Übungen Betriebssysteme (BS)

U2 – Thread Synchronisation

https://sys.cs.tu-dortmund.de/DE/Teaching/SS2021/BS/

Peter Ulbrich

peter.ulbrich@tu-dortmund.de

https://sys.cs.tu-dortmund.de/EN/People/ulbrich/

technische universität
dortmund

arbeitsgruppe
systemsoftware

Agenda

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

Besprechung Aufgabe 1

- Foliensatz *Besprechung A1*

Grundlagen C-Programmierung

- Foliensatz *C-Einführung* (Folie 36 bis 42)

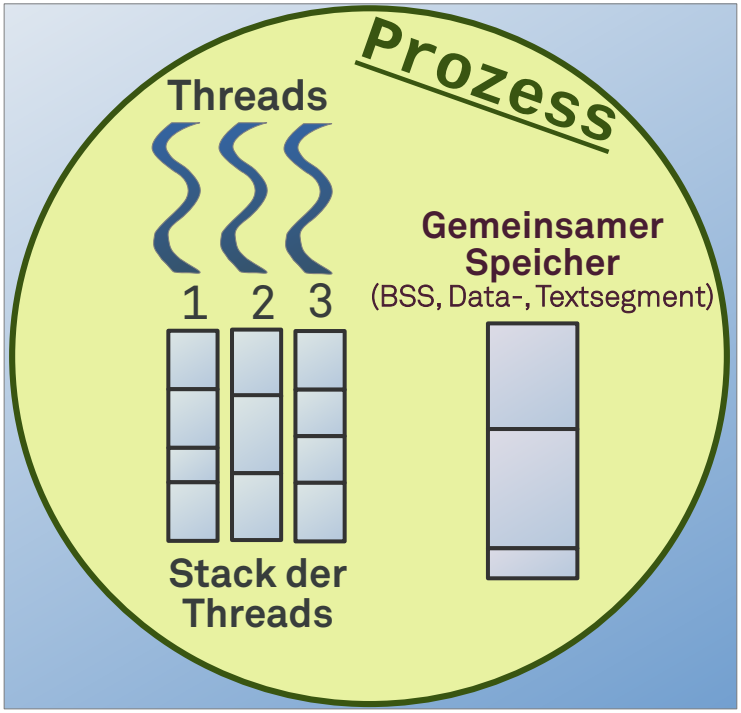
POSIX

- *„**P**ortable **O**perating **S**ystem **I**nterface“*
- von *IEEE* entwickelte Schnittstellen-Standardisierung unter UNIX
  - ermöglicht einfache *Applikationsportierung*
- **POSIX** definiert u.a. eine *standardisierte API* zwischen *Betriebssystem* und einer *Applikation*

UNIX-Prozess vs. POSIX-Threads

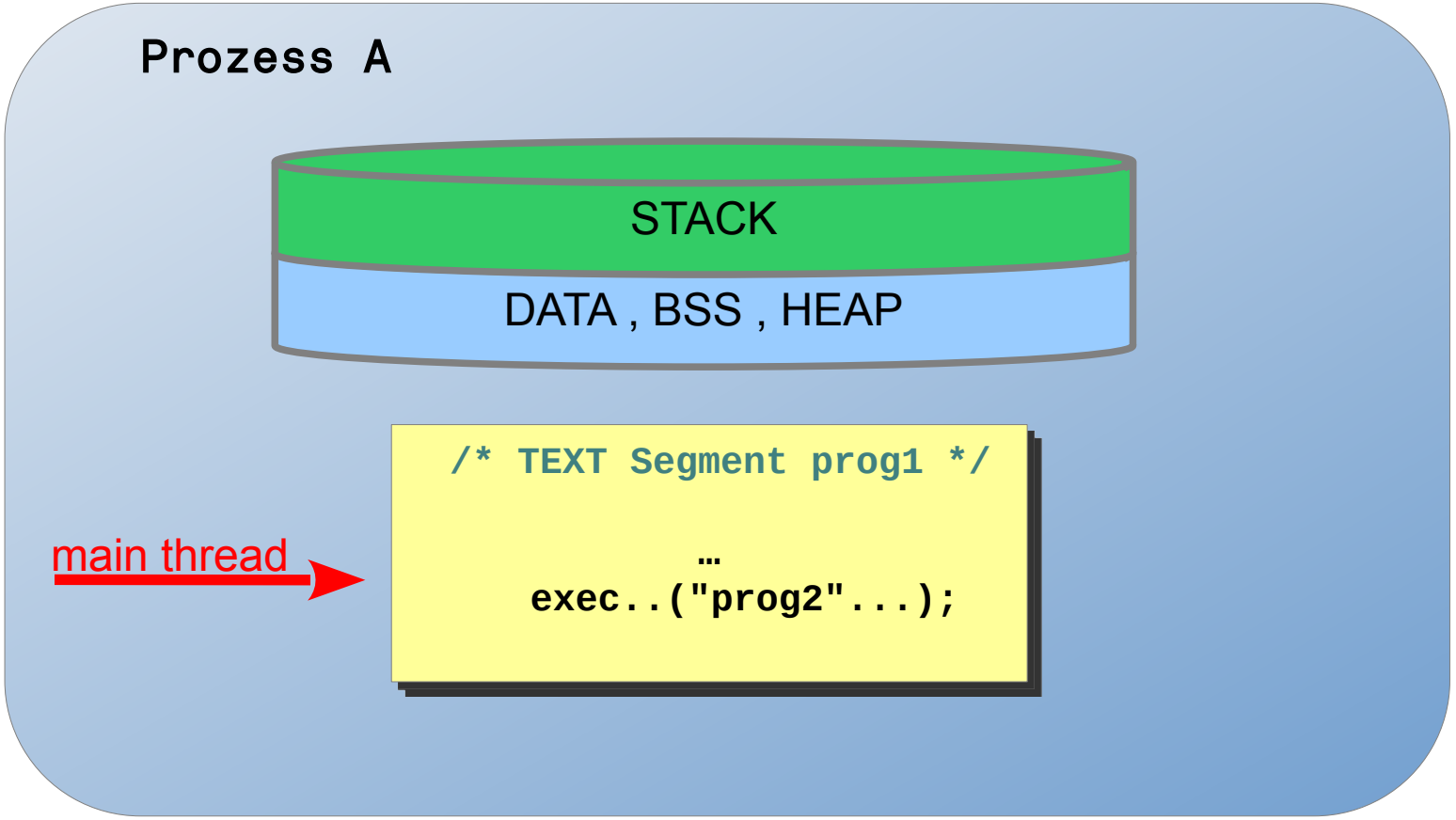
- **UNIX-Prozesse:** *schwergewichtig*
  (haben einen eigenen Adressraum)

- **POSIX-Threads:** *leichtgewichtig* (kurz `pthreads`)
  - 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
    Systemaufrufe
  - jeder `pthread` hat eine *eigene ID*
    (*Typ* `pthread_t`: *`unsigned long int`*)



