

---

# PThreads

**By rac**

Published: 12.01.2008 - 15:12

Dies ist die Liste der wichtigsten Themen, welche für bei SysSoft Abschlussprüfung beherrscht werden sollten. Die Liste orientiert sich frei am PThreads Buch von O'Reilly (ISBN: 1-56592-115-1).

## Kapitel 1

- Was ist ein Thread?
  - Verständnis
  - Speichermodell Prozess -> Thread -> Fork
  - Register (PC, SP, ...)
  - Stack
  - Globale Variablen
  - Heap
  - Programm Text
  - zum Prozess gehörende Ressourcen
- 
- Potentieller Parallelismus
  - Wann ist es sinnvoll Threads einzusetzen?
  - Was sind die Voraussetzungen, die erfüllt sein müssen?
  - In welchen typische Situationen ist ein Programm prallelisierbar?
- 
- Lebenslauf eines Threads
  - Anfang

- 
- Ende
  - Art (detached, ...)
  - Ende des Prozesses
  
  - Was ist unter Race, Deadlock und Starvation zu verstehen?
  - wie äussern sich diese Situationen?
  - weshalb treten sie auf?

## **Kapitel 2**

- Modelle (Boss Slave, Peers, ...)
- Wie sehen die Modelle aus?
- Einsatzgebiete der jeweiligen Modelle
- Voraussetzungen für den Einsatz, zu Beachten bei Implementation (nicht zu viel Kommunikation, nicht zu viel Teilen von Ressourcen etc.)
- Mix von Modellen
  
- Buffering
- Weshalb?
- Welche Punkte sind wichtig bei der Implementation (Lock, Zähler, ...)
  
- Polling
- Was ist das?
- Weshalb ist das nicht gut?
- Wie kann man es verhindern?

- 
- Thread Pools
  - Wieswegen?
  - Wie sieht Implementation aus?

## **Kapitel 3**

- Mutexe
- Wie funktionieren diese?
- Wie werden sie angewendet?
- Warum ist `_trylock` nicht gut?
  
- Bedingungs-Variablen
- Einsatz
- Wie funktionieren diese (Wake Up? Wer wacht auf? Wer bekommt Mutex?)

## **Kapitel 4**

- `pthread_once`
- Einsatz: wann, wie, warum?
  
- Keys
- Einsatz: wann, wie, warum?
  
- Cancellation

- 
- Typen
  - Bedeutung der verschiedenen Typen
  - Funktionsweise

- Cancellation Points
- Cleanup Stack
- Einsatz: weshalb, wie?

- Mutex Attribute
- Inheritance
- Ceiling
- Priority Inversion

## **Kapitel 5**

- Signale
- Verwendung:
  - empfangen/Masken
  - darauf reagieren
  
- fork/exec
- wie funktioniert dies im Pthread Kontext?

- 
- fork-handling Stacks
  - weshalb?
  - wie werden diese eingesetzt?

## C

- typedef
- struct
- Pointer
- referenzieren, dereferenzieren
- zurückgeben aus Funktionen
- Doppelte Pointer: wann werden diese verwendet?

## Related Links

- [Vorlesungs Folien](#)
- [PThread Primer eBook](#)
- [PThread Tutorial von Blaise](#)
- [PThread Tutorial von C. Chapin](#)
- [PThreads Bootcamp Presentation](#)
  
- [Warum Threads](#)
- [Modelle](#)
- [Probleme bei Multithreading](#)
- [Synchronisation](#)
- [Thread pools](#)
- [Management von PThreads](#)
- [PThreads und Unix](#)
- [Multiprozessor Synchronisation](#)
- [Prüfungshilfe](#)

[< Systemsoftware](#) [up](#) [Warum Threads](#) [>](#)

---

## Warum Threads

**By rac**

Published: 12.01.2008 - 15:38

- Anfangs gab es nur langsame Computer -> seriell arbeiten sinnvoll
- Später kamen Multiuser, Timesharing, Multitasking (Unix 197x)
- Warum Multitasking / Multithreading?
- kürzere Ausführungszeiten, bessere Antwortzeiten, Computer besser Ausgenutzt
- Threads sind effizienter innerhalb von Prozessen
  
- Heutige Computer sind Sammlung von Aufgaben
- Threads führen zu kürzeren Ausführungszeiten, besseren Antwortzeiten (fühlt sich besser an!), Computer ist besser ausgenutzt
- Bevor es Threads gab wurde Prozess forking gemacht
- Threads sind effizienter innerhalb von Prozessen
- Sie werden auch "Light Weight Processes" - LWP genannt

## Das Thread Modell:

- prozesseigene Ressourcen: globale Daten, Programm, Instruktionen etc.
- threadeigene Ressourcen: Stack, PC / SC, Register

## PThread Definition

Ein Thread wird innerhalb einer Funktion ausgeführt.

```
kreiere_thread(function, argument)
```

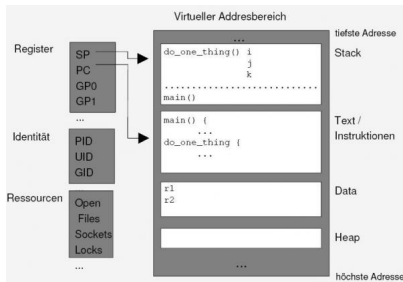
- 
- [Prozesslayout](#)
  - [Potentieller Parallelismus](#)
  - [Konkurrentes Programmieren](#)
  - [Synchronisation](#)
  - [Teilen von Ressourcen](#)
  - [Kommunikation](#)
  - [Identität](#)
  - [Terminierung](#)
  - [Exit Status & Rückgabewert](#)
  - [Bibliotheksaufrufe & Fehler](#)
  - [Warum Threads und nicht Prozesse?](#)
  - [Aufgabenbereiche für Threading](#)
  - [Beispiele für Multithreaded Applikationen](#)

[< PThreads](#) [up](#) [Prozesslayout](#) [>](#)

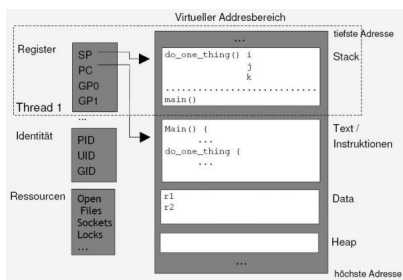
# Prozesslayout

By rac

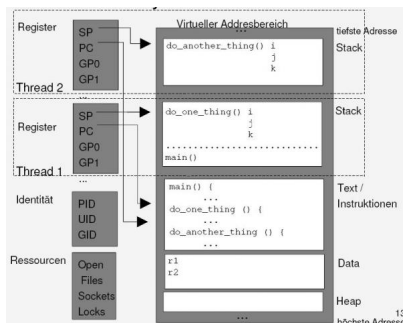
Published: 12.01.2008 - 16:20



# Ein Thread



# Zwei Threads



[< Warum Threads](#) [up](#) [Potentieller Parallelismus](#) [>](#)

---

## Potentieller Parallelismus

**By rac**

Published: 12.01.2008 - 16:34

- Voneinander unabhängige Aufgaben
- Resultat ist unabhängig von der Reihenfolge der Ausführung

## Gründe für Potentiellen Parallelismus

- Blockierender I/O
- Sich überlappende I/O
- Asynchrone Ereignisse (Async events)
- Netz
- Tastatur
- Interrupts
  
- Realtime (Vorgaben aus der Realität)

## Ist Parallelität immer sinnvoll?

Nein, OLPC Projekt als gutes Beispiel (Stromverbrauch senken)

[< Prozesslayout](#) [up](#) [Konkurrentes Programmieren](#) [>](#)

---

## Konkurrentes Programmieren

**By rac**

Published: 12.01.2008 - 16:43

Konkurrent heisst, es könnte möglich sein, dass Parallel Ausgeführt wird.

- Um konkurrentes Programmieren zu ermöglichen, muss dies von der Umgebung / dem System unterstützt werden.

Dazu gehören:

- Parallele Ausführung muss möglich sein
- Synchronisation muss möglich sein
- Kommunikation muss möglich sein

## Parallelität vs. konkurrent

- Parallelität -> Wirklich parallele Ausführung
- Konkurrent -> Kann auch timeshared sein
- Reihenfolge potentiell unbekannt / irrelevant

## Unter UNIX: Multiprocessing

- Fork -> Elternteil / Kind

- 
- eigene PID
  - return Wert ist PID oder 0
  
  - Vor / Nachteile:
  - Speicher / Ressourcenschutz
  - Synchronisation aufwändiger
  - fork einfacher, aber unübersichtlich

## Thread Multiprocessing

- weniger Programmoverhead
- weniger Systemoverhead
- keine Eltern
- Threads sind gleichwertig
- jedoch existiert ein "main" Thread

```
extern int pthread_create(  
    pthread_t *thread_handle,  
    const pthread_attr_t *thread_attribute,  
    void * (*)funktion(void *),  
    void *parameter);
```

[< Potentieller Parallelismus](#) [up Synchronization >](#)

---

## Synchronization

**By rac**

Published: 12.01.2008 - 16:50

- Bei seriellen Programmen -> Reihenfolge des Codes
- Unix multiproc -> waitpid
- Pthreads:
- pthread\_join
- Mutexe (mutual exclusion)
- Bedingungsvariablen -> weniger Wartezeit, feiner, bessere Synchro.
- Mutex muss vom System zur Verfügung gestellt werden

[< Konkurrentes Programmieren](#) [up](#) [Teilen von Ressourcen](#) >

---

## Teilen von Ressourcen

**By rac**

Published: 12.01.2008 - 17:20

- Bei Prozessen wird nichts geteilt
- Bei Threads wird alles geteilt
- Kommunikation über Globale Variablen -> Fehlerquelle
- Potential für Fehler aufgrund der "alles geteilt" Prämisse

[< Synchronization](#) [up](#) [Kommunikation >](#)

---

## Kommunikation

**By rac**

Published: 12.01.2008 - 17:22

- über Mutexe
- über Globale Variablen
- IPC (Inter Process Communication) -> Jedes mal findet ein OS Aufruf statt (=teuer)
- Sockets
- Shared memory
- Message passing

[< Teilen von Ressourcen](#) [up](#) [Identität >](#)

---

## Identität

**By rac**

Published: 12.01.2008 - 17:24

Wer bin ich?

