technische universität
dortmund

Betriebssysteme

Tafelübung 2. Thread-Synchronisation

https://ess.cs.tu-dortmund.de/DE/Teaching/SS2019/BS/

Horst Schirmeier

horst.schirmeier@tu-dortmund.de
https://ess.cs.tu-dortmund.de/~hsc

AG Eingebettete Systemsoftware
Informatik 12, TU Dortmund

mit Material von Olaf Spinczyk,
Universität Osnabrück

Agenda

- Besprechung Aufgabe 1: Prozesse verwalten
- Fortsetzung Grundlagen C-Programmierung
- Aufgabe 2: Thread-Synchronisation
  - POSIX
  - UNIX-Prozesse vs. POSIX-Threads
  - Funktionen von Pthreads
  - Mutex
  - Condition Variables
  - Vergleich: exec..() , fork() , pthread_create()

Besprechung Aufgabe 1

- → Foliensatz Besprechung

Grundlagen C-Programmierung

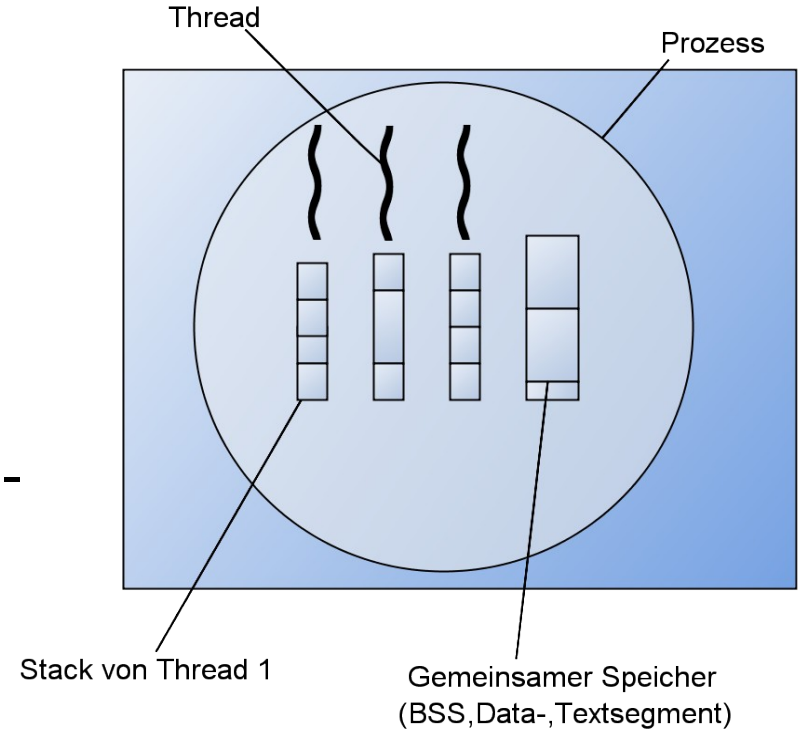
- → Foliensatz C-Einführung (Folie 41 bis Ende)

POSIX

- „Portable Operating System Interface“
- von IEEE entwickelte Schnittstellenstandardisierung unter UNIX
  - ermöglicht einfache Applikationsportierung
- POSIX definiert u.a. eine standardisierte API zwischen Betriebssystem und einer Applikation

UNIX-Prozess vs. POSIX-Threads

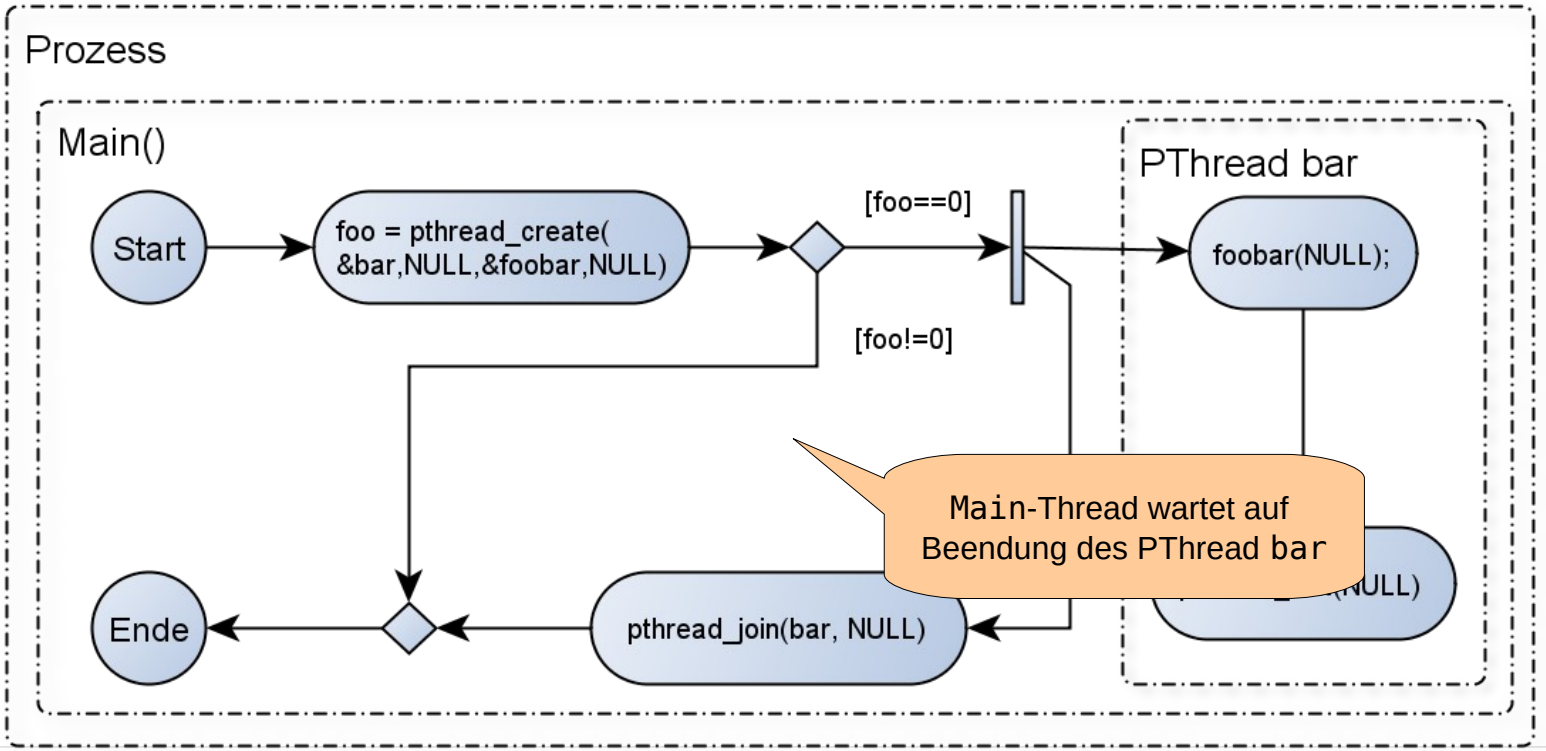
- UNIX-Prozesse: *schwergewichtig* (haben einen eigenen Adressraum)
- POSIX-Threads (kurz Pthreads): *leichtgewichtig*
  - ein Prozess kann mehrere Threads haben (teilen sich den gleichen Adressraum)
  - im Linux Kernel sind sogenannte *linux_threads* deklariert (je nach Kernel unterschiedlich)
  - Pthreads bieten standardisierte Schnittstelle
  - Pthreads verwenden intern System-aufrufe
  - jeder Pthread hat eine eigene ID (Typ pthread_t: unsigned long int)



