Systemnahe Programmierung in C (SPiC)

Teil D Betriebssystemabstraktionen

Daniel Lohmann, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2011

http://www4.informatik.uni-erlangen.de/Lehre/SS11/V_SPIC

Überblick: Teil D Betriebssystemabstraktionen

Programmausführung, Nebenläufigkeit

Programmablauf in einer µC-Umgebung

- Programm läuft "nackt" auf der Hardware
  - ➥ Compiler und Binder müssen ein vollständiges Programm erzeugen
    - ➤ keine Betriebssystemunterstützung zur Laufzeit
  - Funktionalität muss entweder vom Anwender programmiert werden oder in Form von Funktionsbibliotheken zum Programm dazugebunden werden
  - Umgang mit "lästigen Programmierdetails" (z. B. bestimmte Bits setzen) wird durch Makros erleichtert
- Es wird genau ein Programm ausgeführt
  - Programm kann zur Laufzeit "niemanden stören"
  - Fehler betreffen nur das Programm selbst
  - keine Schutzmechanismen notwendig
    - ➥ ABER: Fehler ohne direkte Auswirkung werden leichter übersehen

Programme laden

- generell bei Mikrocontrollern mehrere Möglichkeiten
  - Programm ist schon da (ROM)
  - Bootloader-Programm ist da, liest Anwendung über serielle Schnittstelle ein und speichert sie im Programmspeicher ab
  - spezielle Hardware-Schnittstelle
    - ➤ "jemand anderes" kann auf Speicher zugreifen
    - ➤ Beispiel: JTAG
      spezielle Hardware-Komponente im AVR-Chip, die Zugriff auf die Speicher hat und mit der man über spezielle PINs kommunizieren kann

Programm starten

- Reset bewirkt Ausführung des Befehls an Adresse 0x0000
  - dort steht ein Sprungbefehl auf die Speicheradresse einer start-Funktion, die nach einer Initialisierungsphase die main-Funktion aufruft
  - alternativ: Sprungbefehl auf Adresse des Bootloader-Programms, Bootloader lädt Anwendung, initialisiert die Umgebung und springt dann auf main-Adresse

Fehler zur Laufzeit

- Zugriff auf ungültige Adresse
  - es passiert nichts:
    - ➤ Schreiben geht in's Leere
    - ➤ Lesen ergibt zufälliges Ergebnis
- ungültige Operation auf nur-lesbare / nur-schreibbare Register/Speicher
  - hat keine Auswirkung

Interrupts

- An einer Peripherie-Schnittstelle tritt ein Ereignis auf
  - ➤ Spannung wird angelegt
  - ➤ Zähler ist abgelaufen
  - ➤ Gerät hat Aufgabe erledigt (z. B. serielle Schnittstelle hat Byte übertragen, A/D-Wandler hat neuen Wert vorliegen)
  - ➤ Gerät hat Daten für die Anwendung bereit stehen (z. B. serielle Schnittstelle hat Byte empfangen)
- ? wie bekommt das Programm das mit?
  - ➤ Zustand der Schnittstelle regelmäßig überprüfen (= ***Polling***)
  - ➤ Schnittstelle meldet sich von sich aus beim Prozessor und unterbricht den Programmablauf (= ***Interrupt***)

Polling vs. Interrupts

- Polling
  - \+ Pollen erfolgt **synchron** zum Programmablauf, Programm ist in dem Moment auf das Ereignis vorbereitet
  - – Pollen erfolgt explizit im Programm und meistens umsonst — Rechenzeit wird verschwendet
  - – Polling-Funktionalität ist in den normalen Pragrammablauf eingestreut — und hat mit der "eigentlichen" Funktionalität dort meist nichts zu tun
- Interrrupts
  - \+ Interrupts melden sich nur, wenn tatsächlich etwas zu erledigen ist
  - \+ Interrupt-Bearbeitung ist in einer Funktion kompakt zusammengefasst
  - – Interrupts unterbrechen den Programmablauf irgendwo (**asynchron**), sie könnten in dem Augenblick stören
    - ➥ durch die Interrupt-Bearbeitungentsteht **Nebenläufigkeit**

Implementierung von Interrupts

- typischerweise mehrere Interrupt-Quellen
- Interrupt-Vektor
  - Speicherbereich (Tabelle), der für jeden Interrupt Informationen zur Bearbeitung enthält
    - ➤ Maschinenbefehl (typischerweise ein Sprungbefehl auf eine Adresse, an der eine Bearbeitungsfunktion (***Interrupt-Handler***) steht) oder
    - ➤ Adresse einer Bearbeitungsfunktion
  - feste Position im Speicher — ist im Prozessorhandbuch nachzulesen
- Maskieren von Interrupts
  - Bit im Prozessor-Statusregister schaltet den Empfang aller Interrupts ab
  - zwischenzeitlich eintreffende Interrupts werden gepuffert (nur einer!)
  - die Erzeugung einzelner Interrupts kann am jeweiligen Gerät unterbunden werden

Interrupts: Ablauf auf Hardware-Ebene

① Gerät löst Interrupt aus,
Ablauf des Anwendungsprogramms wird unmittelbar unterbrochen

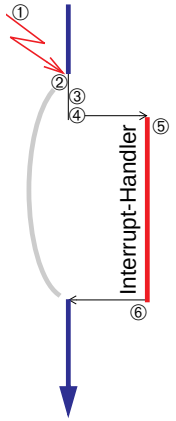
② weitere Interrupts werden deaktiviert

③ aktuelle Position im Programm wird gesichert

④ Eintrag im Interrupt-Vektor ermitteln

⑤ Befehl wird ausgeführt bzw. Funktion aufrufen
(= Sprung in den Interrupt-Handler)

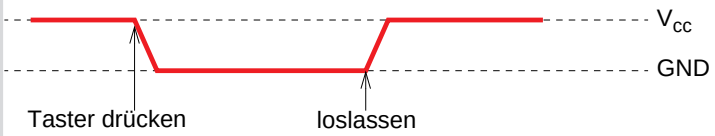
⑥ am Ende der Bearbeitungsfunktion bewirkt ein Befehl "Return from Interrupt" die Fortsetzung des Anwendungsprogramms und die Reaktivierung der Interrupts



! Der Interrupt-Handler muss alle Register, die er ändert am Anfang sichern und vor dem Rücksprung wieder herstellen!

Pegel- und Flanken-gesteuerte Interrupts

- Beispiel: Signal eines (idealisierten) Tasters



- Flanken-gesteuert
  - ➤ Interupt wird durch die Flanke (= Wechsel des Pegels) ausgelöst
  - ➤ welche Flanke einen Interrupt auslöst kann bei manchen Prozessoren konfiguriert werden
- Pegel-gesteuert
  - ➤ solange ein bestimmter Pegel anliegt (hier Pegel = GND) wird immer wieder ein Interrupt ausgelöst

Nebenläufigkeit – Überblick

- Definition von Nebenläufigkeit:
zwei Programmausführungen sind nebenläufig, wenn für zwei einzelne Befehle a und b aus beiden Ausführungen nicht feststeht, ob a oder b tatsächlich zuerst ausgeführt wird
- Nebenläufigkeit tritt auf
  - ➤ bei Interrupts
  - ➤ bei parallelen Abläufen (gleichzeitige Ausführung von Code in einem Mehrprozessorsystem mit Zugriff auf den gleichen Speicher)
  - ➤ bei quasi-parallelen Abläufen (wenn ein Betriebssystem verschiedenen Prozesse den Prozessor jeweils für einen Zeitraum zuteilt und ihn nach Ablauf der Zeit wieder entzieht)
- Problem:
  - ➤ was passiert, wenn die nebenläufigen Ausführungen auf die gleichen Daten im Speicher zugreifen?

