

Überblick: Teil D Betriebssystemabstraktionen

15 Programmausführung, Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme I

18 Dateisysteme

19 Prozesse I

20 Speicherorganisation

21 Prozesse II

22 Betriebssysteme II

V_SPIC_handout



Prozessor

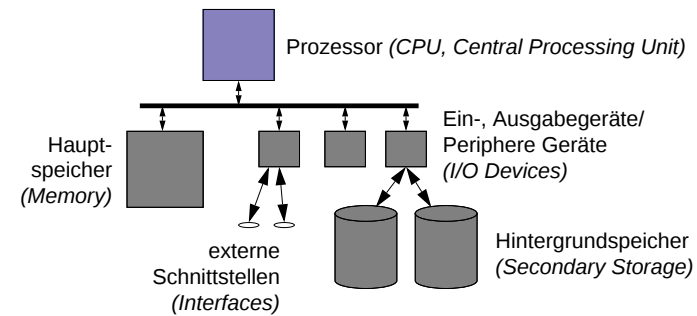
- Register
 - Prozessor besitzt Steuer- und Vielzweckregister
 - Steuerregister:
 - Programmzähler (*Instruction Pointer*)
 - Stapelregister (*Stack Pointer*)
 - Statusregister
 - etc.
- Programmzähler enthält Speicherstelle der nächsten Instruktion
 - Instruktion wird geladen und
 - ausgeführt
 - Programmzähler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

K11.pdf: 2011-07-05



19 Prozesse

■ Einordnung



K11.pdf: 2011-07-05



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
```

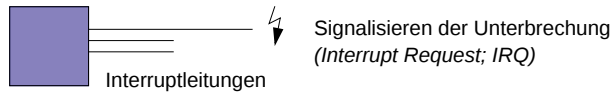
- Prozessor arbeitet in einem bestimmten Modus
 - Benutzermodus: eingeschränkter Befehlssatz
 - privilegierter Modus: erlaubt Ausführung privilegierter Befehle
 - Konfigurationsänderungen des Prozessors
 - Moduswechsel
 - spezielle Ein-, Ausgabebefehle

K11.pdf: 2011-07-05



Prozessor (3)

■ Unterbrechungen (*Interrupts*)



- 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

K1.pdf: 2011-07-05



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)

K1.pdf: 2011-07-05



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

K1.pdf: 2011-07-05



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

K1.pdf: 2011-07-05



Prozesse (2)

- Mehrprogrammbetrieb
 - mehrere Prozesse können quasi gleichzeitig ausgeführt werden
 - steht nur ein echter Prozessor zur Verfügung, werden Zeiteile 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)

K1.pdf: 2011-07-05



Prozesszustände

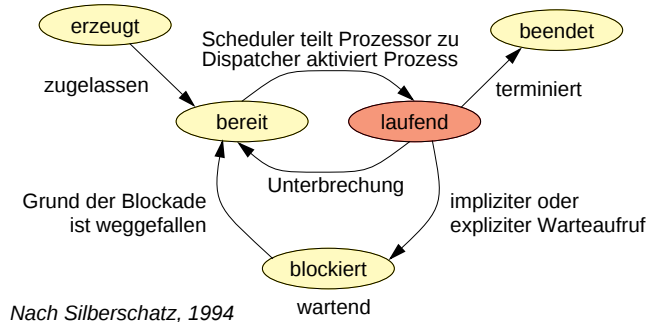
- Ein Prozess befindet sich in einem der folgenden Zustände:
 - **Erzeugt (New)**
Prozess wurde erzeugt, besitzt aber noch nicht alle nötigen Betriebsmittel
 - **Bereit (Ready)**
Prozess besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen
 - **Laufend (Running)**
Prozess wird vom realen Prozessor 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

K1.pdf: 2011-07-05



Prozesszustände (2)

- Zustandsdiagramm

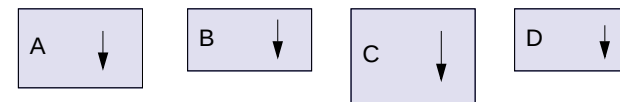


K1.pdf: 2011-07-05



Prozesswechsel

- Konzeptionelles Modell



vier Prozesse mit eigenständigen Befehlszählern

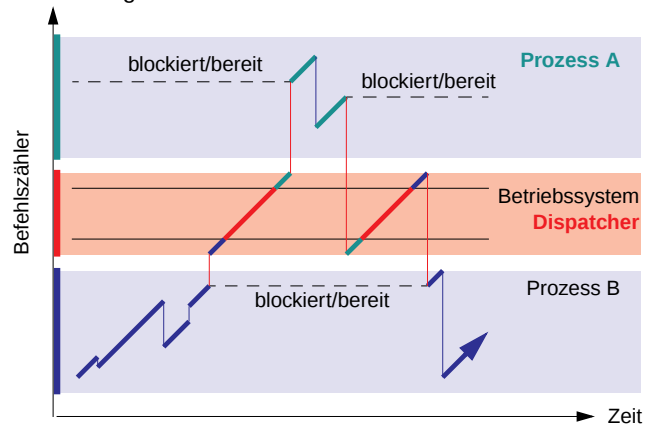
- Umschaltung (**Context Switch**)
 - Sichern der Register des laufenden Prozesses inkl. Programmzähler (Kontext),
 - Auswahl des neuen Prozesses,
 - Ablaufumgebung des neuen Prozesses herstellen (z.B. Speicherabbildung, etc.),
 - gesicherte Register des neuen Prozesses laden und
 - Prozessor aufsetzen.

K1.pdf: 2011-07-05



Prozesswechsel (2)

■ Umschaltung



K1.pdf: 2011-07-05



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
- ...

K1.pdf: 2011-07-05



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 */
}
```