Funktionen für Pthreads (Übersicht)

- pthread_create( );
- pthread_exit( );
- pthread_join( );
- pthread_self();

benötigen:
#include <pthread.h>

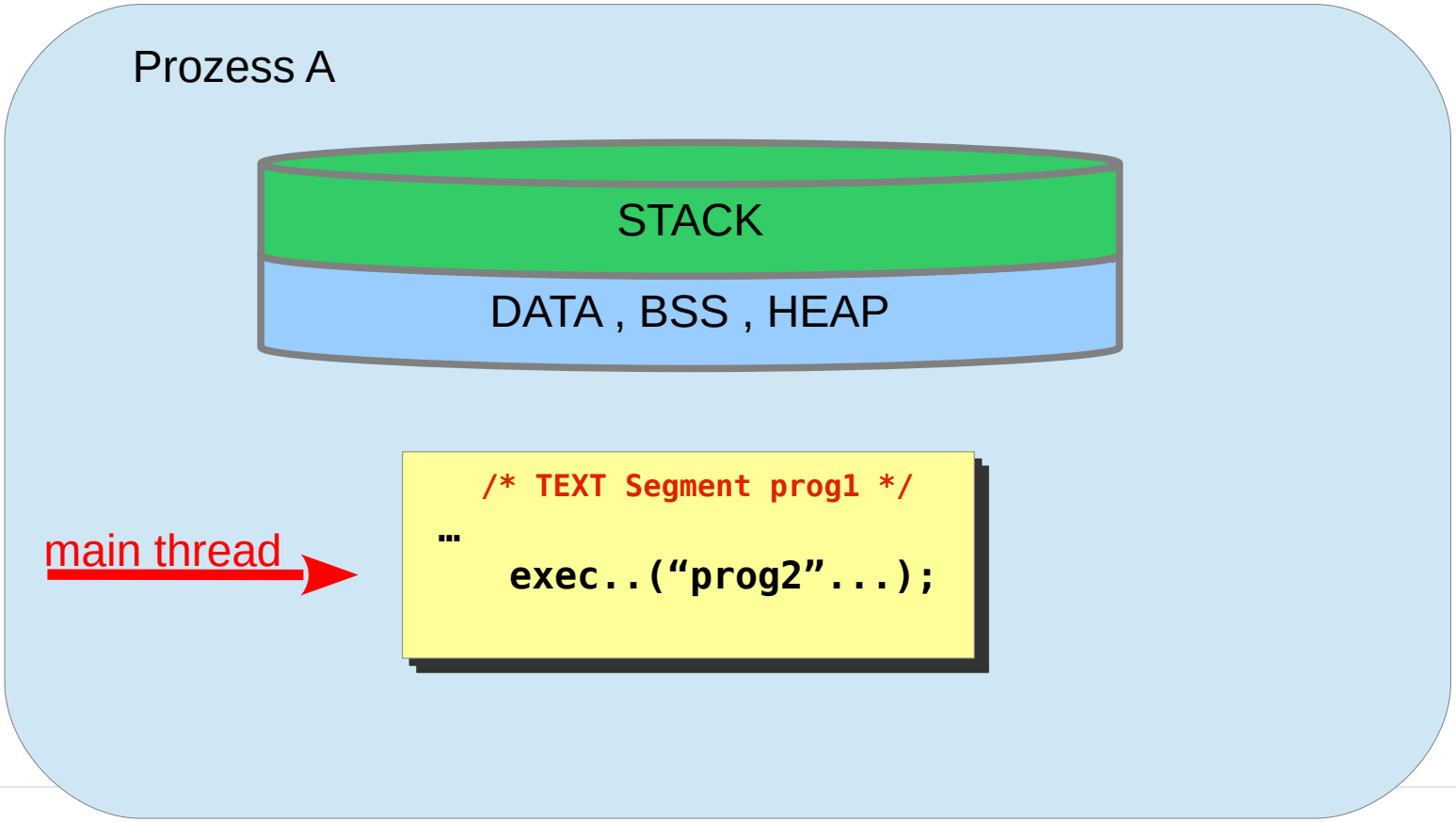


Pthread-Beispiel

```c
#include <pthread.h>
#include <stdio.h>

void* Hello(void *arg) {
    printf("Hello! It's me, thread!”);
    pthread_exit(NULL);
}

int main(void) {
    int status;
    pthread_t thread;

    status = pthread_create(&thread, NULL, &Hello, NULL);
    if (status) { /*Fehlerbehandlung*/ }

    status = pthread_join(thread, NULL);
    if (status) { /*Fehlerbehandlung*/ }

    pthread_exit(NULL);
}
```

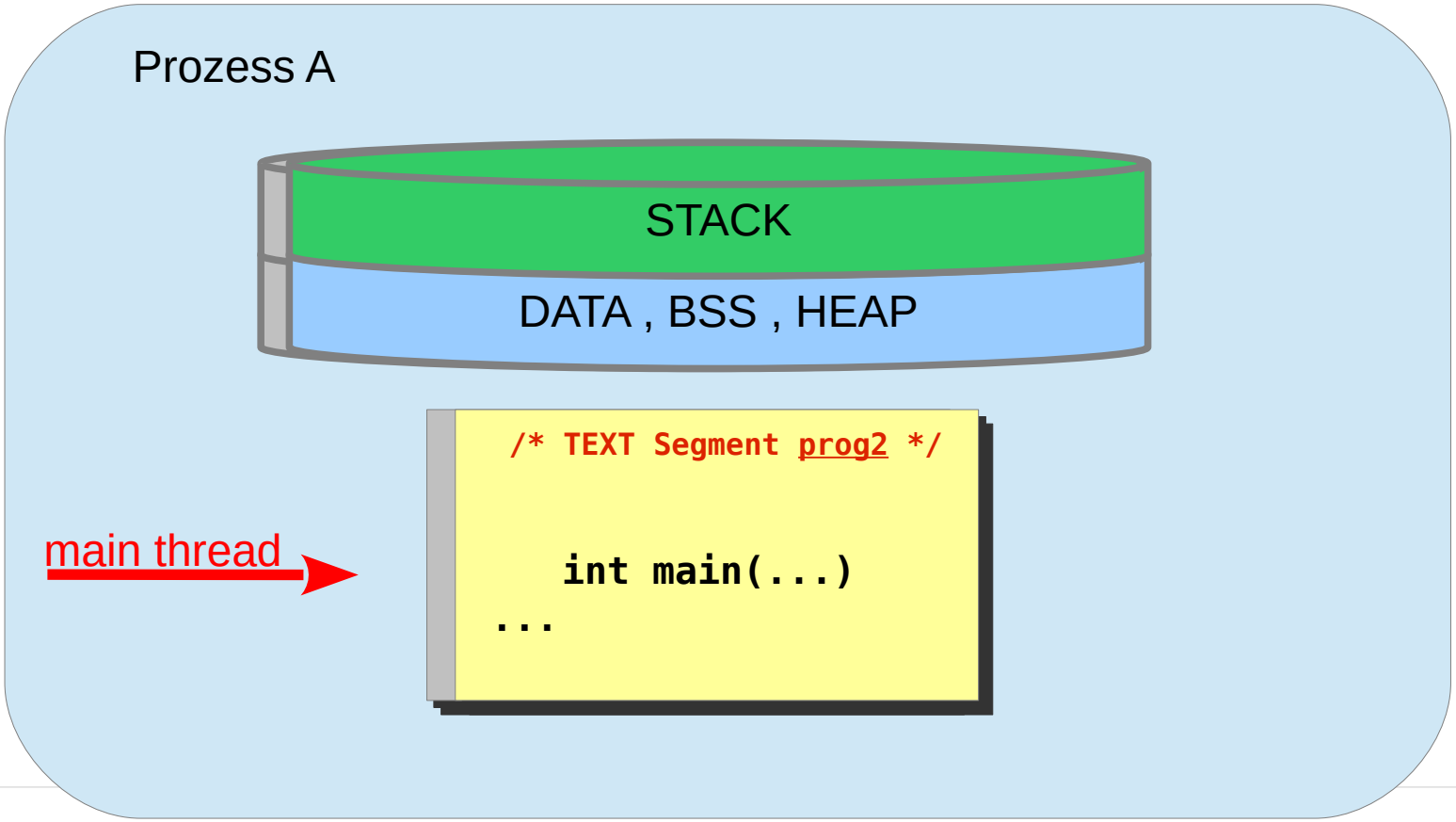
Vergleich: exec..() fork() pthread_create()

