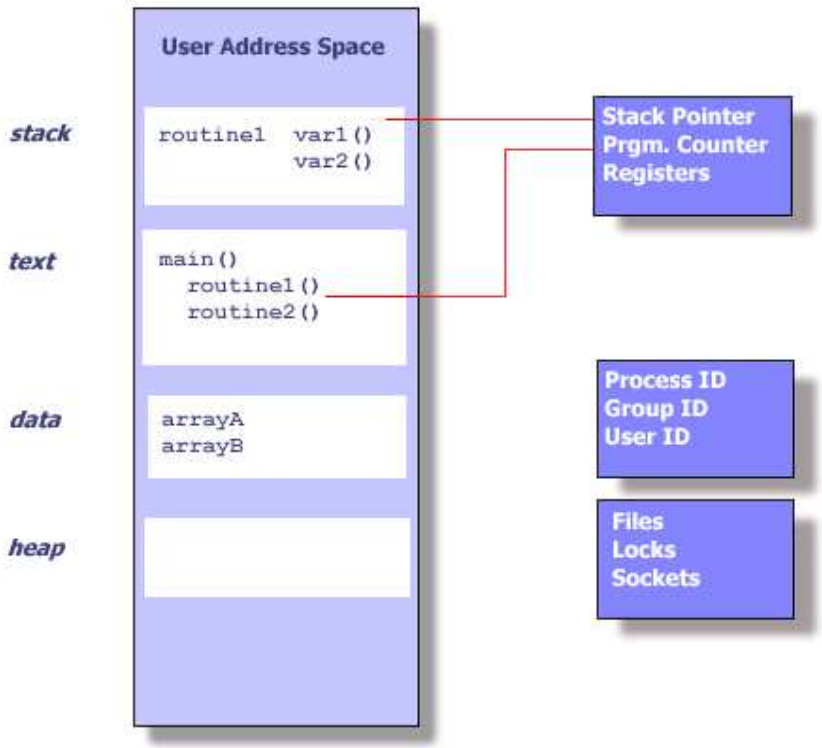
Kapitel 7: POSIX Threads (PThreads)

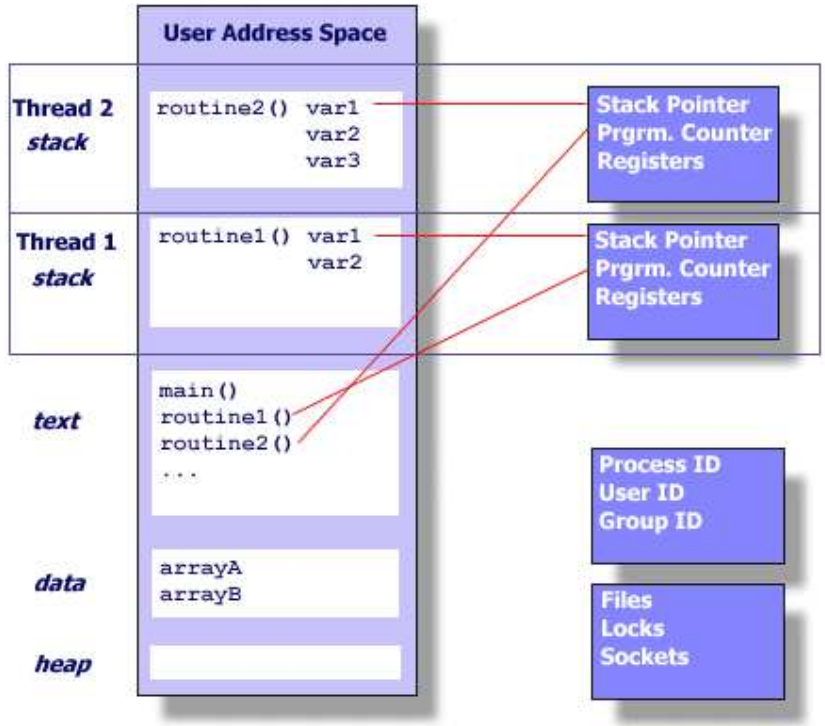
7: POSIX Threads (PThreads)

- IEEE POSIX 1003.1c-1995
  C-Standardinterface (API) für Threads
- Ermöglicht *multi-threaded processes*
- Wird unterstützt von:
  - Unixoiden (BSD, Linux, Mac OS X, Solaris)
  - Microsoft Windows (Subsystems für UNIX-basierte Anwendungen (SUA))
    - Windows Server 2008, Windows Vista (Enterprise und Ultimate)
    - Windows Server 2008 R2, Windows 7 (Enterprise und Ultimate)
    - Windows Server 2012, Windows 8 (Enterprise)
    - ...

