

# Fonaments dels Sistemes Operatius (FSO)

Departament d'Informàtica de Sistemes i Computadors (DISCA)  
*Universitat Politècnica de València*

Bloc Temàtic 2: Gestió de Processos  
Unitat Temàtica 6

## Sincronització: Solucions Bàsiques

f SO

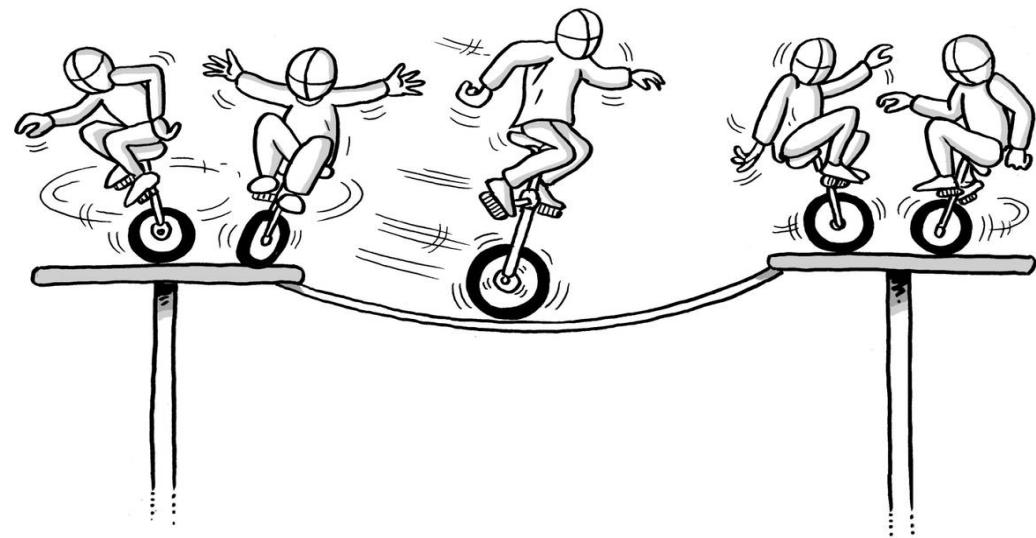
DISCA



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA

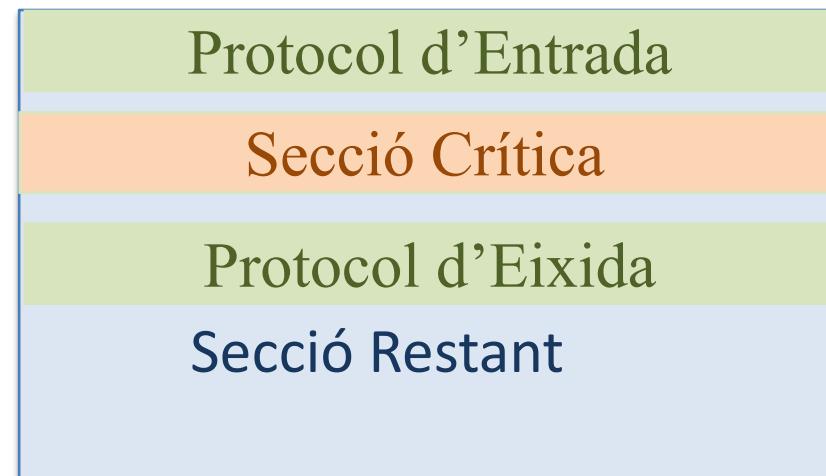
- **Objectius**
  - Familiaritzar-se amb el concepte de **secció crítica**
  - Conèixer els mecanismes de **sincronització** que ofereix el **hardware**
  - Resoldre problemes de sincronització mitjançant solucions **software** i **hardware**
- **Bibliografia**
  - “Fundamentos de sistemas operativos” Silberschatz 7<sup>a</sup> Ed (Capítulo-6)
  - “Sistemas operativos: una visión aplicada” Carretero 2<sup>º</sup> Ed
  - **UNIX Programación Práctica**, Kay A. Robbins, Steven Robbins. Prentice Hall. ISBN 968-880-959-4

- **Contingut**
  - El problema de la secció crítica
  - Solucions software
  - Solucions hardware
  - Espera activa
  - Exercicis



