Fonaments dels Sistemes Operatius

Departament de Informàtica de Sistemes i Computadores (DISCA)

Universitat Politècnica de València

fso

Pràctica 6
Sincronització de Secciones Crítiques

Versió 4.0

1. Objectius

L'objectiu de aquesta sessió és **comprendre quan es produeixen condicions de carrera** utilitzant variables compartides per part de distints fils d'execució, així com els mecanismes més bàsics per a evitar aquest problema. L'alumne haurà de treballar les solucions al problema usant tant **espera activa** com **espera passiva** i comprovar la sobrecarrega, en termes de temps d'execució.

2. Descripció del problema

Per a observar la problemàtica d'utilitzar variables compartides, proposem un problema senzill en el que 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 al final de l'execució la variable V hauria de valdre 100. Per a poder anar seguint els valors que va prenent la variable compartida, s'utilitza un tercer fil "inspecciona()" que consulta el valor de la variable i el mostra per pantalla a intervals d'un segon.

Figura 1. codi de les funcions agrega, resta i inspecciona

El fil "inspecciona()" no provoca condicions de carrera ja que accedeix a la variable V per a llegir el seu valor, però no escriu sobre ella. Els fils "agrega()" i "resta()" accedeixen a la variable V llegint-la i modificant-la repetidament, sense pausa. 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 l'execució d'un decrement V=V-1 degut a un canvi de context o ambdues (increment i decrement) s'executen concurrentment perquè cada activitat s'està executant en un `nucli` diferent del processador, és possible que es produeixi una condició de carrera i la variable V prenga valors inesperats, és a dir, que quan acaben ambdós fils, el seu valor no siga l'inicial (100 en el nostre cas).

Els escenaris en els que se pot produir la condició de carrera varien segons les característiques de la màquina on es treballa, com mostra la figura-2. Per exemple, en un processador amb múltiples nudis d'execució (*`multi-core`*), se podrà observar la condició de carrera fàcilment amb valors relativament baixos de la constant "REPETICIONS" (figura-1). Si el computador té un sol nucli d'execució, és menys probable que es produeixi una condició de carrera. En aquest cas haurà 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.

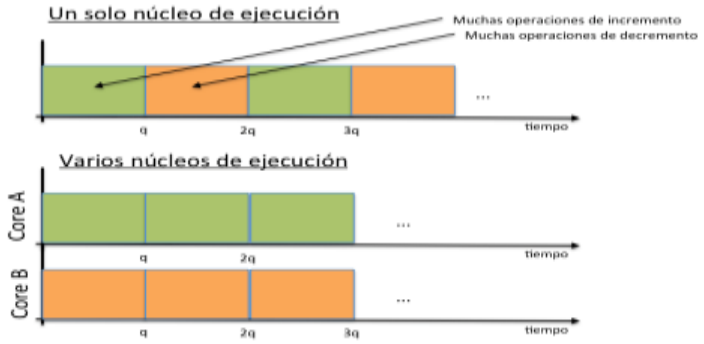

Figura 2. Execució dels fils agrega i resta en CPU, en màquines amb un i dos nuclis.

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	Sustituir 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. Escenario d'increment i decrement amb variable auxiliar.

3. Solucions per a evitar la condició de carrera

Per a evitar condicions de carrera, és necessari sincronitzar l'accés a les ***seccions crítiques del codi***, en el nostre caso les operacions de decrement e increment. Aquesta sincronització ha de garantir la **"exclusió mútua"**, acomplint-se que mentre un fil està executant una secció crítica altre fil no puga executar simultàniament la seua secció crítica. Per a aconseguir això, flanquejarem les seccions crítiques amb unes seccions de codi com protocol d'entrada i sortida, tal com indica la Figura 3.

```
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 de  Sortida o Secció de Sortida
    }
     printf("-------> Fin AGREGA (V = %ld)\n", V);
     pthread_exit(0);
 }
```

Figura 3. Protocol d'entrada i protocol de sortida a la secció critica de agrega().

El codi del protocol d'entrada i sortida depèn 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 tipo *"sem_t"* en POSIX.
  - Objectes "mutex" de la biblioteca "pthreads".

AVIS: En l'annex d'aquesta pràctica es fa una descripció detallada sobre les solucions per a evitar les condicions de carrera que es treballen en les activitats de la pràctica. Es recomanable que l'alumne llegeixi detingudament aquest annex abans de desenvolupar les activitats.

4. Observant condicions de carrera

Descargue el material d'aquesta pràctica del PoliformaT de l'assignatura, on trobarà un arxiu C que conté el programa que apareix en el annex-1 d'aquest butlletí. Per a compilar-ho teclege:

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

Exercici1: Creació de fils "CondCarr.c"

Completa el codi proporcionat en "CondCarr.c" de forma que es creen tres fils: un fil executarà la funció *agrega()*, altre la funció *resta()*, i l'últim fil executarà la funció *inspecciona()***,** veure figura 1. Utilitze 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 lo tant, presteu especial atenció i assegureu-vos de fer "pthread_join()" només per a els fils *agrega()* i *resta()* ja que el programa ha d'acabar quan els fils *agrega()* i *resta()* hagen acabat.

