

Überblick: Teil D Betriebssystemabstraktionen

15 Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme

18 Dateisysteme

19 Programme und Prozesse

20 Speicherorganisation

21 Nebenläufige Prozesse

V_SPiC_handout



Speicherorganisation auf einem μ C

16-Speicher: 2015-04-08



```
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	

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

ELF-Binary

20-2

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 alle globalen/statischen Variablen, sowie den Code → 12-5
- Allokation durch Platzierung in einer **Sektion**

.text – enthält den Programmcode

main()

.bss – enthält alle mit 0 initialisierten Variablen

a

.data – enthält alle mit anderen Werten initialisierten Variablen

b,s

.rodata – enthält alle unveränderlichen Variablen

c

■ Dynamische Allokation – Reservierung zur Laufzeit

- Betrifft lokale auto-Variablen und explizit angeforderten Speicher

Stack – enthält alle **aktuell lebendigen** auto-Variablen

x,y,p

Heap – enthält explizit mit `malloc()` angeforderte Speicherbereiche

*p

© dl SPiC (Teil D, SS15) 20 Speicherorganisation | 20.1 Einführung

20-1

Speicherorganisation auf einem μ C

16-Speicher: 2015-04-08



```
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

Flash / ROM

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

μ -Controller

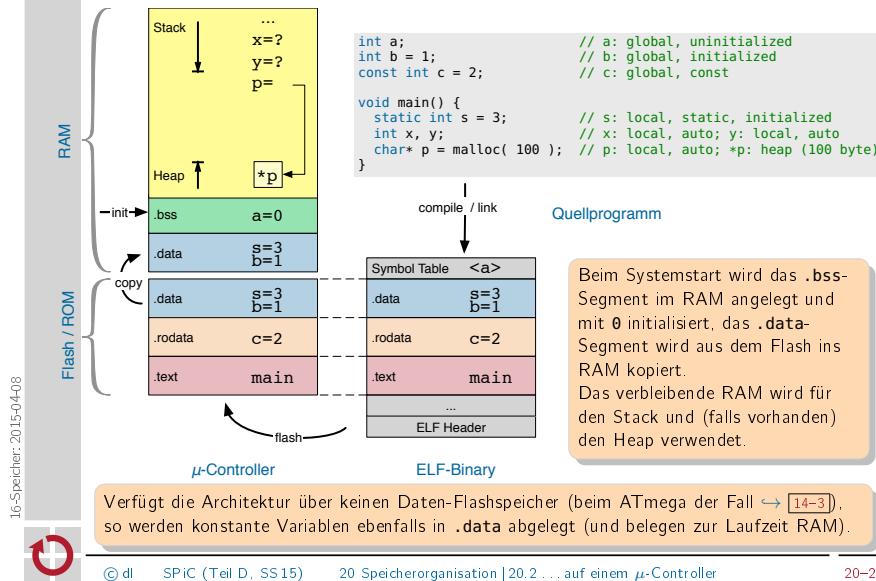
Zur Installation auf dem μ C werden .text und .[ro]data in den Flash-Speicher des μ C geladen.

20-2

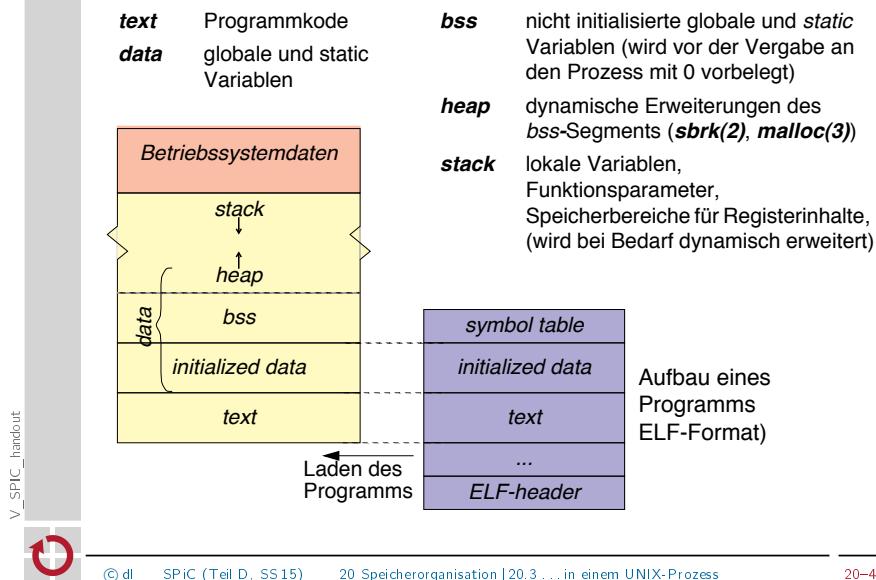
© dl SPiC (Teil D, SS15) 20 Speicherorganisation | 20.2 ... auf einem μ -Controller

20-2

Speicherorganisation auf einem μ C



Speicherorganisation in einem UNIX-Prozess (Forts.)