- Überlagerung eines Prozesses
- Keine gemeinsamen Daten
- schwergewichtig



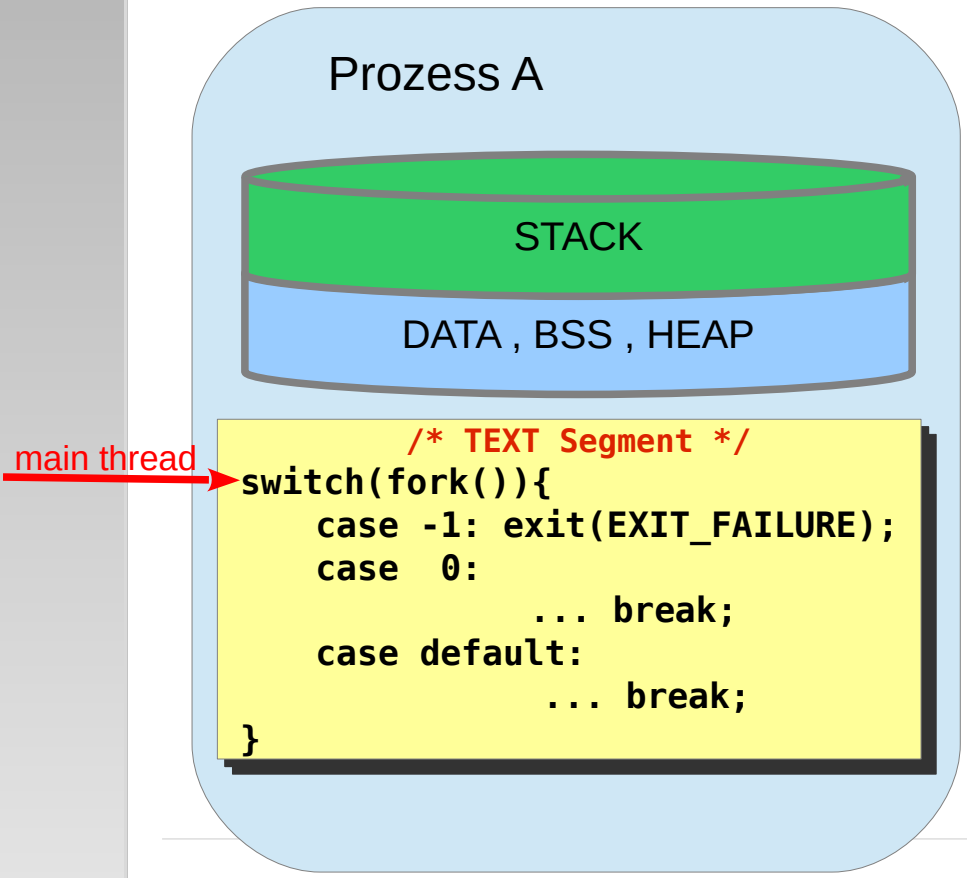
Vergleich: `exec..()` `fork()` `pthread_create()`

- Überlagerung eines Prozesses
- Keine gemeinsamen Daten
- schwergewichtig



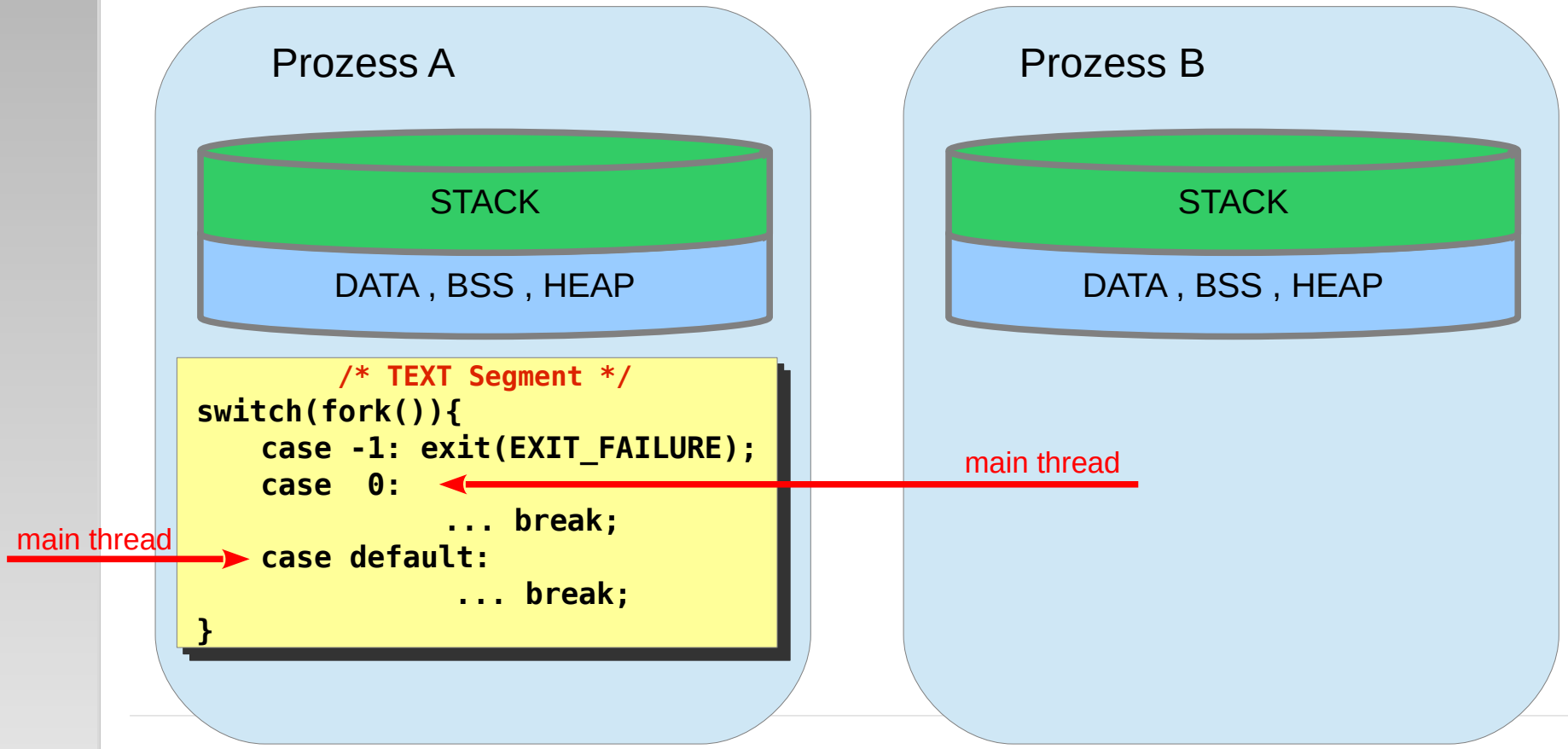
Vergleich: `exec..()` `fork()` `pthread_create()`

