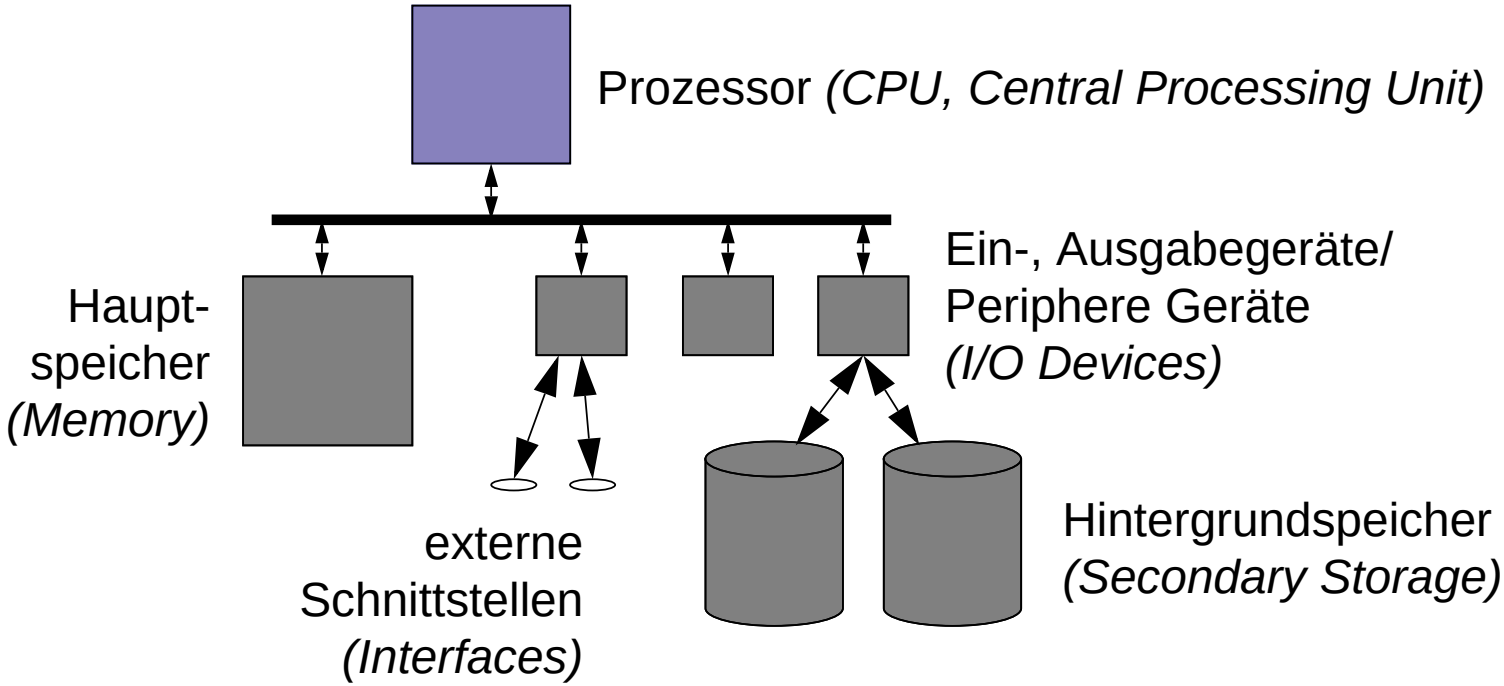
Überblick: Teil D Betriebssystemabstraktionen

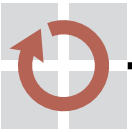
19 Prozesse

- 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)

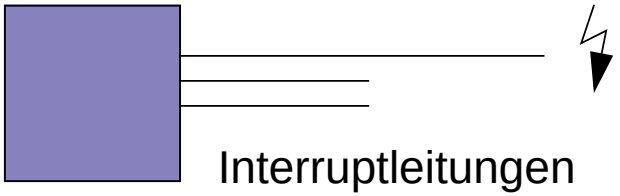
- Beispiel für Instruktionen

```
...
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)

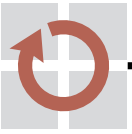
- Ausnahmesituationen, Systemaufrufe *(Traps)*
  - ausgelöst durch eine Aktivität des gerade ausgeführten Programms
    - ➤ fehlerhaftes Verhalten
      (Zugriff auf ungültige Speicheradresse, ungültiger Maschinenbefehl, Division durch Null)
    - ➤ kontrollierter Eintritt in den privilegierten Modus
      (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)

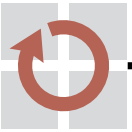
- **Prozess:** Betriebssystemkonzept
  - Programm, das sich in Ausführung befindet, und seine Daten
(Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
  - eine konkrete Ausführungsumgebung für ein Programm
Speicher, Rechte, Verwaltungsinformation (verbrauchte Rechenzeit,...),...

- 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)

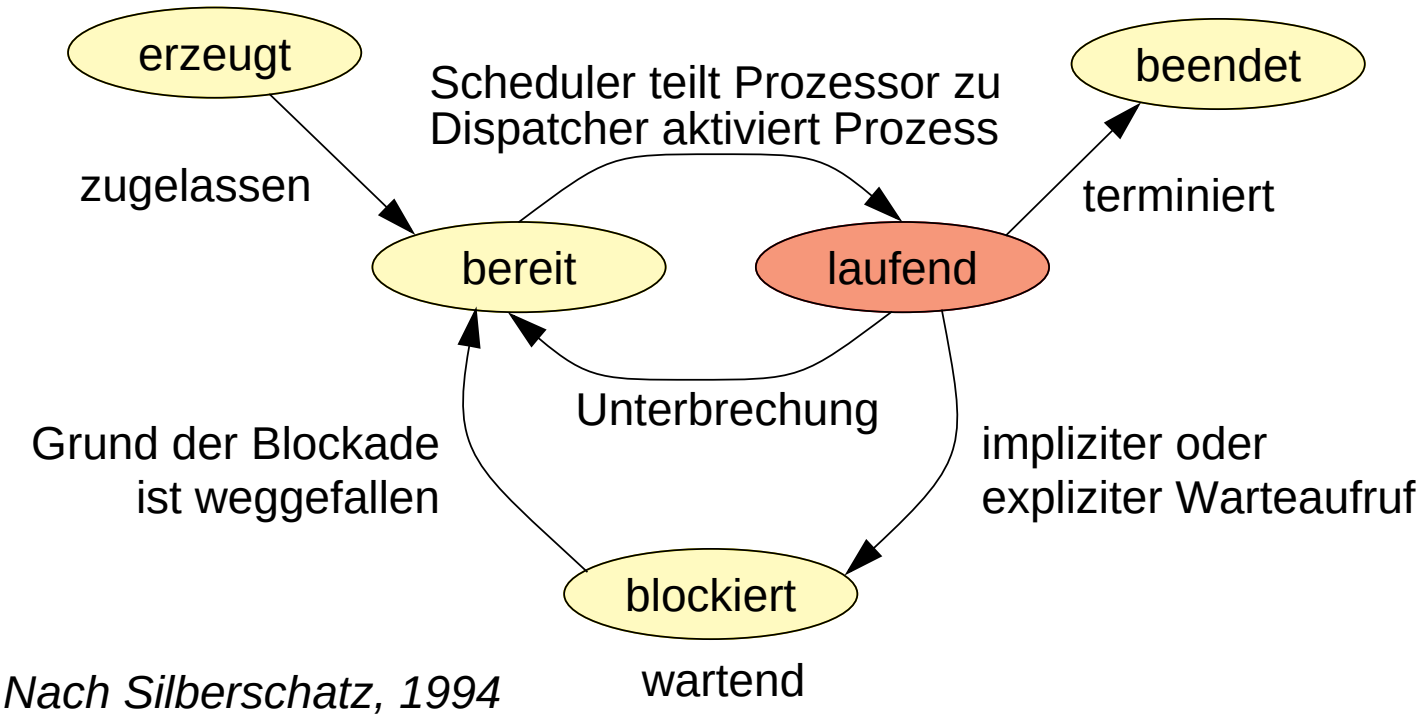
- Mehrprogrammbetrieb
  - mehrere Prozesse können quasi gleichzeitig ausgeführt werden
  - steht nur ein echter Prozessor zur Verfügung, werden Zeitanteile der Rechenzeit an die Prozesse vergeben ***(Time Sharing System)***
  - die Entscheidung, welcher Prozess zu welchem Zeitpunkt wieviel Rechenzeit 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)

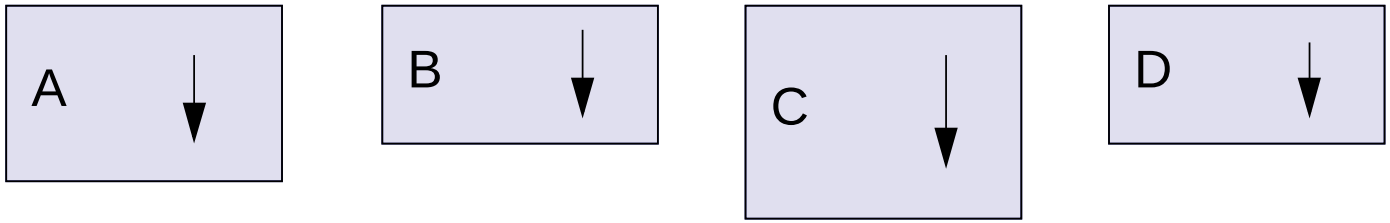
- Zustandsdiagramm



Nach Silberschatz, 1994

Prozesswechsel

- Konzeptionelles Modell

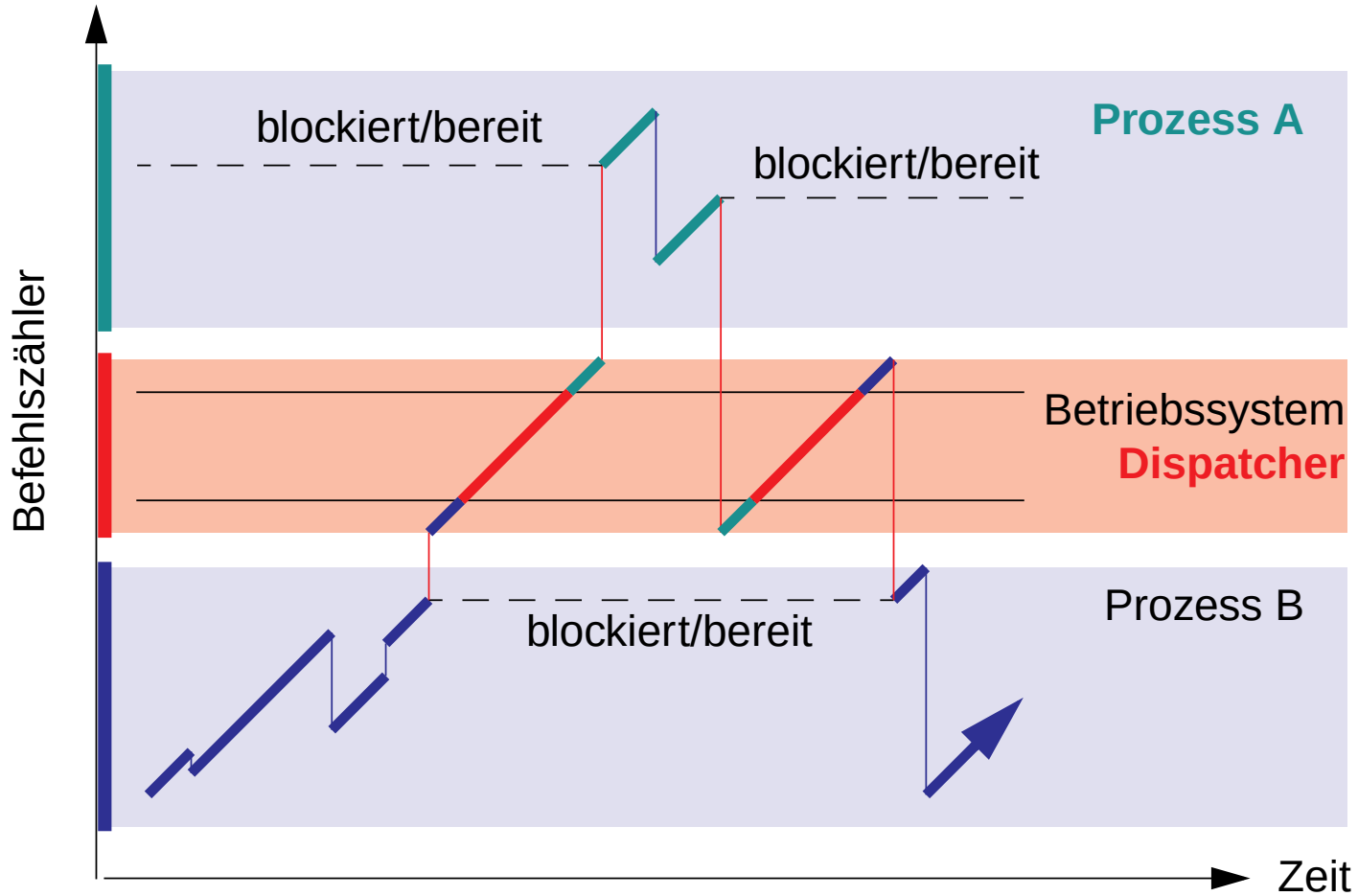


vier Prozesse mit eigenständigen Befehlszählern

- 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.

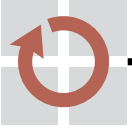
Prozesswechsel (2)

- 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 );
```

```
pid_t p;                                  Vater
...
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

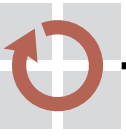
```
pid_t fork( void );
```

Vater

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

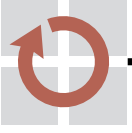
Kind