Nebenläufigkeit durch Interrupts

- Interrupts unterbrechen Anwendungsprogramme "irgendwo"
- Interrupts haben Zugriff auf den gleichen Speicher
- Szenario:
  - ➤ eine Lichtschranke soll Fahrzeuge zählen und alle 10 Sekunden soll der Wert ausgegeben werden

```
static volatile uint16_t a;

void main(void) {
  volatile uint32_t i;
  while(1) {
    for (i=0; i<2000000; i++)
    /* Zählen dauert 10 Sek. */;
    print(a);
    a=0;
  }
}
```

```
/* Lichtschranken-
   Interrupt */
void count(void) {
        a++;
}
```

Nebenläufigkeit durch Interrupts (2)

- Auf C-Ebene führt die Interrupt-Behandlung nur einen Befehl aus: a++
  - nur scheinbar ein Befehl
  - auf Maschinencode-Ebene (Bsp. AVR) sieht die Sache anders aus

```
...
    print(a);
    a=0;
...
```

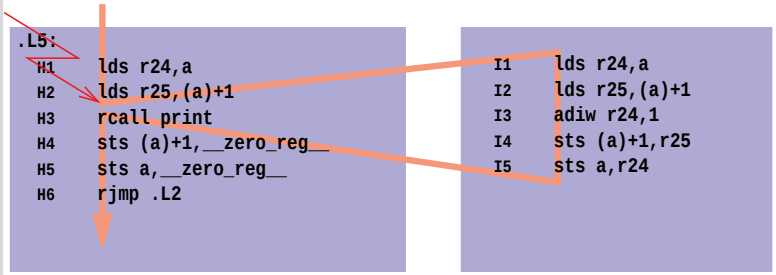
```
void count(void) {
        a++;
}
```

```
...
.L5:
        lds r24,a
        lds r25,(a)+1
        rcall print
        sts (a)+1,__zero_reg__
        sts a,__zero_reg__
        rjmp .L2
...
```

```
...
        lds r24,a
        lds r25,(a)+1
        adiw r24,1
        sts (a)+1,r25
        sts a,r24
...
```


Nebenläufigkeit durch Interrupts (3)

- Annahme1: Interrupt trifft folgendermaßen ein:



  - Folge: ein Fahrzeug wird nicht gezählt
    - ➥ **Lost-Update-Problem**
- Details des Szenarios zeigen mehrere Problemstellen:
  - uint16-Wert wird in zwei Schritten in zwei Register geladen (uint32: 4 Register)
  - Operationen erfolgen in Registern, danach wird in Speicher zurückgeschrieben

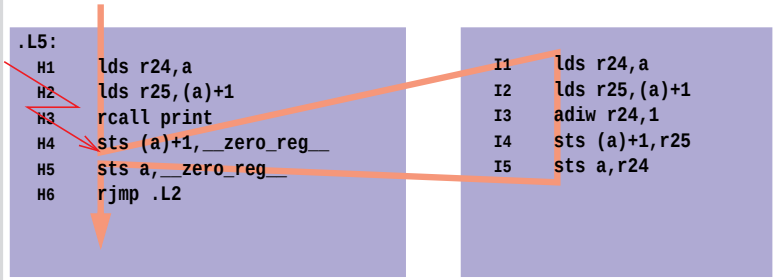
Nebenläufigkeit durch Interrupts (3)

- Skizze zu Annahme 1, a habe initial den Wert 5

Code-zeile	Variable **a**		Prozessor-Register		gesicherte Registerinhalte		Ausgabe von print
	oberes Byte	unteres Byte	**r25**	**r24**	**r25**	**r24**	
initial	**00**	**05**					
H1		05		**05**			
H2	00		**00**				
INT			00	05	**00**	**05**	
I1		05		**05**			
I2	00		**00**	05			
I3			**00**	**06**			
I4	**00**		00				
I5		**06**		06			
ret			**00**	**05**	00	05	
H3			00	05			**5**
H4	**00**						
H5		**00**					

Nebenläufigkeit durch Interrupts (4)

- Annahme 2: Interrupt trifft folgendermaßen ein:



  - Folge: möglicherweise werden 255 Fahrzeuge zuviel gezählt
    - Variable a ist auf 2 Register verteilt →a = 0 nicht atomar zuerst wird obere Hälfte auf 0 gesetzt
    - falls a++ im Interrupt-Handler a zufällig von 255 auf 256 zählt → Bitüberlauf vom "unteren" in's "obere" Register
    - nach Interrupt wird nur noch untere Hälfte auf 0 gesetzt → a = 256

Nebenläufigkeit durch Interrupts (4)

- Skizze zu Annahme 2, a habe initial den Wert 255

Code-zeile	Variable **a**		Prozessor-Register		gesicherte Registerinhalte		Ausgabe von print
	oberes Byte	unteres Byte	**r25**	**r24**	**r25**	**r24**	
initial	**00**	**ff**					
H1		ff		**ff**			
H2	00		**00**				
H3			00	ff			**255**
H4	**00**						
INT			00	ff	**00**	**ff**	
I1		ff		**ff**			
I2	00		**00**	ff			
I3			**01**	**00**			
I4	**01**		01				
I5		**00**		00			
ret			**00**	**ff**	00	ff	
H5	01	**00**					

Nebenläufigkeit durch Interrupts (5)

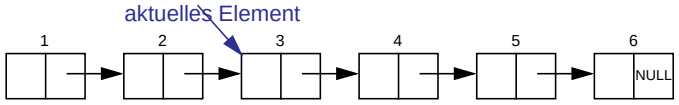
- weiteres Problem bei Zugriff auf globale Variablen:
  - AVR stellt 32 Register zur Vefügung
  - Compiler optimiert Code und vermeidet Speicherzugriffe wenn möglich
    - ➤ Variablen werden möglichst in Registern gehalten
  - Registerinhalte werden bei Interrupt gesichert und am Ende restauriert
    - ➤ Änderungen der Interrupt-Funktion an einer Variablen gehen beim Restaurieren der Register wieder verloren
- Lösung für dieses Problem:
  - Compiler muss Variablen vor jedem Zugriff aus dem Speicher laden und anschließend zurückschreiben
    - ➤ Attribut `volatile`

```
volatile uint16_t a;
```

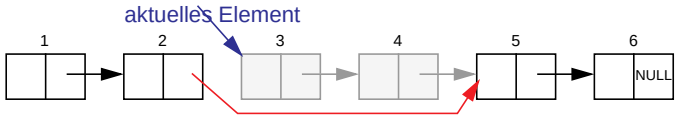
  - Nachteil: Code wird umfangreicher und langsamer
    - ➤ nur einsetzen wo unbedingt notwendig!

Nebenläufigkeitsprobleme allgemein

- Zugriff auf gemeinsame Daten ist bei nebenläufigen Ausführungen generell kritisch
  - ➤ selbst bei einfachen Variablen (siehe vorheriges Beispiel)
  - ➤ Problem bei komplexeren Datenstrukturen (z. B. Einketten einer Struktur in verkettete Liste) noch gravierender: Datenstruktur kann völlig zerstört werden
- Beispiel: Programm läuft durch eine verkettete Liste



  - Interrupt-Handler oder parallel laufendes Programm entfernt Elemente 3 und 4 und gibt den Speicher dieser Elemente frei



Umgang mit Nebenläufigkeitsproblemen

- Gemeinsame Daten möglichst vermeiden
  - ➤ Interrupt-Funktionen sollten weitgehend auf eigenen Daten arbeiten
  - ➤ Parallele Abläufe sollten ebenfalls möglichst eigene Datenbereiche haben
- Kommunikation zwischen Anwendungsabläufen erfordert aber oft gemeinsame Daten
  - ➤ solche Daten sollten deutlich hervorgehoben werden z. B. durch entsprechenden Namen

```
volatile uint16_t INT_zaehler;
```

    - betrifft nur globale Variablen
    - lokale Variablen sind unkritisch (nur in der jeweiligen Funktion sichtbar)
  - ➤ Zugriff auf solche Daten sollte in der Anwendung möglichst begrenzt sein (z. B. nur in bestimmten Funktionen, gemeinsames Modul mit Interrupt-Handlern, vgl. Kap. 12)

Umgang mit Nebenläufigkeitsproblemen (2)

- Zugriffskonflikte mit Interrupt-Handlern verhindern
  - das Programm muss vor kritischen Zugriffen auf gemeinsame Daten Interrupts sperren
    - ➤ Beispiel AVR:
      Funktionen `cli()` (blockiert alle Interrupts)
      und `sei()` (erlaubt Interrupts)
  - Problem: Interrupt-Verluste bei Interrupt-Sperren
    - ➤ trifft ein Interrupt während der Sperre ein, wird im zugehörigen Register das entsprechende Bit gesetzt
    - ➤ treffen weitere Interrupts ein, geht diese Information verloren

➥ Zeitraum von Interruptsperren muss möglichst kurz bleiben!