- **El problema de la secció crítica**
  - Hi ha N processos/fils executant-se concurrentment accedint a dades compartides, amb  $N > 1$
  - En cada procés/fil s'identifiquen àrees de codi denominades:
    - **Secció crítica:** zona de codi en la què **accedeix a les dades compartides**. En el procés/fil accedeix almenys una d'aquestes zones.
    - **Secció restant:** zones de codi en les què no accedeix a dades compartides.
- **Solució**
  - **Protocol:** per a **sincronitzar l'accés a variables compartides** i evitar el problema de la **condició de carrera**

Protocol: Que gestione la **serialització** en l'accés a la **secció crítica** per part dels processos/fils



- El **protocol** de protecció de secció crítica ha d'acomplir les tres **condicions** següents:
  - **Exclusió mútua**: si un procés està executant la seu secció crítica, cap altre procés pot estar executant la seu.
  - **Progrés**: si cap procés està executant la seu secció crítica i n'hi ha d'altres que desitgen entrar a les seues, aleshores la decisió de qui procés entrarà a la secció crítica es pren en un temps finit i només depèn dels processos que desitgen entrar.
  - **Espera limitada**: Després que un procés haja sol·licitat entrar en la seu secció crítica, hi ha un límit en el nombre de vegades que es permet que altres processos entren a les seues seccions crítiques.
- Suposant que:
  - Els processos s'executen a velocitat no nul·la.
  - La correcció no ha de dependre de fer suposicions sobre la velocitat relativa d'execució dels processos.

- El **protocol d' accés a secció crítica** ha d'acomplir les tres **condicions** :

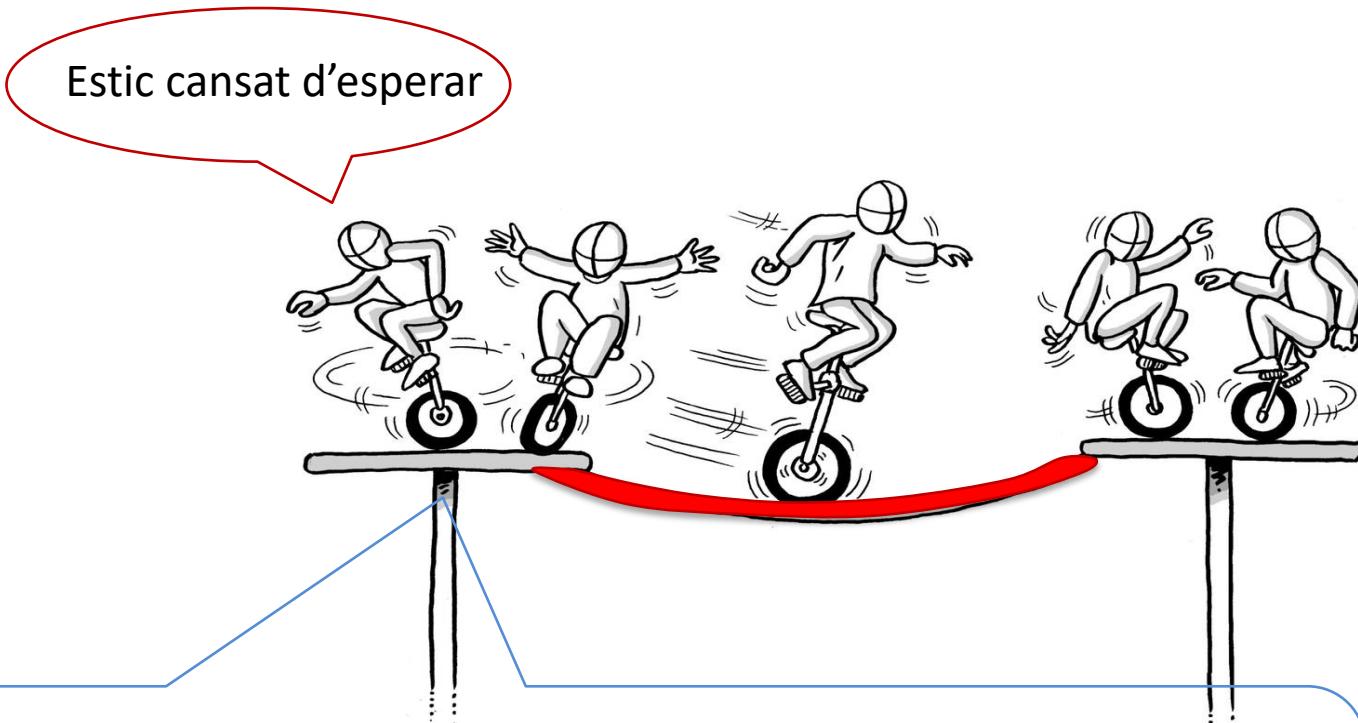


**Exclusió mútua:** si un procés està executant la seu secció crítica, cap altre procés pot estar executant la seu

- El **protocol** d'accés a secció crítica ha d'acomplir les tres **condicions** :



