Tafelübung zu BS

2. Threadsynchronisation

Olaf Spinczyk
Arbeitsgruppe Eingebettete Systemsoftware

Lehrstuhl für Informatik 12
TU Dortmund

olaf.spinczyk@tu-dortmund.de
http://ess.cs.uni-dortmund.de/~os/

Agenda

- Besprechung Aufgabe 1: Prozesse verwalten
- Fortsetzung Grundlagen C-Programmierung
- Aufgabe 2: Threadsynchronisation
  - POSIX
  - UNIX-Prozesse vs. POSIX-Threads
  - Funktionen von Pthreads
  - Mutex
  - Systemcalls

Besprechung Aufgabe 1

- → Foliensatz Besprechung

Grundlagen C-Programmierung

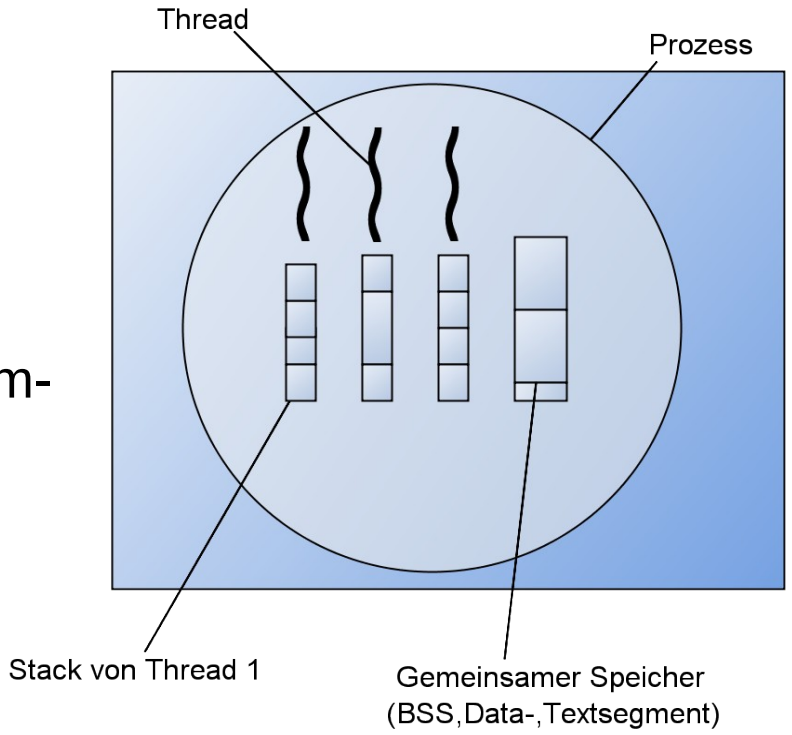
- → Foliensatz C-Einführung (Folie 47 bis Ende)