Speicherorganisation in einem UNIX-Prozess

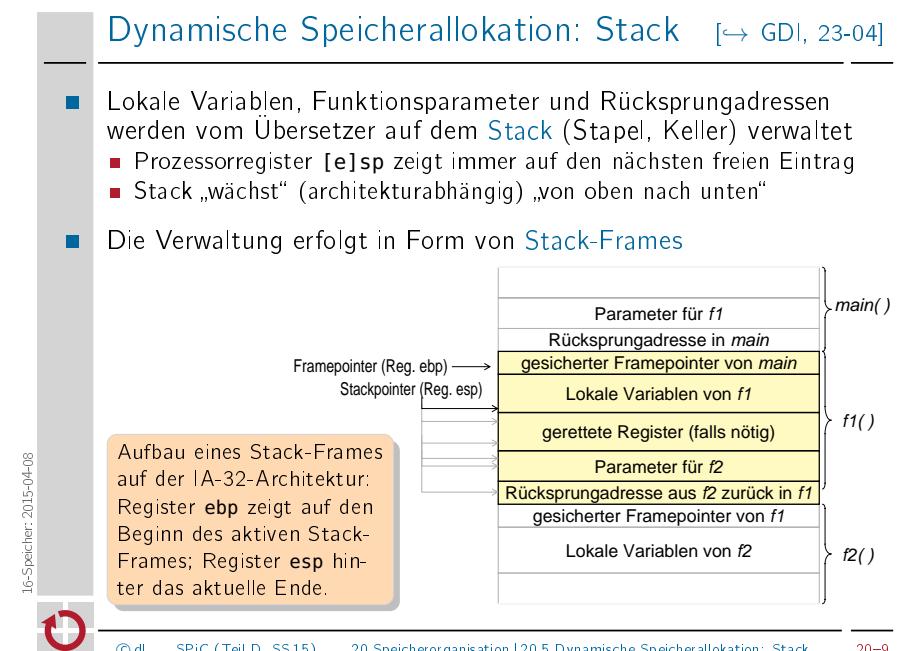
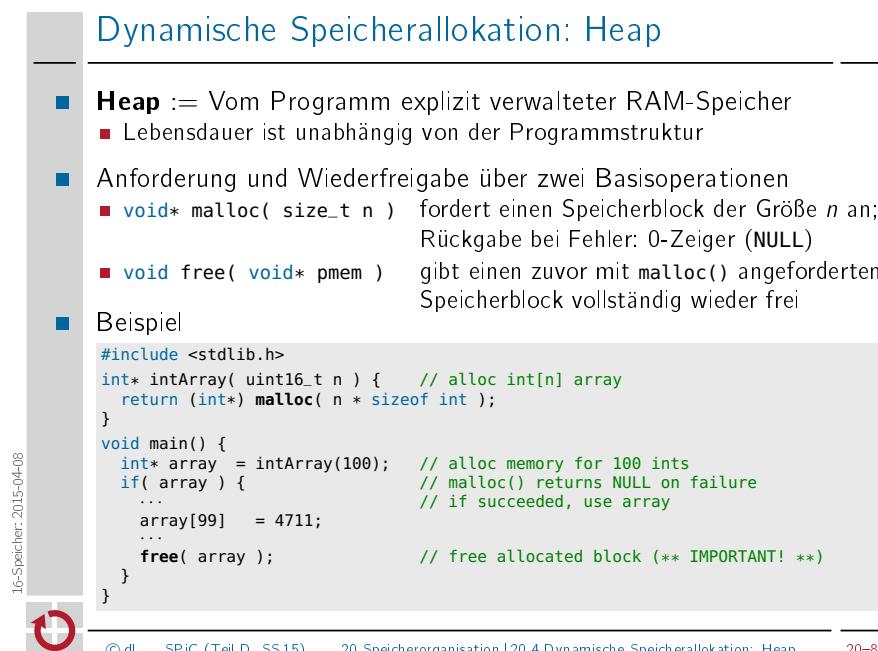
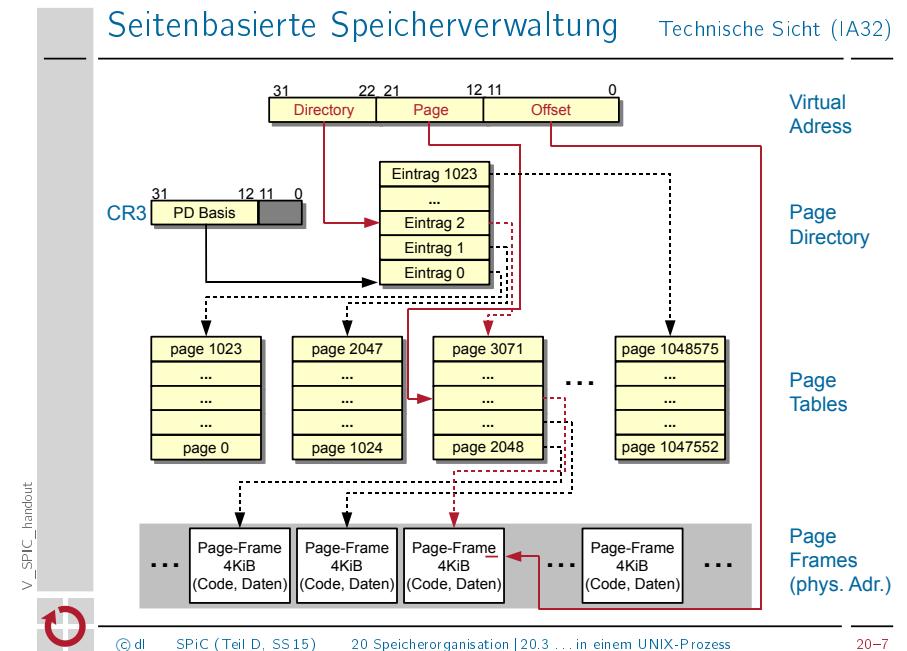
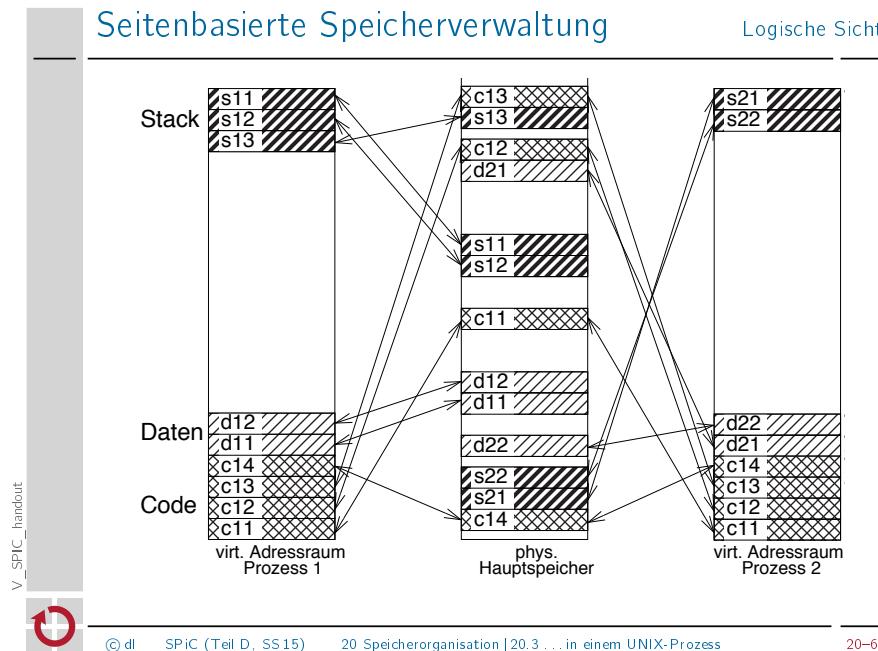
- **Programm:** Folge von Anweisungen
- **Prozess:** Betriebssystemkonzept zur Ausführung von Programmen
 - Programm, das sich in Ausführung befindet, und seine Daten
(Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
 - Eine konkrete **Ausführungsumgebung** für ein Programm (Prozessor, Speicher, ...) → vom Betriebssystem verwalteter **virtueller Computer**
- Jeder Prozess bekommt einen **virtuellen Adressraum** zugewiesen
 - 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