- El **protocol** d'accés a secció crítica ha d'acomplir les tres **condicions** :

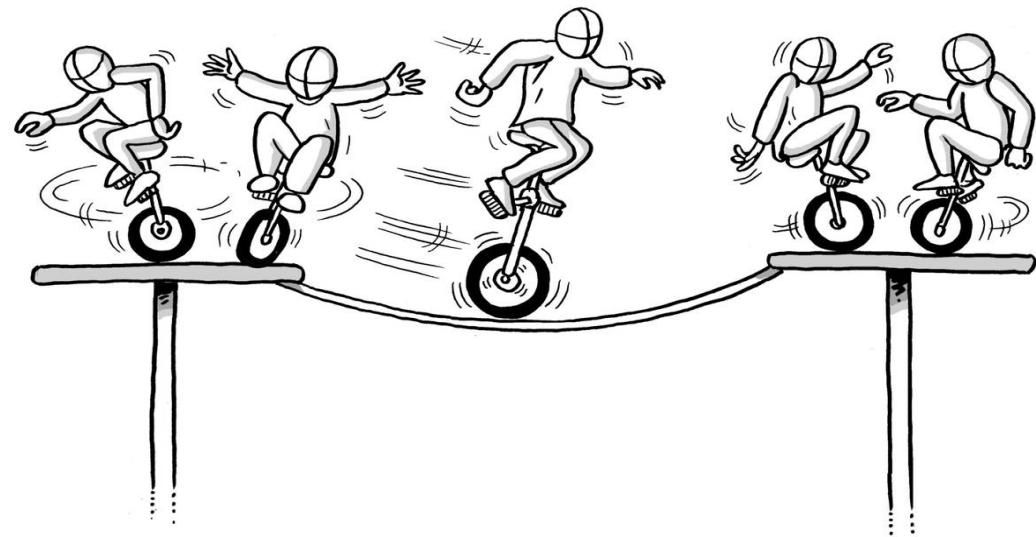


**Espera limitada:** Després que un procés haja sol·licitat entrar en la seua secció crítica, hi ha un límit en el nombre de vegades que es permet que altres processos entren a les seues seccions crítiques

- Alternatives per a implementar el protocol d'accés a la secció crítica
  - Solucions bàsiques
    - Solucions de tipus “**software**”
      - El protocol s'implementa amb codi a nivell d'usuari, sense mecanismes afegits.  
Algorismes de Dekker.
    - Solucions de tipus “**hardware**”
      - S'aprofiten instruccions màquina especials per a implementar el protocol
  - Solucions amb suport del **sistema operatiu**
    - El protocol s'implementa mitjançat mecanismes proporcionats pel sistema operatiu, a través de **crides al sistema**
  - Solucions amb suport del **llenguatge de programació (construccions lingüístiques)**
    - Alguns llenguatges de programació disposen de tipus de dades especials que garantieixen el seu accés en exclusió mutua de forma automàtica
    - Exemples: Ada95, Java

