Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)

Universitat Politècnica de València

Lab session 5

Creation and Synchronization of POSIX threads

Content

1. Objetives

- To **acquire experience using the functions of the POSIX standard for thread creation and waiting**
- To work on a setup where concurrent operations will happen
- **To understand when the race condition problem appears**
- To work with solutions to the race condition problem based on **active waiting** and **event based** synchronization (semaphores and mutexes)

2. Threads creation

The code in Figure-1 is the basic skeleton of an operation implemented with threads.

```c
/**
* Sample program "Hello World" with pthreads.
* To compile type:
*   gcc hello.c -lpthread -o Hello
*/
#include <stdio.h>
#include <pthread.h>
#include <string.h>

void * Print (void * ptr) {
  char * men;
  men =(char*) ptr;
  // EXERCISE 1.b
  write (1, men, strlen (men));
}

int main() {

  pthread_attr_t attrib;
  pthread_t thread1, thread2;

  pthread_attr_init (& attrib);

  pthread_create (& thread1, & attrib, Print, "Hello");
  pthread_create (& thread2, & attrib, Print, "World \n");

  // EXERCISE 1.a
  pthread_join (thread1, NULL);
  pthread_join (thread2, NULL);

}
```

Figure-1: Basic skeleton of a thread based program.

Create a file "hello.c" that contains this code, compile it and run it from the command line.

```
$ gcc hello.c -lpthread -o hello
$ ./hello
```

As shown in Figure-1 code, the novelties introduced by threads management go hand in hand with the necessary functions to initialize and to finish them properly. We have only made use of the more basic ones.

➢ Types **`pthread_t`** and **`pthread_attr_t`** from the header file `pthread.h`.

```
#include <pthread.h >
pthread_t  th;
pthread_attr_t attr;
```

➢ **`pthread_attr_init`** is responsible for assigning default values to the elements of the thread attributes structure. WARNING! If the attributes are not initialized, the thread cannot be created.

```
#include <pthread.h >

int pthread_attr_init(pthread_attr_t *attr)
```

➢ **pthread_create** creates a thread.

```
#include <pthread.h >

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

Pthread_create parameters:

thread: It is the first parameter of this function, *thread,* will contain the ID of the thread
attr: The argument *attr* specifies attributes for the thread. Can take the NULL value, in which case indicates values by default: "*the created thread is joinable (not detached) and have default (non-real - time) scheduling policy*'.
start_routine: the behavior of the thread to be created is defined by the function that is passed as the third parameter *start_routine* and it receive as an argument the pointer *arg*.

Pthread_create () function return value:
Returns 0 if the function runs successfully. In case of error, the function returns a nonzero value.

➢ ***pthread_join*** suspends the thread that calls to it until the thread specified as a parameter ends. This behavior is necessary because when the main thread "ends" destroys the process and, therefore, requires the abrupt completion of all threads that have been created.

```
#include <pthread.h >

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

Pthread_join parameters:

thread: parameter that identifies the thread to wait for.
exit_status: contains the value that the finished thread communicates to the thread that invokes pthread_join (notice that is a pointer to pointer, because the parameter passed by reference is a pointer to void).

➢ ***pthread_exit*** allows a thread to end its execution. The last ending thread in a process sets the process to end. Parameter `exit_status` allows communicating a termination value to another thread waiting for its end, through pthread_join().

```
#include <pthread.h >

 int pthread_exit(void *exit_status);
```

Exercise 1: working with pthread_join and pthread_exit

Check the behavior of pthread_join () call making the following changes in "hello.c" program shown above.

Remove (or comment) pthread_join calls in the main thread.

- What happens? Why does it happen?

Replace pthread_join calls by a single pthread_exit(0) call, close to the program point marked as // Exercise 1.a

- Does the program complete it execution correctly? Why?

