

Überblick: Teil D Betriebssystemabstraktionen

15 Programmausführung, Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme

18 Dateisysteme

19 Programme und Prozesse

20 Speicherorganisation

21 Nebenläufige Prozesse

V_SpIC_handout



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 <code>malloc()</code> angeforderte Speicherbereiche	*p

© dl SPiC (Teil D, SS 13) 20 Speicherorganisation | 20.1 Einführung

20–1

Speicherorganisation auf einem μC

```
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.

16-Speicher: 2013-04-15



© dl SPiC (Teil D, SS 13) 20 Speicherorganisation | 20.2 ... auf einem μ -Controller

20–2

Speicherorganisation auf einem μC

```
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

Flash / ROM

μ -Controller

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

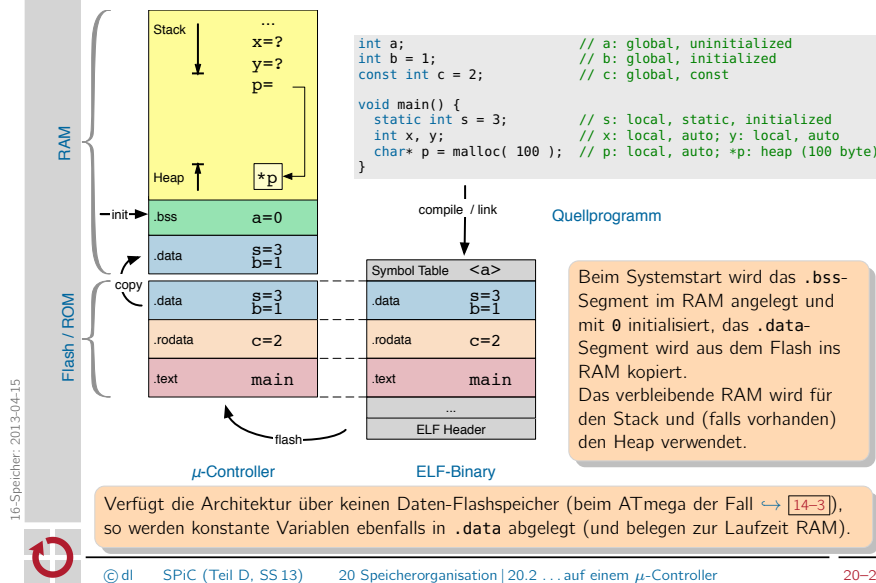
16-Speicher: 2013-04-15



© dl SPiC (Teil D, SS 13) 20 Speicherorganisation | 20.2 ... auf einem μ -Controller

20–2

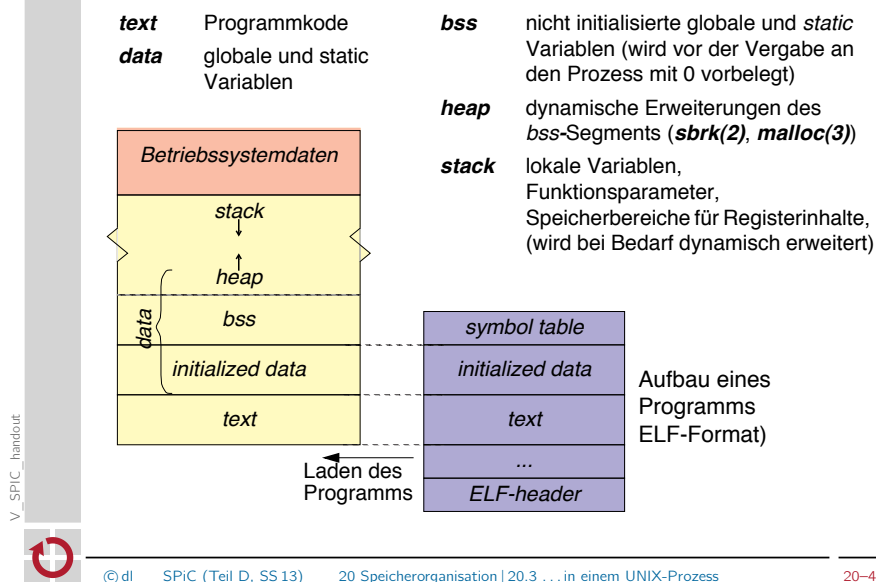
Speicherorganisation auf einem μ C



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

Speicherorganisation in einem UNIX-Prozess (Forts.)