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

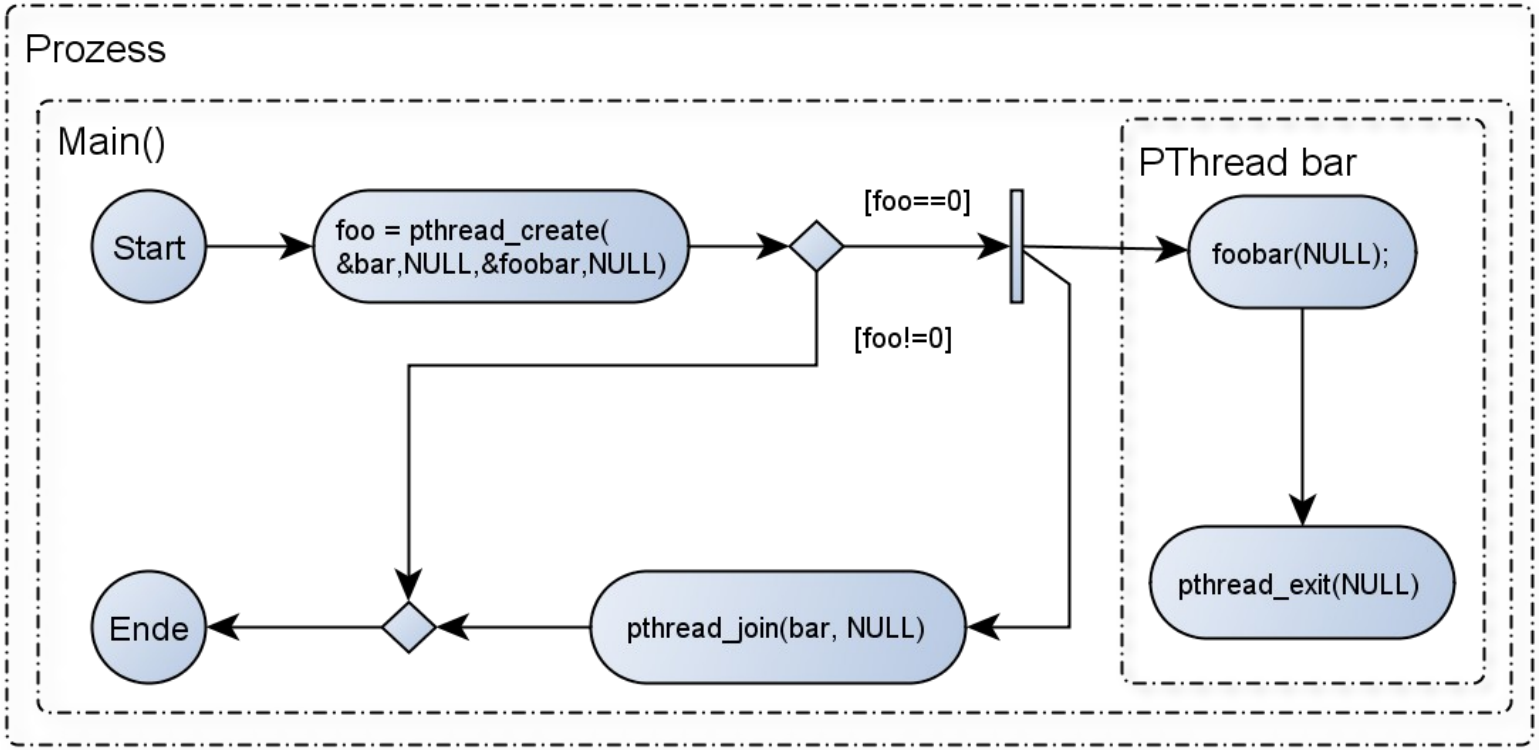
- UNIX-Prozesse: *schwergewichtig* (haben einen eigenen Adressraum)
- POSIX-Threads (kurz Pthreads): *leichtgewichtig*
  - ein Prozess kann mehrere Threads haben (teilen sich den gleichen Adressraum)
  - im Linux Kernel sind sogenannte *linux_threads* deklariert (je nach Kernel unterschiedlich)
  - Pthreads bieten standardisiert 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( );

benötigen:
#include <pthread.h>



Funktionen für Pthreads (1)

```
int pthread_create(pthread_t *thread, NULL, *start_routine, *arg);
```

- erstellt einen neuen Thread
- Argumente:
  - *thread*: ID die dem Thread zugewiesen wird (muss eindeutig sein)
  - *attr*: Attributobjekt (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

Funktionen für Pthreads (2)

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

- beendet den Thread
- *retval*: gibt den Exit-Status an, der zurückgegeben werden soll, wenn der Thread terminiert (optional → NULL)

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 (*thread*) terminiert
- *retval* nimmt den Exit-Status des beendeten Threads entgegen (in unserem Fall: NULL)
- Rückgabewerte:
  - 0, wenn kein Fehler
  - ≠ 0, wenn Fehler