- ➤ alternativ kann es sinnvoll sein, nur Interrupts des Geräts zu sperren, dessen Handler auch auf die kritischen Daten zugreift
  (hängt vom Einzelfall und von Details der Hardware ab!)

Umgang mit Nebenläufigkeitsproblemen (3)

- Warten auf einen Interrupt
  - Häufiges Szenario: im Programm soll auf ein bestimmtes Ereignis gewartet werden, das durch einen Interrupt signalisiert wird
    - ➤ Warten erfolgt meist passiv (Sleep-Modus des Prozessors)
  - Problem: Abfrage ob Ereignis bereits eingetreten ist, ist ein kritischer Zugriff auf gemeinsame Daten mit der Interrupt-Behandlung

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
       sleep_enable();

       while(1) {
              while(event == 0) { /* Warte auf Ereignis */
                     sleep_cpu();
              }
              /* bearbeite Ereignis */
              ...
```

  - Synchronisation erforderlich?

Umgang mit Nebenläufigkeitsproblemen (4)

- ... Warten auf einen Interrupt
  - Was passiert, wenn der Interrupt an dieser Stelle eintrifft?

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
       sleep_enable();

       while(1) {

              while(event == 0) { /* Warte auf Ereignis */
                     sleep_cpu();
              }

              /* bearbeite Ereignis */
              ...
```

Interrupt!

➥ **Lost-Wakeup-Problem**

Umgang mit Nebenläufigkeitsproblemen (5)

- ... Warten auf einen Interrupt
  - kritischer Abschnitt

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
       sleep_enable();

       while(1) {
                     kritischer Abschnitt
              while(event == 0) { /* Warte auf Ereignis */
                     sleep_cpu();
              }

              /* bearbeite Ereignis */
              ...
```

Interrupt!

  - können hier Interruptsperren helfen?

Umgang mit Nebenläufigkeitsproblemen (6)

- ... Warten auf einen Interrupt
  - Problem: Interruptsperre muss vor dem `sleep_cpu()` aufgehoben werden

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
       sleep_enable();

       while(1) {
              cli();   kritischer Abschnitt
              while(event == 0) { /* Warte auf Ereignis */
                     sei();                    Interrupt!
                     sleep_cpu();

              }

              /* bearbeite Ereignis */
              ...
```

    - ➤ aber Interrupt darf nicht zwischen `sei()` und `sleep_cpu()` kommen
    - ➤ Lösung: `sei()` und die Folgeanweisung werden atomar ausgeführt (Hardware-Mechanismus des AVR-Prozessors!)

Umgang mit Nebenläufigkeitsproblemen (7)

- Einseitige Synchronisation
  - Besonderheit bei Nebenläufigkeit durch Interrupts:
    - ➤ der Interrupt kann den normalen Programmablauf unterbrechen
    - ➤ aber nicht umgekehrt
    - ➥ die Interruptbehandlung wird nie unterbrochen (höchstens durch Interrupts mit höherer Prioriät)
- Mehrseitige Synchronisation
  - Standardsituation bei parallelen Abläufen (z. B. auf Mehrkern-Prozessoren)
    - ➤ Interruptsperren helfen hier nicht
  - Lösungen
    - ➤ spezielle atomare Maschinenbefehle (z. B. test-and-set oder compare-and-swap bei Intel-Architekturen)
    - ➤ Software-Synchronisation (lock-Variablen, Semaphore, etc.)
    - ➤ Kommunikation mittels Nachrichten statt gemeinsamer Daten

Überblick: Teil D Betriebssystemabstraktionen

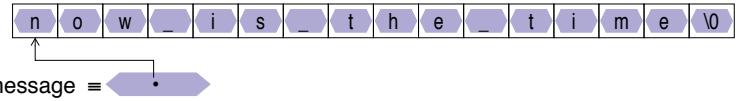
16 Ergänzungen zur Einführung in C

Zeiger, Felder und Zeichenketten

- ➤ Zeichenketten sind Felder von Einzelzeichen (**`char`**), die in der internen Darstellung durch ein `'\0'`–Zeichen abgeschlossen sind

- wird eine Zeichenkette zur Initialisierung eines char–Feldes verwendet, ist der Feldname ein konstanter Zeiger auf den Anfang der Zeichenkette

```
char amessage[] = "now is the time";
```

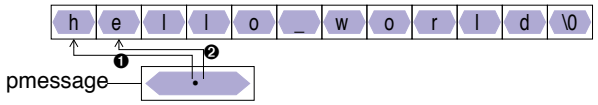


  - ➤ es wird ein Speicherbereich für 16 Bytes reserviert und die Zeichen werden in diesen Speicherbereich hineinkopiert
  - ➤ `amessage` ist ein konstanter Zeiger auf den Anfang des Speicherbereichs und kann nicht verändert werden
  - ➤ der Inhalt des Speicherbereichs kann aber modifiziert werden `amessage[0] = 'h';`

… Zeiger, Felder und Zeichenketten (2)

- wird eine Zeichenkette zur Initialisierung eines **`char`**–Zeigers verwendet, ist der Zeiger eine Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird

```
char *pmessage = "hello world";
```



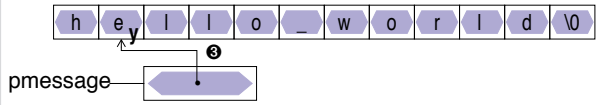

```
pmessage++; ❷
printf("%s", pmessage);  /*gibt "ello world" aus*/
```

  - ➤ es wird ein Speicherbereich für einen Zeiger reserviert (z. B. 4 Byte) und der Compiler legt die Zeichenkette `hello world` an irgendeiner Adresse im Speicher des Programms ab
  - ➤ `pmessage` ist ein variabler Zeiger, der mit dieser Adresse initialisiert wird, aber jederzeit verändert werden darf
    `pmessage++;`

… Zeiger, Felder und Zeichenketten (3)

- wird eine Zeichenkette zur Initialisierung eines **`char`**–Zeigers verwendet, ist der Zeiger eine Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird

```
char *pmessage = "hello world";
```




```
*pmessage = 'y'; ❸
```

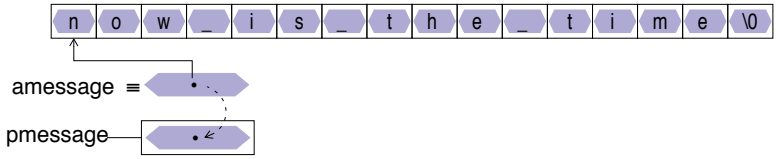
  - ➤ der Speicherbereich von `hello world` darf aber nicht verändert werden
    - – manche Compiler legen solche Zeichenketten in schreibgeschütztem Speicher an
      ➔ Speicherschutzverletzung beim Zugriff
    - – sonst funktioniert der Zugriff obwohl er nicht erlaubt ist
      ➔ Programm funktioniert nur in manchen Umgebungen

… Zeiger, Felder und Zeichenketten (4)

- die Zuweisung eines **`char`**–Zeigers oder einer Zeichenkette an einen **`char`**–Zeiger bewirkt kein Kopieren von Zeichenketten!

```
pmessage = amessage;
```

weist dem Zeiger **`pmessage`** lediglich die Adresse der Zeichenkette **`"now is the time"`** zu



- wird eine Zeichenkette als aktueller Parameter an eine Funktion übergeben, erhält diese eine Kopie des Zeigers

… Zeiger, Felder und Zeichenketten (5)

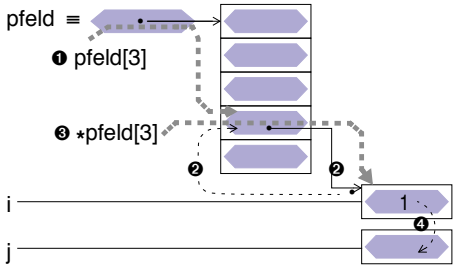
- Zeichenketten kopieren

```
/* 1. Version */
void strcpy(char s[], t[])
{
     int i=0;
     while ( (s[i] = t[i]) != '\0' )
          i++;
}

/* 2. Version */
void strcpy(char *s, *t)
{
     while ( (*s = *t) != '\0' )
          s++, t++;
}

/* 3. Version */
void strcpy(char *s, *t)
{
     while ( *s++ = *t++ )
          ;
}
```

Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden
- Deklaration

```
int *pfeld[5];
int i = 1
int j;
```



- Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i;   ❷
❶
```

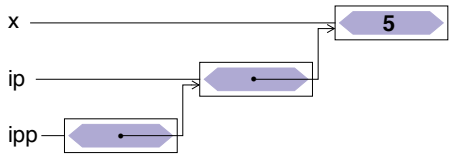
- Zugriffe auf das Objekt, auf das ein Zeiger des Feldes verweist

```
j = *pfeld[3];   ❹
    ❶
❹   ❸
```


Zeiger auf Zeiger

- ein Zeiger kann auf eine Variable verweisen, die ihrerseits ein Zeiger ist

```
int x = 5;
int *ip = &x;

int **ipp = &ip;
/* → **ipp = 5 */
```



➤ wird vor allem bei der Parameterübergabe an Funktionen benötigt, wenn ein Zeiger "call bei reference" übergeben werden muss
(z. B. swap-Funktion für Zeiger)

Argumente aus der Kommandozeile

- beim Aufruf eines Kommandos können normalerweise Argumente übergeben werden
- der Zugriff auf diese Argumente wird der Funktion ***main( )*** durch zwei Aufrufparameter ermöglicht:

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

oder

```
int
main (int argc, char **argv)
{
    …
}
```

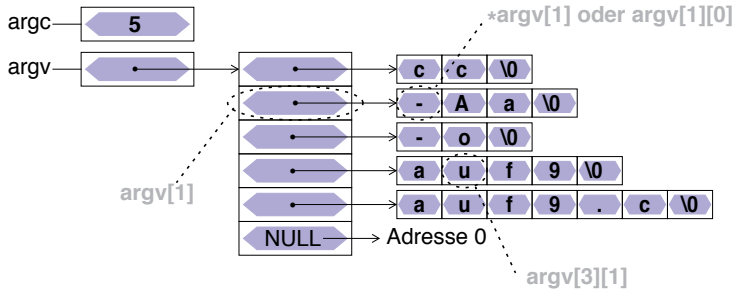
- der Parameter **`argc`** enthält die Anzahl der Argumente, mit denen das Programm aufgerufen wurde
- der Parameter **`argv`** ist ein Feld von Zeiger auf die einzelnen Argumente (Zeichenketten)
- der Kommandoname wird als erstes Argument übergeben (**`argv[0]`**)

Datenaufbau

Kommando: **`cc -Aa -o auf9 auf9.c`**

Datei cc.c:

```
...
main(int argc, char *argv[]) {
...
```

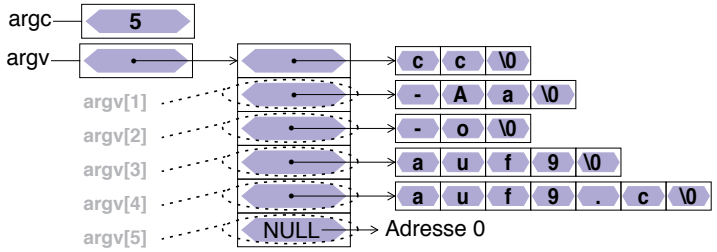


Zugriff
Beispiel: Ausgeben aller Argumente (1)

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

1. Version

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

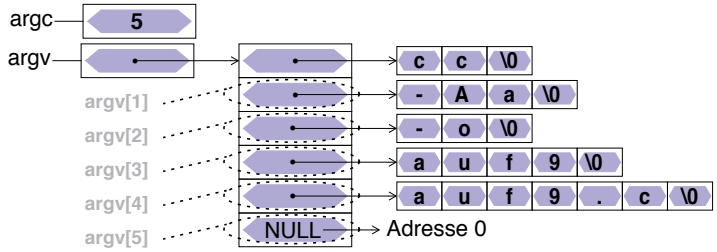


Zugriff
Beispiel: Ausgeben aller Argumente (2)

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

1. Version

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



Strukturen
Rekursive Strukturen

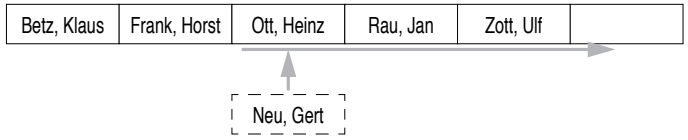
- Strukturen in Strukturen sind erlaubt — aber
  - die Größe einer Struktur muss vom Compiler ausgerechnet werden können
    - ➤ Problem: eine Struktur enthält sich selbst
  - die Größe eines Zeigers ist bekannt (meist 4 Byte)
    - ➤ eine Struktur kann einen Zeiger auf eine gleichartige Struktur enthalten

```
struct liste {
    struct student stud;
    struct liste *rest;
};
```

➥ Programmieren rekursiver Datenstrukturen

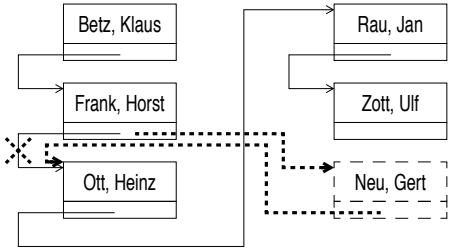
Rekursive Strukturen (2)

- Problem:
  - es sollen beliebig viele Studentendaten eingelesen werden und in sortierter Form im Programm verwaltet werden
- Lösung 1:Feld
  - ➤ wie groß machen? — und was, wenn es nicht reicht?
  - ➤ Einsortieren =
    richtige Position suchen + Rest nach oben verschieben + eintragen



Rekursive Strukturen (3)

- Problem:
  - es sollen beliebig viele Studentendaten eingelesen werden und in sortierter Form im Programm verwaltet werden
- Lösung 2:verkettete Liste von dynamisch angeforderten Strukturen
  - ➤ Speicher für jeden Eintrag mit malloc() anfordern
  - ➤ Einsortieren = richtige Position suchen + zwei Zeiger setzen



Rekursive Strukturen (4)

- Realisierung von Lösung 2 (Skizze):

```
struct eintrag {
       struct student stud;
       struct eintrag *naechster;
};
struct eintrag leer = { {"","",}, NULL}; /* Leeres Listenelement */
struct eintrag *stud_liste;              /* Zeiger auf Listen-Anfang */
struct eintrag *akt_eintrag;             /* aktuell bearbeiteter Eintrag */
struct eintrag *einfuege_pos;            /* Einfuegeposition */

int student_lesen(struct student *);
struct eintrag *suche_pos(struct eintrag *liste, struct eintrag
*element);
/* erstes Listen-Element anfordern */
akt_eintrag = (struct eintrag *)malloc(sizeof (struct eintrag));
stud_liste = &leer;    /* Listenanfang auf leeres Element setzen (vermeidet
später
                          /* eine Sonderbehandlung für Listenanfang) */