- Verzweigung eines Prozesses
- Gemeinsame Daten: shared-memory
- schwergewichtig



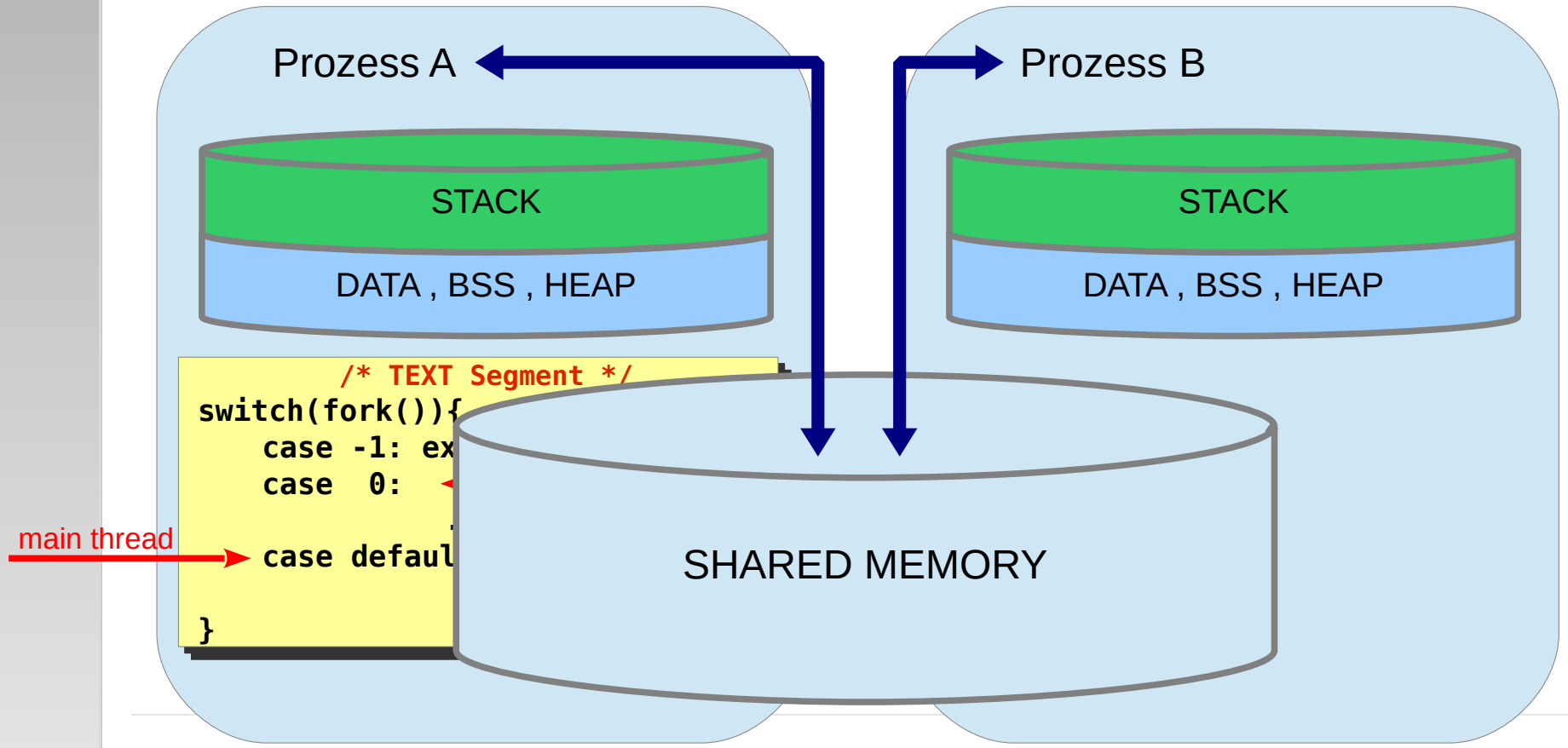
Vergleich: `exec..()` `fork()` `pthread_create()`

- Verzweigung eines Prozesses
- Gemeinsame Daten: shared-memory
- schwergewichtig



