Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

fSO

Lab session 5

Creation and Synchronization of POSIX threads (v2.0)

Content

1. Objectives

- 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>
#include <unistd.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);

  return 0;

}
```

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 behaviour 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 behaviour 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?

Sometimes, "Hello" is printed, and other times "HelloWorld" is printed. This happens because the main thread doesn't wait for the child threads to finish before terminating the program, so some threads may not finish printing what they have to print.

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 its execution correctly? Why?

Both threads finish executing, as pthread_exit(0) run by the thread main ends that thread but doesn't terminate the process. The process will be terminated when every thread finishes, so both threads that print strings to the console will finish.

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(...) from <unistd.h> and whose definition is shown below)

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

- What happens with the proposed changes?

Both threads finish executing, as they have enough time to do so until the main thread reaches the end of main and terminates the program.

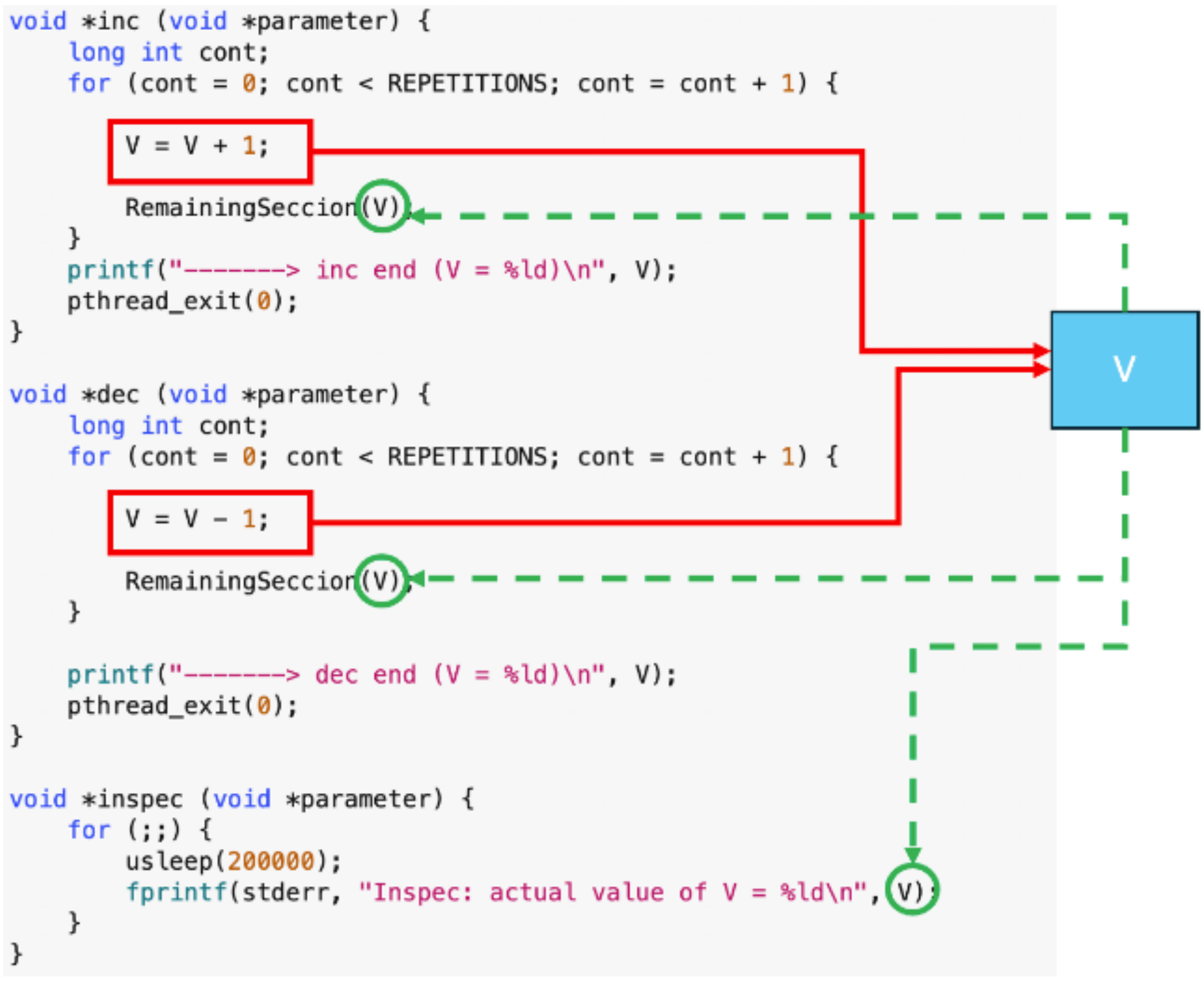
Now put a 2 seconds delay close to comment `// Exercise 1.b`

- What happens now? Why? (I put it right after the comment)

Nothing is printed, as the main thread reaches the end of the main function and terminates the program before the `usleep` in the threads lets them print their string.

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 *parameter) {
    long int cont;
    for (cont = 0; cont < REPETITIONS; cont = cont + 1) {

        V = V + 1;

        RemainingSeccion(V);
    }
    printf("-------> inc end (V = %ld)\n", V);
    pthread_exit(0);
}

void *dec (void *parameter) {
    long int cont;
    for (cont = 0; cont < REPETITIONS; cont = cont + 1) {

        V = V - 1;

        RemainingSeccion(V);
    }

    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);
    }
}
```