© dl SPIC (Teil D, SS15) 20 Speicherorganisation | 20.3 ... in einem UNIX-Prozess 20-3

Seitenbasierte Speicherverwaltung

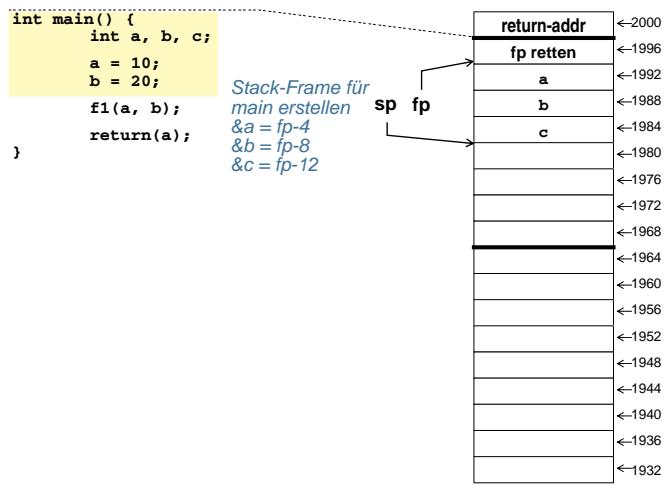
- Die Abbildung von virtuellem Speicher (VS) auf physikalischen Speicher (PS) erfolgt durch **Seitenadressierung (Paging)**
 - VS eines Prozesses ist unterteilt in **Speicherseiten (Memory Pages)**
 - kleine Adressblöcke, üblich sind z. B. 4 KiB und 4 MiB Seiten
 - in dieser Granularität wird Speicher 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 Prozessen 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

© dl SPIC (Teil D, SS15) 20 Speicherorganisation | 20.3 ... in einem UNIX-Prozess 20-5



Stack-Aufbau bei Funktionsaufrufen

16-Speicher: 2015-04-08

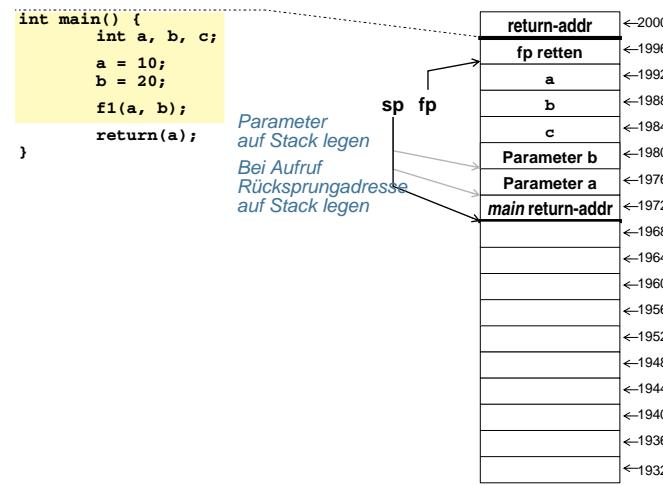


Beispiel hier für 32-Bit-Architektur (4-Byte ints), main() wurde soeben betreten

© dl SPiC (Teil D, SS15) 20 Speicherorganisation | 20.5 Dynamische Speicherallokation: Stack 20-10

Stack-Aufbau bei Funktionsaufrufen

16-Speicher: 2015-04-08

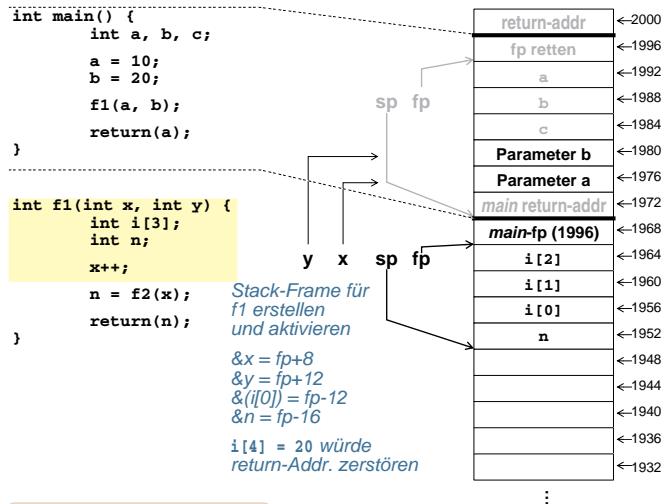


main() bereitet den Aufruf von f1(int, int) vor

© dl SPiC (Teil D, SS15) 20 Speicherorganisation | 20.5 Dynamische Speicherallokation: Stack 20-10

Stack-Aufbau bei Funktionsaufrufen

16-Speicher: 2015-04-08

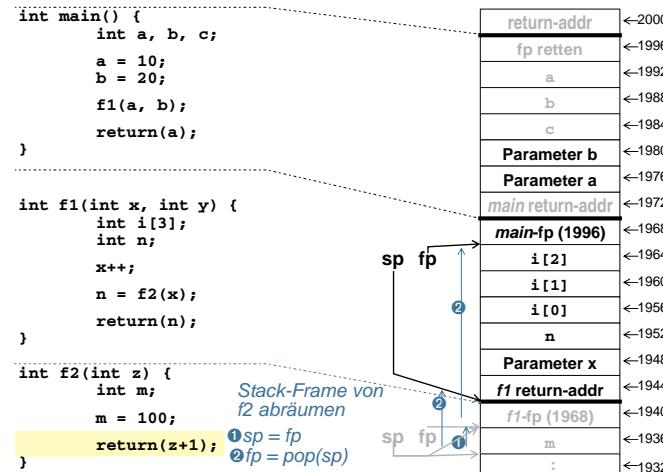


f1() wurde soeben betreten

© dl SPiC (Teil D, SS15) 20 Speicherorganisation | 20.5 Dynamische Speicherallokation: Stack 20-10

