Multithread Programming

Example Program (sequential version)

```
#include <stdio.h>

void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;


void main()
{
  do_one_thing( &r1 );
  do_another_thing( &r2 );
  do_wrap_up(r1, r2);
}



void do_one_thing(int *pnum_times)
{
  int i, j, x;

  for (i=0; i<4; i++) {
    printf("doing one thing\n");
    for (j=0; j<10000; j++) x += i;  // wait some time
    (*pnum_times)++;
  }
}



void do_another_thing(int *pnum_times)
{
  int i, j, x;

  for (i=0; i<4; i++) {
    printf("doing another thing\n");
    for (j=0; j<10000; j++) x += i;  // wait some time
    (*pnum_times)++;
  }
}



void do_wrap_up(int one_times, int another_times)
{
  int total;

  total = one_times + another_times;
  printf("wrap up: one thing %d, another %d, total %d\n",
         one_times, another_times, total);
}
```

Multithread Programming

Example Program (threaded version)

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

void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;

main()
{
  pthread_t thread1, thread2;

  pthread_create(&thread1,                // 1 parameter
                 NULL,                    // 2 parameter
                 (void *) do_one_thing,   // 3 parameter
                 (void *) &r1);           // 4 parameter

  pthread_create(&thread2,
                 NULL,
                 (void *) do_another_thing,
                 (void *) &r2);

  pthread_join(thread1, NULL); //wait for termination of thread
  pthread_join(thread2, NULL); //wait for termination of thread

  do_wrap_up(r1 , r2);
}


// functions from last example
```

Man-pages

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

int pthread_join(   pthread_t th,
                    void **thread_return);
```

Multithread Programming

Posix-Threads (1)

- sind unabhängige Befehlssequenzen innerhalb eines Prozesse
- teilen sich alle Resourcen (Speicher, Filedeskriptoren,...) eines Prozesses

Vorteile

- erhöhter Durchsatz
- Ausnutzung von Shared-Memory-Systemen mit mehreren Prozessoren,
- vermeiden von Deadlocks ...

Einbindung

- Deklarationen in *pthread.h*
- Linken der Library *libpthread*

Erzeugen eines Threads

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

thread liefert die Thread-ID des gestarteten Threads

attr enthält Attribute des Threads (wenn NULL, werden Default-Werte angenommen, ansonsten Scheduling und detachstate einstellbar)

start_fkt

arg

Multithread Programming

Posix-Threads (2)

Beenden eines Threads

```
void pthread_exit(void *status);
```

status kann von einen wartenden Thread verarbeitet werden. Ein return aus der Startroutine beendet Threads ebenso.

Warten auf die Beendigung eines Threads

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

Weitere Funktionen

```
pthread_t pthread_self(); // liefert eigene ID
int pthread_equal(pthread_t t1,pthread_t t2); // vergleicht zwei Threads
```

Multithread Programming

Posix-Threads (3)

Mutex - garantiert die exclusive Ausführung kritischer Code-Sequenzen

Das Mutexobjekt *mutex* wird mittels pthread_mutex_lock() gesperrt. Sollte der Mutex schon gelockt sein, blockiert der aufrufende Thread bis der Mutex verfügbar ist.

```
pthread_mutex_t mutex;
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);  //EBUSY if locked
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Initialisierung

```
int pthread_mutex_init( pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr); //dynamic

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; //static
```

Löschen eines Mutex

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

Lock/Unlock

```
int pthread_mutex_trylock(pthread_mutex_t *mp);
int pthread_mutex_lock(pthread_mutex_t *mp);
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

Ein Thread, welcher einen Mutex lockt, ist sein Besitzer. Nur der Besitzer kann einen Mutex unlocken.

Multithread Programming

Posix-Threads (4)

Condition Variable - warten auf Ereignisse

Initialisierung

```
pthread_cond_t cond;
int pthread_cond_init(  pthread_cond_t *cond,
                        const pthread_condattr_t *attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Löschen

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

Warten auf ein Ereignis

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

Blockiert den aufrufenden Thread bis ein anderer Thread das Eintreten des Ereignisses cond signalisiert. Der Mutex mutex wird unlocked bei Aufruf von *pthread_cond_wait* und wieder gelockt, bevor die Funktion beendet wird.

Signalisieren

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

Busy-wait ohne Condition-Variable

```
for(;;) {
      pthread_mutex_lock(mutex);
      if (cond) break;
      pthread_mutex_unlock(mutex);
}
<hier ist die Bedingung erfüllt>
pthread_mutex_unlock(mutex);
```

Nachteile: verbraucht sehr viel Resourcen, verlangt Programmierdisziplin

Version mit Condition-Variable

```
pthread_mutex_lock(mutex);
pthread_cond_wait(cond,mutex);
<hier ist die Bedingung erfüllt>
pthread_mutex_unlock(mutex);
```

Es ist möglich, daß ein *pthread_cond_signal* mehr als einen Thread aufweckt, ebenso kann ein *pthread_cond_wait* scheitern. Daher sollte nach dem Aufruf von *pthread_cond_wait* getestet werden, ob die Bedingung auch wirklich erfüllt ist.