Unix-Threads-Übersicht



Unix-Prozess



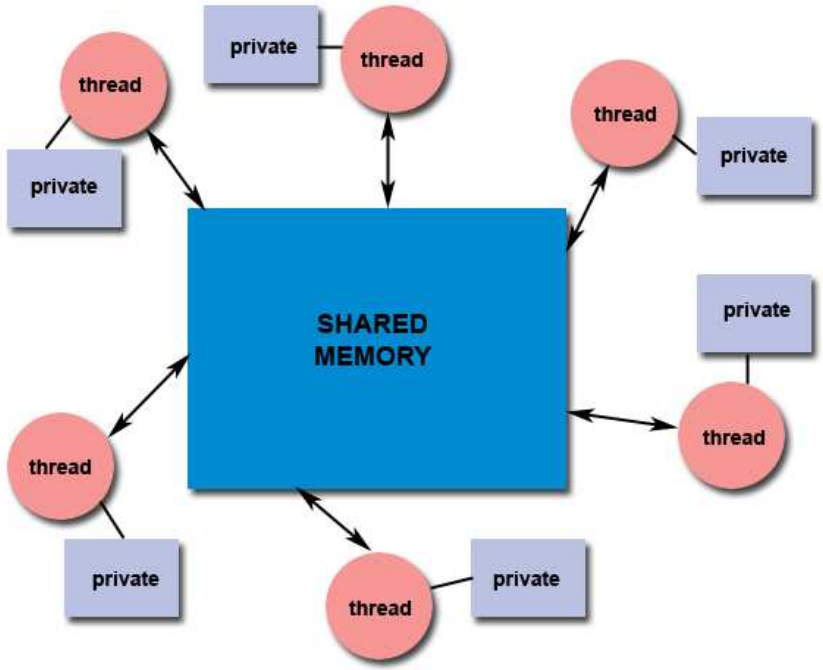
Threads innerhalb eines Unix-Prozesses

Quelle: `https://computing.llnl.gov/tutorials/pthreads/`

Anmerkungen

- Threads existieren innerhalb eines Prozesses
- Threads können vom Betriebssystem-Scheduler verarbeitet werden
- Der Status eines Threads besteht aus:
  - Stack-Pointer und Register
  - `errno`
  - Singalmasken
  - Scheduling-Eigenschaften (z. B. Echtzeit-Prioritäten)
  - Thread-spezifischen Daten (z. B. Thread ID)
- Der Thread-Status ermöglicht die unabhängige Flusskontrolle

Shared Memory



Quelle: `https://computing.llnl.gov/tutorials/pthreads/`

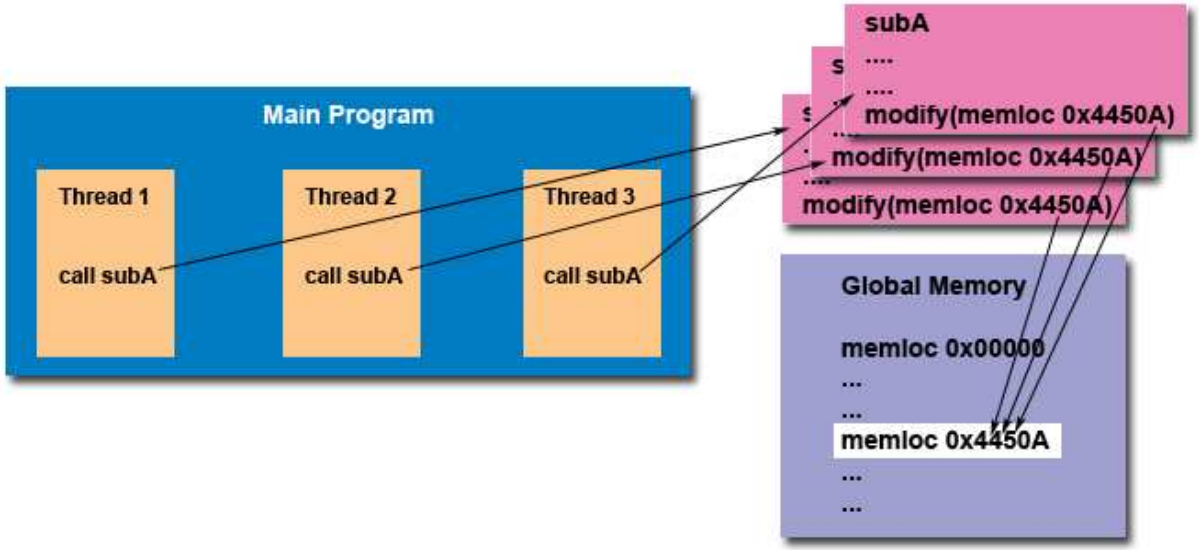
- Jeder Thread hat seinen eigenen Stack (ca. 2MB privater Speicher)
- Threads innerhalb eines Prozesses teilen sich den Heap-Speicher und globale (statische) Variablen (Shared Memory)
- Für die Synchronisation ist der Programmierer verantwortlich
- **Frage:** Welche Probleme können auftreten?