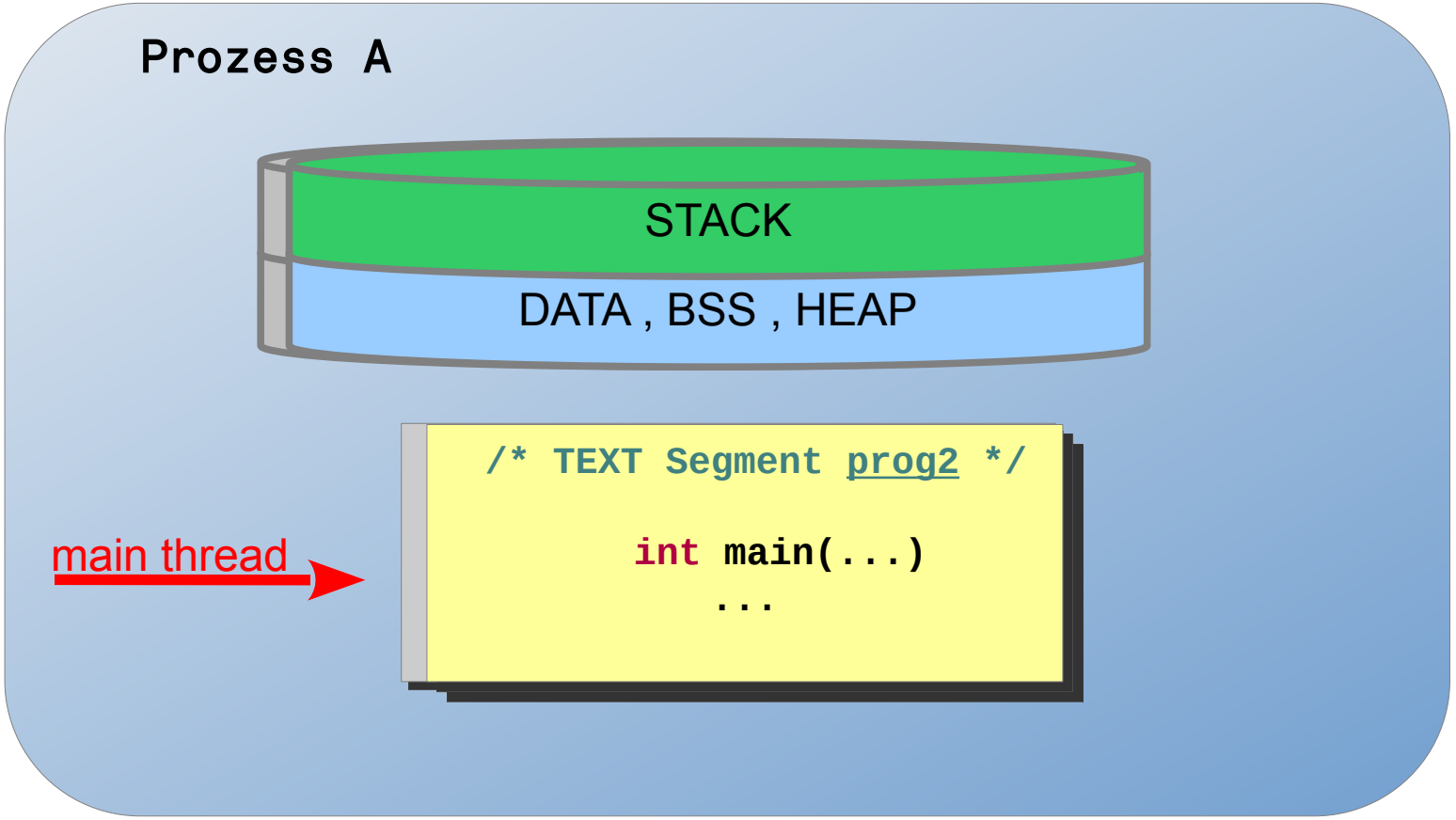
Vergleich: `exec..()` `fork()` `pthread_create()`

- Überlagerung eines Prozesses
- Keine **gemeinsamen** Daten
- *schwergewichtig*



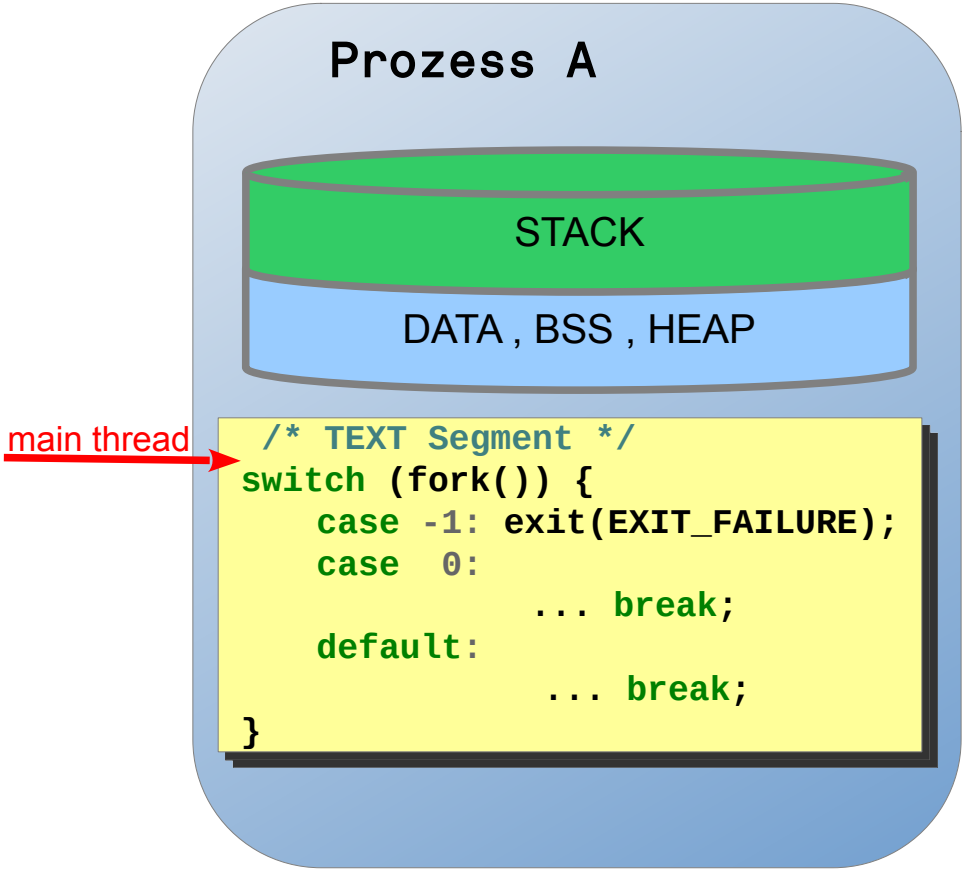
Vergleich: `exec..()` `fork()` `pthread_create()`