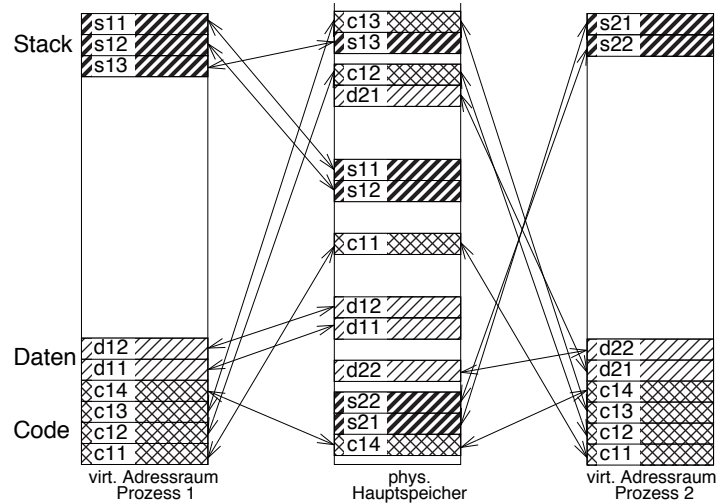


Seitenbasierte Speicherverwaltung

- 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* $\rightarrow$ *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

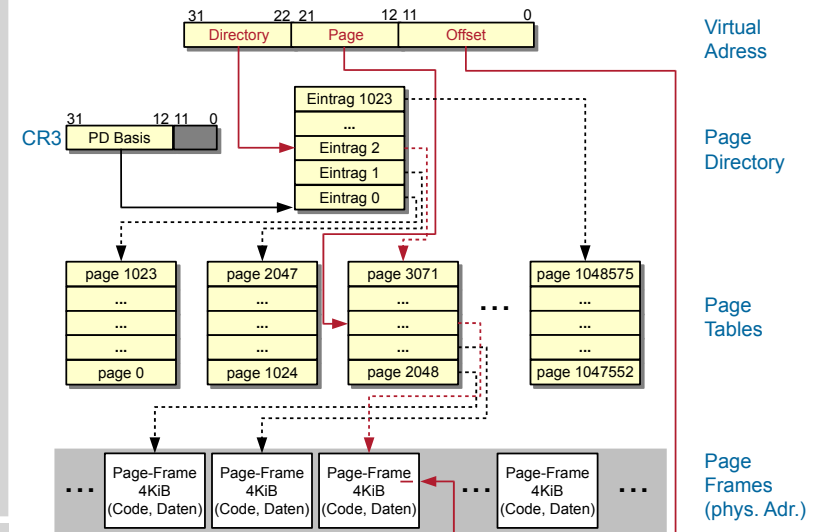
Seitenbasierte Speicherverwaltung

Logische Sicht



Seitenbasierte Speicherverwaltung

Technische Sicht (IA32)



Dynamische Speicherallocation: Heap

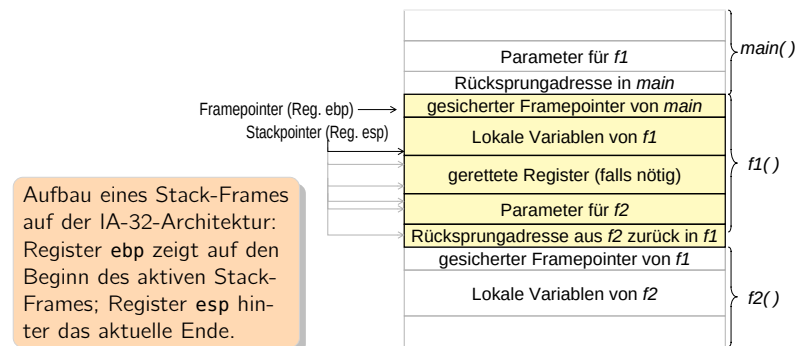
- **Heap** := Vom Programm explizit verwalteter RAM-Speicher
 - Lebensdauer ist unabhängig von der Programmstruktur
- Anforderung und Wiederfreigabe über zwei Basisoperationen
 - `void* malloc( size_t n )` fordert einen Speicherblock der Größe `n` an; Rückgabe bei Fehler: 0-Zeiger (**NULL**)
 - `void 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 Speicherallocation: Stack [↔ GDI, 23-04]

- 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**

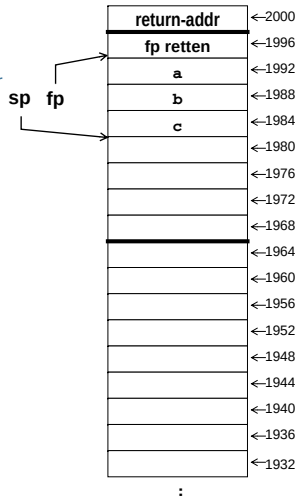


Aufbau eines Stack-Frames auf der IA-32-Architektur: Register `ebp` zeigt auf den Beginn des aktiven Stack-Frames; Register `esp` hinter das aktuelle Ende.