Shared Resources

Neben Shared-Memory teilen sich die Threads innerhalb eines Prozesses u. a. noch folgenden Ressourcen:

- Das `text`-Segment
- Einen Eintrag in der Prozesstabelle
  - Datei-Deskriptoren
  - PID, UID, GID
  - Sockets
  - ...
- **Frage:** Welche Probleme können hier auftreten?

Threadsicherheit (*Thread Safety*)



Quelle: `https://computing.llnl.gov/tutorials/pthreads/`

- Eine Funktion ist threadsicher (reentrant) falls mehrere Threads diese ohne Gefahr parallel aufrufen können
- Funktionen die Shared Memory modifizieren sind oftmals nicht reentrant
- Die Manpage verrät ob eine Funktion threadsicher ist
- Im Zweifel sind Funktionen **nicht** threadsicher

Threads vs. Prozesse

- Die gemeinsame Benutzung von Daten ist bei Threads, im Gegensatz zu Prozessen, einfach
- Die Erzeugung eines Threads ist schneller als die eines Prozesses (Faktor 5 bis 10)
- Der Kontextwechsel zwischen Threads ist einfacher als zwischen Prozessen
- Threads sollten nur threadsichere Funktionen aufrufen
- Ein Fehler in einem Thread kann sich auf die anderen Threads auswirken
- Threads müssen sich einen virtuellen (Userspace-)Adressraum teilen (32-bit Windows: 2GB, 32-Bit Linux: 3 GB)
- Wird ein Prozess beendet, werden dessen Threads terminiert

Einsatzgebiete von Threads

- Ein Server kann für jede Verbindung einen neuen Thread generieren
- Ein Webbrowser kann für jeden Tab einen neuen Thread generieren
- Graphische Oberflächen
- Spiele (KI, Benutzereingabe, Rendering)
- **Frage:** Fallen Ihnen noch weiter Einsatzgebiete ein?

Eigenschaften von PThread

- Das Compilerflag `-pthread` wird benötigt (Beispiel: `# gcc -o foo foo.c -pthread`):
  - Aktivierung des Makros `_REENTRANT`.
  - Programm wird gegen die Bibliothek `libpthread` gebunden.

- Alle PThread Funktionen geben bei Erfolg `0` zurück, ansonsten wird `errno` gesetzt und `-1` zurückgegeben

7.1: Thread Management

In diesem Abschnitt behandeln wir die folgenden Funktionen:

```
pthread_create()
pthread_exit()
pthread_self()
pthread_join()
pthread_detach()
```

PThread-Funktion geben im Erfolgsfall meist `0` zurück, ansonsten wird `errno` gesetzt und ein Wert ungleich Null zurück gegeben.

Thread-Generierung

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void* (*start_routine)(void *),
                   void *arg);
```

- **`thread`**: Thread ID
- **`attr`**: Thread kann verschiedenen Attributen generiert werden. Zunächst setzen wir diesen Wert auf **`NULL`**
- **`start_routine()`**: Startpunkt des Threads
- **`arg`**: Argument der Start-Routine
- **Frage**: Können der Start-Routine mehrere Argumente übergeben werden?

Hallo Welt

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  #define NUM_THREADS     5
7
8  void *start(void *name) {
9    char *t = (char *) name;
10   printf("Hello␣World!␣It's␣me,␣thread␣%c!\n", *t);
11   return EXIT_SUCCESS;
12 }
13
14  int main () {
15    pthread_t thread[NUM_THREADS];
16    char tname[NUM_THREADS];
17    for(int t=0; t<NUM_THREADS; t++){
18      tname[t]='A'+t;
19      if ( pthread_create(&thread[t], NULL, &start, (void *) (tname+t) )) {
20        perror("pthread_create()");
21        exit(EXIT_FAILURE);
22      }
23    }
24    usleep(1000);
25    return EXIT_SUCCESS;
26  }
```