Remove (or comment) all pthread_join or pthread_exit calls (close to comment // Exercise 1.a) and put in that point a 1 second delay (using usleep(…))

```
#include <unistd.h>
  void usleep(unsigned long usec);  // usec in microseconds
```

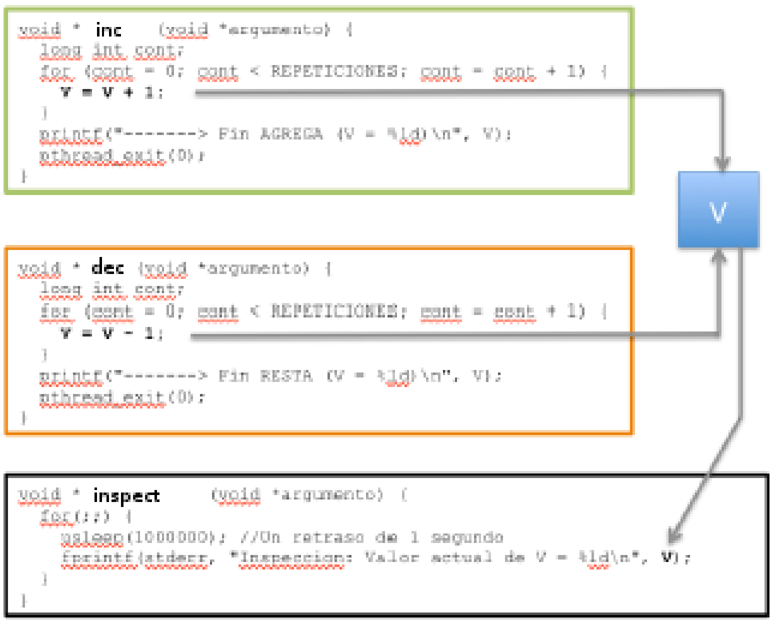
- What happens with the proposed changes?

Now put a 2 seconds delay close to comment `// Exercise 1.b`

- What happens now? Why?

3. Shared variables between threads

To see the issues related to using variables shared by several threads, you will use a simple problem. The problem is to access a variable that is shared by two threads, one "inc()" that increases the variable and another “dec()” one that decrements the variable. The shared variable has an initial value of 100, and has the same increases than decreases, so that at the end of the execution the variable should end up at 100. To be able to follow the values that take the shared variable, we use a third thread "inspect()" that will query the value of the variable and show on the screen the value got at intervals of one second.



```
void * inc    (void *argumento) {
  long int cont;
  for (cont = 0; cont < REPETICIONES; cont = cont + 1) {
    V = V + 1;
  }
  printf("-------> Fin AGREGA (V = %ld)\n", V);
  pthread_exit(0);
}
```

```
void * dec (void *argumento) {
  long int cont;
  for (cont = 0; cont < REPETICIONES; cont = cont + 1) {
    V = V - 1;
  }
  printf("-------> Fin RESTA (V = %ld)\n", V);
  pthread_exit(0);
}
```

```
void * inspect      (void *argumento) {
  for(;;) {
    usleep(1000000); //Un retraso de 1 segundo
    fprintf(stderr, "Inspeccion: Valor actual de V = %ld\n", V);
  }
}
```

Figura 1. Code of inc (increase), dec (decrease) and inspect functions

The "inspect()" thread accesses the variable V only for reading, therefore this thread may not cause race conditions. The "inc()" and "dec()" threads access the variable reading and writing it repeatedly, without pause. The increase operation V = V + 1, reads the variable, increases its value and writes the new value to memory. A race condition can happen if a decrement operation V = V-1 is interleaved during the increment operation (due to a context switch) or both threads run concurrently in different `cores.` Then variable V can take unexpected values. When both threads finish, the value of variable V is not the initial value (100 in our case) as it has to be.

The scenarios in which the race condition can occur vary according to the conditions of the experiment. If the computer that we use has a multi-core processor, you will see the race condition easily with values relatively low of "repetitions" constant and without changing the increment and decrement operations. Otherwise, if the processor has a single-core, it is less likely to cause a race condition. In this case will have to increase the number of repetitions and modify sections of increment and decrement, artificially increasing its machine instructions length using an auxiliary variable, to make it more likely that a change of context in the middle of critical operations.

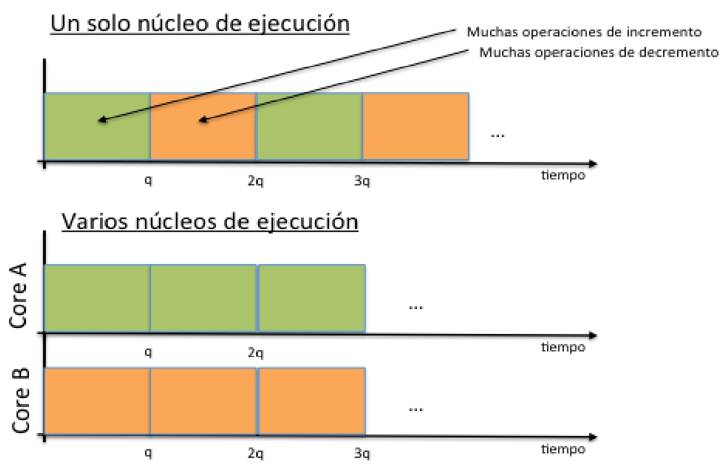

Un solo núcleo de ejecución
Muchas operaciones de incremento
Muchas operaciones de decremento
...
q
2q
3q
tiempo
Varios núcleos de ejecución
Core A
...
q
2q
tiempo
Core B
...
q
2q
3q
tiempo

Figure 2. inc() and dec() threads execution on one and two cores.