Stack-Aufbau bei Funktionsaufrufen

16-Speicher: 2015-04-08

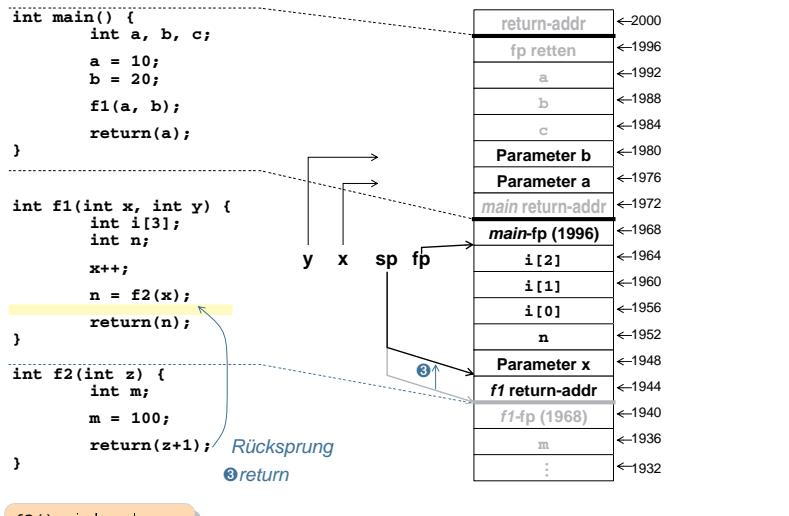


f2() bereitet die Terminierung vor (wurde von f1() aufgerufen und ausgeführt)

© dl SPiC (Teil D, SS15) 20 Speicherorganisation | 20.5 Dynamische Speicherallokation: Stack 20-10

Stack-Aufbau bei Funktionsaufrufen

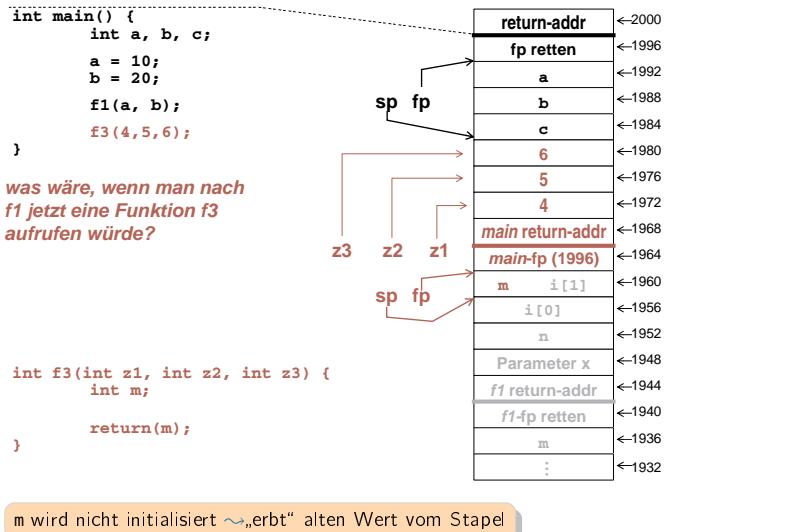
16-Speicher: 2015-04-08



© dl SPiC (Teil D, SS 15) 20 Speicherorganisation | 20.5 Dynamische Speicherallokation: Stack 20-10

Stack-Aufbau bei Funktionsaufrufen

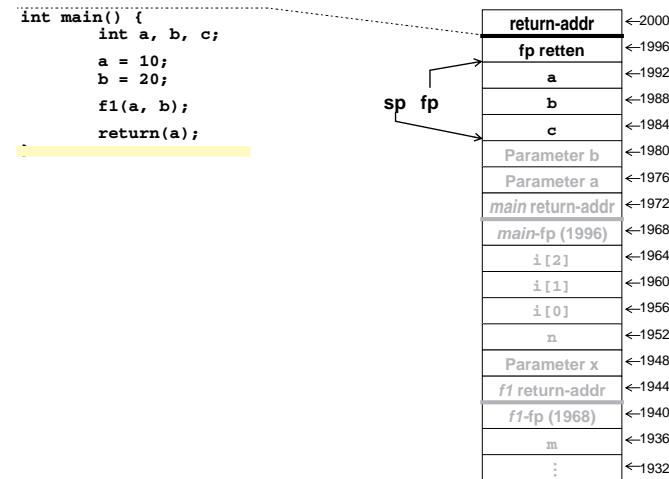
16-Speicher: 2015-04-08



© dl SPiC (Teil D, SS 15) 20 Speicherorganisation | 20.5 Dynamische Speicherallokation: Stack 20-10

Stack-Aufbau bei Funktionsaufrufen

16-Speicher: 2015-04-08



zurück in main()

© dl SPiC (Teil D, SS 15) 20 Speicherorganisation | 20.5 Dynamische Speicherallokation: Stack 20-10

Statische versus dynamische Allokation

- Bei der **μ C-Entwicklung** wird **statische Allokation** bevorzugt
 - Vorteil:** Speicherplatzbedarf ist bereits nach dem Übersetzen / Linken exakt bekannt (kann z. B. mit `size` angegeben werden)
 - Speicherprobleme frühzeitig erkennbar (Speicher ist knapp! ↪ [1-4])

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 Speicher Kosten durch Verwaltungsstrukturen und Code
 - Speicherbedarf zur Laufzeit schlecht abschätzbar
 - Risiko von Programmierfehlern und Speicherlecks

© dl SPiC (Teil D, SS 15) 20 Speicherorganisation | 20.6 Statische vs. Dynamische Allokation 20-11