Terminierung von Threads

```
#include <pthread.h>

void pthread_exit(void *retval);
```

- Funktion terminiert den aufrufenden Thread
- **`retval`**: Exit Code des Threads
- Thread-spezifische Ressourcen (z. B. dessen Stack) werden freigegeben
- Gemeinsame Ressourcen (wie Datei-Deskriptoren oder Semaphoren) werden nicht freigegeben
- Wird der letzte Thread beendet, führt dieser die Anweisung **`exit(0)`** aus
- Ein **`pthread_exit()`** Aufruf ist äquivalent mit einer **`return`**-Anweisung in der Start-Routine des Threads

Einfaches Beispiel

```
 1  #include <pthread.h>
 2  #include <unistd.h>
 3  #include <stdlib.h>
 4
 5  void foo() {
 6    write(STDOUT_FILENO,"foo\n",4);
 7  }
 8
 9  void bar() {
10    write(STDOUT_FILENO,"bar\n",4);
11    pthread_exit(NULL);
12  }
13
14  void* start(void *arg)  {
15    *((int *) arg) ? foo() : bar();
16    return EXIT_SUCCESS;
17  }
18
19  int main () {
20    int a=0, b=1;
21    pthread_t t1, t2;
22
23    pthread_create(&t1, NULL, start, &a);
24    pthread_create(&t2, NULL, start, &b);
25
26    usleep(1000);
27    return EXIT_SUCCESS;
28  }
```

Thread ID

```
#include <pthread.h>

pthread_t pthread_self(void);
```

- Gibt die Thread-ID (TID) des aufrufenden Threads zurück
- Einige PThread-Funktionen benötigen TIDs als Argument. Beispiel: **`pthread_kill()`**
- Unter Linux entspricht **`pthread_t`** einem **`unsigned long int`**
- Vorsicht: Je nach OS kann **`pthread_t`** auch ein **`struct`** sein

```
#include <pthread.h>

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

`pthread_equal()` gibt genau dann **0** zurück wenn die beiden Thread-IDs **ungleich** sind, ansonsten nicht.

Auf die Beendigung eines Threads warten

```
#include <pthread.h>

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

- **`pthread_join()`** wartet bis der Thread **`thread`** terminiert und gibt dessen verwendete Ressourcen wieder frei
- Ist der Thread bereits terminiert, kehrt die Funktion sofort zurück
- **`retval!=NULL`**: Liefert den Rückgabewert des Threads (Start-Funktion oder **`pthread_exit()`**-**`Parameter`**)
- Wollen mehrere Threads mit dem gleichen Thread *joinen*, so ist das Verhalten undefiniert

  Beispiel: *Thread ID Kollision mit neuem Thread*

Unterschiede zwischen pthread_join und waitpid

- **`waitpid()`** kann nur auf einen Kindprozess warten; **`pthread_join()`** kann auf einen beliebigen Thread (innerhalb des Prozess) warten
- Wäre es mit **`pthread_join()`** möglich auf beliebige Threads zu warten, dann könnten u. U. verwendete Bibliotheken nicht mehr den Beendigungsstatus bestimmter Threads erfragen
- Es gibt **`nicht-joinable`** Threads, bei denen der Aufruf von **`pthread_join()`** fehlschlägt
- **`waitpid()`** funktioniert bei allen Kindprozessen

Beispiel: Auf Beendigung eines Threads warten

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <errno.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7  static void* start(void *arg) {
8    puts((char *) arg);
9    size_t *r = malloc(sizeof(size_t));
10   *r = strlen((char *) arg);
11   return (void *) r;
12 }
13
14 int main() {
15   pthread_t tid;
16   void *res;
17
18   if(pthread_create(&tid, NULL, start, "Hello␣World"))
19     perror("thread_create()");
20
21   pthread_join(tid, &res);
22   printf("Thread␣returned␣%ld\n",  *((size_t *) res));
23   free(res);
24 }
```

Beispiel: Double Join

