/Tanenbaum87/
A. S. Tanenbaum: Operating Systems - Design and Implementation. Prentice Hall 1987

2. Einführung und Übersicht

2.1 Betriebssystem

DIN 44300 schreibt vor:

- “Zum Betriebssystem zählen die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.”

Mit einfachen Worten:

- Nach dem Einschalten des Rechners muß ein erstes Programm gestartet werden, das alle Resourcen des Rechners verwaltet.
  Dieses Programm nennt man “Betriebssystem”.
- Das Betriebssystem ist ein Programm, das immer arbeitet, solange der Rechner in Betrieb ist.
- Die Hauptaufgabe des Betriebssystems besteht im Vermitteln zwischen dem Benutzer/Benutzerprogramm und der Hardware.
  Zu diesem Zweck stellt das Betriebssystem alle Leistungen des Rechners den Anwenderprogrammen zur Verfügung, schützt aber gleichzeitig auch Hardware und Software vor unberechtigtem Gebrauch.

2.2 Aufgaben eines Betriebssystems

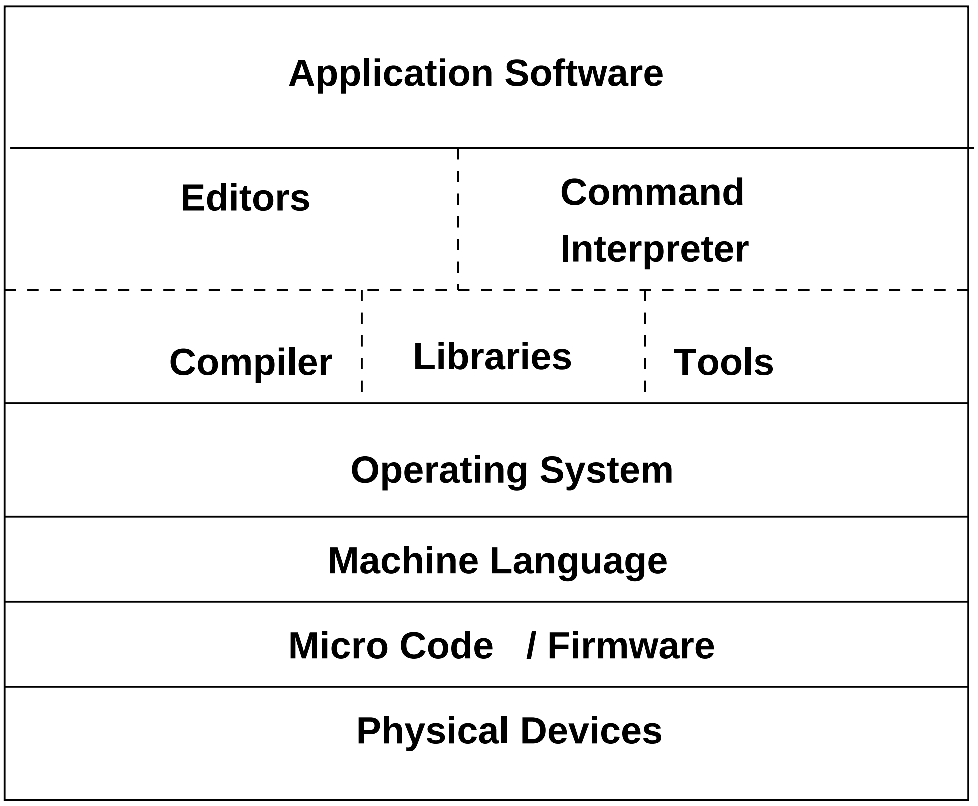
- **Resource Manager**
  Das Betriebssystem kontrolliert alle Hardware- und Software-Komponenten eines Rechners und teilt sie **effizient** den einzelnen Nachfragern zu.
  Das Betriebssystem stellt Basis-Dienstleistungen (Dateizugriff, Prozeßmanagement) zur Verfügung, auf denen

Anwendungsprogramme aufsetzen können.

- **Extended Machine (Virtual Machine Abstraction)**
  Das Betriebssystem besteht aus einer (oder mehreren) Schichten von Software, die über der "nackten" Hardware liegen: *Virtual Machine.*
  Diese ist einfacher zu verstehen und zu programmieren. Komplexe Hardware-Details verbergen sich hinter einfachen und einheitlichen *extended instructions.*
  Der Programmierer erhält vom Betriebssystem eine angenehmere Schnittstelle zur Hardware als von der reinen Hardware (instruction set, register structure, memory organisation, I/O structur, bus structure,...). Er muß sich nicht mehr um sämtliche maschinenabhängige Details kümmern.

2.3 Schichtenmodell

Layered System - Extended Machine



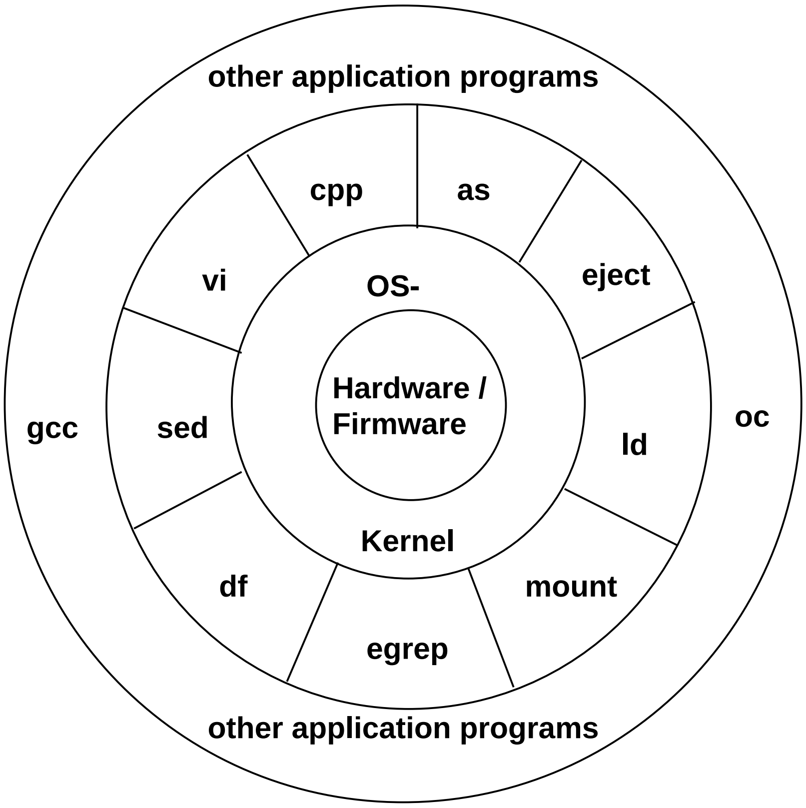
- *Physical Devices*
  IC Chips, Drähte, Platten, Stromversorgung,...
- *Microcode*
  Primitive Software, die die Physical Devices direkt kontrolliert.
  Bietet der nächsten Schicht eine einheitlichere Schnittstelle zu den Physical Devices. Diese Schicht verbirgt bereits etwas von den Details der direkten Gerätesteuerung.
- *Machine Language*
  Schnittstelle zwischen Hardware und Software.
  50 - 300 Instruktionen (ADD, MOVE, JUMP,...)
  Die meisten Instruktionen dienen dem Bewegen, Laden und Speichern von Daten im Hauptspeicher und in den Registern, dem Ausführen von arithmetischen Operationen, oder dem Vergleichen von Werten.
- *Operating System*
  Vermittelt zwischen Benutzer/Benutzerprogrammen und der Hardware.
  Verbirgt die Komplexität der Hardware vor dem Programmierer.
  Bietet angenehmere Instruktionen für's Programmieren:
  READ FROM FILE statt *Move head of disk 1 to track 231, lower head and ...*
- *System Software*
  Compiler, Editoren, Kommando Interpreter, Dienstprogramme

  Diese Programme gehören nicht zum Kern des Betriebssystems, werden aber meist vom Hersteller des Betriebssystem-Kernels mitgeliefert.

  Benutzer können diese Programme durch andere/eigene ersetzen, was bei den Komponenten des Kernels nicht möglich ist.
- *Application Software*
  Von Benutzern zur Lösung ihrer Probleme geschrieben.

2.4 Layered System UNIX

Schalenmodell des UNIX Betriebssystems (frei nach /Bach86/):



Layered System - UNIX

- Abstraktion
  Die Hardware versorgt den Kernel mit Basis-Dienstleistungen. Das Betriebssystem versorgt alle Programme mit System-Dienstleistungen. Es isoliert die Programme von Hardware-Eigenheiten. Der Programmierer kann mit abstrakteren Objekten arbeiten. Programme werden hardwareunabhängig und dadurch leichter portierbar.

- Interaktion
  Programme interagieren mit dem Betriebssystem-Kernel durch Aufrufe von genau definierten *System Calls,* um vom Kernel Dienstleistungen zu erhalten und um mit dem Kernel Daten auszutauschen.

- Kombination
  Die Benutzerprogramme und "Kommandos" befinden sich in der gleichen Schicht. Sie können aber aufeinander aufbauen, sich gegenseitig aufrufen und/oder Daten miteinander austauschen.

  Beispiel: *cc* koordiniert *cpp, comp, as* und *ld.*

Kernel - Dienstleistungen

- System Calls
  Alle Programme fordern Dienstleistungen des Kernels per *System Calls* an. Vom Programm aus erfolgt dies in Form eines Funktionsaufrufs.

  Die System Calls definieren die **Schnittstelle** zwischen Kernel und Benutzerprogramm.

- Subsysteme

  Die System Calls lassen sich nach ihrer Funktion den beiden großen Subsystemen im Kernel zuordnen:

  I/O Subsystem und **Process Subsystem**

  Das I/O Subsystem manipuliert die sogenannten statischen Objekte (Files), das Process Subsystem die dynamischen Objekte (Prozesse) des Rechners.

- Zugriffskontrolle

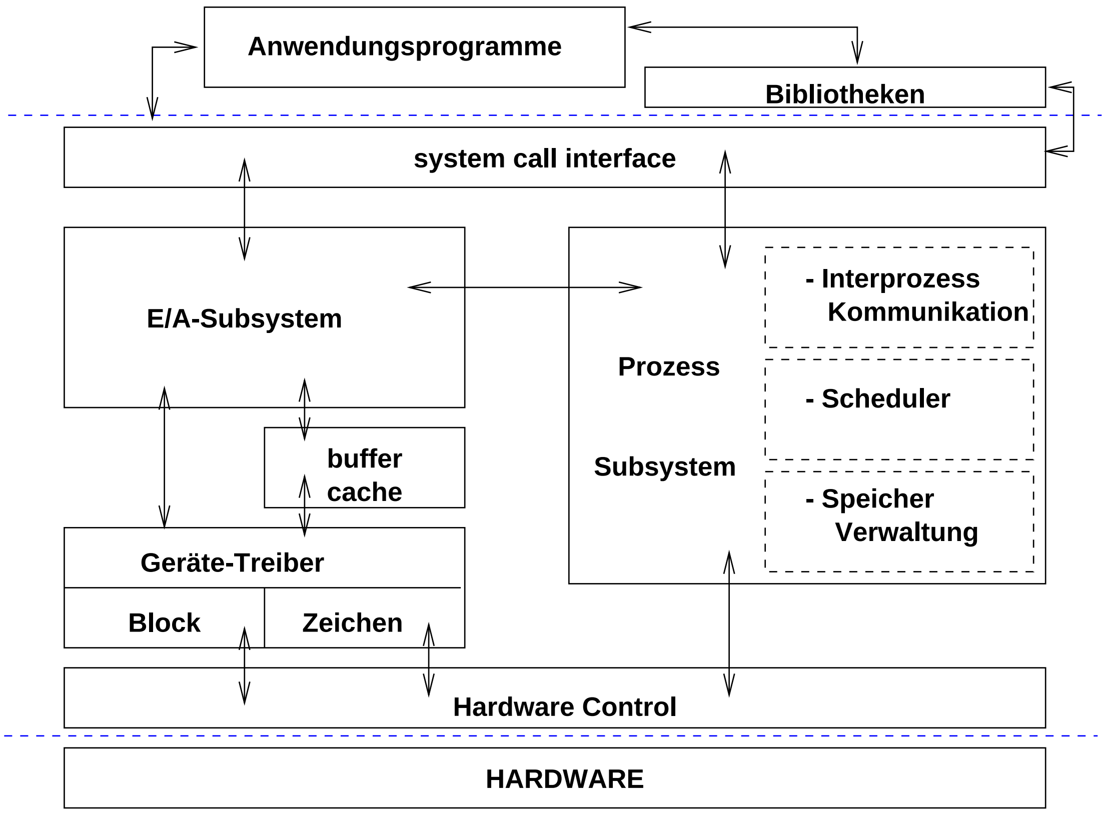
  Benutzerprogramme haben keinen direkten Zugriff auf Resources im Rechner (CPU, Hauptspeicher, Platten,...). Die System Calls bieten den einzig möglichen, aber gut kontrollierbaren Zugang zu den Resources.

- Transparenz

  Die Dienstleistungen des Kernels sind transparent.

  Beispiel: Datei, Gerät, Pipe = unformatierter Bytestrom

Aufbau des UNIX Betriebssystems (Übersicht mit den zentralen Komponenten, frei nach /Bach86/



2.5 Struktur von Unix

2.5.1 Das I/O Subsystem

Aufgaben:

- Zugriff auf Geräte (== Files !!) ermöglichen
- Kontrolle der Zugriffsrechte auf Files
- Speichern und Wiederbeschaffen von Benutzerdaten
- Bereitstellen von Platten-Speicherplatz
- Verwalten von freiem Plattenplatz

Die Interaktion von Prozessen mit dem I/O Subsystem erfolgt über System Calls oder die Standard I/O-Bibliothek.

Beispiele:

chdir	Katalog wechseln
chmod	Zugriffsrechte ändern
chown	Besitzer ändern
close	Schließen einer Datei
creat	Erzeugen einer Datei ("alt")
dup	Filedeskriptor duplizieren
dup2	4.3BSD, ähnlich zu dup
fnctl	Eigenschaften einer offenen Datei ändern
ioctl	ähnlich zu fcntl, geräte-spezifisch
link	Neuer Name für bestehende Datei
lseek	Positionieren in einer Datei
mknod	*special files* erzeugen
mount	Filesystem in bestehendes "einhängen"
open	Öffnen/Erzeugen einer Datei
pipe	*unnamed pipe* erzeugen
read	Lesen aus einer Datei
stat	Attribute einer Datei lesen
umount	Filesystem "aushängen
unlink	Namen entfernen
write	Schreiben in eine Datei

Das I/O Subsystem benutzt einen Puffermechanismus beim Zugriff auf Daten, die auf einem *block device* (z.B. Festplatte) lagern: *Buffer Cache* .

Block devices sind Geräte, die Daten in beliebig adressierbaren Blöcken speichern und übertragen können. Beispiel: Festplatten.

Char devices sind Geräte auf die ungepuffert zugegriffen wird, gemäß dem Modell "unformatierter sequentieller Bytestrom". Beispiele: Terminal, Drucker.

Device Driver heißen die Kernel Module, die die Operationen der "physical devices" im Detail kontrollieren. Sie haben eine *device independent* Schnittstelle nach oben zu den übrigen Kernel-Routinen und eine *device dependent* Schnittstelle nach unten zur Hardware hin.

2.5.2 Das Process Subsystem

Aufgaben:

- Prozesse managen (kreieren, terminieren, synchronisieren)
- Interprocess Communikation (IPC) ermöglichen
- Memory verwalten

Die Ausführung eines Programms heißt *Prozeß* .

