**PTHREADS. Synchronization mechanisms**

**1. Introduction**

**Aim**

The aim of this chapter is to present the synchronization mechanisms offered by the PTHREADS library and to show how these mechanisms can be used.

**Objectives**

This chapter has the following objectives:

1. knowing how to create and use the synchronization mechanisms offered by the PTHREADS library;
2. understanding the way and the situations in which each synchronization mechanism can be used;

**General presentation**

The chapter presents the synchronization mechanisms found in the PTHREADS library. These mechanisms can be used to synchronize the execution of threads belonging to a process, taking into account the fact that the threads access in a concurrent way the common resources.

The synchronization mechanisms presented are: *the mutexes*, *the condition variables* and the mechanism which ensures that some functions are *executed only once*. Each process can create as global variables its own synchronization mechanisms from the ones enumerated above. Because they are declared as global variables, these synchronization mechanisms are visible in all the threads belonging to the process, but they cannot be accessed by the threads of another process; that is why they cannot be used to synchronize the execution of processes.

The mutexes are used to ensure mutual exclusion access to a certain resource of some threads. The condition variables are a specialized mechanism, which waits for a certain event inside the mutual exclusion region. The mechanism of executing some functions only once is used in the case of initialization operations, because these operations have to be executed one time. This chapter presents the functions used to create and use these synchronization mechanisms.

**2. Mutexes**

The mutex variables from the package PTHREADS are of *phtread\_mutex\_t* type. They are used to ensure mutual exclusion access of a resource. Before it can be used, a mutex has to be initialized implicitly or explicitly.

**Initializing and deleting the mutex**

When initializing a mutex, the values of the attributes which describe the functionality of the mutex are established. The access of the attributes of a thread is done by using a structure of *pthread\_mutex\_attr\_t* type. The values stored by the attributes of a thread can be automatically set (implicit values) or can be indicated by the user (explicit values).

The implicit initialization of a mutex is done by using the predefined constant PTHREAD\_MUTEX\_INITIALIZER. Its effect is the creating of a mutex which has implicit values stored by its attributes. This initialization type is done in the following way:

#include <pthread.h>

pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;

The explicit initialization requires the use of the *pthread\_mutex\_init* function. The syntax of this function is the following:

#include <pthread.h>

int pthread\_mutex\_init (

pthread\_mutex\_t \*mutex,

const pthread\_mutexattr\_t \*attr);

The first parameter is set by the system and plays the role of the mutex’s identifier. The second parameter represents the address of a structure which stores the attributes of the mutex. The way in which such a structure is created and initialized is presented below. If the value of the second parameter is equal to NULL, the mutex will have implicit values as its attributes.

When a mutex is not needed anymore, it can be deleted by using the function *pthread\_mutex\_destroy*.

The syntax of the function *pthread\_mutex\_destroy* is the following:

#include <pthread.h>

int pthread\_mutex\_destroy (pthread\_mutex\_t \*mutex);

**The attributes of the mutex**

In order to create a structure for storing the attributes of a mutex, we declare a variable of type *pthread\_mutexattr\_t*. Before the variable can be used, it has to be initialized; this is done by using the function *pthread\_mutexattr\_init*, which has the following syntax:

#include <pthread.h>

pthread\_mutexattr\_init (pthread\_mutexattr\_t \*attr);

Calling this function has as effect the initialization of the attributes from the structure stored at the address *attr* with the implicit values. It is important to notice that the function *pthread\_mutexattr\_init* does not allocate memory for the structure of attributes; the function only initializes the values of the attributes contained by that structure. In order to destroy the mutex’s structure of attributes, we use the following function:

#include <pthread.h>

int pthread\_mutexattr\_destroy (

pthread\_mutexattr\_t \*attr);

The package THREADS implemented by Linux offers the option of setting the value of a single attribute of a mutex. This attribute describes the effect of calling repeatedly the primitive which locks the mutex, by the thread which has already locked the mutex. The functions which set and retrieve the value of this attribute are the following:

#include <pthread.h>

int pthread\_mutexattr\_settype (

pthread\_mutexattr\_t \*attr,

int kind);

int pthread\_mutexattr\_gettype (

pthread\_mutexattr\_t \*attr,

int \*kind);

The possible values of an attribute of this type are:

* PTHREAD\_MUTEX\_FAST\_NP, case in which the thread which has already locked the mutex and which calls the locking function again is suspended for good;
* PTHREAD\_MUTEX\_RECURSIVE\_NP, case in which the thread which has already locked the mutex is not suspended, but in order to unlock the mutex, the unlocking function has to be called as many times as the locking function;
* PTHREAD\_MUTEX\_ERRORCHECK\_NP, case in which when calling the locking function, it is verified if the thread which makes the call is the same as the thread which has already locked the mutex; if this is the case, the function returns as a result an error. The error code is EDEADLK.

The suffix \_NP (*Non-Portable*) of the constants described above indicates that this attribute is specific to the standard PTHREADS implemented by Linux. This means that it is possible for the standard not to be found in other implementations and it is not recommended to be used by portable applications. The implicit value of the attribute which determined the type of a thread is PTHREAD\_MUTEX\_FAST\_NP.

The type of a mutex can also be established when declaring the mutex, by setting some predefined values in the following way:

pthread\_mutex\_t mutex =

PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t mutex =

PTHREAD\_RECURSIVE\_MUTEX\_INITIALIZER\_NP;

pthread\_mutex\_t mutex =

PTHREAD\_ERRORCHECK\_MUTEX\_INITIALIZER\_NP;

**Locking the mutex**

The operation which ensures the mutual exclusion access to a shared resource using a mutex is called locking the mutex. This operation terminates successfully for a single thread at a given moment. The other threads are put in a waiting state in a queue associated to the mutex; these threads stay there until the mutex is unlocked by the thread which locked it. The function which locks a mutex is called *pthread\_mutex\_lock* and has the following syntax:

#include <pthread.h>

int pthread\_mutex\_lock (pthread\_mutex\_t \*mutex);

Another function which can be used in order to try locking a mutex is *pthread\_mutex\_trylock*. The behavior of this function is similar to the one of the function *pthread\_mutex\_lock*; the only difference is that it does not suspend the calling thread if the mutex cannot be locked, case in which the function terminates by returning an error. The syntax of the function is:

#include <pthread.h>

int pthread\_mutex\_trylock (pthread\_mutex\_t \*mutex);

Using the function *pthread\_mutex\_trylock* seems to be trivial and useful, but this is not the case. First of all, the function does not respect the general rule of synchronization, which assumes locking the threads when they do not have access to a shared resource. If the thread can do something else while the resource is busy, it means that the thread does not really need that specific resource. In this case the following problem arises: why has not been created another thread to handle what the first thread could still do while the resource was busy; the newly created thread would not have been involved in the synchronization protocol. On the other hand, the function *pthread\_mutex\_trylock* is used when the thread verifies the fulfillment of some conditions periodically (the *spooling* technique). If the mutex is overused by some other threads, the following situation might occur: the thread which calls *pthread\_mutex\_trylock* does not succeed to lock the mutex. Additionally, the processor is overloaded by verifying periodically the fulfillment of the conditions.

It is useful to use the function *pthread\_mutex\_trylock* in the real-time programming situations (when the thread has to react rapidly to some events) and the situations in which we try to detect and avoid deadlocks by using hierarchically organized mutexes.

**Unlocking the mutex**

Unlocking the mutex is the operation opposed to the locking operation. These two operations are always used together to ensure that no threads wait forever for a certain mutex. The unlocking operation is done by calling the function *pthread\_mutex\_unlock*, which has the following syntax:

#include <pthread.h>

int pthread\_mutex\_unlock (pthread\_mutex\_t \*mutex);

**A practical example of using mutexes**

The following code sequence exemplifies the way in which the mutex are created and used in order to ensure mutual exclusion access of a variable.