```
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



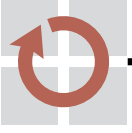
Ausführen eines Programms (UNIX)

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

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

Prozess A

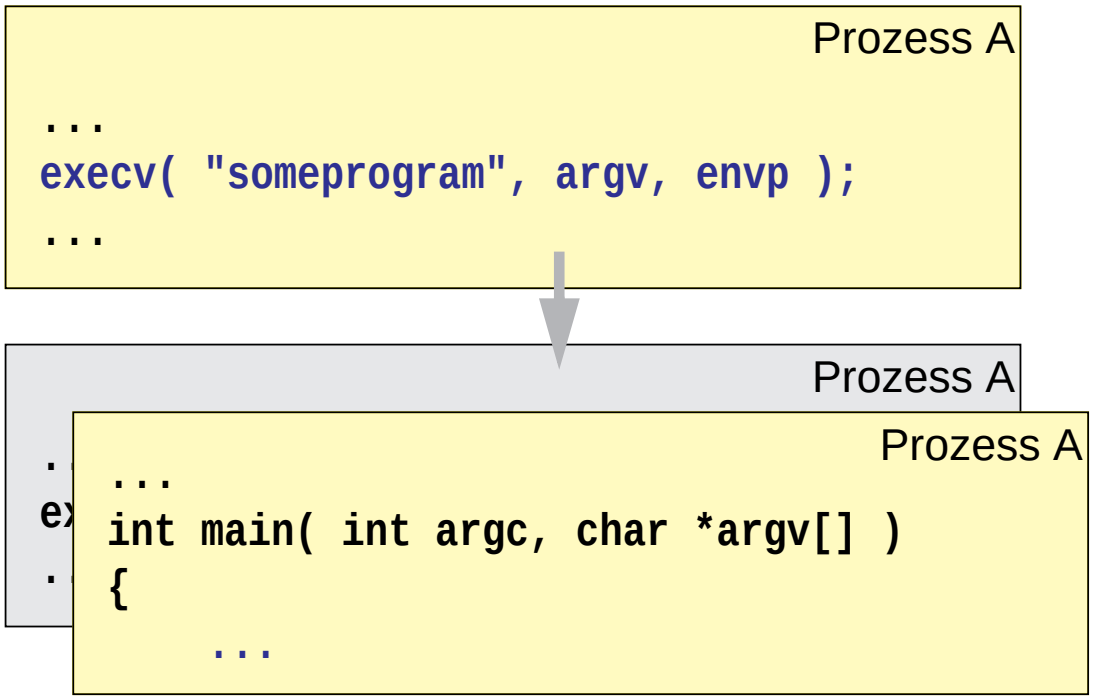
```
...
execv( "someprogram", argv, envp );
...
```