while (student_lesen(&akt_eintrag->stud) != EOF ) {
       /* Eintrag, hinter dem einzufügen ist suchen */
       einfuege_pos = suche_pos(stud_liste, akt_eintrag);
       /* akt_eintrag einfügen */
       akt_eintrag->naechster = einfuege_pos->naechster;
       einfuege_pos->naechster = akt_eintrag;
       /* nächstes Listen-Element anfordern */
```


Überblick: Teil D Betriebssystemabstraktionen

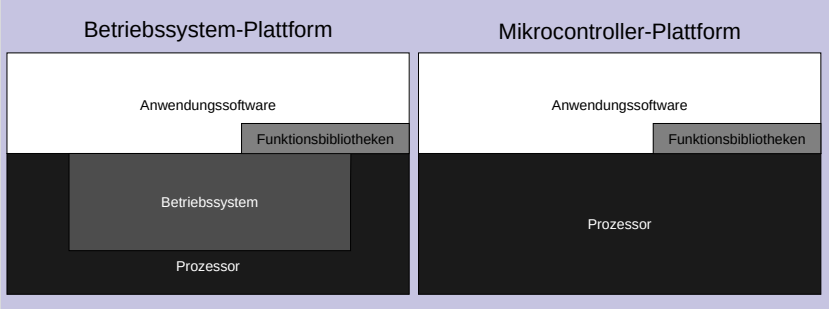
17 Betriebssysteme

Was sind Betriebssysteme?

- DIN 44300
  - „...die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die **Basis der möglichen Betriebsarten** des digitalen Rechensystems bilden und die insbesondere die **Abwicklung von Programmen steuern und überwachen.**“
- Andy Tanenbaum
  - „...eine Software-Schicht ..., die alle Teile des Systems verwaltet und dem Benutzer eine Schnittstelle oder eine *virtuelle Maschine* anbietet, die einfacher zu verstehen und zu programmieren ist [als die nackte Hardware].“
- ★ Zusammenfassung:
  - Software zur Verwaltung und Virtualisierung der Hardwarekomponenten (Betriebsmittel)
  - Programm zur Steuerung und Überwachung anderer Programme

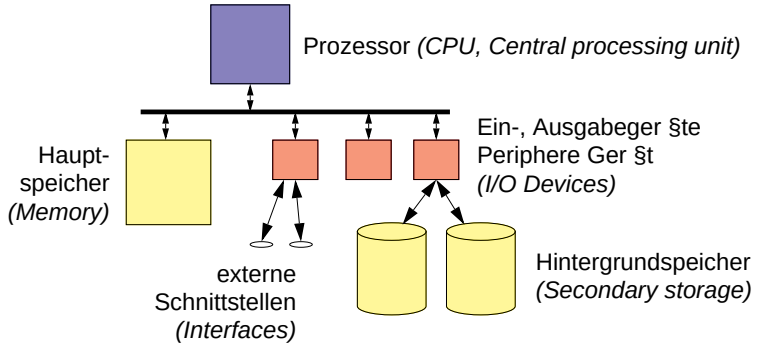
Betriebssystem-Plattform vs. Mikrocontroller

- Entscheidende Unterschiede:
  - Betriebssystem bietet zusätzliche Softwareinfrastruktur für die Ausführung von Anwendungen



  - ➤ Software-Abstraktionen (Prozesse, Dateien, Sockets, Geräte, …)
  - ➤ Schutzkonzepte
  - ➤ Verwaltungsmechanismen

Verwaltung von Betriebsmitteln

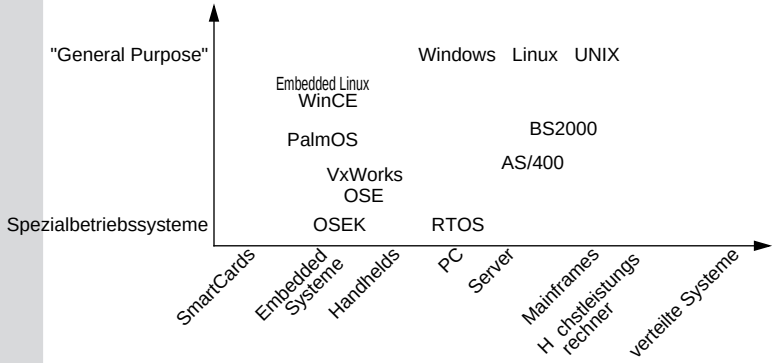


Verwaltung von Betriebsmitteln (2)

- Resultierende Aufgaben
  - Multiplexen von Betriebsmitteln für mehrere Benutzer bzw. Anwendungen
  - Schaffung von Schutzumgebungen
  - Bereitstellen von Abstraktionen zur besseren Handhabbarkeit der Betriebsmittel
- Ermöglichen einer koordinierten gemeinsamen Nutzung von Betriebsmitteln, klassifizierbar in
  - aktive, zeitlich aufteilbare (Prozessor)
  - passive, nur exklusiv nutzbare (periphere Geräte, z.B. Drucker u.Ä.)
  - passive, räumlich aufteilbare (Speicher, Plattenspeicher u.Ä.)
- Unterstützung bei der Fehlererholung

Klassifikation von Betriebssystemen

- Unterschiedliche Klassfikationskriterien
  - ➤ Zielplattform
  - ➤ Einsatzzweck, Funktionalität



Klassifikation von Betriebssystemen (2)

- Wenigen "General Purpose"- und Mainframe/Höchstleistungsrechner-Betriebssystemen steht eine Vielzahl kleiner und kleinster Spezialbetriebssysteme gegenüber:

  C51, C166, C251, CMX RTOS, C-Smart/Raven, eCos, eRTOS, Embos, Ercos, Euros Plus, Hi Ross, Hynet-OS, LynxOS, MicroX/OS-II, Nucleus, OS-9, OSE, OSEK Flex, OSEK Turbo, OSEK Plus, OSEKtime, Precise/MQX, Precise/RTCS, proOSEK, pSOS, PXROS, QNX, Realos, RTMOSxx, Real Time Architect, ThreadX, RTA, RTX51, RTX251, RTX166, RTXC, Softune, SSXS RTOS, VRTX, VxWorks, …

- ➥ Einsatzbereich: Eingebettete Systeme, häufig Echtzeit-Betriebssysteme, über 50% proprietäre (in-house) Lösungen
- Alternative Klassifikation: nach Architektur

Betriebssystemarchitekturen

- Umfang zehntausende bis mehrere Millionen Befehlszeilen
  - Strukturierung hilfreich
- Verschiedene Strukturkonzepte
  - monolithische Systeme
  - geschichtete Systeme
  - Minimalkerne
  - Laufzeitbibliotheken (minimal, vor allem im Embedded-Bereich)
- Unterschiedliche Schutzkonzepte
  - ➤ kein Schutz
  - ➤ Schutz des Betriebssystems
  - ➤ Schutz von Betriebssystem und Anwendungen untereinander
  - ➤ feingranularer Schutz auch innerhalb von Anwendungen

Betriebssystemkomponenten

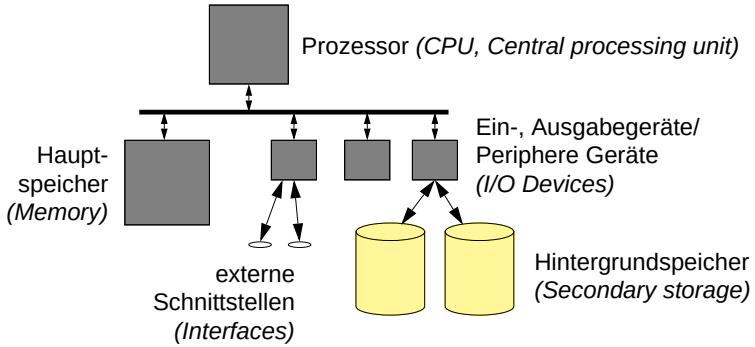
- Speicherverwaltung
  - Wer darf wann welche Information wohin im Speicher ablegen?
- Prozessverwaltung
  - Wann darf welche Aufgabe bearbeitet werden?
- Dateisystem
  - Speicherung und Schutz von Langzeitdaten
- Interprozesskommunikation
  - Kommunikation zwischen Anwendungsausführungen bzw. Teilen einer parallel ablaufenden Anwendung
- Ein/Ausgabe
  - Kommunikation mit der "Außenwelt" (Benutzer/Rechner)

Überblick: Teil D Betriebssystemabstraktionen

18 Dateisysteme

Allgemeine Konzepte

- Einordnung

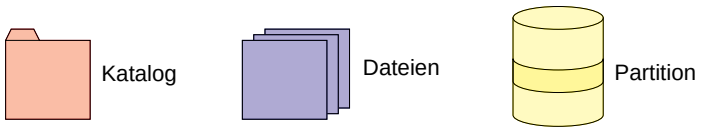


Allgemeine Konzepte (2)

- Dateisysteme speichern Daten und Programme persistent in Dateien
  - Betriebssystemabstraktion zur Nutzung von Hintergrundspeichern (z.B. Platten, CD-ROM, Bandlaufwerke)
    - Benutzer muss sich nicht um die Ansteuerungen verschiedener Speichermedien kümmern
    - einheitliche Sicht auf den Hintergrundspeicher
- Dateisysteme bestehen aus
  - Dateien *(Files)*
  - Katalogen *(Directories)*
  - Partitionen *(Partitions)*

Allgemeine Konzepte (3)

- Datei
  - speichert Daten oder Programme
- Katalog / Verzeichnis (*Directory*)
  - erlaubt Benennung der Dateien
  - enthält Zusatzinformationen zu Dateien
- Partitionen
  - eine Menge von Katalogen und deren Dateien
  - sie dienen zum physischen oder logischen Trennen von Dateimengen.



Ein-/Ausgabe in C-Programmen

Überblick

- 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

Standard Ein-/Ausgabe

- Jedes C-Programm erhält beim Start automatisch 3 E-/A-Kanäle:
  - **`stdin`** Standardeingabe
    - normalerweise mit der Tastatur verbunden, Umlenkung durch **`<`**
    - Dateiende (**`EOF`**) wird durch Eingabe von **`CTRL-D`** am Zeilenanfang signalisiert
  - **`stdout`** Standardausgabe
    - normalerweise mit dem Bildschirm (bzw. dem Fenster, in dem das Programm gestartet wurde) verbunden, Umlenkung durch **`>`**
  - **`stderr`** Ausgabekanal für Fehlermeldungen
    - normalerweise ebenfalls mit Bildschirm verbunden
- automatische Pufferung
  - Eingabe von der Tastatur wird normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem **`NEWLINE`**-Zeichen (**`'\n'`**) an das Programm übergeben!

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
  - ➤ Prototyp:

```
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

Öffnen und Schließen von Dateien (2)

- Beispiel:

```
#include <stdio.h>

int main() {
        FILE *eingabe;
        char dateiname[256];

        printf("Dateiname: ");
        scanf("%s\n", dateiname);

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

        ... /* Programm kann jetzt von eingabe lesen */
        ... /* z. B. mit c = getc(eingabe) */
```

- Schließen eines E/A-Kanals

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

  - ➤ schließt E/A-Kanal **`fp`**

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

Zeichenweise Lesen und Schreiben (2)

- Beispiel: copy-Programm

Teil 1: Dateien öffnen

```
#include <stdio.h>

int main() {
        FILE *quelle;
        FILE *ziel;
        char quelldatei[256], zieldatei[256];
        int c;                  /* gerade kopiertes Zeichen */

        printf("Quelldatei und Zieldatei eingeben: ");
        scanf("%s %s\n", quelldatei, zieldatei);

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

        if ((ziel = fopen(zieldatei, "w")) == NULL) {
                perror(zieldatei);/* Fehlermeldung ausgeben */
                exit(1);        /* Programm abbrechen */
        }

        /* ... */
```

Zeichenweise Lesen und Schreiben (3)

… Beispiel: copy-Programm
— Fortsetzung

Teil 2: kopieren

```
        /* ... */

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

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

Formatierte Ausgabe — Funktionen

- 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
  -> potentielle Sicherheitsprobleme!)
- **`snprintf`** arbeitet analog, schreibt aber maximal nur n Zeichen
  (**`n`** sollte natürlich nicht größer als die Feldgröße sein)

Formatierte Ausgabe — Formatangaben

- 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`** oder **`double`** Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben
`%e`	**`float`** oder **`double`** 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

Formatierte Eingabe — Funktionen

- 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

Formatierte Eingabe — Eingabe-Daten

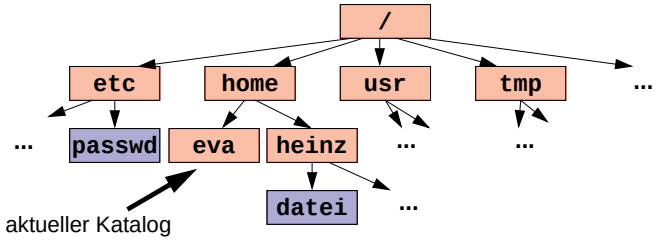
- *White space* (Space, Tabulator oder Newline \n) bildet jeweils die Grenze zwischen Daten, die interpretiert werden
  - ➤ *white space* wird in beliebiger Menge einfach überlesen
  - ➤ Ausnahme: bei Format-Anweisung **`%c`** wird auch *white space* eingelesen
- Alle anderen Daten in der Eingabe müssen 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
- Detail siehe Manual-Seite (`man scanf`)

Dateisystem am Beispiel Linux/UNIX

- Datei
  - einfache, unstrukturierte Folge von Bytes
  - beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - dynamisch erweiterbar
- Katalog
  - baumförmig strukturiert
    - Knoten des Baums sind Kataloge
    - Blätter des Baums sind Verweise auf Dateien
  - jedem UNIX-Prozess ist zu jeder Zeit ein aktueller Katalog *(Current working directory)* zugeordnet
- Partitionen
  - ➤ jede Partition enthält einen eigenen Dateibaum
  - ➤ Bäume der Partitionen werden durch "mounten" zu einem homogenen Dateibaum zusammengebaut (Grenzen für Anwender nicht sichtbar!)

Pfadnamen

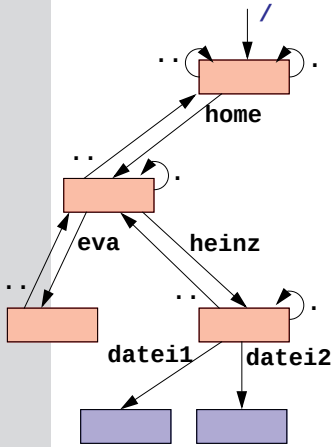
- Baumstruktur



- Pfade
  - z.B. „**`/home/heinz/datei`**“, „**`/tmp`**“, „**`../heinz/datei`**“
  - „**`/`**“ ist Trennsymbol *(Slash)*; beginnender „**`/`**“ bezeichnet Wurzelkatalog; sonst Beginn implizit mit dem aktuellem Katalog

Pfadnamen (2)

- Eigentliche Baumstruktur



- benannt sind nicht Dateien und Kataloge, sondern die Verbindungen (*Links*) zwischen ihnen
  - Kataloge und Dateien können auf verschiedenen Pfaden erreichbar sein z. B. **`../heinz/datei1`** und **`/home/heinz/datei1`**
  - Jeder Katalog enthält
    - einen Verweis auf sich selbst (**`.`**) und
    - einen Verweis auf den darüberliegenden Katalog im Baum (**`..`**)
    - Verweise auf Dateien

Programmierschnittstelle für Kataloge

- Kataloge verwalten
  - Erzeugen

```
int mkdir( const char *path, mode_t mode );
```

  - Löschen

```
int rmdir( const char *path );
```

- Kataloge lesen (Schnittstelle der C-Bibliothek)
  - Katalog öffnen:
    `DIR *opendir( const char *path );`
  - Katalogeinträge lesen:
    `struct dirent *readdir( DIR *dirp );`
  - Katalog schließen:
    `int closedir( DIR *dirp );`

Kataloge (2): opendir / closedir

- Funktionsschnittstelle:

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

DIR *opendir(const char *dirname);

int closedir(DIR *dirp);
```

- Argument von opendir
  - **`dirname`**: Verzeichnisname
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **`DIR`** oder **`NULL`**