```
1  #include <pthread.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  static int fehler;
6
7  void* twait(void *arg) {
8    pthread_t *t = (pthread_t *) arg;
9    if(pthread_join(*t, NULL)) fehler += 2;
10   pthread_exit(NULL);
11 }
12
13 void* tpause(void *arg) {
14   if(!arg) sleep(1);
15   return NULL;
16 }
17
18 int main() {
19   pthread_t t1, t2;
20
21   pthread_create(&t1, NULL, tpause, NULL);
22   pthread_create(&t2, NULL, twait, &t1);
23
24   if(pthread_join(t1, NULL)) fehler += 1;
25   pthread_join(t2, NULL);
26   return fehler;
27 }
```

Frage: Was ist der Rückgabewert dieses Programms?

Threads abwickeln

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

- Die Methode markiert einen Thread als **`detached`**
- Bei der Terminierung des *detachten* Threads werden dessen Resourcen wieder freigegeben
- Falls ein Thread bereits als **`detached`** markiert ist, ist das Verhalten undefiniert
- Ist ein Thread **`detached`**, ist er nicht mehr **`joinable`** und der Aufruf von **`pthread_join()`** schlägt fehl
- Man kann **entweder** **`pthread_join()`** **oder** **`pthread_detach()`** aufrufen um dessen Ressourcen nach der Terminierung wieder freizugeben

Beispiel: Detached (1/3)

```
1  #include <pthread.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <unistd.h>
5
6  void* start(void *arg)  {
7    if(arg)   pthread_exit(NULL);
8    else      pthread_exit(NULL);
9  }
10
11 int main () {
12   pthread_t tid;
13
14   pthread_create(&tid, NULL, &start, NULL);
15   if(pthread_detach(tid)) fputs("Err: detach\n",stderr);
16   if(pthread_join(tid, NULL)) fputs("Err: join\n",stderr);
17
18   usleep(1000);
19   return EXIT_SUCCESS;
20 }
```

Frage: Was ist die Ausgabe des Programms?

Beispiel: Detached (2/3)

```
1  #include <pthread.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <unistd.h>
5
6  void* start(void *arg)  {
7    if(arg)   pthread_exit(NULL);
8    else      pthread_exit(NULL);
9  }
10
11 int main () {
12   pthread_t tid;
13
14   pthread_create(&tid, NULL, &start, NULL);
15   if(pthread_join(tid, NULL)) fputs("Err: join\n",stderr);
16   if(pthread_detach(tid)) fputs("Err: detach\n", stderr);
17
18   usleep(1000);
19   return EXIT_SUCCESS;
20 }
```

Frage: Was ist die Ausgabe des Programms?

Beispiel: Detached (3/3)

```
1  #include <pthread.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <unistd.h>
5
6  void* start(void *arg)  {
7    if(!arg)  puts("foobar");
8    pthread_exit(NULL);
9  }
10
11 int main () {
12   pthread_t tid;
13
14   pthread_create(&tid, NULL, &start, NULL);
15   if(pthread_detach(tid)) fputs("Error: detach", stdout);
16
17   usleep(1000);
18   return EXIT_SUCCESS;
19 }
```

Frage: Was ist die Ausgabe des Programms?

7.2: Pthread Mutex

- In diesem Abschnitt werden Pthread-Mutexe vorgestellt
- Wiederholung:
  - Mutex steht für ***mut****ual* ***ex****clusion*) (**Wechselseitiger Ausschluss**).
  - Mutexe synchronisieren den Zugriff und die Nutzung von gemeinsamen Ressourcen wie Memory.
- **Achtung:** Mutex-Funktionen sollen niemals innerhalb eines Signalhandlers aufgerufen werden, da diese nicht *async-signal safe* sind (Deadlock Gefahr).

Zugriff auf Shared Variables

- Der (Schreib-)Zugriff auf eine gemeinsam genutzte Variable sollte immer *atomar* sein, da kritischer Abschnitt.
- Wir wissen: In einem kritischen Abschnitt darf sich zu jedem Zeitpunkt immer nur max. ein Thread aufhalten.
- Der Wert gemeinsam genutzter Variablen ist **ungewiss**, falls deren Schreibzugriffe **nicht atomar** ist.

Beispiel: Unsynchronisierter Zugriff

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define LOOPS 1000000
6  static volatile int shared = 0;
7
8  static void *start(void *arg) {
9    int n = *(int *) arg;
10
11   for(int j=0; j<n; j++) shared += 1;
12   return NULL;
13 }
14
15 int main() {
16   pthread_t t1, t2;
17   int n = LOOPS;
18
19   pthread_create(&t1, NULL, start, &n);
20   pthread_create(&t2, NULL, start, &n);
21
22   pthread_join(t1, NULL);
23   pthread_join(t2, NULL);
24   printf("Shared: %d\n", shared);
25
26   return EXIT_SUCCESS;
27 }
```

Fehleranalyse

- Was ging schief?
- Warum ist die Anweisung `shared +=1` nicht atomar?
- Wie können wir `shared +=1` als Assembler-Anweisungen ausdrücken?
- Welche Race-Conditions können auftreten?