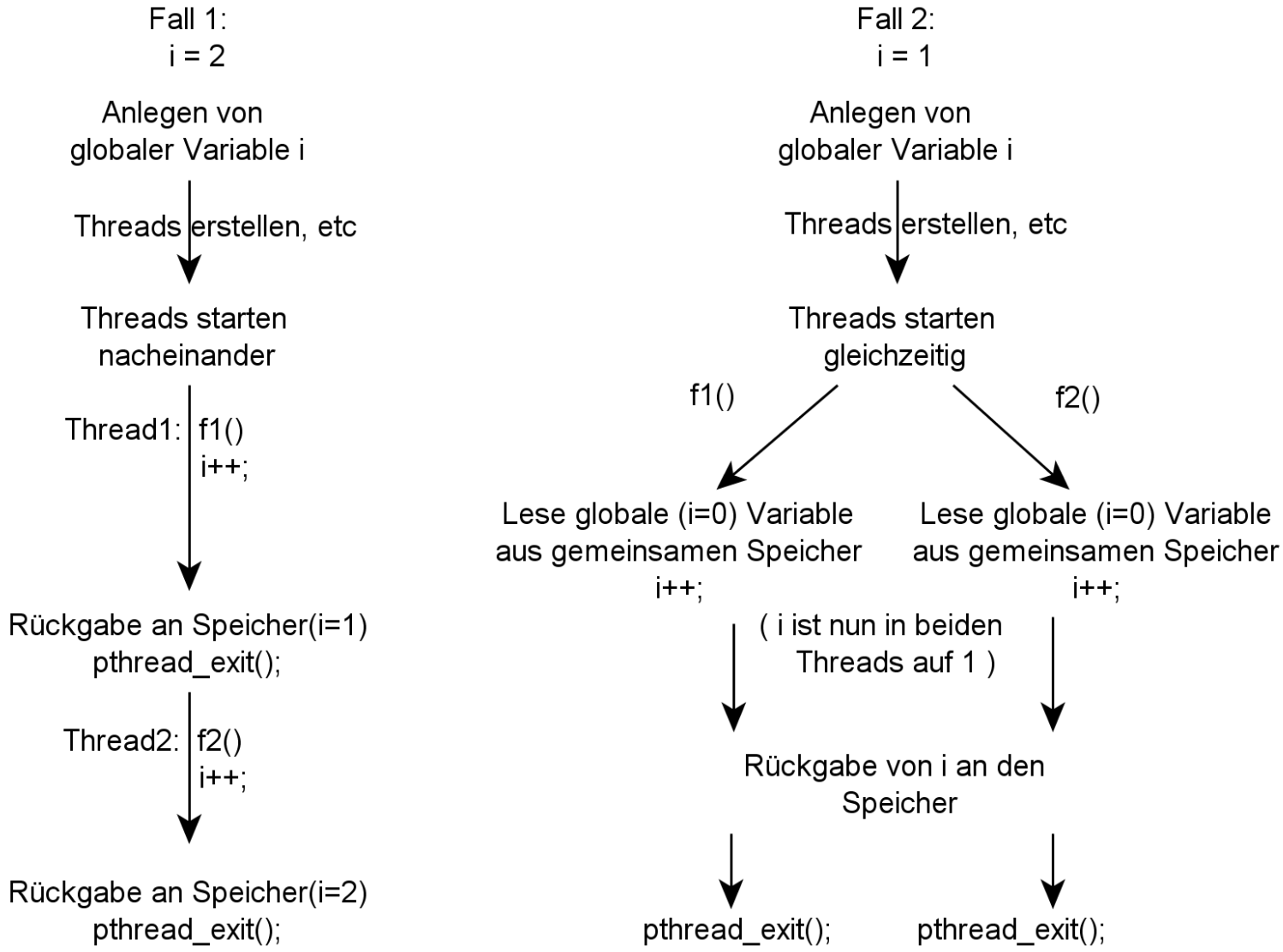
Pthread Beispiel

```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);
}
```