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



© dl SPiC (Teil D, SS 15) 20 Speicherorganisation | 20.6 Statistische vs. Dynamische Allokation

20-12

L.pdf: 2014-11-21



© jk SPiC (Teil D, SS 12) 21 Kontrollfäden / Aktivitätsträger (Threads)

21-1

Überblick: Teil D Betriebssystemabstraktionen

15 Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme

18 Dateisysteme

19 Programme und Prozesse

20 Speicherorganisation

21 Nebenläufige Prozesse

v_SPiC_handout



Prozesse mit gemeinsamem Speicher

- Gemeinsame Nutzung von Speicherbereichen durch mehrere Prozesse
- ▲ Nachteile
 - viele Betriebsmittel zur Verwaltung eines Prozesses notwendig
 - Dateideskriptoren
 - Speicherabbildung
 - Prozesskontrollblock
 - Prozessumschaltungen sind aufwändig
- ★ Vorteil
 - in Multiprozessorsystemen sind echt parallele Abläufe möglich

© jk SPiC (Teil D, SS 12) 21 Kontrollfäden / Aktivitätsträger (Threads) | 21.1 Prozesse mit gemeinsamem Speicher

21-2

Threads in einem Prozess

★ Lösungsansatz:

Kontrollfäden / Aktivitätsträger (*Threads*) oder
leichtgewichtige Prozesse (*Lightweight Processes, LWPs*)

- Jeder Thread repräsentiert einen eigenen aktiven Ablauf:
 - eigener Programmzähler
 - eigener Registersatz
 - eigener Stack
- eine Gruppe von Threads nutzt gemeinsam eine Menge von Betriebsmitteln (gemeinsame Ausführungsumgebung)
 - Instruktionen
 - Datenbereiche (Speicher)
 - Dateien, etc.
- ➔ letztlich wird das Konzept des Prozesses aufgespalten:
eine Ausführungsumgebung für mehrere (parallele oder nebenläufige) Abläufe

L.pdf: 2014-11-21



Threads (2)

- Umschalten zwischen zwei Threads einer Gruppe ist erheblich billiger als eine normale Prozessumschaltung.

- es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf).
- Speicherabbildung muss nicht gewechselt werden.
- alle Systemressourcen bleiben verfügbar.

- ein klassischer UNIX-Prozess ist ein Adressraum mit einem Thread

- Implementierungen von Threads

- User-level Threads
- Kernel-level Threads

L.pdf: 2014-11-21



User-Level-Threads

■ Implementierung

- Instruktionen im Anwendungsprogramm schalten zwischen den Threads hin- und her (ähnlich wie der Scheduler im Betriebssystem)
- Realisierung durch Bibliotheksfunktionen
- Betriebssystem sieht nur einen Kontrollfaden

★ Vorteile

- keine Systemaufrufe zum Umschalten erforderlich
- effiziente Umschaltung (einige wenige Maschinenbefehle)
- Schedulingstrategie in der Hand des Anwendungsprogrammierers

▲ Nachteile

- bei blockierenden Systemaufrufen bleibt die ganze Anwendung (und damit alle User-Level-Threads) stehen
- kein Ausnutzen eines Multiprozessors möglich

L.pdf: 2014-11-21



Kernel-Level-Threads

■ Implementierung

- Betriebssystem kennt Kernel-Level-Threads
- Betriebssystem schaltet zwischen Threads um

★ Vorteile

- kein Blockieren unbeteiligter Threads bei blockierenden Systemaufrufen
- Betriebssystem kann mehrere Threads einer Anwendung gleichzeitig auf verschiedenen Prozessoren laufen lassen

▲ Nachteile

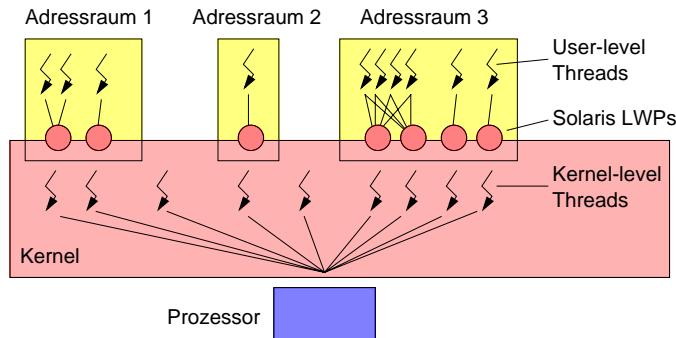
- weniger effizientes Umschalten zwischen Threads (Umschaltung in den Systemkern notwendig)
- Schedulingstrategie meist durch Betriebssystem vorgegeben

L.pdf: 2014-11-21



Mischform: LWPs und Threads (Bsp. Solaris)

- Solaris kennt Kernel-, User-Level-Threads und LWPs



Nach Silberschatz, 1994

- wenige Kernel-level-Threads um Parallelität zu erreichen, viele User-level-Threads, um die unabhängigen Abläufe in der Anwendung zu strukturieren

L.pdf: 2014-11-21

Koordinierungsprobleme

- Grundlegende Probleme

- gegenseitiger Ausschluss (Koordinierung)
 - ein Thread möchte auf einen kritischen Datenbereich zugreifen und verhindern, dass andere Threads gleichzeitig zugreifen
- gegenseitiges Warten (Synchronisation)
 - ein Thread will darauf warten, dass ein anderer einen bestimmten Bearbeitungsstand erreicht hat

- Komplexere Koordinierungsprobleme (Beispiele)

- Bounded Buffer
 - Threads schreiben Daten in einen Pufferspeicher (meist als Feld implementiert), andere entnehmen Daten (kritische Situationen: Zugriff auf den Puffer, Puffer leer, Puffer voll)
- Philosophenproblem
 - ein Thread reserviert sich zuerst Zugriff auf Datenbereich 1, dann auf Datenbereich 2, der andere Thread umgekehrt
 - kann zu Verklemmung führen

L.pdf: 2014-11-21

Koordinierung / Synchronisation

- Threads arbeiten nebenläufig oder parallel auf Multiprozessor
 - Threads haben gemeinsamen Speicher
 - alle von Interrupts und Signalen bekannten Probleme beim Zugriff auf gemeinsame Daten treten auch bei Threads auf
- ★ Unterschied zwischen Threads und Interrupt-Service-Routinen bzw. Signal-Handler-Funktionen:
- "Haupt-Kontrollfaden" der Anwendung und eine ISR bzw. ein Signal-Handler sind nicht gleichberechtigt
 - ISR bzw. Signal-Handler unterbricht den Haupt-Kontrollfaden aber ISR bzw. Signal-Handler werden nicht unterbrochen
 - zwei Threads sind gleichberechtigt
 - ein Thread kann jederzeit zugunsten eines anderen unterbrochen werden (Scheduler) oder parallel zu einem anderen arbeiten (MPS)
 - Interrupts sperren oder Signale blockieren hilft nicht!