/\* The function which creates a mutex \*/

void createMutex(pthread\_mutex\_t \*mutex)

{

pthread\_mutex\_init(mutex, NULL);

}

/\* The function which modifies a variable \*/

/\*Mutual exclusion access of the variable is ensured\*/

void increment(pthread \*mutex, int \*variable)

{

pthread\_mutex\_lock(mutex); // locking the mutex

(\*variable)++; // exclusive access

pthread\_mutex\_unlock(mutex); // unlocking the mutex

}

/\* The function which destroys a mutex \*/

void createMutex(pthread\_mutex\_t \*mutex)

{

pthread\_mutex\_destroy(mutex);

}

**3. Condition variables**

If the mutexes offer the possibility of synchronizing threads which control the access to the variables, the control variables allow synchronization based on the values of the data. Like their name suggests it, the condition variables represent a way of waiting for the fulfillment of a condition, which depends on the values of some of the common data. If that particular condition is not fulfilled, the thread enters in a waiting queue associated to the variable; the thread can be awoke and retrieved from the queue, only by the thread which notices the fulfillment of the condition.

The condition variables are of type *pthread\_cond\_t* and need to be initialized before they can be used. As in the case of the mutexes, their initialization can be done implicitly or explicitly.

**Initializing and deleting condition variables**

The implicit initialization assumes declaring the variable which take the value of the predefined constant PTHREAD\_COND\_INITIALIZER:

#include <pthread.h>

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

The explicit initialization is done with the help of the function *pthread\_cond\_init*, which has the following syntax:

#include <pthread.h>

int pthread\_cond\_init (

pthread\_cond\_t \*cond,

const pthread\_condattr\_t \*attr);

The paramter *cond* is set by the system and represents the identifier of the condition variable. The parameter *attr* represents the address of a structure, which contains the attributes of the condition variable.

If we desire to delete a condition variable we can use the following function:

#include <pthread.h>

int pthread\_cond\_destroy (pthread\_cond\_t \*cond);

**The attributes of the condition variables**

A structure which contains attributes of a condition variable has the type *pthread\_condattr\_t*. The initialization function of such a structure is:

#include <pthread.h>

int pthread\_condattr\_init (pthread\_condattr\_t \*attr);

Destroying a structure of attributes belonging to a condition variable is done by using the following function:

#include <pthread.h>

int pthread\_condattr\_destroy (

pthread\_condattr\_t \*attr);

The Linux implementation of the standard PTHREADS does not offer the possibility of setting the value of an attribute.

**The waiting primitive**

A thread enters a waiting state after fulfilling a condition only by calling one of the functions: *pthread\_cond\_wait* or *pthread\_cond\_timedwait*. The thread exits the waiting state when the function *pthread\_cond\_signal* is called by another thread, which notices the fulfillment of the condition. The two waiting functions have the following syntax:

#include <pthread.h>

int pthread\_cond\_wait (

pthread\_cond\_t \*cond,

pthread\_mutex\_t \*mutex);

int pthread\_cond\_timedwait (

pthread\_cond\_t \*cond,

pthread\_mutex\_t \*mutex,

const struct timespec \*abstime);

In both of the cases, the thread is blocked and put into a waiting queue. The difference between the two functions consists in the way in which the first one blocks the thread for a indefinite period of time, until the call of the function which signals the fulfillment of the condition; on the other hand, the second function denotes a time interval after which the thread is automatically taken out of the waiting queue and executed, even if the condition was not fulfilled.

We notice that both functions have as a parameter a mutex. This happens because condition variables are always used in a code region protected by a mutex. The explanation of this situation is the fact that verifying conditions which have condition variables associated to them, assumes concurrent access to data, which automatically imposes using mutexes for synchronization. When the waiting state is reached, the mutex has to be unlocked in order to ensure that another thread can enter in its own critical region, modify the data, verify the fulfillment of the condition and signal this to the blocked threads. This is the reason for which the address of the mutex blocked by the thread is transmitted when calling one of the waiting functions.