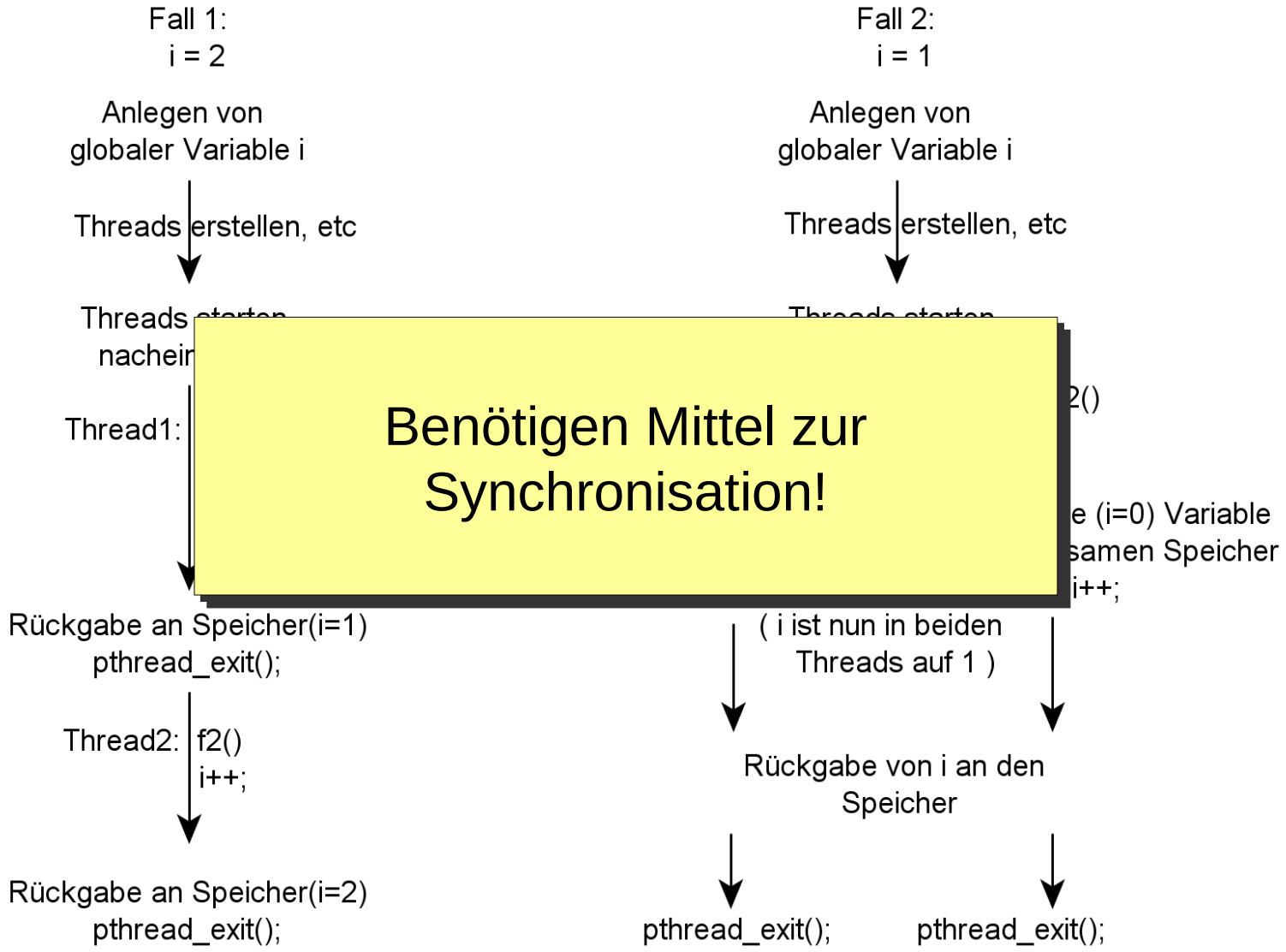
Was wird passieren?

```
/* globale Variable */
int i = 0;

/* Funktion von Thread 1 */
f1(){
   i++;
   pthread_exit(NULL);
}

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

Was kann passieren?

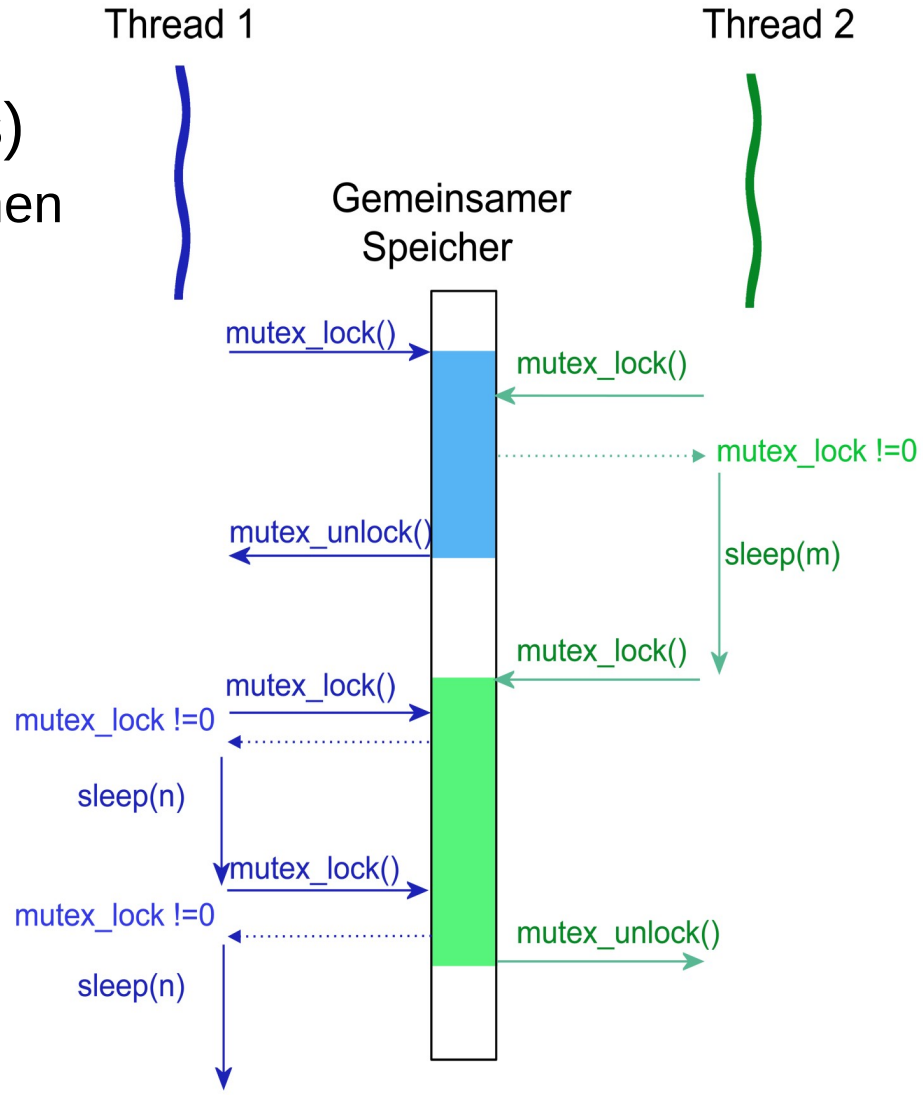


Was kann passieren?



Mutex

- Mutual exclusion (gegenseitiger Ausschluss)
  - Atomares Objekt, welches einen gegenseitigen Ausschluss erzwingt
- Für unsere Übung:
  - pthread_mutex_init();
  - pthread_mutex_destroy( );
  - pthread_mutex_lock( );
  - pthread_mutex_unlock( );



Mutex mit Pthreads (1)