Das Process Subsystem benötigt Service vom I/O Subsystem, wenn ein Programm gestartet (Image von Platte in Speicher) oder wenn Prozeßdaten aus/eingelagert werden müssen.

Benutzerprogramme/Prozesse erhalten vom Process Subsystem Serviceleistungen über System Calls wie:

brk	Datenadreßraum ändern
exec(*)	Text eines Prozesses mit neuem Text überlagern
exit	Prozeß terminieren
fork	Erzeugen eines neuen Prozesses
kill	Signal senden
setuid	User Id ändern
signal	Mit Signalen umgehen
wait	Auf Termination eines erzeugten Prozesses warten

Der *Memory Management* Teil kümmert sich um die Zuteilung des Hauptspeichers an die einzelnen Prozesse. Er übernimmt das Ein- und Auslagern von Prozeßdaten vom Primärspeicher zum Sekundärspeicher und umgekehrt:

- *swapping* (alle Prozeßdaten werden verlagert)
- *demand paging* (nur Teile des Adreßraums, sog. *pages* werden vorgehalten)

Der *Scheduler* teilt die CPU den einzelnen Prozessen zu. Er entscheidet anhand von Prioritäten und der Uhr *(timeslice-* Verfahren) , welcher Prozeß auf der CPU arbeiten darf.

Der *IPC* (Interprocess Communication) Teil versorgt alle Prozesse mit Dienstleistungen zur Kommunikation untereinander:

- *synchroner* Datenaustausch: Pipes, Message Queues, shared memory, sockets
- *asynchrone* Signale.

2.6 System Calls - Execution Mode

- Execution Mode

  Die Hardware führt zu jedem Zeitpunkt entweder Anweisungen eines Benutzerprogramms oder Anweisungen im Kernel-Code aus.

  Bei der Ausführung von Anweisungen und beim Zugriff auf den Speicher unterscheidet die Hardware zwischen zwei Zuständen, in denen sie sich jeweils befinden kann:

  User Mode
  Darin befinden sich Prozesse beim Ausführen der meisten ihrer Anweisungen zur Lösung ihres anwendungsspezifischen Problems.

  Kernel Mode
  Darin befinden sich Prozesse dann, wenn sie Dienstleistungen des Kernels benötigen.

- Instruction Set

  Die *Machine Language* legt die Schnittstelle zwischen Hardware und Software fest. Der Übergang ist definiert durch das *Instruction Set* , die Menge aller Anweisungen, die die Hardware ausführen kann.

  Die Hardware unterteilt diese *Machine Language Instructions* in *privilegierte* (z.B. Manipulation von Prozessor Status Register) und *nicht-privilegierte* Anweisungen und stellt den Schutz dieser Anweisungen sicher:

  Im *User* Mode können **nur** *nicht-privilegierte* Anweisungen,

  im *Kernel* Mode können auch *privilegierte* Anweisungen ausgeführt werden.

  Damit können Prozesse im *User Mode* durch die Hardware von unkontrollierten Manipulationen an wichtigen Datenstrukturen und Registern (Stack pointer, PSW,...) des Kernels abgehalten werden.

- Address Space

  Für den Zugriff auf Speicheradressen stellt die Hardware ebenfalls einen zweistufigen Schutzmechanismus zur Verfügung.

  Jeder Bereich des Speichers, meist auf *Seiten* -Basis, gehört entweder zum *User* oder *Kernel* Adreßraum. Die Memory Management Hardware sorgt dafür, daß

  im *User* Mode nur auf Speicheradressen des *User* Adreßraums und ausschließlich

  im *Kernel* Mode auf beide Bereiche zugegriffen werden kann.

- Hardware Protection

  Abstraktionen wie Prozeß oder Benutzer sind auf Hardware-Ebene unbekannt.

  Die Hardware kann nur Teile des Adreßraums und bestimmte Anweisungen vor unberechtigtem Zugriff schützen.

2.7 System Calls - Kernel Services

Kernel Services

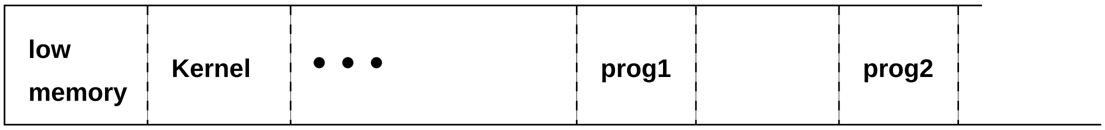
System Calls sind die einzige Möglichkeit, die ein Programm hat, um vom Kernel Dienstleistungen zu erhalten.

Modell

Benutzerprogramm und Betriebssystem (Kernel) sind **nicht** zwei separate Einheiten, die von einander unabhängig laufen. Sie operieren stattdessen als ein grosses logisches Programm mit vielen Unterprogrammen.

Kernel und Programm grenzen sich voneinander ab wie zwei Module eines gut strukturierten Programms. Beide Module besitzen genau definierte Einstiegspunkte für den gegenseitigen Aufruf.

Der Kernel ist speicherresidenter Code, Programme werden in den Speicher geladen und entfernt. Alle Programme nutzen gemeinsam den selben Code im Kernel Adreßraum.



Aufruf- und Ablaufsequenz

- Kernel arbeitet (Kernel Mode)
  Anweisungen im Kernel Adreßraum werden ausgeführt.
- Kernel startet *prog1* (Kernel Mode)
  Kernel richtet Umgebung *(Process Context)* für *prog1* im Speicher ein,
  schaltet in *User* Mode und
  weist die CPU an (Unterprogrammaufruf), die Anweisungen ab der Startadresse von *prog1* auszuführen.

- *prog1* arbeitet (User Mode)
  Dabei kann nur auf Anweisungen und Daten im Adreßraum von *prog1* zugegriffen werden.

- *prog1* fordert Kernel Service an (noch User Mode)

  Parameter hinterlegen - System Call ausführen (Unterprogrammaufruf) - Spezielle *trap* -Anweisung ausführen
  Übergang in *Kernel* Mode.
  Im *low memory* (Kernel Adreßraum) ist festgelegt, wo nach einer *trap* die Ausführung weitergehen soll.

- Kernel arbeitet (Kernel Mode)

  Entsprechend dem aufgerufenen System Call wird eine Serviceroutine im Kernel Adreßraum ausgewählt, die den geforderten Kernel Service liefert.

Die Serviceroutine hat sowohl Zugriff auf Adressen des Kernel Adreßraums als auch auf Adressen des User Adreßraums.
Als letzte Aktion veranlaßt die Serviceroutine das Zurückschalten in den *User* Mode und den Rücksprung zur Stelle des System Call Aufrufs in *prog1*

- *prog1* arbeitet (User Mode)

- • • •

- *prog1* terminiert (User Mode)

  Implizite oder explizite Ausführung des System Calls *exit().*
  Rückkehr zum *Kernel* Mode und Adreßraum.

- Kernel arbeitet (Kernel Mode)

  Kernel gibt alle von *prog1* belegten Speicherflächen (Umgebung) frei und beendet damit die Existenz von *prog1*

2.8 System Calls - Beispiel *read*

```
count = read(fileDeskriptor, buffer, nbytes);
```

Beschreibung:

Vom offenen Kanal, identifiziert über eine kleine Zahl *(File Deskriptor),* in den *buffer* bis zu *nbytes* Daten lesen.

Zurück kommt die Anzahl der tatsächlich gelesenen Bytes.

Ablauf

- 1. Schritt

  read(..) ist ein normaler Aufruf einer Bibliotheksfunktion.

  Die diesem System Call entsprechende *Stub -Funktion* befindet sich in der Systembibliothek, die für jeden System Call eine Stub-

Funktion enthält.

- 2. Schritt

  Die Stub-Funktion für *read* führt die *trap* -Anweisung mit den entsprechenden Parametern aus und bewirkt damit einen Wechsel in den Kernel Mode.

- 3. Schritt

  Kernel-Routinen arbeiten, um den *read* System Call zu befriedigen. Nach Abschluß aller Kernel-Routinen erfolgt der Rückwechsel in den User Mode und der Rücksprung zur Stub-Funktion.

- 4. Schritt

  Die Stub-Funktion kümmert sich noch um den Return-Wert und gibt ihrerseits zurück zum Aufrufer von *read.*

- 5. Schritt

  Das Programm prüft den Rückgabewert (!!) des System Calls und gegebenfalls auch den Wert von *errno* .

```
if((count = read( fileDeskriptor, buffer, nbytes)) < 0 ){
        perror( "read" );
        exit( 1 );
}
```

Anschließend kann das Programm mit den durch den System Call erhaltenen Daten weiterarbeiten.

Anmerkungen:

- Die meisten System Calls liefern als Rückgabewert -1 bei Fehler, eine Zahl >=0, falls okay
- Viele System Calls sind hinter C-Bibliotheksfunktionen verborgen (C-Interface zu Unix)
- Das C-Interface enthält eine globale Variable **int errno**, die bei Auftreten eines Fehlers die Unix-Fehlernummer enthält.
- Die Datei **<errno.h>** enthält dazu Fehlertexte
- Bei erfolgreicher Ausführung eines System Calls wird die Fehlernummer (*errno* **nicht** zurückgesetzt - also nur abfragen, falls tatsächlich Fehler passiert ist.
- Manche System Calls (*stat, fstat*) liefern Zeiger auf Struktur.

2.9 Benutzer- / Datei- / Prozeß-Identifikatoren

Dateien, die von Prozessen geöffnet sind, werden ebenso wie die Prozesse selbst über Integer-Zahlen identifiziert.

- *File Descriptors:* siehe System Calls im I/O-Subsystem
- *Process ID* siehe Kommando *ps* oder Funktion *int getpid()*
- *Real User ID:* siehe /etc/passwd

```
# include <stdio.h>
# include <unistd.h>

void main() {
        printf("User Id: %d\n", getuid());
}
```

- *Real Group ID:* Benutzergruppen (siehe /etc/group, /etc/passwd)

```
# include <stdio.h>
# include <unistd.h>

void main() {
        printf("Group Id: %d\n", getgid());
}
```

- *Effective User ID:*
  Im Normalfall identisch mit *real user ID*

  ABER: Manche Programme erhalten ein spezielles Flag (*set-user-ID-Bit*), das besagt: "Wenn dieses Programm ausgeführt wird, ändere die effektive User Id des Prozesses in die User ID des Besitzers der Datei"

  unsigned short geteuid();

- *Effective Group ID*
  siehe oben;

  unsigned short getegid();

3. Das I/O-Subsystem

3.1 Einführung

In einem *UNIX* System wird jede Art von Information in Dateien *(files)* gespeichert. Ein *file* ist schlicht ein Behälter für Information.

Neuartig am *UNIX* File System ist die Idee, praktisch alle Objekte des Systems als Files zu repräsentieren. Trotz dieses gewaltigen Abstraktionsschrittes benötigt *UNIX* nur wenige verschiedene Dateiarten *(filetypes):*

- *ordinary (or regular) file:*
  Container für jede Art von Daten (Text wie binär).
- *directory file:*
  Verzeichnis für die Namen von Files und zugehörige Verweise auf ‘‘Verwaltungsinformation’’ (Inode).
- *named pipes (FIFO’s):*
  zur unidirektionalen Prozeßkommunikation
- *device special files (character special file / block special file):*
  Geräte und Hardware allgemein, zu Files abstrahiert
  (Terminal, Drucker, Platten, Memory, ...).
- *symbolic link*
  Verweis auf den Pfadnamen einer anderen (u.U. nicht existierenden) Datei (4.3BSD)
- *socket:*
  Kommunikationsstruktur, insb. in Rechnernetzen, 4.3BSD
- und einige wenige mehr (System V, BSD)

3.2 Der Dateibegriff

- Definition
  Every file is a sequence of bytes.

Zu einer Datei gehören

- ein oder auch mehrere Namen
- Inhalt und Aufbewahrungsort (Menge von Blöcken auf der Platte)
- Verwaltungsinformationen (Besitzer, erlaubter Zugriff, Zeitstempel, Länge, Dateityp, ...).

UNIX verlangt / unterstellt bei regulären (gewöhnlichen) Dateien keinerlei Struktur und unterstützt auch keine. Die Konzepte "variabel oder konstant lange Records" sind im Kernel von *UNIX* nicht implementiert.

Ein File wird abstrakt als **Datenstrom** repräsentiert.

3.3 File System

Ein UNIX-Dateisystem besteht i.d.R. aus mehreren Dateisystemen - ein spezielles Dateisystem ist das *root file system*.

Layout:

boot block	super block	inode list	data blocks

- **boot block**
  enthält beim **root file system** den sog. *bootstrap code*, der beim "Hochfahren" der des Betriebssystems zuerst geladen wird
  bei anderen File Systemen ist er vorhanden, aber leer

- **super block**
  – Größe des Filesystems
  – Anzahl freier Blöcke
  – Liste der freien Blöcke
  – Zeiger auf nächsten freien Block in der Liste der freien Blöcke
  – Größe der Inode-Liste
  – Anzahl freier Inodes
  – Liste der freien Inodes
  – Zeiger auf nächste freie Inode

– "Sperr-Felder" für Liste der freien Blöcke / Inodes
– Anzeigefeld, ob *super block* verändert wurde

- **Inode List**
  siehe unten - eine *inode* ist die sog. **root-inode**, die den Wurzel-Katalog beschreibt und beim Zusammenfügen von Dateisystemen (System Call **mount**) wichtig ist.

3.4 Directory

Dieser Dateityp weist eine Struktur auf:

	Lokaler Dateiname	Verweis auf Inode-Liste (Index)
	.	**4711**
	..	**4709**
	onefile	**47119**
	otherfile	**471110**
	subdir_1	**471130**

Damit ergibt sich die bekannte Benutzer-Sicht: Hierarchisches Dateisystem oder Dateibaum!

Die Blätter im Baum sind *ordinary files, named pipes* oder *device special files,* die anderen Knoten *directories.*

Anmerkung: Deshalb nennt man Directories auch **Kataloge** oder **Verzeichnisse.**

3.5 File System - Standard Directories

- **root directory**

  Der Katalog am oberen Ende des Dateibaums '/' heißt **Wurzel** (**root**). Dieser Katalog ist direkter oder indirekter Vorgänger aller Files und Directories im gesamten Dateibaum und hat selbst keinen Vorgänger.

- **top level directories**

  Im oberen Bereich des Dateibaums befinden sich Kataloge mit verschiedenen Systemdaten.

/bin	ausführbare Programme (*binaries*)
/lib	Bibliotheken und Hilfsdateien
/tmp	Platz für temporäre Dateien, benutzbar für alle User
/etc	Konfigurationsdateien und Systemverwaltungs-kommandos
/dev	Gerätedateien (*device special files*)
/usr	Userdaten und eher lokale Systemdateien /usr/bin */usr/lib* */usr/tmp* /usr/man
/usr/spool	Zwischenablage für Druckaufträge, externe Post, usw. (*spooling area*)
/usr/local	lokale Software, meist ebenfalls unterteilt in Kataloge mit den Namen *bin, lib, man* und *src*

- Dateinamen
  Dateinamen sind Verweise von einem Directory zu einem File.

  Sie bestehen aus Zahlen, Buchstaben und Sonderzeichen. Erlaubt sind im Prinzip alle Zeichen außer / (Slash) und *0* (Null-Byte). Verschiedene Zeichen tragen jedoch eine Sonderbedeutung bei der Kommandoeingabe (Shell) und eignen sich daher nur bedingt für Dateinamen.

  Dateinamen sind in älteren *UNIX* Versionen auf eine Länge von 14 Zeichen begrenzt, in neueren Versionen auf 256 (genauer: 255 + Null-Byte) Zeichen oder mehr.

