

G Nebenläufigkeit

G.1 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?

G.2 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 int a;

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

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

G.2 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;
...

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

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

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

G.2 Nebenläufigkeit durch Interrupts (3)

- Annahme1: Interrupt trifft folgendermaßen ein:

```
.L5:
H1  lds r24,a
H2  lds r25,(a)+1
H3  rcall print
H4  sts (a)+1,__zero_reg__
H5  sts a,__zero_reg__
H6  rjmp .L2

I1  lds r24,a
I2  lds r25,(a)+1
I3  adiw r24,1
I4  sts (a)+1,r25
I5  sts a,r24
```

- ▶ Folge: ein Fahrzeug wird nicht gezählt

- Details des Szenarios zeigen mehrere Problemstellen:
 - ▶ int-Wert wird in zwei Schritten in zwei Register geladen (long: 4 Register)
 - ▶ Operationen erfolgen in Registern, danach wird in Speicher zurückgeschrieben

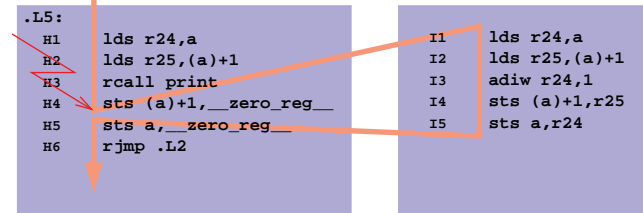
G.2 Nebenläufigkeit durch Interrupts (3)

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

Codezeile	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			
I2			00	05			
I3			00	06			
I4	00		00				
I5		06		06			
ret			00	05	00	05	
H3			00	05			5
H4	00						
H5		00					

G.2 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

G.2 Nebenläufigkeit durch Interrupts (4)

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

Codezeile	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			
I2	00		00	ff			
I3			01	00			
I4	01		01				
I5		00		00			
ret			00	ff	00	ff	
H5	01	00					

G.2 Nebenläufigkeit durch Interrupts (5)

- weiteres Problem bei Zugriff auf globale Variablen:
 - AVR stellt 32 Register zur Verfügung
 - 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 int a;
```

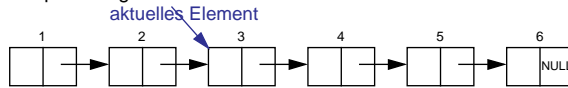
- Nachteil: Code wird umfangreicher und langsamer
→ nur einsetzen wo unbedingt notwendig!

G.3 Nebenläufigkeitsprobleme allgemein

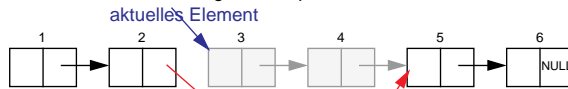
G.3 Nebenläufigkeitsprobleme allgemein

- 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



SPIC

G.4 Umgang mit Nebenläufigkeitsproblemen (2)

G.4 Umgang mit Nebenläufigkeitsproblemen

- Zugriffskonflikte mit Interrupt-Handlern verhindern
 - ◆ das Programm muss vor kritischen Zugriffen auf gemeinsame Daten Interrupts sperren
 - Beispiel AVR: Funktionen `clic()` (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!
 - es kann sinnvoll sein, nur Interrupts des Geräts zu sperren, dessen Handler auch auf die kritischen Daten zugreift (hängt von Details der Hardware ab!)

SPIC

G.4 Umgang mit Nebenläufigkeitsproblemen

G.4 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 int INT_zaeehler;
```

- 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. D.9-3)

SPIC

G.4 Umgang mit Nebenläufigkeitsproblemen (3)

G.4 Umgang mit Nebenläufigkeitsproblemen

- 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?

SPIC

G.4 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(); // Interrupt!
        }
        /* bearbeite Ereignis */
        ...
    }
}
```

→ Lost-wakeup-Problem

G.4 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(); // Interrupt!
        }
        /* bearbeite Ereignis */
        ...
    }
}
```

- ◆ können hier Interruptsperrern helfen?