- Vergleiche anhand von PThread Handle
- pthread\_self -> eigene Identität
- pthread\_equal -> Vergleich mit anderer Identität

[< Kommunikation](#) [up](#) [Terminierung >](#)

---

## Terminierung

**By rac**

Published: 12.01.2008 - 17:27

- Prozess wird am Ende von main() beendet
  - "main\_entry" und "main\_exit"
  - werden zu jedem Programm gelinkt
  - zuständig für Rückgabe von Exit Wert
  - Befreiung von Systemressourcen
- 
- Thread wird am Ende der entsprechenden Funktion beendet
  - alternativ pthread\_exit() oder pthread\_cancel()

[< Identität up Exit Status & Rückgabewert >](#)

---

## Exit Status & Rückgabewert

**By rac**

Published: 12.01.2008 - 17:31

- Pthread kann joinable oder detached sein > Rückgabewert
- pthread\_join() nur für joinable Pthread möglich

```
int pthread_join( pthread_t th, void **thread_return);
```

- Rückgabewert liefert entweder pthread\_exit() oder die Funktion, welche pthread\_create() ausführt
- pthread\_create() gibt (void \*) zurück
- evtl. Wert umwandeln (cast)
- Achtung: den Wert PTHREAD\_CANCELLED nicht verwenden! Dieser ist per Def. weder eine gültige Adresse (Pointer!) noch NULL.

[< Terminierung](#) [up Bibliotheksaufrufe & Fehler](#) [>](#)

---

## Bibliotheksaufrufe & Fehler

**By rac**

Published: 12.01.2008 - 17:39

Pthread Bibliotheksaufrufe geben bei Erfolg Null und ansonsten einen Fehlercode zurück, die Fehlercodes sind in errno.h definiert.

```
1.
#include <errno.h>

2.
#include <stdio.h>

3.
...

4.
if (rtn = pthread_create(...)) {

5.
    /* Fehler ! */

6.
    fprintf(stderr, "Fehler: pthread_create, ");

7.
    if (rtn == EAGAIN)

8.
        fprintf(stderr, "Ressourcen reichen nichtn");

9.
```

---

```
else if (rtn == EINVAL)

10.     fprintf(stderr, "Ungültige Argumenten");

11.     exit(1);

12.
}

13.
/* kein Fehler! */

14.
...
```

alternativ kann der Klartext des Fehlers ausgegeben werden, wenn die Funktionalität zur Verfügung steht

```
1.
#include <string.h>

2.
#include <stdio.h>

3.
...
```

---

```
4.
if (rtn = pthread_create(...)) {

5.
    fprintf(stderr, "Fehler: pthread_create, %sn", strerror(rtn));

6.
    exit(1);

7.
}

8.
/* kein Fehler! */

9.
...

10.
/* oder alternativ */

11.
if (rtn = pthread_create(...)) {

12.
    perror("Fehler aufgetreten: ");

13.
    exit(1);

14.
}
```

---

[◀ Exit Status & Rückgabewert](#) [up](#) [Warum Threads und nicht Prozesse?](#) ▶

---

## Warum Threads und nicht Prozesse?

**By rac**

Published: 12.01.2008 - 17:40

- Prozesse teurer (Setupzeit, Speicherverbrauch, IPC Kontextswiches)
- Threads einfacher
- grosser Teil kann in Userspace erledigt werden (pthreads Bibliothek, Kommunikation, Synchronisation)

[< Bibliotheksaufrufe & Fehler](#) [up](#) [Aufgabenbereiche für Threading](#) >

---

## Aufgabenbereiche für Threading

**By rac**

Published: 12.01.2008 - 17:52

- Potentieller Parallelismus
- Unterschiedliche Ressourcen werden verwendet
- Ist unabhängig vom Resultat eines anderen Tasks

Zu Beachten:

Maximale Konkurrenz und minimale Synchronisation.

Je mehr Abhängigkeiten, um so mehr geblockte Tasks, welche aufeinander warten.

- Überschneidende oder blockierende I/O
- Währenddessen andere Arbeit durchführbar?
- Zuweisung einer I/O Aufgabe zu einem Thread
  
- Starke CPU Beanspruchung
- Kryptografische Funktionen, Matrizen, Kompression. etc.
- Während ein oder mehrere Tasks Berechnungen durchführen, kann das Programm auf I/O reagieren
- Eventuell Zuordnung einer CPU zu einer Berechnung
  
- Asynchrone Ereignisse
- Zufällige Intervalle zwischen Daten I/O
- Benutzer I/O, Netzwerk Aktivität, Hardware Interrupts, Sensoren

- 
- Behandlungsroutinen können in einem Thread gekapselt werden
  
  - Realtime scheduling
  - Einige Aufgaben sind wichtiger (haben höhere Priorität)
  - Schnellere Reaktionszeit
  - Feste Bearbeitungszeit
  - Ausführung an spezifischen Zeitpunkten

[< Warum Threads und nicht Prozesse? up Beispiele für Multithreaded Applikationen >](#)

---

## Beispiele für Multithreaded Applikationen

**By rac**

Published: 12.01.2008 - 18:17

- Server: Datenbanken, Fileserver, Printserver, HTTPd, P2P
- Rechnen & Signalprocessing
- Realtime für Server & Multitasking Apps
- effizienter als Multiprocessing
- Scheduling
- weniger Komplexität bei asynch. Programmen
- Threads warten auf Ereignisse, seriellles Programm springt zwischen Kontexten durch Interrupts

[< Aufgabenbereiche für Threading](#) [up Modelle](#) >

---

## Modelle

**By rac**

Published: 12.01.2008 - 18:02

- Richtiges Modell zu finden ist nicht immer offensichtlich
- Trial & Error, mehrere Versuche oder iterative Verbesserung
  
- Modelle unterscheiden sich durch die Art, wie Threads miteinander kommunizieren und durch die ART der Aufteilung der Arbeit:
  - Boss / Worker
  - Peers
  - Pipeline
  
- [Boss / Worker Model](#)
- [Peer Model](#)
- [Pipeline Model](#)
- [Kombination von Modellen](#)
- [Datenbuffering zwischen Threads](#)

[< Beispiele für Multithreaded Applikationen](#) [up](#) [Boss / Worker Model](#) [>](#)

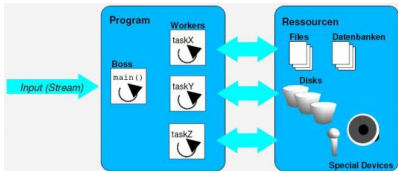
---

## Boss / Worker Model

**By rac**

Published: 12.01.2008 - 19:54

Auch Boss / Slave genannt



Der Boss Thread ist für die Verteilung der Aufgaben Verantwortlich und die Arbeiter Threads führen die zugewiesene Arbeit aus.

Der Boss wartet eventuell auf das Beenden der Arbeit oder der Worker Thread beendet sie selber.

1.  
main { /\* Boss \*/
2.  
  for (;;) {
3.  
    hole aufgabe
4.  
    switch aufgabe
5.  
      case X: pthread\_create task\_X

---

6.            case Y: pthread\_create task\_Y

7.            ...

8.            }

9.            task\_X { /\* Arbeiter \*/

10.           nimm daten, führe arbeit aus, synchroniziere nach bedarf bei geteilten ressourcen

11.           fertig

12.           }

13.           task\_Y ...

Obiges Beispiel verwendet "on demand" Threads

Alternative sind ThreadPools:

- definierte Menge von Threads erstellen
- Arbeit ausführen, auf nächste warten
- Arbeitspakete in FIFO
- weniger Overhead mit Erstellen, mehr mit Verwalten

---

1.  
main { /\* Boss \*/
2.  
    for (anzahl der arbeiter)
3.  
        pthread\_create(... thread\_pool[x], arbeiter\_routine ...);
4.  
    for (;;)
5.  
        hole aufgabe
6.  
        stelle aufgabe in fifo
7.  
        signalisiere neue aufgabe
8.  
    }
9.  
arbeiter\_routine {
10.  
    for(;;)
11.  
        warte bis von boss aufgeweckt
12.  
        hole aufgabe aus fifo
- 13.

---

switch aufgabe

14.

case X: task\_X()

15.

...

16.