If it were required to make this change in the increment and decrement sections, it would be enough with the changes proposed in table 1. The local variable 'aux' should be declared in each thread.

	Original code	Replace with...

inc()	V=V+1;	aux=V;
		aux=aux+1;
		V=aux;

deca()	V=V-1	aux=V;
		aux=aux-1;
		V=aux;

Table 1. Increment and decrement with auxiliary variable.

4. Observing race conditions

Download the source code from FSO site in PoliformaT. The content of “RaceCond.c” is shown in annex-1 of this bulletin. To compile the C file, do:

```
$ gcc RaceCond.c -lpthread -o RaceCond
```

Exercise 2: Threads creation “RaceCond.c”

Complete the provided code in RaceCond.c to create three threads: a thread that executes the inc() function, another one with dec() function, and the last thread with the inspect() function, see Figure 1. Use calls "pthread_attr_init ()", "pthread_create ()" and "pthread_join ()".

Note that the inspect() thread consists of an infinite loop and if the function main() do "pthread_join ()" on it, the program will never end. Therefore, always pay attention and make sure you ONLY do "pthread_join ()" on inc() and dec() threads. We want the program to finish when the inc() and dec() threads finish.

Compile and run the deployed code. Observe the V value and determine whether there has been a race condition or not.

At first it could be expected that the concurrent access to variable V without any protection can cause a race condition in such a way that the final value of V be different from the initial one (100). However, for a low number of repetitions, maybe this does not happen and the final value of variable V is the initial one (100). This is because on a single-core processor system, there is not enough time such that both threads run concurrently with interleaved context switches. If the first thread is created, it starts to run, and finishes before the second thread starts, so both threads do not run concurrently. In multi-core processors, it is easier to see a race condition since the competition is real. If race condition does not happen just increase the number of repetitions.

Exercise 3: Causing race condition

Modify the RaceCond.c code by progressively increasing the values to the constant REPETITIONS to observe both situations, i.e. the situation where a race condition is not observed and the situation where it is observed. Note down for both cases the values of REPETITIONS in the following table.

REPETITIONS Race condition not observed	REPETITIONS Race condition observed

The **time** command shows the time it takes for a program to run:

```
        $ time ./RaceCond
```

It returns the real time (like if we measure it with a hand chronometer) and the CPU times (measured by the scheduler) executing instructions in user and kernel mode. With the help of the **time** command, find out the time of execution of the `RaceCond.c` program with race conditions, since the critical section is not protected. Write the displayed times:

RaceCond.c Critical section unprotected	
Actual time of execution	
Run time in user mode	
Run time in system mode	

Note: If the execution time is very short, generously increase the repetitions value until the actual program execution time is observable by a human (in the order of 200ms). This will give us a version of the program very appropriate to cause race conditions.

5. Solutions to avoid race condition

To avoid race conditions, it is mandatory to synchronize the access to critical sections of the code, in our case the decrease and increase operations. This synchronization must be such that while a thread is executing a critical section, the corresponding critical section of another thread cannot be performed simultaneously. This is called "mutual exclusion".

To achieve this, we circumvent the critical sections with some sections of code that implement the input and output protocols, as shown in figure 3.

```
void* inc (void * argument)
 {
   long int cont;
   long int aux;

  for (cont = 0;  cont < repetitions; cont = cont + 1 ) {
      Input protocol or input section
      V = V +1;
      Output protocol or output section
   }
   printf ('-----> inc end (V = % ld) \n ", V);
   pthread_exit (0 ));
 }
```

Figure 3. Input and output protocols to the critical section of inc()

The code that implements these protocols for input and output will depend on the method of synchronization that we use. In this lab session, we will study three synchronization methods:

- Synchronization using active waiting using "test_and_set" function.
- Synchronization with operating system support and suspension of the waiting process (event based). We will study the mechanisms offered by POSIX:
  - Semaphores
  - Mutexes.