```
int pthread_mutex_init(pthread_mutex_t *mutex, *attr);
```

- 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

Mutex mit Pthreads (2)

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

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

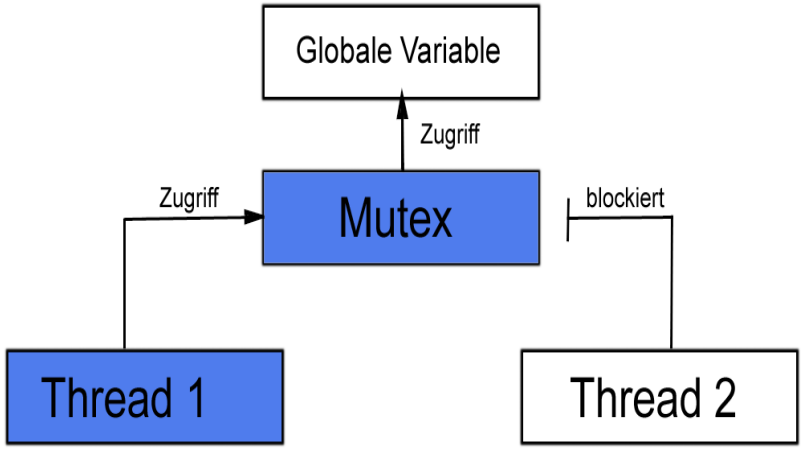
```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- verlässt den kritischen Bereich und gibt ihn wieder frei

