Fonaments dels Sistemes Operatius

Departament de Informàtica de Sistemes i Computadores (DISCA)
Universitat Politècnica de València

fSO

Pràctica 5

Creació i Sincronització de Fils POSIX (v1.0)

Contingut

1. Objectius

- **Adquirir experiència en el maneig de funcions estàndard POSIX per a la creació i espera de fils**.
- Treballar amb un escenari on es produeixin operacions concurrents.
- **Comprendre quan es produeixen condicions de carrera**, així com els mecanismes més bàsics per a evitar aquest problema.
- Treballar solucions al problema de condició de carrera amb **espera activa** i **espera passiva**.

2. Creació de fils

El codi de la figura-1 constitueix l'esquelet bàsic d'una funció que utilitza fils en la seua implementació.

```
/**
* Programa d'exemple "Hola mon" amb pthreads.
* Per a compilar tecleja:  gcc hola.c -lpthread -o hola
**/
#include <stdio.h>
#include <pthread.h>
#include <string.h>

void *Imprime( void *ptr )
 {
   char *men;
   men=(char*)ptr;

   //EXERCICI1.b
   write(1,men,strlen(men));
 }
 int main()
 {

  pthread_attr_t atrib;
  pthread_t fil1, fil2;

  pthread_attr_init( &atrib );

  pthread_create( &fil1, &atrib, Imprime, "Hola \n");
  pthread_create( &fil2, &atrib, Imprime, "mon \n");

  //EXERCICI1.a
  pthread_join( fil1, NULL);
  pthread_join( fil2, NULL);
 }
```

Figura-1: Esquelet bàsic d'un programa amb fils POSIX.

Creeu un arxiu "hola.c" que continga aquest codi, compileu-lo i executeu-lo des de la línia d'ordres.

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

Com s'observa en el codi de la figura-1, les novetats que introdueixen el maneig de fils van de la ma de les funcions necessàries per a inicialitzar-los, de les quals només hem fet ús de les més bàsiques o imprescindibles.

- Tipus **`pthread_t`** i **`pthread_attr_t`** que s'accedeixen des de l'arxiu de capçalera `pthread.h`.

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

- Funció **`pthread_attr_init`** encarregada d'assignar uns valors per defecte als elements de l'estructura d'atributs d'un fil. **¡AVIS!** Si no s'inicialitzen els atributs, el fil no es pot crear

```
#include <pthread.h >

 int pthread_attr_init(pthread_attr_t *attr)
```

- Funció **pthread_create** encarregada de crear un fil.

```
#include <pthread.h >

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

Paràmetres de pthread_create:

thread: Contindrà l'identificador del fil
attr: Especifica els atributs del fil. Amb valor NULL, reben valors per defecte: "*the created thread is joinable (not detached) and has default (non real-time) scheduling policy*".
start_routine: Funció que defineix el comportament del fil que s'està creant.
arg: Argument que es passa a la funció del fil (*start_routine*) i l'us de la qual dependrà de cada funció.

Valor de retorn de la funció pthread_create():
Retorna 0, si la funció s' executa amb èxit. En caso d'error, la funció retorna un valor distint de zero.

- Funció ***pthread_join***. El seu efecte es suspendre al fil que la invoca fins que el fil que se li especifica com a paràmetre termine. Aquest comportament és necessari ja que quan el fil principal "acaba" destrueix el procés i, per tant, obliga a l'acabament de tots els fils que s'hagen creat.

Paràmetres de pthread_join:

thread: Paràmetre que identifica al fil a esperar.
exit_status: conté el valor que el fil acabat comunica al fil que invoca a pthread_join

```
#include <pthread.h >

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

- Funció ***pthread_exit*** Permet a un fil acabar voluntàriament la seua execució. La finalització de l'últim fil d'un procés finalitza al procés. Mitjançant el paràmetre `exit_status` pot comunicar un valor d'acabament a un altre fil que estiguera esperant la seua finalització.

```
#include <pthread.h >

 int pthread_exit(void *exit_status);
```

Exercici 1: Treballant amb pthread_join i pthread_exit

Comproveu el comportament de la crida pthread_join() realitzant les següents modificacions en el codi del programa "hola.c" mostrat anteriorment.