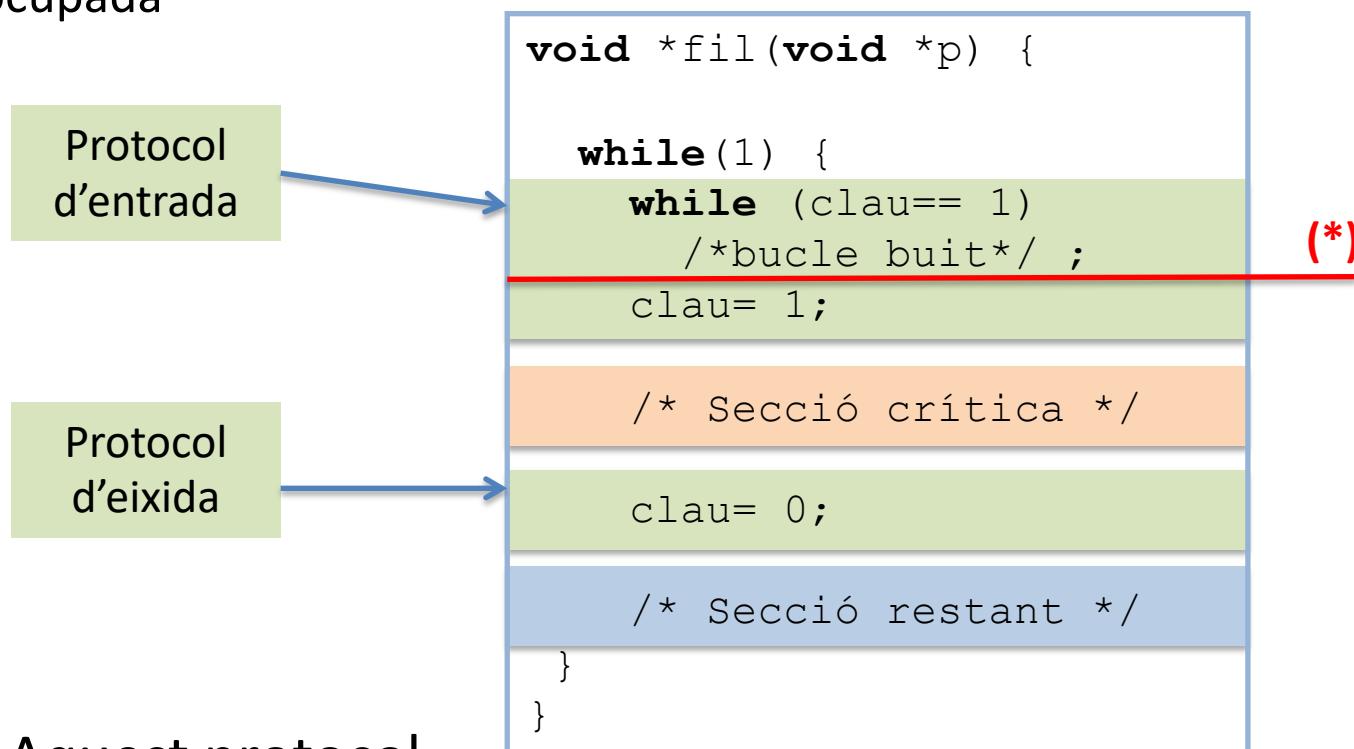
- **Contingut**

- El problema de la secció crítica
- **Solucions software**
- Solucions hardware
- Espera activa
- Exercicis



## • Algorisme bàsic

- Protocol per a diversos fils del mateix tipus, que es sincronitzen mitjançant una variable global "clau" que indica si la secció crítica està ocupada



- Aquest protocol

- No acompleix “l'exclusió mútua”. Si es produeix un canvi de context en (\*) és possible que diversos fils entren en la seua secció crítica.

- “Dekker nº 1”, alternància estricta
  - Protocol per a dos fils, que es sincronitzen mitjançant una variable global “torn” inicializada a I o J que indica a quin fil li toca executar la secció crítica

```
void *fil_I(void *p) {  
  
    while(1) {  
        while (torn != I)  
            /*bucle buit*/ ;  
  
        /* Secció crítica */  
  
        torn = J;  
  
        /* Secció restante */  
    }  
}
```

```
void *fil_J(void *p) {  
  
    while(1) {  
        while (torn != J)  
            /*bucle buit*/ ;  
  
        /* Secció crítica */  
  
        torn = I;  
  
        /* Secció restante */  
    }  
}
```