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[]);
```



das zuvor ausgeführte Programm wird dadurch beendet.

Operationen auf Prozessen (UNIX)

- Prozess beenden

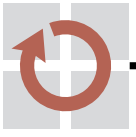
```
void _exit( int status );
[ void exit( int status ); ]
```

- Prozessidentifikator

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

- Warten auf Beendigung eines Kindprozesses

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

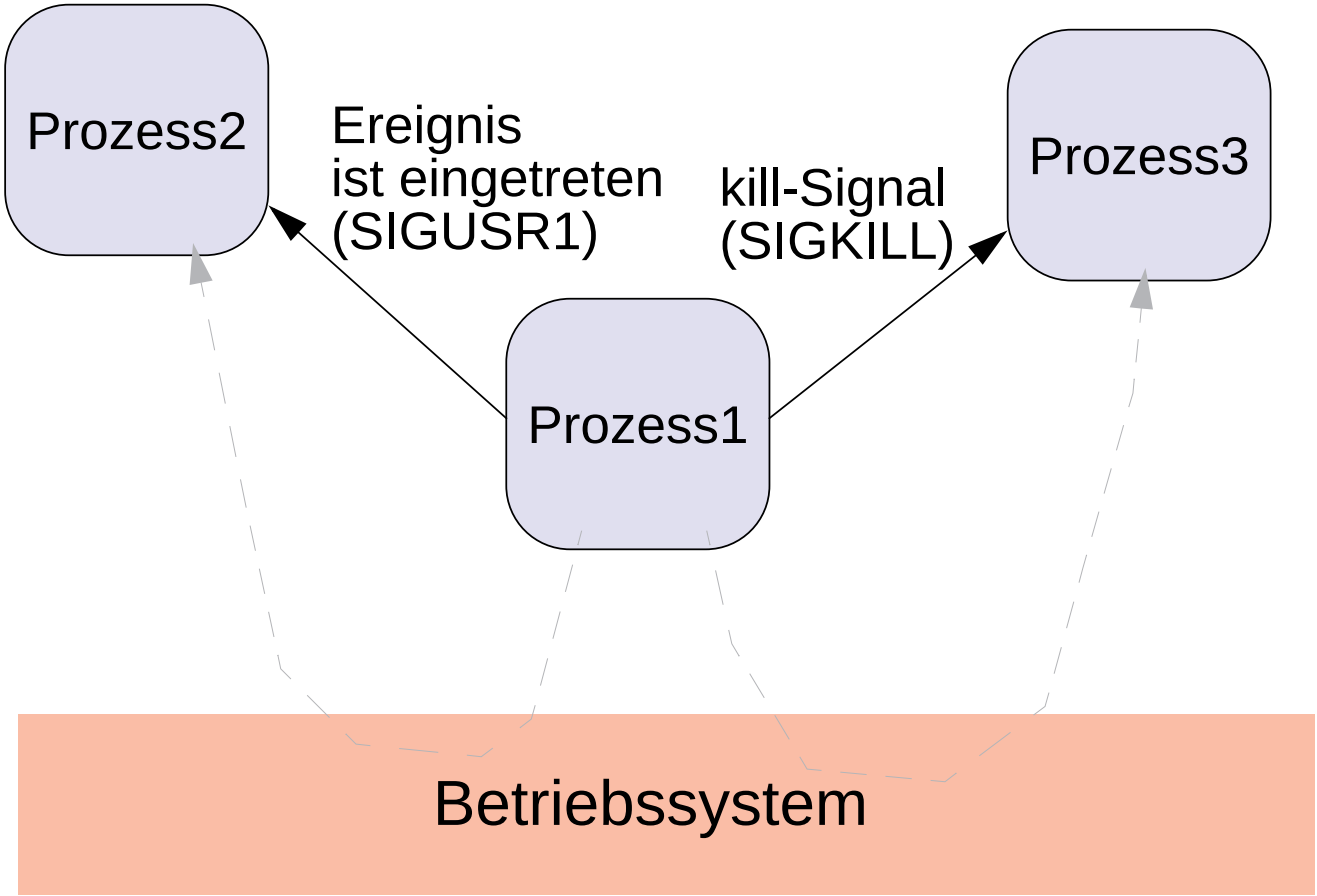
Signale

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

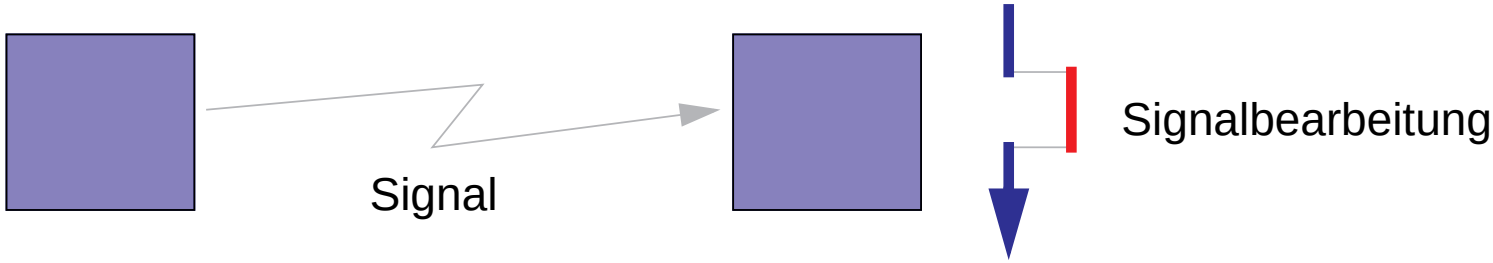


Reaktion auf Signale

- 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

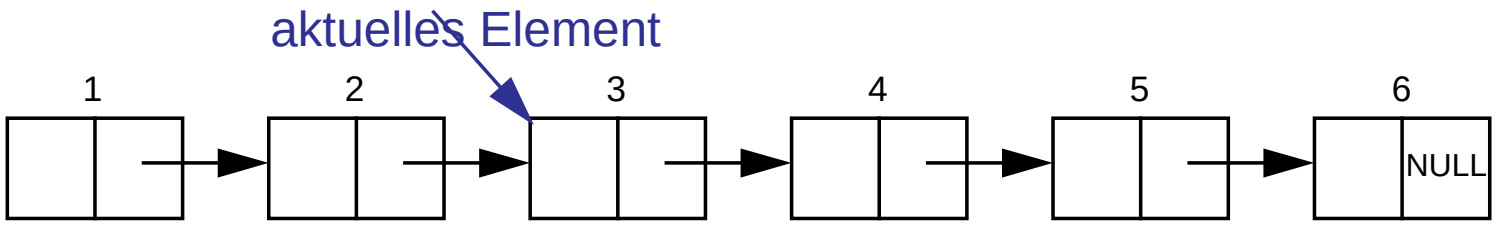
- Betriebssystemschnittstelle zum Umgang mit Signalen
- Signal bewirkt Aufruf einer Funktion (analog ISR)



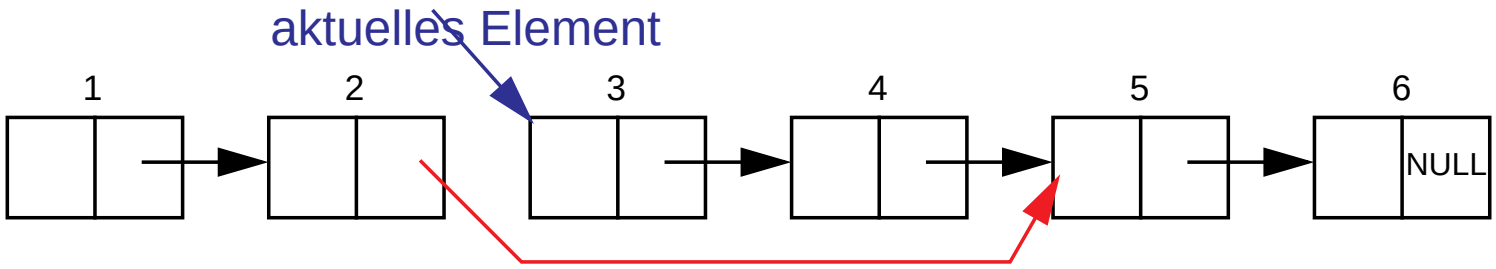
  - nach der Behandlung läuft der Prozess an der unterbrochenen Stelle weiter
- Systemschnittstelle
  - sigaction – Anmelden einer Funktion = Einrichten der ISR-Tabelle
  - sigprocmask – Blockieren/Freigeben von Signalen ≈ cli() / sei()
  - sigsuspend – Freigeben + passives Warten auf Signal + wieder Blockieren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



Überblick: Teil D Betriebssystemabstraktionen

Speicherorganisation

```
int a;                          // a: global, uninitialized
int b = 1;                      // b: global, initialized
const int c = 2;                // c: global, const

void main() {
  static int s = 3;             // s: local, static, initialized
  int x, y;                     // x: local, auto; y: local, auto
  char* p = malloc( 100 );      // p: local, auto; *p: heap (100 byte)
}
```

Wo kommt der Speicher für diese Variablen her?

- **Statische Allokation** – Reservierung beim Übersetzen / Linken
  - Betrifft globale und modullokale Variablen, sowie den Code
  - Allokation durch Platzierung in einer Sektion

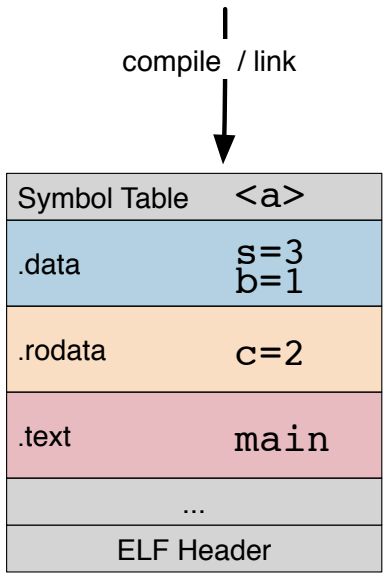
.text	– enthält den Programmcode	`main()`
.bss	– enthält alle uninitialisierten / mit 0 initialisierten Variablen	`a`
.data	– enthält alle initialisierten Variablen	`b,s`
.rodata	– enthält alle initialisierten unveränderlichen Variablen	`c`

- **Dynamische Allokation** – Reservierung zur Laufzeit
  - Betrifft lokale Variablen und explizit angeforderten Speicher

Stack	– enthält alle aktuell gültigen lokalen Variablen	`x,y,p`
Heap	– enthält explizit mit `malloc()` angeforderte Speicherbereiche	`*p`

Speicherorganisation auf einem μC

```c
int a;                          // a: global, uninitialized
int b = 1;                      // b: global, initialized
const int c = 2;                // c: global, const

