Fundamentos de los Sistemas Operativos (FSO)

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

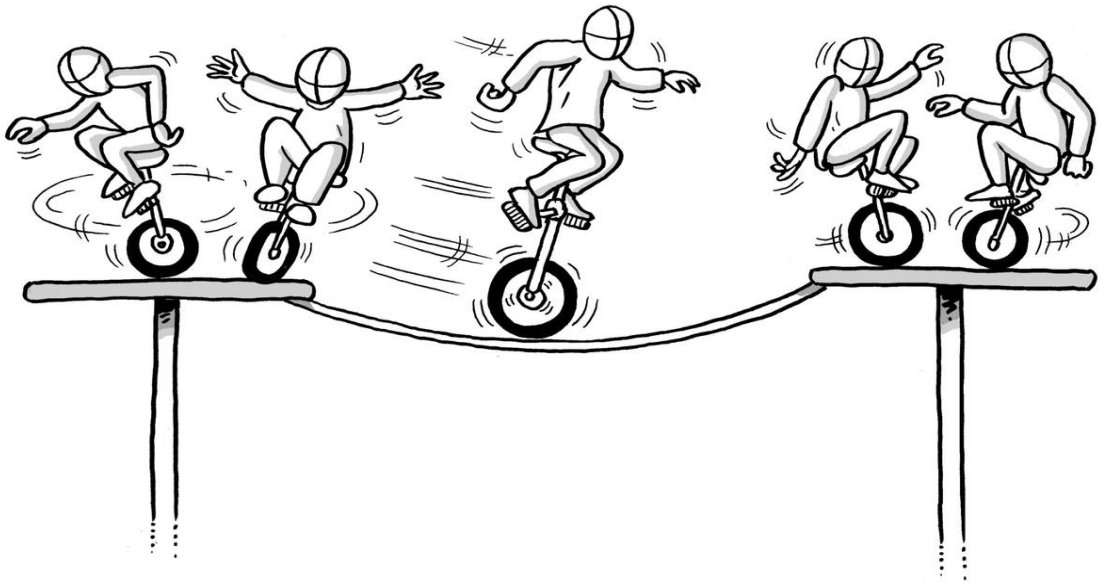
Bloque Temático 2: Gestión de Procesos
Unidad Temática 6

Sincronización: Soluciones Básicas

fSO

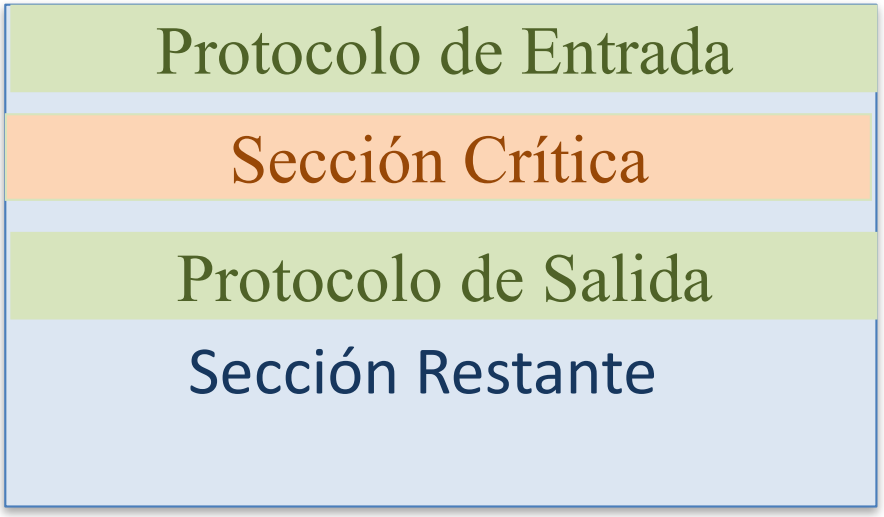
- **Objetivos**
  - Familiarizarse con el concepto de **sección crítica**
  - Conocer los mecanismos de **sincronización** que ofrece el **hardware**
  - Resolver problemas de sincronización mediante soluciones **software** y **hardware**
- **Bibliografía**
  - "Fundamentos de sistemas operativos" Silberschatz 7ª Ed (Capítulo-6)
  - "Sistemas operativos: una visión aplicada" Carretero 2º Ed
  - UNIX Programación Práctica", Kay A. Robbins, Steven Robbins. Prentice Hall. ISBN 968-880-959-4