}

- gut geeignet für Server
- asynchrone Anfragen / Ereignisse -> Boss  
Erledigung -> Slave
- Kommunikation Boss Worker sollte minimiert werden
- Abhängigkeit zwischen Arbeitern minimieren
- 1 Input channel, N ressourcen

[< Modelle](#) [up](#) [Peer Model](#) [>](#)

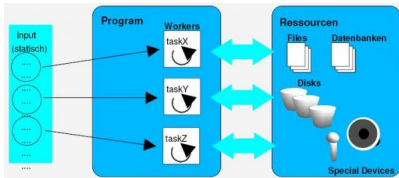
---

## Peer Model

**By rac**

Published: 12.01.2008 - 20:04

Auch Workcrew genannt



- kein Chef
- gleichzeitiges Erledigen der Arbeiten
- ein Thread muss alle anderen erstellen
- bei Boss / Worker ist Boss für Input verantwortlich
- bei Peer:
  - kennt Input im Voraus (Zuständigkeit) eigener Weg um Input zu bekommen
  - teilt Input mit anderen

```
1.  
main() {  
  
2.  
    pthread_create(... thread1, task1 ...);  
  
3.  
    pthread_create(... thread2, task2 ...);  
  
4.  
    ...  
}
```

---

5.  
  signalisiere start

6.  
  warte auf beendigung aller threads

7.  
  aufräumen

8.  
}

9.  
task1() {

10.  
  auf start warten

11.  
  aufgabe ausführen, synchronisieren

12.  
  fertig

13.  
}

## Anwendungsbereich

- fest oder gut definierte Menge von Eingaben
- voneinander unabhängige Aufgaben mit wenig Koordination

---

## Beispiele

- Matrix Operationen
  - Parallel suchen in DBs
  - Primzahlen
  - Generatoren
  - Simulationen
- 
- da kein Boss muss Zugang zu I/O synchronisiert sein
  - bei viel Sync. -> Blockaden -> langsam
  - deshalb geteilte Ressourcen minimieren

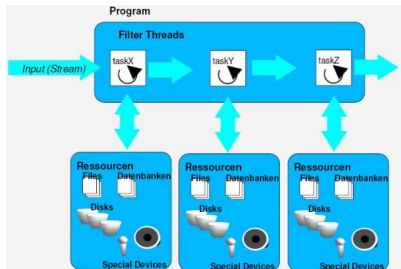
[< Boss / Worker Model](#) [up](#) [Pipeline Model >](#)

---

## Pipeline Model

**By rac**

Published: 13.01.2008 - 14:02



- Input - Strom
- Suboperationen (Filter, Etapen), Einheiten mit eigenem Input
- unabhängig zwischen den Etapen > Parallele Ausführung der Etapen

## Beispiele

- Fließband in einer Autofabrik
- RISC CPU:
  - Operation holen
  - decode
  - berechnen
  - speichern
- > grösserer Durchsatz
- erster Thread erhält Input
- gibt Resultat weiter
- letzter Thread stellt Output bereit

---

## Anwendungsgebiete

- Bildverarbeitung
- Signalverarbeitung
- Textverarbeitung (Unix)
- Filtering

## Pseudocode

```
1.
main() {

2.
    pthread_create(... stufe1 ...);

3.
    pthread_create(... stufe2 ...);

4.
    ...

5.
    warte auf beendigung aller pipeline threads

6.
    aufräumen

7.
}

8.
stufeX() {

9.
    for(;;)
```

- 
10. hole input für von stufe(X1)
  11. führe stufeX bearbeitung aus
  12. gib resultat an stufe(X+1) in der pipeline weiter
  13.  
}

[< Peer Model](#) [up](#) [Kombination von Modellen](#) >

---

## Kombination von Modellen

**By rac**

Published: 13.01.2008 - 14:15

Es kann sinnvoll sein, verschiedene Modelle nach Bedarf zu kombinieren

Beim Pipeline Modell funktioniert das ganze Programm nur so schnell, wie die langsamste Stufe.

Daraus folgt: man kann bei einer Stufe Input parallel an mehrere Threads multiplexen

Wichtig dabei ist, dass bei Pipeline einzelne Stufen ausbalanciert sind, damit sie in etwa gleich lang brauchen.

Bei Peer Modell können einzelne Peers Pipelines verwenden.

[< Pipeline Model](#) [up](#) [Datenbuffering zwischen Threads](#) [>](#)

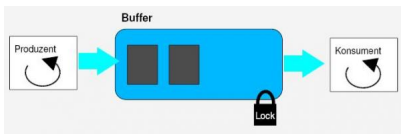
---

# Datenbuffering zwischen Threads

**By rac**

Published: 13.01.2008 - 14:30

- Zum Austauschen von Daten zwischen Threads
- geht in verschiedenen Modellen



Wichtige Begriffe:

- Buffer
- Lock
- Suspend / resume Mechanismus
- Zustandsinfo (Status): wieviel Daten sind im Buffer

1.  
producer() {

2.  
...

3.  
lock shared buffer

4.  
place results in buffer

5.

---

unlock buffer

6.  
wake up any consumer threads

7.  
...

8.  
}

9.  
consumer() {

10.  
...

11.  
lock shared buffer

12.  
while buffer is empty

13.  
release lock and sleep

14.  
wake up and reacquire lock

15.  
fetch data from buffer

16.  
unlock buffer

17.  
...

---

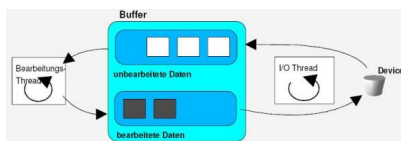
18.  
}

Was Passiert, wenn der Produzent schneller als der Konsument ist?

Dafür gibt es Double Buffering!

Wann kann es angewendet werden:

- wenn Threads miteinander Daten tauschen
- Thread gegenüber anderem Thread sowohl Produzent als auch Konsument
- z.B. bei Peer Modell



[< Kombination von Modellen](#) [up Probleme bei Multithreading](#) >

---

## Probleme bei Multithreading

**By rac**

Published: 13.01.2008 - 14:35

- Bugs!
- Konzentration, Geduld
- Problem meist Shared Resource Management
- Locks
- Race Conditions
- Deadlocks
- Ressource Starvation
  
- Debugging in Kapitel 6 vom PThreads Buch beschrieben
- Technik:
- Lock bevor auf geteilte Ressourcen zugegriffen wird
- Lock loslassen sobald sie nicht mehr verwendet werden
  
- nicht so einfach, wenn Locks dynamisch erstellt werden müssen

## Symptome von MT Fehlern

- nicht korrekte Resultate
- Datenkorruption
- schwierig zu reproduzieren
- 100x OK, 1x falsch

- 
- ein Teil des Programmes steht
  - warten auf Lock, welche nicht losgelassen wird
  - warten auf Bedingung, welche nie erreicht wird

[< Datenbuffering zwischen Threads](#) [up Synchronisation](#) >

---

## Synchronisation

**By rac**

Published: 13.01.2008 - 14:48

- Bsp. Bankomat Geld holen / einzahlen
- atomare Operationen: entweder / oder - isoliert von anderen Transaktionen

## synchronizationsWerkzeuge

- pthread\_join
- Mutexes
- Bedingungsvariablen
- warten auf Ereignisse und Signalisation
- (warten auf Erfüllung von Bedingungen )
- Parameter in Struktur
  
- pthread\_once
- einmalige Ausführung

## Komplexere Werkzeuge

- können aus einfachen Konstrukten kombiniert werden
- Reader / Writer Exclusion
  
- Monitore, Threadsichere Datenstrukturen / Objekte
- Semaphore / Mutexe mit Zählern

- 
- [Mutexe](#)
  - [Bedingungsvariablen](#)
  - [Leser / Schreiber Locks](#)

[< Probleme bei Multithreading](#) [up](#) [Mutexe](#) [>](#)

---

## Mutexe

**By rac**

Published: 13.01.2008 - 15:04

### Benutzung

- Mutex Variablen
- gemeinsame Daten schützen
- exklusiven Zugang zu Ressourcen bieten
- Kritische Sektionen (Critical Sections)
- wie lange kann/muss CS sein?
- beliebig lange
- minimum eine Maschinenanweisung
- tsl rx, lock (Test Set Lock)

### Vorgehen

- jede einzelne Ressource schützen
- Mutex definieren
- lock/unlock bei Zugriff

### Korrekter Code

- keine Änderung während Lese/ Schreibzugriff
- exklusiver, alleiniger Zugriff

---

## Initialisierung

- Benutzung von Mutexen
- Initialisierung
- statisch

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- 
- dynamisch

```
pthread_mutex_t* lock_p;  
lock_p = (pthread_mutex_t *)  
malloc( sizeof( pthread_mutex_t ) );  
pthread_mutex_init( lock_p, NULL);  
                ^^^ pthread_mutexattr_t
```

## Aufrufe

- Initialisierung in main(...)
- oder in Bibliothek: pthread\_once(...)
- Manipulation von Mutexen
- pthread\_mutex\_init
- pthread\_mutex\_lock
- pthread\_mutex\_unlock
- pthread\_mutex\_timedlock
- pthread\_mutex\_trylock
  
- Aufwachen auf Mutexen in zufälliger Reihenfolge

- 
- endliche Wartezeit garantiert

## **Fehlererkennung**

- Fehlererkennung anhand von Rückgabewerten
- viele Fehler nicht standardisiert
- pthread\_mutex\_init kann bsp. zurückgeben
- EAGAIN
- ENOMEM
  
- Fehlbenutzung von Mutexes muss nicht erkannt werden
- Locken von nicht initialisierten Mutexes
- Locken von Mutexes, die man schon besitzt
- Unlock von Mutexes, die man nicht hat

## **pthread\_mutex\_trylock**

- Benutzung weist auf potentiell Designproblem hin
- entgegen der Philosophie von multithreaded Design
- sobald Ressource verwendet werden soll, entweder blockieren oder benutzen
  
- blockiert nicht - Multithreading ist effizient weil es blockiert
- bessere Lösung: neuer Thread, der Arbeit erledigt, während man wartet
- einfacher zu verstehen: kein ifelse Code für Zustandsverwaltung
- Arbeit in Funktion gekapselt

- 
- Polling
  - Overhead
  - Ressource Starvation
  
  - mögliche Verwendung
  - Pollen für Zustandsänderung
  - Deadlocks in komplexen Lock Hierarchien vermeiden (Achtung Konsistenz!)
  - Priority Inversion

## **Alternativen**

- Ressourcen aufteilen
- EreignisSynchronization: Barrieren
- an gewissen Punkten Ergebnisse austauschen
- für Synchronization Bedingungsvariablen verwenden
- oder Zähler, um festzustellen, ob alle schon die Barriere erreicht haben

## **Nachteile**

- Ressourcen überprüfen
- lock, Ausschluss von anderen Threads
- Reader/Writer Lock
- rekursive Locks (kann durch setzen von Lock Attributen erreicht werden: PTHREAD\_MUTEX\_RECURSIVE)