void main() {
  static int s = 3;             // s: local, static, initialized
  int x, y;                     // x: local, auto; y: local, auto
  char* p = malloc( 100 );      // p: local, auto; *p: heap (100 byte)
}
```



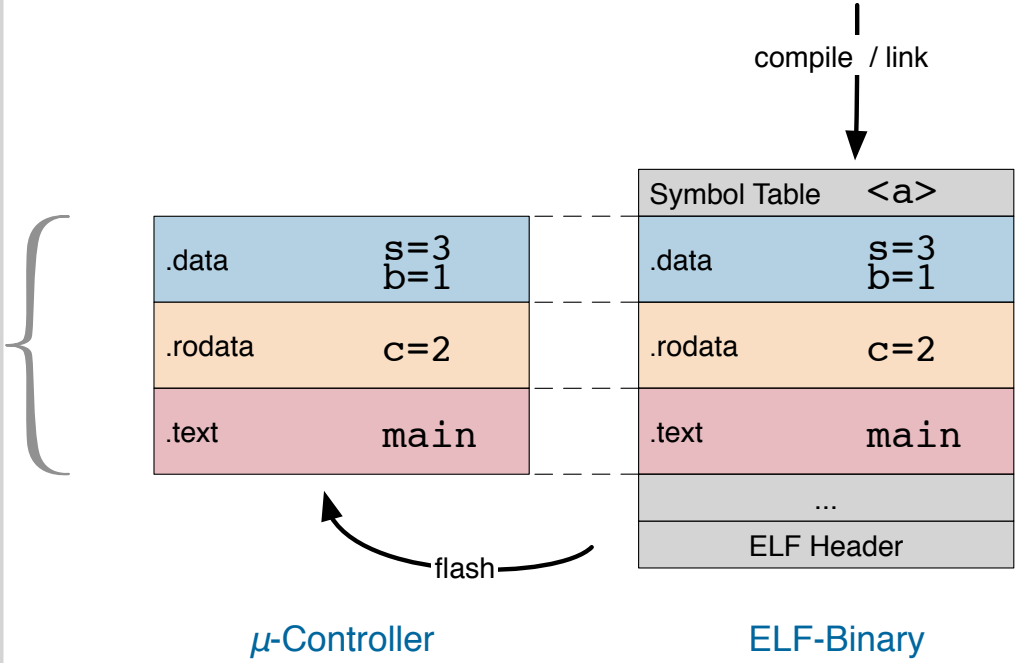
Quellprogramm

ELF-Binary

Beim Übersetzen und Linken werden die Programmelemente in entsprechenden Sektionen der ELF-Datei zusammen gefasst. Informationen zur Größe der `.bss`-Sektion landen ebenfalls in `.rodata`.

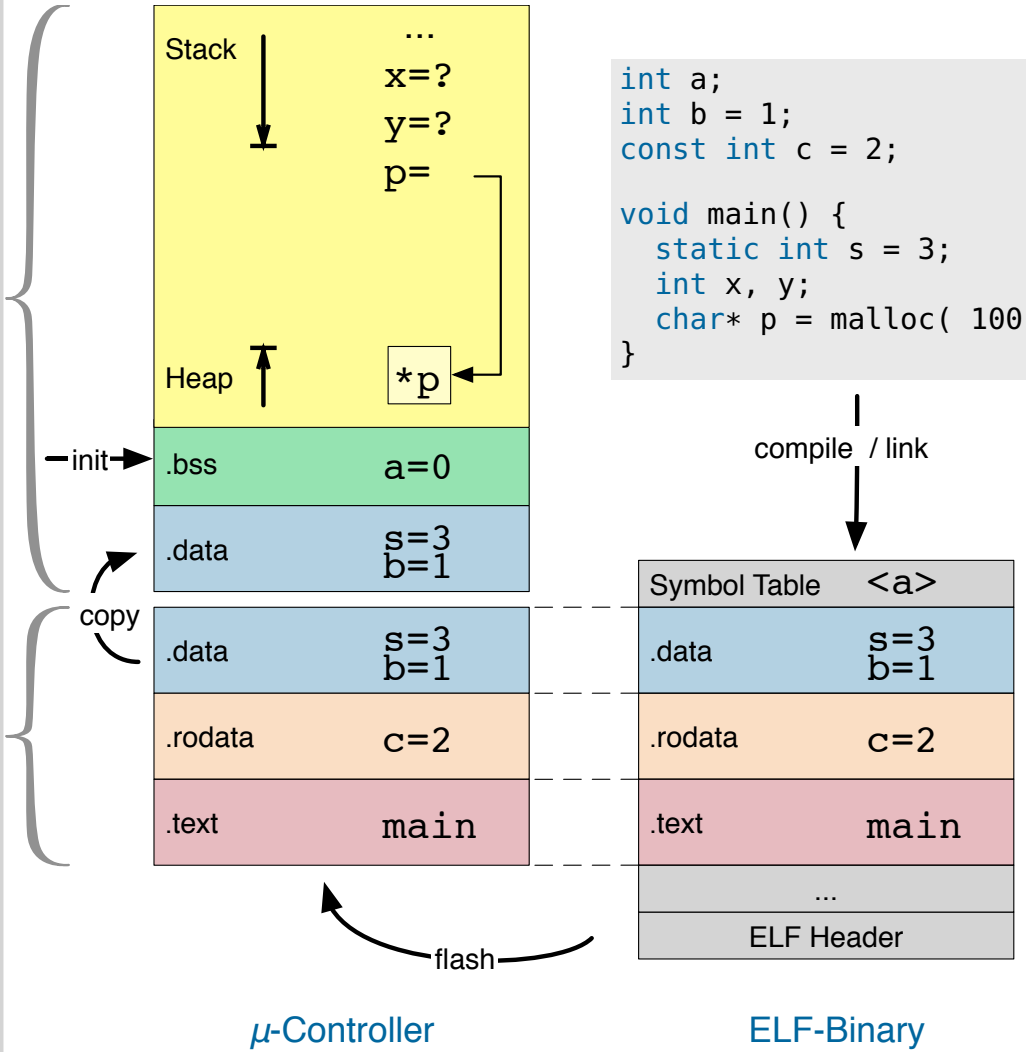
Speicherorganisation auf einem μC

```c
int a;                          // a: global, uninitialized
int b = 1;                      // b: global, initialized
const int c = 2;                // c: global, const

void main() {
  static int s = 3;             // s: local, static, initialized
  int x, y;                     // x: local, auto; y: local, auto
  char* p = malloc( 100 );      // p: local, auto; *p: heap (100 byte)
}
```



Zur Installation auf dem μC werden `.text` und `.[ro]data` in den Flash-Speicher des μC geladen.

Speicherorganisation auf einem μC




```c
int a;                          // a: global, uninitialized
int b = 1;                      // b: global, initialized
const int c = 2;                // c: global, const

