

# 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**



# 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
- Allokation durch Platzierung in einer [Sektion](#)

↪ 12-5

<code>.text</code>	– enthält den Programmcode	<code>main()</code>
<code>.bss</code>	– enthält alle mit 0 initialisierten Variablen	<code>a</code>
<code>.data</code>	– enthält alle mit anderen Werten initialisierten Variablen	<code>b,s</code>
<code>.rodata</code>	– enthält alle unveränderlichen Variablen	<code>c</code>

## ■ Dynamische Allokation – Reservierung zur Laufzeit

- Betrifft lokale auto-Variablen und explizit angeforderten Speicher

<code>Stack</code>	– enthält alle <a href="#">aktuell lebendigen</a> auto-Variablen	<code>x,y,p</code>
<code>Heap</code>	– enthält explizit mit <code>malloc()</code> angeforderte Speicherbereiche	<code>*p</code>



# Speicherorganisation auf einem $\mu\text{C}$

```
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 der Symboltabelle.



# Speicherorganisation auf einem $\mu\text{C}$

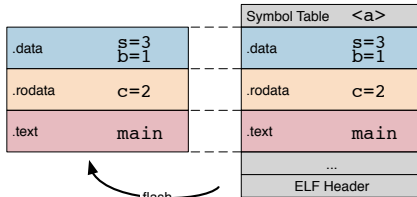
```
int a;                // a: global, uninitialized
int b = 1;            // b: global, initialized
const int c = 2;      // c: global, const

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

Flash / ROM



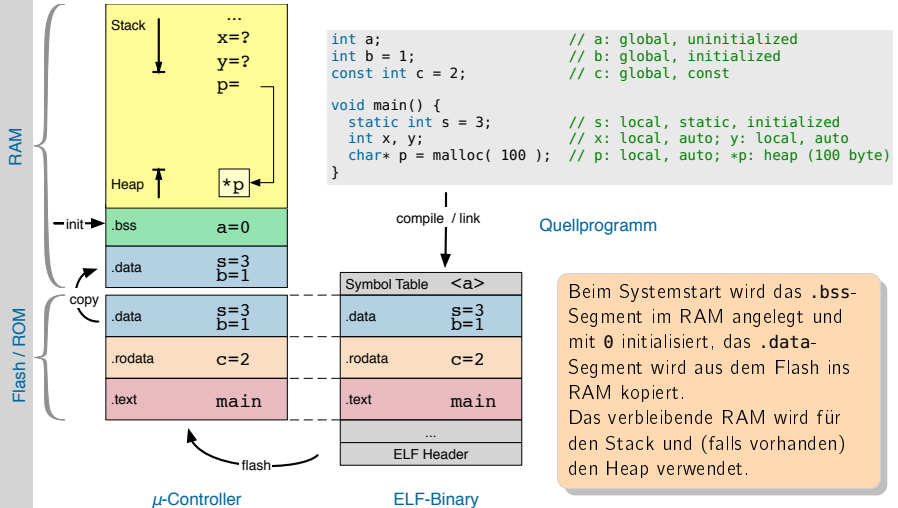
$\mu\text{-Controller}$

ELF-Binary

Zur Installation auf dem  $\mu\text{C}$  werden `.text` und `.[ro]data` in den Flash-Speicher des  $\mu\text{C}$  geladen.



# Speicherorganisation auf einem $\mu\text{C}$



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

- **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** 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



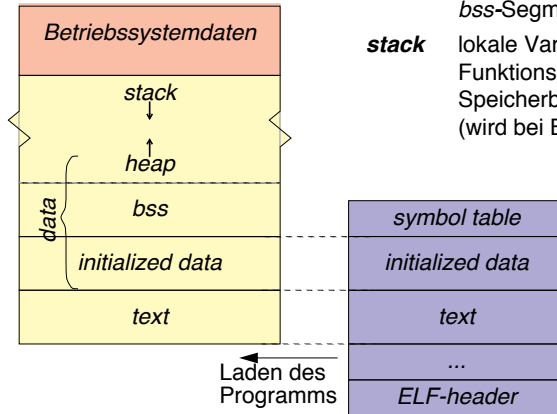
# Speicherorganisation in einem UNIX-Prozess (Forts.)