Compileu i execute 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 manera que el valor final de V fos diferent de l'inicial (100). No obstant això, per a valors baixos de REPETICIONES, això Pot no ocórrer observant-se que el valor final de la variable V és l'inicial (100). Això és degut al fet que en Sistemes amb un sol processador, no dóna temps a què a els 2 fils s'executin concurrentment i hagi Canvis de context. Si es crea el primer fil, AQUEST comença a executar-se, i acaba abans que comenci a executar-se el segon fil, els dos fils no arriben a executar concurrentment. A Sistemes multi-core és més fàcil observar una condició de carrera ja que la concurrència és real.

Exercici2: Provocant condicions de carrera

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

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

Nota: Si està treballant amb un computador *mono-core*, és probable que, per a observar una condició de carrera, necessite modificar la secció crítica, com se explica en la Tabla1. Feu-lo si és necessari i anoteu-lo.

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

```
$ time ./CondCarr
```

Aquesta ordre nos mostra el temps real (com si cronometrásemos) i els temps de CPU (mesurats pel planificador) executant instruccions d'usuari i del sistema operativo.

Amb l'ajuda de l'ordre **time** esbrineu el temps d'execució del programa *CondCarr.c* amb condicions de carrera, ja que no se 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, augmenteu generosament el valor REPETICIONES fins que el temps real d'execució del programa siga observable per un humà (del ordre de 200ms). Això ens proporcionarà una versió del programa molt propicia per a que se produeixin condicions de carrera.

Observant els resultats obtinguts identifiqueu si el computador usat disposa d'un o diversos *cores*. Raoneu la vostra resposta:

¿Un o diversos *cores*?

5. Protegint secció crítica

Treballeu únicament sobre la **versió del codi en la que SÍ se observen condicions de carrera (CondCarr.c). Si el valor original de REPETICIONES produeix condicions de carrera utilitzar aquest.**

En els següents passos de la pràctica, modificarem el codi per a protegir secció crítica i comprovar que no es produeixen condicions de carrera. També mesurarem els temps d'execució de les diferents versions per a determinar el cost en temps d'execució que implica la introducció de la exclusió mútua en l'accés a la secció crítica.

Exercici3: Solució de sincronització amb "test_and_set"

Una vegada comprovat que es produeixen condicions de carrera copieu l'arxiu `CondCarr.c` sobre `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 el següent:

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

2. Utilitzeu el comando **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 reescriviu les seccions d'entrada i sortida i les situa en els llocs indicats en la Figura 4

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

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

Figura 4: Nou lloc de col·locació de la seccions d'entrada i sortida

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

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

5. A la llum dels resultats obtinguts identifiqueu quina diferència hi ha entre sincronitzar les seccions crítiques com s'indica en la Figura 3 o fer-ho com indica la Figura 4.

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

¿Quins avantatge té sincronitzar les seccions crítiques com s'indica en la Figura 3?

Exercici4: Solució de sincronització amb semàfors

Copieu l'arxiu *CondCarr.c* sobre *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 es produeixen condicions de carrera.

2. Torne a executar el codi amb la ordre *time* per a obtenir el temps d'execució i anoteu els resultats en la següent taula.

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	

Exercici5: 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 se descriu en la Taula 5 (Annex 4). Executeu el nou programa i comproveu que no es produeixen condicions de carrera.

2. Utilitzant l'ordre *time* execute 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 la espera activa o la passiva?

En general, ¿En quines condicions creieu que és millor usar espera activa?

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

6. Actividats Opcionals

Per a comprovar què passa quan la secció crítica es gran i cóm 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	Sustituir 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: Nueva secció crítica a trabajar

El menut retràs introduït de mig mil·lisegon en el codi proposat en la Taula 2 fa que augmente considerablement la probabilitat de que se 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 s'indica en la Taula 2. Disminueix també en els quatre arxius el valor de REPETICIONS per a que el temps real d'execució de la versió que no inclou sincronització estiga en torno al mig segon (un valor de REPETICIONS entre 1000 i 10000 sol ser adequado, **fer les proves amb 10000**). Per a que els resultats siguen comparables, òbviament, ha d'usar el mateix valor de REPETICIONS en els quatre arxius.

2. Compileu i executeu, amb la ordre **time**, les quatre versions del codi i anoteu els resultats dels temps d'execució en les següents taules.

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. A la llum d'aquests resultats reviseu les respostes que heu donat a les qüestions formulades en l'exercici 5.

7. Annexos

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 *argument) {
   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 necessaries.
    pthread_t hiloSuma, hiloResta, hiloInspeccion;
    pthread_attr_t attr;

 // Inicialitzacio dels 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" finalitzen
  // 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 tipus 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 faça un mostreig ininterrompudament el valor de la variable *spinlock*. El programa 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 es necessari que l'operació de comprovació del valor de la variable i la seua assignació al valor "1" se faça de forma atòmica ja que es possible que se 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 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" es el que apareix en la Figura 5 i està inclòs en el codi de recolzament que se 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 instrucción 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 cóm en llenguatge C se pot incloure codi escrit en assemblador.