- Rückgabewerte:
  - wenn 0 erfolgreich,
  - ≠ 0 Mutex konnte nicht gelockt/unlockt werden

Mutex Beispiel

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

main() {
    pthread_mutex_init(&lock, NULL);

    /* erstelle zwei Threads... */
}

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

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



Condition Variables (1)

- Bedingungsvariablen
  - an Bedingung gebundene Synchronisation
  - ohne Bedingungsvariable müsste ständig die Bedingung innerhalb des kritischen Abschnitts abgefragt werden → kostet unnötig Zeit
  - wird immer zusammen mit Mutex verwendet

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
```

- initialisiert eine Bedingungsvariable mit *cond* und optionalen Attributen

- Rückgabewerte:
  - wenn 0 erfolgreich,
  - ≠ 0 Bedingungsvariable konnte nicht initialisiert werden

Condition Variables (2)

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

- blockiert den Zugang zum Mutex solange bis die Bedingung gilt
  - sollte nur aufgerufen werden, wenn der kritische Abschnitt betreten wurde
  - sobald die Bedingung erfüllt ist, bekommt der Thread automatisch den Mutex und sperrt diesen

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- weckt einen anderen Thread auf, der auf die Bedingung *cond* wartet.
  - welcher Thread aufgeweckt wird, bestimmt eine interne Warteschlange

Systemaufrufe durch Programme

- Systemaufruf = Aufruf von Funktionen, die das Betriebssystem zur Verfügung stellt
  - Zugriffe auf angeschlossene Hardware
  - Funktionen zur Speicherverwaltung
  - Funktionen zur Prozessverwaltung
- Syscalls: Funktionen, die nur in einem privilegierten Modus ausgeführt werden können, d.h. mit erweiterten Rechten
- Linux: Aufteilung in User- und Kernelspace
- Problem: Ein einfacher Funktionsaufruf in die Kernelfunktionen ist nicht möglich
  - Ein fehlerhaftes Anwendungsprogramm könnte das System zum Absturz bringen
  - Jedes Anwendungsprogramm hätte volle Zugriffsrechte → z.B. Scheduling und Rechteverwaltung

Linux-Systemcalls (hier für x86)

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

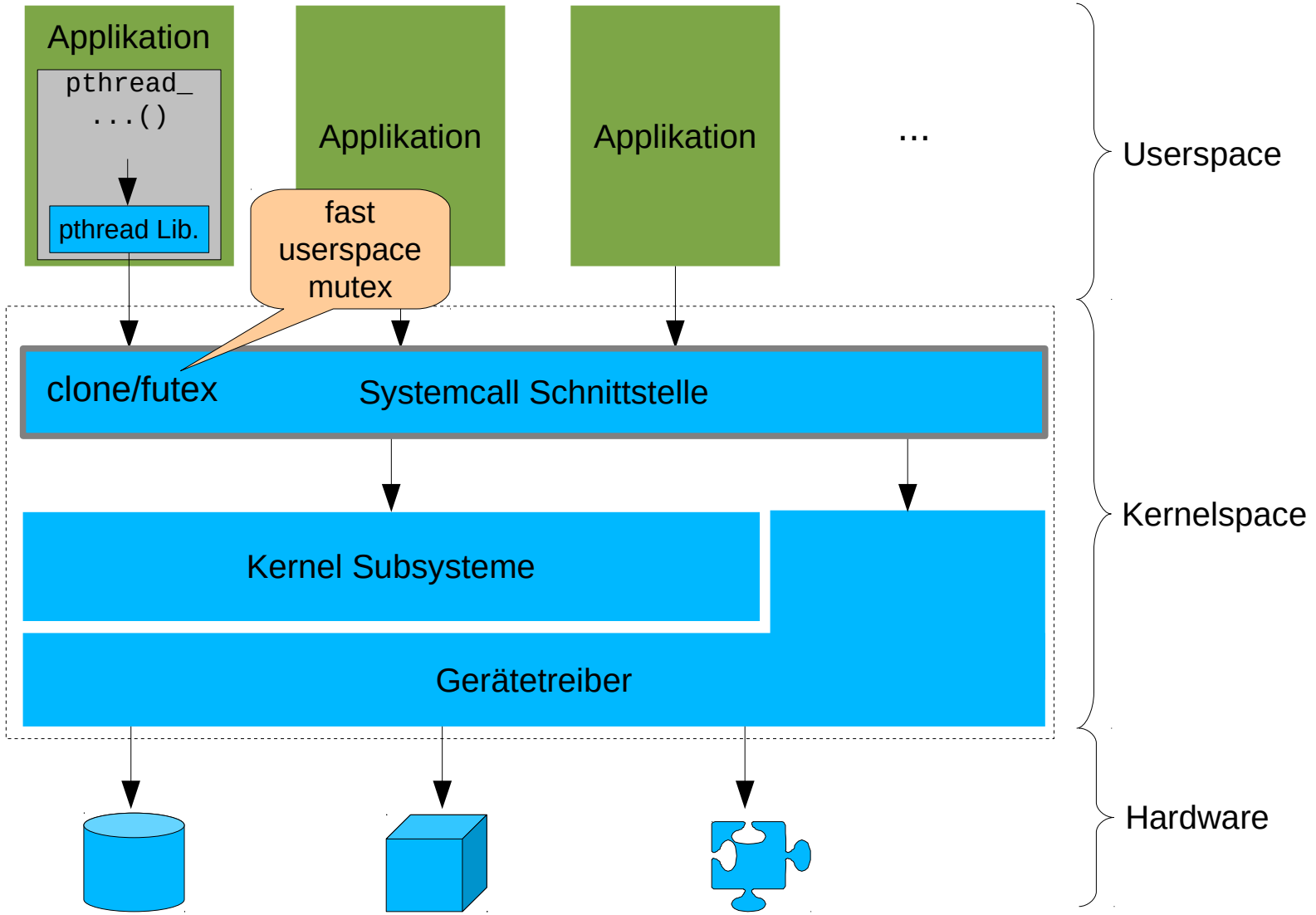
arch/x86/kernel/syscall_table_32.S