- Überlagerung eines Prozesses
- Keine **gemeinsamen Daten**
- *schwergewichtig*



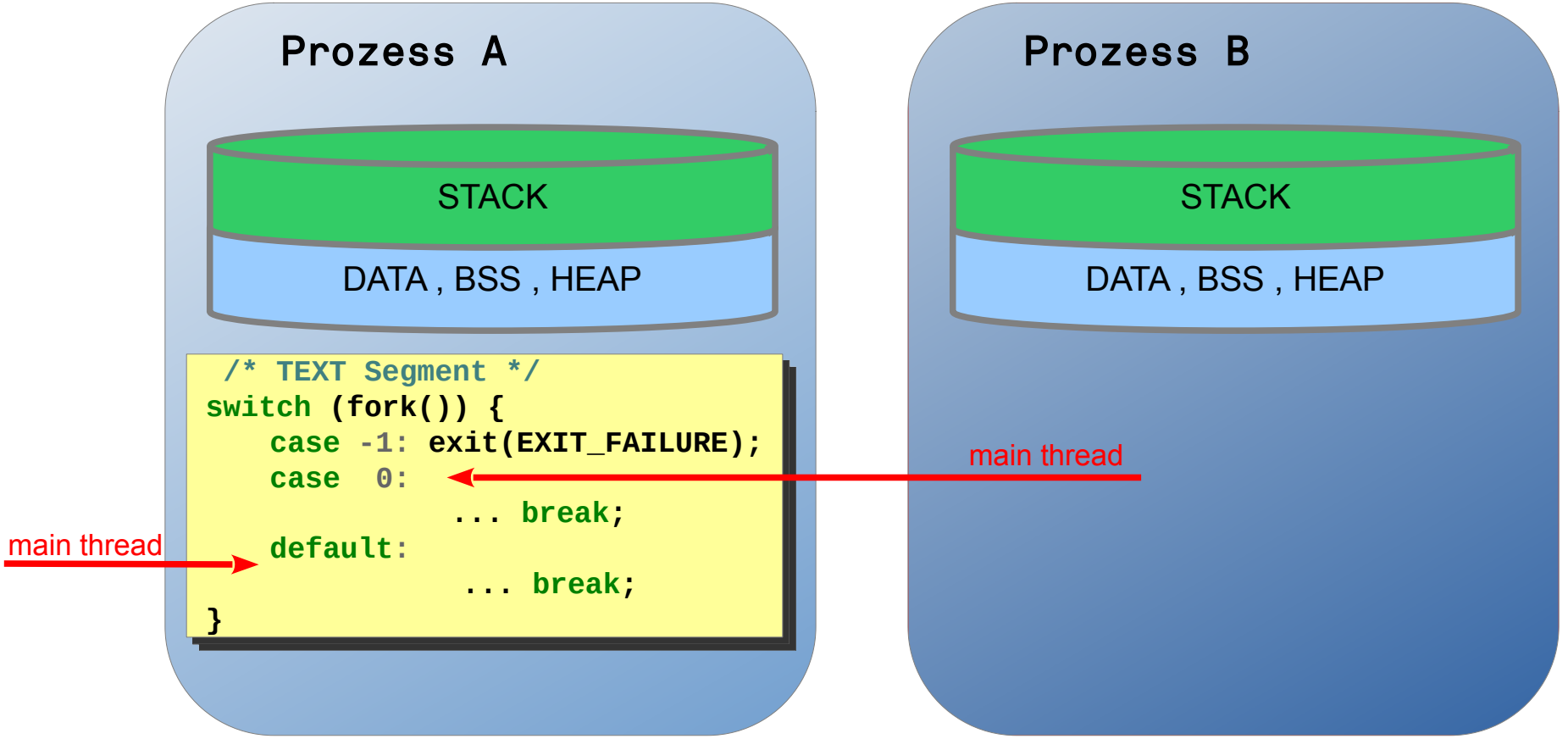
Vergleich: `exec..()` `fork()` `pthread_create()`

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



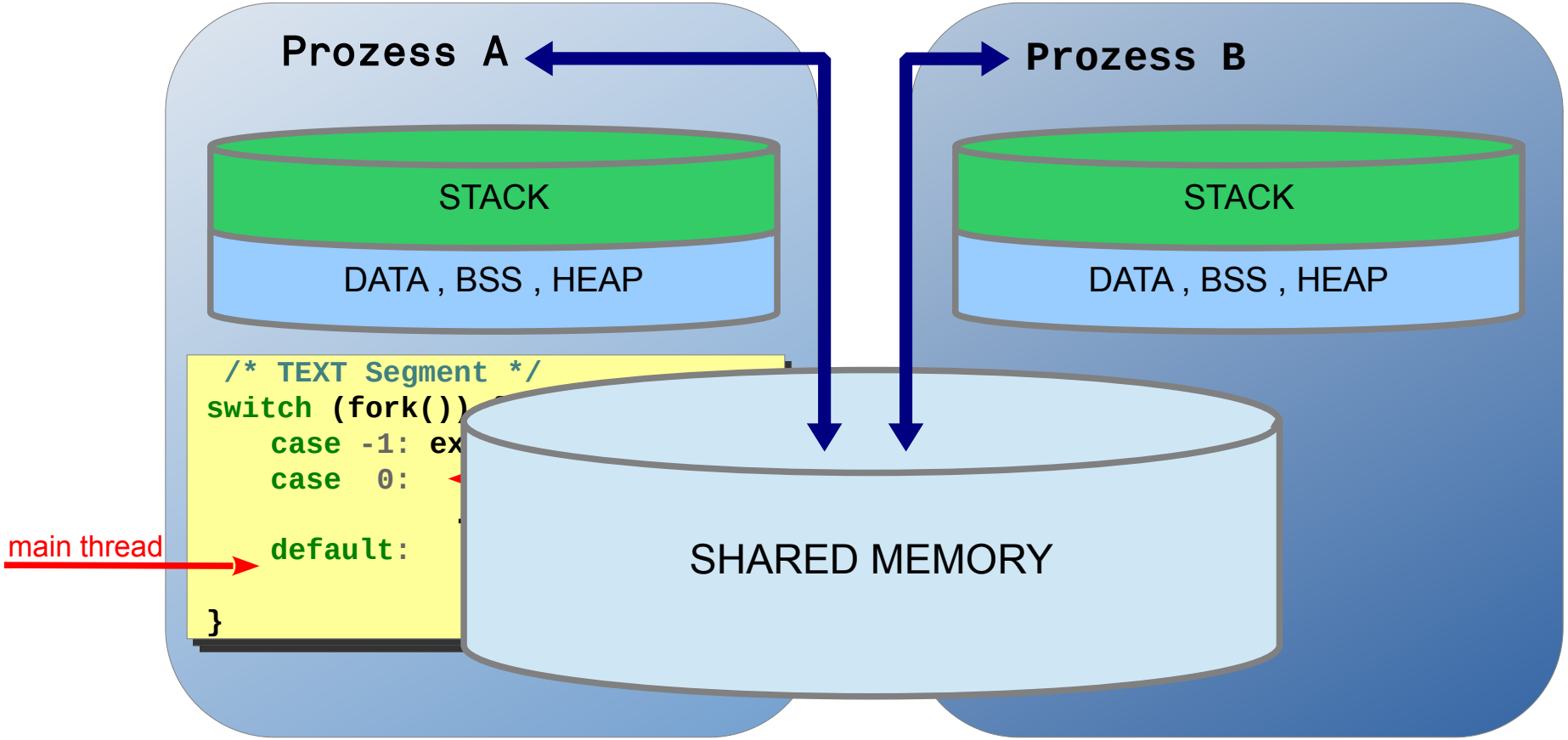
Vergleich: `exec..()` `fork()` `pthread_create()`

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



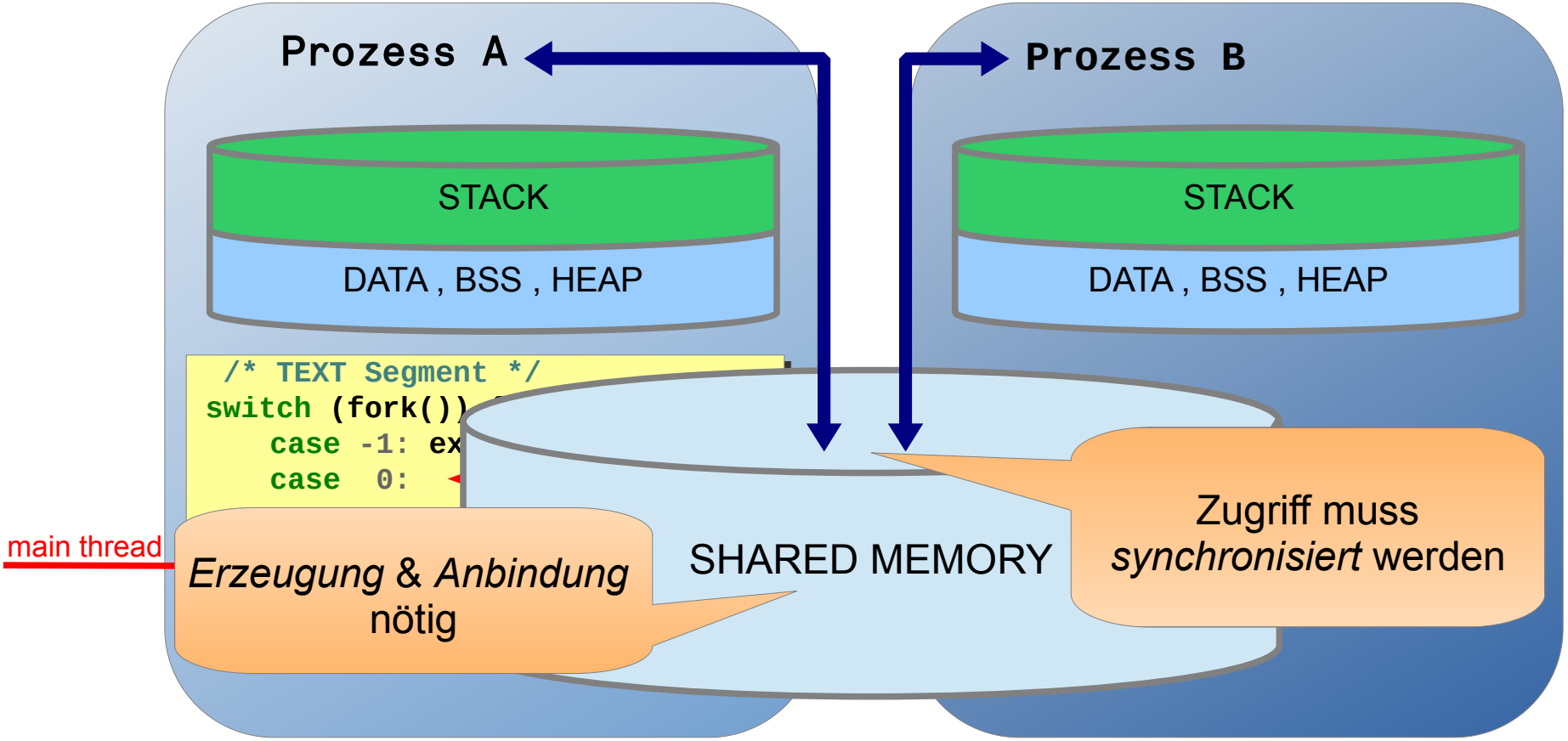
Vergleich: `exec..()` `fork()` `pthread_create()`