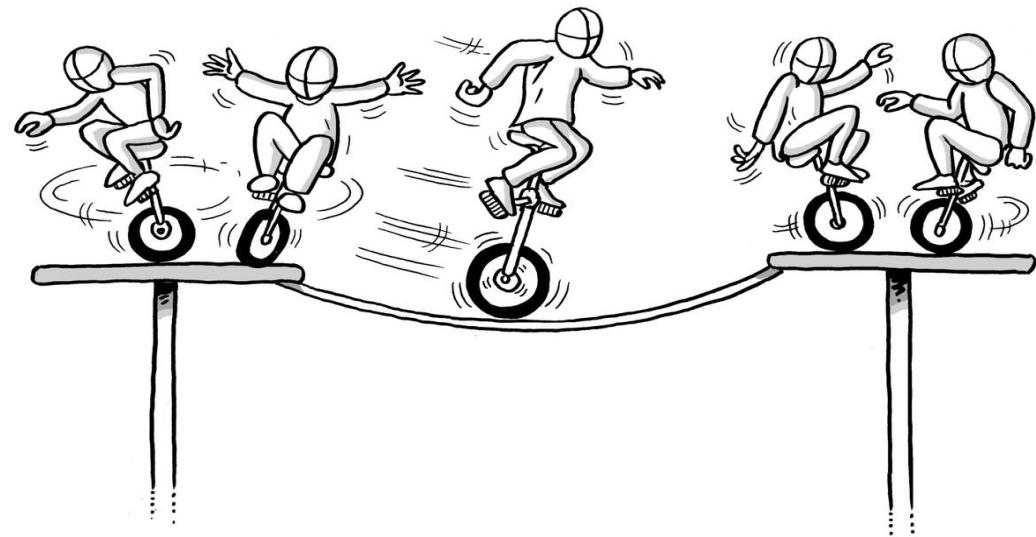
- Aquest protocol

- Només serveix per a dos fils. Necessitat de conèixer a priori la quantitat de fils a sincronitzar.
- No compleix la condició de “progrés”. Imposa una “velocitat relativa” entre els processos/fils.

- Solucions **software**
  - Hi ha solucions software completamente correctes: Algorismes de Dekker, Algorisme de Peterson, Algorismes de Lamport... L'estudi d'aquests algorismes seria objecte de cursos més avançats.
  - S'apliquen en **sistemes distribuïts** o, en general, quan les activitats a sincronitzar (fils) no s'executen en el mateix processador.

- **Contingut**

- El problema de la secció crítica
- Solucions software
- **Solucions hardware**
- Espera activa
- Exercicis



- Solucions **hardware**
  - Inhibició d'interrupcions
    - Requereix que els fils s'executen en el mateix processador
  - Instruccions **atòmiques** per a la manipulació de memòria.
    - **Test-and-set**
    - **Intercanvi**



- **Inhibició d'interrupcions (Solució hardware)**

- Es realitza utilitzant les instruccions:

- **DI (disable interrupts)**: inhabilitar interrupcions del processador
    - **EI (enable interrupts)**: tornar a habilitar interrupcions

- L'**exclusió mútua** s'aconsegueix **inhibitint els canvis de context** durant l'execució de la secció crítica, obligant així a que les seccions crítiques s'executin de manera atòmica:

```
...
DI ;
/* Secció crítica */
EI ;
/* Secció restant */
...
```

Aquesta tècnica, impideix que es realitzent **canvis de context** mentre un fil executa la seua secció crítica. Això és **molt més del necessari**. Només cal impedir que s'executen les seccions crítiques d'altres fils, ja que les seues seccions restants sí es poden executar

- Solució només viable al **nivell del nucli** del sistema operatiu:

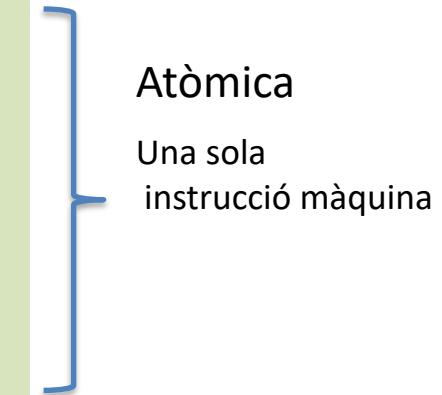
- no és desitjable deixar el control de les interrupcions en mans dels processos d'usuari, les instruccions DI i EI només s'hi poden executar en **mode privilegiat**



# Solucions bàsiques

- “**test-and-set**” instrucció atòmica. (Solució hardware)
  - La instrucció “test and set” permet avaluar i modificar una variable atòmicament en una sola instrucció màquina
  - L’**especificació funcional** de la instrucció “test and set” és la següent:

```
int test_and_set (int *objectiu) {  
    int aux;  
  
    aux = *objectiu;  
    *objectiu = 1;  
    return aux;  
}
```



- retorna el valor que conté la variable “objectiu” (1 o 0, indicant cert o fals) i assigna 1 (cert) a “objectiu”, tot de forma atòmica.