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

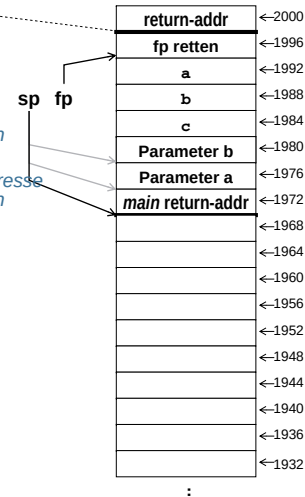


Beispiel hier für 32-Bit-Architektur (4-Byte ints), main() wurde soeben betreten

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

Parameter
auf Stack legen
Bei Aufruf
Rücksprungadresse
auf Stack legen



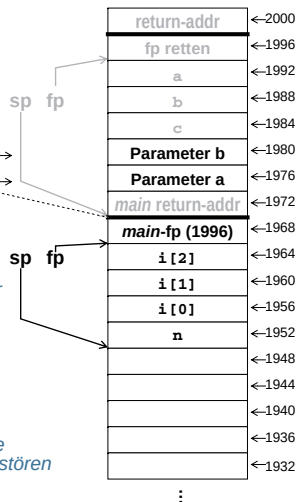
main() bereitet den Aufruf von f1(int, int) vor

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

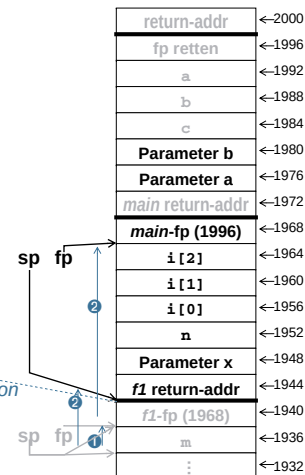
Stack-Aufbau bei Funktionsaufrufen

```
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
1 sp = fp
2 fp = pop(sp)



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

Stack-Aufbau bei Funktionsaufrufen

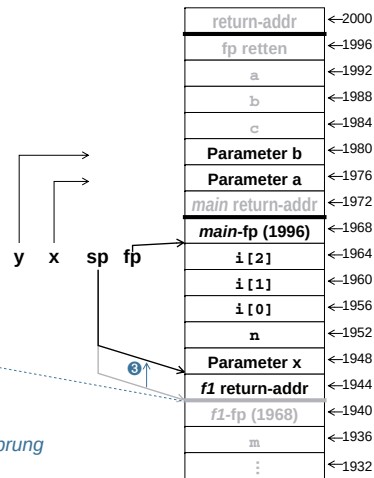
```
int main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

```
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);
}
```

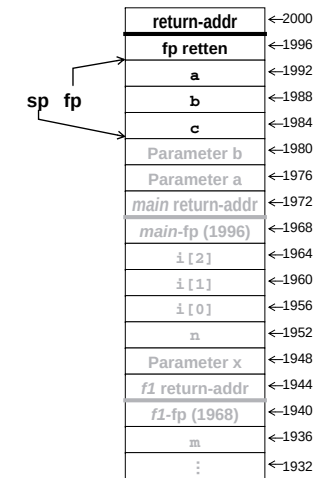
Rücksprung
return

f2() wird verlassen



Stack-Aufbau bei Funktionsaufrufen

```
int main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```



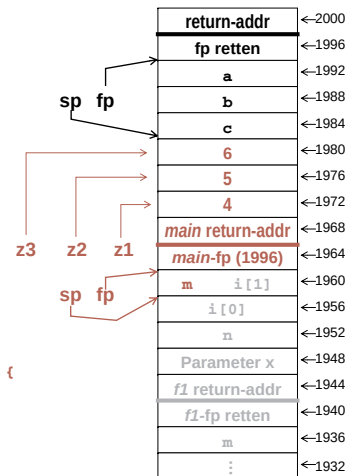
zurück in main()

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@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

- 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!**