G.4 Umgang mit Nebenläufigkeitsproblemen (6)

- ... Warten auf einen Interrupt
 - ◆ Problem: Interruptsperrere 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();
            sleep_cpu(); // Interrupt!
        }
        /* bearbeite Ereignis */
        ...
    }
}
```

- ▶ aber Interrupt darf nicht zwischen sei() und sleep_cpu() kommen
- ▶ Lösung: sei() und die Folgeanweisung werden atomar ausgeführt

G.4 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 Priorität)
- Mehrseitige Synchronisation
 - ◆ Standardsituation bei parallelen Abläufen (z. B. auf Mehrkern-Prozessor)
 - ▶ Interruptsperrern 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

H Programme, Prozesse und Speicher

H 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

SPIC

Systemnahe Programmierung in C
© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010

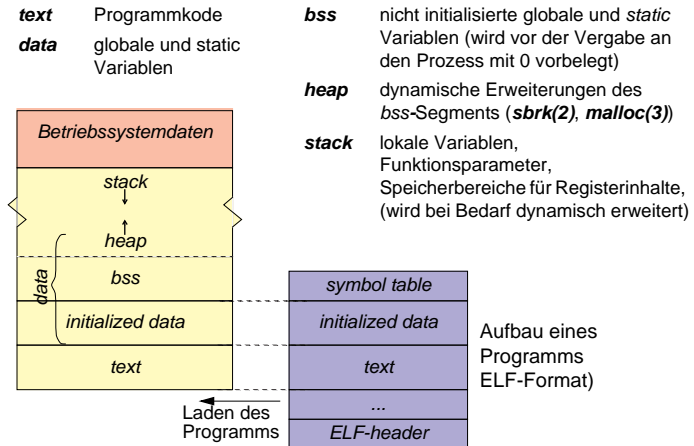
H-ProzesseSpeicher.fm 2010-06-22 18.50

H.1

Reproduktionsrecht vorbehalten. Nachdruck, Vervielfältigung und Verbreitung, auch auszugsweise, ist ohne schriftliche Genehmigung der Universität Erlangen-Nürnberg.

H.1 Speicherorganisation eines Prozesses

H.1 Speicherorganisation eines Prozesses



SPIC

Systemnahe Programmierung in C
© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010

H-ProzesseSpeicher.fm 2010-06-22 18.50

H.2

Reproduktionsrecht vorbehalten. Nachdruck, Vervielfältigung und Verbreitung, auch auszugsweise, ist ohne schriftliche Genehmigung der Universität Erlangen-Nürnberg.

H.1 Speicherorganisation eines Prozesses (2)

H.1 Speicherorganisation eines Prozesses

- Abbildung des virtuellen Adressraums in den realen Hauptspeicher durch Seitenadressierung (**Paging**)
 - Adreßraum ist in kleine (4 oder 8 kB) Stücke unterteilt (**Seiten**)
 - jede Seite wird über eine Tabelle in ein entsprechendes Stück des Hauptspeichers (**Kachel**) abgebildet
 - bei jedem Speicherzugriff wird die virtuelle Adresse in die entsprechende physikalische Adresse umgerechnet (spezielle Hardware: Memory Management Unit - MMU)