**The signaling primitive**

There are two such primitives: the function *pthread\_cond\_signal* (used to take out a thread from the waiting list) and the function *pthread\_cond\_broadcast* (used to take out all the threads from the waiting list of a condition variable). The syntax of the functions is the following:

#include <pthread.h>

int pthread\_cond\_signal (pthread\_cond\_t \*cond);

int pthread\_cond\_broadcast (pthread\_cond\_t \*cond);

**Example**

The code presented below is the implementation of the functions specific to the **producer-consumer** problem. The scenario of the problem assumes that more threads communicate through a memory area of type *circular buffer of messages*. Some threads, called producers, add messages on the buffer; the other threads, called consumers, read the messages found on the buffer, in the order in which the messages were added. We have to avoid the situations which bring the buffer in an inconsistent state or lead to flawed results; in order to do this, mutual exclusion access of the buffer is required - not only for the producer threads, but also for the consumer threads. In order to solve efficiently the situations in which the producer threads have to wait for space to be freed on the buffer or the situations in which the consumer threads have to wait for more messages to be added on the buffer, condition variables are used.

#define DIMBUFFER 100

int buffer[DIMBUFFER];

int msgNo = 0; // nr. of unread messages

int indexProd = 0; // index for adding messages

int indexCons = 0; // index for reading messages

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t empty = PTHREAD\_COND\_INITIALIZER;

pthread\_cond\_t full = PTHREAD\_COND\_INITIALIZER;

/\* The function called by the producers \*/

/\* Mutual exclusion access \*/

void produce(int item)

{

pthread\_mutex\_lock(&mutex); // lock mutex

while (msgNo == DIMBUFFER) // if buffer is full

pthread\_cond\_wait(&full, &mutex);

buffer[indexProd] = item; // add message

// update producer index

indexProd = (indexProd + 1) % DIMBUFFER;

// increment nr. of unread messages

msgNo++;

// wake up a consumer

pthread\_cond\_signal(&empty);

pthread\_mutex\_unlock(&mutex); // unlock mutex

}

/\* Function called by the consummers \*/

/\* Mutual exclusion access\*/

void consumer(int \*item)

{

pthread\_mutex\_lock(&mutex); // lock mutex

while (msgNo == 0) // if buffer is empty

pthread\_cond\_wait(&empty, &mutex);

\*item = buffer[indexProd]; // read message

// update consummer index

indexCons = (indexCons + 1) % DIMBUFFER;

// decrement nr. of unread messages

msgNo--;

// wake up a producer

pthread\_cond\_signal(&full);

pthread\_mutex\_unlock(&mutex); // unlock mutex

}

**4. The mechanism of executing a function only once.**

Many times, the applications which assume the co-existence of more threads need functions which are called in the initializing phase. Such a function can, for example, to open a file, initialize a variable or a mutex. Usually, these procedures should be executed only once. If all threads are identical from the functional point of view, calling such an initialization function occurs in all threads, but the functions should be executed only once.

The standard PTHREADS specifies a mechanism useful in situations such as the one described above, ensuring that the user can be positive that a certain function was executed only once, regardless of the number of different threads calling it. This mechanism will be called in the following paragraphs *pthread\_once*. The mechanism *pthread\_once* is a synchronization mechanism in the initialization phase of the threads.

If we call a function only by using the mechanism *pthread\_once*, we can make the following assumptions:

1. Regardless of the number of function calls from one or more threads, the function will be executed only once by the first thread which calls it.
2. No thread which calls the function by using the mechanism *pthread\_once* can overpass it (we do not return from the function) until its first call terminated successfully. This ensures that the thread cannot pass the initialization point, until the initialization is finished.

In order to call a function by using the mechanism *pthread\_once*, we first have to declare a variable of type *pthread\_once\_t*, which needs to be initialized with the value PTHREAD\_ONCE\_INIT.