Qüestions Exercici 1:

Elimineu (o comenteu) les crides pthread_join del fil principal.

- ¿Què ocorre? ¿Per què?

Substituïu les crides pthread_join per una crida pthread_exit(0), prop del punt del programa marcat com a //EXERCICI1.a)

- ¿Completa ara correctament el programa la seua execució?¿Per què?

Elimineu (o comenteu) qualsevol crida a pthread_join o pthread_exit (prop del comentari //EXERCICI1.a) e introdueixi en eixe mateix punt un retràs d'1 segon (**utilitzant la funció usleep(...) de la llibreria <unistd.d> i la definició del qual es mostra a continuació**")

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

- ¿Què passa rere la realització de les modificacions proposades?

Introduïu ara un retràs de 2 segons prop del comentari `//EXERCICI1.b`

- ¿Què passa ara? ¿Per què?

3. Variables compartides entre fils

Per a observar la problemàtica d'utilitzar variables compartides, proposem un problema senzill en el qual dos fils requereixen accedir a una variable compartida **V**. Un fil "agrega()" que incrementa la variable i altre "resta()" que decrementa la variable. El valor inicial de **V** és de 100, i se realitzen els mateixos increments que decrements, per tant, rere l'execució la variable **V** hauria de valdre 100. Per a visualitzar els valors de la variable **V**, s'utilitza un tercer fil "inspecciona()" que consulta el valor de **V** i el mostra per pantalla a intervals d'un segon.

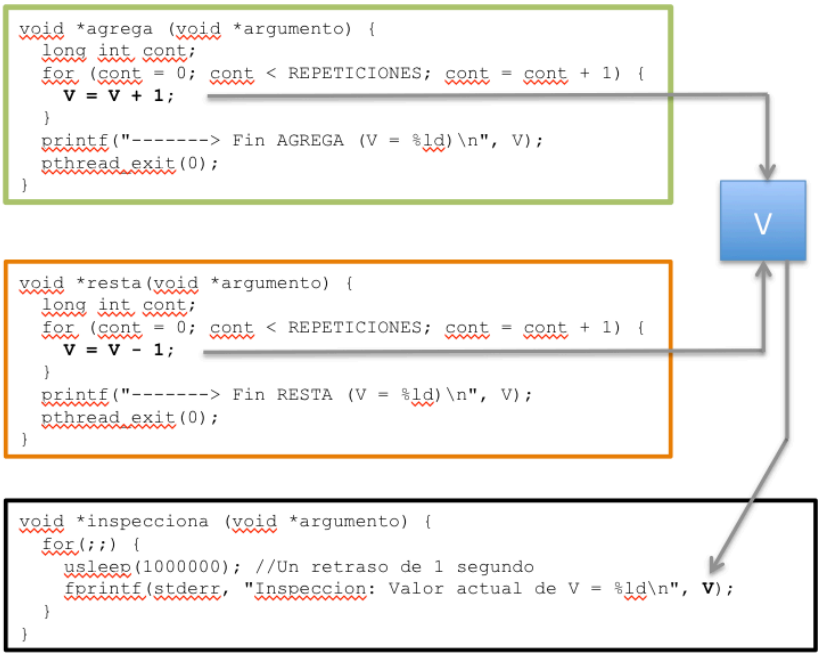

Figura-2: Codi de les funcions agrega, resta i inspecciona

El fil "inspecciona()" accedeix a V per a llegir el seu valor, però no escriu sobre ella, per tant, no provoca condicions de carrera. Els fils "agrega()" i "resta()" accedeixen a V llegint-la i modificant-la repetidament. L'operació increment, V=V+1, llig la variable, incrementa el seu valor i escriu en memòria el nou valor. Si durant l'operació d'increment s'intercala la de decrement V=V-1 degut a un canvi de context o a l'execució concurrent d'increment i decrement en nuclis diferents del processador, és possible que es produeixi una condició de carrera i la variable V prenga valors inesperats. Es a dir, que al finalitzar ambdós fils el valor de V no siga l'inicial, 100 en el nostre cas.