3.6 Links

→ man ln

Hardlink

```
ln name anothername
```

Die bisher über den Namen *name* erreichbaren Daten sind jetzt auch über den Namen *anothername* erreichbar. Beim Zugriff besteht zwischen *name* und *anothername* keinerlei Unterschied.

Hardlinks können nur auf *ordinary files* oder *device special files* eingerichtet werden. Beide Objekte, *name* und *anothername,* müssen sich im gleichen File System befinden. Der Eintrag *name* muß bereits existieren.

Softlink / Symbolic Link

```
ln -s name anothername
```

Das bisher über den Namen *name* erreichbare Daten-Objekt ist jetzt auch über den Namen *anothername* erreichbar.

Falls das Daten-Objekt *name* existiert, so bestehen beim Zugriff auf diese Daten via *anothername* aus Sicht des Benutzer keinerlei Unterschiede zum Zugriff über *name* . Im Kernel laufen jedoch zusätzliche Operationen ab (für *anothername* wirde eine Inode angelegt!). Softlinks sind folglich “teurer” als Hardlinks.

Der Eintrag *name* muß nicht unbedingt existieren. Softlinks können außerdem auch auf *Directories* eingerichtet werden.

Konsequenz:

- Durch Hardlinks wird aus dem Dateibaum ein zyklenfreier gerichteter Graph.
- Durch Softlinks können auch Zyklen im File System entstehen!!!

3.7 Zugriffschutz

- Benutzergruppen
  Zur Identifikation eines Benutzers gehört der *Username* und der *Groupname* . Während der Username pro System eindeutig ist, fassen die Gruppen jeweils mehrere Benutzer (zum Beispiel alle Mitarbeiter eines bestimmten Projekts oder alle Mitarbeiter einer Abteilung) zu einer Einheit zusammen.

  Ein Benutzer kann eine eigene Gruppe bilden oder zusammen mit anderen Benutzern in einer oder in mehreren verschiedenen Gruppen eingetragen sein. Die Gruppenzuordnung jedes Benutzers läßt sich dynamisch verändern. Die jeweils aktuelle Gruppenzugehörigkeit wirkt sich auf die Zugriffsrechte des Benutzers aus.

- Benutzerklassen
  Die Zugriffsrechte für alle Objekte des File Systems sind nach drei Klassen gestaffelt und für jede Klasse individuell definierbar.

  Klasse 1: *user*
  die Rechte des Besitzers des Objekts

Klasse 2: *group*
die Rechte für Mitglieder in dieser Gruppe.

Klasse 3: *other*
die Rechte aller nicht durch die Klassen 1 und 2 erfaßten Benutzer.

Wie die Benutzer des Systems tragen auch alle Files einen Username und einen Groupname. Der Username definiert den Besitzer. Der Groupname bestimmt die *group* -Klasse für den Zugriffsschutz.

- Individueller Schutz
  Der Schutzmechanismus in *UNIX* ermöglicht das individuelle Festlegen von Zugriffsrechten für jedes Objekt im Dateibaum. Der Besitzer kann bei jedem Objekt die Rechte für jede der drei Benutzerklassen separat festlegen: Kommando *chmod*

- Zugriffsrechte

  r read
  Berechtigt zum Lesen einer Datei oder eines Katalogs.

  w write
  Berechtigt zum Verändern einer Datei oder eines Katalogs.
  Bei Katalogen erlaubt dies primär das Hinzufügen von neuen Einträgen. Details zum Löschen von Einträgen

  x execute
  Berechtigt zum Ausführen einer Datei oder zum Zugriff (Kommando *cd*) auf einen Katalog.

- Konsequenzen

  Dieser Mechanismus kann sowohl einzelne Dateien als auch ganze Teilbäume vor dem Zugriff einiger, vieler oder aller Benutzer schützen. Beispiel: */etc/passwd*

In früheren *UNIX* Versionen genügte bereits das Schreib-Recht für einen Katalog, um alle darin befindlichen Dateien löschen zu können. In neueren *UNIX* Versionen kann der Besitzer des Katalogs verlangen, daß zusätzlich mindestens eine der folgenden Bedingungen erfüllt ist, bevor ein Benutzer eine Datei aus dem Katalog löschen kann:

– Benutzer ist Besitzer der Datei
– Benutzer ist Besitzer des Katalogs
– Benutzer hat Schreib-Recht auf die Datei
– Benutzer ist Super-User

Will der Katalogbesitzer diese Sicherheitsmaßnahme aktivieren, muß er das Kommando **chmod +t** auf seinen Katalog anwenden.

Der *Super-User* ist ein besonders ausgezeichneter Benutzer (Username *root* , id=0), bei dem sämtliche Mechanismen des Zugriffsschutz **nicht** greifen.

VORSICHT bei Operationen als Super-User!

3.8 Inode

Die Datenstruktur *Inode* ist die interne Repräsentation einer Datei. Die *Inode Number* ist ein Index in die *ilist* und ist die **eindeutige** Identifikation einer Datei, **nicht** der Dateiname.

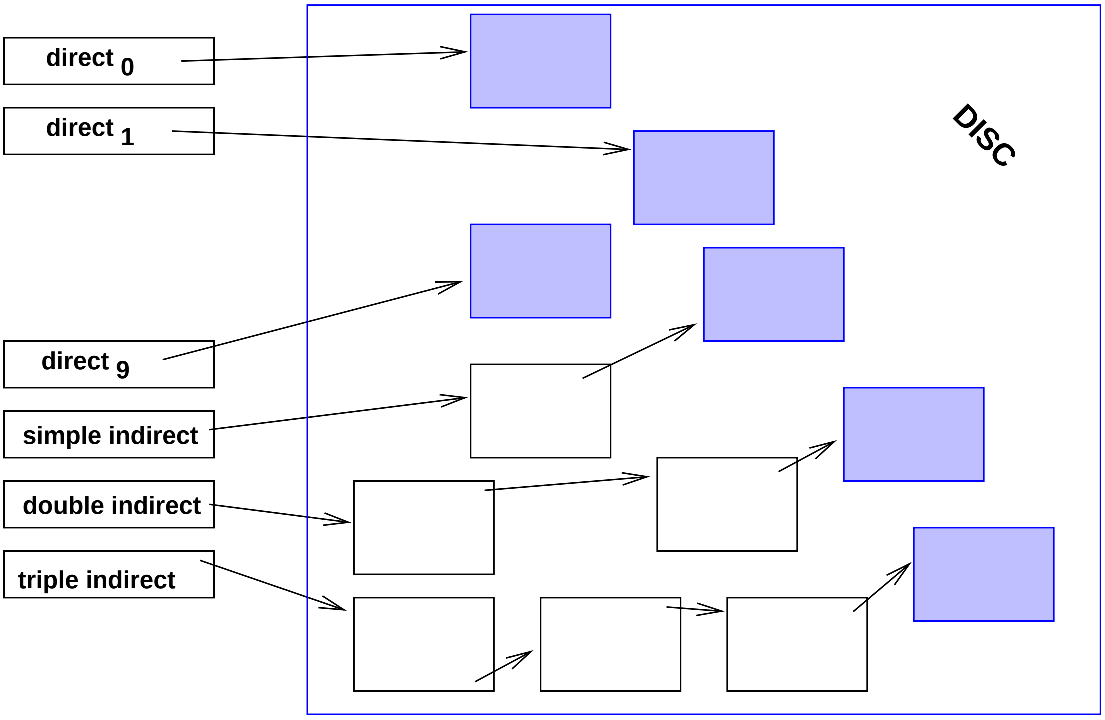
In der *Inode* enthalten sind sämtliche Informationen, die eine Datei beschreiben (siehe z.B. Kommando **ls**):

owner swg
group staff
type regular file
perms rw-r--r--
accessed Aug 28 1999 9.30 P.M.
modified Aug 29 1999 6.00 A.M
inode Aug 29 1999 1.15 P.M.
link count 2
size 600 Bytes
disk addresses

- "accessed" - wann wurde der Inhalt zuletzt gelesen?
- "modified" - wann wurde der Inhalt der Datei zuletzt verändert?
- "inode" - wann wurde die Inode zuletzt verändert?

Blockadressen

Herstellen der logischen Sicht (Byte-Folge) aus der "Menge" der jeweiligen Plattenblöcke:



Annahme: Blockgröße 1kB, Block adressierbar mit 4 Byte (32 Bit), d.h. ein Block kann 256 Adressen aufnehmen

Dies ergibt:

$10 * 1kB + 1 * 256 * 1kB + 1 * 256^2 * 1kB + 1 * 256_3 * 1kB =$

$10kB + 2^8 kB + 2^{16} kB + 2^{24} kB = 16.843.018kB = 16.843MB = 16GByte$

Die eigentliche Beschränkung liegt allerdings in der Angabe der Dateigröße: mit 32 Bit läßt sich eine Angabe von max. 4GB realisieren!

Abhängig vom aktuellen Speicherplatz gibt es diese Datenstruktur in zwei verschiedenen Ausprägungen:

- Inode als Vektorelement in der Ilist eines File Systems.
- Inode als Element der Kernel Inode Tabelle im Hauptspeicher.

Beide Ausprägungen sind funktional identisch, nur im Aufbau etwas unterschiedlich.

```
/* sys/ino.h: Inode structure as it appears on a disk block. */
struct dinode {
   ushort   di_mode;        /* mode and type of file */
   short    di_nlink;       /* number of links to file */
   ushort   di_uid;         /* owner's user id */
   ushort   di_gid;         /* owner's group id */
   off_t    di_size;        /* number of bytes in file */
   char     di_addr[39];    /* disk block addresses */
   char     di_gen;         /* file generation number */
   time_t   di_atime;       /* time last accessed */
   time_t   di_mtime;       /* time last modified */
   time_t   di_ctime;       /* time of last file status change */
};
```

In-core Inode

Zusätzliche Einträge:

- Status
  - – Inode gesperrt
  - – Prozess wartet auf Freigabe
  - – Inhalt ist anders als bei *disk inode*
  - – Datei ist ein *mount point*
- Logische Geräte-Nummer
- Eigene *inode number*
- Referenz-Zähler - gibt die Zahl der aktiven Instanzen dieser Datei an (siehe *file tables*)

Teile der Kernel Inode Tabelle sind für Benutzer zugänglich:

```
/* sys/stat.h: stat structure, used by stat(2) and fstat(2) */
struct stat {
   dev_t     st_dev;    /* device number */
   ino_t     st_ino;    /* inode number */
   mode_t    st_mode;   /* mode and type of file */
   nlink_t   st_nlink;  /* number of links to file */
   uid_t     st_uid;    /* user ID of the file's owner   */
   gid_t     st_gid;    /* group ID of   the file's group */
   dev_t     st_rdev;   /* ID of (raw)device if block/char special */
   off_t     st_size;   /* file size in bytes */
   time_t    st_atime;  /* time of last access   */
   time_t    st_mtime;  /* time of last data modification */
   time_t    st_ctime;  /* time of last file status change */
};
```

(jeweils aktuell unter /usr/include/sys/stat.h)

Auslesen der Inode-Informationen: stat und fstat

```
/* Datei-Typ und Zeitstempel ausgeben - Namen aus Kommandozeile */

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>

int main(int argc, char *argv[]) {
   int i;
   struct stat statbuffer; /*aus stat.h -> spaeter mehr*/
   char * p;

   for(i=1; i<argc; i++) {
      printf("%s:\n", argv[i]);
         /* System Call stat */
      if (stat(argv[i], &statbuffer) < 0) {
         perror("stat error");
         exit(1);
      }

      printf("\tlast access:       %s\n",
             ctime(&statbuffer.st_atime) );
      printf("\tlast modification: %s\n",
             ctime(&statbuffer.st_mtime) );
      printf("\tlast change:       %s\n",
             ctime(&statbuffer.st_ctime) );

      switch (statbuffer.st_mode & S_IFMT) {
         case S_IFDIR: p = "directory"; break;
         case S_IFCHR: p = "char special"; break;
         case S_IFBLK: p = "block special"; break;
         case S_IFREG: p = "regular"; break;
         case S_IFLNK: p = "sym link"; break ;
         case S_IFSOCK: p = "socket"; break;
         case S_IFIFO: p = "fifo"; break;
         default: p = "UNKNOWN"; break;
      }
      printf("\ttype:              %s\n", p);
   }
   exit(0);
}
```

```
byron$ gcc -Wall -o ftype.exe ftype.c
byron$ ftype.exe ftype.c ftype.exe .. .
ftype.c:
        last access:       Thu Dec  9 18:39:08 1999

        last modification: Thu Dec  9 18:37:05 1999
```

```
        last change:       Thu Dec  9 18:37:05 1999

        type:              regular
ftype.exe:
        last access:       Thu Dec  9 18:39:28 1999

        last modification: Thu Dec  9 18:39:08 1999

        last change:       Thu Dec  9 18:39:08 1999

        type:              regular
..:
        last access:       Thu Dec  9 18:38:50 1999

        last modification: Thu Dec  9 18:05:09 1999

        last change:       Thu Dec  9 18:05:09 1999

        type:              directory
.:
        last access:       Thu Dec  9 18:38:36 1999

        last modification: Thu Dec  9 18:39:08 1999

        last change:       Thu Dec  9 18:39:08 1999

        type:              directory
byron$
```

Inode <-> Links

Jedes File wird durch *genau eine Inode* repräsentiert, es kann aber *mehrere Dateinamen* besitzen.

Ein “Dateiname” ist ein Verweis, genannt *Link* , aus einem Directory heraus auf einen Inode.