K1.pdf: 2011-07-05



Prozesserzeugung (UNIX)

■ Erzeugen eines neuen UNIX-Prozesses

- Duplizieren des gerade laufenden Prozesses

`pid_t fork( void );`

<pre>pid_t p; Vater ... p= fork(); if(p == (pid_t)0) { /* child */ ... } else if(p!=(pid_t)-1) { /* parent */ ... } else { /* error */ }</pre>		<pre>pid_t p; Kind ... p= fork(); if(p == (pid_t)0) { /* child */ ... } else if(p!=(pid_t)-1) { /* parent */ ... } else { /* error */ }</pre>
---	--	--

K1.pdf: 2011-07-05



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

K1.pdf: 2011-07-05



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 );  
...
```

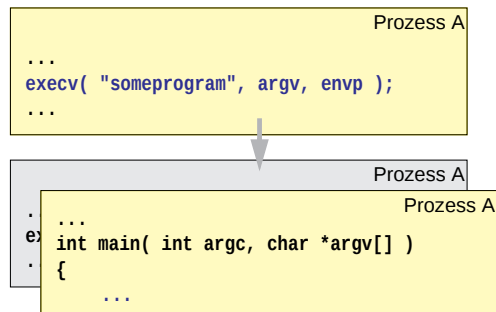
K1.pdf: 2011-07-05



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.

K1.pdf: 2011-07-05



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 );
```

K1.pdf: 2011-07-05



Signale

Signalisierung des Systemkerns an einen Prozess

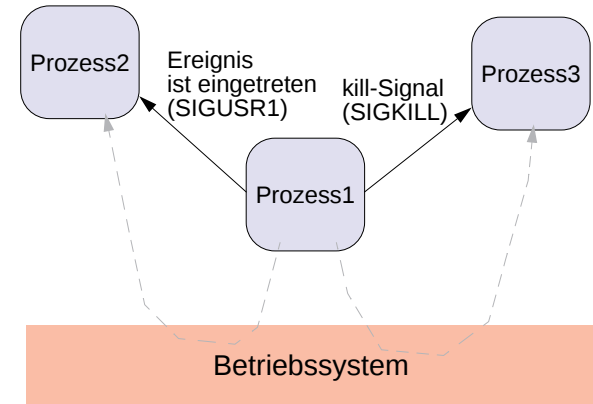
- Software-Implementierung der Hardware-Konzepte
 - **Interrupt:** asynchrones Signal aufgrund eines "externen" Ereignisses
 - CTRL-C auf der Tastatur gedrückt (Interrupt-Signal)
 - Timer abgelaufen
 - Kind-Prozess terminiert
 - ...
 - **Trap:** synchrones Signal, ausgelöst durch die Aktivität des Prozesses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