---

## LockContention (Streit, Zank)

- Ideal: mehrere Threads -> höchste Priorität bekommt Lock
- jedoch Priority Inversion
- Lösung: weniger gemeinsame Ressourcen zwischen Threads verschiedener Prioritäten

- [Lock Granularität](#)
- [Mutexe und Prozesse](#)

[< Synchronisation](#) [up Lock Granularität](#) [>](#)

---

## Lock Granularität

**By rac**

Published: 13.01.2008 - 15:11

## Komplexe Datenstrukturen & Lock Granularität

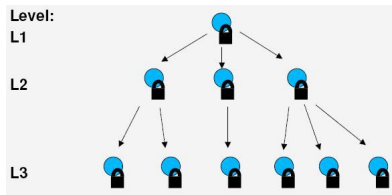
Ziel und Anforderung an Synchronization

- keine Races
- keine Deadlocks
- so wenig Leistungsverlust wie möglich
- Kritische Sektionen so kurz machen, wie möglich
- also keine globalen Locks

## Zugriffsmuster & Granularität:

- read/write on list -> individuelle Locks
- insert -> globale Lock oder kompliziertere Struktur
- grob / fein strukturierte Lock Granularität
- mehr Programmieraufwand
- mehr Verwaltungsaufwand
- mehr Daten
- bessere Parallelität

## Lock Hierarchien



- "deadly embrace" möglich
- Locking Hierarchie
- gleiche Lock Reihenfolge
  
- Support für Lock Hierarchie muss selbst gebastelt werden

[< Mutexe](#) [up](#) [Mutexe und Prozesse](#) >

---

## Mutexe und Prozesse

**By rac**

Published: 13.01.2008 - 15:19

- Mutex nur in Prozess sichtbar
- für mehrere Prozessen gültige Mutexe möglich
- wenn `_POSIX_THREAD_PROCESS_SHARED` definiert
- Achtung: `#include <unistd.h>`, damit `_POSIX` Konstanten bekannt sind!

```
pthread_mutexattr_t  
pthread_mutexattr_init(pthread_mutexattr_t* ma)  
pthread_mutexattr_destroy  
pthread_mutexattr_setshared( &ma, PTHREAD_PROCESS_SHARED)
```

- in shared Memory

[< Lock Granularität](#) [up](#) [Bedingungsvariablen >](#)

---

## Bedingungsvariablen

**By rac**

Published: 13.01.2008 - 15:23

- Condition Variable
  - nicht wirklich eine Variable, sondern ein Werkzeug, wie auch eine Mutex eines ist
  - Mutex - Kontrolle des Zugangs zu Ressourcen
  - Bedingungsvariablen - Kontrolle des Zustands der Ressourcen
  - Benachrichtigungssystem
- 
- printf nicht threadsafe!

## Benutzung

- warten auf Bedingung (Signal)
  - evtl. mit Timeout
- 
- Signal

```
pthread_cond_signal( &cv )  
pthread_cond_broadcast( &cv )
```

- CV Signal != Unix Signal!

## Wenn viele Threads warten

- 1. Priorität
- 2. FIFO

- 
- mit mehreren Prioritäten Starvation möglich!
  - Broadcast alle wachen auf und versuchen zu locken

## **Unerwünschtes Aufwecken (spurious wakeups)**

- Standard erlaubt der Library aufwach Signale zu generieren
- evtl. wegen Implementation im Userspace?

Deshalb:

```
lock(&m)
while(!bedingung)
    cond_wait(&cv, &m)
unlock(&m)
```

## **CV Attribute**

```
pthread_condattr_t
pthread_condattr_init
pthread_condattr_destroy
```

- wie gehabt: `_POSIX_THREAD_PROCESS_SHARED` ist definiert, wenn OS Kommunikation von Bedingungsvariablen zwischen mehreren Prozessen erlaubt.

## **Signal Handler & CVs**

- Standard definiert nicht, was passiert, wenn CV aus Signalhandler signalisiert wird
- Bed.Variablen sind synchron
- Unix Signale sind asynchron!

[< Mutexe und Prozesse](#) [up](#) [Leser / Schreiber Locks](#) [>](#)

---

## Leser / Schreiber Locks

**By rac**

Published: 13.01.2008 - 15:48

- RW-Locks sind semantisch nicht eindeutig
- Queue für Reader / Writer?
- ansonsten Starvation möglich, wenn Reader immer Lock holen

[< Bedingungsvariablen](#) [up](#) [Thread pools](#) >

---

## Thread pools

**By rac**

Published: 13.01.2008 - 15:52

- Alternative zu "on demand" Threads
- Aufwand für's Erstellen / Löschen von Threads

## Eigenschaften eines Threadpools

- Zuvor definierte Anzahl Worker Threads
- Grösse der Queue
- Was passiert wenn queue voll ist?
- Blockieren
- Fehlermeldung

[< Leser / Schreiber Locks](#) [up](#) [Management von PThreads](#) >

---

## Management von PThreads

**By rac**

Published: 13.01.2008 - 15:59

### folgende Pthread Eigenschaften werden untersucht

- Thread Attribute
- Detached Status
- Stack Konfiguration
- Scheduling
  
- pthread\_once
- Keys
- ermöglicht „statische“, Threadeigene Daten in Funktionen oder Bibliotheken
- „globale Datenpointer“ können dadurch jeweils auf einen anderen Datensatz zeigen, je nach dem, welcher Thread auf den Datenpointer zugreift
- jeder Thread kann eigene Kopie/Version der Daten verwenden, d.h. eigenen Zustand innerhalb einer Bibliothek oder Funktion haben
  
- Cancellation
- wann können Thr. beendet werden?
- last minute Stack mit Aktionen, welche vor Terminierung noch ausgeführt werden müssen
  
- Scheduling
- wie wird CPU Zeit verteilt?
- wie werden CPUs verteilt?
- wie lange läuft ein Thread?

- 
- Priorität von Aufgaben
  
  - Mutex Scheduling Attribute
  
  - Massnahmen gegen Priority Inversion

- [Thread Attribute](#)
- [Thread Stack](#)
- [pthread\\_once](#)
- [Keys - Thread-spezifische Daten](#)
- [Beendigung von Threads - Cancellation](#)
- [Scheduling von PThreads](#)

[< Thread pools](#) [up Thread Attribute](#) [>](#)

---

## Thread Attribute

**By rac**

Published: 13.01.2008 - 16:01

- detached / joinable
- Thread Stack Grösse und Platzierung kann manipuliert werden, wenn `_POSIX_THREAD_ATTR_STACKSIZE` bzw. `_POSIX_THREAD_ATTR_STACKADDR` definiert sind
- Scheduling Policy, wenn `_POSIX_THREAD_ATTR_PRIORITY_SCHEDULING` definiert
- wie werden Attribute gesetzt?
- `pthread_attr_t`
- `pthread_attr_init`
- Aufruf spezifischer Funktion, um Attr. zu setzen
- `pthread_create` mit Attributen

## Thread Ausführungs Art

### Detached State

```
pthread_attr_t pth_attr;
```

```
...
```

```
pthread_attr_setdetachedstate(&pth_attr, PTHREAD_CREATE_[DETACHED]JOINABLE);
```

```
pthread_create( &pth, &pth_attr, ... );
```

### Thread Attribute kombinieren

- analog können verschiedene Thread Attribute miteinander kombiniert werden

```
pthread_attr_t pth_attr;
```

```
pthread_attr_init(...);
```

```
pthread_attr_setdetachedstate(...);
```

```
pthread_attr_setstacksize(...);
```

```
pthread_create( &pth, &pth_attr, ... );
```

---

## Thread Attribute löschen

- per `pthread_attr_init` initialisierte Attribute können mittels `pthread_attr_destroy` gelöscht werden
- dies beeinflusst schon erstellte Threads nicht

[< Management von PThreads](#) [up](#) [Thread Stack](#) >

---

## Thread Stack

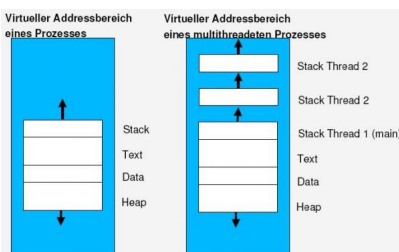
**By rac**

Published: 13.01.2008 - 16:06

### Stack Grösse

- per Thread
- abhängig von:
  - Anzahl Lokaler Variablen
  - Anzahl aufgerufener Routinen
  
- nicht unbedingt portabel:
- vgl. Tanenbaum Speicherverwaltung
- Pages vs. Segmente
- möglich, dass OS Segmente separat verwaltet

```
pthread_attr_t pth_attr;  
...  
pthread_attr_setstacksize( &pth_attr,(size_t)stack_size);  
pthread_attr_getstacksize( ..., &...);
```



### Stack Adresse

```
pthread_attr_t pth_attr;
```

---

...  
pthread\_attr\_setstackaddr( &pth\_attr,(void \*)stack\_addr);  
pthread\_attr\_getstackaddr( ..., &...);  
[< Thread Attribute](#) [up pthread\\_once >](#)

---

## pthread\_once

**By rac**

Published: 13.01.2008 - 16:14

- normalerweise BossThread für Initialisierung der Ressourcen verantwortlich
- nicht immer möglich
  
- ohne pthread\_once schwierig Ressourcen zu erstellen, welche initialisiert werden müssen
- z.B. Bibliothek
- egal wieviel Mal aufgerufen -> eine einzige Ausführung wird garantiert
- kein Thread beendet pthread\_once bevor der erste pthread\_once Thread beendet hat
- Verwendung des "once" Blocks

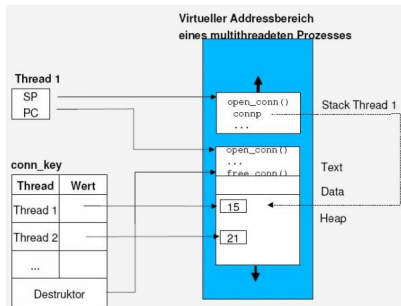
[< Thread Stack up Keys - Thread-spezifische Daten >](#)

---

## Keys - Thread-spezifische Daten

**By rac**

Published: 13.01.2008 - 16:22



- Threads rufen Routinen auf
- Routinelokale Daten kommen und gehen
- Alternativen
- Routine Pointer auf Daten übergeben
- globale Variablen
- selbst hacken mit pthread\_self
  
- Bibliothek
- hat interne Daten
- umschreiben auf MT
- API bleibt gleich
- hat jetzt Threadspezifische interne Daten

---

## Key

- eine Art Pointer
- wenn dieser Pointer von einem Thread verwendet wird, zeigt er jedes Mal auf die Threadeigenen Daten

## Alternativen

- open gibt connection ID zurück
- receive, send (id, ...)
- API wechselt
- Kapselung, Abstraktion?
  
