1 Monitors

If mutual exclusion between threads is required, we can use a more sophisticated synchronization mechanism - a **monitor**. Monitors combine a simple binary semaphore (**mutex**) with condition variables to guarantee mutual exclusion while enabling threads to wait for specific conditions or events to be met.

1.1 POSIX mutexes

The POSIX standard provides a basic interface for interacting with mutexes:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
   const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

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

The `pthread_mutex_init` function is used to initialize a mutex. The `mutex` argument points to a previously allocated mutex structure, the `attr` argument points to a structure that defines some properties of the newly created mutex (if NULL is passed, the mutex will be initialized with the default properties).

The `pthread_mutex_lock` function attempts to lock the specified mutex. Each thread that attempts to lock a previously locked mutex shall block until the mutex is released by another thread. The `pthread_mutex_trylock` function also attempts to lock the specified mutex but returns an error if the mutex is locked. These functions should be used only when entering a critical section.

1.2 POSIX condition variables

We can use condition variables to delay a thread's execution until a specific condition is met. These variables can be used only after their corresponding mutex is locked by the same thread. Failing to do so results in undefined behaviour.

The POSIX standard provides a basic interface for interacting with condition variables:

```
#include <pthread.h>

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

The `pthread_cond_init` function is used to initialize a previously allocated condition variable structure. The `pthread_cond_wait` function is used to wait on a specified condition variable, which blocks the calling thread until it is released by another thread using `pthread_cond_signal` or `pthread_cond_broadcast`

The `mutex` argument points to a previously allocated mutex structure, the `cond` argument points to a previously allocated condition variable structure and the `attr` argument can be used to specify properties of the newly created condition variables (NULL is used to specify default properties).

Upon caling `pthread_cond_wait`, the calling thread is placed in a waiting queue associated with the condition variable and releases the mutex. Once the thread is released using `pthread_cond_{signal, broadcast}`, the mutex is reacquired before the thread resumes. The `pthread_cond_signal` function releases only one thread from the waiting queue, while `pthread_cond_broadcast` releases all blocked threads.

POSIX mutexes and condition variables can be combined to form a monitor.

1.3 Examples

As we previously briefly mentioned, a monitor conceptually consists of a set of internal data structures and procedures which operate on these internal structures. These procedures can not be executed in parallel. A thread enters the monitor, possibly tests some condition, and proceeds to execute the critical section. Upon completion, the thread releases the monitor and releases one thread waiting to enter the monitor. Should any of the conditions required by a thread not be met after entering the monitor, the thread blocks on a corresponding condition variable.

The following example shows how to realize monitors in the C programming language by combining POSIX mutexes and condition variables. The sample program attempts to acquire shared resources (`p` and `q`).

```
void acquire_resources (int p, int q)
{
        pthread_mutex_lock (&mtx);
        while ( p == 0 || q == 0 )
                pthread_cond_wait (&cond, &mtx);
        p = q = 0;
        pthread_mutex_unlock (&mtx);
  /* use acquired resources */
}

void release_resources (int p, int q)
{
        pthread_mutex_lock (&mtx);
        p = q = 1;  /* release resources */
        pthread_cond_broadcast (&cond);
        pthread_mutex_unlock (&mtx);
}
```

The following example shows a basic usage of monitors:

```
...
pthread_mutex_t m;
pthread_cond_t cond;
...
void *thread (void *p)
{       ...
        pthread_mutex_lock (&m);
        ...
        while ( blocking_condition )
                pthread_cond_wait (&cond, &m);
        ...
        pthread_mutex_unlock (&m);
        ...
        pthread_mutex_lock (&m);
        ...
        if ( release_condition )
                pthread_cond_signal (&cond); // or
                   pthread_cond_broadcast (&cond);
        ...
        pthread_mutex_unlock (&m);
        ...
}
int main ()
{       ...
        pthread_mutex_init (&m, NULL);
```

```
        pthread_cond_init (&cond, NULL);
        ...
        pthread_create (..., ..., thread, ...);
        ...
        pthread_join (...);
        ...
        return 0;
}
```

Detailed information about each the whole POSIX thread interface can be found in the following manpages: `pthread`, `pthread_create`, `pthread_exit`, `pthread_detach`, `pthread_join`, `pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_mutex_destroy`, `pthread_cond_init`, `pthread_cond_wait`, `pthread_cond_signal`, `sem_init`, `sem_wait`, `sem_post`, `sem_destroy`...