L.pdf: 2014-11-21

Gegenseitiger Ausschluss (*mutual exclusion*)

- Einfache Implementierung durch *mutex*-Variablen

```
mutex m = 1;
volatile int counter = 0;

...           Thread 1
lock(&m);
counter++;
unlock(&m);

...           Thread 2
lock(&m);
printf("%d\n", counter);
counter = 0;
unlock(&m);
```

→ nur der Thread, der das *lock* gemacht hat, darf das *unlock* aufrufen!

- Realisierung (konzeptionell)

```
void lock (mutex *m) {
    while (*m == 0) {
        /* warten */
    }
    m = 0;
}

void unlock (mutex *m) {
    *m = 1;
    /* ggf. wartende
       Threads wecken */
}
```

atomare
Funktionen

21–8

L.pdf: 2014-11-21

Zählende Semaphore

- Ein Semaphor (griech. Zeichenträger) ist eine Datenstruktur des Systems mit zwei Operationen (nach Dijkstra)

- P-Operation (*proberen; passeren; wait; down*)

► wartet bis Zugang frei

```
void P( int *s ) {
    while( *s <= 0 ) {
        /* warten */
    };
    *s = *s-1;
}
```

atomare Funktion

- V-Operation (*verhogen; vrijgeven; signal; up*)

► macht Zugang für anderen Thread / Prozess frei

```
void V( int *s ) {
    *s = *s+1;
    /* ggf wartende Threads/Prozesse wecken */
}
```

atomare Funktion

► P und V müssen nicht vom selben Thread/Prozess aufgerufen werden!

L.pdf: 2014-11-21



Gegenseitiges Warten

- Implementierung mit Hilfe eines Semaphors

```
int barrier = 0;
int result;
```

...
P(&barrier);
f1(result);
...

Thread 1

...
result = f2(...);
V(&barrier);
...

Thread 2

► Thread 2 läuft immer ungehindert durch

► Thread 1 blockiert an P, falls Thread 2 die V-Operation noch nicht ausgeführt hat (und wartet auf die V-Operation) – andernfalls läuft Thread 1 auch durch

L.pdf: 2014-11-21



Mutex im Detail: spin lock vs. sleeping lock

- spin lock

- aktives Warten bis Mutex-Variable frei (= 1) wird
 - entspricht konzeptionell einem Pollen
 - Thread bleibt im Zustand *laufend*
- Problem: wenn nur ein Prozessor verfügbar ist, wird Rechenzeit vergeudet bis durch den Scheduler eine Umschaltung erfolgt
 - nur ein anderer, laufender Thread kann den Mutex frei geben

- sleeping lock

- passives Warten
 - Thread geht in den Zustand *blockiert* (Schlafen, bis ein Ereignis eintrifft)
 - im Rahmen von unlock() wird der blockierte Thread in den Zustand *bereit* zurückgeführt
- Problem: bei sehr kurzen kritischen Abschnitten ist der Aufwand für das Blockieren/Aufwecken und die Umschaltungen unverhältnismäßig teuer

L.pdf: 2014-11-21



Implementierung von spin locks

- zentrales Problem: Atomarität von mutex-Abfrage und -Setzen

```
void lock (mutex *m) {
    while ( *m == 0 ) { ; }
    m = 0;
}
```

- Lösung: spezielle Maschinenbefehle, die atomar eine Abfrage und Modifikation auf einer Hauptspeicherzelle ermöglichen

► *Test-and-Set, Compare-and-Swap, Load-link/store-conditional, ...*

- Beispiel: *Test-and-set* – atomarer Maschinenbefehl mit folgender Wirkung

► wenn zwei Threads den Befehl gleichzeitig ausführen wollen, sorgt die Hardware dafür, dass ein Thread den Befehl vollständig zuerst ausführt

```
bool test_and_set(bool *plock) {
    bool initial= *plock;
    *plock= TRUE;
    return initial;
}
```

► Ausgangssituation: **plock == FALSE*

► Ergebnis von *test_and_set*:
der Thread, der den Befehl zuerst ausführt,
erhält *FALSE*,
der andere *TRUE*

L.pdf: 2014-11-21



Implementierung von spin locks (2)

- Realisierung von mutex-Operationen mit dem *Test-and-Set*-Befehl

```
mutex m = FALSE;
```

```
void lock (mutex *m) {  
    while(test_and_set(&m)){ ; }  
    /* got lock */  
}
```

```
void unlock (mutex *m) {  
    *m = FALSE;  
}
```



Implementierung von sleeping locks

- zwei Probleme

- Konflikt mit einer zweiten lock-Operation: Atomarität von mutex-Abfrage und -Setzen
- Konflikt mit einem unlock: *lost-wakeup*-Problem

```
void lock (mutex *m) {  
    while (*m == 0) { sleep(); }  
    m = 0;  
}
```

- Ursachen

- (1) Prozessumschaltung während der lock-Operation
- (2) Bei Multiprozessoren:
gleichzeitige Ausführung von lock auf einem anderen Prozessor



Implementierung von sleeping locks (2)

- Behebung von Ursache (1): Prozessumschaltungen verhindern
 - Prozessumschaltung ist ein Funktion des Betriebssystem-Kerns
 - erfolgt im Rahmen eines Systemaufrufs
 - oder im Rahmen einer Interrupt-Behandlung
 - lock/unlock werden ebenfalls im BS-Kern implementiert, BS-Kern läuft unter Interrupt-Sperre

```
void lock (mutex *m) {  
    enter_OS(); cli();  
    while (*m == 0) {  
        block_thread_and_schedule();  
    }  
    m = 0;  
    sei(); leave_OS();  
}
```

```
void unlock (mutex *m) {  
    enter_OS(); cli();  
    *m = 1;  
    wakeup_waiting_threads();  
    sei(); leave_OS();  
}
```