Els escenaris en els quals es pot produir la condició de carrera varia segons les característiques de la màquina on es treballa, com mostra la figura-3. Per exemple, en un processador amb múltiples nuclis d'execució (*multi-core)*, es podrà observar la condició de carrera fàcilment amb valors relativament baixos de la constant "REPETICIONS" (figura-2). Si el computador te un sol nucli d'execució, és menys probable que es produeixi una condició de carrera. En aquets cas caldrà que augmentar el nombre de REPETICIONS i modificar les seccions d'increment i decrement, amb una variable auxiliar durant les operacions, per a augmentar la probabilitat de que es produeixi un canvi de context en mig de l'operació.

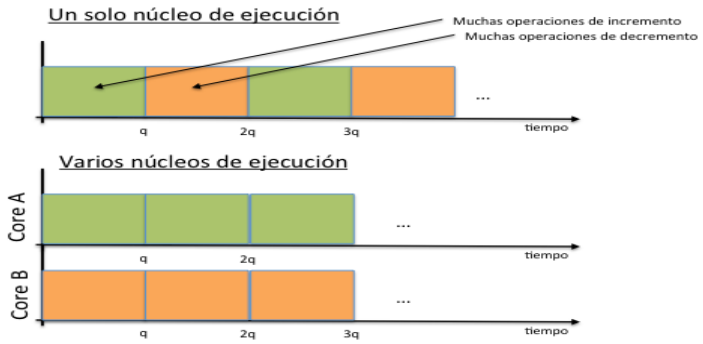

Figura-3: Execució dels fils agrega i resta en CPU, en màquines amb un i dos nuclis.

Quan siga necessari augmentar el temps de còmput de les seccions increment i decrement, introdueixi els canvis de la taula 1. Ha de declarar la variable local "aux" en cada fil del tipus "long int".

	Codi original	Substituir per…..

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

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

Taula 1. Escenari d'increment i decrement amb variable auxiliar.

4. Observant condicions de carrera

Descarregueu el material d'aquesta pràctica del PoliformaT de la assignatura, on trobareu un arxiu C que conté el programa que apareix en l'annex-1 d'aquest butlletí. Per a compilar-lo teclegeu:

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

Exercici 2: Creació de fils "CondCarr.c"

Completeu el codi proporcionat en "CondCarr.c" de forma que se creen tres fils: un executarà la funció *agrega()*, altre la funció *resta()*, i l'últim fil executarà la funció *inspecciona()***,** veure figura 2. Utilitzeu les crides *"pthread_attr_init()"*, *"pthread_create()"*, i *"pthread_join()*".

Observeu que el fil *inspecciona()* consisteix en un bucle infinit i si en la funció *main()* fem "pthread_join()" sobre ell, el programa mai acabarà. Per tant, presteu especial atenció i assegureu-vos de fer "pthread_join()" **només per als fils *agrega()* i *resta()*** ja que el programa ha d'acabar quan els fils *agrega()* i *resta()* acaben.
Compileu i executeu el codi implementat. Observeu el valor de **V i** determineu de forma justificada si s'ha produït una condició de carrera o no.

En principi es podria esperar que l'accés concurrent a la variable **V** sense cap tipus de protecció provoqués una condició de carrera de forma que el valor final de **V** fora distint del inicial (100). Tan mateix, per a valors baixos de REPETICONS, això pot no passar observant-se que el valor final de V és l'inicial (100). Això es degut a que, en sistemes amb un sol processador, no dona temps a que els dos fils s'executen concurrentment i hi haja canvis de context. Si es crea el primer fil, aquest comença a executar-se, i acaba abans de que comence a executar-se el segon fil, ambdós fils no arriben a executar-se concurrentment. En sistemes *multi-core* és més fàcil observar una condició de carrera ja que la concurrència és real.

Exercici 3: Provocant condicions de carrera

Modifiqueu el codi de CondCarr.c augmentant progressivament els valors a la constant REPETICIONS per a observar ambdues situacions, és a dir, la situació en la que no s'observa una condició de carrera i la situació en la qual si s'observa. Anoteu per a ambdós casos els valors de REPETICIONS en la següent taula.