- **Contenido**
  - **El problema de la Sección Crítica**
  - Soluciones software
  - Soluciones hardware
  - Espera activa
  - Ejercicios



- **El problema de la sección crítica**
  - Hay N procesos/hilos ejecutándose concurrentemente accediendo a datos compartidos, con N>1
  - En cada proceso/hilo se identifican áreas de código denominadas:
    - **Sección Crítica:** zona de código en la que se **accede a los datos compartidos**. En el proceso/hilo aparece al menos una de estas zonas
    - **Sección Restante**: zonas de código en que no se acceden datos compartidos
- **Solución**
  - **Protocolo:** para **sincronizar el acceso a variables compartidas** y evitar el problema de la **condición de carrera**

Protocolo: Que gestione la ***serialización*** en el **acceso a la sección crítica** por parte de los procesos/hilos



- **Protocolo** de acceso a **sección critica** debe cumplir tres condiciones:

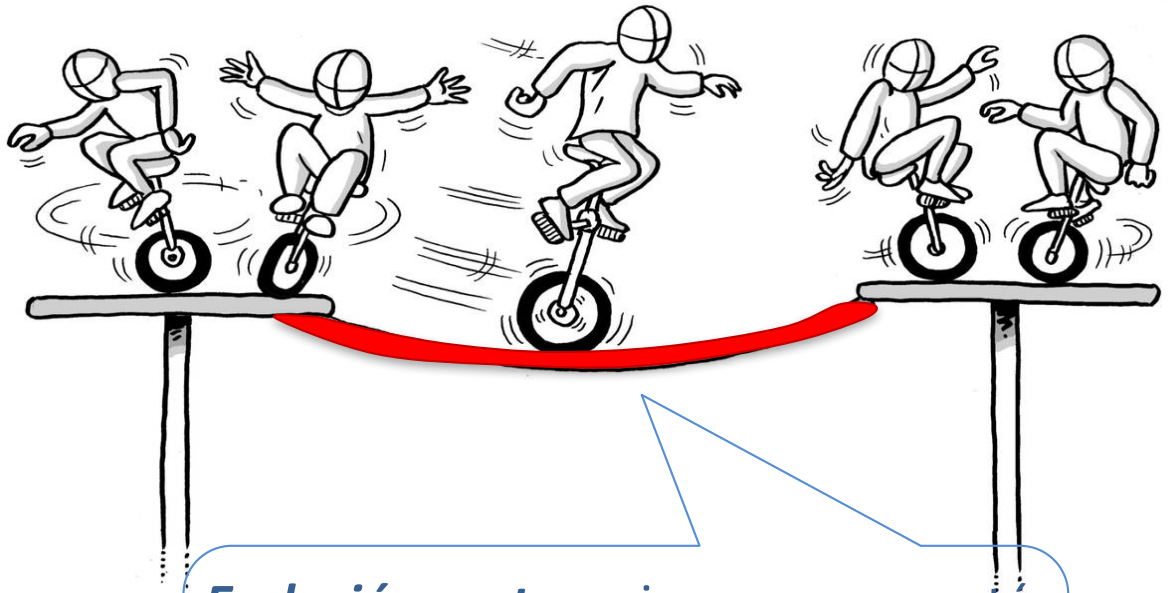
Exclusión mutua: si un proceso está ejecutando su sección crítica, ningún otro proceso puede estar ejecutando la suya

Progreso: si ningún proceso está ejecutando su sección crítica y hay otros que desean entrar a las suyas, entonces la decisión de qué proceso entrará a la sección crítica se toma en un tiempo finito y sólo depende de los procesos que desean entrar

Espera limitada: Después de que un proceso haya solicitado entrar en su sección crítica, existe un límite en el número de veces que se permite que otros procesos entren a sus secciones críticas