 - zu jeder Seite sind Zugriffsrechte vermerkt (nur lesen, lesen+schreiben, Maschinenbefehle ausführen)
 - eine Seite kann bei Speichermangel von Betriebssystem auf Festplatte ausgelagert werden und bei Bedarf automatisch wieder eingelagert werden

SPIC

Systemnahe Programmierung in C
© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010

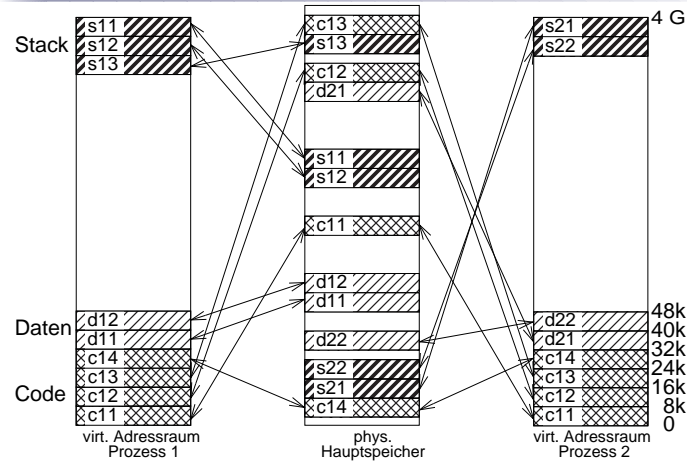
H-ProzesseSpeicher.fm 2010-06-22 18.50

H.3

Reproduktionsrecht vorbehalten. Nachdruck, Vervielfältigung und Verbreitung, auch auszugsweise, ist ohne schriftliche Genehmigung der Universität Erlangen-Nürnberg.

H.1 Speicherorganisation eines Prozesses(3)

H.1 Speicherorganisation eines Prozesses



SPIC

Systemnahe Programmierung in C
© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010

H-ProzesseSpeicher.fm 2010-06-22 18.50

H.4

Reproduktionsrecht vorbehalten. Nachdruck, Vervielfältigung und Verbreitung, auch auszugsweise, ist ohne schriftliche Genehmigung der Universität Erlangen-Nürnberg.

H.2 Stackaufbau eines Prozesses

1 Prinzip

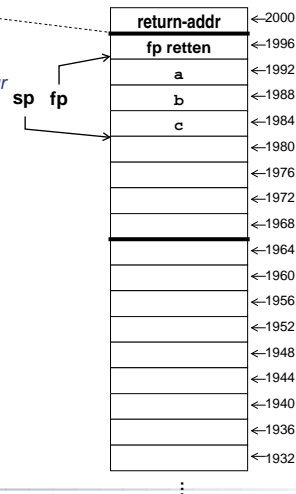
- für jede Funktion wird ein **Stack-Frame** angelegt, in dem
 - lokale Variablen der Funktion
 - Aufrufparameter an weitere Funktionen
 - Registerbelegung der Funktion während des Aufrufs weiterer Funktionen
 gespeichert werden
- Stackorganisation ist abhängig von
 - Prozessor
 - Compiler und
 - Betriebssystem
- Beispiele aus einem UNIX auf Intel-Prozessor (typisch für CISC)
 - RISC-Prozessoren mit Registerfiles gehen teilweise anders vor!

SPIC

2 ■ Stack mehrerer Funktionsaufrufe

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

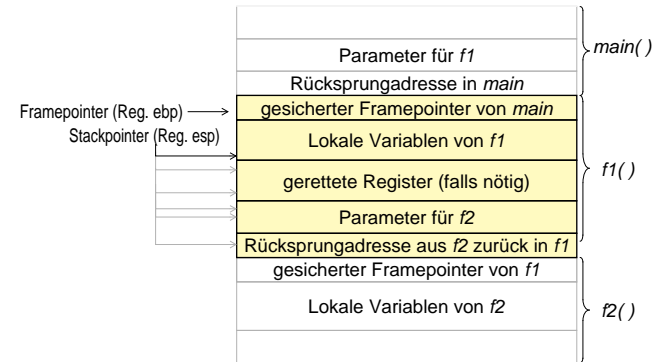
Stack-Frame für main erstellen
&a = fp-4
&b = fp-8
&c = fp-12



SPIC

2 Beispiel

■ Aufbau eines Stack-Frames (Funktionen main(), f1(), f2())



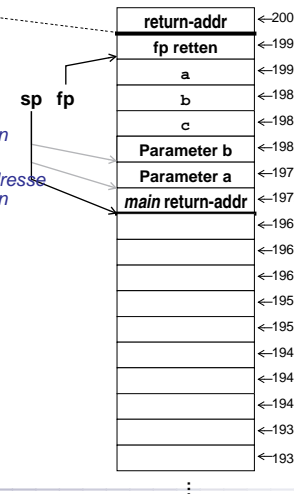
► Achtung: architekturabhängige Optimierungen können zu Padding (Füllbytes) führen!

SPIC

2 ■ Stack mehrerer Funktionsaufrufe

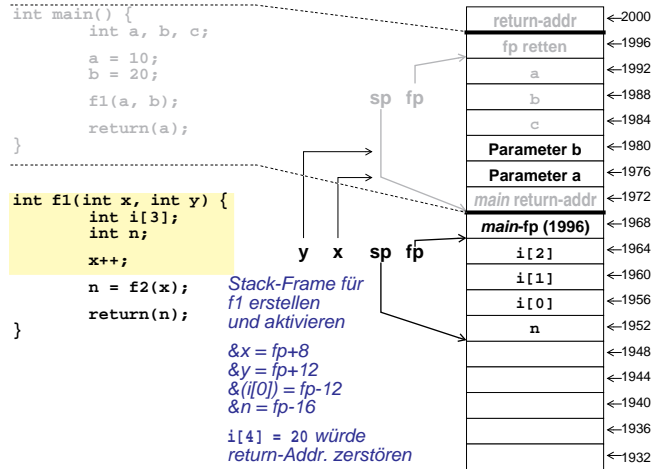
```
int main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