Wieviele Directory-Einträge auf denselben Inode verweisen, spiegelt sich als *link count* im Inode-Feld *nlink* wieder.

```
hypatia$ pwd
/home/swg/soft/soft1/9/progs
hypatia$  ls -al
total 11
drwxr-xr-x   2 swg      users        1024 Dec 11 07:46 .
drwxr-xr-x   5 swg      users        1024 Dec 11 07:41 ..
-rw-r--r--   1 swg      users        1190 Dec 11 06:59 ftype.c
-rwxr-xr-x   1 swg      users        4921 Dec 11 06:59 ftype.exe
-rw-r--r--   1 swg      users         143 Dec 11 06:59 ftype.out
-rw-r--r--   1 swg      users         780 Dec 11 06:59 ftype.se
-rw-r--r--   1 swg      users           0 Dec 11 07:46 typescript
hypatia$ ls -al ..
total 6
drwxr-xr-x   5 swg      users        1024 Dec 11 07:41 .
drwxr-xr-x  21 swg      users        2048 Dec 11 07:45 ..
drwxr-xr-x   2 swg      users        1024 Dec 11 07:35 pics
drwxr-xr-x   2 swg      users        1024 Dec 11 07:46 progs
drwxr-xr-x   2 swg      users        1024 Dec 11 07:44 src
hypatia$
```

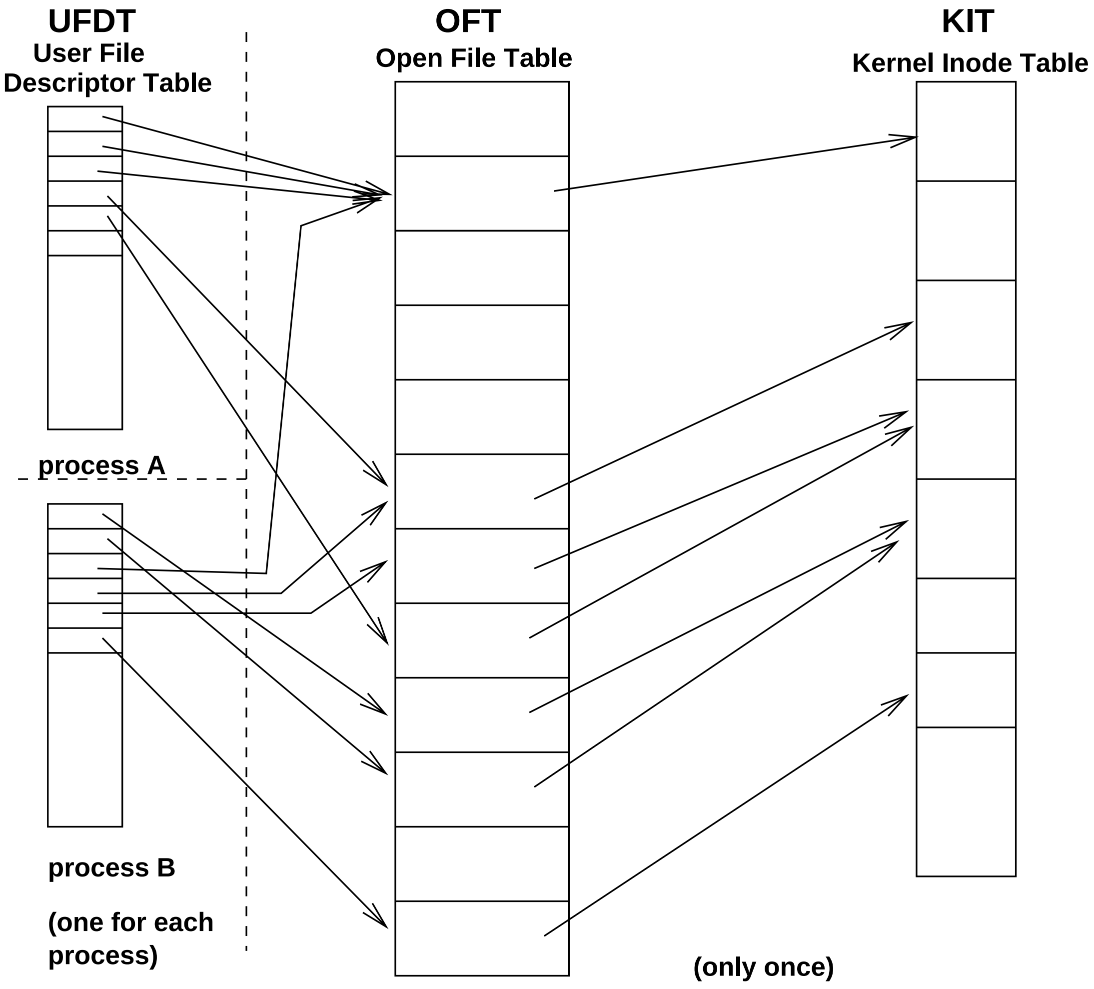
Links können aus verschiedenen Directories heraus auf ein und denselben Inode installiert sein. Die Directories müssen jedoch alle zum selben File System gehören, da Directory-Einträge nur die *inum* enthalten (und keine Identifikation des File Systems).

Eine Ausnahme dazu stellen die *symbolischen* Links dar (BSD 4.3 und System V ab Rel 4.0). Für einen symbolischen Link legt das I/O Subsystem einen zweiten Inode an und speichert den Pfadnamen des Objekts ab, auf das der Link “zeigt”. Der *link count* im Inode des Objekts wird nicht erhöht. Das Objekt muß nicht einmal existieren, wenn der symbolische Link installiert wird.

3.9 Datenstrukturen für I/O-Verbindungen

Vom Prozeß zum File via Kernel:

Der Kernel verwaltet in seinem Bereich des Hauptspeichers eine Reihe von Datenstrukturen, die die System Call Serviceroutinen beim Zugriff auf Files benutzen.



Die *User File Descriptor Table* identifiziert alle offenen I/O-Verbindungen eines Prozesses. Der Kernel legt diese Tabelle für jeden Prozeß aber im Kernel-Adreßraum an. Sie gehört zum *Context* des Prozesses. Jeder File Deskriptor eines Prozesses ist Index in seine *UFDT.* Als einzige Komponente enthalten die Slots der *UFDT* einen Zeiger in die Open File Table des Kernels.

Die *Open File Table* existiert nur einmal im Adreßraum des Kernels. Alle Slots der *OFT* enthalten

- einen Zeiger in die Kernel Inode Tabelle,
- die **oflag-Bits,**
- einen *File Offset Pointer*
- und einen Referenz-Zähler.

Die oflag-Bits stammen aus dem *open (2)* Aufruf. Der File Offset Pointer legt fest, bei welchem Byte die nächste *read (2)* oder *write (2)* Operation stattfindet. Der Referenz-Zähler zählt mit, wie viele *UFDT* Slots auf diesen *OFT* Slot zeigen.

Die *Kernel Inode Table* existiert nur einmal. Der Kernel koordiniert daüber sämtliche Zugriffe auf Files. Die *in-core* Version der Inodes enthält alle Komponenten der Plattenversion plus Informationen, die nur bei offenen I/O-Verbindungen von Bedeutung sind: ein Referenz-Zähler, die Device Nummer,...

Jede offene Datei belegt genau einen Slot, unabhängig wie oft und von wievielen Prozessen sie geöffnet wurde.

3.10 File Deskriptoren

- eine kleine Integer-Zahl als Index in die UFDT (Maximum: NOFILE aus /usr/include/sys/param.h)
- 0 (stdin), 1 (stdout), 2 (stderr) - Konvention

3.11 Zugriffsrechte

Prozeß haben vier ID's:

- real user ID
- real group ID
- effective user ID
- effective group ID

Einige Datei-Attribute:

- owner’s user ID (16-Bit Integer)
- owner’s group ID (16-Bit Integer)
- user-read permission (1-Bit Flag)
- user-write permission (1-Bit Flag)
- user-execute permission (1-Bit Flag)
- group-read permission (1-Bit Flag)
- group-write permission (1-Bit Flag)
- group-execute permission (1-Bit Flag)
- other-read permission (1-Bit Flag)
- other-write permission (1-Bit Flag)
- other-execute permission (1-Bit Flag)
- set-user-ID (1-Bit Flag)
- set-group-ID (1-Bit Flag)

Zugriff eines Prozesses auf Datei:

1) Ist die effektive User-Id 0 (superuser), so ist Zugriff erlaubt.

2) Stimmt die effektive User-ID des Prozesses mit dem Besitzer der Datei überein und sind die Zugriffsrechte entsprechend dem Zugriff (read, write) richtig gesetzt, so ist Zugriff möglich.

3) Stimmt die effektive User-ID des Prozesses nicht mit dem Besitzer der Datei überein
und
stimmt die effektive Group-ID des Prozesses mit der Group-Id des Besitzers der Datei überein
und
und stimmen Zugriffswunsch und Zugriffsrechte überein,
so ist der Zugriff erlaubt.

4) Stimmt die effektive User-ID des Prozesses nicht mit dem Besitzer der Datei überein
und
stimmt die effektive Group-ID des Prozesses nicht mit der Group-Id des Besitzers der Datei überein
und

und stimmen Zugriffswunsch und Zugriffsrechte für "others" überein,
so ist der Zugriff erlaubt.

3.12 File Access Mode Word

Darstellung obiger Dateiattribute (*file acces mode word*):

Oktalwert	Bedeutung
04000	set user ID on execution
02000	set group ID on execution
01000	save text image after execution ('sticky bit')
00400	Read by user
00200	Write by user
00100	Execute by user
00040	Read by group
00020	Write by group
00010	Execute by group
00004	Read by other
00002	Write by other
00001	Execute by other

Herausfinden

- der User-Rechte: Maske 0700 (Dual: 000...000 111 000 000)
- der Group-Rechte: Maske 070 (Dual: 000...000 111 000)
- der Other-Rechte: Maske 07 (Dual: 000...000 111)

sticky bit: ist das Programm ausführbar als *read-only*, so verbleibt eine Kopie des Textes im Swap-Bereich (nach Beendigung des Programms) und kann beim nächsten Start schneller geladen werden.

3.13 File Mode Creation Mask

Jeder Prozeß hat eine "Maske" zum Anlegen der Rechte neuer Dateien, die mit dem *umask* System Call gesetzt wird:

```
NAME
       umask - set file creation mask

SYNOPSIS
       #include <sys/stat.h>

       int umask(int mask);

DESCRIPTION
       umask sets the umask to mask & 0777.

       The umask is used by open(2) to set initial file permissions
       on a newly-created file.  Specifically,  permissions
       in  the umask are turned off from 0666 (so, for example,
       the common umask default value of 022 results in new files
       being  created  with  permissions
       0666 & ~022 = 0755 = rw-r--r--).

RETURN VALUE
       This system call always succeeds and the previous value of
       the mask is returned.

CONFORMING TO
       SVr4, SVID, POSIX, X/OPEN, BSD 4.3

SEE ALSO
       creat(2), open(2)
```

3.14 Return-Wert von System Calls

Grundregel:

Jeder System Call zeigt mit seinem Return-Wert an, ob die Ausführung erfolgreich war oder ob eine Fehlersituation aufgetreten ist. Ein stabil codiertes Programm überprüft den Return-Wert bei jedem System Call.

Beispiel

```
if( ( fd = creat(path, 0444) ) < 0 )
    perror( path),  exit(1);
```

Wenn der System Call *creat()* (Erzeugen einer Datei - alte Version, siehe *open*) bei der Ausführung im Kernel Mode in irgendwelche Probleme läuft, liefert er *-1* als Ergebnis.

Frage:

Auf welches Problem traf *creat()?*

Woher weiß *perror (),* welches Problem aufgetreten ist?

errno:

Tritt bei der Ausführung eines System Calls ein Fehler auf, liefert der Aufruf einen eigentlich unmöglichen Wert zurück. Dies ist meist *-1* oder ein *NULL -Zeiger.* Die Details beschreibt die Definition der einzelnen System Calls im Kapitel 2 der *Reference Manuals.*

Neben diesem Fehlerindikator macht der Kernel in der globalen Variable *errno* eine Fehlernummer verfügbar. Mit Hilfe dieser Fehlernummer kann *perror()* oder eine selber geschriebene Funktion eine entsprechende Fehlermeldung produzieren.

```
void myperror(char * sys_call) {
  extern int errno;
  fprintf(stderr, "Error %d occurred during system call %s\n",
            errno, sys_call);
}
```

Für aussagekräftigere Fehlermeldungen kann man *errno* als Index in den globalen Vektor *sys_errlist* mit den Standard-Fehlermeldungen benutzen, siehe dazu auch *perror.*

```
hypatia$ cat myerror.c

/* myerror.c */

# include <stdio.h>
# include <errno.h>
# include <fcntl.h>

extern int errno;
extern char *sys_errlist[];
extern int sys_nerr;

void myperror2(char * sys_call) {

  if ( 0 <= errno  &&  errno < sys_nerr ){
    fprintf(stderr, "ERROR: ");
    fprintf(stderr, "%s\n", sys_errlist[ errno ]);
  } else
    fprintf(stderr, "Error %d occurred during system call\n", errno);
}

int main() {
   int fd;
   if( (fd = open("xxx", O_WRONLY)) < 0)
        myperror2("open"), exit(1);
   exit(0);
}

hypatia$ ls -l xxx
-r--r--r--   1 swg      users           0 Dec 12 19:16 xxx

hypatia$ gcc -Wall myerror.c

hypatia$ a.out
ERROR: Permission denied

hypatia$ echo $?
1
hypatia$
```

Die Datei */usr/include/errno.h* definiert symbolische Konstanten für alle vom Kernel benutzten Fehlernummern.

errno	#define	error message
1	**EPERM**	No permission
2	**ENOENT**	No such file or directory
3	**ESRCH**	No such process
4	**EINTR**	interrupted system call
...		
19	**ENODEV**	No such device
20	**ENOTDIR**	Not a directory
...		
23	**ENFILE**	File table overflow
24	**EMFILE**	Too many open files
25	**ENOTTY**	Not a typewriter
...		
64	**ENONET**	Machine is not on the network
65	**ENOPKG**	Package not installed
...		

Anwendung von errno:

Wird ein "langsamer" System Call (z.B. *read(2))* durch ein Interrupt-Signal unterbrochen, so liefert er *-1* als Fehlerindikator zurück, setzt aber *errno* auf den Wert *EINTR* . Dies bedeutet, daß eigentlich **kein** Fehler aufgetreten ist, der System Call nur "gestört" worden war. Das Programm kann nun entscheiden, ob es den System Call erneut aufsetzt oder auf die Unterbrechung hin anders weiterarbeitet.

```
extern int errno;

        ...

do {
    errno = 0;           /* clear before system call */

    nread = read( fd, buf, BUFSIZ ); /* slow system call */

} while ( nread < 0  &&  errno == EINTR );

if ( nread < 0 )        /* there was a real error */
```

```
  perror( "read" ),   exit( 1 );
```

3.15 I/O-Verbindungen

Auf- / Abbau

- Syntax der System Calls: *open (2),* und *close (2)*

```
#include <fcntl.h>


int open ( /* open file */
                char *path,      /* path name */
                int oflag        /* option flag, r/w */
         );
/* returns file descriptor or -1 on error */


int close ( /* close file */
                int fd           /* file descriptor */
          );
/* returns 0 on success or -1 on error */
```

- Beschreibung

open ():

Der System Call *open (2)* stellt eine I/O-Verbindung her zwischen einem existierenden File und dem ausführenden Prozeß.

open (2) kennt drei Arten von I/O-Verbindungen:

	oflag	Prozeß kann das File
0	**O_RDONLY**	nur lesen
1	**O_WRONLY**	nur schreiben
2	**O_RDWR**	lesen und schreiben

Für **oflag** enthält *<fcntl.h>* die Definitionen der symbolischen Konstanten.

Der System Call *open (2)* schlägt fehl, wenn der Prozeß die nötigen Rechte für **path** nicht besitzt oder **path** nicht existiert.

Eine erfolgreiche Ausführung des System Calls liefert einen *File Descriptor* zurück. Dies ist eine kleine, positive Zahl, die bei allen nun folgenden I/O-Operationen (*read (2), write (2),..., close (2))* diese I/O-Verbindung identifiziert. Sie stellt einen Index in eine entsprechende Tabelle dar. Für *stdin* ist der Filedeskriptor 0, für *stdout* 1 und für *stderr* 2. Diese sind beim Start eines Prozesses typischerweise automatisch geöffnet!

close ():

Der System Call *close (2)* beendet eine I/O-Verbindung, die durch einen File Descriptor repräsentiert ist.

Da der Kernel die Zahl der I/O-Verbindungen pro Prozeß limitiert (*NOFILE* aus *<sys/param.h>),* ist es ratsam, nicht mehr gebrauchte I/O-Verbindungen mit *close (2)* abzuschließen.

Terminiert ein Prozeß so führt der Kernel einen impliziten *close (2)* System Call für alle noch offenen I/O-Verbindungen aus.

Manipulation der I/O-Datenstrukturen

Manipulation nur per System Call:

Um die Integrität der I/O-Datenstrukturen zu gewährleisten, behält der Kernel sämtliche Tabellen in seinem Adreßraum. Prozesse können deshalb nur durch System Calls auf diese Tabellen einwirken.