void main() {
  static int s = 3;             // s: local, static, initialized
  int x, y;                     // x: local, auto; y: local, auto
  char* p = malloc( 100 );      // p: local, auto; *p: heap (100 byte)
}
```

Quellprogramm

Beim Systemstart wird das `.bss`-Segment im RAM angelegt und mit `0` initialisiert, das `.data`-Segment wird aus dem Flash ins RAM kopiert.
Das verbleibende RAM wird für den Stack und (falls vorhanden) den Heap verwendet.

Verfügt die Architektur über keinen Daten-Flashspeicher (beim ATmega der Fall ↪ 14–3), so werden konstante Variablen ebenfalls in `.data` abgelegt (und belegen zur Laufzeit RAM).

Speicherorganisation in einem UNIX-Prozess

- **Programm:** Folge von Anweisungen
- **Prozess:** Betriebssystemkonzept zur Ausführung von Programmen
  - Programm, das sich in Ausführung befindet, und seine Daten (Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
  - Eine konkrete Ausführungsumgebung für ein Programm (Prozessor, Speicher, ...) $\mapsto$ vom Betriebssystem verwalteter *virtueller Computer*
- Jeder Prozess bekommt einen virtuellen Adressraum zugeteilt
  - 4 GB auf einem 32-Bit-System, davon bis zu 3 GB für die Anwendung
    - In das verbleibende GB werden Betriebssystem und *memory-mapped* Hardware (z. B. PCI-Geräte) eingeblendet
    - Daten des Betriebssystems werden durch Zugriffsrechte geschützt
  - Zugriff auf andere Prozesse ist nur über das Betriebssystem möglich
  - Virtueller Speicher wird durch das Betriebssystem auf physikalischen (Hintergrund-)Speicher abgebildet

Speicherorganisation in einem UNIX-Prozess (Forts.)

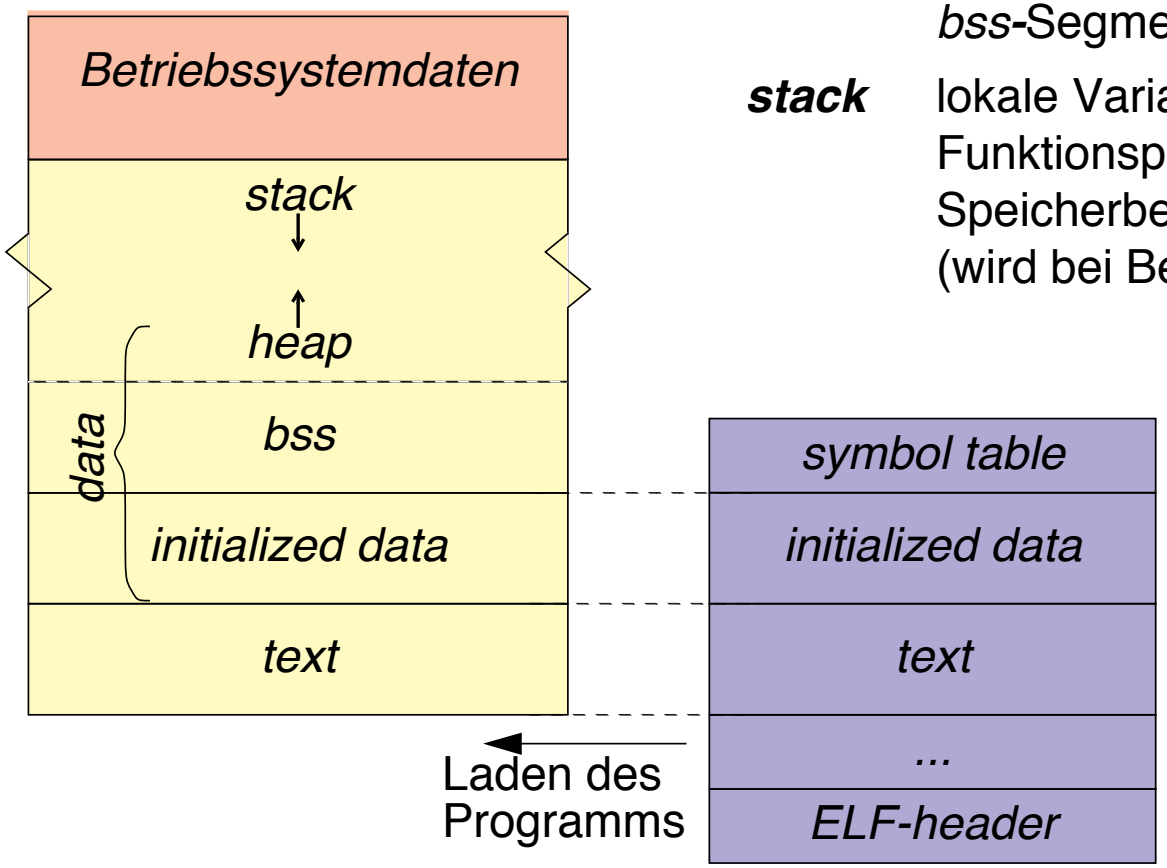
text Programmkode

data globale und static Variablen

bss nicht initialisierte globale und *static* Variablen (wird vor der Vergabe an den Prozess mit 0 vorbelegt)

heap dynamische Erweiterungen des *bss*-Segments (***sbrk(2)***, ***malloc(3)***)

stack lokale Variablen, Funktionsparameter, Speicherbereiche für Registerinhalte, (wird bei Bedarf dynamisch erweitert)

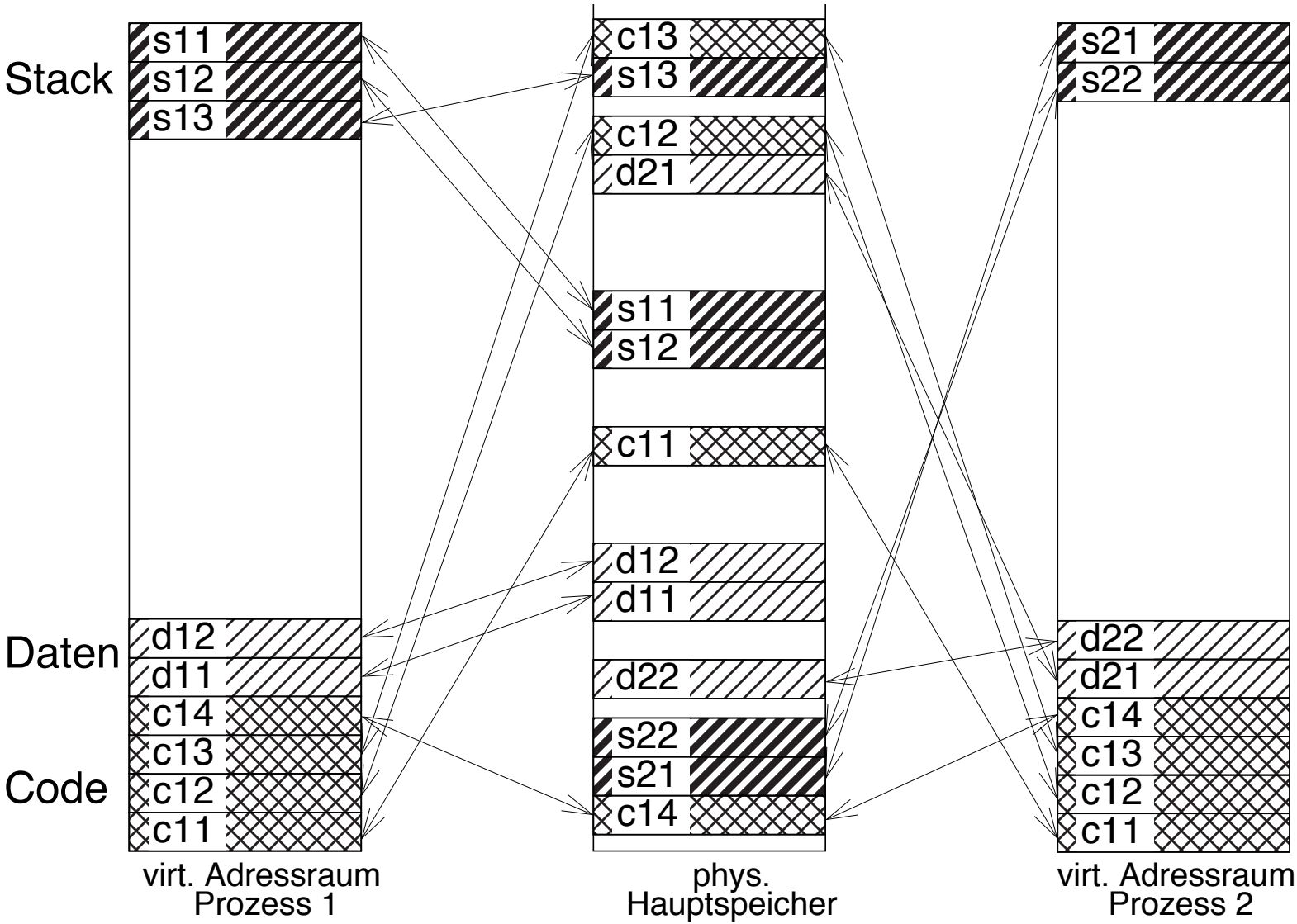


Aufbau eines Programms (ELF-Format)

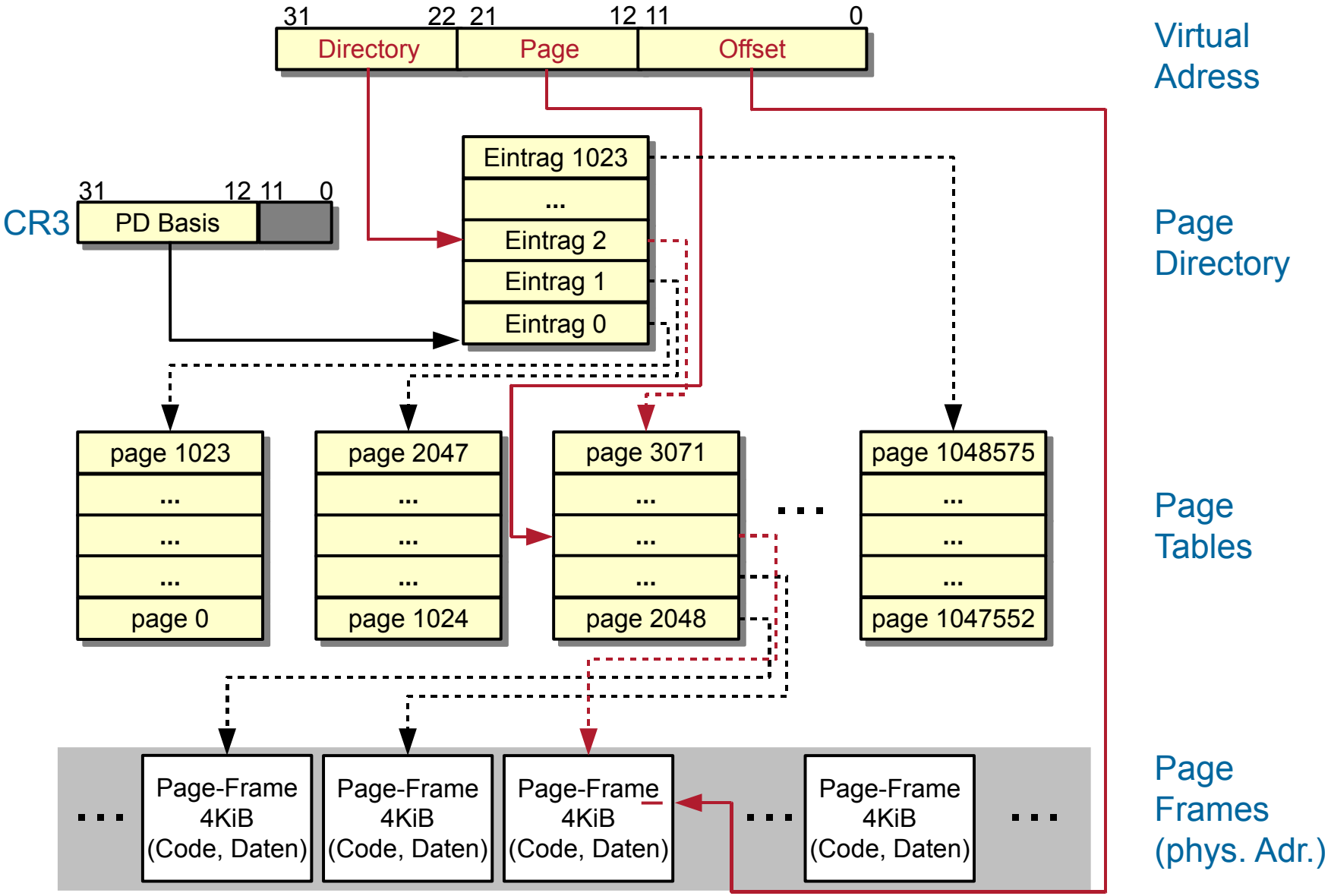
Seitenbasierte Speicherverwaltung

- Die Abbildung von virtuellem Speicher (*VS*) auf physikalischen Speicher (*PS*) erfolgt durch Seitenaddressierung (*Paging*)
  - *VS* eines Prozesses ist unterteilt in Speicherseiten (*Memory Pages*)
    - kleine Adressblöcke, üblich sind z. B. 4 KiB und 4 MiB Seiten
    - in dieser Granularität wird Speicher vom Betriebssystem zugewiesen
  - *PS* ist analog unterteilt in Speicherrahmen (*Page Frames*)
  - Abbildung: *Seite* $\mapsto$ *Rahmen* über eine Seitentabelle (*Page Table*)
    - Umrechnung *VS* auf *PS* bei jedem Speicherzugriff
    - Hardwareunterstützung durch MMU (*Memory Management Unit*)
    - Betriebssystem kann Seiten auf den Hintergrundspeicher auslagern
    - Abbildung ist nicht linkseindeutig: Seiten aus mehreren Prozesse können auf denselben Rahmen verweisen (z. B. gemeinsamer Programmcode)
- Seitenbasierte Speicherverwaltung ist auch ein Schutzkonzept
  - Seiten sind mit Zugriffsrechten versehen: *Read*, *Read–Write*, *Execute*
  - MMU überprüft bei der Umrechnung, ob der Zugriff erlaubt ist

Seitenbasierte Speicherverwaltung



Seitenbasierte Speicherverwaltung



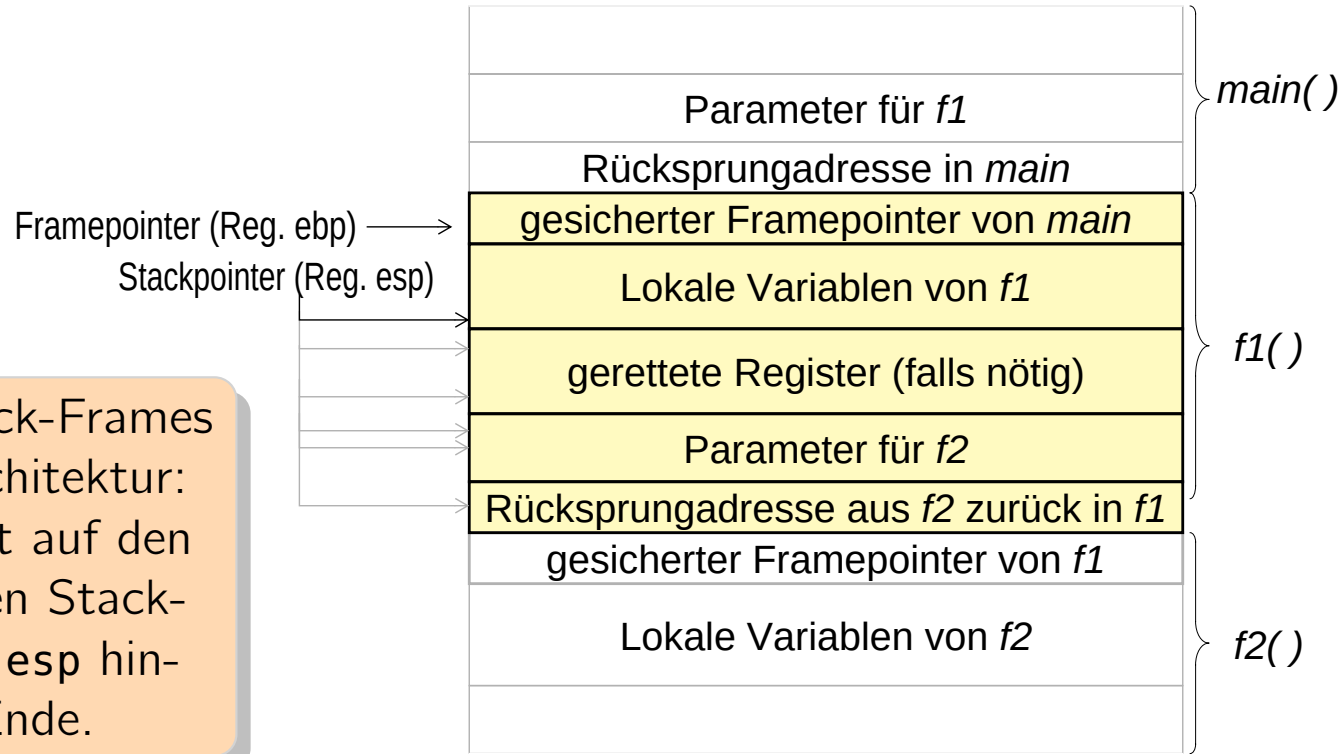
Dynamische Speicherallokation: Heap

- **Heap** := Vom Programm explizit verwalteter RAM-Speicher
  - Lebensdauer ist unabhängig von der Programmstruktur
- Anforderung und Wiederfreigabe über zwei Basisoperationen
  - `void* malloc( size_t n )` fordert einen Speicherblock der Größe n an; Rückgabe bei Fehler: 0-Zeiger (`NULL`)
  - `void free( void* pmem )` gibt einen zuvor mit `malloc()` angeforderten Speicherblock vollständig wieder frei
- Beispiel