Parameter auf Stack legen
Bei Aufruf Rücksprungadresse auf Stack legen

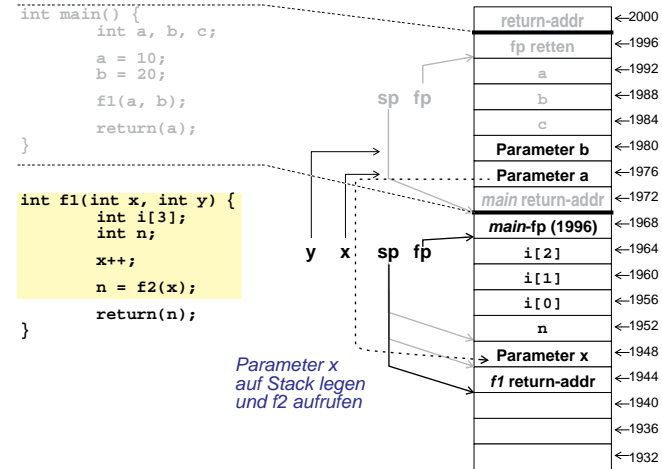


SPIC

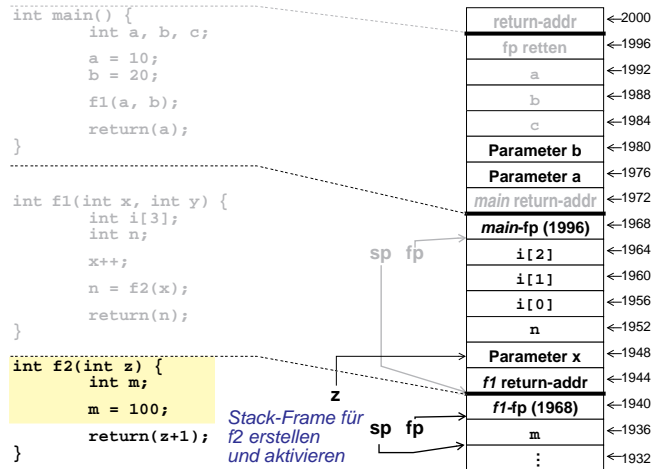
2 ■ Stack mehrerer Funktionsaufrufe



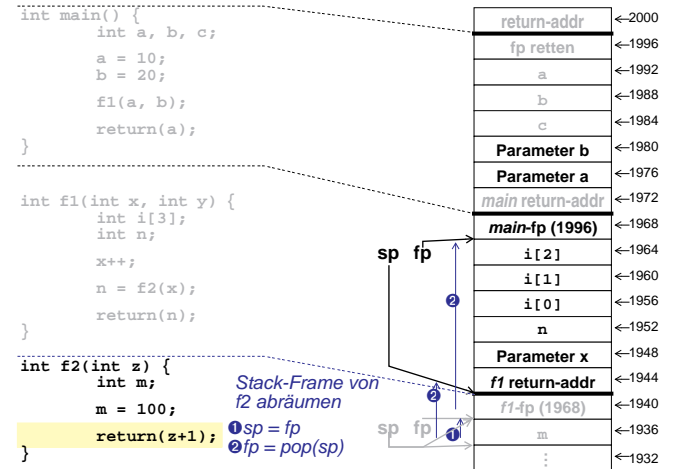
2 ■ Stack mehrerer Funktionsaufrufe



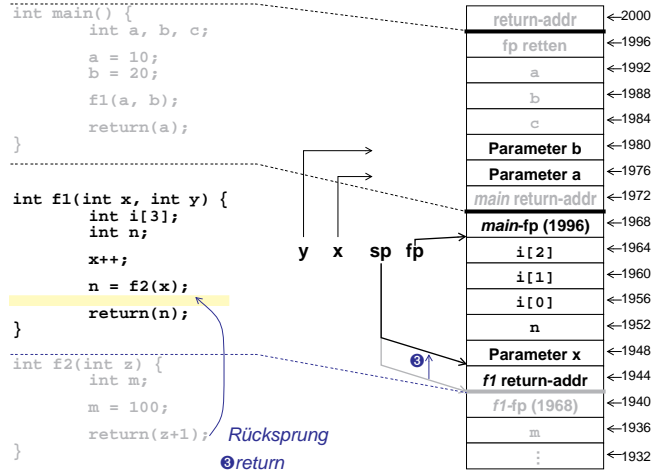
2 ■ Stack mehrerer Funktionsaufrufe



2 ■ Stack mehrerer Funktionsaufrufe

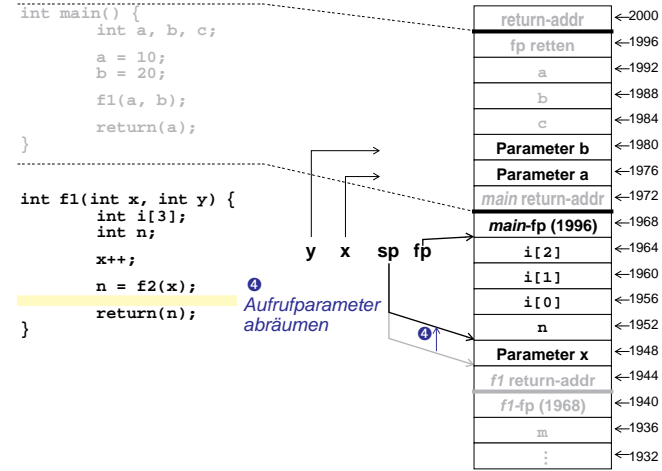


2 ■ Stack mehrerer Funktionsaufrufe



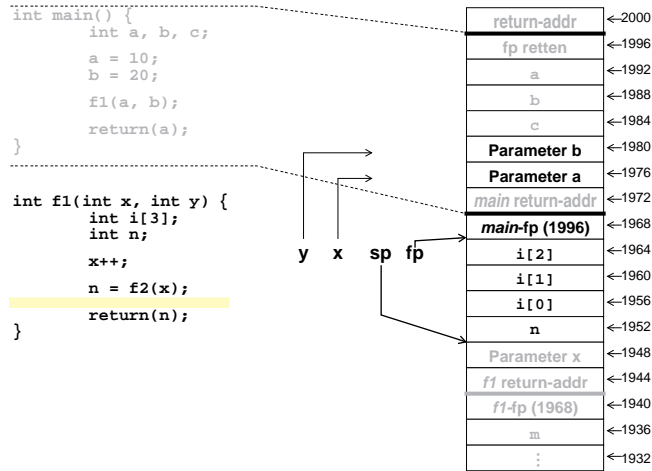
SPIC

2 ■ Stack mehrerer Funktionsaufrufe



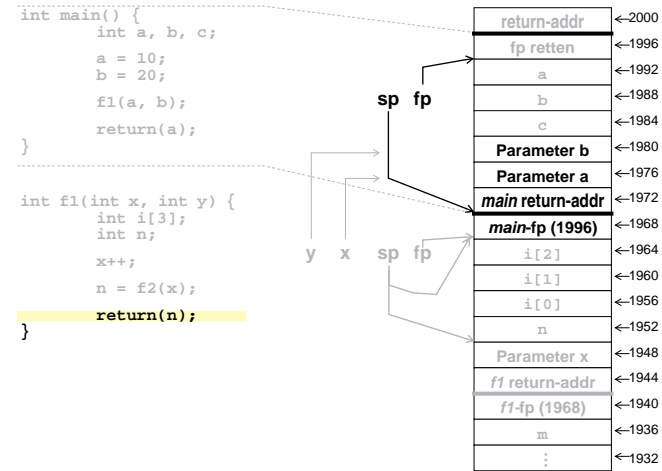
SPIC