- Array auf Heap anlegen und mit pthread\_self referenzieren
- entweder hässlich oder rel. Aufwendig

## Anwendungsbeispiele

- Modul für Speicher oder File Management
- Profiling Modul
- Exception Manager
- speichert Ort an den bei einer Exception gesprungen werden soll
  
- Zufallszahlen Generator

## Generell

- Bibliotheken oder Routinen mit internem Zustand

- 
- Kontext soll nicht als Parameter übergeben werden
  - üblicherweise wird Kontext in single threaded Programmen als static declariert
  - bei nichtBibliotheken sind Keys nicht unbedingt vorteilhaft
  - globale Variablen
  
  - Key kann auch direkt per pthread\_key\_delete wieder gelöscht werden.
  
  - [Daten einem Key zuordnen](#)
  - [Daten aus einem Key herausholen](#)

[< pthread\\_once up Daten einem Key zuordnen >](#)

---

## Daten einem Key zuordnen

**By rac**

Published: 13.01.2008 - 16:24

```
1.
int open_connection(char* host)

2.
{

3.
    int* connp;

4.
    ...

5.
    connp = (int *)malloc(sizeof(int));

6.
    pthread_setspecific( conn_key, (void *)connp);

7.
    ...

8.
}
```

---

[< Keys - Thread-spezifische Daten](#) [up](#) [Daten aus einem Key herausholen](#) >

---

## Daten aus einem Key herausholen

**By rac**

Published: 13.01.2008 - 16:25

```
1.
int send_data(char* data)

2.
{

3.
    int* saved_connp;

4.
    ...

5.
    saved_connp = pthread_getspecific( conn_key );

6.
    write( *saved_connp, ...);

7.
    ...

8.
}

9.
int receive_data(char** data)
```

---

```
10.
{
11.   int* saved_connp;
12.   ...
13.   saved_connp = pthread_getspecific( conn_key );
14.   read( *saved_connp, ...);
15.   ...
16. }
}
```

[< Daten einem Key zuordnen](#) [up](#) [Beendigung von Threads - Cancellation](#) >

---

## Beendigung von Threads - Cancellation

**By rac**

Published: 13.01.2008 - 16:35

### Warum?

- Ressourcen zurückholen, wenn Thread nicht mehr gebraucht wird
- z.B. Parallele Suche in DB
  
- Cancellability State : J/N
- Cancellability Type : asynchron/verzögert (deferred)
- Warum nicht? Probleme
- Daten Konsistenz -> zurücksetzen
- Locks -> freigeben
  
- Thread sollte nur an sicheren Punkten "cancellable" sein
- Zu jedem Thread gehört ein "Cleanup Stack"

### Cancellation Typen und Zustände

- Thread setzt seine Cancellability selbst
- Wechseln möglich und sinnvoll
- Cancellability kann nur dynamisch zur Laufzeit gesetzt werden
- Default: enabled, deferred cancellation
- verzögerte Cancellation
- Thread muss zuerst einen Cancel Point erreichen

---

## Cancellation Points

- State: enabled, Type: deferred
- Automatic Cancellation Points
- pthread\_cond\_wait
- pthread\_cond\_timedwait
- pthread\_join
- sigwait
- (blockierende Aufrufe)
  
- Vom Programmierer bestimmte Cancellation Points
- pthread\_testcancel
- am besten vor CPU intensiven Arbeiten
  
- Cancellation Points bei System und Library Aufrufen
- Achtung: wenn CANCEL\_ASYNC
- Cancellation auch während
- System und Bibliotheksaufrufen!
- wenn Bibliothek nicht async-cancel sicher ist, vorher CANCEL\_ASYNC ausschalten
  
- Achtung: automatische Cancel Points, können das Konzept durcheinander bringen
- wenn Code Abschnitte davon abhängen, keinen Cancelpoint zu haben, sollte dies explizit dokumentiert werden (`/* ... */`)
- blockierende Aufrufe -> Cancel Point
- System oder Bibliotheken können auch Cancel Points enthalten!

- 
- [Cleanup Stacks](#)

[< Daten aus einem Key herausholen](#) [up Cleanup Stacks >](#)

---

## Cleanup Stacks

**By rac**

Published: 13.01.2008 - 16:37

- werden bei Cancellation aufgerufen
- sind für "Aufräumarbeiten" da
- werden auch bei pthread\_exit ausgeführt
- pthread\_cleanup\_push( aufräum\_routine, (void \*)argument\_p )
- pthread\_cleanup\_pop( exec )
- 1: ausführen
- 0: nicht ausführen
  
- push muss immer ein entsprechendes pop haben
- kann als Macro definiert sein

[< Beendigung von Threads - Cancellation](#) [up Scheduling von PThreads >](#)

---

## Scheduling von PThreads

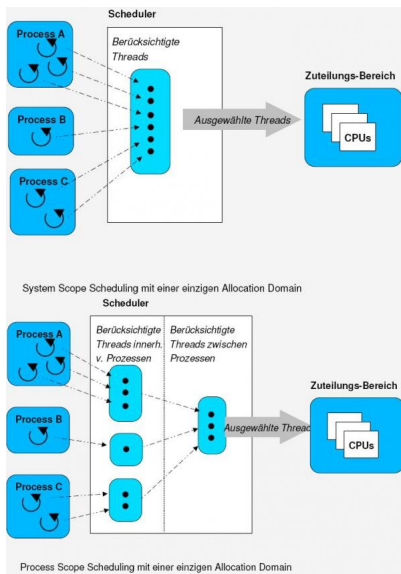
**By rac**

Published: 13.01.2008 - 16:58

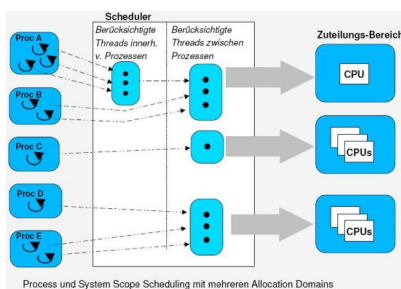
- Scheduling bestimmt, welche Aktivität wie lange laufen soll
- Posix Realtime Erweiterung ermöglicht garantierte Zeiten
- Scheduling nur unterstützt, wenn `_POSIX_THREAD_PRIORITY_SCHEDULING` definiert ist
- Auswahl des nächsten Threads geschieht aufgrund von
  - Priorität
  - Policy (wie werden Threads mit gleicher Priorität abgearbeitet)

## Scheduling Prioritäten und Domänen

- Domäne: eine Gruppierung von CPUs
- Scope: Stufe auf welcher Scheduling von Threads stattfindet
- je nach Fähigkeiten des OS, beinhaltet Scheduling Scope von Threads
  - nur ProcessScope (Userspace Implementationen – green threads)
  - nur SystemScope (1:1 Implementationen – red threads)
  - oder beides (N:M Implementationen)
  
- Gruppierung von CPUs in Domänen, denen Threads zugewiesen werden können
- Domänen werden in Pthreads berücksichtigt, Konfigurationsmöglichkeiten jedoch systemspezifisch



- Process scope = Standard
- Bei Scheduling in nur einem Bereich (Domain) auf ProzessEbene (Scope) können Threads mit niedriger Priorität innerhalb eines Prozesses vom Scheduling ausgeschlossen werden; POSIX ist erfüllt, wenn Threads mit höchster Priorität läuft bis er beendet ist...
- Kann ebenfalls bei mehreren Prozessoren passieren (nicht verwendete CPU)
- Wenn Process Scope Thread blockiert -> System kann ganzen Prozess als blockiert ansehen -> kein Scheduling innerhalb dieses Prozesses mehr.
- System Scope bei Realtime Applikationen besser.



- 
- unter Umständen kann das System besser ausgenutzt werden, wenn weniger Allocation Domains verwendet werden
  - wenn auf vorhergehender Seite B und D gestrichen werden, dann ist A und E's Verhalten besser vorhersehbar, da nicht von höherer Priorität preempted
  - E hat in dem Falle mehrere CPUs zur Verfügung
  - wenn alle Threads in einer Domain geblockt sind (C), dann ist die entsprechende Domain unbenutzt

```
pthread_attr_t custom_sched_attr;  
...  
pthread_attr_init( &custom_sched_attr );  
pthread_attr_setscope( &custom_sched_attr , PTHREAD_SCOPE_SYSTEM);  
pthread_create( &thread, &custom_sched_attr , ...);  
...  
pthread_attr_getscope( &custom_sched_attr , &attr)
```

## Laufbereite und Blockierte Threads

- Auswahl für's Scheduling der auszuführenden Threads
- runnable, blocked?
- genügend freie CPUs?
- Policy & Priorität?
  