UFDT	User File Descriptor Table
OFT	Open File Table
KIT	Kernel Inode Table

- System Call *open ()*
  open (2) belegt immer je einen Slot in der *UFDT* und der *OFT* . Der Referenz-Zähler im *OFT* Slot wird mit 1 initialisiert. Bestand keine Verbindung zu der Datei, belegt *open (2)* auch einen Slot in der *KIT* und initialisiert diesen neuen in-core Inode. Besteht bereits eine Verbindung zu der Datei (vom selbem oder einem anderen Prozeß), wird der schon für die Datei reservierte *KIT* Slot verwendet. Der Kernel installiert nur den Zeiger vom *OFT* Slot auf den *KIT* Slot und

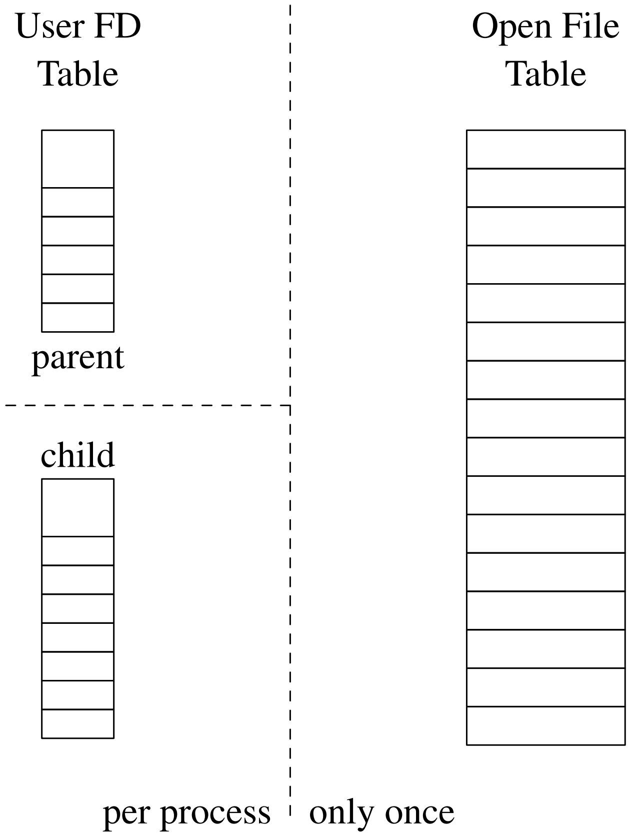
inkrementiert den Referenz-Zähler in diesem in-core Inode.

Folge: mehrere *OFT* Slots können auf den selben *KIT* Slot zeigen. Der Kernel hält von jedem Platten-Inode maximal eine Kopie *in core* .

- System Call *close ()*
  close (2) gibt einen Slot in der *UFDT* frei und dekrementiert den Referenz-Zähler im *OFT* Slot. Wird dieser Referenz-Zähler dadurch zu 0, kann auch der *OFT* Slot freigegeben und der Referenz-Zähler im *KIT* Slot dekrementiert werden. Wird nun dieser Referenz-Zähler zu 0, gibt der Kernel auch den *KIT* Slot frei. Sollte im Inode der Link Count 0 sein, kann der Kernel jetzt auch den Platten-Inode und Plattenplatz freigeben.
- System Call *dup ()*
  dup (2) belegt den niedersten, noch freien Slot in der *UFDT* , kopiert den Inhalt eines bereits belegten Slots dorthin und inkrementiert den Referenz-Zähler in dem *OFT* Slot, auf den nun beide *UFDT* Slots zeigen.

  Folge: mehrere *UFDT* Slots eines Prozesses können auf den selben *OFT* Slot zeigen.
- System Call *fork ()*
  fork (2) erzeugt einen neuen Prozeß indem er den kompletten Kontext des ausführenden Prozesses verdoppelt. Dabei muß der Kernel u.a. sowohl den *UFDT* Vektor kopieren, als auch die Referenz-Zähler in den entsprechenden *OFT* Slots inkrementieren.

  Folge: mehrere *UFDT* Slots aus verschiedenen aber verwandten Prozessen können auf den selben *OFT* Slot zeigen (Vererbung). Diese Prozesse können sich dadurch einen File Offset Pointer teilen.
- Kerneltabellen



- System Call Aufrufe

Parent Prozeß | Child Prozeß

fd1 = open("file1", O_RDONLY);
UFDT Slot belegen,
OFT Slot belegen, Refz = 1;
KIT Slot belegen, Refz = 1;

fd2 = open("file2", O_RDWR);

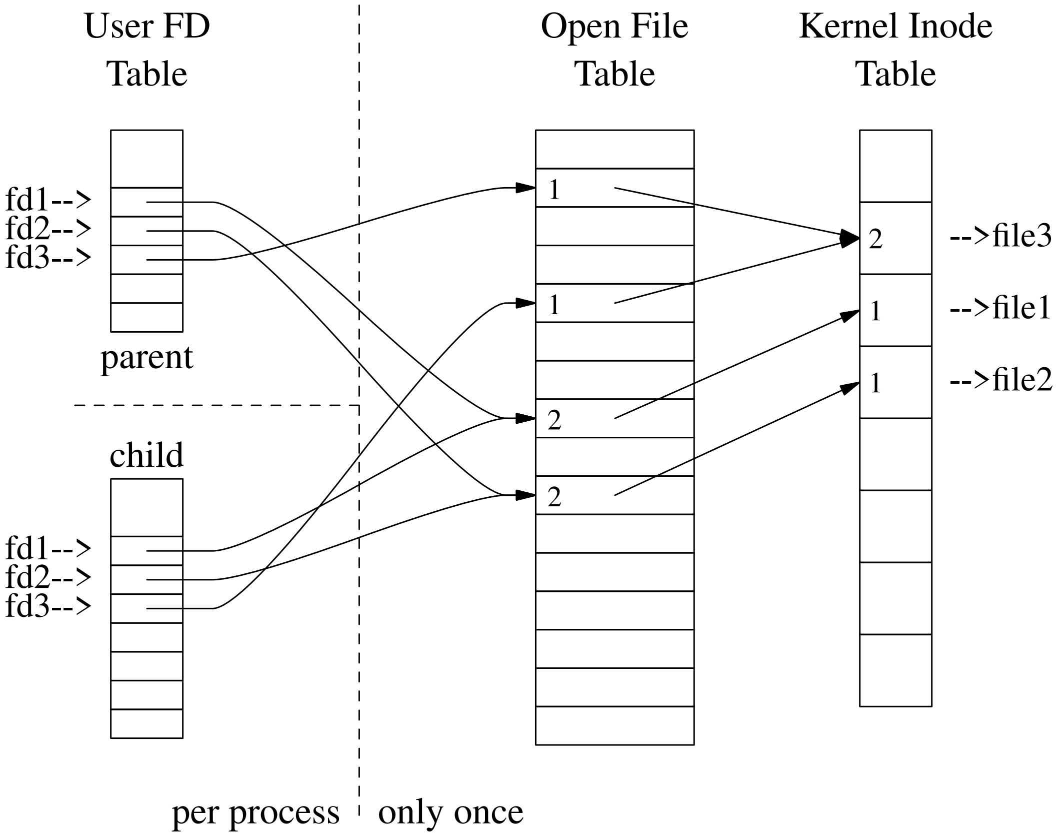
fork();
UFDT Vektor kopieren,
in allen betroffenen *OFT* Slots ++Refz;
die *KIT* Slots bleiben unverändert

fd3 = open("file3", O_RDONLY); (Child Prozeß)

fd3 = open("file3", O_WRONLY);

UFDT Slot belegen,
OFT Slot belegen, Refz = 1;
bereits belegten *KIT* Slot benutzen,
++Refz;

- Kerneltabellen



- System Call Aufrufe

Parent Prozeß | Child Prozeß

fd4 = open("file4", O_RDONLY);

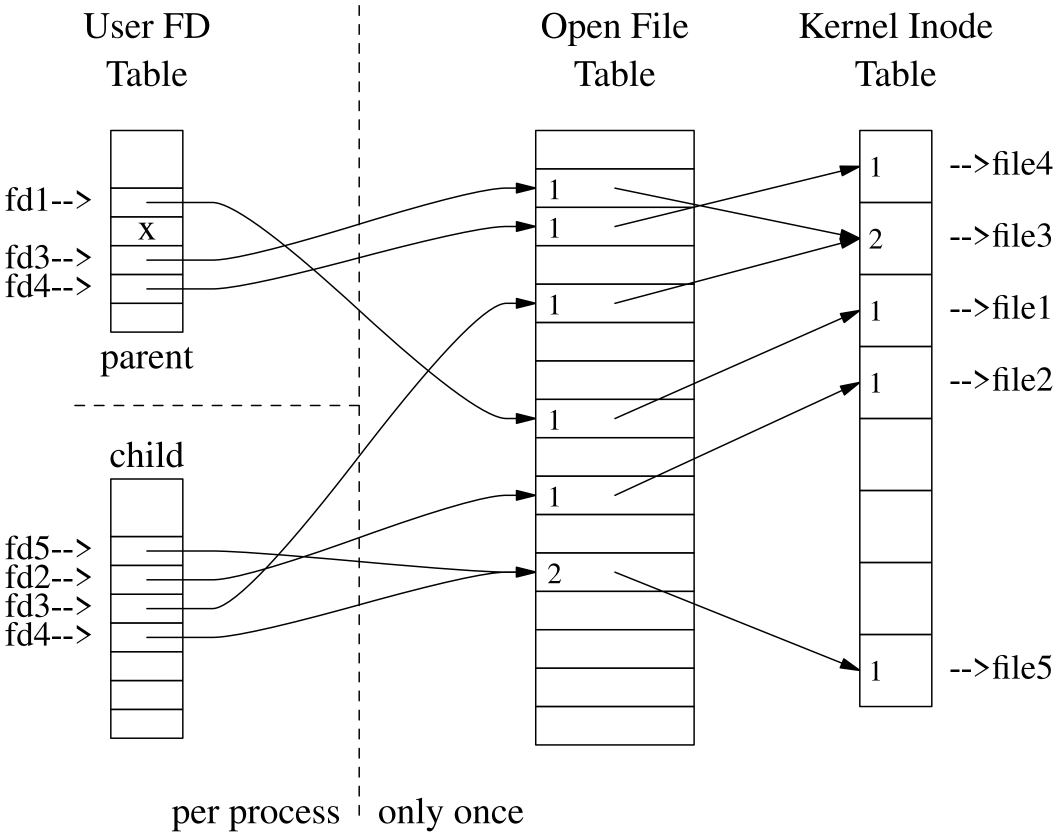
fd4 = open("file5", O_RDONLY);

close(fd1);
UFDT Slot freigeben,
im *OFT* Slot --Refz; Refz > 0 ==>
==> *KIT* Slot bleibt unverändert

fd5 = dup(fd4);
freien *UFDT* Slot belegen,
im bereits belegten *OFT* Slot ++Refz;
KIT Slot bleibt unverändert

close(fd2);
UFDT Slot freigeben,
im *OFT* Slot --Refz; Refz > 0 ==>
==> *KIT* Slot bleibt unverändert

- Kerneltabellen



- System Call Aufrufe

Parent Prozeß Child Prozeß

close(fd4);

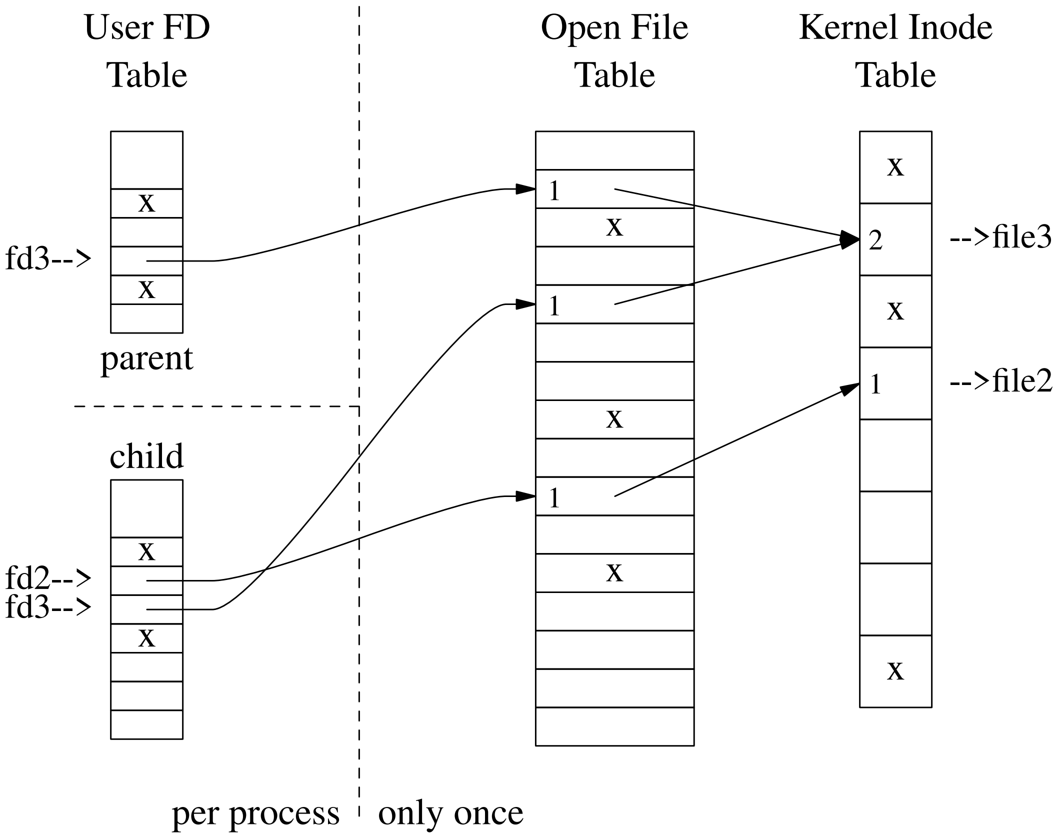
UFDT Slot freigeben,
im *OFT* Slot --Refz; Refz > 0 ==>
==> *KIT* Slot bleibt unverändert

close(fd5);
UFDT Slot freigeben,
im *OFT* Slot --Refz; Refz == 0 ==>
==> *OFT* Slot freigeben und
im *KIT* Slot --Refz; Refz == 0 ==>
auch *KIT* Slot freigeben (Link Count?)

close(fd1);
UFDT Slot freigeben,
OFT Slot --Refz == 0 ==> freigeben
KIT Slot --Refz == 0 ==> freigeben

close(fd4);

- Kerneltabellen



- System Call Aufrufe

Parent Prozeß	Child Prozeß
	close(fd2); *UFDT* Slot freigeben, *OFT* Slot - -Refz == 0 ==> freigeben *KIT* Slot - -Refz == 0 ==> freigeben
close(fd3); *UFDT* Slot freigeben, im *OFT* Slot - -Refz; Refz == 0 ==> ==> *OFT* Slot freigeben und im *KIT* Slot - -Refz; Refz > 0 ==> *KIT* Slot noch nicht freigeben	
	close(fd3); *UFDT* Slot freigeben, *OFT* Slot - -Refz == 0 ==> freigeben *KIT* Slot - -Refz == 0 ==> freigeben

3.16 System Call “open”

- Syntax

```
#include <fcntl.h>

int open ( /* open file with three args */
                char *path,      /* path name */
                int oflags,      /* option flags */
                int perms        /* permission bits */
         );
        /* returns file descriptor or -1 on error */
```

- Beschreibung

Die neuere Form des System Calls *open (2)* hat 2 oder 3 Argumente. Sie ist komplett kompatibel zu der älteren Form mit 2 Argumenten.

Der C-Compiler *cc* erlaubt variabel viele Argumente bei Funktionsaufrufen. Die gerufene Funktion muß wissen, wieviele Argumente gültige Werte enthalten. *open (2)* entscheidet anhand des 2. Arguments, ob das 3. Argument genutzt wird.