Kataloge (3): readdir

- Funktionsschnittstelle:

```
#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!)
- Probleme: Der Speicher für **`struct dirent`** wird von der Funktion readdir beim nächsten Aufruf wieder verwendet!
  - wenn Daten aus der Struktur (z. B. der Dateiname) länger benötigt werden, reicht es nicht, sich den zurückgegebenen Zeiger zu merken sondern es müssen die benötigten Daten kopiert werden

Kataloge (4): struct dirent

- Definition unter Linux (/usr/include/bits/dirent.h)

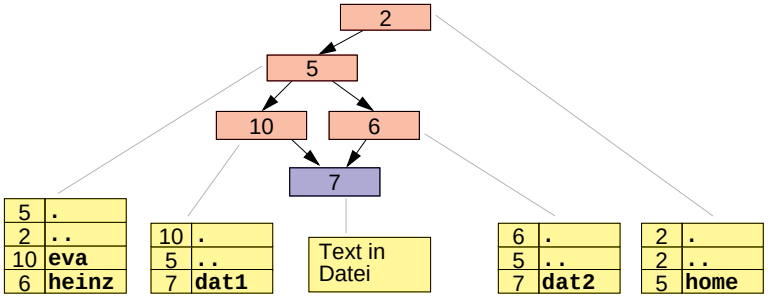
```
struct dirent {
    __ino_t d_ino;
    __off_t d_off;
    unsigned short int d_reclen;
    unsigned char d_type;
    char d_name[256];
};
```

Programmierschnittstelle für Dateien

- siehe C-Ein/Ausgabe (Schnittstelle der C-Bibliothek)
- C-Funktionen (fopen, printf, scanf, getchar, fputs, fclose, ...) verbergen die "eigentliche" Systemschnittstelle und bieten mehr "Komfort"
  - ➤ Systemschnittstelle: open, close, read, write

Inodes

- Attribute (Zugriffsrechte, Eigentümer, etc.) einer Datei und Ortsinformation über ihren Inhalt werden in **Inodes** gehalten
  - Inodes werden pro Partition numeriert *(Inode number)*
- Kataloge enthalten lediglich Paare von Namen und Inode-Nummern
  - Kataloge bilden einen hierarchischen Namensraum über einem eigentlich flachen Namensraum (durchnumerierte Dateien)



Inodes (2)

- Inhalt eines Inode
  - Dateityp: Katalog, normale Datei, Spezialdatei (z.B. Gerät)
  - Eigentümer und Gruppe
  - Zugriffsrechte
  - Zugriffszeiten: letzte Änderung *(mtime)*, letzer Zugriff *(atime)*, letzte Änderung des Inodes *(ctime)*
  - Anzahl der Hard links auf den Inode
  - Dateigröße (in Bytes)
  - Adressen der Datenblöcke des Datei- oder Kataloginhalts

Inodes — Programmierschnittstelle: stat / lstat

- liefert Datei-Attribute aus dem Inode
- Funktionsschnittstelle:

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

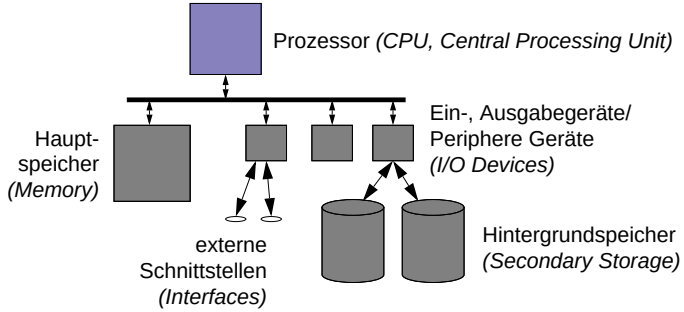
- Argumente:
  - **`path`**: Dateiname
  - **`buf`**: Zeiger auf Puffer, in den Inode-Informationen eingetragen werden
- Rückgabewert: 0 wenn OK, -1 wenn Fehler
- Beispiel:

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

Überblick: Teil D Betriebssystemabstraktionen

19 Prozesse

- Einordnung



Prozessor

- Register
  - Prozessor besitzt Steuer- und Vielzweckregister
  - Steuerregister:
    - Programmzähler *(Instruction Pointer)*
    - Stapelregister *(Stack Pointer)*
    - Statusregister
    - etc.
- Programmzähler enthält Speicherstelle der nächsten Instruktion
  - Instruktion wird geladen und
  - ausgeführt
  - Programmzähler wird inkrementiert
  - dieser Vorgang wird ständig wiederholt

Prozessor (2)

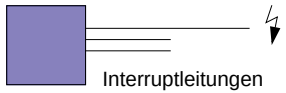
- Beispiel für Instruktionen

```
...
0010 5510000000    movl DS:$10, %ebx
0015 5614000000    movl DS:$14, %eax
001a 8a            addl %eax, %ebx
001b 5a18000000    movl %ebx, DS:$18
```

- Prozessor arbeitet in einem bestimmten Modus
  - Benutzermodus: eingeschränkter Befehlssatz
  - privilegierter Modus: erlaubt Ausführung privilegierter Befehle
    - Konfigurationsänderungen des Prozessors
    - Moduswechsel
    - spezielle Ein-, Ausgabebefehle

Prozessor (3)

- Unterbrechungen *(Interrupts)*



Signalisieren der Unterbrechung
(Interrupt Request; IRQ)

  - ausgelöst durch ein Signal eines externen Geräts
  - ➥ asynchron zur Programmausführung
    - ➤ Prozessor unterbricht laufende Bearbeitung und führt eine definierte Befehlsfolge aus (vom privilegierten Modus aus konfigurierbar)
    - ➤ vorher werden alle Register einschließlich Programmzähler gesichert (z.B. auf dem Stack)
    - ➤ nach einer Unterbrechung kann der ursprüngliche Zustand wiederhergestellt werden
    - ➤ Unterbrechungen werden im privilegierten Modus bearbeitet

Prozessor (4)

- Ausnahmesituationen, Systemaufrufe *(Traps)*
  - ausgelöst durch eine Aktivität des gerade ausgeführten Programms
    - ➤ fehlerhaftes Verhalten
      (Zugriff auf ungültige Speicheradresse, ungültiger Maschinenbefehl, Division durch Null)
    - ➤ kontrollierter Eintritt in den privilegierten Modus
      (spezieller Maschinenbefehl - *Trap* oder *Supervisor Call*)
      – Implementierung der Betriebssystemschnittstelle
  - ➥ synchron zur Programmausführung
  - Prozessor schaltet in privilegierten Modus und führt definierte Befehlsfolge aus (vom privilegierten Modus aus konfigurierbar)
    - ➤ Ausnahmesituation wird geeignet bearbeitet (z. B. durch Abbruch der Programmausführung)
    - ➤ Systemaufruf wird durch Funktionen des Betriebssystems im privilegierten Modus ausgeführt (partielle Interpretation)
  - ➤ Parameter werden nach einer Konvention übergeben (z.B. auf dem Stack)

Programme, Prozesse und Speicher

- **Programm:** Folge von Anweisungen
  (hinterlegt beispielsweise als ausführbare Datei auf dem Hintergrundspeicher)
- **Prozess:** Betriebssystemkonzept
  - ➤ Programm, das sich in Ausführung befindet, und seine Daten
    (Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
  - ➤ eine konkrete Ausführungsumgebung für ein Programm
    Speicher, Rechte, Verwaltungsinformation (verbrauchte Rechenzeit,...),...
- jeder Prozess bekommt einen eigenen virtuellen Adressraum zur Verfügung gestellt
  - ➤ eigener (virtueller) Speicherbereich von 0 bis 2 GB (oder mehr bis 4 GB)
  - ➤ Datenbereiche von verschiedenen Prozessen und Betriebssystem sind gegeneinander geschützt
  - ➤ Datentransfer zwischen Prozessen nur durch Vermittlung des Betriebssystems möglich

Prozesse

- Bisherige Definition:
  - Programm, das sich in Ausführung befindet, und seine Daten
- eine etwas andere Sicht:
  - ein virtueller Prozessor, der ein Programm ausführt
    - ➤ Speicher → virtueller Adressraum
    - ➤ Prozessor → Zeitanteile am echten Prozessor
    - ➤ Interrupts → Signale
    - ➤ I/O-Schnittstellen → Dateisystem, Kommunikationsmechanismen
    - ➤ Maschinenbefehle → direkte Ausführung durch echten Prozessor
      oder partielle Interpretation von Trap-Befehlen durch Betriebssystemcode

Prozesse (2)

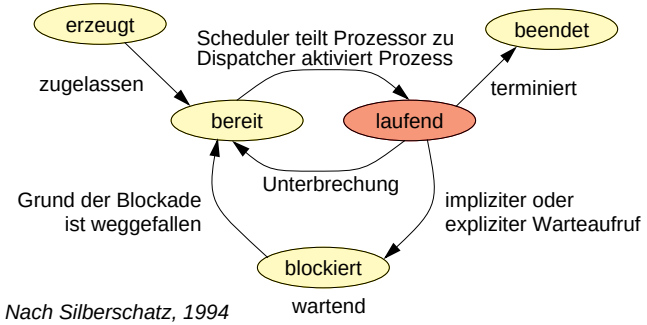
- Mehrprogrammbetrieb
  - mehrere Prozesse können quasi gleichzeitig ausgeführt werden
  - steht nur ein echter Prozessor zur Verfügung, werden Zeitanteile der Rechenzeit an die Prozesse vergeben ***(Time Sharing System)***
  - die Entscheidung, welcher Prozess zu welchem Zeitpunkt wieviel Rechenzeit zugeteilt bekommt, trifft das Betriebssystem ***(Scheduler)***
  - die Umschaltung zwischen Prozessen erfolgt durch das Betriebssystem ***(Dispatcher)***
  - Prozesse laufen nebenläufig
    (das ausgeführte Programm weiß nicht, an welchen Stellen auf einen anderen Prozess umgeschaltet wird)

Prozesszustände

- Ein Prozess befindet sich in einem der folgenden Zustände:
  - **Erzeugt** *(New)*
    Prozess wurde erzeugt, besitzt aber noch nicht alle nötigen Betriebsmittel
  - **Bereit** *(Ready)*
    Prozess besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen
  - **Laufend** *(Running)*
    Prozess wird vom realen Prozessor ausgeführt
  - **Blockiert** *(Blocked/Waiting)*
    Prozess wartet auf ein Ereignis (z.B. Fertigstellung einer Ein- oder Ausgabeoperation, Zuteilung eines Betriebsmittels, Empfang einer Nachricht); zum Warten wird er blockiert
  - **Beendet** *(Terminated)*
    Prozess ist beendet; einige Betriebsmittel sind jedoch noch nicht freigegeben oder Prozess muss aus anderen Gründen im System verbleiben

Prozesszustände (2)

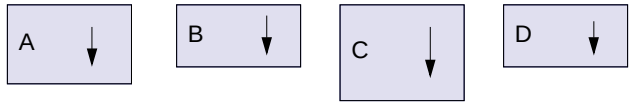
- Zustandsdiagramm



Nach Silberschatz, 1994

Prozesswechsel

- Konzeptionelles Modell

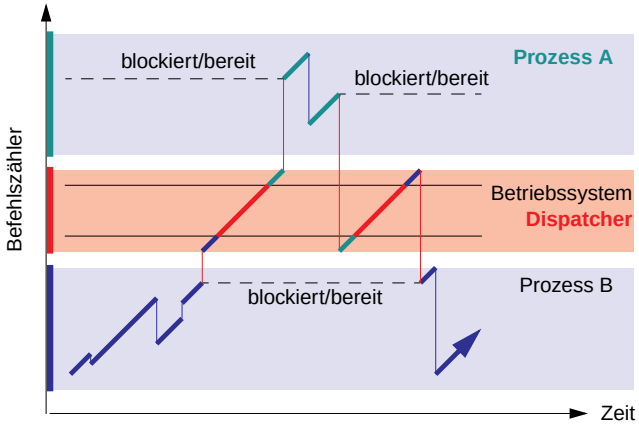


vier Prozesse mit eigenständigen Befehlszählern

- Umschaltung *(Context Switch)*
  - Sichern der Register des laufenden Prozesses inkl. Programmzähler (Kontext),
  - Auswahl des neuen Prozesses,
  - Ablaufumgebung des neuen Prozesses herstellen
    (z.B. Speicherabbildung, etc.),
  - gesicherte Register des neuen Prozesses laden und
  - Prozessor aufsetzen.

Prozesswechsel (2)

- Umschaltung



Prozesswechsel (3)

- Prozesskontrollblock *(Process Control Block; PCB)*
  - Datenstruktur des Betriebssystems, die alle nötigen Daten für einen Prozess hält. Beispielsweise in UNIX:
    - Prozessnummer *(PID)*
    - verbrauchte Rechenzeit
    - Erzeugungszeitpunkt
    - Kontext (Register etc.)
    - Speicherabbildung
    - Eigentümer *(UID, GID)*
    - Wurzelkatalog, aktueller Katalog
    - offene Dateien
    - …

Prozesserzeugung (UNIX)

- Erzeugen eines neuen UNIX-Prozesses
  - Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

Vater

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
```