2 ■ Stack mehrerer Funktionsaufrufe



SPIC

2 ■ Stack mehrerer Funktionsaufrufe

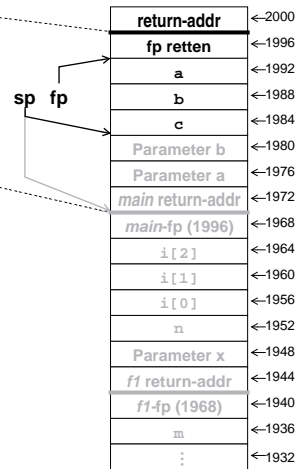


SPIC

2 ■ Stack mehrerer Funktionsaufrufe

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

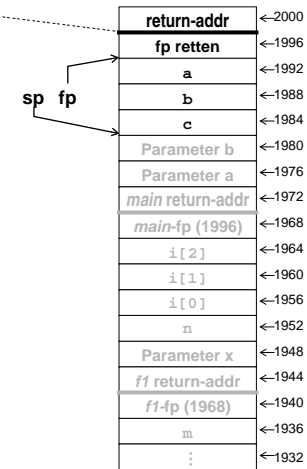
int f1(int x, int y) {
    int i[3];
    int n;
    x++;
    n = f2(x);
    return(n);
}
```



SPIC

2 ■ Stack mehrerer Funktionsaufrufe

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



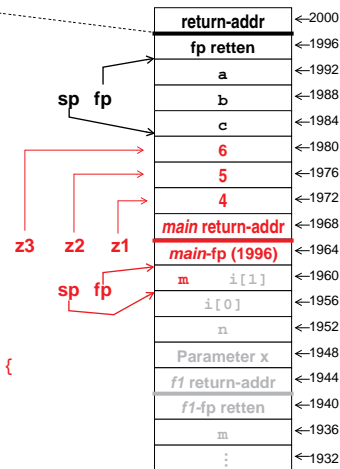
SPIC

2 ■ Stack mehrerer Funktionsaufrufe

```
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);
}
```



SPIC