```
#include <stdlib.h>
int* intArray( uint16_t n ) {    // alloc int[n] array
  return (int*) malloc( n * sizeof int );
}
void main() {
  int* array  = intArray(100);   // alloc memory for 100 ints
  if( array ) {                  // malloc() returns NULL on failure
    ...                          // if succeeded, use array
    array[99]   = 4711;
    ...
    free( array );               // free allocated block (** IMPORTANT! **)
  }
}
```


Dynamische Speicherallokation: Stack [↪ GDI, V-50]

- Lokale Variablen, Funktionsparameter und Rücksprungadressen werden vom Übersetzer auf dem Stack (Stapel, Keller) verwaltet
  - Prozessorregister `[e]sp` zeigt immer auf den nächsten freien Eintrag
  - Stack „wächst“ (architekturabhängig) „von oben nach unten“
- Die Verwaltung erfolgt in Form von Stack-Frames

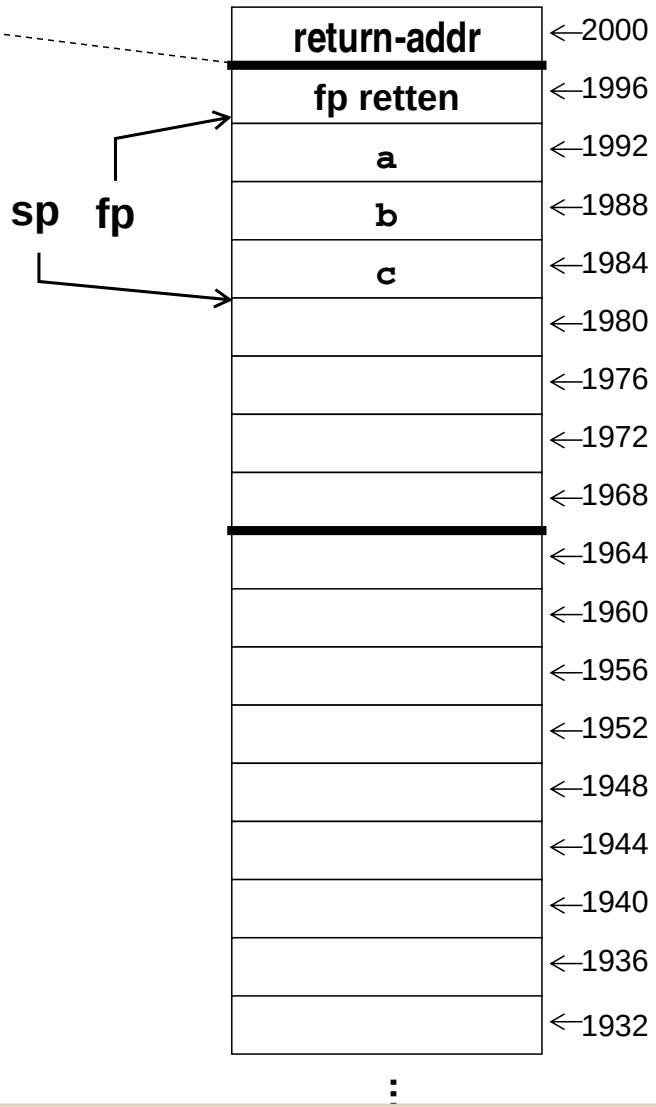


Aufbau eines Stack-Frames auf der IA-32-Architektur: Register **ebp** zeigt auf den Beginn des aktiven Stack-Frames; Register **esp** hinter das aktuelle Ende.