**Atòmica = Indivisible, no és pot interrompre la seva execució**

# Solucions bàsiques

- “**test-and-set**” instrucció atòmica. (Solució Hardware)
  - Amb “test-and-set” podem construir un protocol d'accés a la secció crítica a partir de “l'algorisme bàsic” estudiat abans.
  - N processos/fils que executen el codi següent,
  - Tots els fils comparteixen la variable **clau** que està inicialitzada a 0 (fals).

```

void *fil_i(void *p) {
    int clau = 0;

    while(1) {
        while (test_and_set(&clau))
            /*bucle buit*/ ;

        /* Secció crítica */

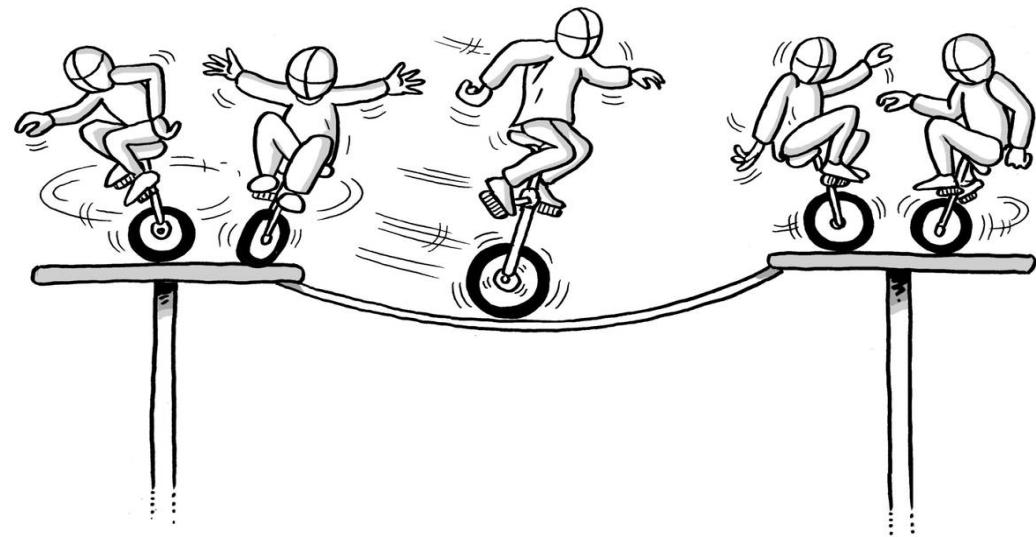
        clau= 0; /*fals*/
    }
}

```

- Aquesta solució acompleix exclusió mútua i progrés, però no acompleix “l'espera limitada”.
  - Una vegada que un fil ha sortit de la secció crítica, el següent a executar el "test\_and\_set" serà qui entre, i atès que el protocol d'entrada no defineix cap orde no és possible assegurar que tots els processos que estan esperant puguen entrar.

- **Contingut**

- El problema de la secció crítica
- Solucions software
- Solucions hardware
- **Espera activa**
- Exercicis



- Concepte d'**espera activa**

- Les solucions anteriors (software i hardware tipus "test-and-set") comparteixen una característica anomenada **espera activa**.
- El protocol d'entrada impedeix que un procés entre en la seua secció crítica, fent que aquest procés execute un bucle buit que **consumeix temps de CPU**

## ESPERA ACTIVA

```
void *fil_I(void *p) {  
  
    while(1) {  
        while (torn != I)  
            /*bucle buit*/ ;  
        /* Secció crítica */  
        torn = J;  
        /* Secció restant */  
    }  
}
```