- Warteschlangen für Threads der gleichen Priorität
- Warteschlangen werden nach Prioritäten abgearbeitet
- sobald ein Thread wieder lauffähig wird, "preemptet" er einen Thread niedrigerer Priorität

## Scheduling Policy

SCHED\_FIFO

- solange abarbeiten bis blockiert oder beendet. Ein blockierter Thread steht wieder hinten an.

---

## SCHED\_RR

- jeder Thread kann während eines "Quantums" an Zeit laufen, dann kommt der nächste dran

## SCHED\_OTHER

- Default. Übernimmt das System Scheduling. Üblicherweise wie SCHED\_RR aber mit Prioritätsanpassung. Ein Thread, der blockiert, bevor er sein Quantum ausgenutzt hat wird bevorzugt -> Interaktive Threads werden CPUintensiven Threads bevorzugt.
- Nur wenn `_POSIX_THREAD_PRIORITY_SCHEDULING` definiert ist. Bei Linux in `unistd.h`.

## Benutzung von Policies und Prioritäten

- Realtime Anwendungen
- SCHED\_FIFO
- hohe Priorität
- soll immer ausgeführt werden, wenn möglich
- funktionieren wie EventHandler - sobald was passiert wird's bearbeitet und abgeschlossen
  
- andere Anwendungen SCHED\_RR bzw. SCHED\_OTHER
- Posix verlangt mindestens 32 PrioritätsStufen
- dynamisch Prioritäten ändern

```
pthread_setschedparam( &thread, &sched_param);
```

- Achtung damit kann man sich selbst preempten ;-)
- Es ist möglich Scheduling Eigenschaften an erstellte Threads weiterzuerben

---

## Scheduling bei Mutexen

Problem: Prioritäts-Umkehrung (Priority inversion)

- Scheduling bei Mutexen wird komplex und kann das Problem noch verschlimmern

### Priority Ceiling

- Mutex besitzt eigene Priorität
- sobald ein Thread die Mutex lockt, wird seine Priorität auf diejenige der Mutex gehoben
- Priority Ceiling nur möglich, wenn `_POSIX_THREAD_PRIO_PROTECT` definiert ist
  
- `PTHREAD_PRIO_NONE` kein Priority Ceiling
- `PTHREAD_PRIO_PROTECT` Priority Ceiling
- `PTHREAD_PRIO_INHERIT` Prioritätsvererbung

### Prioritätsvererbung

- Mutex bekommt die Priorität des höchsten wartenden Threads
- bei unserem Beispiel würde der Statistik Thread nur eine höhere Priorität bekommen, wenn der Kontroll Thread am warten ist
- Priority Inheritance nur möglich, wenn `_POSIX_THREAD_PRIO_INHERIT` definiert ist

[< Cleanup Stacks](#) [up PThreads und Unix](#) >

---

## PThreads und Unix

**By rac**

Published: 13.01.2008 - 17:33

- Threads und Signale
- Threadsichere Bibliotheksfunktionen und Systemaufrufe
- Cancellationsichere Bibliotheksfunktionen und Systemaufrufe
- Threadblockierende Bibliotheksfunktionen und Systemaufrufe
- Threads & Prozess Verwaltung
- Multiprozessor Speicher Synchronisierung
- Threads wurden später in Unix eingeführt
- System musste an Threads angepasst werden
  
- div. SystemFunktionen, welche auf Prozessen arbeiten, müssen mit Threads funktionieren
- Signale
- fork, exec
- Bibliotheken, welche Systemaufrufe machen
- threadsafe?
- cancelationsafe?
  
- Pthread Standard definiert Methoden, welche mit herkömmlichem Process Handling kompatibel sind

- 
- [Signale](#)
  - [Threadsichere Bibliotheks- und Systemfunktionen](#)
  - [Cancelsichere Bibliotheks- und Systemfunktionen](#)
  - [Thread blockierende Bibliotheks- und Systemfunktionen](#)
  - [Threads und Prozessmanagement](#)
  - [Fork Handling Stacks](#)
  - [Exiting](#)

[< Sheduling von PThreads](#) [up](#) [Signale >](#)

---

# Signale

**By rac**

Published: 13.01.2008 - 17:54

- Signale sind asynchron ("überraschend")
  - ein Signal, welches den Programmfluss unterbricht kann an jeder Stelle im Code eintreffen
  - Division by Zero
  - Segmentation Fault
  - Timer I/O
  - I/O Operation fertig
  - SIGUSR
  - Suspend
  - SIGTERM
- 
- Bedingung für Thread Signalhandling Mechanismus
  - muss kompatibel bleiben
  - es muss festgelegt werden können, wer ein Signal bekommt/sieht
  - was ist einem Thread im Signalhandler erlaubt, sodass dies die sonstige Arbeit des Threads nicht beeinflusst?
- 
- ein Signal geht immer an einen Prozess, und nicht an einen Thread
  - Maske, welche die Reaktion eines Prozesses auf ein Signal festlegt (sigaction) wird von allen Threads geteilt
  - jeder Thread hat Maske, welche definiert, welche ankommenden Signale er sieht
  - Verhalten der Pthread Werkzeuge selbst innerhalb eines Signalhandlers ist nicht definiert!

- 
- keine Synchronisation mit pthread Mitteln innerhalb eines Signalhandlers möglich

## Traditionelle Signalverarbeitung

- sigaction erlaubt es einem Signal eine Aktion zuzuordnen
- SIG\_IGN Signal ignorieren
- SIG\_DFL Default. Reaktion ist abhängig vom empfangenen Signal. Die meisten Signale beenden den Prozess, einige werden jedoch ignoriert. SIGSTOP bzw. SIGTSTP unterbrechen und SIGCONT nimmt den Prozess wieder auf.
- bearbeiten Signal wird empfangen und eine spezifische Handling Routine ausgeführt
- die Reaktionen auf SIGSTOP und SIGKILL können nicht verändert werden
- bei Signal wird Prozess unterbrochen und Handler ausgeführt. Danach wird die Ausführung am Unterbrechungsort fortgesetzt
- SIGCONT setzt einen Prozess fort welcher durch wait, sigsuspend, sleep oder pause unterbrochen wurde

## Signalhandling und Threads

Beim Empfang, werden verschiedene Signale unterschieden:

- Synchrone Signale
- ausgelöst durch Operation des Programms
- SIGFPE - Teilung durch Null
- SIGSEGV - illegaler Speicherzugriff
- SIGPIPE - Pipe ist kaputt
- ...

- 
- Signal wird an den Thread ausgeliefert, der das Problem ausgelöst hat
  - explizit mit pthread\_kill innerhalb eines Prozesses an einen anderen Thread gesendete Signale (kein IPC!)
  
  - Asynchrone Signale
  - können nicht einfach auf einen bestimmten Thread zurückgeführt werden
  - asynchron dahingehend, dass sie unabhängig von den Aktivitäten bzw. dem ausgeführten Code sind
  - üblicherweise Job Kontroll Signale
  - SIGALRM
  - SIGHUP
  - SIGINT
  - SIGKILL
  - SIGUSRx
  
  - diese können, sofern nicht maskiert, von einem beliebigen Thread bearbeitet werden

| Wie wurde Signal ausgelöst | Was hat's ausgelöst             | Ziel des Signals    | Bearbeitung des Signals               |
|----------------------------|---------------------------------|---------------------|---------------------------------------|
| synchron                   | System aufgrund einer Exception | spezifischer Thread | durch auslösenden Thread              |
| synchron                   | pthread_kill                    | spezifischer Thread | durch Ziel-Thread                     |
| asynchron                  | externer Prozess mittels kill   | Prozess             | durch Thread mit entsprechender Maske |

## Per Thread Signalmasken

---

- wie Prozesse auch besitzen Threads Signalmasken, welche anzeigen, welche Signale sie entgegennehmen (deblockiert) und welche nicht (blockiert)

- Masken werden geerbt
- vom Prozess
- vom Thread, der fork ausgeführt hat
- vom Thread, welcher pthread\_create ausgeführt hat

- pthread\_sigmask
- ein ankommendes, asynchrones Signal wird an genau einen Thread ausgeliefert

## **Threads in Signal Handlern**

- Aufrufe, welche aus Signal Handlern gemacht werden können werden als "asynchronous signalsafe" bezeichnet
- sind "reentrant", da mehrere Ausführungsstränge gleichzeitig in ihnen aktiv sein können
- siehe Liste von Systemfunktionen p170 im PThreads Buch
- Pthread Funktionen nicht in der Liste
- Synchronization über sem\_post
- sigwait verwenden, um auf Signal zu warten
- Signale maskieren
- Thread erstellen, welcher sigwait durchführt
- demaskieren

## **Probleme mit Signalen**

- 
- manche POSIX Funktionen geben EINTR zurück, wenn sie von einem Signal gestört werden
  - sie müssen darauf noch einmal ausgeführt werden

[< PThreads und Unix up Threadssichere Bibliotheks- und Systemfunktionen >](#)

---

## Threadssichere Bibliotheks- und Systemfunktionen

**By rac**

Published: 13.01.2008 - 18:00

- gleiche Probleme, wie wir bisher gesehen haben
- Bibliotheken und Funktionen werden als "reentrant" bezeichnet
- mehrere Ausführungsstränge möglich
- üblicherweise mit Eliminierung jeglicher statischer Daten gelöst
  
- "threadsafe" analog für Threads
- wird mit Mutexen, Cond. Variablen etc. gelöst
  
- Code ist nicht threadsicher?
- Lösung: Arbeitsbuffer lokal machen
  
- weiteres Problem ist die Verwendung von errno in Bibliotheken
- errno ist eine globale Variable
- Pthread Standard definiert errno als Makro
- glibc definiert errno als (\* \_\_error\_location())