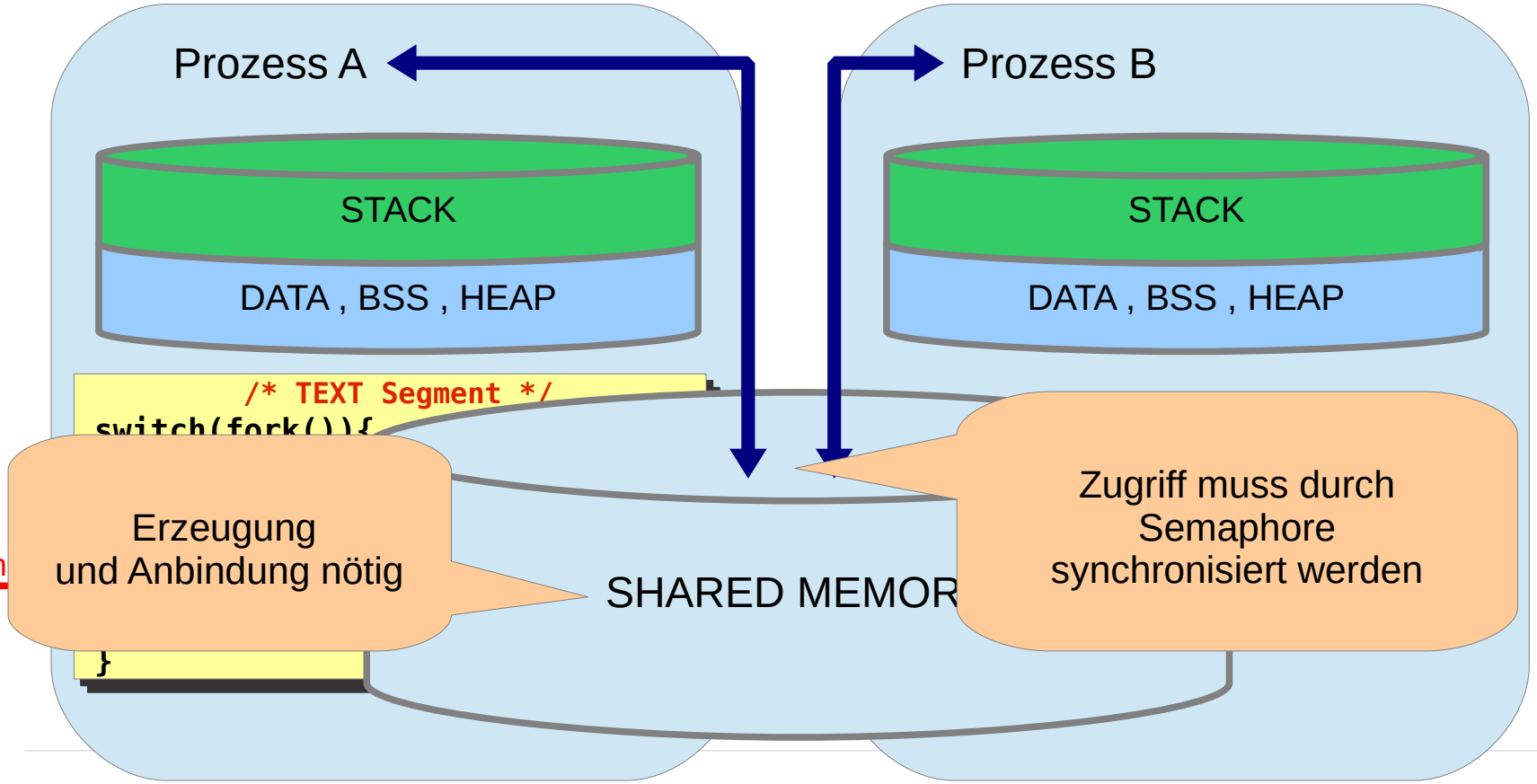
Vergleich: `exec..()` `fork()` `pthread_create()`

- Verzweigung eines Prozesses
- Gemeinsame Daten: shared-memory
- schwergewichtig



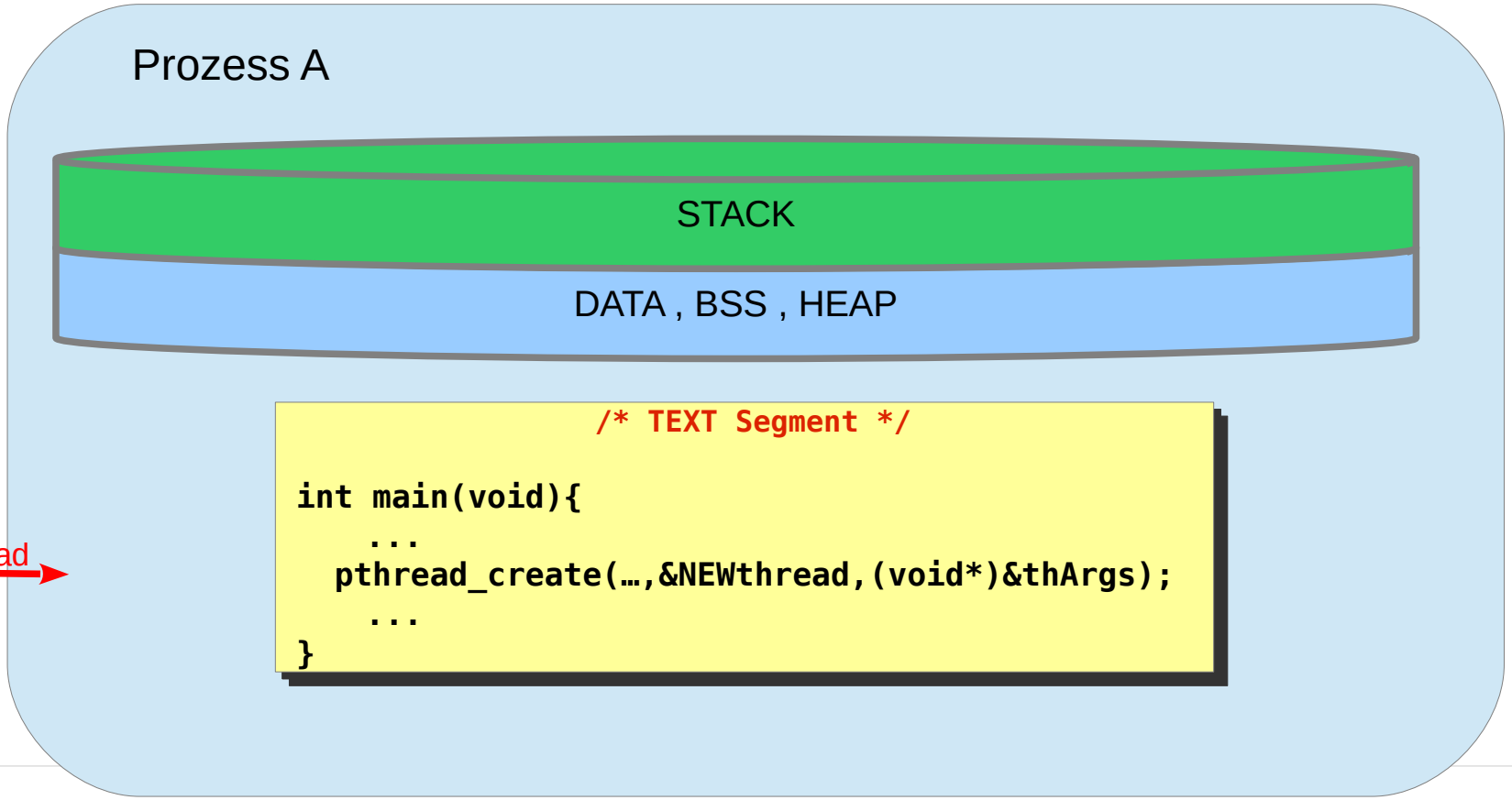
Vergleich: `exec..()` `fork()` `pthread_create()`

- Verzweigung eines Prozesses
- Gemeinsame Daten: shared-memory
- schwergewichtig