Sperren und Entsperren

```
#define __GNU_SOURCE
#include<pthread.h>
PTHREAD_MUTEX_INITIALIZER
PTHREAD_MUTEX_ERRORCHECK_NP

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

- Makros **`PTHREAD_MUTEX_INITIALIZER`** und **`PTHREAD_MUTEX_ERRORCHECK_NP`** initalisieren statisch allokierte Mutexe
- **`pthread_mutex_lock()`**: Aufrufender Thread sperrt (*locked*) den Mutex **`mutex`**. Falls der Mutex bereits gesperrt ist, wird der Thread blockiert bis **`mutex`** wieder freigegeben (*unlocked*) wurde
- **`pthread_mutex_unlock()`**: Aufrufender Thread gibt den Mutex **`mutex`** wieder frei

Mutexe

- **`PTHREAD_MUTEX_INITIALIZER`**
  Mutex kann von einen bliebigen Thread freigegeben werden

- **`PTHREAD_MUTEX_ERRORCHECK_NP,`**
  Mutex kann nur von dem sperrenden Thread wieder freigegeben werden

  Benötigtes Makro: `__GNU_SOURCE`

Beispiel: Synchronisierter Zugriff

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  static int loops = 1000000;
6  static volatile int shared = 0;
7  static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
8
9  static void *start(void *arg) {
10   int n = *(int *) arg;
11
12   for(int j=0; j<n; j++) {
13     pthread_mutex_lock(&mutex);
14     shared += 1;
15     pthread_mutex_unlock(&mutex);
16   }
17   return NULL;
18 }
19
20 int main() {
21   pthread_t t1, t2;
22
23   pthread_create(&t1, NULL, start, &loops);
24   pthread_create(&t2, NULL, start, &loops);
25   pthread_join(t1, NULL);
26   pthread_join(t2, NULL);
27   printf("Shared: %d\n", shared);
28
29   return EXIT_SUCCESS;
30 }
```

Mutex testen

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Methode blockiert nicht sondern liefert **`EBUSY`** zurück falls **`mutex`** gesperrt ist; andernfalls wird der Mutex gesperrt und `0` zurückgegeben
- Vermeidung von Deadlocks durch try-then-back- off-Strategie

  `trylock()` Aufruf schlägt fehl → Thread gibt Mutexe die er *besitzt* frei, und versucht die Aktion nochmals auszuführen
- Threads mit unabhängigen Haupt- und Nebenaufgaben

  Beispiel: Wäsche waschen und Wohnung saugen. Falls die Waschmaschine besetzt ist, dann wird die Wohnung gesaugt

Beispiel: Haupt- und Nebenaufgabe I

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  static int loops = 100;
7  static volatile int shared1, shared2;
8  static pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
9  static pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
10
11 void optional_task() {
12   pthread_mutex_lock(&mutex2);
13   shared2 += 1;
14   pthread_mutex_unlock(&mutex2);
15 }
16
17 void task() {
18   while(pthread_mutex_trylock(&mutex1)) optional_task();
19   shared1 += 1;
20   usleep(1000); // 1ms
21   pthread_mutex_unlock(&mutex1);
22 }
```

Beispiel: Haupt- und Nebenaufgabe II

```
23
24 static void *start(void *arg) {
25   int n = *(int *) arg;
26   for(int j=0; j< n; j++) task();
27   return NULL;
28 }
29
30 int main() {
31   pthread_t t1, t2;
32   pthread_create(&t1, NULL, start, &loops);
33   pthread_create(&t2, NULL, start, &loops);
34
35   pthread_join(t1, NULL);
36   pthread_join(t2, NULL);
37
38   printf("Shared1:␣%d\n", shared1);
39   printf("Shared2:␣%d\n", shared2);
40   return EXIT_SUCCESS;
41 }
```

Frage: Was können Sie über die Ausgabe des Programms sagen?

Deadlocks