**text**    Programmcode  
**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)



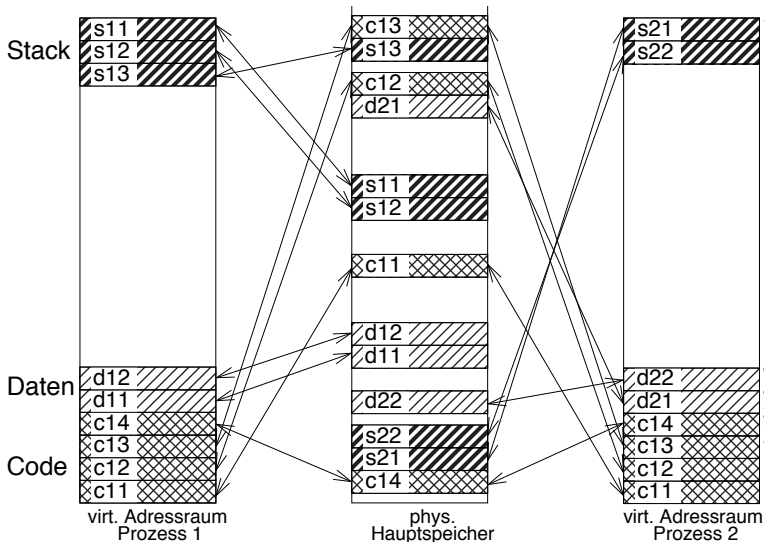
Aufbau eines  
Programms  
ELF-Format)

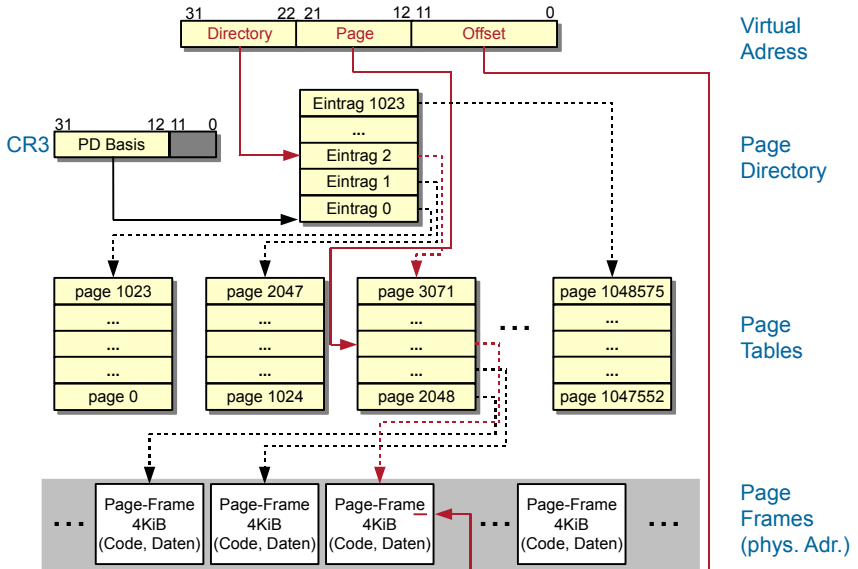


- Die Abbildung von virtuellem Speicher (VS) auf physikalischen Speicher (PS) erfolgt durch **Seitenadressierung** (*Paging*)
  - VS eines Prozesses ist unterteilt in **Speicherseiten** (*Memory Pages*)
    - kleine Adressblöcke, üblich sind z. B. 4 KiB und 4 MiB Seiten
    - in dieser Granularität wird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









