5. *PThreads – Thread-basierte Programmierung von Multicore-Systemen*

Pthreads: POSIX Thread API

- Standard Thread API – IEEE POSIX 1003.1c standard

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

- 3 Klassen von Routinen
  - Threadverwaltung
    Erzeugung, Termination, Synchronisation,
    Setzen und Abfragen von Attributwerten
  - Wechselseitiger Ausschluss (mutexes)
    Erzeugung, Löschen, Sperren und Entsperren von Mutexvariablen;
    Setzen und Modifikation von Mutex.-Attributen
  - Bedingungsvariablen
    Erzeugung, Entfernen, Warten und Signalisieren auf
    Bedingungsvariablen, Setzen/Abfragen von Attributwerten

Namenskonventionen

Alle Bezeichner beginnen mit `pthread_`.

Präfix	Funktionale Gruppe
`pthread_`	Threads selbst und Routinen
`pthread_attr_`	Thread Attributobjekte
`pthread_mutex_`	Mutexes
`pthread_mutexattr_`	Mutex Attributobjekte
`pthread_cond_`	Bedingungsvariablen
`pthread_condattr_`	Attributobjekte zu Bedingungsvariablen

Compilation von Programmen mit `gcc -pthread`

Threadverwaltung

- Thread-Erzeugung

```
int pthread_create (
  pthread_t  *thread_handle,
  const pthread_attr_t *attribute,
  void * (*thread_function){void *},
  void  *arg);
```

- Thread-Synchronisation

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

- Thread-Termination
  - Thread beendet Ausführung
  - Aufruf von `pthread_exit()`
  - Annulierung durch anderen Prozess mittels `pthread_cancel`
  - Prozess terminiert

Beispiel: Hello World

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *PrintHello(void *threadid)  {
   int tid; tid = (int)threadid;
   printf("Hello World! It's me, thread #%d!\n", tid);
   pthread_exit(NULL);
}
int main (int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS]; int rc, t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);       }   }
   pthread_exit(NULL);
}
```

Argumentübergabe

- Weitergabe eines Arguments an die Start-Routine eines Threads
- call-by-reference Übergabe mit einem cast zu `(void *)`

```
int *taskids[NUM_THREADS];
...
for(t=0;t<NUM_THREADS;t++) {
  taskids[t] = (int *) malloc(sizeof(int));
  *taskids[t] = t; printf("Creating thread %d\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello,
                           (void *) taskids[t]);
  ...  }
```

- Mehrere Argumente müssen in einer Struktur zusammengefasst werden.

Beispiel: Übergabe mehrerer Argumente

```
struct thread_data  {
        int     thread_id;    int  sum;     char *message;    };
struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)   {
   ...; struct thread_data *my_data;
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;    sum = my_data->sum;
   hello_msg = my_data->message;   ...    }

int main(int argc, char *argv[])   {  ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *)
                          &thread_data_array[t]);  ...
  }
```

Beispiel: Thread Synchronisation

Das Attributargument von `pthread_create ()` wird genutzt, um einen Thread als „joinable“ oder „detached“ (nicht joinable) zu erzeugen.
typischer Ablauf:

```
pthread_attr_t attr;           /* Deklaration */
...
pthread_attr_init(&attr);      /* Initialisierung */
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
                        /* Status joinable sicherstellen */
...
rc = pthread_create(&thread[t], &attr, BusyWork, NULL);
...
pthread_attr_destroy(&attr);
....
rc = pthread_join(thread[t], &status);
```

=> Programm: join1.c

Fallstudie: Berechnung von pi

```
#include <pthread.h>
#include <stdlib.h>
...
#define MAX_THREADS 512
void *compute_pi (void *);
...
main()
{  ...
   pthread_t
       p_threads[MAX_THREADS];
   pthread_attr_t  attr;
   ...
   pthread_attr_init (&attr);
   ...
```

```
for (i=0; i< num_threads; i++)
   {  hits[i][0] = i;
      pthread_create(
         &p_threads[i], &attr,
         compute_pi,
         (void *) &hits[i][0]);
   }
for (i=0; i< num_threads; i++)
   {  pthread_join(
         p_threads[i], NULL);
      total_hits += hits[i][0];
   }
   ...
   }
}
```

Beispiel: Berechnung von pi (Forts.)