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



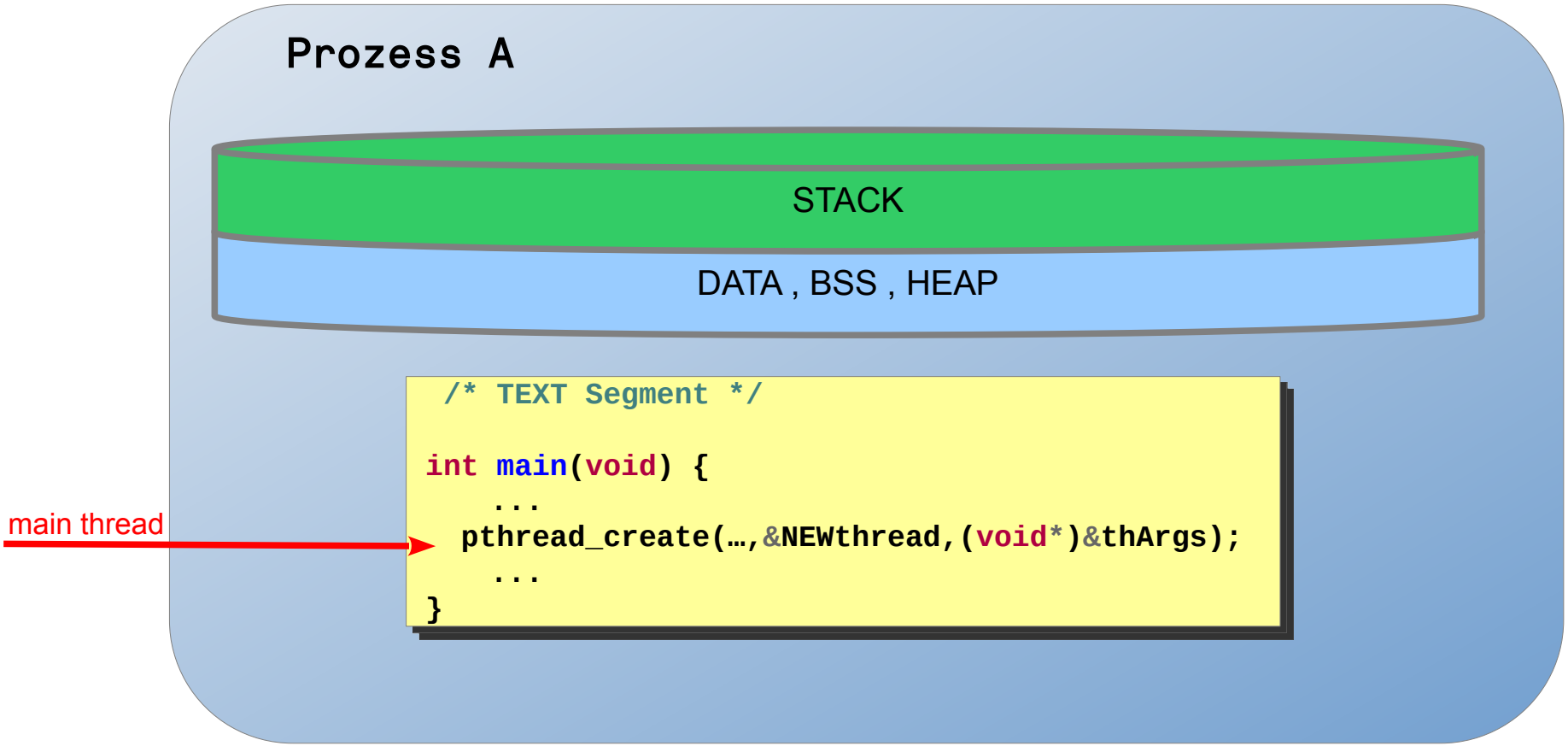
Vergleich: `exec..()` `fork()` `pthread_create()`

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



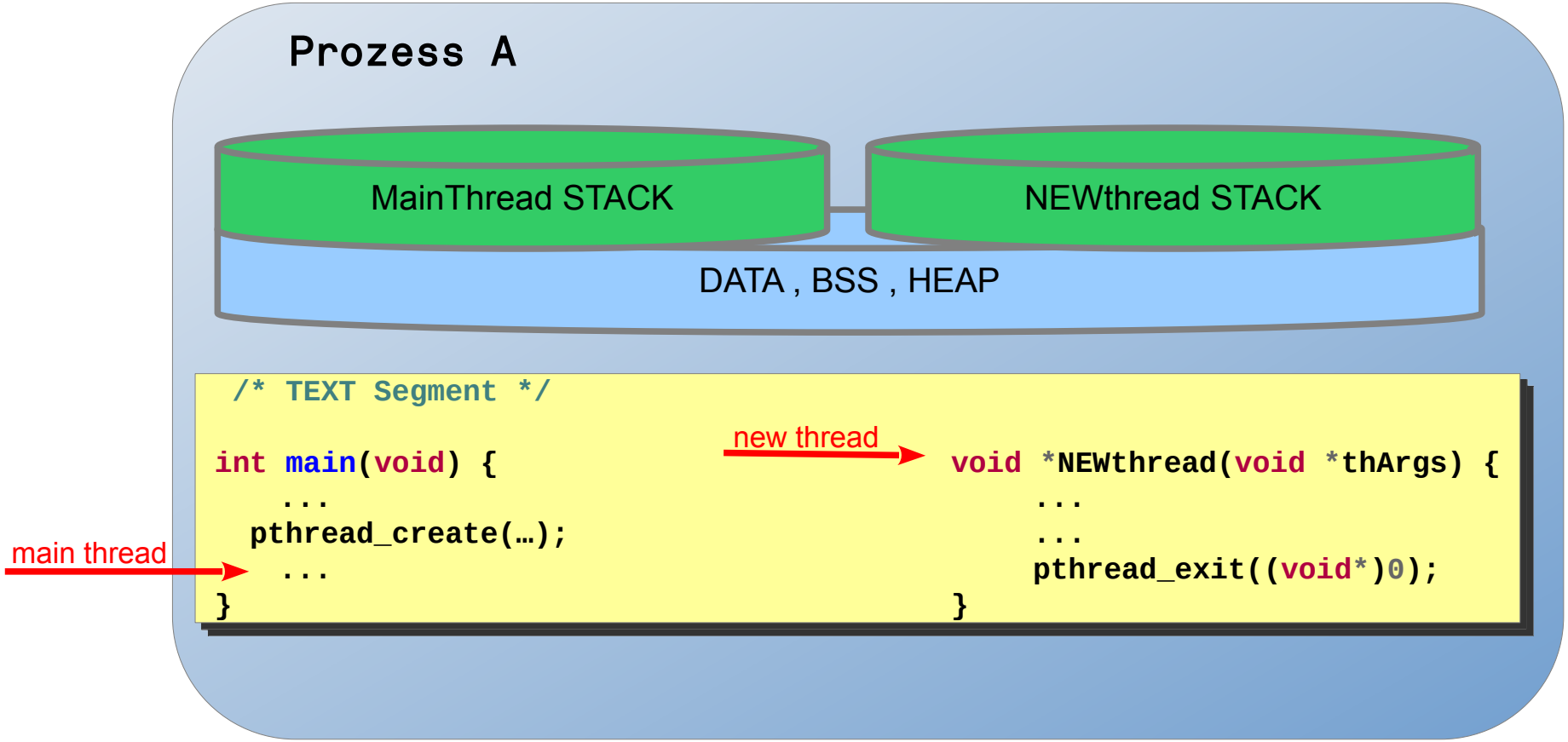
Vergleich: `exec..()` `fork()` `pthread_create()`