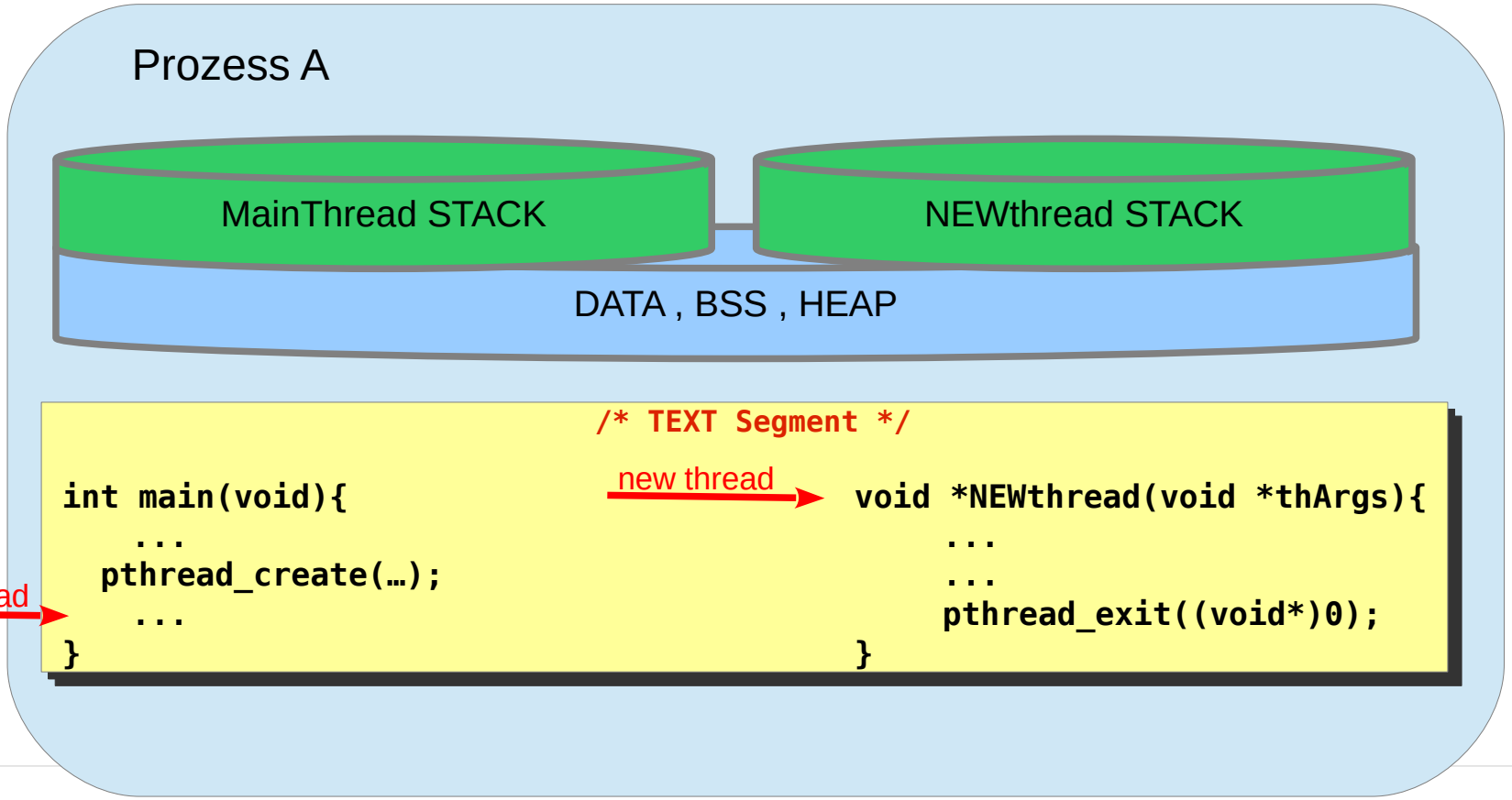
Vergleich: `exec..()` `fork()` `pthread_create()`

- Aufteilung eines Prozesses
- Gemeinsame Daten: Data, BSS, Heap, shared-memory
- leichtgewichtig



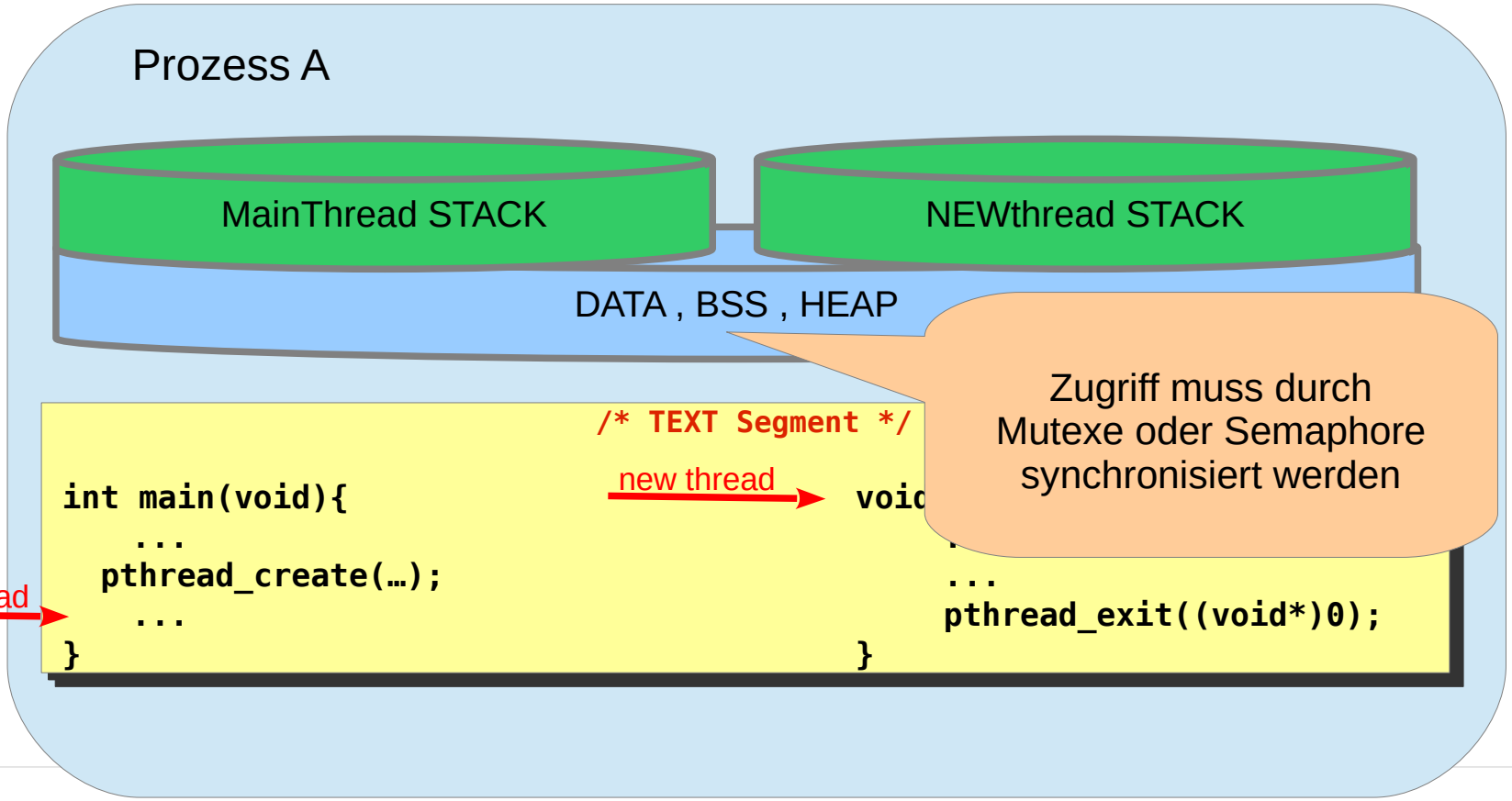
Vergleich: `exec..()` `fork()` `pthread_create()`