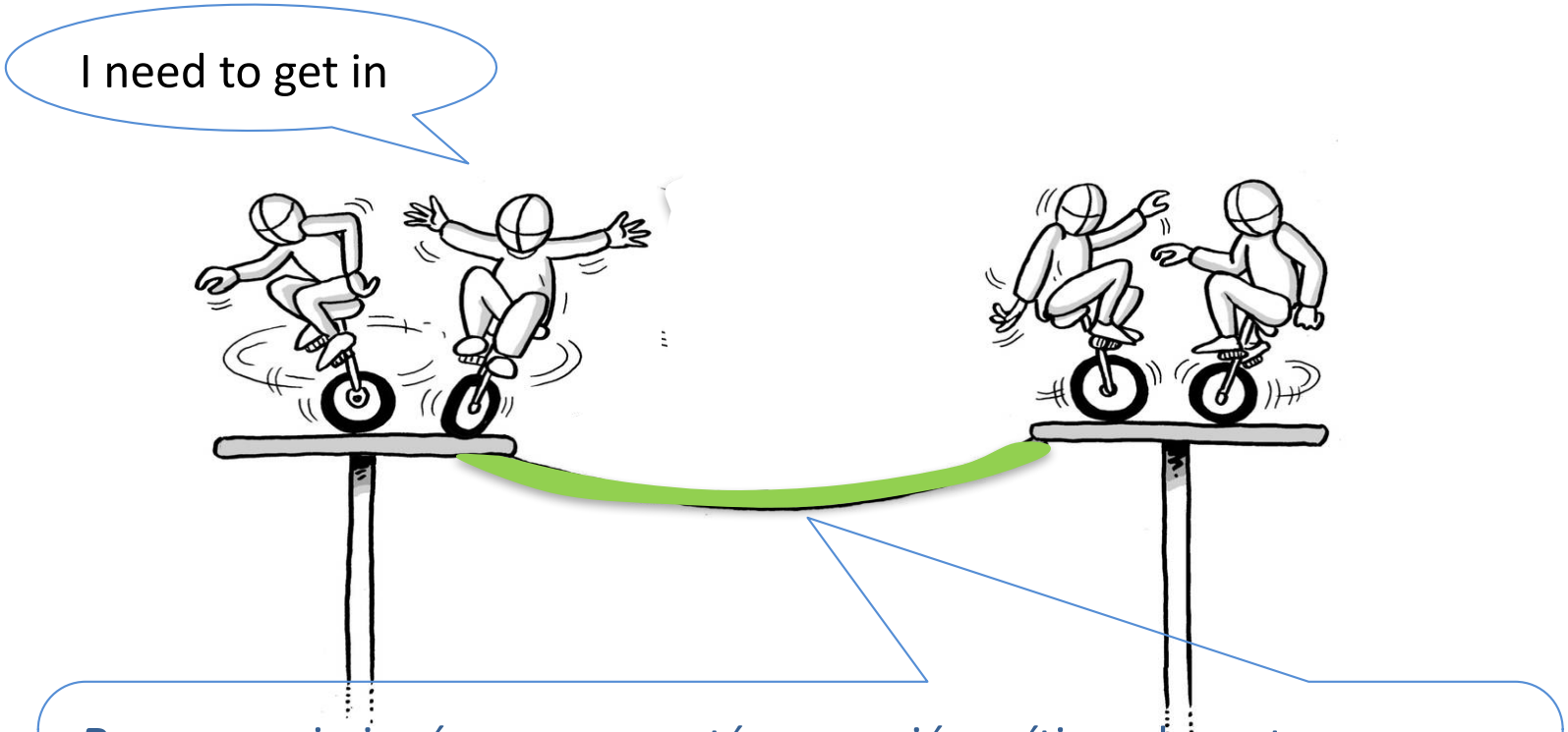
- Estas condiciones se deben cumplir suponiendo que:
  - Los procesos se ejecutan a velocidad no nula
  - La corrección no ha de depender de hacer suposiciones sobre la velocidad relativa de ejecución de los procesos

- **Protocolo de acceso a sección critica** debe cumplir tres condiciones:



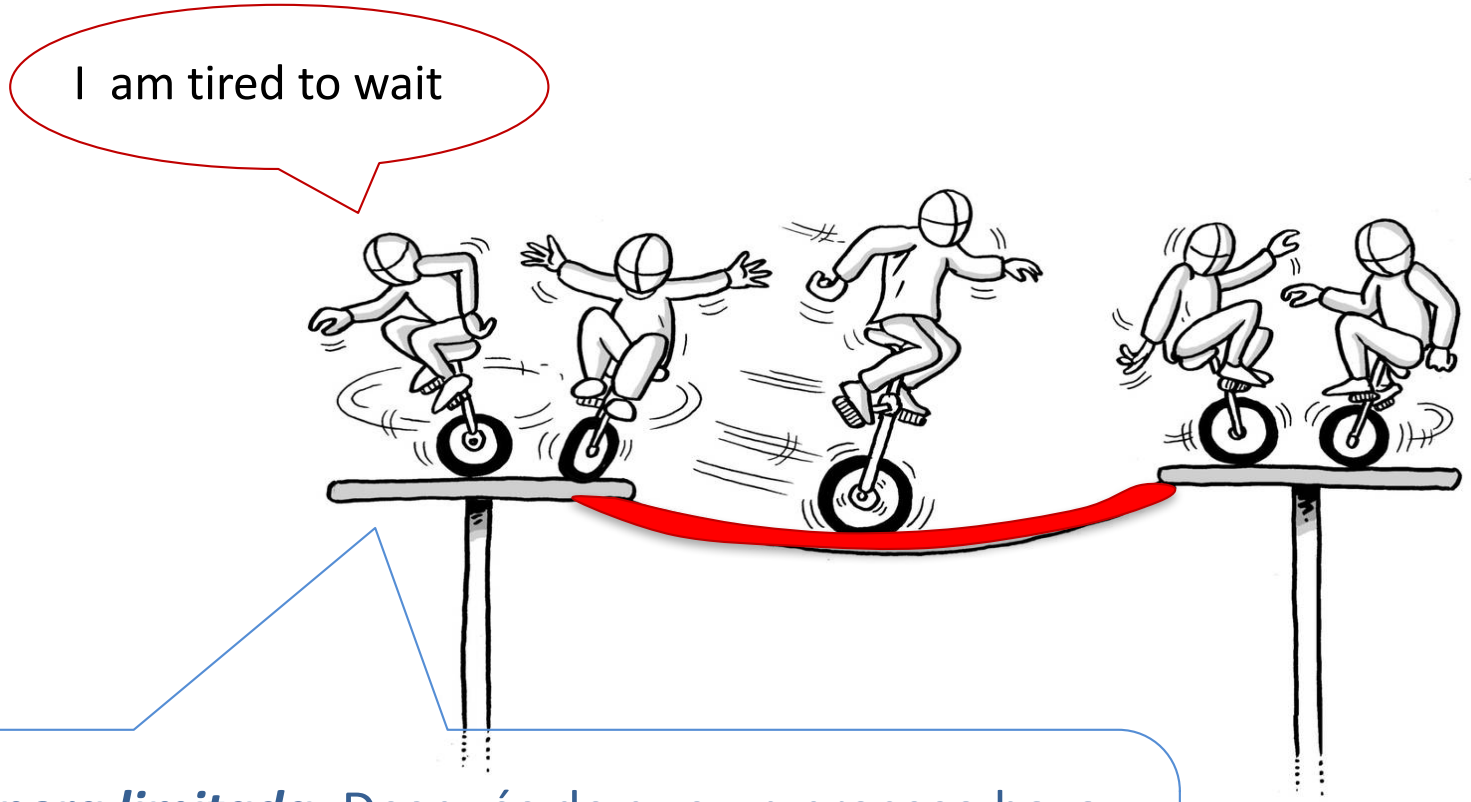
Exclusión mutua: si un proceso está en sección crítica, ningún otro proceso puede estar al mismo tiempo en su sección critica

- **Protocolo de acceso a sección critica** debe cumplir tres condiciones:



Progreso: si ningún proceso está en sección crítica y hay otros que desean entrar, entonces la decisión de qué proceso entrará a la sección crítica se toma en un tiempo finito y sólo depende de los procesos que desean entrar