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



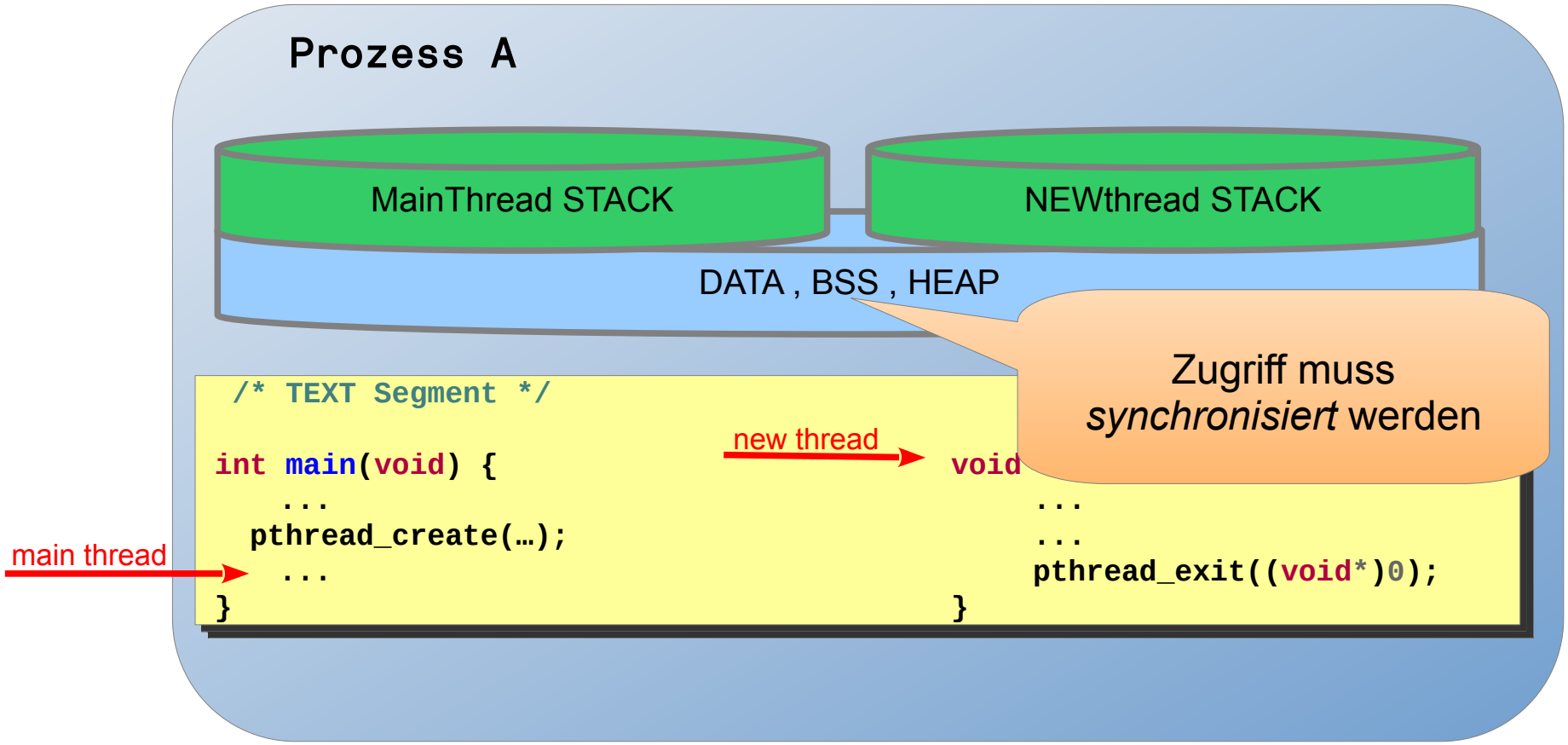
Vergleich: `exec..()` `fork()` `pthread_create()`

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



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

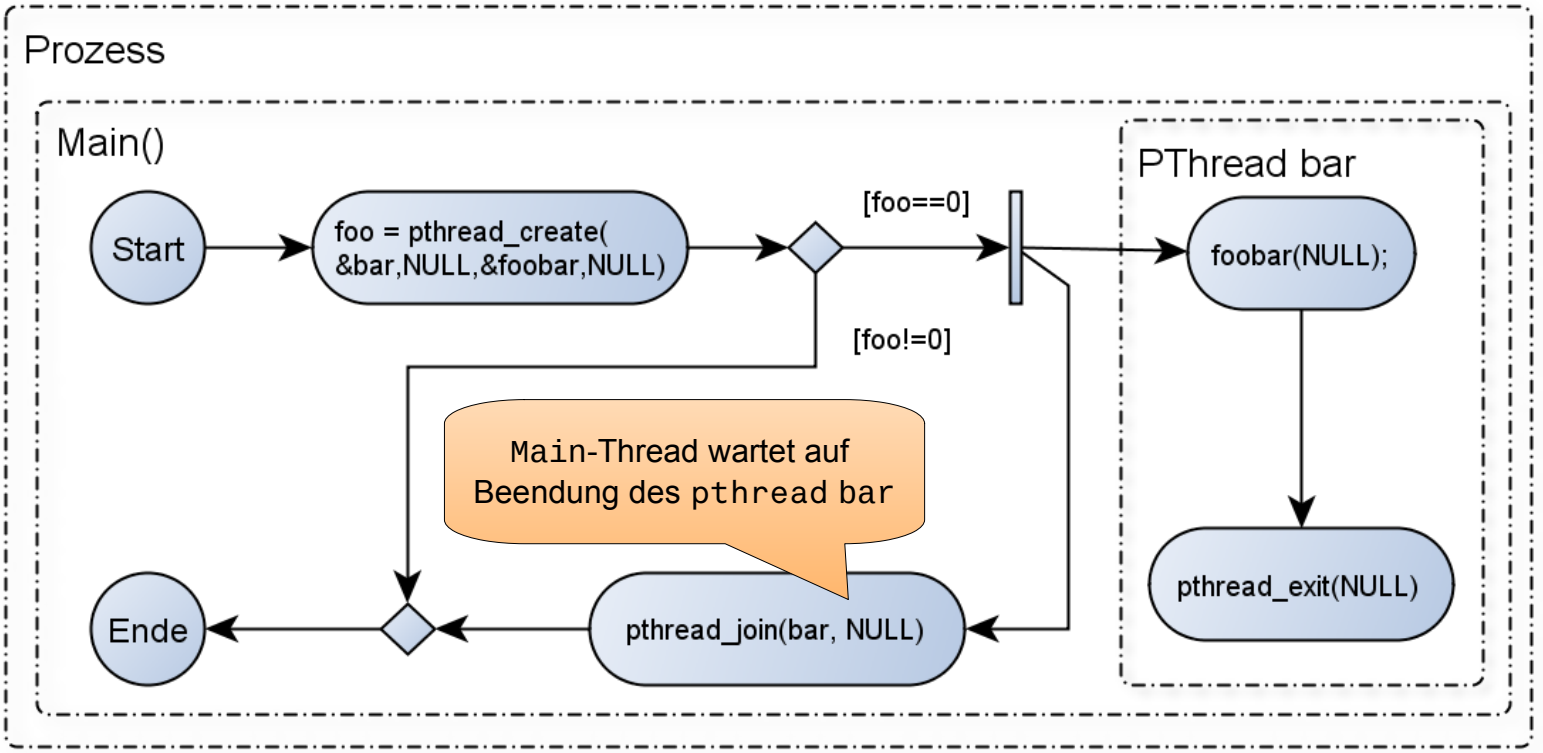
- Aufteilung eines Prozesses
- **Gemeinsame Daten:** Data, BSS, Heap, shared-memory
- *leichtgewichtig*