REPETICIONS No s'observa condició de carrera	REPETICIONS Sí s'observa condició de carrera

L'ordre ***time*** mostra el temps que tarda en executar-se un programa. Executeu:

```
$ time ./CondCarr
```

time mostra el temps real (com si cronometréssim) i els temps de CPU (mesurats pel planificador) executant instruccions d'usuari i del sistema operatiu.

> **Nota**: Si estem treballant amb una màquina virual amb un únic processador, hem de tenir en compte que els temps que obtindrem amb l'ordre time seran coherents amb aquesta circumstància. Si els nostres processos treballen sobre un únic processador, encara que es generen diferents fils, no tindrem la concurrència que ens podrien oferir aquests fils treballant sobre diversos cores. El planificador de la CPU anirà assignant l'ús de la CPU als diferents fils, però no hi haurà concurrència en l'execució dels mateixos. I, per tant, els temps de CPU mesurats pel planificador seran molt similars a el temps real d'execució. No hi haurà una millora significativa en el temps real d'execució pel fet d'utilitzar fils. En qualsevol cas, serà possible configurar la màquina virtual amb més d'un processador, però només si la nostra màquina té recursos suficients.

Amb l'ajuda de l'ordre ***time*** esbrineu el temps d'execució del programa *CondCarr.c* amb condicions de carrera, ja que no s'ha protegit la secció crítica. Anoteu els temps mostrats

CondCarr.c Sense protegir la secció crítica	
Temps real d'execució	
Temps d'execució en mode usuari	
Temps d'execució en mode sistema	

> **Nota:** Si el temps d'execució es molt curt, augmente generosament el valor REPETICIONS fins que el temps real d'execució del programa siga observable per un humà (de l'ordre de 200ms). Això ens proporcionarà una versió del programa molt propicia per a que es produeixin condicions de carrera.

5. Soluciones per a evitar la condició de carrera

Per a evitar condicions de carrera, es necessari sincronitzar l'accés a les ***seccions crítiques del codi***, en el nostre caso les operacions de decrement i increment sobre **V**. Aquesta sincronització ha de garantir la **"exclusió mútua"**, acomplint-se que mentre un fil està executant una secció crítica altre fil no pot executar simultàniament la seua secció crítica. Per a aconseguir això, protegirem les seccions crítiques amb unes seccions de codi com el protocol d'entrada i eixida, tal como indica la Figura 4.

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

  for (cont = 0; cont < REPETICIONS; cont = cont + 1)
    {  Protocol d'Entrada o Secció d'Entrada
                  V = V + 1;
       Protocol d'Eixida o Secció d'Eixida
    }
     printf("-------> Fi AGREGA (V = %ld)\n", V);
     pthread_exit(0);
 }
```

Figura-4: Protocol d'entrada i protocol d'eixida a la secció critica d'*agrega()*.

El codi del protocol d'entrada i eixida dependrà del mètode de sincronització. En aquesta pràctica estudiarem tres mètodes de sincronització:

- Sincronització mitjançant espera activa utilitzant la funció "test_and_set".
- Sincronització amb espera passiva, estudiarem els dos mecanismes que ofereix POSIX:
  - Semàfors: variables de tipus "sem_t" en POSIX.
  - Objectes "mutex" de la biblioteca "pthreads".

¡AVÍS¡: L'annex d'aquest document inclou una descripció detallada sobre les solucions, que se treballen en aquesta pràctica, per a evitar les condicions de carrera. Es recomanable que l'alumne llisca detingudament aquest annex abans de desenvolupar les activitats.

6. Protegint secció crítica

Treballeu únicament sobre la versió del codi en la que SÍ s'observen condicions de carrera (CondCarr.c). Si el valor original de REPETICIONES ja produeix condicions de carrera utilitzar aquest.
En els següents exercicis, modificarem el codi protegint la secció crítica per a comprovar que no es produeixen condicions de carrera. També mesurarem els temps d'execució de les diferents versions i determinarem el cost en temps d'execució que implica l'ús d'exclusió mútua en l'accés a la secció crítica.

Exercici 4: Solució de sincronització amb "test_and_set"

Una vegada comprovat que es produeixen condicions de carrera copieu l'arxiu `CondCarr.c` en `CondCarrT.c`. Modifiqueu el codi `CondCarrT.c` per a garantir que l'accés a la variable compartida V és en exclusió mútua. Per a això realitzeu els següents passos:

1. Identifiqueu la part del codi corresponent a secció crítica (S.C), protegiu-la amb la funció `test_and_set`, seguint l'esquema de la figura-4 i la Taula 3 (Annex 2). Executeu el programa i comproveu que no es produeixen condicions de carrera.

2. Utilitzeu el comandament ***time*** per a conèixer el temps d'execució del programa amb la secció crítica protegida i anoteu-los en la següent taula

CondCarrT.c Protegint la secció crítica amb test_and_set	
Temps real d'execució	
Temps d'execució en mode usuari	
Temps d'execució en mode sistema	

3. Copieu CondCarrT.c en CondCarrTB.c. Observeu en CondCarrTB.c què passa si reescriu les seccions d'entrada i eixida i les situa en els llocs indicats en la Figura 4

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

        Protocol d'Entrada
     for (cont = 0; cont < REPETICIONS; cont = cont + 1) {
      V = V + 1;
     }
        Protocol d'Eixida
     printf("-------> Fin AGREGA (V = %ld)\n", V);
     pthread_exit(0);
}
```