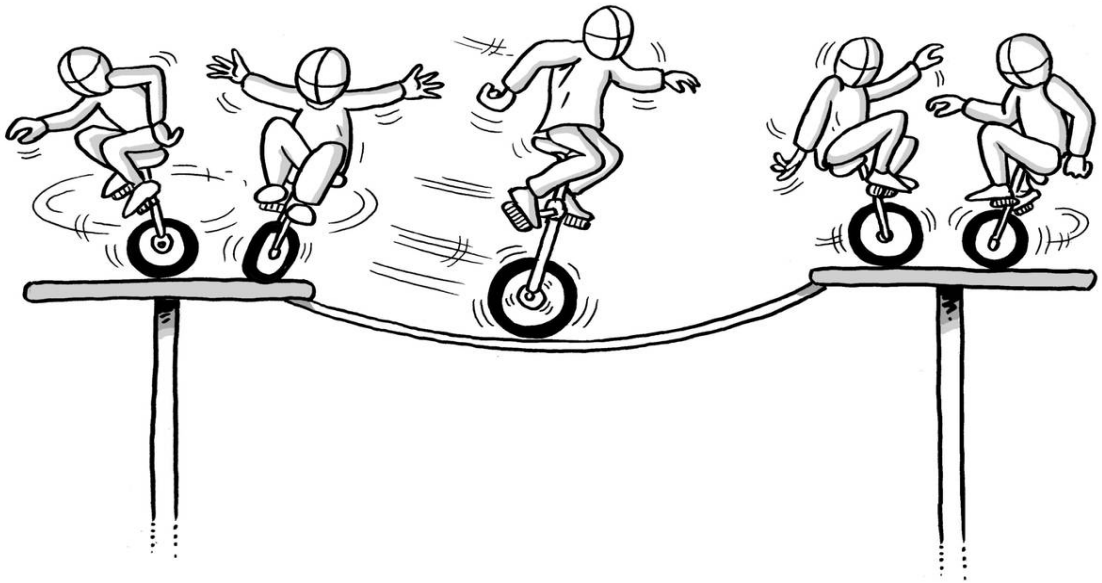
- **Protocolo de acceso a sección critica** debe cumplir tres condiciones:



Espera limitada: Después de que un proceso haya solicitado entrar en su sección crítica, existe un límite en el número de veces que se permite que otros procesos entren a sus secciones críticas

- Alternativas para implementar el protocolo de acceso a la sección crítica
  - Soluciones **Básicas**
    - Soluciones de tipo "**software**"→ El protocolo se implementa con código a nivel de usuario, sin mecanismos añadidos: Algoritmos de Dekker
    - Soluciones de tipo "**hardware**" →Se aprovechan instrucciones máquina especiales para implementar el protocolo
  - Soluciones con soporte del **sistema operativo**
    - El protocolo se implementa mediante mecanismos proporcionados por el sistema operativo, a través de **llamadas al sistema**
  - Soluciones con soporte del **lenguaje de programación** (construcciones lingüísticas)
    - Algunos lenguajes de programación poseen tipos de datos especiales que garantizan su acceso en exclusión mutua de forma automática
    - Ejemplos: Ada95, Java

- **Contenido**
  - El problema de la Sección Crítica
  - **Soluciones software**
  - Soluciones hardware
  - Espera activa
  - Ejercicios



- **Algoritmo básico**
  - Protocolo para varios hilos del mismo tipo, que se sincronizan mediante una variable global "llave" que indica si la sección crítica está ocupada

Protocolo de entrada

Protocolo de salida

```
void *hilo(void *p) {

  while(1) {
    while (llave == 1)
      /*bucle vacío*/ ;
    llave = 1;

    /* Sección crítica */

    llave = 0;

    /* Sección restante */
  }
}
```

(*)

  - Este protocolo
    - **No cumple la "exclusión mutua".** Si se produce un cambio de contexto en (*) es posible que varios hilos entren en su sección crítica

