### 3.8 Semaphores and Locks in Pthreads

Mutex locks are part of the Pthreads (POSIX1.c) standard.

Semaphores are not a part of Pthreads, but are in POSIX1.b.

#### 3.8.1 Mutex

A Pthreads mutex is a lock with behavior similar to that of a Win32 CRITICAL_SECTION.

Operations `pthread_mutex_lock()` and `pthread_mutex_unlock()` are analogous to `EnterCriticalSection()` and `LeaveCriticalSection()`, respectively:

- A thread that calls `pthread_mutex_lock()` on a mutex is granted access to the mutex if no other thread owns the mutex; otherwise the thread is blocked.
- A thread that calls `pthread_mutex_lock()` on a mutex and is granted access to the mutex becomes the owner of the mutex.
- A thread releases its ownership by calling `pthread_mutex_unlock()`. A thread calling `pthread_mutex_unlock()` must be the owner of the mutex.
- There is a conditional wait operation `pthread_mutex_trylock()` that will never block the calling thread.
  - If the mutex is currently locked, then the operation returns immediately with the error code EBUSY.
  - Otherwise, the calling thread becomes the owner.

Listing 3.27 shows how to use a Pthreads mutex.

- You initialize a mutex by calling the `pthread_mutex_init()` function. The first parameter is the address of the mutex. If you need to initialize a mutex with non-default attributes, the second parameter can specify the address of an attribute object.
- When the mutex is no longer needed, it is destroyed by calling `pthread_mutex_destroy()`.

```c
#include <pthread.h>

pthread_mutex_t mutex;

void* Thread1(void* arg) {
    pthread_mutex_lock(&mutex);
    /* critical section */
    pthread_mutex_unlock(&mutex);
    return NULL;
}

void* Thread2(void* arg) {
    pthread_mutex_lock(&mutex);
    /* critical section */
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main() {
    int status; // error code
    pthread_attr_t threadAttribute; // thread attribute
    // initialize mutex
    status = pthread_mutex_init(&mutex,NULL);
    if (status != 0) { /* See Listing 1.4 for error handling */ }
    // initialize the thread attribute object
    status = pthread_attr_init(&threadAttribute);
    if (status != 0) { /* … */}
    // set the scheduling scope attribute
    status = pthread_attr_setscope(&threadAttribute,
                                  PTHREAD_SCOPE_SYSTEM);
    if (status != 0) { /* … */}
    // Create two threads and store their IDs in array threadArray
    status = pthread_create(&threadArray[0], &threadAttribute, Thread1, (void*) 1L);
    if (status != 0) { /* … */}
    status = pthread_create(&threadArray[1], &threadAttribute, Thread2, (void*) 2L);
    if (status != 0) { /* … */}
    status = pthread_attr_destroy(&threadAttribute); // destroy attribute object
    if (status != 0) { /* … */}
    // Wait for threads to finish
    status = pthread_join(threadArray[0],NULL);
    if (status != 0) { /* … */}
    status = pthread_join(threadArray[1],NULL);
    if (status != 0) { /* … */}
    // Destroy mutex
    status = pthread_mutex_destroy(&mutex);
    if (status != 0) { /* … */}
}
```

Listing 3.27 Using Pthreads mutex objects.
When you declare a static mutex with default attributes, you can use the PTHREAD_MUTEX_INITIALIZER macro instead of calling pthread_mutex_init().

In Listing 3.27, we could have written:
```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

You do not need to destroy a mutex that was initialized using the PTHREAD_MUTEX_INITIALIZER macro.

By default, a Pthreads mutex is not recursive, which means that a thread should not try to lock a mutex that it already owns. However, the POSIX 1003.1 2001 standard allows a mutex's type attribute to be set to recursive:

```
pthread_mutex_t mutex;
pthread_mutexattr_t mutexAttribute;
int status = pthread_mutexattr_init (&mutexAttribute);
if (status !=0) { /* … */ }
status = pthread_mutexattr_settype(&mutexAttribute, PTHREAD_MUTEX_RECURSIVE);
if (status != 0) { /* … */ }
status = pthread_mutex_init(&mutex,&mutexAttribute);
if (status != 0) { /* … */ }
```

If a thread that owns a recursive mutex tries to lock the mutex again, the thread is immediately granted access. An owning thread must release a recursive mutex the same number of times that it requested ownership before another thread can become the owner.

3.8.2 Semaphore

POSIX semaphores are counting semaphores. Operations sem_wait() and sem_post() are equivalent to P() and V(), respectively. POSIX semaphores have the following properties:
- A semaphore is not considered to be owned by a thread – one thread can execute sem_wait() on a semaphore and another thread can execute sem_post().
- When a semaphore is created, the initial value of the semaphore is specified, where 0 ≤ initial value ≤ SEM_VALUE_MAX.
- Semaphore operations follow a different convention for reporting errors. They return 0 for success. On failure, they return a value of -1 and store the appropriate error number into errno. The C function perror(const char* string) can be used to transcribe the value of errno into a string and print that string to stderr.
- The conditional wait operation sem_trywait(sem_t* sem) never blocks the calling thread.
  - If the semaphore value is greater than 0, then the value is decremented and the operation returns immediately.
  - Otherwise, the operation returns immediately with the error code EAGAIN indicating that the semaphore value was not greater than 0.

Listing 3.28 shows how Pthreads semaphore objects are used.
- Header file <semaphore.h> must be included to use the semaphore operations.
- Semaphores are of the type sem_t.
- A semaphore is created by calling the sem_init() function.
  - The first argument is the address of the semaphore.
  - If the second argument is non-zero, the semaphore can be shared between processes. Otherwise, it can be shared only between threads in the same process.
  - The third argument is the initial value.
- When the semaphore is no longer needed, it is destroyed by calling sem_destroy().
```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