Die **oflags** können nicht nur die r/w Werte *O_RDONLY* , *O_WRONLY* oder *O_RDWR* annehmen, sondern differenziertere Flagwerte. Diese entstehen durch OR-en von einem oder mehreren der folgenden Flagwerte zu dem r/w Flag:

`oflag`	Bedeutung
`O_CREAT`	Falls das File nicht existiert, wird es angelegt. Vgl. *creat (2).* Nur bei diesem Flagwert findet das 3. Argument Anwendung.
`O_TRUNC`	Falls das File existiert, wird seine Länge auf 0 verkürzt — der alte Inhalt gelöscht.
`O_APPEND`	Dieses Flag stellt sicher, daß jede Write-Operation am Ende der Datei erfolgt — auch bei konkurrierenden Write-Operationen verschiedener Prozesse.
`O_EXCL`	Zusammen mit dem *O_CREAT -Flag* stellt dieses Flag sicher, daß *open (2)* fehlschlägt, wenn **path** bereits existiert.
`O_NONBLOCK` **`O_NDELAY`**	Prozeß blockiert nicht beim Öffnen eines noch nicht bereiten IPC-Kanals. Zusätzlich wirkt dieses Flag auf nachfolgende *read (2)* und *write (2)* System Calls.
`O_SYNC`	Bei jedem *write (2)* auf dieses File blockiert der Prozeß bis die Write-Operation *physikalisch* abgeschlossen ist.

Beliebige Kombinationen sind möglich, aber nicht alle sinnvoll!

- Beispiele

creat (2)-Aufgabe: Datei zum Schreiben öffnen. Falls Datei noch nicht existiert, anlegen. Falls Datei bereits existiert, auf Länge 0 bringen (alten Inhalt löschen).

```
open( path, O_WRONLY | O_CREAT | O_TRUNC, perms )
```

“Harmloseres” *creat :* Datei zum Schreiben öffnen. Falls die Datei noch nicht existiert, anlegen. Auf keinen Fall Daten löschen.

```
open( path, O_WRONLY | O_CREAT, perms )
```

Datei zum Schreiben und Lesen öffnen. Falls Datei noch nicht existiert, anlegen. Mit *creat (2)* und *open (2)* (alte Form):

```
if ( (fd = creat( path, perms )) < 0 )
        sysfatal( path, "can’t creat \"%s\"", path );

if ( close( fd ) < 0  ||  (fd = open( path, O_RDWR )) < 0 )
        sysfatal( path, "can’t reopen \"%s\"", path );
```

Soll außerdem ein alter Inhalt erhalten bleiben, müßten einige weitere System Calls ausgeführt werden. Die neue Form von *open (2)* erlaubt dagegen eine differenzierte Anforderung der Kernel-Dienstleitstung in nur einem System Call:

```
if ( (fd = open( path, O_RDWR | O_CREAT, perms )) < 0 )
        sysfatal( path, "can’t get r/w access to \"%s\"", path );
```

- Bemerkung

In der neuen Form umfaßt *open (2)* alle Funktionen der älteren Form von *open (2)* plus die Funktion des System Calls *creat (2).* Durch die neuen Flagwerte ist aber eine wesentlich differenziertere Anwendung der einzelnen Funktionen möglich.

One may well ask why creat is still in UNIX as a system call. (...) The answer is probably emotional; creat has been a system call since the very beginning, and has evidently been granted tenure.

[Marc J. Rochkind, Seite 26]

3.17 System Calls “link” und “unlink”

Mit *unlink (2)* entfernt man Dateien wieder aus dem Directory:

```
int unlink ( /* unlink file */
                char *path      /* path name */
           );
/* returns 0 on success or -1 on error */
```

Besteht bereits ein Verweis aus einem Directory heraus auf einen Inode (kein Directory !), so kann man mit *link (2)* einen weiteren Verweis (*hardlink)* innerhalb des selben File Systems auf diesen Inode installieren.

```
int link ( /* link to file */
                char *oldpath,          /* existing file */
                char *newpath           /* new directory entry */
         );
/* returns 0 on success or -1 on error */
```

Der System Call *link (2)* incrementiert den *link count* im Inode um den Wert 1, *unlink (2)* decrementiert ihn um 1.

Einen *symbolischen* Link installiert man mit folgendem System Call:

```
int symlink (   /* symbolic link to file */
                char *oldpath,          /* existing file */
                char *newpath           /* new directory entry */
            );
/* returns 0 on success or -1 on error */
```

Symbolische Links bestehen lediglich aus einem Pfadnamen. Sie können sowohl über File System Grenzen hinweg, als auch auf Directories, als auch auf nicht existente Objekte installiert werden. Beim *open (2)* unterscheiden sie sich aus Benutzersicht durch nichts von Hardlinks — im Kernel aber sehr wohl.

Sämtliche Operationen unterliegen dem üblichen *UNIX* Zugriffsschutz. Nur mit den entsprechenden Rechten können Prozesse Directories lesen oder verändern.

link

- Funktion

Der System Call *link (2)* installiert einen zusätzlichen Verweis auf eine bereits existierende Datei, genauer, auf einen Inode. Dazu trägt der System Call lediglich einen Verweis in das durch den Directory-Prefix von **newpath** bestimmte Directory ein. Der Verweis besteht aus der File-Komponente von **newpath** und der gleichen *inum* wie bei dem Eintrag **oldpath .** Außerdem erhöht *link (2)* den Link Count im Inode um 1.

Directory vorher:

Dateiname	inum
`d.out`	23550
`sub`	9267
`output`	17734

Installation des Link:

link d.out x.in

Directory nachher:

Dateiname	inum
`d.out`	23550
`sub`	9267
`output`	17734
`x.in`	23550

- Bemerkungen

Links können zwar aus verschiedenen Directories auf den selben Inode installiert werden, müssen aber innerhalb des selben File Systems bleiben.

Auf Directories können keine Links installiert werden.

Das mit **oldpath** bezeichnete Objekt muß bereits existieren.

Für den Eintrag braucht der Prozeß (*euid /egid) Write-Recht auf das Directory.*

Der System Call *link (2)* **schlägt fehl,** wenn das Objekt **newpath** bereits existiert.

unlink

- Funktion

Der System Call *unlink (2)* entfernt den Eintrag **path** aus dem entsprechenden Directory und verringert den *Link Count* im zugehörigen Inode um 1. Wird der *Link Count* dadurch zu 0, kann das I/O Subsystem den durch die Datei belegten Plattenplatz und Inode freigeben.

Um einen Directory-Eintrag entfernen zu können und damit eine Datei zu löschen, braucht der Prozeß (*euid /egid) nur Write-Recht für das Directory. Die Rechte der Datei bleiben unberücksichtigt.*

Dieses potentielle Sicherheitsloch wurde in neueren *UNIX* Versionen geschlossen. Setzt der Besitzer das *sticky bit* für sein Directory, so kann ein Prozeß dort nur dann Einträge löschen, wenn zusätzlich zum Write-Recht auf das Directory noch eine der folgenden Bedingungen erfüllt ist:

euid /egid des Prozesses hat Write-Recht auf die Datei

euid des Prozesses ist Besitzer der Datei

euid des Prozesses ist Besitzer des Directory

euid des Prozesses ist 0 (Superuser).

- Undelete File ??

Freigeben des Plattenplatzes bedeutet, daß das I/O Subsystem die der Datei zugeordneten Blöcke aus dem Inode entfernt und in die *Free List* einhängt.

Bei der nächsten Anforderung für neue Blöcke können die gerade freigegebenen Blöcke bereits wiederverwendet werden. Wann dies tatsächlich geschieht, hängt von den gerade aktiven Prozessen *(multiuser / multitasking)* und vom Free List Algorithmus ab.

- Besonderheit

Das I/O Subsystem verzögert das Freigeben von Inode und Plattenplatz solange noch ein Prozeß die Datei geöffnet hat. Da jedoch der Directory-Eintrag entfernt wurde, ist ein weiterer Zugriff auf die Datei, z.B. mit *open (2)* oder *stat (2),* über den Namen nicht mehr möglich.

Die File Deskriptoren der Prozesse bilden die ‘‘letzte’’ Verbindung zu der Datei. Sie können für *read (2), write (2), fstat (2), ...* und *close (2)* benutzt werden. Nach dem letzten expliziten oder impliziten *close ()* System Call mit einem File Deskriptor, der noch auf die Datei verwiesen hat, gibt das I/O Subsystem die Daten entgültig frei.

Dieses Verhalten des Kernels vereinfacht den Gebrauch (und das ordentliche Entfernen) von **temporären Dateien**:

```
if ( (fd = creat( TMPFILE, mode )) < 0 )
        sysfatal( TMPFILE, "can’t creat temp file" );
else
        unlink( TMPFILE );
```

Der Prozeß kann die temporäre Datei benutzen, wenn er jedoch − egal aus welchem Grund − terminiert, gibt das I/O Subsystem automatisch allen Plattenplatz frei, den die Datei belegt hat. Es entstehen keine unliebsamen ‘‘Überbleibsel’’.

3.18 System Call ‘‘write’’

- Syntax

```
int write ( /* write on file */
                int fd,         /* file descriptor */
                char *buf,      /* buffer address  */
                int nbytes      /* number of bytes to write */
          );
/* returns number of bytes written or −1 on error */
```

- Beschreibung

Der System Call *write (2)* schreibt aus dem Hauptspeicher in eine I/O-Verbindung. **nbytes** bestimmt die Datenmenge, **buf** die Startadresse im Hauptspeicher und der File Deskriptor **fd** die I/O-Verbindung.

Eine *write -Operation* ist ein 1:1 Kopiervorgang. Es findet keinerlei Daten-Konvertierung oder Veränderung statt.

Führt die I/O-Verbindung zu einem *regular File* bestimmt der File Offset Pointer im *OFT* Slot das Ziel des Kopiervorgangs. Die übertragenen Bytes überschreiben bereits vorher gespeicherte Daten. Anschließend zeigt der File Offset Pointer hinter das letzte übertragene Byte.

- Achtung

Die Modellannahme, daß die Daten nach Abschluß des *write* System Calls physikalisch auf die Platte geschrieben wurden, ist nicht vollständig korrekt. Das I/O Subsystem kopiert die Daten in den Buffer Cache und gibt die Kontolle zurück, etwa mit folgender Meldung:

> *I've taken note of your request, and rest assured that your file deskriptor is OK, I've copied your data successfully, and there's enough disk space. Later, when it's convenient for me, and if I'm still alive, I'll put your data on the disk where it belongs. If I discover an error then I'll try to print something on the console, but I won't tell you about it (indeed, you may have terminated by then). If you, or any other process, tries to read this data before I've written it out, I'll give it to you from the buffer cache, so, if all goes well, you'll never be able to find out when and if I've completed your request. You may ask no further questions. Trust me. And thank me for the speedy reply.*
>
> [Marc J. Rochkind, Seite 29]

Mit dem *O_SYNC* Flag könnte man dies verhindern. Ein deutlicher Performanceverlust wäre jedoch die Folge.

3.19 System Call “lseek”

- Syntax

```
long lseek ( /* move file offset pointer */
                int fd,         /* file descriptor */
                long offset,    /* offset in file  */
                int whence      /* interpretation of offset */
           );
/* returns file offset pointer or -1 on error */
```

- Beschreibung

Der System Call *lseek (2)* verändert den *File Offset Pointer* im *OFT* Slot. Diesen neuen Startpunkt für die nächste *read* - oder *write* -Operation bestimmt *lseek (2)* aus **offset** und **whence:**

	whence	Neuer Wert des File Offset Pointer ist
0	**SEEK_SET**	*offset* Bytes vom Dateianfang
1	**SEEK_CUR**	momentane Position plus *offset*
2	**SEEK_END**	Dateiende plus *offset*

offset kann auch eine negative Zahl sein.

Die symbolischen Konstanten für **whence** definiert (in neueren *UNIX* Versionen) die Header-Datei *<unistd.h>* .

lseek (2) kann nicht auf alle I/O-Verbindungen angewendet werden. Bei einem Terminal oder einer Pipe hat der File Offset Pointer keine sinnvolle Bedeutung. *lseek (2)* signalisiert deshalb einen Fehler mit dem Rückgabewert *-1* . Im Erfolgsfall gibt *lseek (2)* den neuen Wert des File Offset Pointer zurück.

- Implizites *lseek*

Enthalten die **oflags** im *OFT* Slot einer I/O-Verbindung *O_APPEND ,* so verursacht jeder *write* System Call vor dem Datentransfer eine implizite *lseek -Operation,* die den File Offset Pointer zuverlässig ans Ende der Datei positioniert.

Zwei explizite System Call Aufrufe, ein *lseek (2)* gefolgt von einem *write (2),* sind in einem *multiuser / multitasking* Betriebssystem wie *UNIX* nicht notwendig eine **atomare** Operation. Ein konkurrierender Prozeß könnte seinen *write* System Call zwischen dem *lseek* und dem *write* System Call des ersten Prozesses ausführen. Dadurch wäre der File Offset Pointer des ersten Prozesses nicht mehr auf das neue Ende der Datei positioniert und der erste Prozeß würde die Daten des zweiten Prozesses überschreiben. Das *O_APPEND* Flag ermöglicht dagegen das sichere und geordnete Anfügen an eine Datei. Der implizite *lseek* und der *write* System Call laufen als eine atomare Operation ab.

3.20 System Call “read”

- Syntax

```
int read ( /* read from file */
                int fd,         /* file descriptor */
                char *buf,      /* buffer address  */
                int nbytes      /* number of bytes to read */
         );
/* returns number of bytes read, 0 on EOF, or -1 on error */
```

- Beschreibung

Der System Call *read (2)* ist das Gegenstück zum System Call *write (2). read* liest **nbytes** von der I/O-Verbindung **fd** in den Puffer mit der Startadresse **buf** im Adreßraum des Prozesses.

Die Datenübertragung beginnt mit dem Byte, auf das der File Offset Pointer im *OFT* Slot zeigt. Nach Abschluß des *read* System Calls zeigt der File Offset Pointer auf das dem letzten übertragenen Byte folgende Byte.

Der Rückgabewert von *read* , — Anzahl der gelesenen Bytes — kann kleiner ausfallen als **nbytes .** Dies tritt auf, wenn über die I/O-Verbindung im Moment weniger als **nbytes** Daten zur Verfügung stehen.

Der Rückgabewert *0* zeigt *EOF* und *-1* einen Fehler an. Im Fehlerfall gibt *errno* genauere Auskunft über die Fehlerursache.

- Besonderheit: Non-Blocking Read

Durch die Abstraktion aller Datenquellen/ziele zu *Files* , kann mit *read* nicht nur von (Platten-) Dateien, sondern auch von Terminals, *Pipes* und *Streams -Devices* gelesen werden.

Wenn über eine derartige I/O-Verbindung gerade keine Daten zur Verfügung stehen und ein Prozeß führt darauf einen *read* System Call aus, dann blockiert der Prozeß bis Daten eintreffen. Wurde jedoch beim *open (2)* oder durch *fcntl (2)* das *O_NDELAY / O_NONBLOCK* Flag für diese I/O-Verbindung gesetzt, so kehrt *read* ohne zu blockieren zurück. Bei Terminals und Pipes produziert *read 0* als Rückgabewert, was ein Entdecken von *EOF* unmöglich macht. Dieses Problem vermeidet *read* bei den neueren Stream-Devices, hier kommt der Wert *-1* zurück und *errno* enthält den Wert *EAGAIN* .

3.21 System Call “fcntl”

- Syntax

```
#include <fcntl.h>

int fcntl (      /* control file */
                 int fd,          /* file descriptor */
                 int cmd,         /* command  */
                 int arg /* argument */
          )
/* returns value, file descriptor, or −1 on error */
```

- Beschreibung

Der System Call *fcntl (2)* verändert die Eigenschaften von bereits offenen I/O-Verbindungen. Er manipuliert die **oflag -Bits** im *OFT* Slot. Damit können nachträglich Flagbits wie *O_APPEND* oder *O_SYNC* für eine I/O-Verbindung gesetzt oder gelöscht werden.