Figura 2. Code of inc (increase), dec (decrease) and inspect functions

The “inspec()” thread accesses **V** to read its value, but does not write to it, so it does not cause race conditions. The threads “inc()” and “dec()” access **V** by reading and modifying it repeatedly. In detail, in these two threads, it can be seen that there is a *for* loop that is executed REPETITIONS times, where in each iteration the variable **V** is modified and then a function RemainingSection(V) is executed that works with that variable **V** but without modifying it. The increment operation, V=V+1, reads the variable, increments its value and writes the new value to memory. If during the increment operation the decrement operation V=V-1 is interleaved due to a context switch or concurrent execution of increment and decrement on different processor cores, it is possible that a race condition occurs and the variable **V** takes unexpected values. That is, at the end of both threads the value of **V** is not the initial one, 100 in our case.

The scenarios in which the race condition can occur vary according to the conditions of the experiment, as shown in Figure 3. 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, we will have to increase the number of REPETITIONS.

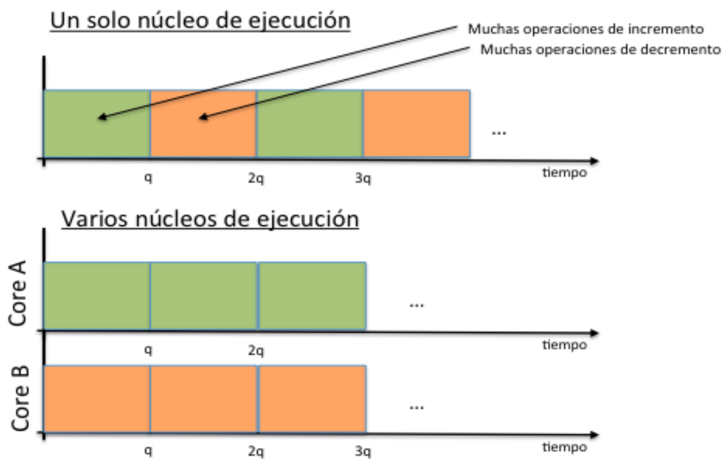

Un solo núcleo de ejecución
Muchas operaciones de incremento
Muchas operaciones de decremento
...
q
2q
3q
tiempo
Varios núcleos de ejecución
Core A
Core B

Figure 3. inc() and dec() threads execution on one and two cores.

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 2. 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.