K1.pdf: 2011-07-05



Kommunikation zwischen Prozessen

- ein Prozess will einem anderen ein Ereignis signalisieren



K1.pdf: 2011-07-05



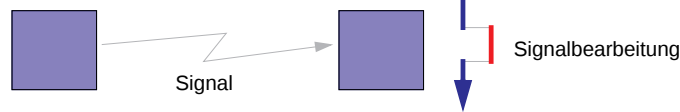
Reaktion auf Signale

- 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

K1.pdf: 2011-07-05



POSIX Signalbehandlung

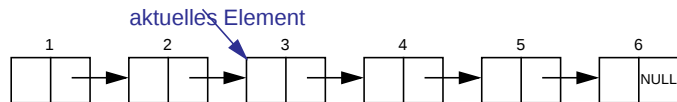
- Betriebssystemschnittstelle zum Umgang mit Signalen
- Signal bewirkt Aufruf einer Funktion (analog ISR)
- nach der Behandlung läuft der Prozess an der unterbrochenen Stelle weiter
- Systemschnittstelle
 - sigaction – Anmelden einer Funktion = Einrichten der ISR-Tabelle
 - sigprocmask – Blockieren/Freigeben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

K1.pdf: 2011-07-05

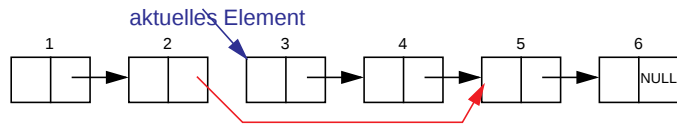


Signale und Nebenläufigkeit → Race Conditions

- 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



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
 - .bss – enthält alle uninitialisierten / mit 0 initialisierten Variablen
 - .data – enthält alle initialisierten Variablen
 - .rodata – enthält alle initialisierten unveränderlichen Variablen

```
main()
a
b,s
c
```

- **Dynamische Allokation** – Reservierung zur Laufzeit
 - Betrifft lokale Variablen und explizit angeforderten Speicher
 - Stack – enthält alle **aktuell gültigen** lokalen Variablen
 - Heap – enthält explizit mit `malloc()` angeforderte Speicherbereiche

```
x,y,p
*p
```

Überblick: Teil D Betriebssystemabstraktionen

- 15 Programmausführung, Nebenläufigkeit
- 16 Ergänzungen zur Einführung in C
- 17 Betriebssysteme I
- 18 Dateisysteme
- 19 Prozesse I
- 20 Speicherorganisation
- 21 Prozesse II
- 22 Betriebssysteme II