- **"Dekker nº 1", alternancia estricta**
  - **Protocolo para dos hilos**, que se sincronizan mediante una variable global "turno" inicializada a I o J que indica a qué hilo le toca ejecutar la sección crítica

```
void *hilo_I(void *p) {

  while(1) {
    while (turno != I)
      /*bucle vacío*/ ;

    /* Sección crítica */

    turno = J;

    /* Sección restante */
  }
}
```

```
void *hilo_J(void *p) {

  while(1) {
    while (turno != J)
      /*bucle vacío*/ ;

    /* Sección crítica */

    turno = I;

    /* Sección restante */
  }
}
```

  - Este protocolo
    - Sólo sirve para dos hilos. Necesidad de conocer a priori la cantidad de hilos a sincronizar.
    - **No cumple la condición de "progreso".** Impone una "velocidad relativa" entre los procesos.

- Existen **soluciones software** completamente correctas:

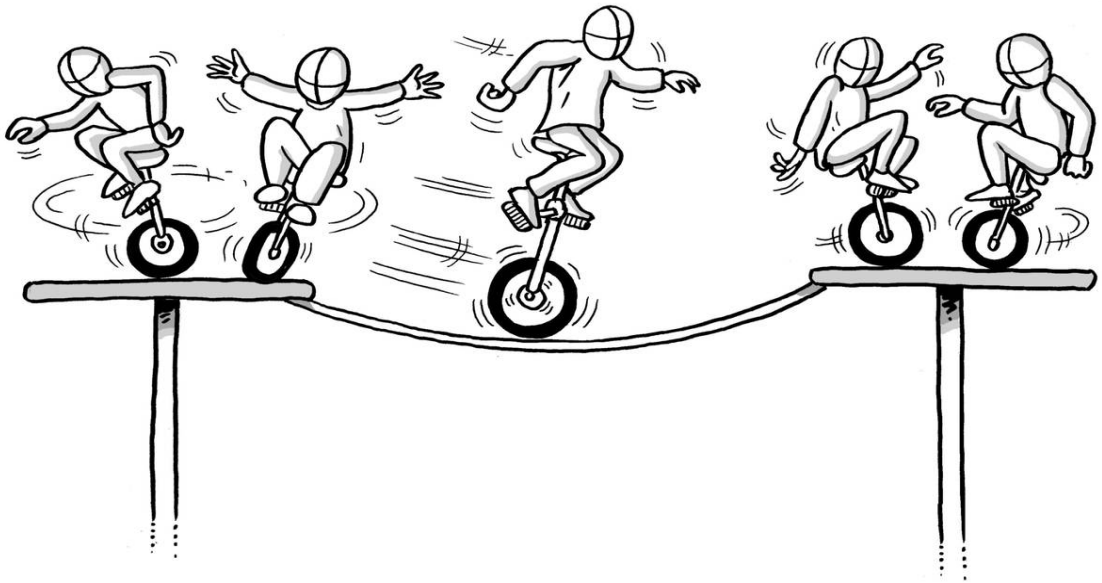
  Algoritmos de Dekker,

  Algoritmo de Peterson,

  Algoritmos de Lamport…

  – Se aplican en **sistemas distribuidos**, en general, cuando las actividades a sincronizar (hilos) no se ejecutan en el mismo procesador

- **Contenido**
  - El problema de la Sección Crítica
  - Soluciones software
  - **Soluciones hardware**
  - Espera activa
  - Ejercicios



- Soluciones **Hardware**
  - Inhibición de interrupciones
    - Requiere que los hilos se ejecuten en el mismo procesador.
  - Instrucciones **atómicas** para la manipulación de memoria
    - **Test-and-set**
    - **Intercambio**

- **Inhibición de Interrupciones**
  - Se realiza utilizando las instrucciones:
    - **DI (disable interrupts):** inhabilita interrupciones del procesador
    - **EI (enable interrupts):** habilita interrupciones
  - la **exclusión mutua** se consigue **inhibiendo los cambios de contexto** durante la ejecución de sección crítica, obligando así a que las secciones críticas se ejecuten de manera atómica:

```
...
DI;
/* Sección crítica */
EI;
/* Sección restante
*/
...
```