Funktionen für `pthreads` (Übersicht)

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

benötigen:
`#include <pthread.h>`



Funktionen für `pthreads` (1)

```
int pthread_create(thread[1], NULL[1], start_routine[1], arg[1]);
```

vorgeschriebene Signatur:
*void** threadFunc(*void** arg)

- erstellt einen *neuen* Thread
- Argumente:
  - **thread:** Hier wird die eindeutige ID des erzeugten Threads abgelegt
  - **attr:** Optionale Attribute (in unserem Fall: *NULL*)
  - **start_routine:** Zeiger auf auszuführende Funktion
  - **arg:** Zeiger auf Argument, welches ***start_routine*** übergeben wird
- Rückgabewerte:
  - **0**, wenn erfolgreich
  - **≠ 0**, wenn Fehler

[1]: Die Parameter-Typen können in der ManPage zu `pthread_create(3)` nachgeschlagen werden

Funktionen für `pthreads` (2)

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

- *beendet* den Thread
- Argumente:
  - **`retval`:** gibt den *Exit-Status* an, der zurückgegeben werden soll, wenn der Thread terminiert (*optional* → *NULL*)
  - zur Erinnerung: ***void**** stellvertretend für *alle möglichen* Zeiger. *NULL* ist gleichbedeutend mit *(void*)0*

Funktionen für `pthreads` (3)

```
int pthread_join(pthread_t thread, void **retval);
```

- sorgt dafür, dass der Aufrufer *warten* muss, bis der `thread` mit der *passenden ID* terminiert
- Argumente:
  - **`Thread:`** Thread-ID , auf dessen Terminierung gewartet wird
  - **`retval:`** nimmt *Exit-Status* des beendeten Threads entgegen *(interessiert uns nicht, daher: NULL)*
- Rückgabewerte:
  - **0**, wenn *kein* Fehler
  - **≠ 0**, wenn Fehler

Funktionen für `pthreads` (4)

```
pthread_t pthread_self(void);
```

- Liefert die ID des aufrufenden Threads zurück
  - Der Rückgabewert ist *äquivalent* zu dem Zeiger `thread` aus der Funktion `pthread_create(thread …)` als dieser Thread erzeugt wurde
  - Rückgabe ist garantiert
- Vergleichsfunktion:

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

- Rückgabewerte:
  - ≠ 0, bei Gleichheit
  - = 0, bei Ungleichheit

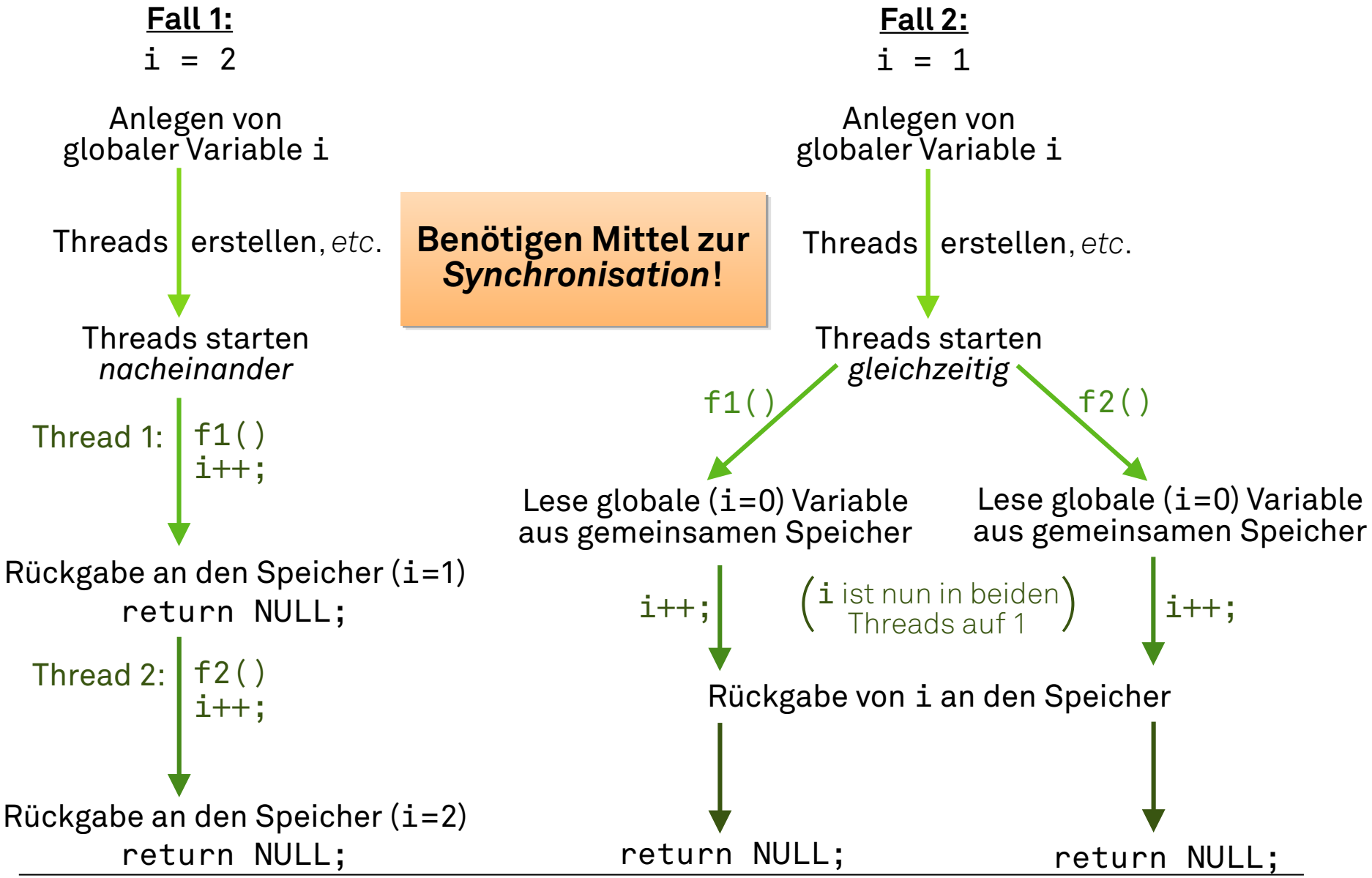
pthread-Beispiel

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

void* Hello(void *arg) {
    printf("Hello! It's me, thread!”);
    pthread_exit(NULL); // oder: return 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); // Da wegen pthread_join() nur noch
}                       // ein Thread läuft, genügt return 0;
```

Was wird passieren?

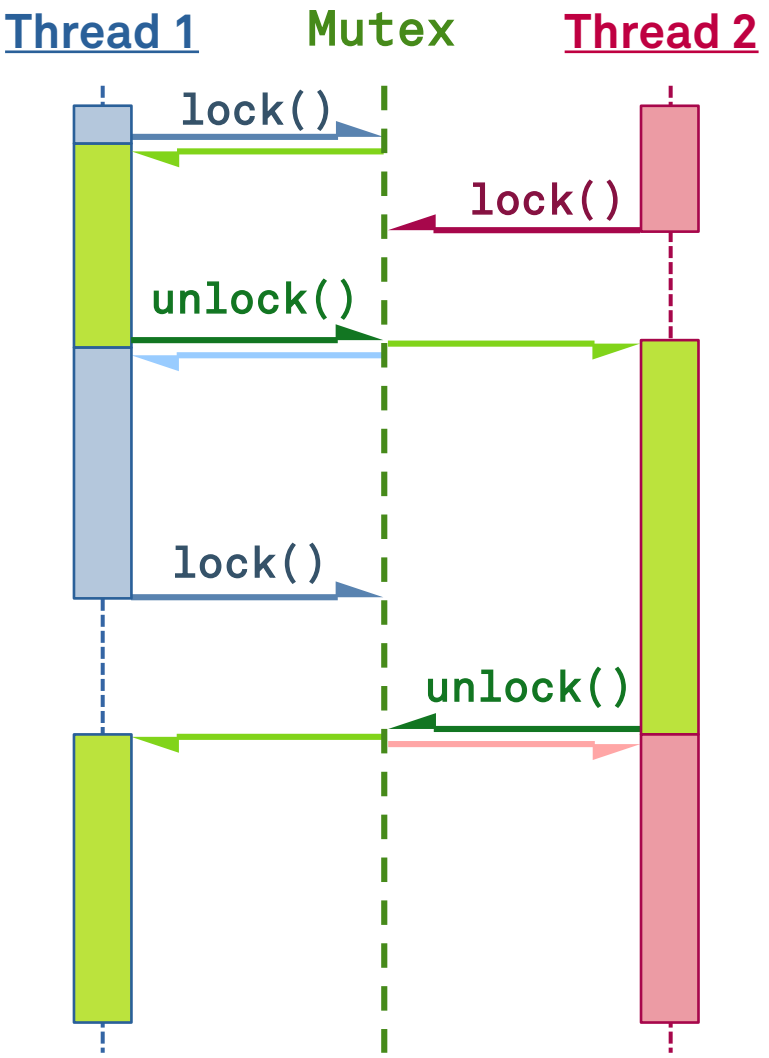
```c
#include <stdio.h>

/* globale Variable */
int i = 0;

/* Funktion von Thread 1 */
void *f1(void *arg) {
    i++;
    return NULL;
}

/* Funktion von Thread 2 */
void *f2(void *arg) {
    i++;
    return NULL;
}
```

Was kann passieren?



Mutex

- **Mutual exclusion** (*gegenseitiger Ausschluss*)
  - Objekt zum *Erzwingen* von gegenseitigem Ausschluss mit *atomaren Operationen*



- Für unsere Übung:
  - `pthread_mutex_init();`
  - `pthread_mutex_destroy();`
  - `pthread_mutex_lock();`
  - `pthread_mutex_unlock();`

`Mutex` =Thread ist im Besitz des Mutex, darf ungestört auf gemeinsamem Speicher operieren

Mutex mit `pthreads` (1)

```
int pthread_mutex_init(mutex[2], attr[2]);
```

- initialisiert den `Mutex`,
  - **`attr:`** gibt spezielle Anforderungen für den Mutex an *(meistens NULL)*
  - `Mutex` wird *nicht sperrend* initialisiert!

```
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

- entfernt den `Mutex`, wenn er nicht weiter verwendet werden soll
- Rückgabewerte:
  - **0**, wenn erfolgreich
  - **≠ 0**, `Mutex` konnte nicht initialisiert/zerstört werden

[2]: Die Parameter-Typen können in der ManPage zu `pthread_mutex_init(3)` nachgeschlagen werden

Mutex mit `pthreads` (2)

```
int pthread_mutex_lock(pthread_mutex_t* mutex);
```

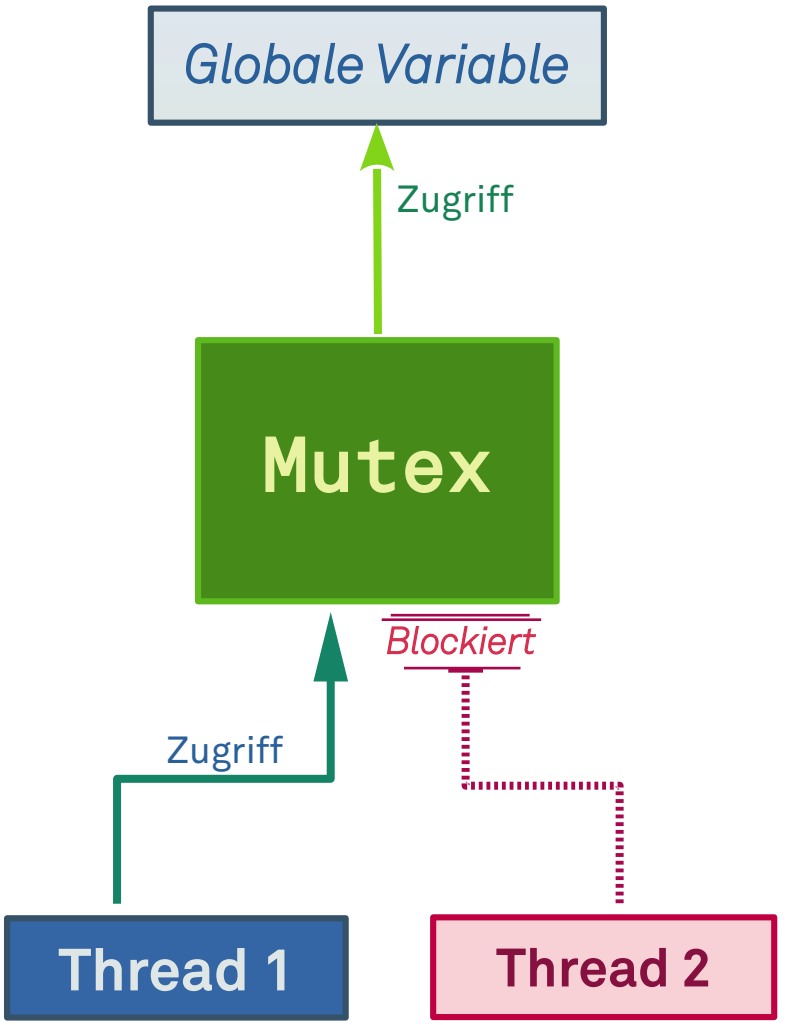
- betritt und sperrt den kritischen Bereich, den der `Mutex` beschreibt

```
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

- verlässt den kritischen Bereich und gibt ihn wieder frei
  - **`Mutex`** kann nur von dem Thread freigegeben werden, der ihn vorher gesperrt hat.
- <u>Rückgabewerte</u>:
  - **0**, wenn erfolgreich
  - **≠ 0**, `Mutex` konnte nicht gelockt/unlocked werden

Mutex-Beispiel

```c
#include <pthread.h>
int i = 0;
pthread_mutex_t lock;

main() {
    pthread_mutex_init(&lock, NULL);
    /* erstelle zwei Threads ... */
    /* warte auf Beenden ... */
    pthread_mutex_destroy(&lock);