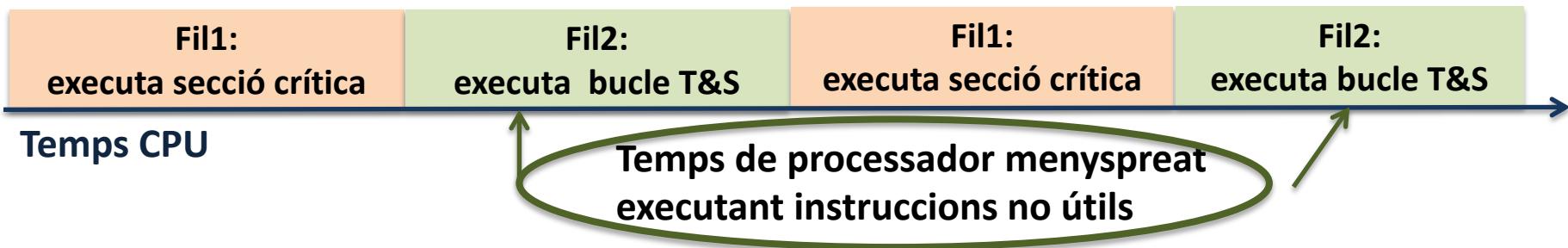
```
void *fil_i(void *p) {  
  
    while(1) {  
        while test_and_set(&clau)  
            /*bucle buit*/ ;  
        /* Secció crítica */  
        clau = 0; /*falso*/  
        /* Secció restant */  
    }  
}
```

Solucions software  
("Dekker nº 1", alternància estricta)

Protocol “test-and-set” instrucció atòmica

- **Problemes de l'espera activa**

- L'espera activa es l'única forma d'impedir que un procés/fil passe d'un cert punt del seu codi **sense recórrer al sistema operatiu**
- L'espera activa no és desitjable, perquè
  - **Si la planificació dels fils és per prioritats,**
    - Quan un fil queda “atrapat” en el protocol d'entrada tenint més prioritat que el fil que s'hi troba executant la secció crítica, aleshores tots els fils de l'aplicació es quedaran **bloquejats per a sempre**. (inversió de prioritat)
  - Inclús en **escenaris de planificació menys problemàtics, com per exemple torn rotatori**,
    - l'espera activa presenta el problema d'**infrautilitzación del processador**, donat que els fils poden consumir molts quants de temps sense fer res més que executar el bucle buit



- Alternatives a l'espera activa
  - Recórrer a l'ajuda del **Sistema Operatiu** per a **suspendre** el fil quan executa el `test_and_set` i no consigueix entrar en la secció crítica.

```
while( test_and_set(&clau) ) {usleep(periode);}
```
  - O simplement que el fil **abandone la CPU** quan executa el `test_and_set` i no aconsigueix entrar en la secció crítica.

```
while( test_and_set(&clau) ) {yield();}
```
  - Que el Sistema Operatiu oferisca **objetes en els què un procés es puga suspendre fins que altre procés el desperte.**

## . SEMÀFORS I MUTEX

- **S'implementen com a crides al sistema.** El S.O. proporciona primitives específiques per a resoldre el problema de la secció crítica
- **No s'hi produeix espera activa.** El S.O. pot detenir processos/fils de forma eficient, marcant-los com suspesos en el seu PCB.

Espera per esdeveniment

- Exemple: Productor/consumidor

- Solució hardware

- `test_and_set`, instrucció atòmica iindivisible

```
void *func_prod(void *p) {  
    int item;  
  
    while(1) {  
        item = produir();  
  
        while (test_and_set(&clau))  
            /*bucle buit*/ ;  
  
        while (comptador == N)  
            /*bucle buit*/ ;  
        buffer[entrada] = item;  
        entrada = (entrada + 1) % N;  
        comptador = comptador + 1;  
  
        clau = 0;  
    }  
}
```