# 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

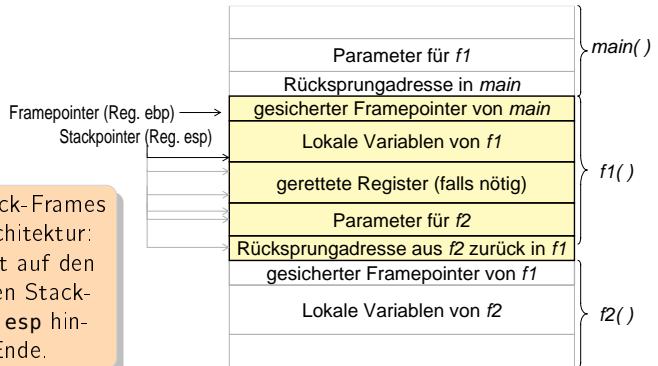
```
#include <stdlib.h>

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! **)
    }
}
```



- 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*

sp fp

return-addr	←2000
fp retten	←1996
a	←1992
b	←1988
c	←1984
	←1980
	←1976
	←1972
	←1968
	←1964
	←1960
	←1956
	←1952
	←1948
	←1944
	←1940
	←1936
	←1932

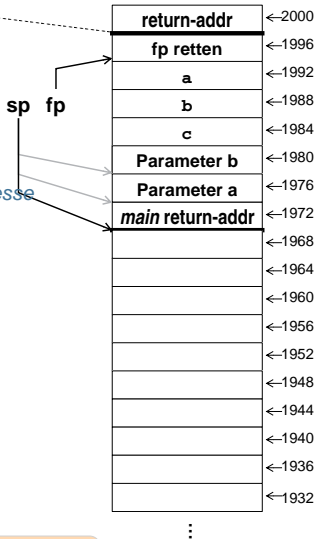
Beispiel hier für 32-Bit-Architektur (4-Byte `ints`), `main()` wurde soeben betreten



# 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

return-addr	←2000
fp retten	←1996
a	←1992
b	←1988
c	←1984
Parameter b	←1980
Parameter a	←1976
main return-addr	←1972
main-fp (1996)	←1968
i [2]	←1964
i [1]	←1960
i [0]	←1956
n	←1952
	←1948
	←1944
	←1940
	←1936
	←1932

⋮

f1() wurde soeben betreten



# Stack-Aufbau bei Funktionsaufrufen

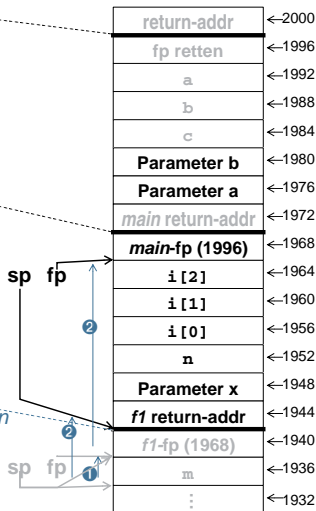
```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);  
}
```

```
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*  
③ return

return-addr	←2000
fp retten	←1996
a	←1992
b	←1988
c	←1984
Parameter b	←1980
Parameter a	←1976
main return-addr	←1972
main-fp (1996)	←1968
i [2]	←1964
i [1]	←1960
i [0]	←1956
n	←1952
Parameter x	←1948
f1 return-addr	←1944
f1-fp (1968)	←1940
m	←1936
⋮	←1932

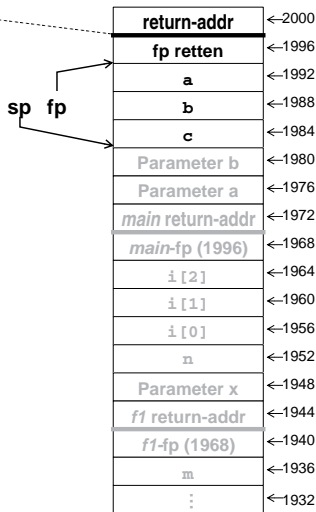
y x sp fp

f2() wird verlassen



# Stack-Aufbau bei Funktionsaufrufen

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



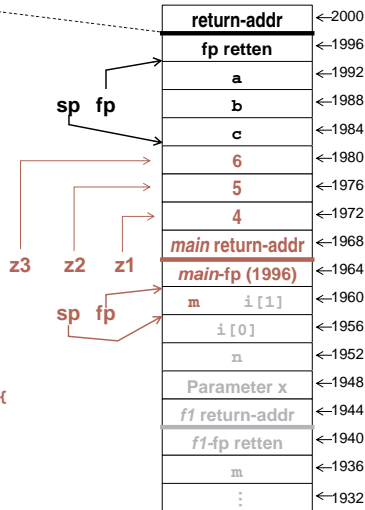
zurück in main()

# 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! → 1-4)