The final value of V in my case is 11059, which means there has been a race condition, as the value should be 100.

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 the race condition does not happen just increase the number of repetitions.

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	12.116s
Run time in user mode	24.192s
Run time in system mode	0.004s

5. Sequential execution

To avoid the race condition we can force sequential execution of the add() and subtract() threads. This is not a solution to the race condition, it is simply a way to avoid the problem.

Exercise 3: Sequential execution “RaceCondHS.c”

Create a copy of the modified source in the previous section (“RaceCond.c”) and call it “RaceCondHS.c”.

The idea is to run 'inc’ thread first and when it finishes 'dec’ thread. Modify the main of “RaceCondHS.c” so that it first creates the Inspect thread, then the inc thread, waits for this thread, creates the dec thread and finally waits for this one.

Compile “RaceCondHS.c” and run the implemented code. Observe the value of **V** and justifiably determine whether a race condition has occurred or not.

In this version of the code, no race condition occurs, as the threads that modify the variable (and thus may cause a race condition if they run concurrently) run separately, first the incrementing thread and then the decrementing one.

As in the previous section, using the time command displays the time it takes to execute a program. Execute:

```
       $ time ./RaceCondHS
```

Write the displayed times:

RaceCondHS.c	**Sequential**
Actual time of execution	20.577s
Run time in user mode	20.575s
Run time in system mode	0.000s

Compared to the previously measured times (see Table “RaceCond.c”), you can see that the actual execution time has almost doubled. What is the reason?

Now, threads don't run concurrently as before, which means the time it takes to run the program almost doubles, as instead of running both functions "at the same time", they are run one after the other.

As can be seen, by sequentially executing the threads, the actual execution time has been increased, and we do not take advantage of the possible concurrency in the execution of the threads.

6. 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 4.

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

   for (cont = 0;  cont < REPETITIONS; cont = cont + 1 ) {
        Input protocol or input section
                  V = V +1;
        Output protocol or output section

        RemainingSection(V);

    }
    printf ('-----> inc end (V = % ld) \n ", V);
    pthread_exit (0 ));
 }
```

Figure 4. 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**.

7. 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-4 and Table 1 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	12.333s
Run time in user mode	24.604s
Run time in system mode	0.034s

3. Compare these times with those of “RaceCond.c”. Is there any significant difference? With respect to sequential execution (“RaceCondHS”), what have we achieved?

There is no significant time difference compared with the incorrect version of the program \`RaceCond.c\`, which means we have achieved the same results as in \`RaceCondHS\` but in around half the time.

4. Copy “RaceCondT.c” into “RaceCondTB.c”. Note in “RaceCondTB.c” what happens if you rewrite the input and output sections and place them in the locations shown in Figure-5.

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

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

Figure 5. New location of protection protocols.

5. Record the run time results of in the following table:

RaceCondTB.c: protecting for loop with test_and_set	
Actual execution time	22.462s
Run time in user mode	34.462s
Run time in system mode	0.004s

6. What has happened in this case and why has the execution time in user mode also increased?

In this case, the real time is around the same as in `RaceCondHS.c`, as we selected the entire for loop of the functions as the critical section, which means the increase and decrease of the variable will run separately. The execution time in user mode also increases because when one of the threads starts running the for loop (which takes around 10s), the other one is in execution but waiting for the first one to finish. That is why the difference between the actual time and user time is of around 10s.

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 3 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	12.331s
Run time in user mode	24.640s
Run time in system mode	0.010s

3. What happened in this case? Compare the results with those of test_and_set.

In this case, the final value of V is correct. The time it takes to run this specific program in my computer is more or less the same as what it takes to run the version with test_and_set.

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 4 (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	12.561s
Run time in user mode	24.946s
Run time in system mode	0.084s

3. Answer the following questions:

In the example developed in this lab session, what is most efficient: active waiting or event based synchronization? In this case, at least in my computer, all solutions take around the same time to execute.

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

If there are short wait times and the critical section is very short.

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

If many threads will try to access the critical section at the same time, if the critical section is long, and if wasting CPU resources is not wanted.

Annex

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