#include <pthread.h>

pthread\_once\_t once\_block = PTHREAD\_ONCE\_INIT;

We can associate to this variable a single function, whose calls will be made only through the mechanism *pthread\_once*. The call of the function associated to the variable will be made by using the function *pthread\_once*, which has the following syntax:

#include <pthread.h>

int pthread\_once (

pthread\_once\_t \*once\_block,

void (\*init\_routine) (void));

The first parameter represents the variable of type *pthread\_once\_t* and the second parameter is the function associated to the variable. It is important to remember that a function called by using the mechanism *pthread\_once*, should not be called from anywhere else unless this mechanism is used; if this is not done, the imposed synchronization principle is no longer followed.

**6. Problems**

1.      Write the C programs to test the effect of the following operations:

a.       locking an non initialized mutex

b.      a thread which has already locked a mutex, tries to lock it again

c.       a thread unlocks a mutex blocked by another thread

Identify the possible types of errors which might appear in each case described above.

2.      Implement the producer/consumer problem with a circular buffer, using the functions described above. A random number of producer and consumer threads will be created; each thread will do a random number of steps. During the ongoing execution list the number of producer and consumer threads which are waiting.

3.      Implement the protocol for crossing a bridge undergoing repairs. The cars can cross the bridge in only one direction at one point. We assume that the bridge can sustain maximum N cars. When a car arrives in front of the bridge, the thread associated to the car will execute the following function:

void Car(int dir)

{

EnterBridge(dir);

CrossBridge(dir);

ExitBridge(dir);

}

Implement the functions listed above, in order to ensure the safety of the traffic.

4.      Test the behavior of a variable *pthread\_once\_t* when it is associated with two or more different functions.

5.      Write a C program which creates more threads, every thread initializing the global variable *var* with the value 1 and then increments it with 1. The main thread displays the final value of the variable. Test the effect of the program's execution not only when the mechanism *pthread\_once* is used to initialize the variable, but also when this mechanism is not used.

6.      Implement, using mutexes, a class which simulates the behavior of the mechanism *pthread\_once*.

7.      The dining philosophers problem. Assume that N philosophers wish to eat dinner in a room which contains a single table. On the table there are N spaghetti plates and N forks. In order to be able to eat, a philosopher needs two forks: his own and the left neighbor's fork. Each philosopher has a unique identification number and is represented by a thread which calls in an infinite loop two functions: *think(int idPhilosopher)* and *eat(int idPhilosopher)*. Write the program which generates the threads corresponding to the philosopher and the code of the two functions, such as the philosophers can eat dinner without any problems. The situations which have to be avoided are: deadlock and starving (when a philosopher waits to eat for an indefinite period of time).

8.      We assume that at a hairdresser's there exists only a hairdresser and N waiting chairs. A single client can be hair-cut at one moment and only maximum N clients can wait inside; the other clients have to wait outside, in the street. The hairdresser and the clients are represented by using threads. Write the functions executed by these two types of threads such as to avoid breaking any of the listed rules. The hairdresser should not stay idle if there are clients waiting to be served. We have to avoid the case in which a client waiting outside the hairdresser's is served before a client waiting inside the hairdresser's. The arrival order of the clients can also be taken into consideration.

9.      We assume that at one point in a process more threads are executed concurrently. Each thread represents either an H atom or an O2 atom. Write the code of the functions executed by the two types of threads, such as, whenever it is possible, two H atoms and one O2 atom should combine to form a water molecule. If the molecule cannot be formed, we should wait for the needed atoms. Each atom will enter the composition of a single water molecule. Assuming that each atom has a unique identifier, list in a file, the composition of every water molecule.

10.  On the bank of a river there are missionaries and cannibals. All of them desire to cross the river but they have only one boat. The boat is full if three persons are inside and the crossing of the river is done only with a full boat. There have to be more missionaries than cannibals in a boat in order to avoid the death of the missionaries. Write the code of the functions executed by the threads which play the role of missionaries and cannibals; these threads should synchronize their execution in order to cross the river. If there are enough persons for a full boat without breaking the rules, a crossing should be made without delay.