Amb tot això, per a assegurar la exclusió mútua en l'accés a la secció crítica utilitzant aquest mètode, cal que modificar el codi tal i com se 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 de sortida	`clau=0;`

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

Annex 3: Sincronització per espera passiva. Semàfors

La 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 programadores puguen 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 5, està compost 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 dues operacions:

- Operació **sem_wait()** (operació P en la notació de Dijkstra): Aquesta operació decrementa el comptador del semàfor y, 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 y, 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 ordreació FIFO.

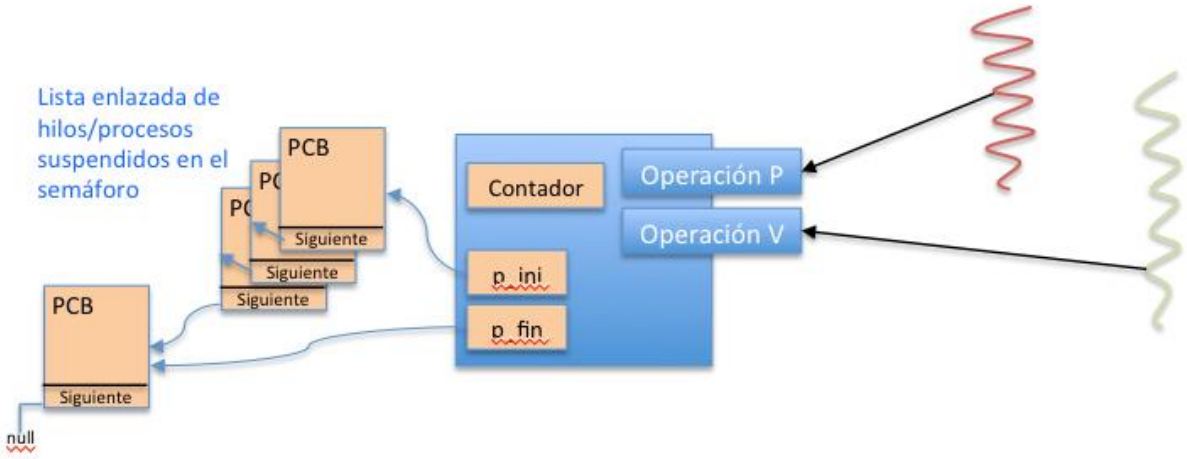

Figura 5: Estructura d'un semàfor i les seues operacions.

> **Nota:** Tot o que els semàfors POSIX (sem_t) formen part del estàndard, en MacOSX no funcionen. Compte si feu proves sobre aquest Sistema Operatiu. MacOSX proporciona altres objectes (semaphore_t) que es comporten de manera anàloga i poden usar se per a oferir la mateixa interfície que ofereixen els semàfors POSIX.

Dependent 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 és 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 un. Como 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 anomenar "mutex" i són els que ens interessen en aquesta pràctica.

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 com s'indica en la taula 4.

//Incloure les capçaleres de la llibreria de semàfors.
`#include <semaphore.h>`
//Declarar una variable global, el "semàfor" que utilitzarán tots els fils
`sem_t sem; // No està inicializat, només declarat.`	

Sección d'entrada	`sem_wait(&sem);`
Sección de sortida	`sem_post(&sem);`

//En el programa principal "main()" cal inicializar el semàfor.
`sem_init(&sem,0,1); // El segon paràmetre indica que el semàfor no és compartit`
`// i l'últim paràmetre indica el valor inicial,`
`// "1" en nostre cas (exclusió mútua).`	

Taula 4. Descripció del protocol d'entrada i sortida a la secció critica amb semàfors

Annex 4: Sincronització per espera passiva. "Mutex de pthreas"

A més de semàfors que ofereix el S.O. baix l'estàndard POSIX, la llibreria 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 no i poden considerar-se com semàfors amb valor inicial "1" i el valor màxim dels quals és també "1". Òbviament són objectes específics creats per a assegurar l'exclusió mútua i no poden ser utilitzats com comptadors de recursos.

Al igual que s'ha fet amb els altres mètodes de sincronització, es mostra l'ús dels "mutex de pthreads" en la següent taula

//Incloure les capçaleres de la llibreria de pthreads. Normalment ja està inclosa perquè estem utilitzant fils.
`#include <pthread.h>`
//Declarar una variable global, el "mutex" que utilitzaran tots els fils
`pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; //Això ho declara i inicialitza.`	

Sección d'entrada	`pthread_mutex_lock(&mutex);`
Sección de sortida	`pthread_mutex_unlock(&mutex);`

Taula 5 : Descripció del protocol d'entrada i sortida a la secció critica amb Mutex.