Esto es **mucho más de lo necesario**. Sólo hay que impedir que se ejecuten las secciones críticas de otros hilos, ya que sus secciones restantes sí se pueden ejecutar

  - Solución únicamente viable al **nivel del núcleo** del sistema operativo:
    - no es deseable dejar el control de las interrupciones en manos de los procesos de usuario, las instrucciones DI y EI sólo se pueden ejecutar en **modo privilegiado**

Soluciones hardware

- **“test-and-set” instrucción atómica**
  - La instrucción “test and set” permite evaluar y modificar una variable atómicamente en una sola instrucción máquina
  - La **especificación funcional** de la instrucción “test and set” es la siguiente:

```
int test_and_set (int *objetivo) {
  int aux;

  aux = *objetivo;
  *objetivo = 1;
  return aux;
}
```

Atómica

Una sola instrucción máquina

  - devuelve el valor de la variable “objetivo” (1 ó 0, indicando cierto o falso) y asigna 1 (cierto) a “objetivo”, todo ello de forma atómica

Atómica = Indivisible, no se puede interrumpir su ejecución

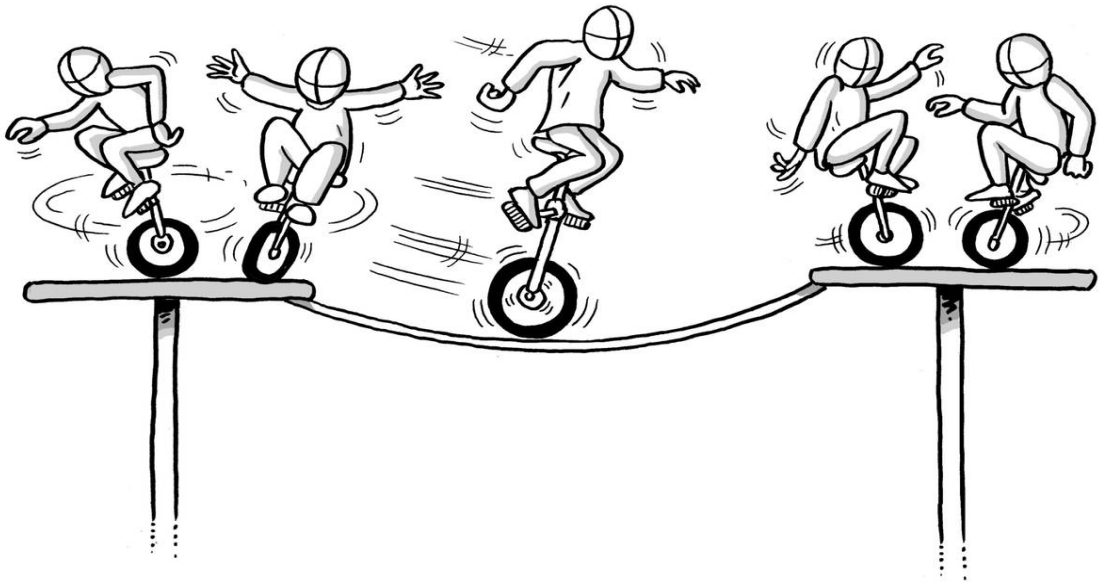
- **"test-and-set" instrucción atómica**
  - Protocolo de acceso a la sección crítica
    - N procesos/hilos que ejecutan el siguiente código,
    - Todos los hilos comparte la variable "**llave" que esta inicializada a 0** (falso)

```
void *hilo_i(void *p) {

while(1) {
     while (test_and_set(&llave))
       /*bucle vacio*/ ;

     /* Sección crítica */

     llave = 0; /*falso*/

     /* Sección restante */
  }
}
```

```
/* Global variable*/
int  llave=0;
```

  - Esta solución **cumple exclusión mutua y progreso**, pero **no cumple "espera limitada"**
    - El siguiente en ejecutar el "test_and_set" será quien entre en sección critica, y por tanto, dado que el protocolo de entrada no define ningún orden no es posible asegurar que todos los procesos que están esperando puedan entrar.