Speicherorganisation auf einem μ 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)
}
```

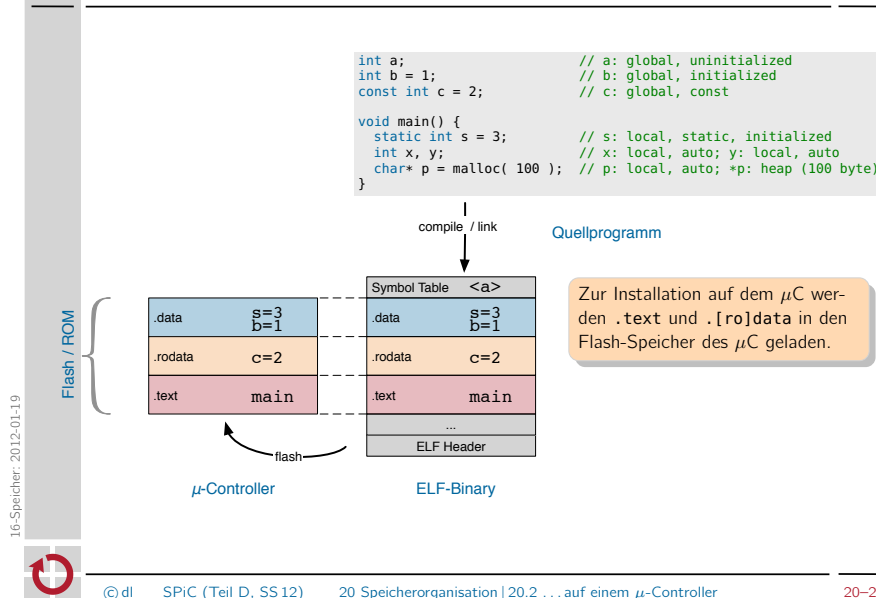
compile / link
Quellprogramm

Symbol Table	<a>
.data	s=3 b=1
.rodata	c=2
.text	main
...	
ELF Header	

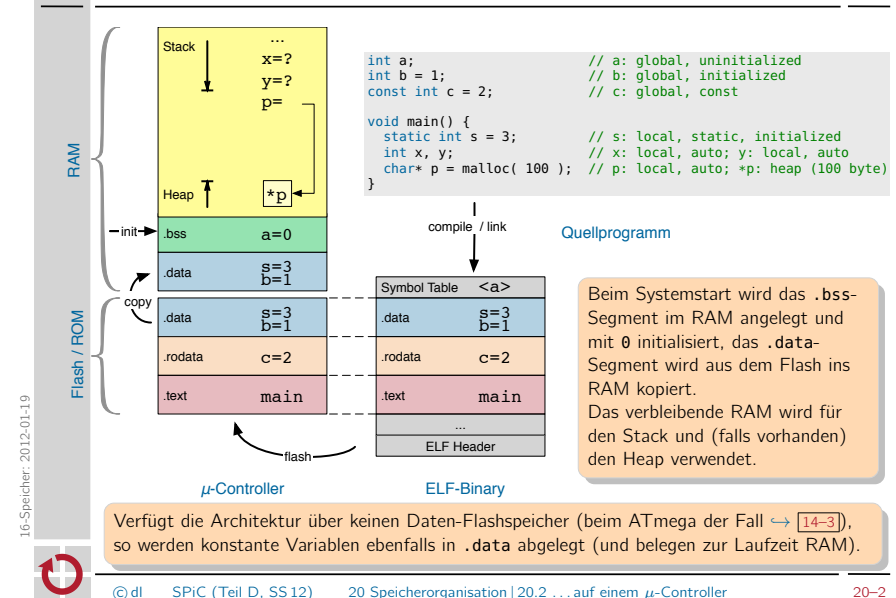
ELF-Binary

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



Speicherorganisation auf einem μC



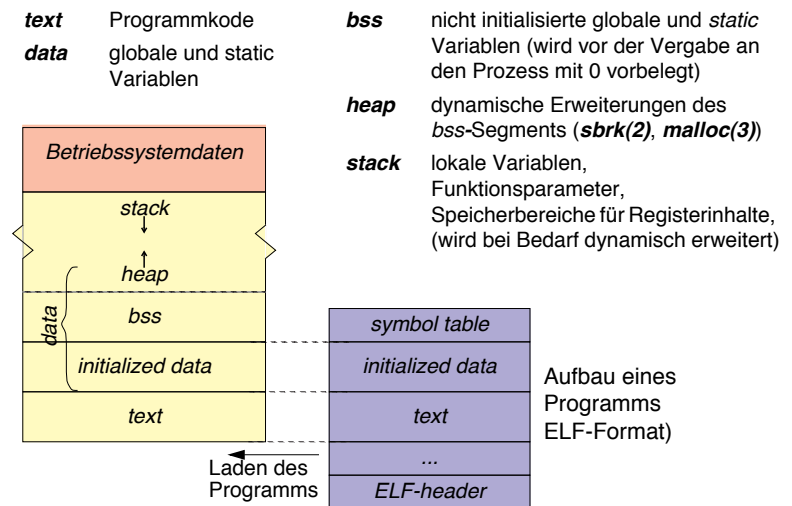
Speicherorganisation in einem UNIX-Prozess

- **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, ...) $\rightarrow$ 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

V_SPiC_handout



Speicherorganisation in einem UNIX-Prozess (Forts.)



V_SPiC_handout

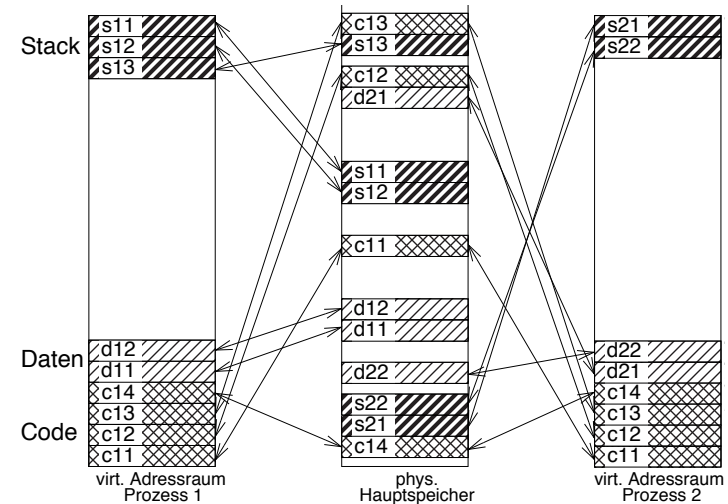


Seitenbasierte Speicherverwaltung

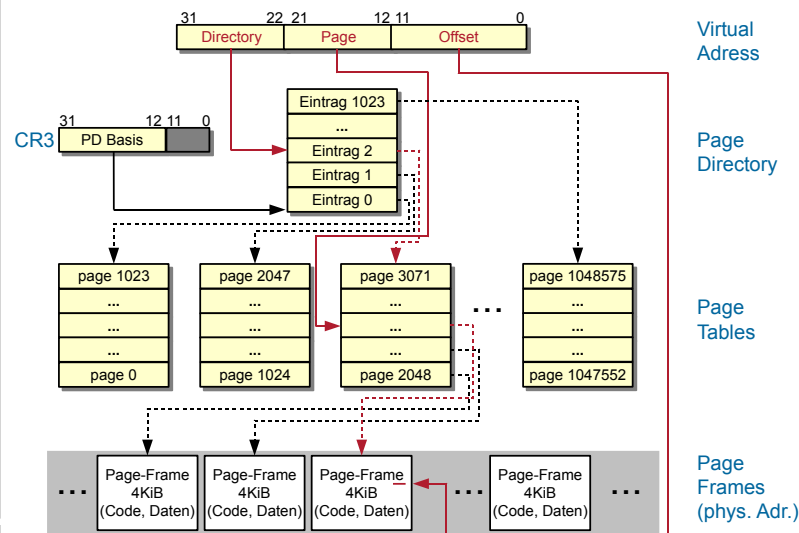
- Die Abbildung von virtuellem Speicher (VS) auf physikalischen Speicher (PS) erfolgt durch **Seitenadressierung** (Paging)
 - VS eines Prozesses ist unterteilt in **Speicherseiten** (Memory Pages)
 - kleine Adressblöcke, üblich sind z. B. 4 KiB und 4 MiB Seiten