- Aufteilung eines Prozesses
- Gemeinsame Daten: Data, BSS, Heap, shared-memory
- leichtgewichtig



Vergleich: `exec..()` `fork()` `pthread_create()`

- Aufteilung eines Prozesses
- Gemeinsame Daten: Data, BSS, Heap, shared-memory
- leichtgewichtig



Semaphor (*semaphore*)

- Eine ”nicht-negative ganze Zahl“,
  für die zwei **unteilbare Operationen** definiert sind:

-1 **P** (hol. prolaag, „erniedrige“; auch *down*, ***wait***)

  - hat der Semaphor den Wert 0, wird der laufende Prozess blockiert
  - ansonsten wird der Semaphor um 1 dekrementiert

+1 **V** (hol. verhoog, „erhöhe“; auch *up*, ***signal***)

  - auf den Semaphor ggf. blockierter Prozess wird deblockiert
  - ansonsten wird der Semaphor um 1 inkrementiert

- Eine **Betriebssystemabstraktion** zum Austausch von Synchronisationssignalen zwischen nebenläufig arbeitenden schwergewichtigen oder leichtgewichtigen Fäden.

Synchronisationsmuster (1)

- “gegenseitiger Ausschluss”

```
/* gem. Speicher */
Semaphore elem;
struct list l;
struct element e;
```

```
/* Initialisierung */
elem = 1;
```

```
void producer() {
  wait(&elem);
  enqueue(&l, &e);
  signal(&elem);
}
```

```
void consumer() {
  struct element *x;
  wait(&elem);
  x = dequeue(&l);
  signal(&elem);
}
```

- “betriebsmittelorientierte Synchronisation”

```
/* gem. Speicher */
Semaphore resource;
```

```
/* Initialisierung */
resource = N; /* N > 1 */
```

sonst wie beim
gegenseitigen Ausschluss

Synchronisationsmuster (2)

- “Erzeuger-Verbraucher-Problem”

```
/* gem. Speicher */
Semaphore frei;
Semaphore belegt;
struct list l;
struct element e;
```

```
void producer() {
  wait(&frei);
  enqueue(&l, &e);
  signal(&belegt);
}
```

```
void consumer() {
  struct element *x;
  wait(&belegt);
  x = dequeue(&l);
  signal(&frei);
}
```

```
/* Initialisierung */
frei = 1;
belegt = 0;
```

- “einseitige Synchronisation”

```
/* gem. Speicher */
Semaphore elem;
struct list l;
struct element e;
```

```
void producer() {
  enqueue(&l, &e);
  signal(&elem);
}
```

```
void consumer() {
  struct element *x;
  wait(&elem);
  x = dequeue(&l);
}
```

```
/* Initialisierung */
elem = 0;
```

POSIX-Semaphore - Schnittstelle (1)

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- Initialisiert eine Semaphore
- Argumente:
  - *sem*: Zeiger auf die zu initialisierende Semaphore
  - *pshared*: Gibt an, ob die Semaphore zw. Threads oder Prozessen geteilt wird
  - *value*: Initialer Wert der Semaphore
- Rückgabewerte:
  - 0, wenn erfolgreich
  - ≠ 0, wenn Fehler

Analog dazu: **sem_destroy(3)**

POSIX-Semaphore - Schnittstelle (2)

```
int sem_wait(sem_t *sem);
```

Analog dazu: **sem_post(3)**

- Dekrementiert den Zähler
- Blockiert ggf., wenn Zähler bereits 0
- Argumente:
  - *sem*: Zeiger auf die zu verwendende Semaphore
- Rückgabewerte:
  - 0, wenn erfolgreich
  - ≠ 0, wenn Fehler

POSIX-Semaphore – Beispiel

```c
#include <pthread.h>
#include <semaphore.h>
int i=0;
sem_t mySem;

main() {
    sem_init(&mySem, 0, 1);

    /* erstelle zwei Threads... */

    sem_destroy(&mySem);
}
```

```c
f1() { /* Thread 1 */
    sem_wait(&mySem);
    i++;
    sem_post(&mySem);
}

f2() { /* Thread 2 */
    sem_wait(&mySem);
    i++;
    sem_post(&mySem);
}
```

Exkurs: Linux-Systemcalls (hier für x86)

- Einzige Möglichkeit für Userspace-Programme auf Kernelspace-Funktionen zuzugreifen
- Jedem Systemcall ist eine eindeutige Nummer zugeordnet

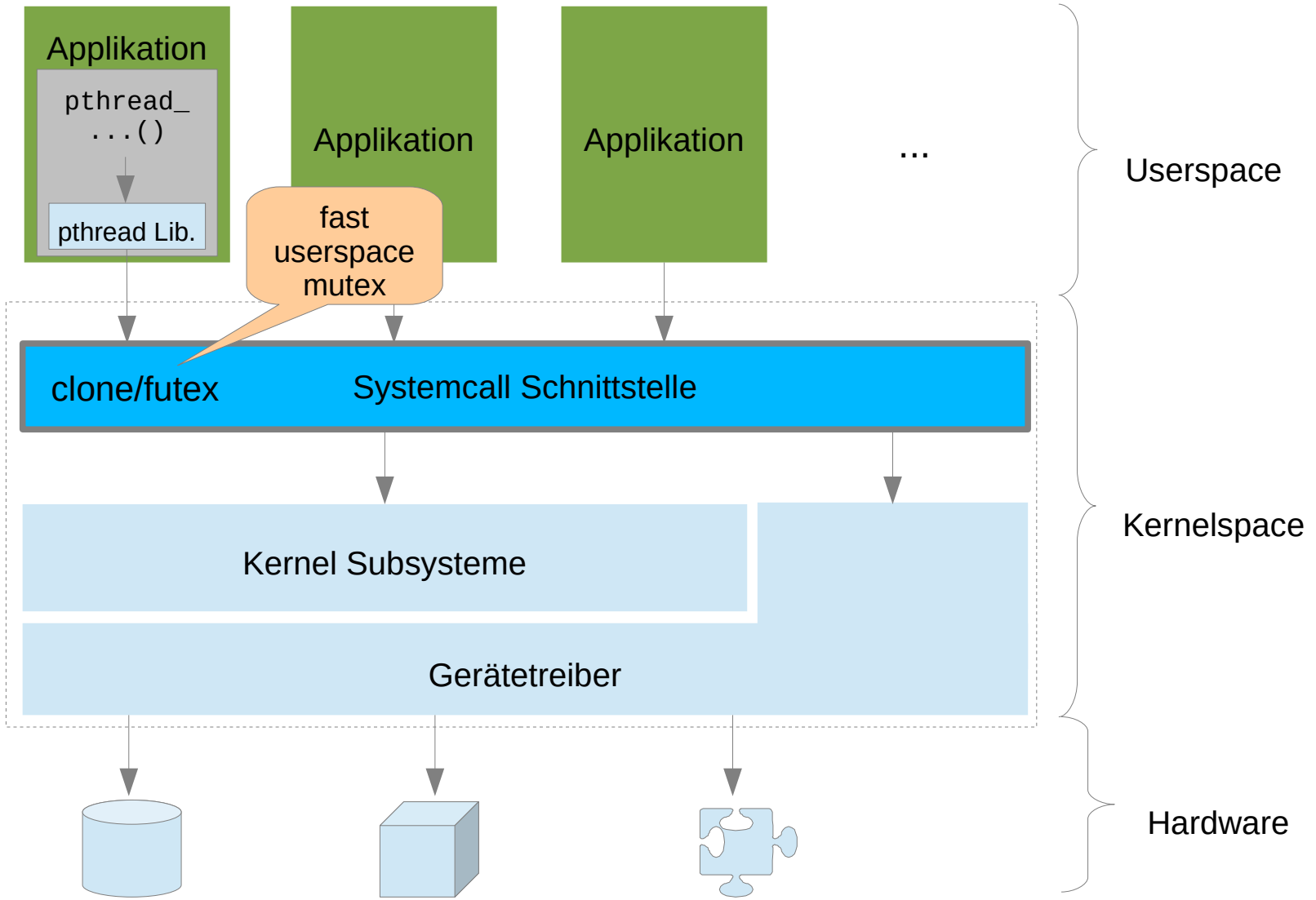
arch/x86/kernel/syscall_table_32.S

```
ENTRY(sys_call_table)
        .long sys_restart_syscall /* 0 */
        .long sys_exit            /* 1 */
        .long sys_fork            /* 2 */
        .long sys_read            /* 3 */
        .long sys_write           /* 4 */
        .long sys_open            /* 5 */
...
```