- Beispiel

```
#include <fcntl.h>

void clear_append(int fd)              /* clears O_APPEND flagbit */
{
        int oflags;

        if ( (oflags = fcntl( fd, F_GETFL, 0 )) < 0 )
                sysfatal( "fcntl()", "can't get oflags" );

        if ( (oflags & O_APPEND) != O_APPEND )
                return;         /* not set */

        oflags &= ~O_APPEND;    /* clear flagbit */

        if ( fcntl( fd, F_SETFL, oflags ) < 0 )
                sysfatal( "fcntl()", "can't set oflags" );
}
```

3.22 System Call “ioctl”

- Syntax

```
int ioctl (      /* control i/o device */
                 int fd,        /* file descriptor */
                 int cmd,       /* command  */
                 int arg /* argument */
          )
/* returns 0 on success or -1 on error */
```

- Beschreibung

Der System Call *ioctl (2)* verändert die Eigenschaften von offenen I/O-Verbindungen zu Geräten oder IPC-Kanälen. Ein Prozeß kann damit direkt auf einen *Device Driver* Einfluß nehmen. Deshalb sind auch die erlaubten Kommandos sehr abhängig von den einzelnen Gerätetreibern und davon, welche Treibermodule in den Kernel konfiguriert wurden. Kapitel 7 des *Reference Manuals* beschreibt die verschiedenen Möglichkeiten bei den einzelnen Device Drivers.

- Beispiel: Terminal I/O

Der Eintrag *termio (7)* (➔ **#include <termio.h>**) beschreibt den Terminal-Treiber und die Möglichkeiten, auf ihn Einfluß zu nehmen. Mit folgenden Kommandos kann ein Prozeß den *tty -Treiber* parametrisieren:

cmd	Funktion
TCGETA	Abholen der aktuellen Parameterwerte. Komponenten von *arg* werden gefüllt.
TCSETA	Setzen der neuen Parameterwerte in *arg* .
TCSETAW	Warten, bis alle gepufferten Zeichen ausgegeben, dann setzen der neuen Parameterwerte.
TCSETAF	Warten bis Output erledigt, löschen — *flush* — der Input-Queue, dann setzen der neuen Parameterwerte,

Die Übertragung der Parameterwerte erfolgt durch eine Datenstruktur als drittes Argument beim *ioctl -Aufruf.* Die Terminal-Datenstruktur enthält circa 50 Flagbits, die den Terminal-Treiber anweisen, wie die ein- und ausgehenden Zeichen zu bearbeiten, wie die Parameter der Schnittstelle zu setzen sind usw. (vgl. *termio (7)).*

```
struct termio
{       unsigned short  c_iflag;        /* input modes */
        unsigned short  c_oflag;        /* output modes */
        unsigned short  c_cflag;        /* control modes */
        unsigned short  c_lflag;        /* line discipline modes */
        char            c_line;         /* line discipline */
        unsigned char   c_cc[NCC];      /* control chars */
};
```

Der *INIT* -Prozeß, zuständig für die Login-Sequenz an allen Terminal-Leitungen, oder Rechner-Rechner Kommunikationsprogramme (*kermit (1), gonzo (1),...)* wenden diesen System Call intensiv an.

- Anwendung: Kein Echo

Gemäß Voreinstellung für die Kommunikation über ein Terminal, stellt der tty-Treiber alle eingetippten Zeichen am Bildschirm dar — er produziert ein Echo —. Bei verschiedenen Gelegenheiten will man dies aber verhindern, man will das “Echo abschalten”.

```
/* switch off echo-ing characters typed in */
void no_echo( int fd )
{
   struct termio buf;

   if ( ioctl( fd, TCGETA, &buf ) < 0 )
      sysfatal( "ioctl()", "can't get line parms (no_echo)" );

   if ( (buf.c_lflag & ECHO) != ECHO )
      return;      /* not set */

   buf.c_lflag &= ~ECHO;   /* clear bit */

   if ( ioctl( fd, TCSETA, &buf ) < 0 )
      sysfatal( "ioctl()", "can't set line parms (no_echo)" );
}


/* switch on echo-ing characters typed in */
void set_echo( int fd )
{
   struct termio buf;

   if ( ioctl( fd, TCGETA, &buf ) < 0 )
      sysfatal( "ioctl()", "can't get line parms (set_echo)" );

   if ( (buf.c_lflag & ECHO) == ECHO )
      return;      /* already set */

   buf.c_lflag |= ECHO;   /* set bit */

   if ( ioctl( fd, TCSETA, &buf ) < 0 )
      sysfatal( "ioctl()", "can't set line parms (set_echo)" );
}
```

3.23 Synchronisation (mutual exclusion)

- Problem - Concurrency

Ein Prozeß trägt Ergebnisse in eine Datei ein, ein anderer Prozeß versucht gleichzeitig daraus zu lesen - es ist nicht sichergestellt, daß der lesende Prozeß kein "Mischmasch" aus alten und neuen Daten bekommt..

- Lösung: Semaphore

Eine Semaphore ist eine "Ampel", die immer nur **einem** Prozeß grün (Datei benutzen erlaubt) und allen anderen Prozessen rot (Benutzung z.Zt. nicht erlaubt) signalisiert.

Alle Prozesse verplichten sich, nur bei "grün" bestimmten Code auszuführen, z.B. auf eine gemeinsame Datei zuzugreifen. Dadurch, daß immer nur ein Prozeß grün erhält (*mutual exclusion),* synchronisieren sich die Prozesse über die Semaphore.

- Codiertechnik

Eine Semaphore bewacht die Anweisungen des kritischen Teils. Bevor ein Prozeß den kritischen Teil betritt, muß er sich von der Semaphore Grün signalisieren lassen (warten, bis grün gesetzt ist). Wenn er den kritischen Teil verläßt, teilt er dies der Semaphore mit (wieder auf grün setzen). Anschließend kann die Semaphore einem anderen Prozeß Grün signalisieren. Der Notation des Informatiker **Dijkstra** folgend nennt man diese Operationen

P(sema) → Semaphore/Strecke für sich reservieren *(protect).*

V(sema) → Semaphore/Strecke freigeben *(vrej).*

In einem Programm könnte dies etwa folgendermaßen aussehen:

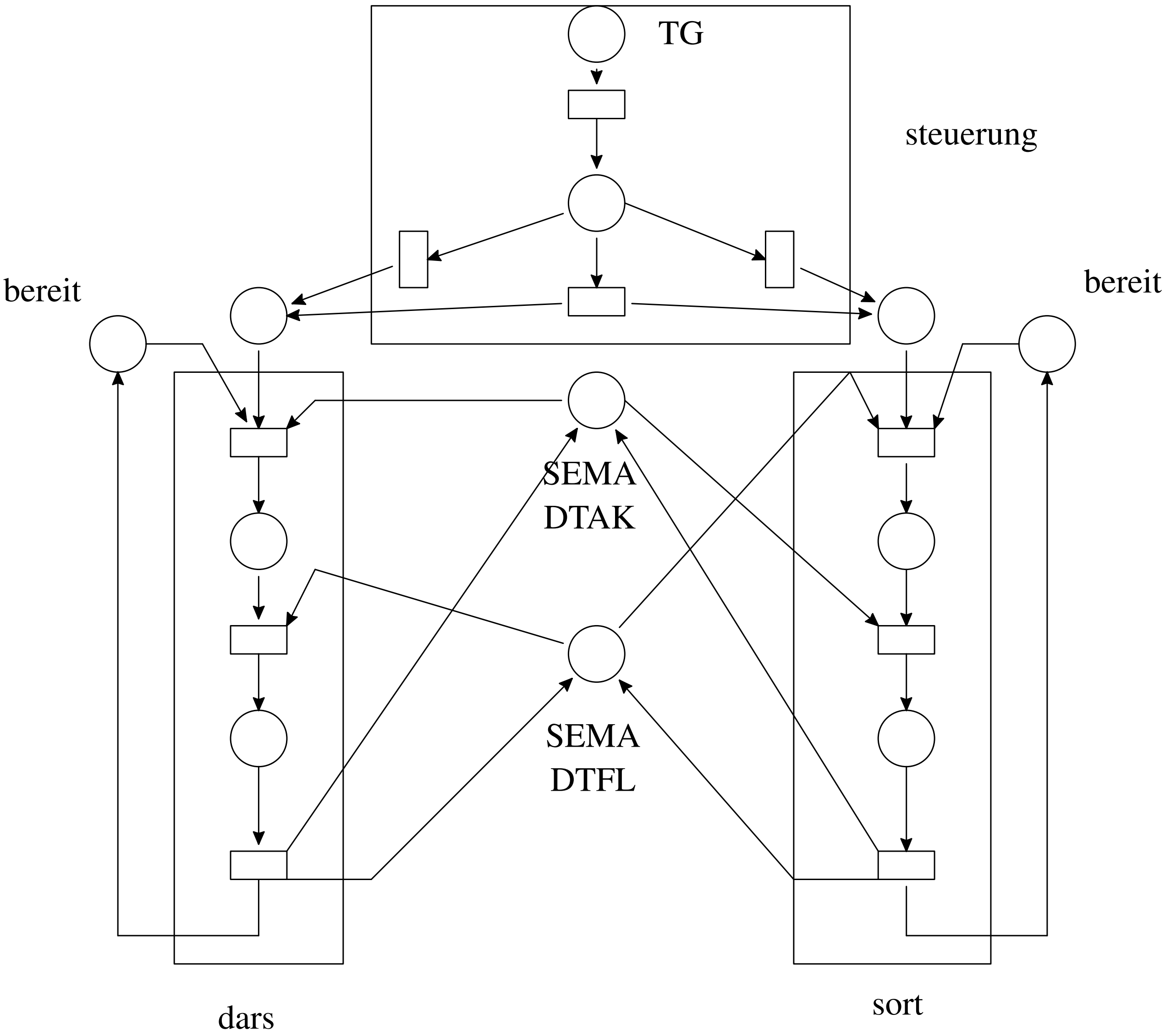
```
/* non-critical region */
        ...

P(semaphore);  /* get sema */
/* begin critical region */
        ...
/* end critical region */
V(semaphore);  /* release sema */

/* non-critical region */
        ...
```

Bem.: Ist die Semaphore auf "rot", so wartet der Prozeß (Operation **P**) solange, bis sie (von wem auch immer) auf "grün gesetzt wurde.

Problem: **Deadlock**

zwei Prozesse arbeiten z.B. gleichzeitig auf zwei Dateien
jeder hat eine Datei gesperrt und wartet darauf, daß der andere seine Datei freigibt, um weiterarbeiten zu können

als Petri-Netz:



Erläuterungen:

Das obige Petri-Netz ist ein sog. **Bedingungs-/Ereignis-Netz.** Die Kreise *(Stellen)* stellen passive Einheiten (Bedingungen, Zustände), die Rechtecke *(Transitionen)* aktive Einheiten dar. Grundsätzlich können nur Stellen mit Transitionen und Transitionen mit Stellen verbunden sein.

Daß eine Bedingung erfüllt ist oder ein Zustand gegeben ist, wird durch eine sog. Markierung in den Stellen ausgedrückt ("Token"). Damit läßt sich ein dynamisches Verhalten erklären:

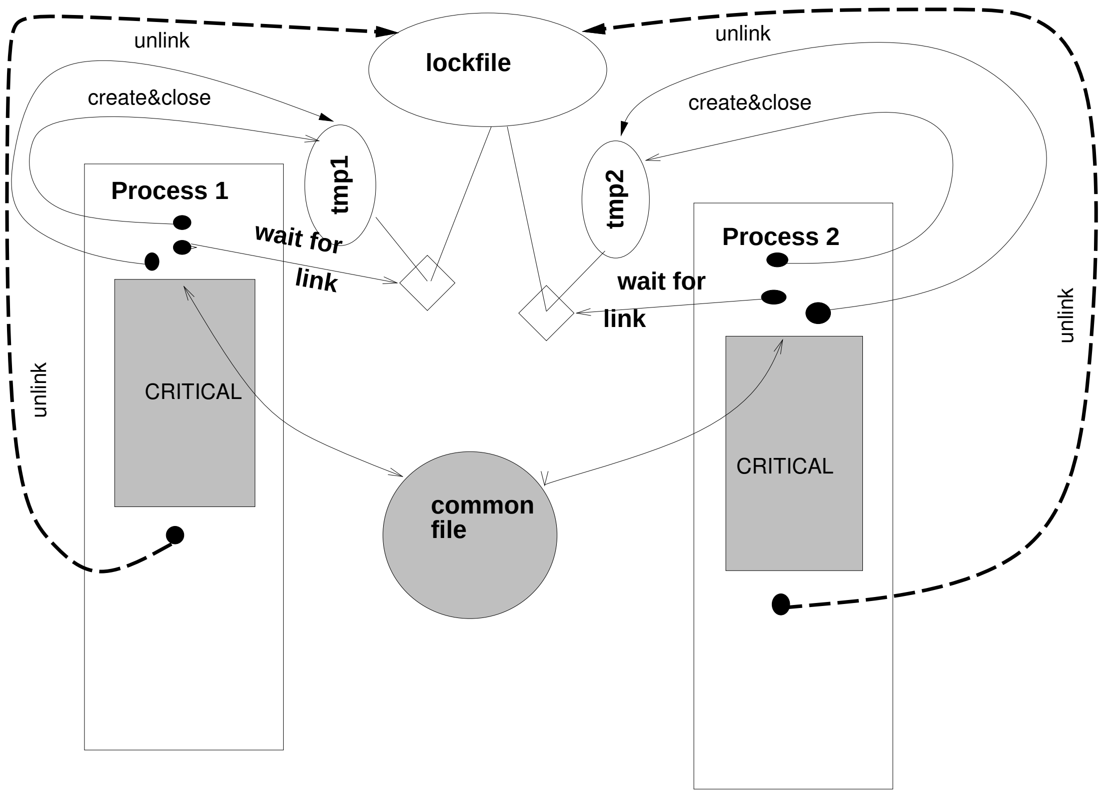
- Eine Transition heißt **schaltbereit,** wenn alle Eingangsstellen markiert sind und alle Ausgangsstellen nicht markiert sind (alternativ dazu könnte der zweite Teil dieser Bedingung auch entfallen; dies hängt davon ab, was wie modelliert werden soll).
- Eine schaltbereite **Transition schaltet,** indem sie die Marken von den Eingangsstellen abzieht und alle Ausgangsstellen markiert. Dieser Vorgang ist zeitlos und unteilbar!

3.24 Synchronisation mit link

Der System Call *link (2)* installiert in einem Directory einen zusätzlichen Verweis auf einen Inode. Sollte bereits ein Eintrag mit diesem Namen existieren, schlägt *link (2)* fehl — unabhängig von der Userid.

Die **P ()** Operation besteht aus einem *creat (2)* oder *open* Aufruf und aus, evtl. wiederholten, *link (2)* Aufrufen.

Die **V ()** Operation besteht aus einem *unlink (2)* Aufruf.