```
lohmann@fau148a:~$ 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!**



# 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**



- Mehrere Prozesse zur Strukturierung von Problemlösungen
  - Aufgaben einer Anwendung leichter modellierbar, wenn sie in mehrere kooperierende Prozesse unterteilt wird
    - z. B. Anwendungen mit mehreren Fenstern (ein Prozess pro Fenster)
    - z. B. Anwendungen mit vielen gleichzeitigen Aufgaben (Webbrowser)
  - Multiprozessorsysteme werden erst mit mehreren parallel laufenden Prozessen ausgenutzt
    - früher nur bei Hochleistungsrechnern (Aerodynamik, Wettervorhers.)
    - durch Multicoresysteme jetzt massive Verbreitung
  - Client-Server-Anwendungen unter UNIX:
    - pro Anfrage wird ein neuer Prozess gestartet
    - z. B. Webserver



# 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





# 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



# 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



# 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



# 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

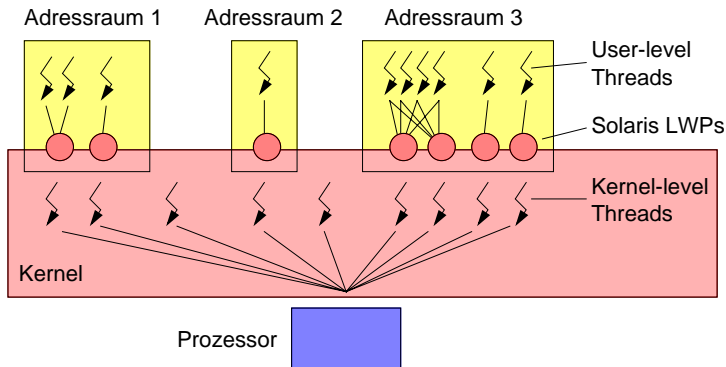
## ▲ Nachteile

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



# 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



- 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!



## ■ 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



# 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 = 1;  
}
```

atomare  
Funktionen

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





# 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!



# Gegenseitiges Warten

## ■ Implementierung mit Hilfe eines Semaphors

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

```
... Thread 1  
P(&barrier);  
f1(result);  
...
```

```
... Thread 2  
result = f2(...);  
V(&barrier);  
...
```

- 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



# 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



# 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**



# 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): Parallele 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 = 1;  
    spin_unlock();  
    sei(); leave_OS();  
}
```