Warning. In the annexes, there is a detailed description about the solutions to avoid race conditions used in the proposed activities. It is recommended to read carefully the annexes **before starting the activities**.

6. Protecting critical sections

From now on, will only work on the **version of the code where there is race condition (RaceCond.c)**.
In the following steps of the practice, we will modify the code to see that when we protect the critical section there is no race condition. We will also measure the execution times of the different versions to determine the cost in execution time that involves including mutual exclusion to access the critical section.

Exercise 4: Synchronization solution with "test_and_set"

Once verified that there are race conditions, modify the code to ensure the access to the shared variable V in mutual exclusion. Copy the file `RaceCond.c` on `RaceCondT.c.` On the `RaceCondT.c` code, do the following:

1. Identify the code section corresponding to the critical section then protect it with `test_and_set`, following the template shown in figure-3 and Table 3 in the Annex 2. Execute the program and verify that there are no race conditions.

2. Use command **time** and execute again the code to know the execution time with the critical section protected. Annotate the time values in the following table:

RaceCondT.c: critical section protected with test_and_set	
Actual execution time	
Run time in user mode	
Run time in system mode	

Exercise 5: Synchronization solution with semaphores

Copy RaceCond.c on RaceCondS.c and do modifications on the latter file.

1. Protect the critical section using a POSIX semaphore (sem_t) as described in table 4 of the Annex 3. Run the program and check if there are no race conditions.

2. Run the code again with time command to know what the execution time is and write the results in the following table.

RaceCondS.c protecting the critical section with semaphores sem_t	
Actual execution time	
Run time in user mode	
Run time in system mode	

Exercise 6: Synchronization solution with mutexes

Copy RaceCond.c on RaceCondM.c and do modifications on the latter file.

1. Protect the critical section writing the input and output sections using a pthreads mutex (pthread_mutex_t) as described in table 5 (Annex 4). Run the new program and check if there are not race conditions.

2. Using **time**, run the code to find out what is the execution time and write the results in the following table.

RaceCondM.c protecting the critical section with pthreads mutex	
Actual execution time	
Run time in user mode	
Run time in system mode	

Summarize the execution times in the following table

Short critical section	**Unprotected RaceCond.c**	**Test and Set RaceCondT.c**	**Semaphore RaceCondS.c**	**Mutex RaceCondM.c**
Actual execution time				
Run time in user mode				
Run time in system mode				

In the example developed in this lab session, what is most efficient: active waiting or event based synchronization?

In general, under what conditions is it better to use active waiting?

In general, under what conditions is it better to use event based synchronization?

Exercise 7: Extend the critical section

To check what happens when the critical section is large and how this influences the method of synchronization chosen, we can increase the duration of the critical section artificially similarly to as proposed in table 1, but introducing a delay before assigning the new value to the shared variable V, as described at table 2.

	Original code	Replace with...

inc()	V=V+1;	aux=V;
		aux=aux+1;
		usleep(500);
		V=aux;

dec()	V=V-1;	aux=V;
		aux=aux-1;
		usleep(500);
		V=aux;

Table 2: New critical section to work on

The small delay of half millisecond introduced in the code proposed in Table 2 increases noticeably the race condition probability; furthermore, it increases quite a bit the program execution time. In order to make the execution time easily observable in all cases you have to diminish the value of REPETITIONS.

1. Modify the code in RaceCond.c, RaceCondT.c, RaceCondS.c and RaceCondM.c as shown in table 2. Decrease the value of repetitions also in the four files so that the actual time of execution of the version that does not include synchronization is around a half of a second (a REPETITIONS value between 1000 and 10000 tends to be appropriate**, use the value 10000**). To make the results comparable, obviously, you must use the same value of repetitions in the four files.

2. Compile and run, with the command **time**, the four versions of the code and write the execution times in the following table.

Long critical section	**Unprotected RaceCond.c**	**Test and Set RaceCondT.c**	**Semaphore RaceCondS.c**	**Mutex RaceCondM.c**
Actual execution time				
Run time in user mode				
Run time in system mode				

3. From the obtained results, review the answers given to questions in exercise 6.

Annex

Annex 1: Support source code, “RaceCond.c”.