- Realisierung

```
/*sema2.c*/

#include <stdio.h>
#include <errno.h>

#define MAXTRIES  3
#define NAPTIME   1
#define TMPFILE   "LCK_XXXXXX"

/* try to ``get'' the lockfile using `creat(2) and link(2)'
 *      return 0 on success, -1 on errror
 */
int lock( lockfile )
  char *lockfile;
{
  extern int errno;
  static char tmpfile[] = { TMPFILE };
  int tmpfd, link_ret, tries = 0;
  char *mktemp();

  /* generate an unique tmp file name */
  if ( mktemp( tmpfile ) == (char *) 0 )
    perror("can't generate an unique tmp file name (lock)");

  /* create tmp file, then close it */
  if ( (tmpfd = creat( tmpfile, 0 )) < 0 )
    perror( tmpfile);
  if ( close( tmpfd ) < 0 )
    perror( tmpfile );

  /* now try to link tmp file to lock file */
  while ((link_ret=link(tmpfile,lockfile))<0 && errno == EEXIST )
  {     if ( ++tries > MAXTRIES )
        {       unlink( tmpfile );
                return -1;      /* couldn't get the lock */
        }
        sleep( NAPTIME );
  }

  if ( link_ret < 0 )
  {     unlink( tmpfile );
        perror( "link" );
  }

  /* clean up */
  if ( unlink( tmpfile ) < 0 )
        perror( "unlink" );

  return 0;                     /* all fine... */
}
```

Hinweis: In diesem Programm ist auch eine Möglichkeit gezeigt, **eindeutige** (zumindest während der Prozeß lebt) **Dateinamen** zu erzeugen!

Andere Möglichkeiten zur Synchronisation werden in Verbindung mit dem Systemaufruf *open* gezeigt.

3.25 Synchronisation mit lockf

- Syntax

```
#include <sys/types.h>
#include <lockf.h>

int lockf ( /* file and record locking */
                int fd,         /* file descriptor */
                int cmd,        /* command  */
                int size        /* size of locked region */
          );
/* returns 0 on success or -1 on error */
```

- Beschreibung

File und *Record Locking* wurde erst relativ spät in *UNIX* eingeführt. Einige ältere System V Versionen (z.B. *XELOS)* erhielten einen System Call *lockf (2),* andere, neuere Versionen (z.B. System V Rel 3) erweitern die Funktionalität von *fcntl (2)* und stellen eine Bibliotheksfunktion *lockf (3c)* zur Verfügung.

Im einfachsten Fall kann *lockf* eine Datei komplett vor simultanem Zugriff schützen und somit als Semaphore brauchbar machen. *lockf* bietet aber zusätzlich die Möglichkeit, auf genau definierten Teilen einer Datei exklusiven Zugriff durchzusetzen.

Folgende Kommandos unterstützt *lockf :*

cmd	Bedeutung
F_LOCK	Eine Region der Datei für exklusiven Gebrauch nur durch den ausführenden Prozeß reservieren.
F_ULOCK	Vorher reservierte Region der Datei wieder freigeben.

F_TEST	Eine Region der Datei auf Reservierungen durch andere Prozesse überprüfen.
F_TLOCK	Falls in der Region keine Reservierung durch einen anderen Prozeß besteht, die Region für den ausführenden Prozeß reservieren, sonst Rückkehr mit Fehleranzeige. Prozeß blockiert nicht!

Der Bereich der Datei, auf den das Lock-Kommando angewendet werden soll, bestimmt sich aus dem Wert des File Offset Pointer im *OFT* Slot und dem Wert des Argumentes **size .**

- Unteilbare Operation

Ein Aufruf mit *F_LOCK* blockiert solange, bis das Lock gesetzt werden kann.

Um das Blockieren zu verhindern, könnte man eine Sequenz von *lockf -Aufrufen* mit *F_TEST* und *F_LOCK* benutzen. Zwischen den beiden Aufrufen kann aber ein anderer Prozeß erfolgreich *F_LOCK* anwenden, worauf der erste Prozeß doch blockiert. Abhilfe schafft hier *F_TLOCK ,* was *F_TEST* und *F_LOCK* zu einer atomaren, nicht blockierenden Operation zusammenfaßt.

- Idee: System Call oder Büchereifunktion *lockf ()*

Der System Call *lockf (2)* oder die Büchereifunktion *lockf (3c)* schlagen fehl, wenn auf die Datei gerade ein *LOCK* installiert ist — unabhängig von der Userid des ausführenden Prozesses.

Die **P ()** Operation besteht aus, evtl. wiederholten, *lockf ()* Aufrufen mit *F_TLOCK* als Kommando.

Die **V ()** Operation besteht aus einem *lockf ()* Aufruf mit *F_ULOCK* als Kommando.

- Realisierung

```
/* lock.c */

#include <errno.h>
#include <fcntl.h>
#include <unistd.h>

#define MAXTRIES 3
#define NAPTIME 1
static int lock_fd = -1;
```

```
/* try to ``get'' the lockfile using `lockf()'
 * system call/lib routine
 * return 0 on success, -1 on errror
 */
int lock( char * lockfile )
{
  extern int errno;
  int locked, tries = 0;
  int oflags = O_WRONLY | O_CREAT;

  /* open/creat lockfile */
  if ( (lock_fd = open( lockfile, oflags, 0644 )) < 0 )
        perror( "lock" );

  /* try to lock lockfile */
  while ( (locked = lockf( lock_fd, F_TLOCK, 0 )) < 0 )
  {     if ( ++tries > MAXTRIES )
                return -1;      /* couldn't get the lock */
        sleep( NAPTIME );
  }

  return 0;             /* all fine... */
}

/* free lockfile */
void unlock( char * lockfile )
{
  if ( lock_fd < 0  ||  lockf( lock_fd, F_ULOCK, 0 ) < 0 )
        perror( "unlock");
}



int main() {

        char * file = "myfile";
        if (lock(file) == -1 )
                printf("lock error - may be file is in use\n");
        else
                printf("file is locked\n");
}
```

3.26 Kernel Buffering ↔ User Buffering

- Kernel Buffering
  Um den Engpaß “Datentransfer zwischen Hauptspeicher und Platte” zu minimieren, benutzt der Kernel den Buffer Cache.

  Beim Schreiben von Daten auf die Platte verkürzt sich damit die Wartezeit für den Prozeß. Beim Lesen versucht das I/O Subsystem durch Vorauslesen den Prozeß vom Warten auf den tatsächlichen I/O Transfer zu entkoppeln. Häufig benutzte Daten befinden sich dadurch fast immer bereits im Hauptspeicher. Bei Anforderung durch den entsprechenden System Call, müssen die Daten lediglich zwischen dem Buffer Cache — Kernel-Adreßraum — und dem Benutzerprozeß — User-Adreßraum — kopiert werden.

- User Buffering
  Ein Benutzerprozeß muß jeden Datentransfer in/aus seinem Adreßbereich durch einen System Call initiieren.

  System Calls sind teuer !

  Beabsichtigt ein Programm, eine Datei sequentiell zu lesen oder zu schreiben, empfiehlt es sich, nicht für jedes einzelne Zeichen einen System Call abzusetzen. Effizienter ist es, im User-Adreßraum Daten zwischenzulagern und mit den System Calls jeweils größere Datenmengen zwischen User- und Kernel-Adreßraum zu bewegen. Innerhalb seines Adreßraums kann der Prozeß die Daten ohne System Calls in beliebig kleinen Einheiten bearbeiten.

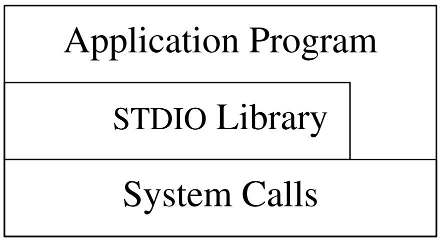
  User Buffering kann mit “günstigen” User-Instruktionen erfolgen. Die Anzahl der System Calls geht zurück und die Ausführungsgeschwindigkeit steigt an.

  Noch steigern läßt sich der Effekt, wenn man beim Transfer vom/zum Kernel die Blockgrößen und -kanten des I/O Subsystems mitberücksichtigt. Die Standard I/O Bibliothek realisiert genau diesen Ansatz.

3.27 High Level I/O - Standard I/O Bibliothek

- Prinzip: Abstrakter Datentyp

  Die Standard I/O Bibliothek definiert sowohl eine Datenstruktur, *FILE* , die eine *high level* I/O-Verbindung repräsentiert, als auch eine Menge von Zugriffsfunktionen, die Objekte vom Typ *FILE* manipulieren.



Die von der Standard I/O Bibliothek gebildete Software-Schicht realisiert das *User Buffering* und bildet die Zugriffsfunktionen auf System Calls ab.

- Zentraler Datentyp: File Pointer

  Neben dem tatsächlichen Pufferbereich benötigt die Standard I/O Bibliothek noch Platz, um Verwaltungsinformationen über die I/O-Verbindungen abzulegen. Eine Datenstruktur faßt pro I/O-Verbindung den Puffer und die Verwaltungsinformationen zusammen.

```
#include <stdio.h>
        ...
FILE *fp;
```

Da sich Datenstrukturen in C relativ einfach mit Pointern manipulieren lassen, heißt der dem *File Deskriptor* entsprechende Begriff zur Identifikation einer high level I/O-Verbindung *File Pointer* .

- Operationsprinzip: User Buffering

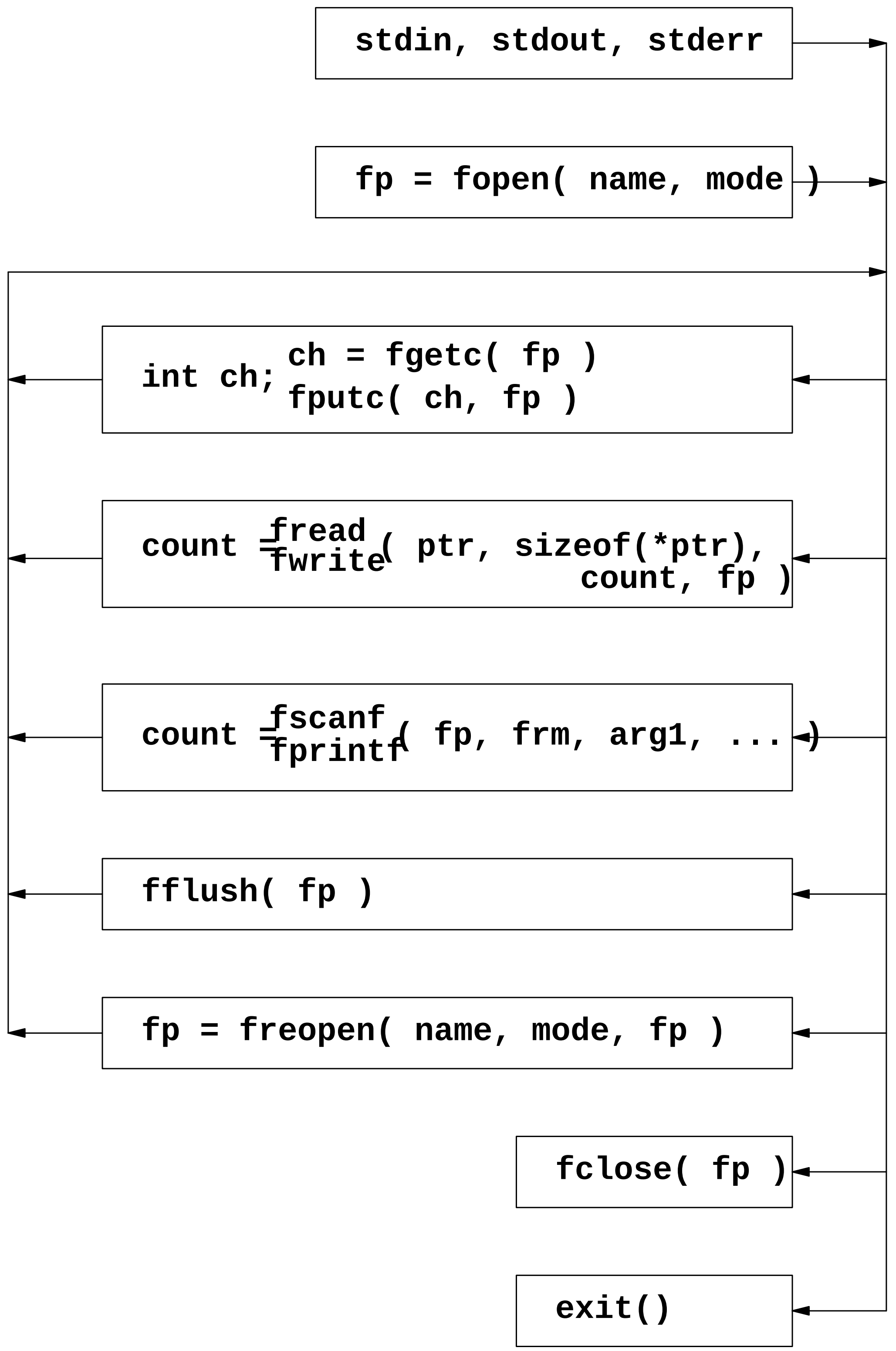
  Alle (Zugriffs-) Funktionen manipulieren die *File Pointer Datenstruktur* . Zwar lautet das Motto der Standard I/O Bibliothek “möglichst wenig System Calls”, doch jede high level I/O-Verbindung über einen File Pointer benutzt eine low level I/O-Verbindung über einen File Deskriptor. Deshalb müssen die Bibliotheks-Funktionen System Calls absetzen, die die low level I/O-Verbindungen kontrollieren.

stdio	System Call
`fopen`	**`open`**
`fclose`	**`close`**
`fwrite, fputc, fprintf`	**`write`**
`fread, fgetc, fscanf`	**`read`**
`fseek`	**`lseek`**
...	...

  Die Bibliotheks-Funktion *fileno(fp)* liefert den zum File Pointer **fp** gehörenden File Deskriptor.

3.27.1 Stationen im Leben eines File Pointers

09:53:08 27.01.00
/home/thales/sai/skripte/soft/soft1.2000/./11/pics/fplife.pic


stdin, stdout, stderr
fp = fopen(name, mode)
int ch;
ch = fgetc(fp)
fputc(ch, fp)
count =
fread
fwrite
(ptr, sizeof(*ptr),
count, fp)
count =
fscanf
fprintf
(fp, frm, arg1, ...)
fflush(fp)
fp = freopen(name, mode, fp)
fclose(fp)
exit()

3.27.2 Vorteile der Standard I/O Bibliothek

- Komfort

  Während durch die System Calls, gemäß *UNIX* -Philosophie nur die einfachsten und nötigsten Dienstleistungen zur Verfügung stehen, bieten die Routinen der Standard I/O Bibliothek auch Programmierkomfort:

☞ formatierte Umwandlung von Werten in Text

☞ Operationen nicht nur auf Zeichen/Byte-Basis, sondern auf ‘‘höheren’’ Objekten (Worten, Token)

☞ *ungetc()*

☞ Stringoperationen als Nebenprodukt (*sprintf() ,...)*

- Effizienz

  User Buffering ermöglicht die effizientere Nutzung der Kernel-Resourcen.

  Da es sich bei der Standard I/O Bibliothek um eine sehr zentrale Bibliothek handelt, bringt die Investition in aufwendige Optimierung der einzelnen Routinen wesentliche Performancegewinne. In der Tat kann der Programmierer sicher sein, daß die Routinen der Standard I/O Bibliothek seine I/O-Transfer Aufträge effizient erledigen.

- Portabilität

  Die Standard I/O Bibliothek gehörte früher als de-facto und gehört heute als *ANSI* -Standard zu praktisch jedem C-Kompiler. Wer seine C-Programme mit den Routinen der Standard I/O Bibliothek schreibt, produziert nicht nur effizienten sondern auch portablen Code.

  Die Portabilität bleibt nicht ausschließlich auf verschiedene Versionen des *UNIX* Betriebssystems beschränkt, sondern erstreckt sich auch auf C-Entwicklungsumgebungen unter anderen Betriebssystemen (MS-DOS, OS/2, VMS, BS-2000, ...). Bei den

Nicht-*UNIX* Implementationen ist jedoch etwas Vorsicht geboten, da hier *UNIX* -Konzepte — mehr oder weniger erfolgreich — nachgebildet werden müssen.