Figura 4: Nou lloc de col·locació dels protocols de protecció

4. Anoteu els resultats de temps d'execució en la següent taula:

CondCarrTB.c Protegint tot el bucle "for" amb test_and_set	
Temps real d'execució	
Temps d'execució en mode usuari	
Temps d'execució en mode sistema	

5. Observant els resultats obtinguts identifiqueu quines diferències hi ha entre sincronitzar les seccions crítiques com s'indica en la Figura 3 o fer-ho com indica la Figura 4.

¿Què ha passat a l'utilitzar l'esquema de sincronització de la Figura 4?

¿Qui avantatge te sincronitzar les seccions crítiques com s'indica en la Figura 3?

Exercici 5: Solució de sincronització amb semàfors

Copieu l'arxiu CondCarr.c en CondCarrS.c i realitzeu les modificacions sobre aquest últim.

1. Protegiu la secció crítica utilitzant un semàfor POSIX (sem_t) com es descriu en la Taula 4 (Annex 3). Executeu el programa i comproveu que no se produeixen condicions de carrera.
2. Torneu a executar el codi amb l'ordre ***time*** per a obtenir el temps d'execució i anoteu els resultats.

CondCarrS.c Protegint la secció crítica amb semàfors sem_t	
Temps real d'execució	
Temps d'execució en mode usuari	
Temps d'execució en mode sistema	

Exercici6: Solució de sincronització amb mutex

Copieu l'arxiu CondCarr.c sobre CondCarrM.c i realitzeu les modificacions sobre aquest últim.

1. Protegiu la secció crítica utilitzant un mutex (pthread_mutex_t) tal i com es descriuen la Taula 5 (Annex 4). Executeu el nou programa i comproveu que no es produeixen condicions de carrera.

2. Utilitzant l'ordre ***time*** executeu el codi per a conèixer el temps d'execució del programa i anoteu els resultats en la següent taula.

CondCarrM.c Protegint la secció crítica amb mutex de "pthreads"	
Temps real d'execució	
Temps d'execució en mode usuari	
Temps d'execució en mode sistema	

En l'exemple desenvolupat en aquesta pràctica ¿què és més eficient l'espera activa o la passiva?

En general, ¿En quines condicions creeu que es millor usar espera activa? En general, ¿En quines condicions creeu que es millor usar espera passiva?

7. Activitats Opcionals

Per a comprovar què passa quan la secció crítica és gran i com influeix això en el mètode de sincronització triat, podem augmentar la durada de la secció crítica artificialment de forma anàloga a com es va proposar en la Taula 1, però introduint un retràs abans d'assignar el nou valor a la variable compartida V. Aquesta és la modificació que se proposa en la Taula 2.

	Codi original	Substituir per…..

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

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

Taula 2: Nova secció crítica a treballar

El xicotet retràs introduït de mig mil·lisegon en el codi proposat en la Taula 2 fa que augmente considerablement la probabilitat de que es produeixi una condició de carrera a més d'augmentar, també considerablement, el temps d'execució del programa.