 - in dieser Granularität wird Speicher vom Betriebssystem zugewiesen
 - PS ist analog unterteilt in **Speicherrahmen** (Page Frames)
 - Abbildung: *Seite* → *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

Logische Sicht



Seitenbasierte Speicherverwaltung Technische Sicht (IA32)



Dynamische Speicherallokation: Heap

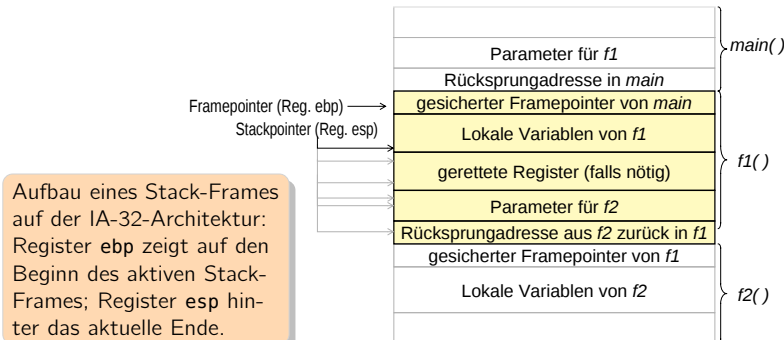
- Heap** := Vom Programm explizit verwalteter RAM-Speicher
 - Lebensdauer ist unabhängig von der Programmstruktur
- Anforderung und Wiederfreigabe über zwei Basisoperationen
 - `void* malloc( size_t n )` fordert einen Speicherblock der Größe *n* an; Rückgabe bei Fehler: 0-Zeiger (NULL)
 - `void free(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

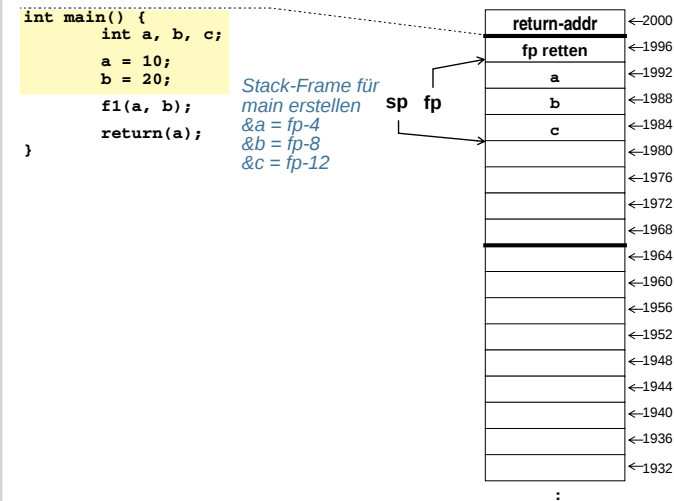
void main() {
    int* array = intArray(100); // alloc memory for 100 ints
    if( array ) { // malloc() returns NULL on failure
        // if succeeded, use array
        array[99] = 4711;
        free( array ); // free allocated block (** IMPORTANT! **)
    }
}
```