Implementierung von sleeping locks (3)

- Behebung von Ursache (2): Parallelie Ausführung auf anderem Prozessor verhindern
 - Problem (1) (Prozessumschaltungen) bleibt gleichzeitig bestehen
 - Gegenseitiger Ausschluss mit anderen Prozessoren durch spin locks

```
void lock (mutex *m) {  
    enter_OS(); cli();  
    spin_lock();  
    while (*m == 0) {  
        block_thread_and_schedule();  
    }  
    m = 0;  
    spin_unlock();  
    sei(); leave_OS();  
}
```

```
void unlock (mutex *m) {  
    enter_OS(); cli();  
    spin_lock();  
    *m = 1;  
    wakeup_waiting_threads();  
    spin_unlock();  
    sei(); leave_OS();  
}
```



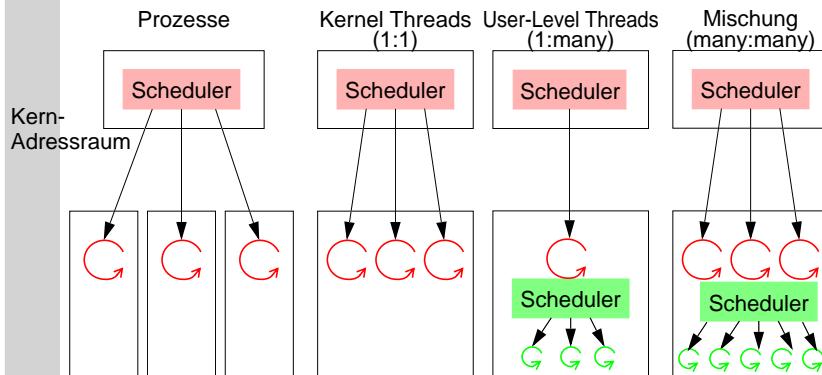
Thread-Konzepte in UNIX/Linux

- verschiedene Implementierungen von Thread-Paketen verfügbar
 - reine User-Level-Threads
eine beliebige Zahl von User-Level-Threads wird auf einem Kernel Thread "gemultiplexed" (*many:1*)
 - reine Kernel-Level-Threads
jedem auf User-Level sichtbaren Thread ist 1:1 ein Kernel-Level-Thread zugeordnet (*1:1*)
 - Mischungen: eine große Zahl von User-Level Threads wird auf eine kleinere Zahl von Kernel Threads abgebildet (*many:many*)
 - + User-Level Threads sind billig
 - + die Kernel Threads ermöglichen echte Parallelität auf einem Multiprozessor
 - + wenn sich ein User-Level-Thread blockiert, dann ist mit ihm der Kernel-Level-Thread blockiert in dem er gerade abgewickelt wird — aber andere Kernel-Level-Threads können verwendet werden um andere, lauffähige User-Level-Threads weiter auszuführen

L.pdf: 2014-11-21



Thread-Konzepte in UNIX/Linux (2)



L.pdf: 2014-11-21



pthread-Benutzerschnittstelle

■ Pthreads-Schnittstelle (Basisfunktionen):

- `pthread_create`** Thread erzeugen & Startfunktion angeben
- `pthread_exit`** Thread beendet sich selbst
- `pthread_join`** Auf Ende eines anderen Threads warten
- `pthread_self`** Eigene Thread-Id abfragen
- `pthread_yield`** Prozessor zugunsten eines anderen Threads aufgeben

■ Funktionen in Pthreads-Bibliothek zusammengefasst

gcc ... -pthread

L.pdf: 2014-11-21



pthread-Benutzerschnittstelle (2)

■ Thread-Erzeugung

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine)(void *),
                  void *arg)
```

thread Thread-Id

attr Modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). **NULL** für Standardattribute.

Thread wird erzeugt und ruft Funktion **start_routine** mit Parameter **arg** auf.

Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode als Ergebnis zurückgeliefert.

L.pdf: 2014-11-21



pthread-Benutzerschnittstelle (3)

- Thread beenden (bei return aus `start_routine` oder):

```
void pthread_exit(void *retval)
```

Der Thread wird beendet und `retval` wird als Rückgabewert zurück geliefert (siehe `pthread_join`)

- Auf Thread warten und exit-Status abfragen:

```
int pthread_join(pthread_t thread, void **retvalp)
```

Wartet auf den Thread mit der Thread-ID `thread` und liefert dessen Rückgabewert über `retvalp` zurück.

Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode als Ergebnis zurückgeliefert.

L.pdf: 2014-11-21



Beispiel (Multiplikation Matrix mit Vektor)

```
double a[100][100], b[100], c[100];
int main(int argc, char* argv[]) {
    pthread_t tids[100];
    ...
    for (i = 0; i < 100; i++)
        pthread_create(tids + i, NULL, mult,
                       (void*)(c + i));
    for (i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
    ...
}

void *mult(void *cp) {
    int j, i = (double *)cp - c;
    double sum = 0;

    for (j = 0; j < 100; j++)
        sum += a[i][j] * b[j];
    c[i] = sum;
    return 0;
}
```

L.pdf: 2014-11-21



Pthreads-Koordinierung

- UNIX stellt zur Koordinierung von Prozessen komplexe Semaphor-Operationen zur Verfügung
 - Implementierung durch den Systemkern
 - komplexe Datenstrukturen, aufwändig zu programmieren
 - für die Koordinierung von Threads viel zu teuer
- Bei Koordinierung von Threads reichen meist einfache **Mutex**-Variablen
 - gewartet wird durch Blockieren des Threads oder durch *busy wait* (*Spinlock*)

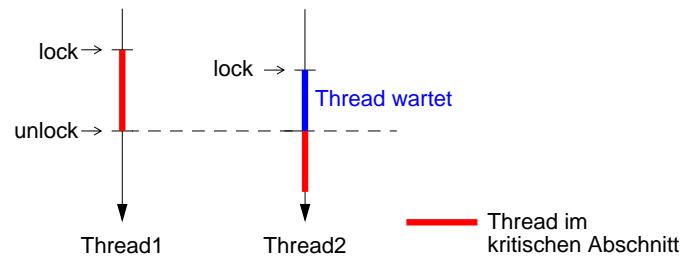
L.pdf: 2014-11-21



Pthreads-Koordinierung (2)

★ Mutexes

- Koordinierung von kritischen Abschnitten



L.pdf: 2014-11-21



Pthreads-Koordinierung (3)

... Mutexes (2)

■ Schnittstelle

■ Mutex erzeugen