Per a que el temps d'execució del programa siga fàcilment observable en tots els casos, cal disminuir el valor de la constant REPETICIONS.

1. Modifiqueu el codi de CondCarr.c, CondCarrT.c, CondCarrS.c i CondCarrM.c com indica la Taula 2. Disminuïu també en els quatre arxius el valor de REPETICIONS per a que el temps real d'execució de la versió sense sincronització estiga al voltant de mig segon (un valor de REPETICIONS entre 1000 i 10000 sol ser adequat, fer proves amb 10000). Per a que els resultats siguen comparables, òbviament, heu d'utilitzar el mateix valor de REPETICIONS en els quatre arxius.

2. Executeu amb l'ordre ***time*** les quate versions del codi i anoteu els temps d'execució.

Secció crítica llarga	Sense protegir CondCarr.c	testandSet CondCarrT.c	Semaphore CondCarrS.c	Mutex CondCarrM.c
Temps real d'execució				
Temps d'execució en mode usuari				

Temps d'execució en mode sistema				

3. Segons aquests resultats reviseu les respostes que heu donat a les qüestions formulades en l'exercici 6.

8. Annexes

Annex 1: Codi font de recolzament, "CondCarr.c".

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

#define REPETICIONS 20000000   //**CONSTANT

//****VARIABLES GLOBALS (COMPARTIDES)
  long int V = 100;       // Valor inicial

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

 //** FUNCIONS QUE EXECUTEN ELS FILS
 void *agrega (void *argumento) {
   long int cont, aux;
       for (cont = 0; cont < REPETICIONS; cont = cont + 1) {
        V = V + 1;
       }
       printf("-------> Fin AGREGA (V = %ld)\n", V);
       pthread_exit(0);
 }
 void *resta (void *argument) {
   long int cont,aux;
       for (cont = 0; cont < REPETICIONS; cont = cont + 1) {
        V = V - 1;
       }
       printf("-------> Fin RESTA  (V = %ld)\n", V);
       pthread_exit(0);
 }
void *inspecciona (void *argument) {
       for (;;) {
        usleep(200000);
        fprintf(stderr, "Inspeccio: Valor actual de V = %ld\n", V);
       }
 }
 //** PROGRAMA PRINCIPAL
 int main (void) {
   //Declaracio de les variables necessàries.
    pthread_t filSuma, filResta, filInspeccion;
    pthread_attr_t attr;

 // Inicializacio els atributs de les tasques (per defecte)
     pthread_attr_init(&attr);

 // EXERCICI: Creeu els tres fils proposats amb els atributs
 // EXERCICI: El fil principal ha d'esperar a que les
 // tasques "agrega" i "resta" acaben
  // Fin del programa principal
   fprintf(stderr, "-------> VALOR FINAL: V = %ld\n\n", V);
   exit(0);
}
```

Annex 2: Sincronització per espera activa. Test_and_set

L'espera activa és una tècnica de sincronització que consisteix en **establir una variable global de tipo booleà (*spinlock*) que indica si la secció crítica està ocupada**. La semàntica d'aquesta variable és: valor 0 indica FALSE i significa que la secció crítica no està ocupada; valor 1 indica TRUE i significa que la secció crítica està ocupada.

El mètode consisteix en implementar en la secció d'entrada un bucle que mostrege ininterrompudament el valor de la variable *spinlock*. De manera que només passarà a executar la secció crítica si està lliure, però abans d'entrar haurà d'establir el valor de la variable a "ocupat" (valor 1). Per a fer això de forma segura és necessari que l'operació de comprovació del valor de la variable i la seua assignació al valor "1" es faça de forma atòmica ja que és possible que es produeixi un canvi de context (o execució simultània en computadores *multi-core*) entre la comprovació de la variable i la seua assignació, produint-se així una condició de carrera en l'accés a la variable *spinlock*.

Per a aquesta raó, els processadors moderns incorporen en el seu joc d'instruccions operacions específiques que permeten comprovar i assignar el valor a una variable de forma atòmica. Concretament, en els processadors compatibles x86, existeix una instrucció "xchg" que intercanvia el valor de dos variables. Com l'operació consisteix en una sola instrucció màquina, la seua atomicitat està assegurada. Usant la instrucció "xchg", es pot construir una funció "test_and_set" que realitze de forma atòmica les operacions de comprovació i assignació comentades anteriorment. El codi que implementa aquesta operació "test_and_set" és el que apareix en la Figura 5 i està inclòs en el codi de recolzament que es proporciona amb aquesta pràctica.

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

 }
```