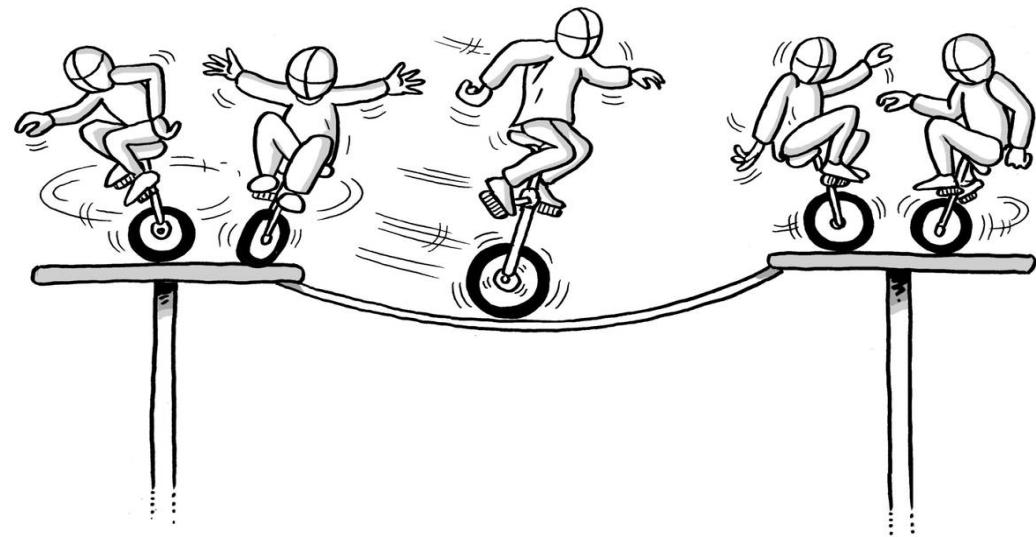
```
#define N 20  
int buffer[N];  
int entrada, eixida,  
comptador;  
int clau = 0;
```

```
void *func_cons(void *p) {  
    int item;  
  
    while(1) {  
        while (test_and_set(&clau))  
            /*bucle buit*/ ;  
  
        while (comptador == 0)  
            /*bucle vacio*/ ;  
        item = buffer[eixida];  
        eixida = (eixida + 1) % N;  
        comptador = comptador - 1;  
  
        clau = 0;  
  
        consumir(item);  
    }  
}
```

- Aquest codi té problemes i no funciona
    - On apareixen els problemes?

- **Contingut**

- El problema de la secció crítica
- Solucions software
- Solucions hardware
- Espera activa
- **Exercicis**



## Exercici 6.1:

Raoneu adequadament si el codi següent codi és una bona solució al problema de secció crítica per a dos fils (fil\_0 i fil\_1)

```
#include <stdio.h>
/**Compartides */
int flag[2];
```

```
fil_i(void) {
    while ( 1 ) {

        secció_restant;

        flag[i] = 1;
        while(flag[(i+1) % 2]);

        secció_crítica;

        flag[i] = 0;
    }
}
```

Determineu si s'acompleixen les tres condicions del protocol de la secció crítica

# Exercicis

## Exercici 6.2:

Compareu les tres solucions proposades amb test and set al problema de secció crítica

```
/* Solució a */
void *fil(void *p) {

    while(1) {
        while (test_and_set(&clau));
        /* Secció crítica */
        clau= 0;
        /* Secció restant */
    }
}
```

```
/* Solució b */
void *fil(void *p) {

    while(1) {
        while (test_and_set(&clau)) usleep(1000);
        /* Secció crítica */
        clau = 0;
        /* Secció restant */
    }
}
```

```
#include <stdio.h>
/**Compartides */
int clau=0;
```

```
/* Solución c*/
void *fil(void *p) {

    while(1) {
        while (test_and_set(&clau)) yield();
        /* Secció crítica */
        clau = 0;
        /* Secció restant */
    }
}
```

Nota: La crida **yield()** permet a un procés cedir el que li resta del seu quant de temps de CPU.  
D'aquesta manera, el procés que invoca **yield()** passa a PREPARAT i allibera la CPU, i dóna al planificador l'oportunitat de seleccionar un altre procés per a la seva execució.

## Ejercicio 6.3:

Observeu el fragment de codi corresponent a dos fils que pertanyen al mateix procés i s'executen concurrentment. Indiqueu quins recursos compartits apareixen en el codi i quins mecanismes s'utilitzen per evitar les condicions de carrera.

```
void *agrega (void *argument)
{
    int ct,tmp;
    for (ct=0;ct<REPE;ct++)
    {
        while(test_and_set(&clau)==1);

        tmp=V;
        tmp++;
        V=tmp;
        clau=0;
    }
    printf("->AGREGA (V=%ld) \n",V);
    pthread_exit(0);
}
```

```
void *resta (void *argument)
{
    int ct,tmp;
    for (ct=0;ct<REPE;ct++)
    {
        while(test_and_set(&clau)==1);

        tmp=V;
        tmp--;
        V=tmp;
        clau=0;
    }
    printf("->RESTA (V=%ld) \n", V);
    pthread_exit(0);
}
```

Nota: Les variables i funcions que no estan definides dins de les funcions `agrega` i `resta` són definides com globals.