```
void *compute_pi (void *s)  {
  int thread_no, i, *hit_pointer; double rand_no_x, rand_no_y;
  int hits;
  hit_pointer = (int *) s; thread_no = *hit_pointer; hits = 0;

  srand48(thread_no);
  for (i = 0; i < sample_points_per_thread; i++) {
    rand_no_x = (double)(rand_r(&thread_no))/(double)((2 <<30) - 1);
    rand_no_y = (double)(rand_r(&thread_no))/(double)((2 <<30) - 1);
    if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
      (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
      (*hit_pointer) ++;
    hits ++;
  }
  *hit_pointer = hits;
}
```

Wechselseitiger Ausschluss

mittels binärer Semaphor-Variablen: mutex-locks

- **Sperren**

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

- **Entsperren**

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

- **Initialisieren**

```
int pthread_mutex_init (
   pthread_mutex_t *mutex_lock,
   const pthread_mutexattr_t *lock_attr);
```

Beispielanwendung

```
pthread_mutex_t minimum_value_lock;
...
main() {
    ....
    pthread_mutex_init(&minimum_value_lock, NULL);
    ....
}
void *find_min(void *list_ptr) {
    ....
    pthread_mutex_lock(&minimum_value_lock);
    if (my_min < minimum_value)
    minimum_value = my_min;
    pthread_mutex_unlock(&minimum_value_lock);
}
```

=> Programme dotprod_serial.c, dotprod_mutex.c

Verringerung des Locking-Overheads

`pthread_mutex_trylock` als Alternative zu `pthread_mutex_lock`

```
pthread_mutex_trylock.
int pthread_mutex_trylock (
     pthread_mutex_t *mutex_lock);
```

- keine Blockade, sondern Rückgabe von EBUSY
- schneller, da keine Warteschlangenverwaltung

Beispiel: k Kopien eines Elementes finden

```
void *find_entries(void *start_pointer) {
  struct database_record *next_record; int count;
  current_pointer = start_pointer;
  do {     next_record = find_next_entry(current_pointer);
           count = output_record(next_record);
  } while (count < requested_number_of_records); }

int output_record(struct database_record *record_ptr) {
 int count;
 pthread_mutex_lock(&output_count_lock);
 output_count ++; count = output_count;
 pthread_mutex_unlock(&output_count_lock);
 if (count <= requested_number_of_records)
  print_record(record_ptr);
 return (count);
}
```

Beispiel: ... mit trylock

```
int output_record(struct database_record *record_ptr) {
 int count; int lock_status;
 lock_status=pthread_mutex_trylock(&output_count_lock);
 if (lock_status == EBUSY) {
  insert_into_local_list(record_ptr);
  return(0);
 }
 else {
  count = output_count;
  output_count += number_on_local_list + 1;
  pthread_mutex_unlock(&output_count_lock);
  print_records(record_ptr, local_list,
                  requested_number_of_records - count);
  return(count + number_on_local_list + 1);
 }
}
```

Bedingungsvariablen

- zur einseitigen Synchronisation
- immer mit mutex-lock assoziiert

- Funktionen

```
int pthread_cond_wait(  pthread_cond_t *cond,
                        pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_init(  pthread_cond_t *cond,
                    const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Beispiel: Producer-Consumer

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */
main() {
   /* declarations and initializations */
   task_available = 0;
   pthread_cond_init(&cond_queue_empty, NULL);
   pthread_cond_init(&cond_queue_full, NULL);
   pthread_mutex_init(&task_queue_cond_lock, NULL);
   /* create and join producer and consumer threads */
}
```

Beispiel (Forts.): Producer

```
void *producer(void *producer_thread_data) {
    int inserted;
    while (!done()) {
        create_task();
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 1)
            pthread_cond_wait(&cond_queue_empty,
                &task_queue_cond_lock);
        insert_into_queue();
        task_available = 1;
        pthread_cond_signal(&cond_queue_full);
        pthread_mutex_unlock(&task_queue_cond_lock);
    }
}
```

Beispiel (Forts.): Consumer