Figura 5. Codi de instrucció test_and_set del processador d'Intel

Tot i que la comprensió del codi subministrat per a la funció "test_and_set" no és l'objectiu d'aquesta pràctica, es interesant observar com en llenguatge C es pot incloure codi escrit en assemblador

Amb tot això, per a assegurar l'exclusió mútua en l'accés a la secció crítica utilitzant aquest mètode, cal modificar el codi tal i como s'indica en la taula 3.

//Declarar una variable global, el "spinlock" que utilitzaran tots els fils `int clau = 0;  // inicialment FALSE → secció crítica NO està ocupada.`	
Secció d'entrada	`while(test_and_set(&clau));`
Secció d'eixida	`clau=0;`

Taula 3: Protocol d'entrada i eixida amb Test_and_Set.

Annex 3: Sincronització per espera passiva. Semàfors

L'espera passiva s'aconsegueix amb l'ajuda del Sistema Operatiu. Quan un fil ha d'esperar per a entrar en la secció crítica (perquè altre fil està executant la seua secció crítica), es "suspèn" eliminant-lo de la llista de fils en estat "preparat" del planificador. D'aquesta forma els fils en espera no consumeixen temps de CPU, en lloc d'esperar en un bucle de consulta com en el cas d'espera activa.

Per a que els programadors puguin utilitzar l'espera passiva, el Sistema Operatiu ofereix uns objectes específics que s'anomenen "semàfors" (tipus sem_t). Un semàfor, il·lustrat en la Figura 6, està composat per un comptador, el valor inicial del qual es pot fixar en el moment de la seua creació, i una cua de fils suspesos a l'espera de ser reactivats. Inicialment el comptador ha de ser major o igual a zero i la cua de processos suspesos està buida. El semàfor suporta dos operacions:

- Operació sem_wait() (operació P en la notació de Dijkstra): Aquesta operació decrementa el comptador del semàfor i, si després d'efectuar el decrement el comptador és estrictament menor que zero, suspèn en la cua del semàfor al fil que va invocar l'operació.
- Operació sem_post() (operació V en la notació de Dijkstra): Aquesta operació incrementa el comptador del semàfor i, si després d'efectuar l'increment el comptador és menor o igual que zero, desperta al primer fil suspès en la cua del semàfor, aplicant una ordenació FIFO.

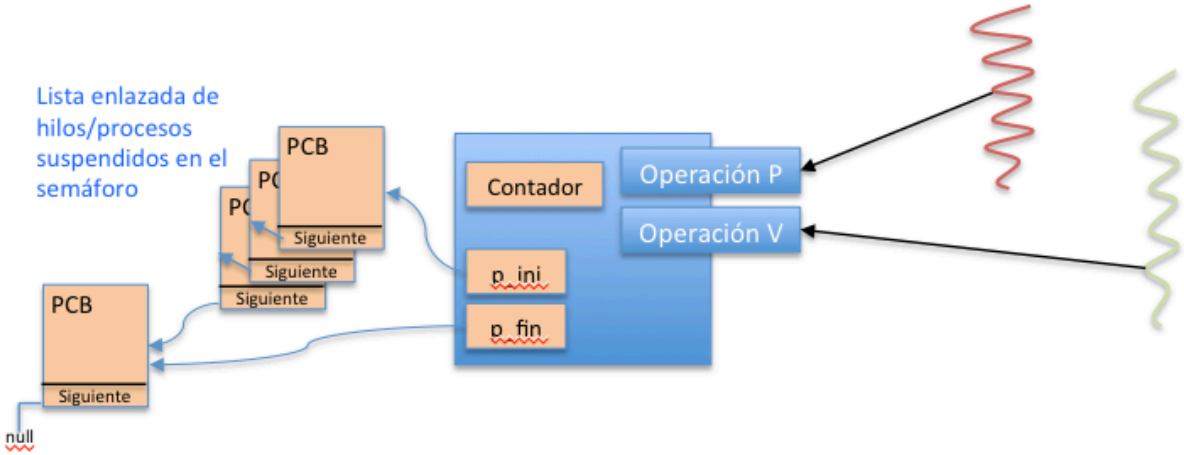

Figura 6: Estructura d'un semàfor i seues operacions.