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**

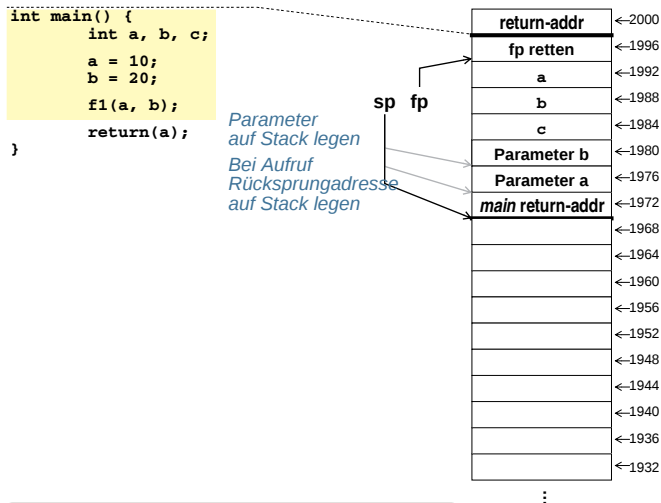


Stack-Aufbau bei Funktionsaufrufen



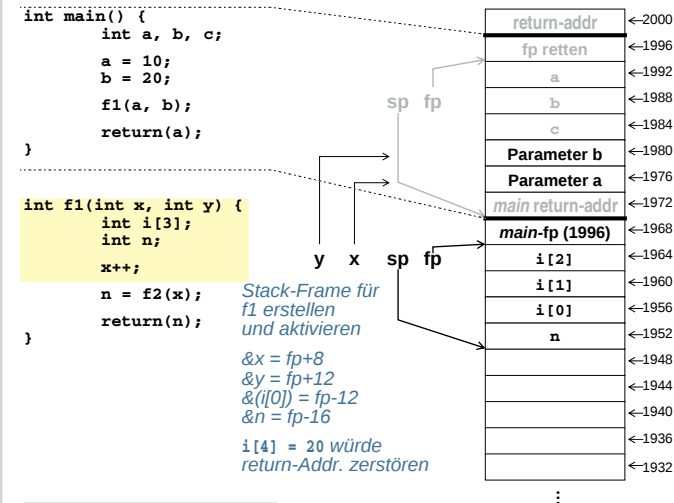
Beispiel hier für 32-Bit-Architektur (4-Byte **ints**), **main()** wurde soeben betreten

Stack-Aufbau bei Funktionsaufrufen



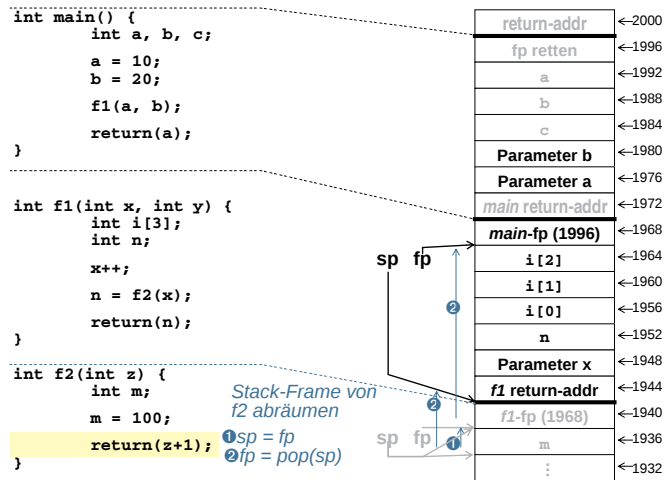
main() bereitet den Aufruf von **f1(int, int)** vor