- 
- Pthread Standard definiert Funktionen, welche threadsafe sein müssen
  - fast alle POSIX.1 Funktionen
  - Ausnahmen
  - Funktionen, welche Pointer auf statische Daten zurückgeben
  - zeitkritische Funktionen
  - werden oft in Schleifen für I/O eingesetzt
  - `getc`, `getchar`, `putc`, `putchar`
  
  - Pthread bestimmt, dass diese Funktionen threadsafe sein müssen
  - es werden jedoch auch schnellere, nicht threadsichere Funktionen angeboten
  - `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked`
  
  - Funktionen für's DateiLocking
  - `flock` blockiert eine Datei nur zwischen Prozessen
  - funktioniert nicht zwischen Threads
  - eine Lösung mittels Mutex wäre machbar
  - spezielle Funktionen für Locking zwischen Threads
  - `flockfile`, `ftrylockfile`, `funlockfile`

- 
- Alternativen zu Funktionen, die Pointer auf statische Daten zurückgeben
  - readdir gibt nächsten Verzeichniseintrag zurück
  - localtime gibt Pointer auf lokale Zeit zurück
  - siehe Liste p180
  - haben jeweils ein \_r am Schluss
  - readdir\_r, localtime\_r, ...
- 
- Versionen von Bibliotheken
  - oft gibt es Bibliotheken welche nicht threadsichere Funktionen enthalten parallel zu threadsicheren Bibliotheken
  - es muss also u.U. darauf geachtet werden mit welcher Bibliothek man linkt
- 
- Benutzung von threadunsicheren Funktionen in MT Programmen
  - wie gehabt als geteilte Ressource behandeln

```
pthread_mutex_t funktions_mutex;  
#define sichere_funktion(x)  
    pthread_mutex_lock( &funktions_mutex );  
    unsichere_funktion();  
    pthread_mutex_unlock( &funktions_mutex );
```

[< Signale up Cancelsichere Bibliotheks- und Systemfunktionen >](#)

---

## Cancelsichere Bibliotheks- und Systemfunktionen

**By rac**

Published: 13.01.2008 - 18:30

### Asynchronous CancellationSafe Functions

- grundsätzlich sind POSIX Bibliotheksfunktionen nicht asynchron cancelsicher
- d.h. wenn Programm async. cancelsafe sein soll, dann sollte man einen Wrapper rund um die Funktion machen

```
#define async_cancel_safe_read(fd, buf, amt)
{
  int oldtype;
  pthread_setcanceltype( PTHREAD_CANCEL_DEFERRED, &oldtype);
  if ( read( fd, buf, amt) < 0 )
    perror("read"), exit(-1);
  pthread_setcanceltype(oldtype, NULL);
  pthread_testcancel();
}
```

### Cancellation Punkte in System- und Bibliotheksaufrufen

- bei Deferred Cancellation kann ein Thread nur an wohl definierten Punkten unterbrochen werden
- grundsätzlich dienen alle Funktionen, welche tendenziell lange brauchen (meistens I/O) als Cancellation Points
- siehe Liste p184

[< Threadsichere Bibliotheks- und Systemfunktionen](#) [up Thread blockierende Bibliotheks- und Systemfunktionen](#) [>](#)

---

## Thread blockierende Bibliotheks- und Systemfunktionen

**By rac**

Published: 13.01.2008 - 18:31

- ursprünglich blockierten die meisten I/O Funktionen einen Prozess
- um dem zu entkommen konnte man nichtblockierende Funktionen verwenden, indem man z.B. das O\_NONBLOCK (POSIX) bzw. O\_NDELAY (BSD) Flag setzte
- POSIX verlangt nun, dass die meisten blockierende Funktionen nur den aufrufenden Thread anhalten
- was macht man, wenn man eine Funktion erwischt, welche den ganzen Prozess blockiert?
- ganze Applikation anhalten lassen
- einen zweiten Prozess forken, welcher die Funktion abarbeitet
- eine nichtblockierende Alternative verwenden

[< Cancelsichere Bibliotheks- und Systemfunktionen](#) [up](#) [Threads und Prozessmanagement](#) >

---

## Threads und Prozessmanagement

**By rac**

Published: 13.01.2008 - 18:32

exec

- Code („Text“) wird durch den des neuen Programmes ersetzt

fork

- exakte Kopie des Speichers wird erstellt
- inklusive Zustand, Filedeskriptoren, Semaphoren...
- Prozess bekommt neue PID

oft wird nach einem fork sofort ein exec ausgeführt.

- fork aus einem Thread ausführen
- wie fork aus einem Prozess, ausser dass nur der aufrufende Thread im Kind "überlebt"
- nur der vom Thread und die von seinen Eltern benutzen Stacks werden kopiert
- Locks werden mitkopiert!
- Heap wird mitkopiert!

---

- pthread\_atfork

[< Thread blockierende Bibliotheks- und Systemfunktionen](#) [up](#) [Fork Handling Stacks](#) [>](#)

---

## Fork Handling Stacks

**By rac**

Published: 13.01.2008 - 18:36

- Fork Handling Stack hilft bei der Vorbereitung und beim Aufräumen nach einem Fork
- folgende Fork Handling Stacks existieren
- prepare Stack - wird vor den Fork ausgeführt
- parent Stack - wird nach dem Fork im Elternteil ausgeführt
- child Stack - wird nach dem Fork im Kind ausgeführt
  
- die Stacks stellen eine Prozessweite Ressource dar, auf die jeder Thread zugreifen kann
- Stack wird nicht abgebaut
- komplex
- Alternativen
- prefork - bevor Threads erstellt werden, wird geforkt
- wenn forkexec ausgeführt wird: Programm in Shared Library umwandeln und so aufrufen
- Ersatzeltern Model - ein Prozess wird weggeforkt, welcher single-threaded bleibt und nur dazu dient neue Prozesse wegzuforken

[< Threads und Prozessmanagement](#) [up](#) [Exiting >](#)

---

## Exiting

**By rac**

Published: 13.01.2008 - 18:38

### Exec Aufrufe aus Threads:

- nur Thread überlebt, welcher exec aufruft und fängt in der main Routine an

### Exit und Threads

- ein Prozess wird beendet
- sobald ein Thread exit aufruft
- main beendet wird
- ein fatales Signal empfangen wird
  
- bei Beendigung werden alle Threads beendet und alle Ressourcen freigegeben
- bei einem direkten Aufruf von `_exit` garantiert das System die Aufräumarbeiten nicht

[< Fork Handling Stacks](#) [up](#) [Multiprozessor Synchronisation](#) >

---

## Multiprozessor Synchronisation

**By rac**

Published: 13.01.2008 - 20:50

- der Pthread Standard verlangt bei einigen Gelegenheiten Schreibzugriffe zu synchronisieren
- bei einem pthread\_mutex\_lock muss z.B. garantiert sein, dass alle Schreibzugriffe in den Speicher committed wurden
- normalerweise über eine SpeicherBarrieren-Anweisung implementiert, welche Cache in Speicher zurückschreibt
- siehe Liste p191 für Funktionen welche Speichersynchronization garantieren

[< Exiting](#) [up](#) [Prüfungshilfe](#) [>](#)

---

## Prüfungshilfe

**By rac**

Published: 20.01.2008 - 16:24

### Threads

#### Was ist ein Thread?

Ein Thread ist ein Teil eines Programms, der unabhängig von anderen Programmteilen oder noch nicht vorhandenen Ergebnissen ausgeführt werden kann.

#### Speichermodell Prozess -> Thread -> Fork

Fork() kopiert alle Daten des Prozesses (wenn aus einem Thread aufgerufen, nur alle Stammdaten und den aufrufenden Thread sowie die eltern, restliche threads im prozess werden verworfen)

**Register (PC, SP, ...)**

**Stack**

**Globale Variablen**

**Heap**

**Programm Text**

**zum Prozess gehörende Ressourcen**

#### Potentieller Parallelismus

- voneinander unabhängige Aufgaben

d.h. Resultat ist unabhängig von der Reihenfolge der Ausführung!

#### Wann ist es sinnvoll Threads einzusetzen?

Wenn die Aufgaben unabhängig voneinander und von der Zeit / Reihenfolge ausgeführt werden können  
Gründe für pot. //

- 
- blockender I/O
  - sich überlappenden I/O
  - Asynchrone Ereignisse: Netz, Tastatur, Interrupts
  - Realtime

### **Was sind die Voraussetzungen, die erfüllt sein müssen?**

- Wenn unterschiedliche Ressourcen verwendet werden
- Wenn Unabhängigkeit von Resultaten anderer Tasks besteht

zu beachten:

max. Konkurrenz und minimale Synchronisation

je mehr Abhängigkeiten, um so mehr geblockte Tasks, welche aufeinander warten

### **In welchen typischen Situationen ist ein Programm parallelisierbar?**

- Starke CPU Beanspruchung
- kryptografische Funktionen, Matrizen, Kompression, etc.
- während ein oder mehrere Tasks Berechnungen durchführen, kann das Programm auf I/O reagieren
- evtl. Zuordnung einer CPU zu einer Berechnung
  
- asynchrone Ereignisse
- zufällige Intervalle zwischen Daten I/O
- Benutzer I/O, Netzwerk Aktivität, Hardware Interrupts, Sensoren
- Behandlungsroutinen können in einen Thread gekapselt werden unterschiedliche Behandlungsprioritäten
- einige Aufgaben sind wichtiger
- schnellere Reaktionszeit
- feste Bearbeitungszeit
- Ausführung an spezifischen Zeitpunkten

- 
- Beispiele für MT Apps
  - Server: Datenbanken, Fileserver, Printserver, HTTPd, P2P
  - Rechnen & Signalprocessing
  - Realtime für Server & Multitasking Apps
  - effizienter als MP
  - Scheduling
  - weniger Komplexität bei asynch. Programmen
  - Threads warten auf Ereignisse serielles Programm springt zwischen Kontexten durch Interrupts

## **Lebenslauf eines Threads**

### **Anfang**

Wenn er mit `pthread_create()` erstellt wird

### **Ende**

wenn er zur letzten Anweisung im Code des Threads kommt oder zu einer `return` Anweisung im Hauptcode des Threads.