Nota: Tot i que els semàfors POSIX (sem_t) formen part de l'estàndard, en MacOSX no funcionen. MacOSX proporciona altres objectes (semaphore_t) que se comporten de manera anàloga i poden usar-se per a oferir la mateixa interfície que ofereixen els semàfors POSIX.

Depenent de l'ús que li vullgam donar a un semàfor, definirem el seu valor inicial. El valor inicial d'un semàfor pot ser major o igual a zero i la seua semàntica associada es la de "nombre de recursos disponibles inicialment". Essencialment un semàfor és un comptador de recursos que poden ser sol·licitats (amb l'operació sem_wait) i alliberats (amb l'operació sem_post) de forma que quan no hi ha recursos disponibles, els fils que sol·liciten recursos es suspenen a l'espera de que algun recurs siga alliberat.

Especialment rellevants són els semàfors amb valor inicial igual a u. Com només hi ha un recurs lliure inicialment, només un fil podrà executar la secció crítica en exclusió mútua amb la resta. Aquests semàfors se solen aomenar "mutex" i són els que ens interesen en aquesta pràctica.

Con tot això, per a assegurar l'exclusió mútua en l'accés a la secció crítica utilitzant aquest mètode, cal modificar el codi tal i com s'indica en la taula 4.

<table>
<tr><td colspan="2">//Incloure les capçaleres de la llibreria de semàfors.
<pre>#include <semaphore.h></pre>
//Declarar una variable global, el “semàfor” que usaran tots els fils
<pre>sem_t sem; // No està inicialitzat, només declarat.</pre></td></tr>
<tr><td>Secció d’entrada</td><td><pre>sem_wait(&sem);</pre></td></tr>
<tr><td>Secció d’eixida</td><td><pre>sem_post(&sem);</pre></td></tr>
<tr><td colspan="2">//En el programa principal “main()” cal inicialitzar el semàfor.
<pre>sem_init(&sem,0,1); // El segon paràmetre indica que el semàfor no és compartit
                    // i el últim paràmetre indica el valor inicial,
                    // “1” en el nostre cas (exclusió mútua).</pre></td></tr>
</table>

Taula 4. Descripció del protocol d’entrada i eixida a la secció critica amb semàfors

Annex 4: Sincronització per espera passiva. “Mutex de pthreas”

A més de semàfors que ofereix el S.O. sota l’estàndard POSIX, la biblioteca de suport per a fils d’execució “pthread” proporciona altres objectes de sincronització: els “mutex” i les variables condició (“condition”). Els “mutex”, objectes “pthread_mutex_t”, s’usen per a resoldre el problema de l’exclusió mútua com indica el seu nom i poden considerar-se com a semàfors amb valor inicial “1” i valor màxim dels quals també és “1”. Òbviament són objectes específics creats per a assegurar l’exclusió mútua i no poden ser utilitzats com a comptadors de recursos.

De la mateixa manera que s’ha fet amb els altres mètodes de sincronització, es mostra l’ús dels “mutex de pthreads” en la següent taula

<table>
<tr><td colspan="2">//Incloure les capçaleres de la biblioteca de pthreads. Normalment ja està inclosa perquè estem usant fils.
<pre>#include <pthread.h></pre>
//Declarar una variable global, el “mutex” que utilitzaran tots els fils
<pre>pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; //Això ho declara i inicialitza.</pre></td></tr>
<tr><td>Secció d’entrada</td><td><pre>pthread_mutex_lock(&mutex);</pre></td></tr>
<tr><td>Secció d’eixida</td><td><pre>pthread_mutex_unlock(&mutex);</pre></td></tr>
</table>

Taula 5 : Descripción del protocol d’entrada i eixida a la secció critica amb Mutex.