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

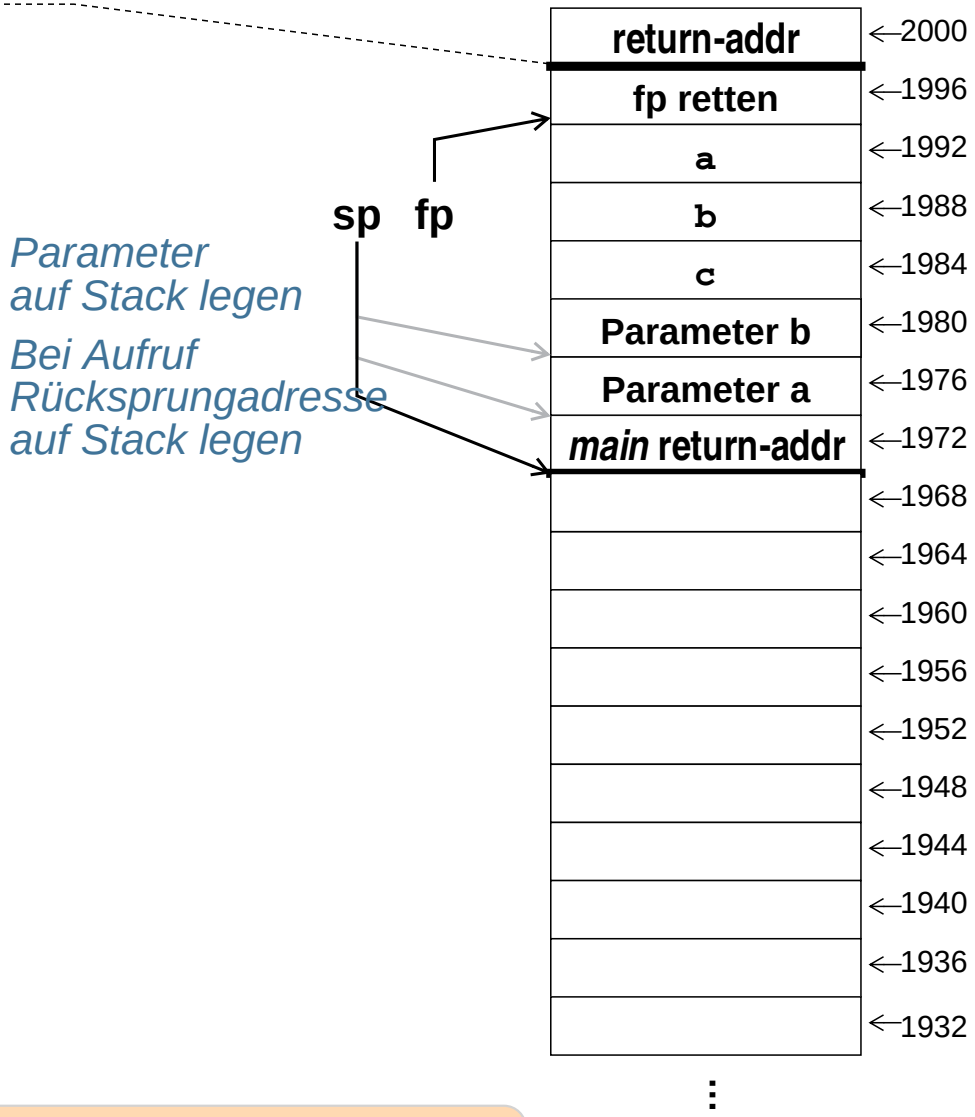
Stack-Frame für main erstellen
&a = fp-4
&b = fp-8
&c = fp-12



Beispiel hier für 32-Bit-Architektur (4-Byte `int`s), `main()` wurde soeben betreten

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

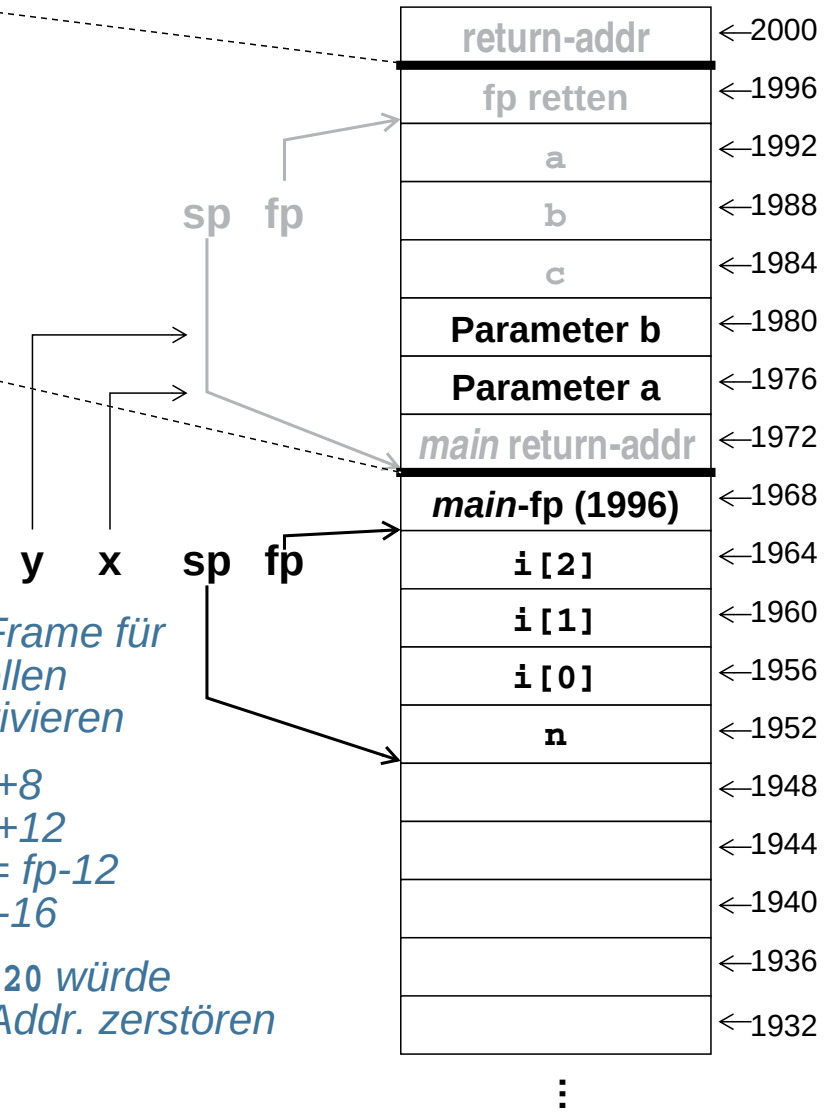


`main()` bereitet den Aufruf von `f1(int, int)` vor

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



Stack-Frame für f1 erstellen und aktivieren

&x = fp+8
&y = fp+12
&(i[0]) = fp-12
&n = fp-16

`i[4] = 20` *würde return-Addr. zerstören*

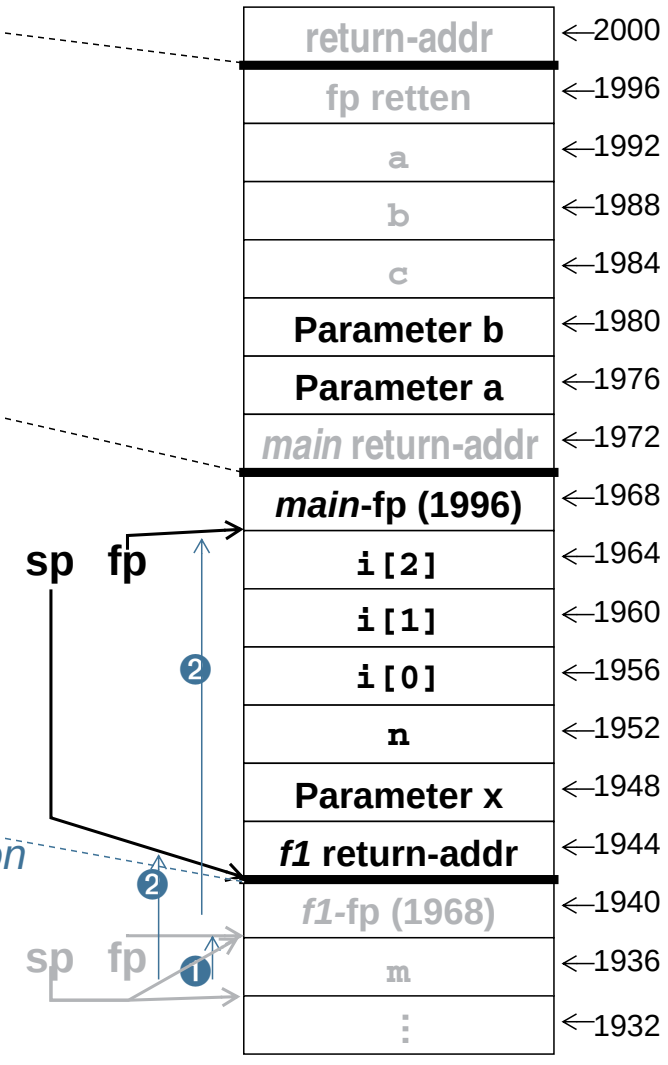
`f1()` wurde soeben betreten

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```

Stack-Frame von f2 abräumen

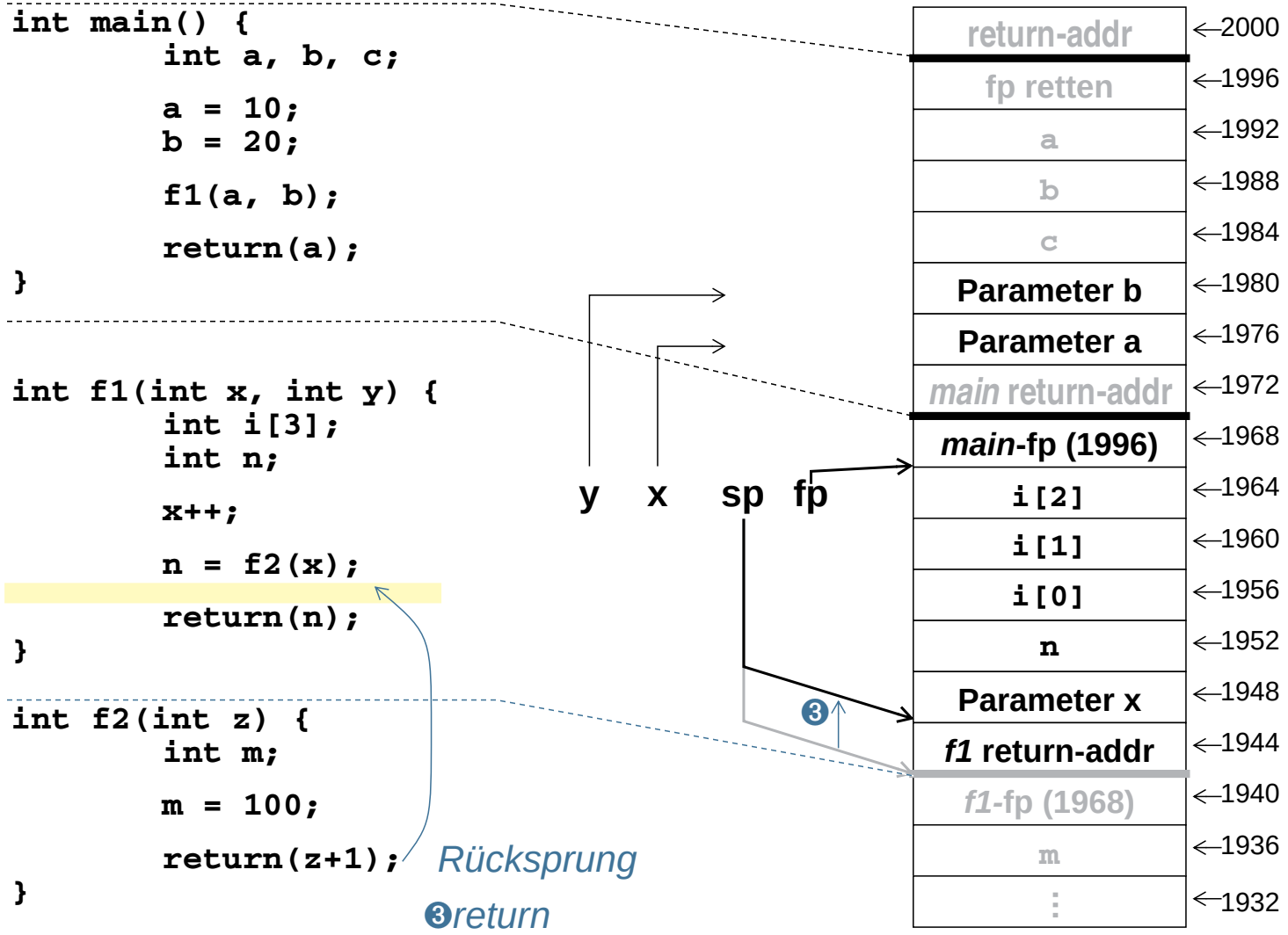
❶ *sp = fp*
❷ *fp = pop(sp)*



f2() bereitet die Terminierung vor (wurde von f1() aufgerufen und ausgeführt)

Stack-Aufbau bei Funktionsaufrufen

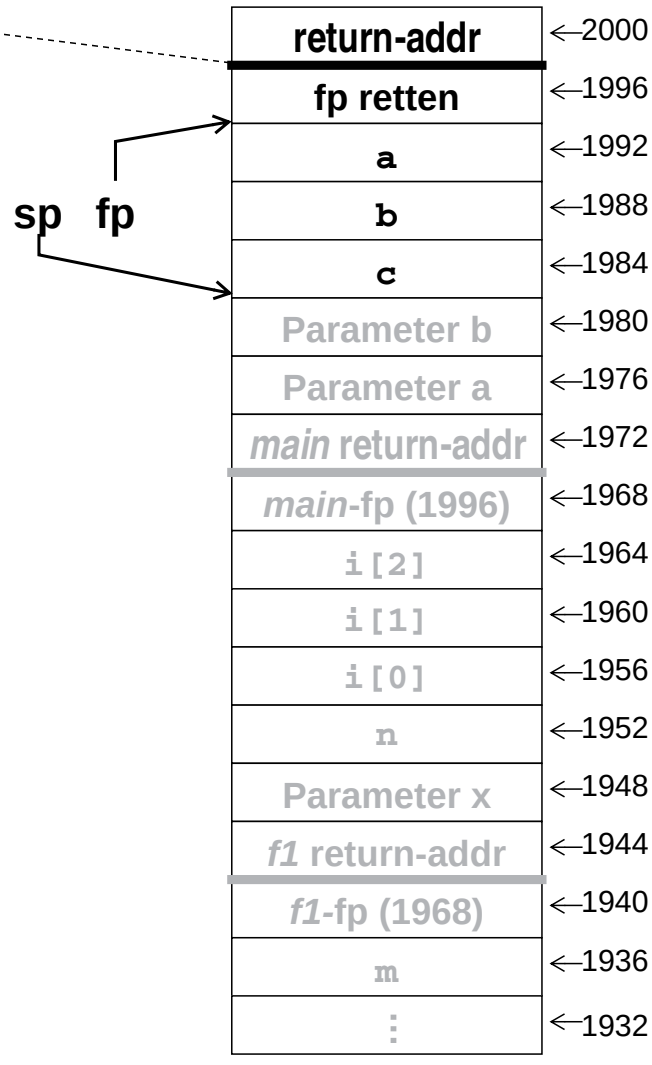
```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```