```
void *consumer(void *consumer_thread_data) {
    while (!done()) {
    pthread_mutex_lock(&task_queue_cond_lock);
    while (task_available == 0)
        pthread_cond_wait(&cond_queue_full,
            &task_queue_cond_lock);
        my_task = extract_from_queue();
        task_available = 0;
        pthread_cond_signal(&cond_queue_empty);
        pthread_mutex_unlock(&task_queue_cond_lock);
        process_task(my_task);
    }
}
```

Fallstudie: Lese-/Schreibsperren (read/write locks)

- erweiterter Synchronisationsmechanismus
- vgl. Leser-/Schreiber-Problem
- realisierbar mit mutex- und Bedingungsvariablen

Beispielrealisierung einer RW-Sperrvariablen

vom Typ `mylib_rwlock_t`

- Zähler readers für Leser
- Zähler writer für Schreiber (ist 0 oder 1)
- 2 Bedingungsvariablen readers_proceed writer_proceed
- Zähler pending_writers
- wartender Schreiber
- Mutex-Variable read_write_lock zum Schutz des Datenbereichs

```
typedef struct {
  int readers;
  int writer;
  pthread_cond_t readers_proceed;
  pthread_cond_t writer_proceed;
  int pending_writers;
  pthread_mutex_t read_write_lock;
} mylib_rwlock_t;
```

- Initialisierung

```
void mylib_rwlock_init (mylib_rwlock_t *l) {
  l->readers = l->writer = l->pending_writers = 0;
  pthread_mutex_init( &(l->read_write_lock),   NULL);
  pthread_cond_init(  &(l->readers_proceed),   NULL);
  pthread_cond_init(  &(l->writer_proceed),    NULL);
}
```

- Funktionen
  - `mylib_rwlock_rlock` - Leserzugang
  - `mylib_rwlock_wlock` - Schreiberzugang
  - `mylib_rwlock_unlock` - Freigabe

Leserzugang

```
void mylib_rwlock_rlock(mylib_rwlock_t *l) {
   /* if there is a write lock or pending writers,
     perform condition wait.. else increment count of
     readers and grant read lock */
   pthread_mutex_lock(&(l->read_write_lock));
   while ( (l->pending_writers > 0) ||
           (l->writer > 0))
      pthread_cond_wait(&(l->readers_proceed),
                        &(l->read_write_lock));
   l -> readers ++;
   pthread_mutex_unlock(&(l->read_write_lock));
}
```

Schreiberzugang

```
void mylib_rwlock_wlock(mylib_rwlock_t *l) {
   /* if there are readers or writers, increment pending
     writers count and wait. On being woken, decrement
     pending writers count and increment writer count */

   pthread_mutex_lock(&(l->read_write_lock));
   while ((l->writer > 0) || (l->readers > 0)) {
      l -> pending_writers ++;
      pthread_cond_wait(&(l->writer_proceed),
                        &(l->read_write_lock));
      }
   l->pending_writers--; l -> writer++;
   pthread_mutex_unlock(&(l->read_write_lock));
}
```

Freigabe

```
void mylib_rwlock_unlock(mylib_rwlock_t *l) {
/* if there is a write lock then unlock, else if there are
   read locks, decrement count of read locks. If the count
   is 0 and there is a pending writer, let it through, else
   if there are pending readers, let them all go through */
pthread_mutex_lock(&(l->read_write_lock));
if (l->writer > 0)  l->writer = 0;
else if (l->readers > 0) l->readers--;
if ((l->readers == 0) && (l->pending_writers > 0))
   pthread_cond_signal(&(l->writer_proceed));
else if (l->readers > 0)
   pthread_cond_broadcast(&(l->readers_proceed));
pthread_mutex_unlock(&(l->read_write_lock));
}
```

Fallstudie: Lineare Barriere

```
typedef struct {
   pthread_mutex_t   count_lock;
   pthread_cond_t    ok_to_proceed;
   int count;
} mylib_barrier_t;


void mylib_init_barrier(mylib_barrier_t *b) {
   b -> count = 0;
   pthread_mutex_init( &(b -> count_lock),    NULL);
   pthread_cond_init(  &(b -> ok_to_proceed), NULL);
}
```

```
void mylib_barrier (mylib_barrier_t *b, int num_threads) {
    pthread_mutex_lock(&(b -> count_lock));
    b -> count ++;
    if (b->count == num_threads) {
        b->count = 0;
        pthread_cond_broadcast(&(b->ok_to_proceed));
    }
    else
        while (pthread_cond_wait(  &(b->ok_to_proceed),
                                   &(b->count_lock)) != 0);
    pthread_mutex_unlock(&(b -> count_lock));
}
```