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

#define REPETITIONS 20000000   // CONSTANT

// GLOBAL SHARED VARIABLES
long int V = 100;      // Initial value

// AUXILIARY FUNTION
int test_and_set(int *spinlock) {
    int ret;
    __asm__ __volatile__(
    "xchg %0, %1"
    : "=r"(ret), "=m"(*spinlock)
    : "0"(1), "m"(*spinlock)
    : "memory");
    return ret;
}
// THREAD FUNCTIONS
void *inc (void *parameter) {
    long int cont, aux;
    for (cont = 0; cont < REPETITIONS; cont = cont + 1) {
        V = V + 1;
    }
    printf("-------> inc end (V = %ld)\n", V);
    pthread_exit(0);
}
void *dec (void *parameter) {
    long int cont,aux;
    for (cont = 0; cont < REPETITIONS; cont = cont + 1) {
        V = V - 1;
    }
    printf("-------> dec end (V = %ld)\n", V);
    pthread_exit(0);
}
void *inspec (void *parameter) {
    for (;;) {
        usleep(200000);
        fprintf(stderr, "Inspec: actual value of V = %ld\n", V);
    }
}
// MAIN FUNCTION
int main (void) {
    // Declaring the requiered variables
    pthread_t incThread, decThread, inspecThread;
    pthread_attr_t attr;

    // Default thread attributes
     pthread_attr_init(&attr);

    // EXERCISE: Create threads inc, dec and inspec with attr attributes
    // EXERCISE: The main thread has to wait inc and dec threads to end

    // Main program end
    fprintf(stderr, "-------> FINAL VALUE: V = %ld\n\n", V);
    exit(0);
}
```

Annex 2: Busy waiting synchronization with Test_and_set

The active waiting is a synchronization technique that **sets a global variable of Boolean type (*spinlock*) that indicates if the critical section is busy**. The semantics of this variable are: value of 0 indicates FALSE and means that the critical section is not busy; a value of 1 indicates TRUE and means that the critical section is busy.

The method is to implement a loop that continuously samples the value of the variable *spinlock* in the input section. The program will only pass to execute the critical section if it is free, but before entering, you must set the value of the variable to "busy" (value 1). To do this safely it is necessary to the operation of checking the value of the variable and assigning it to the value "1" be ATOMIC (uninterruptible) since otherwise it is possible that a change of context (or simultaneous execution on mulri-core computers) happen between the variable checking and its assignment, causing a race condition in the variable *spinlock* access.

For this reason, modern processors incorporate in its instruction set specific operations that allow you to verify and assign a value to a variable atomically. Specifically, in x86 processors there is the instruction "xchg" which swaps the value of two variables. As the operation consists in a single instruction, its atomicity is guaranteed. Using the statement "xchg", you can build a function "test_and_set" that makes atomically check and assignment operations discussed above. The code that implements this operation "test_and_set" is the one that is shown in Figure 5 and is included in the supporting code provided.

```
int test_and_set(int *spinlock) {
 int ret;
 __asm__ __volatile__(
 "xchg %0, %1"
 : "=r"(ret), "=m"(*spinlock)
 : "0"(1), "m"(*spinlock)
 : "memory");
 return ret;