f2() wird verlassen

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

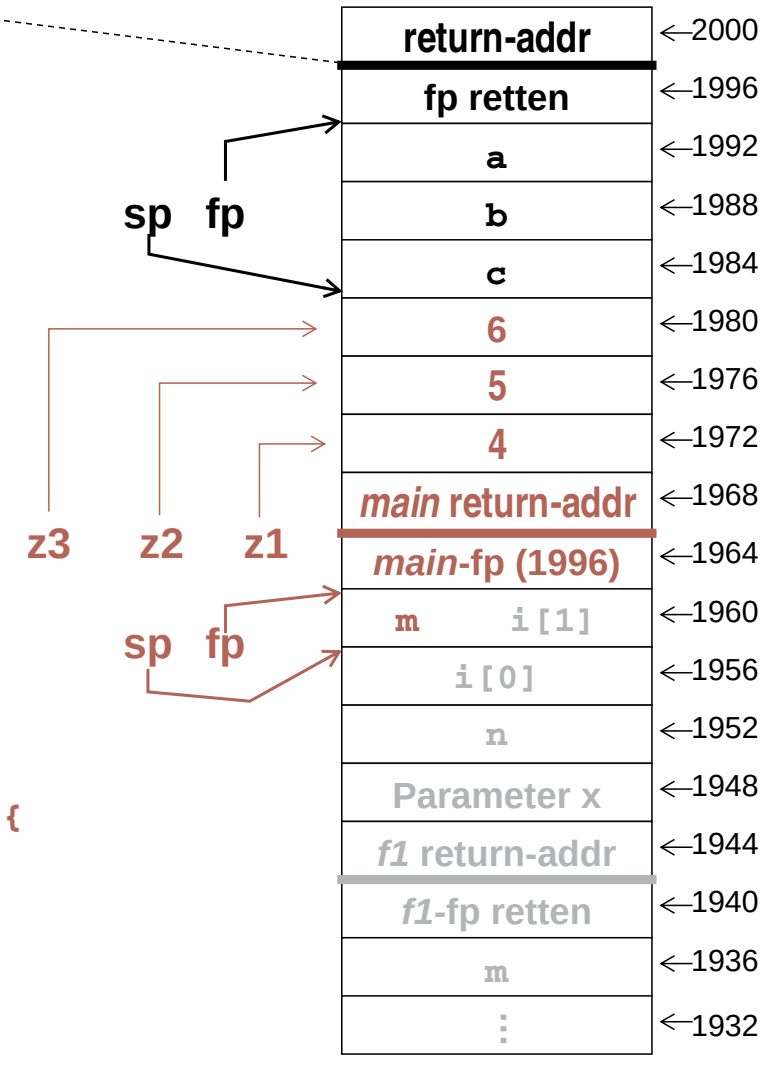


zurück in `main()`

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        f3(4,5,6);
}
```

was wäre, wenn man nach f1 jetzt eine Funktion f3 aufrufen würde?




```
int f3(int z1, int z2, int z3) {
        int m;

        return(m);
}
```

m wird nicht initialisiert ⇝„erbt“ alten Wert vom Stapel

Statische versus dynamische Allokation

- Bei der **μC-Entwicklung** wird statische Allokation bevorzugt
  - **Vorteil:** Speicherplatzbedarf ist bereits nach dem Übersetzen / Linken exakt bekannt (kann z. B. mit `size` ausgegeben werden)
  - Speicherprobleme frühzeitig erkennbar (Speicher ist knapp! ↪ 1–3)

```
lohmann@faui48a:$ size sections.avr
text    data    bss    dec    hex filename
682     10      6      698    2ba sections.avr
```

Sektionsgrößen des Programms von ↪ 20–1

- ⇝ Speicher möglichst durch `static`-Variablen anfordern
  - Regel der geringstmöglichen Sichtbarkeit beachten ↪ 12–6
  - Regel der geringstmöglichen Lebensdauer „sinnvoll“ anwenden
- Ein Heap ist verhältnismäßig teuer ⇝ wird möglichst vermieden
  - Zusätzliche Speicherkosten durch Verwaltungsstrukturen und Code
  - Speicherbedarf zur Laufzeit schlecht abschätzbar
  - Risiko von Programmierfehlern und Speicherlecks

Statische versus dynamische Allokation (Forts.)

- Bei der Entwicklung für eine **Betriebssystemplattform** ist dynamische Allokation hingegen sinnvoll
  - **Vorteil:** Dynamische Anpassung an die Größe der Eingabedaten (z. B. bei Strings)
  - Reduktion der Gefahr von *Buffer-Overflow*-Angriffen

⇝ Speicher für Eingabedaten möglichst auf dem Heap anfordern

- Das Risiko von Programmierfehlern und Speicherlecks bleibt!