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

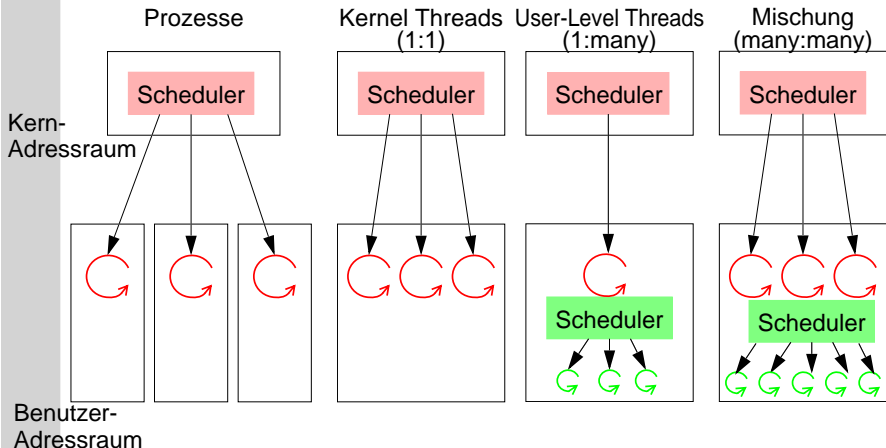




- verschiedene Implementierungen von Thread-Paketen verfügbar
  - reine User-Level-Threads  
eine beliebige Zahl von User-Level-Threads wird auf einem Kernel Thread "gemultiplexed" (*many:1*)
  - reine Kernel-Level-Threads  
jedem auf User-Level sichtbaren Thread ist 1:1 ein Kernel-Level-Thread zugeordnet (*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



# Thread-Konzepte in UNIX/Linux (2)



■ Programmierschnittstelle standardisiert: **Pthreads-Bibliothek**

➡ IEEE-POSIX-Standard P1003.4a

## ■ Pthreads-Schnittstelle (Basisfunktionen):

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

## ■ Funktionen in Pthreads-Bibliothek zusammengefasst

`gcc ... -pthread`



# 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.



# 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.



# 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;
}
```



# Pthreads-Koordinierung

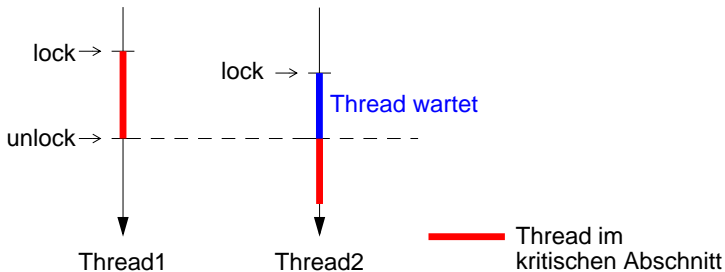
- UNIX stellt zur Koordinierung von Prozessen komplexe Semaphor-Operationen zur Verfügung
  - Implementierung durch den Systemkern
  - komplexe Datenstrukturen, aufwändig zu programmieren
  - für die Koordinierung von Threads viel zu teuer
- Bei Koordinierung von Threads reichen meist einfache **Mutex**-Variablen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



# Pthreads-Koordinierung (2)

## ★ Mutexes

### ■ Koordination von kritischen Abschnitten





## ... 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);
```



## ... 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

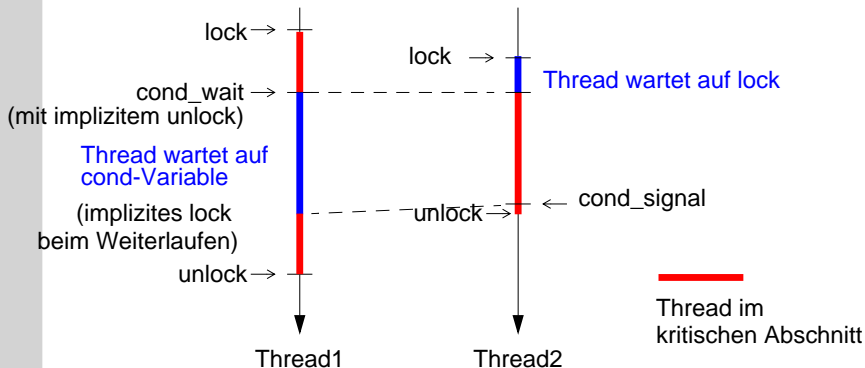
➡ **Condition Variables**



# 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



...

## 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
- Deblocierter Thread muss als erstes den kritischen Abschnitt neu betreten (lock)
- Da möglicherweise mehrere Threads deblockiert wurden, muss die Bedingung nochmals überprüft werden



# 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);  
}
```



## ... 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 höchstpriorität 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



# 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
```



# 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