Z.B. `return(NULL)`

### **Art (detached, joinable)**

Detached: Ist losgelöst vom Aufrufer

Joinable: Aufrufer wartet auf Beendigung

### **Ende des Prozesses**

---

Wenn das Ende des Main Blocks erreicht ist oder der Prozess ein Kill-Signal vom OS bekommt.

## **Was ist unter Race, Deadlock und Starvation zu verstehen?**

### **Race condition**

Es wurden Annahmen von zeitlichen Abläufen im Programmcode angenommen, die zur Runtime jedoch nicht bestanden.

Es kann zu Abstürzen oder auch nur zu einem Fehlverhalten führen, wie zum Beispiel falsche Berechnungen usw.

### **Deadlock**

Alle Threads warten auf ein Ereignis eines Threads, der befindet sich aber z.B. in einem Loop aus dem er nicht mehr herauskommt.

Das Programm steht still.

### **Starvation**

Z.B. Ein Leser-Thread hat eine höhere Priorität als der Schreiber-Thread auf eine Mutex, der Leser-Thread macht ein Polling und verunmöglicht so den Zugriff für den Writer-Thread, der eigentlich die Infos abliefern möchte.

Das Programm steht still oder ist seeeeeeeehr langsam.

## **Modelle (Boss Slave, Peers, ...)**

### **Wie sehen die Modelle aus?**

Script ab Seite 38ff

### **Einsatzgebiete der jeweiligen Modelle**

### **Voraussetzungen für den Einsatz, zu beachten bei Implementation**

nicht zu viel Kommunikation, nicht zu viel Teilen von Ressourcen etc.

---

## **Mix von Modellen**

### **Buffering**

Daten Austausch zwischen Threads

#### **Weshalb?**

Da man sonst Gefahr läuft, das ein writer warten muss weil er schneller ist als der reader oder umgekehrt.

#### **Welche Punkte sind wichtig bei der Implementation (Lock, Zähler, ...)**

### **Polling**

#### **Was ist das?**

Wenn ein Thread in einer schlaufe (Timer) immer wieder nachfragt ob die Ressource frei ist oder geändert hat.

#### **Weshalb ist das nicht gut?**

Frisst CPU Time für andere Threads und macht eine grosse Last.

#### **Wie kann man es verhindern?**

Mit Condition Variables zum Beispiel.

## **Thread Pools**

#### **Weswegen?**

#### **Wie sieht Implementation aus?**

---

## **Mutexe**

- gemeinsame Daten schützen
- exklusiven Zugang zu Ressourcen bieten
- Kritische Sektionen (Critical Sections)

### **Wie funktionieren diese?**

Script Seite 73ff

### **Wie werden sie angewendet?**

1. jede einzelne Ressource schützen
2. Mutex definieren
3. lock/unlock bei Zugriff

### **Warum ist `_trylock` nicht gut?**

Wen man es benutzen muss, hat man bereits im Design Fehler gemacht.  
Es entspricht nicht dem Konzept von Mutexen ein Polling zu machen.

Script Seite 78ff

## **Bedingungs-Variablen**

- Kontrolle des Zustands der Ressourcen
- Benachrichtigungssystem

## **Einsatz**

1. warten auf Bedingung (Signal) evtl. mit Timeout
2. Signal:

```
pthread_cond_signal( &cv )  
pthread_cond_broadcast( &cv )
```

## **Wie funktionieren diese**

(Wake Up? Wer wacht auf? Wer bekommt Mutex?)

1. Priorität
2. FIFO

-> mit mehreren Prioritäten Starvation möglich!

1. Broadcast - alle wachen auf und versuchen zu locken

## **pthread\_once**

Dieser code wird nur beim ersten aufrufenden thread ausgeführt für das ganze programm (z.B. initialize routines)

- egal wieviel Mal aufgerufen -> eine einzige Ausführung wird garantiert
- kein Thread beendet pthread\_once bevor der erste pthread\_once Thread beendet hat

## **Einsatz**

---

wann, wie, warum?

## **Keys**

- Thread spezifische Daten (vergleich mit Hash Tables)
- eine Art Pointer
- wenn dieser Pointer von einem Thread verwendet wird, zeigt er jedes Mal auf die Threadeigenen Daten

Script Seite 100 / 115ff

## **Einsatz**

wann, wie, warum?

## **Cancellation**

- Thread sollte nur an sicheren Punkten "cancellable" sein
- Zu jedem Thread gehört ein "Cleanup Stack"

Script Seite 123ff

## **Typen**

enabled, deferred und automatic cancellation

## **Bedeutung der verschiedenen Typen**

---

Enabled: Thread cancellation ist allgemein eingeschaltet für diesen thread  
deferred cancelation: Thread muss zuerst einen Cancel Point erreichen (verzögerte cancellation)  
Automatic: Werden von pthreads gesetzt

### **Funktionsweise**

Cleanup stacks werden aufgerufen

### **Cancellation Points**

Automatic:

- pthread\_cond\_wait
- pthread\_cond\_timedwait
- pthread\_join
- sigwait

Source code:

- pthread\_testcancel

### **Cleanup Stack**

- sind für "Aufräumarbeiten" da
- werden auch bei pthread\_exit ausgeführt

pthread\_cleanup\_push( aufräum\_routine, (void \*)argument\_p )  
pthread\_cleanup\_pop( exec )

---

1: ausführen 0: nicht ausführen

- push muss immer ein entsprechendes pop haben: kann als Macro definiert sein

### **Einsatz**

weshalb, wie?

### **Mutex Attribute**

#### **Inheritance**

#### **Ceiling**

Script Seite 147ff

### **Priority Inversion**

### **Signale**

Script Seite 156ff

### **Verwendung**

**empfangen/Masken**

**darauf reagieren**

**fork/exec**

Script Seite 175ff

**wie funktioniert dies im Pthread Kontext?**

**fork-handling Stacks**

---

**weshalb?**

**wie werden diese eingesetzt?**

## **C**

### **typedef**

Die Programmiersprache C bietet dem Benutzer die Möglichkeit, mit dem Schlüsselwort `typedef` eigene Datentypen zu definieren. Die Schaffung solcher eigenen Datentypen dient dabei im Wesentlichen dazu, die Lesbarkeit der Programme zu steigern. Durch das Schlüsselwort `typedef` werden zunächst Synonyme erzeugt, also gleichbedeutende Datentypen. Diese Synonyme können dann genauso verwendet werden, wie die von C vorgegebenen.

Am häufigsten werden selbstdefinierte Datentypen dazu verwendet, zusammengesetzte Datenstrukturen wie Arrays oder Strukturen zu bezeichnen.

#### **Beispiel**

```
typedef int Ganze_Zahl;  
typedef char Zeichen;  
Ganze_Zahl a = 12;  
Zeichen b;  
b = 'e';
```

### **struct**

Eine Struktur ist eine Folge von einem oder mehreren Elementen, die inhaltlich zusammengehören und über den Strukturnamen und den Elementnamen ansprechbar sind. In der Deklaration einer Struktur muss das Schlüsselwort `struct` erscheinen. Die einzelnen Elemente einer Struktur müssen nicht vom selben Typ sein, es sind auch komplexere Elemente wie Strukturen oder Arrays als Elemente einer Struktur zugelassen. Um die einzelnen Elemente anzusprechen gibt es zwei Wege. Zum einen kann ein Element über den Punkt (Bezeichner) angesprochen werden. Dazu muss der Name der Struktur und des Elementes, beide durch einen Punkt getrennt, eingegeben werden. Die andere Möglichkeit besteht darin die einzelnen Elemente über den Pfeil (Zeiger) anzusprechen. Für die Deklaration einer Struktur gibt es verschiedene Möglichkeiten. Hierbei muss zwischen Strukturmuster, das den Aufbau der Struktur angibt und keinen Speicherplatz benötigt, und der eigentlichen Struktur unterschieden werden.

#### **Beispiel 1**

```
/* Ohne Strukturmuster */  
struct  
{  
    char name[20];  
    long verdienst;  
} akte; /* Definition einer Struktur akte */
```

---

Der Struktur wird der Name akte, über den sie dann auch angesprochen wird, zugeordnet.

## Beispiel 2

```
/* Strukturmuster und Struktur */  
struct AKTE  
{  
    char name[20];  
    long verdienst;  
};  
struct AKTE akte; /* Definition der Struktur */
```

Zunächst wird ein Strukturmuster mit dem Namen AKTE definiert. Durch das Strukturmuster wird kein Speicherplatz belegt; die eigentliche Zuordnung des Strukturnamens erfolgt erst in der letzten Zeile.

## Pointer

Ein Pointer (Zeiger) ist ein Datentyp, bei dem Speicheradressen verwaltet werden. Eine Variable von einem Pointertyp kann also jeweils eine konkrete Speicheradresse (z.B. von einer anderen statischen Variablen) beinhalten. Pointer sind insoweit dynamisch, als sie mit ihrer Deklaration zunächst keinen weiteren Speicherplatz zugewiesen bekommen als den für die eigentliche Adresse.

### referenzieren

Datentyp \*zeigervariable;

Der Datentyp des Zeigers muss vom selben Datentyp wie der sein, auf den er zeigt (referenziert).

### dereferenzieren

Zeigervariable = &variable

### zurückgeben aus Funktionen

Zeiger\_Rückgabety \*Funktionsname(Parameter)

### Doppelte Pointer: wann werden diese verwendet?

- Zum iterieren

---

[< Multiprozessor Synchronisation](#) [up](#) [Memory Management >](#)