```
Thread A                        Thread B

pthread_mutex_lock(&m1);        pthread_mutex_lock(&m2);
pthread_mutex_lock(&m2);        pthread_mutex_lock(&m1);
```

- Wiederholung: Deadlocks sind zyklische Abhängigkeiten
- Deadlock-Vermeidungsstrategie: Aufbau einer logischen Mutex-Hierarchie bei der Mutexe nur nach bestimmter Reihenfolge gesperrt werden dürfen

Beispiel: Aufsteigende Reihenfolge

Seien $M_1, \ldots, M_n$ die genutzten Mutexe. Dann darf ein Thread nach Sperrung des Mutex M_i nur noch die Mutexe M_j mit $j > i$ sperren.

Beispiel: Deadlock I

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  static volatile int shared;
7  static pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
8  static pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
9
10 void foo() {
11   pthread_mutex_lock(&mutex2);
12   pthread_mutex_lock(&mutex1);
13   shared += 2;
14   pthread_mutex_unlock(&mutex1);
15   pthread_mutex_unlock(&mutex2);
16 }
17
18 void bar() {
19   pthread_mutex_lock(&mutex1);
20   pthread_mutex_lock(&mutex2);
21   shared += 1;
22   pthread_mutex_unlock(&mutex2);
23   pthread_mutex_unlock(&mutex1);
24 }
25
26 static void *start(void *arg) {
27   if(arg) for(int i=0; i<1000; i++) foo();
28   else    for(int i=0; i<1000; i++) bar();
29   return NULL;
```

Beispiel: Deadlock II

```
30  }
31
32  int main() {
33    pthread_t t1, t2;
34    int i=1;
35    pthread_create(&t1, NULL, start, &i);
36    pthread_create(&t2, NULL, start, NULL);
37
38    pthread_join(t1, NULL);
39    pthread_join(t2, NULL);
40
41    printf("Shared:␣%d\n", shared);
42    return EXIT_SUCCESS;
43  }
```

Frage: Was können Sie über die Ausgabe des Programms sagen?

Beispiel: Kein Deadlock I

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  static volatile int shared;
7
8  static pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
9  static pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
10
11 void foo() {
12   pthread_mutex_lock(&mutex2);
13   pthread_mutex_unlock(&mutex2);
14   pthread_mutex_lock(&mutex1);
15   pthread_mutex_lock(&mutex2);
16   shared += 2;
17   pthread_mutex_unlock(&mutex2);
18   pthread_mutex_unlock(&mutex1);
19 }
20
21 void bar() {
22   pthread_mutex_lock(&mutex1);
23   pthread_mutex_lock(&mutex2);
24   shared += 1;
25   pthread_mutex_unlock(&mutex2);
26   pthread_mutex_unlock(&mutex1);
27 }
28
29 static void *start(void *arg) {
```

Beispiel: Kein Deadlock II

```
30    if(arg) for(int i=0; i<1000; i++) foo();
31    else    for(int i=0; i<1000; i++) bar();
32    return NULL;
33  }
34
35  int main() {
36    pthread_t t1, t2;
37    int i=1;
38    pthread_create(&t1, NULL, start, &i);
39    pthread_create(&t2, NULL, start, NULL);
40
41    pthread_join(t1, NULL);
42    pthread_join(t2, NULL);
43
44    printf("Shared: %d\n", shared);
45    return EXIT_SUCCESS;
46  }
```

Frage: Was können Sie über die Ausgabe des Programms sagen?

7.3: Vorzeitiges Beenden von Threads

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

- Funktion sendet ein Beendigungssignal (**cancellation request**) an den Thread **thread**
- Das Verhalten des Threads hängt von zwei seiner Attribute ab: **cancelability state** und **cancelability type**
- Bei der vorzeitigen Beendigung wird wie folgt vorgegangen:
  1. Die Cleanup-Handler werden der Reihe nach aufgerufen
  2. Destruktoren für lokale Daten werden aufgerufen
  3. Der Thread wird mittels **pthread_exit()** beendet

Cancelability State

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *ostate);
int pthread_setcanceltype(int type, int *oldtype);
```

- **`setcancelstate()`**: Setzt den **`cancelability state`** auf den Wert **`state`** und gibt den alten Status (**`ostate`**) zurück
  - **`PTHREAD_CANCEL_ENABLE`**: Der Thread kann vorzeitig beendet werden (Standardeinstellung)
  - **`PTHREAD_CANCEL_DISABLE`**: Beendigungssignal hängt bis der Status wieder auf **`PTHREAD_CANCEL_ENABLE`** zurückgesetzt wird
- **`setcanceltype()`**: Analog zu **`setcancelstate()`**
  - **`PTHREAD_CANCEL_ASYNCHRONOUS`**: Der Thread kann jederzeit beendet werden.
  - **`PTHREAD_CANCEL_DEFERRED`**: Der Thread wird beim nächsten Unterbrechungspunkt beendet (Standardeinstellung)

Unterbrechungspunkte (Cancellation Points)

- POSIX:2008 (Single UNIX Specification 4) schreibt für eine Liste von (blockierenden) Funktionen Unterbrechungspunkte vor (siehe `http://pubs.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_09.html`)