 }
```

Figure 5. Test_and_set on Intel processors

Although the understanding of the code supplied for the function 'test_and_set' is not the objective of this practice, it is interesting to note how C language can include code written in assembly language.

With all this, to ensure mutual exclusion in accessing the critical section using this method, you should modify the code as shown in the following table.

// Declare a global variable, the "spinlock" that all threads will use `int key = 0;     // initialy FALSE → critical secction free`	
Input section	`while(test_and_set(&key));`
Output section	`key = 0;`

Table 3: Test and set based input and output protocols

Annex 3: Event based synchronization with Semaphores

Event based synchronization is achieved relaying on the operating system. When a thread has to wait to enter the critical section (because another thread is executing its critical section) it is "suspended" by eliminating it from the list of threads in the "ready" state by the scheduler. So waiting threads do not consume CPU time, rather than waiting in a polling loop such as busy waiting.

To allow programmers to use the passive standby, the operating system offers some specific objects that are called "semaphores" (type sem_t). A semaphore, illustrated in Figure 5, is composed of a counter, whose initial value can be set at the time of its creation, and a queue of suspended threads waiting to be reactivated. Initially the counter must be greater than or equal to zero and the suspended process queue is empty. Semaphores support two operations:

- Operation sem_wait() (operation P in Dijkstra's notation): this operation decrements the semaphore count and, if after the decrement the count is strictly less than zero, the thread that invoked the operation is suspended in the semaphore queue.
- Operationsem_post() (operation V in Dijkstra's notation): this operation increments the semaphore counter and, if after increasing the counter is less than or equal to zero, wakes up to the first thread suspended in the semaphore queue, applying FCFS policy.

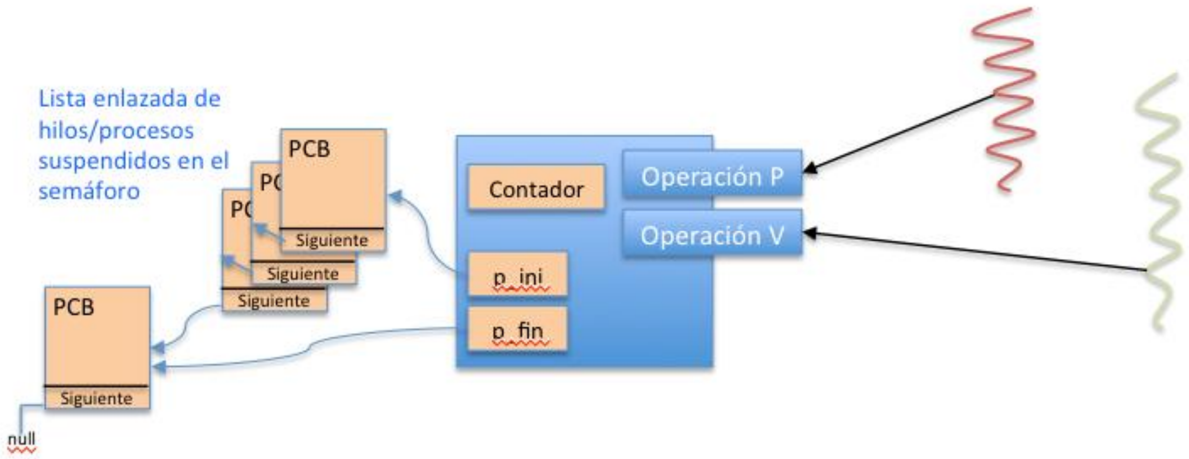


Figura 5: Semaphore structure and operations

Note: Although the POSIX semaphores (sem_t) are part of the standard, on Mac OSX they do not work. Be careful if you do testing on this operating system. Mac OSX provides other objects (semaphore_t) that behave similarly and can be used to provide the same interface that provide POSIX semaphores.

Depending on the use that you intend to give to a semaphore, we will define its initial value. The initial value of a semaphore can be greater than or equal to zero and its associated semantics is the "number of resources initially". Essentially a semaphore is a resource counter that may be requested (with operation sem_wait) and released (with the operation sem_post) in such a way that when there is no available resources, the threads that request resources are suspended waiting to some resource be released.

Especially relevant are the semaphores with initial value equal to one. As there is only a free resource initially, only one thread can execute the critical section in mutual exclusion with others. These semaphores are often called "mutex" and are those that interest us in this lab session.

As has been done with other synchronization methods, the use of the "pthreads mutex" is shown in the table below:

```
// Include the header of semaphore library
#include <semaphore.h>
// Declare a global variable, the semaphore that all threads will use
```

`sem_t sem;  // It is not initialized, only declared`	
Input section	`sem_wait(&sem);`
Output section	`sem_post(&sem);`

// In the main function "main()" the semaphore must be initialized
`sem_init(&sem,0,1); // The second parameter indicates that the traffic is not shared`
`                    // and the last parameter indicates the initial value,`
`                    // "1" in our case (mutual exclusion)`	

Table 4. Description of the critical section input and output protocols with semaphores

Annex 4: Event based synchronization with pthreads Mutexes

In addition to semaphores provided by the POSIX standard, the phtread library provides other synchronization objects: mutex and condition variable. The "mutex" object "pthread_mutex_t", is used to solve the problem of mutual exclusion as its name suggests and can be considered as a semaphore with initial value '1' and with maximum value '1'. Obviously they are created to ensure mutual exclusion and cannot be used as resource counters.

As has been done with other synchronization methods, the use of the "pthreads mutex" is shown in the table below:

// Include the header of pthreads library, it is already included when we use threads
`#include <pthread.h>`
// Declare a global variable, the "mutex" used by all threads
`pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // This declares and initializes`	

Input section	`pthread_mutex_lock(&mutex);`
Output section	`pthread_mutex_unlock(&mutex);`

Table 5 : Description of the critical section input and output protocols with mutexes