Stack-Aufbau bei Funktionsaufrufen



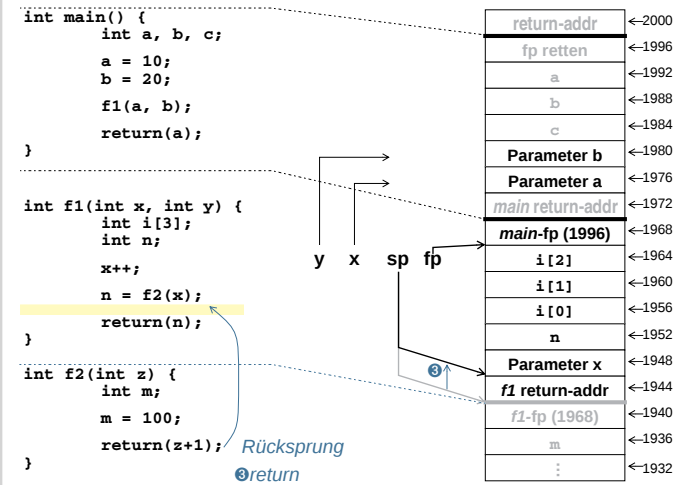
f1() wurde soeben betreten

Stack-Aufbau bei Funktionsaufrufen



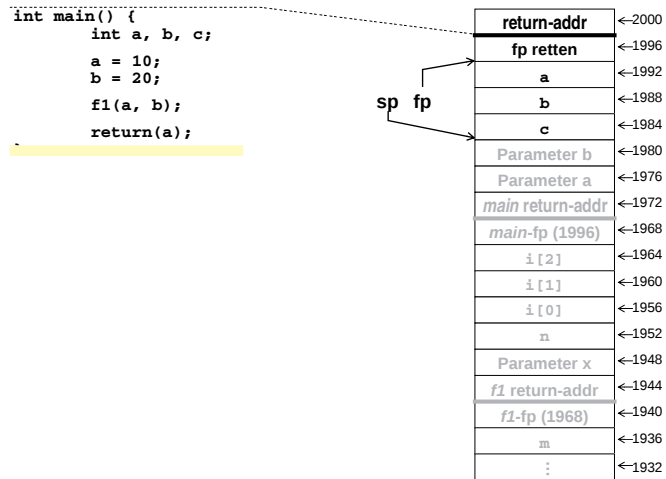
f2() bereitet die Terminierung vor (wurde von f1() aufgerufen und ausgeführt)

Stack-Aufbau bei Funktionsaufrufen



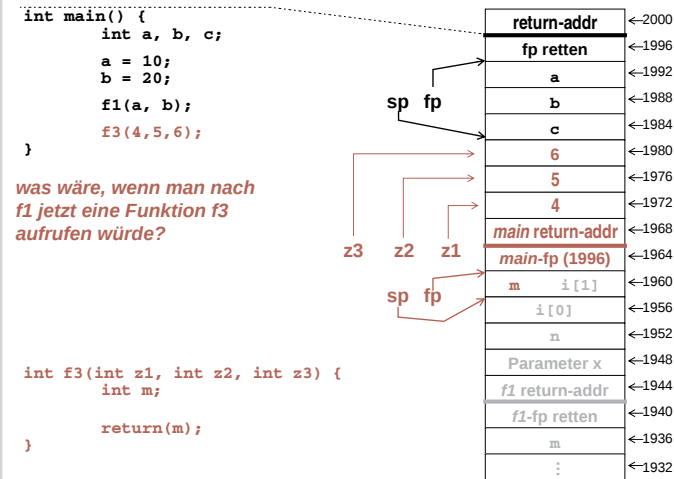
f2() wird verlassen

Stack-Aufbau bei Funktionsaufrufen



zurück in main()

Stack-Aufbau bei Funktionsaufrufen



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@fai48a:~$ 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

16-Speicher: 2012-01-19



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!**

V SPiC handout