- Bei einem Unterbrechungspunkt übernimmt der Scheduler wieder die Kontrolle über die CPU und deaktiviert den Thread
- Dem Thread bleibt damit der Zugriff auf die CPU verwehrt
- Durch Unterbrechungspunkte können invalide Programmzustände vermieden werden (z. B. **`write()`**)

Beispiel: Vorzeitiges Beenden von Threads

```
 1  #include <pthread.h>
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <unistd.h>
 5
 6  static void* start(void *arg) {
 7    if(!arg) puts("Thread:␣started");
 8    for(int i=1; i >0 ; i++) {
 9      printf("%d\n",i);
10      sleep(1);
11    }
12    return NULL;
13  }
14
15  int main() {
16    pthread_t tid;
17    void *res;
18
19    pthread_create(&tid, NULL, start, NULL);
20    sleep(3);
21    pthread_cancel(tid);
22    pthread_join(tid, &res);
23
24    if(res == PTHREAD_CANCELED) puts("Thread:␣canceled");
25
26    return EXIT_SUCCESS;
27  }
```

Unterbrechungspunkte anlegen

```
#include <pthread.h>

void pthread_testcancel(void);
```

- Die Funktion generiert einen Unterbrechungspunkt
- Ein aufrufenden Prozess mit den Attributen **PTHREAD_CANCEL_ENABLE** und **PTHREAD_CANCEL_DEFERRED** wird bei einem hängenden Beendigungssignal beendet
- Wichtig bei Threads die keine Funktion mit Unterbrechungspunkt aufrufen

Cleanup Handler

- Durch das vorzeitige Beenden eines Threads kann dieser in einen inkonsistenten Zustand geraten (z. B. gesperrte Mutexe)
- Dies kann zu Deadlocks oder Programmabstürzen führen

- Daher können Threads Cleanup-Handler registrieren, welche nach dem vorzeitige Beenden automatisch aufgerufen werden
- Hauptaufgaben eines Cleanup-Handlers
  1. Gesperrte Mutexe freigeben
  2. Globale Variablen modifizieren
  3. Freigabe von alloziertem Speicher

- Jeder Thread verfügt über einen Stack von Cleanup-Handlern
- Der Thread terminiert, nachdem alle Cleanup-Handler abgearbeitet wurden

Cleanup Handler Stack

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *),
                          void *arg);
void pthread_cleanup_pop(int execute)
```

- **`pthread_cleanup_push()`**
  Legt den Cleanup-Handler **`routine`** auf den Stack. Beim Aufruf wird das Argument **`arg`** übergeben
- **`pthread_cleanup_pop(int execute)`**
  Nimmt den obersten Cleanup-Handler vom Stack
- Der Cleanup-Handler wird ausgeführt falls **`execute != 0`** gilt
- Eine Funktion muss für jede **`push()`**-Funktion die sie aufruft auch wieder eine **`pop()`**-Funktion aufrufen

Beispiel: Cleanup Handler I

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  static volatile int shared = 0;
7  static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
8
9  static void handler1(void *arg) {
10   free(arg); puts("Buffer freed");
11 }
12
13 static void handler2(void *arg) {
14   if(!arg) pthread_mutex_unlock(&mutex);
15   puts("Bye");
16 }
17
18 static void *start(void *arg) {
19   char *buf = malloc(100);
20   pthread_cleanup_push(handler1, buf);
21   pthread_cleanup_push(handler2, NULL);
22
23   while(!arg) {
24     pthread_mutex_lock(&mutex);
25     shared+=1;
26     pthread_testcancel();
27     pthread_mutex_unlock(&mutex);
28   }
29   pthread_cleanup_pop(0);
```

Beispiel: Cleanup Handler II

```
30    pthread_cleanup_pop(1);
31
32    return NULL;
33  }
34
35  int main() {
36    pthread_t t1, t2;
37
38    pthread_create(&t1, NULL, start, NULL);
39    pthread_create(&t2, NULL, start, NULL);
40
41    sleep(1);
42    pthread_cancel(t1);
43
44    sleep(1);
45    pthread_cancel(t2);
46
47    pthread_join(t1, NULL);
48    pthread_join(t2, NULL);
49
50    printf("shared:␣%d\n", shared);
51    return EXIT_SUCCESS;
52  }
```

Zusammenfassung

Nach diesem Kapitel sollten Sie . . .

- . . . Pthreads generieren und terminieren können.
- . . . wissen wie man auf einen Pthreads wartet.
- . . . den Umgang mit Mutexen beherrschen.
- . . . wissen wie sich PThreads vorzeitig beendet lassen.
- . . . wissen was ein Unterbrechungspunkt ist.