```
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)`

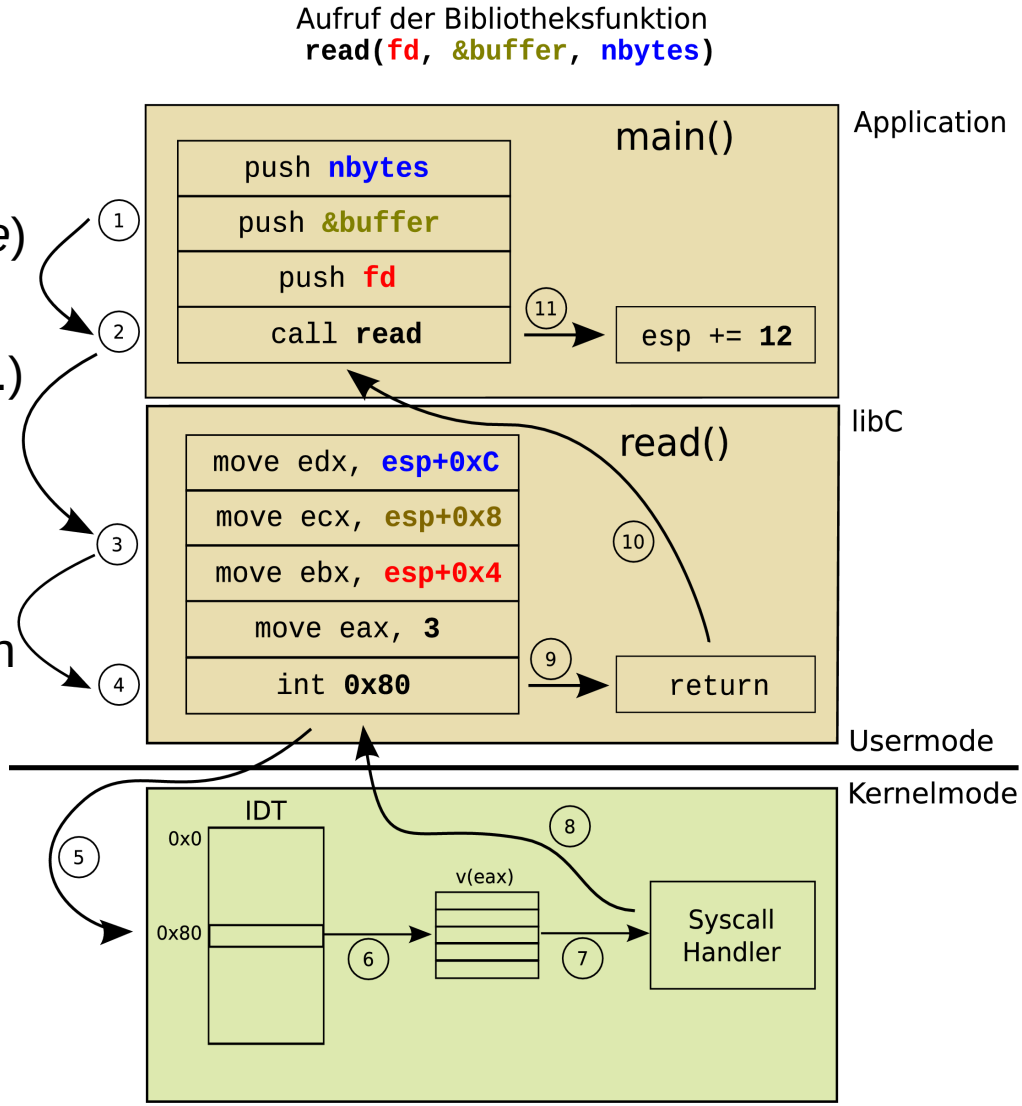
```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h> /* hier wird SYS_read=3 definiert */
#include <sys/types.h>

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