sem_t s;

void* Thread1(void* arg) {
    int status;
    status = sem_wait(&s);
    if (status != 0) {
        std::cout << __FILE__ << ":" << __LINE__ << " - " << flush;
        perror("sem_wait failed"); exit(status);
    }
    /* critical section */
    status = sem_post(&s);
    if (status != 0) {
        std::cout << __FILE__ << ":" << __LINE__ << " - " << flush;
        perror("sem_post failed"); exit(status);
    }
    return NULL; // implicit call to pthread_exit(NULL);
}

void* Thread2(void* arg) {
    int status;
    status = sem_wait(&s);
    if (status != 0) {
        std::cout << __FILE__ << ":" << __LINE__ << " - " << flush;
        perror("sem_wait failed"); exit(status);
    }
    /* critical section */
    status = sem_post(&s);
    if (status != 0) {
        std::cout << __FILE__ << ":" << __LINE__ << " - " << flush;
        perror("sem_post failed"); exit(status);
    }
    return NULL; // implicit call to pthread_exit(NULL);
}

int main() {
    pthread_t threadArray[2]; // array of thread IDs
    int status; // error code
    pthread_attr_t threadAttribute; // thread attribute

    // initialize semaphore s
    status = sem_init(&s,0,1);
    if (status != 0) {
        std::cout << __FILE__ << ":" << __LINE__ << " - " << flush;
        perror("sem_init failed"); exit(status);
    }
    // initialize the thread attribute object
    status = pthread_attr_init(&threadAttribute);
    if (status != 0) { /* see Listing 1.4 for Pthreads error handling */}
    // set the scheduling scope attribute
    status = pthread_attr_setscope(&threadAttribute, PTHREAD_SCOPE_SYSTEM);
    if (status != 0) { /* … */}
    // Create two threads and store their IDs in array threadArray
    status = pthread_create(&threadArray[0], &threadAttribute, Thread1, (void*) 1L);
    if (status != 0) { /* … */}
    status = pthread_create(&threadArray[1], &threadAttribute, Thread2, (void*) 2L);
    if (status != 0) { /* … */}
    status = pthread_attr_destroy(&threadAttribute); // destroy the attribute object
    if (status != 0) { /* … */}
    // Wait for threads to finish
    status = pthread_join(threadArray[0], NULL);
    if (status != 0) { /* … */}
    status = pthread_join(threadArray[1], NULL);
    if (status != 0) { /* … */}
    // Destroy semaphore s
    status = sem_destroy(&s);
    if (status != 0) {
        std::cout << __FILE__ << ":" << __LINE__ << " - " << flush;
        perror("sem_destroy failed"); exit(status);
    }
    return 0;
}
```

Listing 3.28 Using POSIX semaphore objects.
Listing 3.29 shows wrapper class \textit{POSIXSemaphore}.

#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <iostream>
const int maxDefault = 999;
class POSIXSemaphore {
private:
    sem_t s;
    int permits;
public:
    void P();
    void V();
    POSIXSemaphore(int initial);
    ~POSIXSemaphore();
};
POSIXSemaphore::POSIXSemaphore (int initial) : permits(initial) {
    // assume semaphore is accessed by the threads in a single process
    int status = sem_init(&s, 0, initial);
    if (status !=0) {
        std::cout << __FILE__ << ":" << __LINE__ << ":" << flush;
        perror("sem_init failed"); exit(status);
    }
}
POSIXSemaphore::~POSIXSemaphore () {
    int status = sem_destroy(&s);
    if (status !=0) {
        std::cout << __FILE__ << ":" << __LINE__ << ":" << flush;
        perror("sem_destroy failed"); exit(status);
    }
}
void POSIXSemaphore::P() {
    int status = sem_wait(&s);
    if (status !=0) {
        std::cout << __FILE__ << ":" << __LINE__ << ":" << flush;
        perror("sem_wait failed"); exit(status);
    }
}
void POSIXSemaphore::V() {
    int status = sem_post(&s);
    if (status !=0) {
        std::cout << __FILE__ << ":" << __LINE__ << ":" << flush;
        perror("sem_post failed"); exit(status);
    }
}
Listing 3.29 Class \textit{POSIXSemaphore}. 