```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;
}

void RemainingSection(int V)
{
    int i;
    long tot;

    tot = 0;
    for (i = 0; i < 300; i++) {
       tot = tot+V;
    }
}

// THREAD FUNCTIONS
void *inc (void *parameter) {
    long int cont;
    for (cont = 0; cont < REPETITIONS; cont = cont + 1) {

        V = V + 1;

        RemainingSection(V);
    }
    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;

        RemainingSection(V);
    }
    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);
}
```

1.2. 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 6 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 6. 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;     // initially FALSE -> critical section free`	
Input section	`while(test_and_set(&key));`
Output section	`key = 0;`

Table 1: Test and set based input and output protocols

For ARM processors (for example, those used by the new Apple computers), there is no instruction identical to x86's xchg that atomically swaps values in a single operation. But you can use gcc spinlocks, which are implemented in the __sync_lock_test_and_set and __sync_lock_release functions. Unlike the previous implementation for the x86, in the output protocol you can NOT simply do key=0, and you have to make use of the release function (see code in Figure 7). In table 2, you can see how the input and output protocol looks like. These new functions are generic and can also be used in x86 processors.

```
int test_and_set(int *spinlock) {
    return (__sync_lock_test_and_set(spinlock, 1));
}

void release(int *spinlock) {
    __sync_lock_release(spinlock);
}
```

Figure 7. Instruction code test_and_set and reléase using spinlocks

// Declare a global variable, the “spinlock” that all threads will use `int key = 0;     // initially FALSE -> critical section free`	
Input section	`while(test_and_set(&key));`
Output section	`release(key);`

Table 2: Test and set and release based input and output protocols

1.3. Annex 3: Event based synchronization with Semaphores

Event based synchronization is achieved relaying on the operating system. When a thread must 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 8, 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.

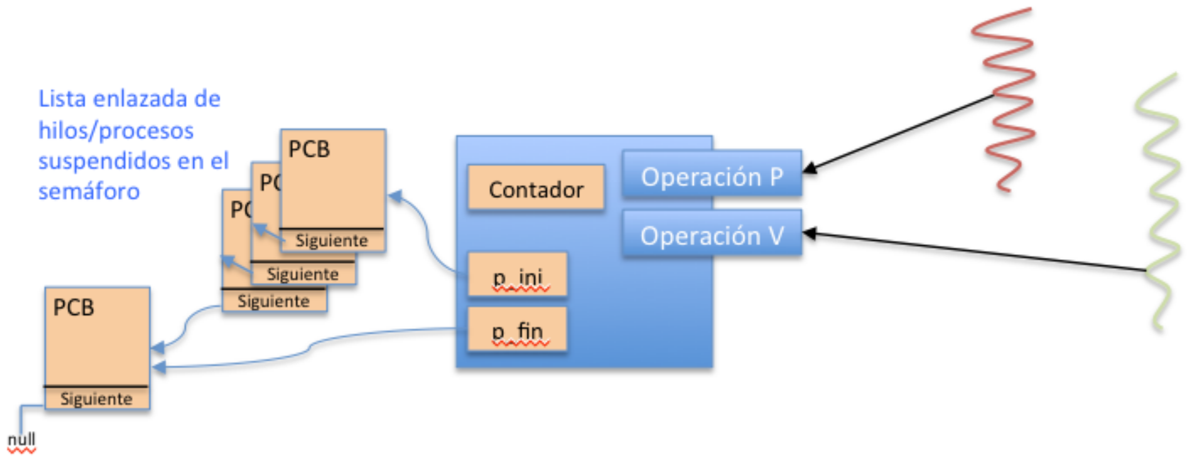


Figura 8: Semaphore structure and operations

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

1.4. **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 variables. 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 4 : Description of the critical section input and output protocols with mutexes