```
pthread_mutex_t m1;  
pthread_mutex_init(&m1, NULL);
```

■ Lock & unlock

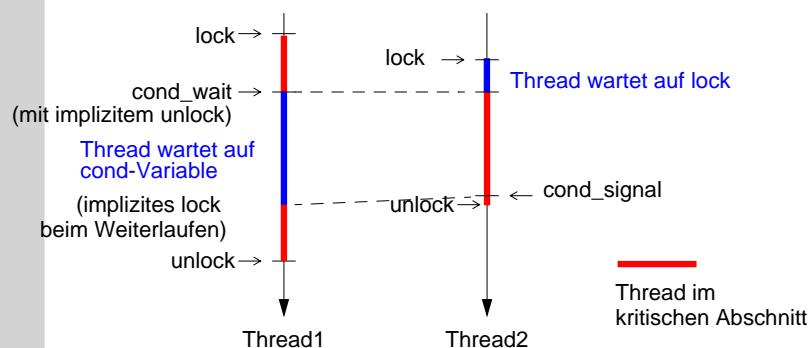
```
pthread_mutex_lock(&m1);  
... kritischer Abschnitt  
pthread_mutex_unlock(&m1);
```

L.pdf: 2014-11-21

Pthreads-Koordinierung (5)

★ Condition Variables

■ Mechanismus zum Blockieren (mit gleichzeitiger Freigabe des aktuellen kritischen Abschnitts) und Aufwecken (mit neuem Betreten des kritischen Abschnitts) von Threads



L.pdf: 2014-11-21

Pthreads-Koordinierung (4)

... Mutexes (3)

■ Komplexere Koordinierungsprobleme können alleine mit Mutexes nicht implementiert werden

- Problem:
 - Ein Mutex sperrt die eine komplexere Datenstruktur
 - Der Zustand der Datenstruktur erlaubt die Operation nicht
 - Thread muss warten, bis die Situation durch anderen Thread behoben wurde
 - Blockieren des Threads an einem weiteren Mutex kann zu Verklemmungen führen

- Lösung: Mutex in Verbindung mit sleep/wakeup-Mechanismus

➤ Condition Variables

L.pdf: 2014-11-21

Pthreads-Koordinierung (6)

Condition Variables (2)

■ Realisierung

- Thread reiht sich in Warteschlange der Condition Variablen ein
- Thread gibt Mutex frei
- Thread gibt Prozessor auf
- Ein Thread der die Condition Variable "frei" gibt weckt einen (oder alle) darauf wartenden Threads auf
- Deblockierter Thread muss als erstes den kritischen Abschnitt neu betreten (lock)
- Da möglicherweise mehrere Threads deblockiert wurden, muss die Bedingung nochmals überprüft werden

L.pdf: 2014-11-21

Pthreads-Koordinierung (7)

... Condition Variables (3)

■ Schnittstelle

- Condition Variable erzeugen

```
pthread_cond_t c1;  
pthread_cond_init(&c1, NULL);
```

- Beispiel: zählende Semaphore

P-Operation

```
void P(int *sem) {  
    pthread_mutex_lock(&m1);  
    while (*sem == 0)  
        pthread_cond_wait  
            (&c1, &m1);  
    (*sem)--;  
    pthread_mutex_unlock(&m1);  
}
```

V-Operation

```
void V(int *sem) {  
    pthread_mutex_lock(&m1);  
    (*sem)++;  
    pthread_cond_broadcast(&c1);  
    pthread_mutex_unlock(&m1);  
}
```

L.pdf: 2014-11-21

Threads und Koordinierung in Java

- Thread-Konzept und Koordinierungsmechanismen sind in Java integriert
- Erzeugung von Threads über Thread-Klassen
 - ▶ Beispiel

```
class MyClass implements Runnable {  
    public void run() {  
        System.out.println("Hello\n");  
    }  
  
    ...  
    MyClass o1 = new MyClass(); // create object  
    Thread t1 = new Thread(o1); // create thread to run in o1  
  
    t1.start(); // start thread  
  
    Thread t2 = new Thread(o1); // create second thread to run in o1  
  
    t2.start(); // start second thread
```

L.pdf: 2014-11-21

Pthreads-Koordinierung (8)

... Condition Variables (4)

- Bei **pthread_cond_signal** wird mindestens einer der wartenden Threads aufgeweckt — es ist allerdings nicht definiert welcher

- ▶ evtl. Prioritätsverletzung wenn nicht der höchspriore gewählt wird
- ▶ Verklemmungsgefahr wenn die Threads unterschiedliche Wartebedingungen haben

- Mit **pthread_cond_broadcast** werden alle wartenden Threads aufgeweckt

- Ein aufwachender Thread wird als erstes den Mutex neu belegen — ist dieser gerade gesperrt bleibt der Thread solange blockiert

L.pdf: 2014-11-21

Threads und Koordinierung in Java (2)

★ Koordinierungsmechanismen

- Monitore: exklusive Ausführung von Methoden eines Objekts
 - ▶ es darf nur jeweils ein Thread die **synchronized**-Methoden betreten
 - ▶ Beispiel:

```
class Bankkonto {  
    int value;  
    public synchronized void AddAmmount(int v) {  
        value=value+v;  
    }  
    public synchronized void RemoveAmmount(int v) {  
        value=value-v;  
    }  
    ...  
    Bankkonto b=....  
    b.AddAmmount(100);
```

- Conditions: gezieltes Freigeben des Monitors und Warten auf ein Ereignis

L.pdf: 2014-11-21