Prozesserzeugung (UNIX)

- Erzeugen eines neuen UNIX-Prozesses
  - Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

Vater

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
```

Kind

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
```

Prozesserzeugung (2)

- Der Kind-Prozess ist eine perfekte Kopie des Vaters
  - gleiches Programm
  - gleiche Daten (gleiche Werte in Variablen)
  - gleicher Programmzähler (nach der Kopie)
  - gleicher Eigentümer
  - gleiches aktuelles Verzeichnis
  - gleiche Dateien geöffnet (selbst Schreib-, Lesezeiger ist gemeinsam)
  - ...
- Unterschiede:
  - Verschiedene PIDs
  - **`fork()`** liefert verschiedene Werte als Ergebnis für Vater und Kind

Ausführen eines Programms (UNIX)

- Das von einem Prozess gerade ausgeführte Programm kann durch ein neues Programm ersetzt werden

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

Prozess A

```
...
execv( "someprogram", argv, envp );
...
```

Prozess A

```
...
int main( int argc, char *argv[] )
{
    ...
```

das zuvor ausgeführte Programm wird dadurch beendet.

Ausführen eines Programms (UNIX)

- Das von einem Prozess gerade ausgeführte Programm kann durch ein neues Programm ersetzt werden

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

Prozess A

```
...
execv( "someprogram", argv, envp );
...
```

Operationen auf Prozessen (UNIX)

- Prozess beenden

```
void _exit( int status );
[ void exit( int status ); ]
```

- Prozessidentifikator

```
pid_t getpid( void );                /* eigene PID */
pid_t getppid( void );               /* PID des Vaterprozesses */
```

- Warten auf Beendigung eines Kindprozesses

```
pid_t wait( int *statusp );
```

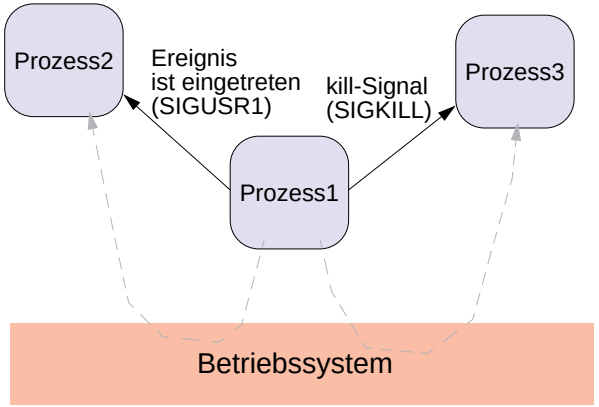
Signale

Signalisierung des Systemkerns an einen Prozess

- Software-Implementierung der Hardware-Konzepte
  - **Interrupt**: asynchrones Signal aufgrund eines "externen" Ereignisses
    - CTRL-C auf der Tastatur gedrückt (Interrupt-Signal)
    - Timer abgelaufen
    - Kind-Prozess terminiert
    - …
  - **Trap:** synchrones Signal, ausgelöst durch die Aktivität des Prozesses
    - Zugriff auf ungültige Speicheradresse
    - Illegaler Maschinenbefehl
    - Division durch NULL
    - Schreiben auf eine geschlossene Kommunikationsverbindung
    - …

Kommunikation zwischen Prozessen

- ein Prozess will einem anderen ein Ereignis signalisieren

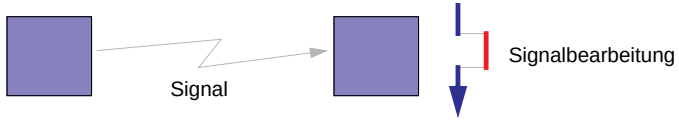


Reaktion auf Signale

- abort
  - erzeugt 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
- signal handler
  - Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses

POSIX Signalbehandlung

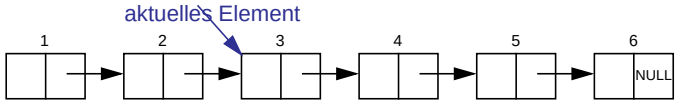
- Betriebssystemschnittstelle zum Umgang mit Signalen
- Signal bewirkt Aufruf einer Funktion (analog ISR)



  - nach der Behandlung läuft der Prozess an der unterbrochenen Stelle weiter
- Systemschnittstelle
  - sigaction – Anmelden einer Funktion = Einrichten der ISR-Tabelle
  - sigprocmask – Blockieren/Freigeben von Signalen ≈ cli() / sei()
  - sigsuspend – Freigeben + passives Warten auf Signal + wieder Blockieren
    ≈ sei() + sleep_cpu() + cli()
  - kill – Signal an anderen Prozess verschicken

Signale und Nebenläufigkeit → Race Conditions

- Signale erzeugen Nebenläufigkeit innerhalb des Prozesses
- resultierende Probleme völlig analog zu Nebenläufigkeit bei Interrupts auf einem Mikrocontroller
- 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