}

f1() { /* Thread 1 */
    pthread_mutex_lock(&lock);
    i++;
    pthread_mutex_unlock(&lock);
}

f2() { /* Thread 2 */
    pthread_mutex_lock(&lock);
    i++;
    pthread_mutex_unlock(&lock);
}
```



Exkurs: Linux-Systemcalls (hier für x86)

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

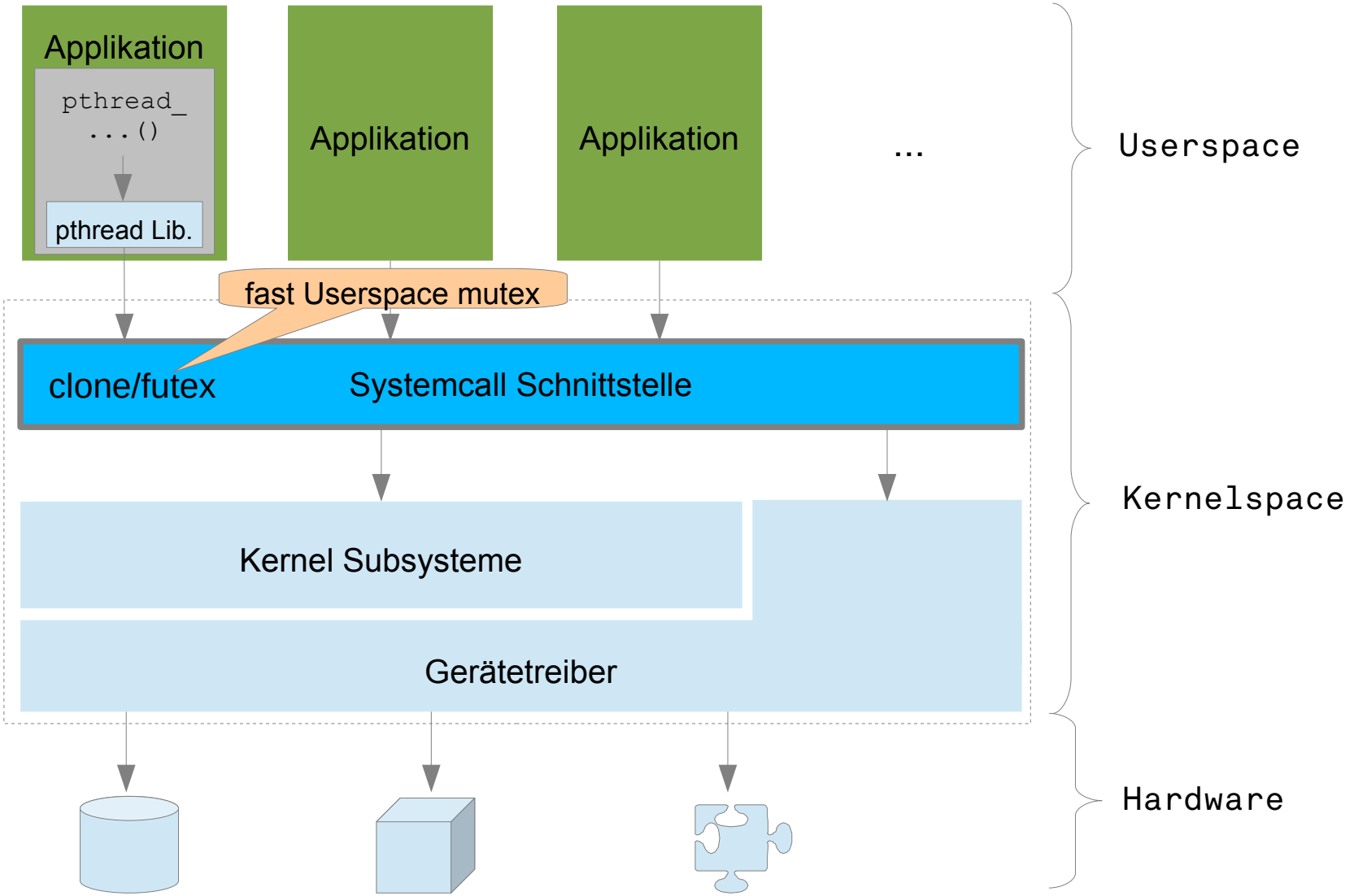
arch/x86/kernel/syscall_table_32.S

```
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)`

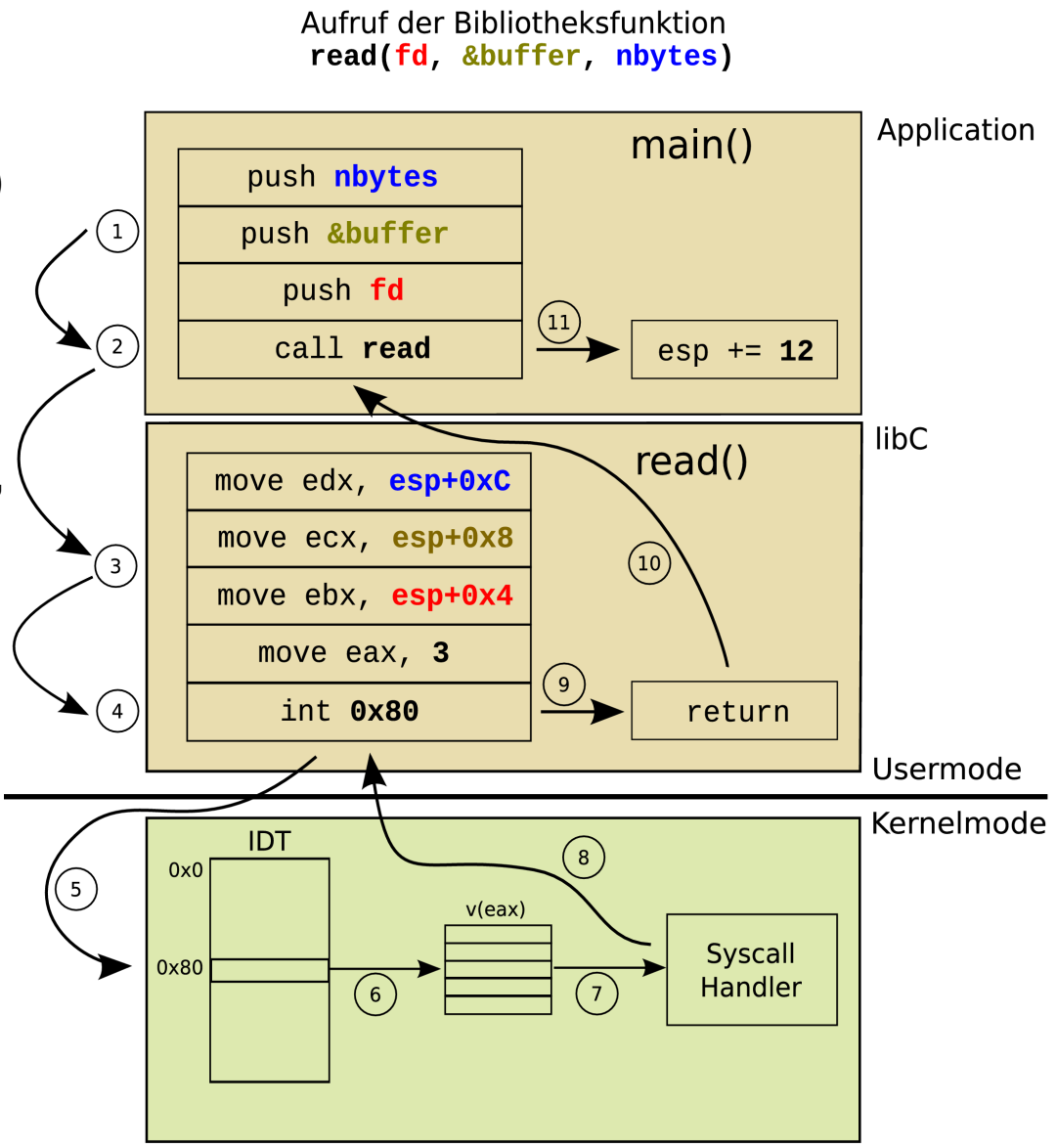
```c
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h> /* hier wird SYS_read=3 definiert */
#include <sys/types.h>

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 $?
255
pohl@host:~$
```