- Direkter Aufruf von Systemcalls z.B. per `syscall(2)`

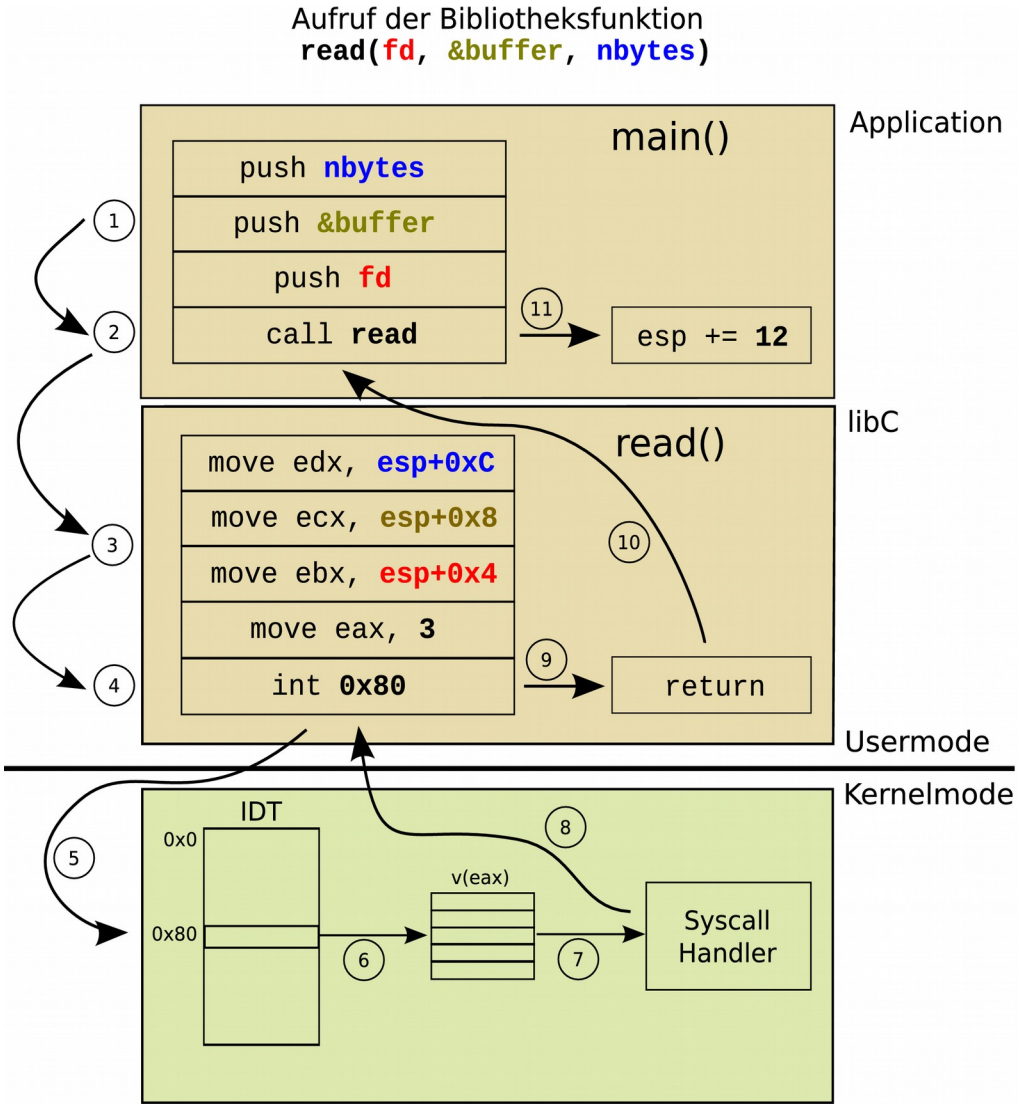
```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h> /* hier wird SYS_read=3 definiert */
#include <sys/types.h>

int main(int argc, char *argv[]) {
  ...
  syscall(SYS_read, fd, &buffer, nbytes); /* read(fd, &buffer, nbytes) */
  return 0;
}
```

Systemstruktur



Ablauf eines Systemcalls

1) Argumente → Stack
   (Konvention: Letztes zuerst)
2) Aufruf der Bibliotheksfunktion
   (Implizit: push *Rücksprungadresse*)
3) Argumente in Register laden
   (Stack für User und Kernel versch.)
4) Interrupt auslösen
5) Interruptnummer Index in Tabelle,
   hält Adressen der Zielfunktionen
6) Zielfunktion wählt mit **eax** Funktion
   aus (Array aus Funktionspointern)
7) Kernel: sys_read()
8) Mode-Wechsel (alter Userstack)
9) Ausführung fährt fort
10) Rücksprungadr. noch auf Stack
11) Stack aufräumen



Beispiel: _exit(255) „per Hand“

- Parameter von Systemcalls:
  - < 6 Parameter: Parameter werden in den Registern `ebx`, `ecx`, `edx`, `esi`, `edi` abgelegt
  - >= 6 Parameter: `ebx` enthält Pointer auf Userspace mit Parametern
- Aufruf des sys_exit Systemcalls per Assembler
  - `void _exit(int status)` (beende den aktuellen Prozess mit Statuscode `status`)
  - `sys_exit` Systemcall hat die Nr. 0x01

myexit.c

```
int main(void) {
        asm("mov $0x01, %eax\n" /* syscall # in eax */
            "mov $0xff, %ebx\n" /* Parameter 255 in ebx */
            "int $0x80\n");   /* Softwareinterrupt an Kernel */
        return 0;
}
```

```
pohl@host:~$ ./myexit
pohl@host:~$ echo $?
255
pohl@host:~$
```