- **Contenido**
  - El problema de la Sección Crítica
  - Soluciones software
  - Soluciones hardware
  - **Espera activa**
  - Ejercicios



Espera activa

- Concepto de **espera activa**
  - Las soluciones anteriores (software y hardware tipo "test-and-set") comparten una característica denominada **espera activa**
  - El protocolo de entrada impide que un proceso entre en su sección crítica, haciendo que dicho proceso ejecute un **bucle vacío que consume tiempo de CPU**

ESPERA ACTIVA

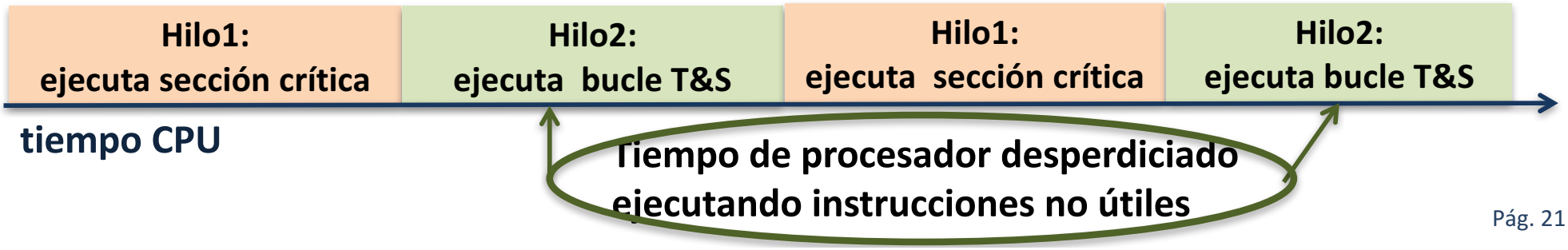
```
void *hilo_I(void *p) {

  while(1) {
    while (turno != I)
      /*bucle vacío*/ ;
    /* Sección crítica */
    turno = J;
    /* Sección restante */
  }
}
```

```
void *hilo_i(void *p) {

  while(1) {
    while test_and_set(&llave))
      /*bucle vacío*/ ;
    /* Sección crítica */
    llave = 0; /*falso*/
    /* Sección restante */
  }
}
```

Soluciones software
("Dekker nº 1", alternancia estricta)

Protocolo **"test-and-set" instrucción atómica**

- **Problemas de la espera activa**
  - La espera activa es la única forma de impedir que un proceso/hilo pase de un cierto punto de su código **sin recurrir al sistema operativo**
  - La espera activa no es deseable, porque
    - **Si la planificación de los hilos es por prioridades,**
      - Cuando un hilo queda "atrapado" en el protocolo de entrada teniendo más prioridad que el hilo que se encuentra ejecutando la sección crítica, entonces todos los hilos de la aplicación se quedarán **bloqueados para siempre** (inversión de prioridad).
    - Incluso en **escenarios de planificación menos problemáticos, como por ejemplo turno rotatorio,**
      - la espera activa presenta el problema de **infrautilización del procesador**, puesto que los hilos pueden consumir muchos cuantos de tiempo no haciendo nada más que ejecutando el bucle vacío



- Alternativas a la espera activa
  - Recurrir a la ayuda del **Sistema Operativo** para **suspender** al hilo cuando ejecuta el test_and_set y no consigue entrar en la sección crítica

```
while( test_and_set(&llave) ) {usleep(periodo);}
```

  - O simplemente que el hilo **abandone la CPU** cuando ejecuta el test_and_set y no consigue entrar en la sección crítica

```
while( test_and_set(&llave) ) {yield();}
```

  - Que el **Sistema Operativo ofrezca objetos** en los que un **proceso se pueda suspender** hasta **que otro proceso lo despierte:**

MUTEX Y SEMÁFOROS

• **Se implementan como llamadas al sistema.** El S.O. proporciona primitivas específicas para resolver el problema de la sección crítica

• **No se produce espera activa.** El S.O. puede detener procesos/hilos de forma eficiente, marcándolos como suspendidos en su PCB.

Espera por evento

Espera activa

- Ejemplo: Productor/consumidor
  - Solución hardware
    - test_and_set , instrucción atómica e indivisible

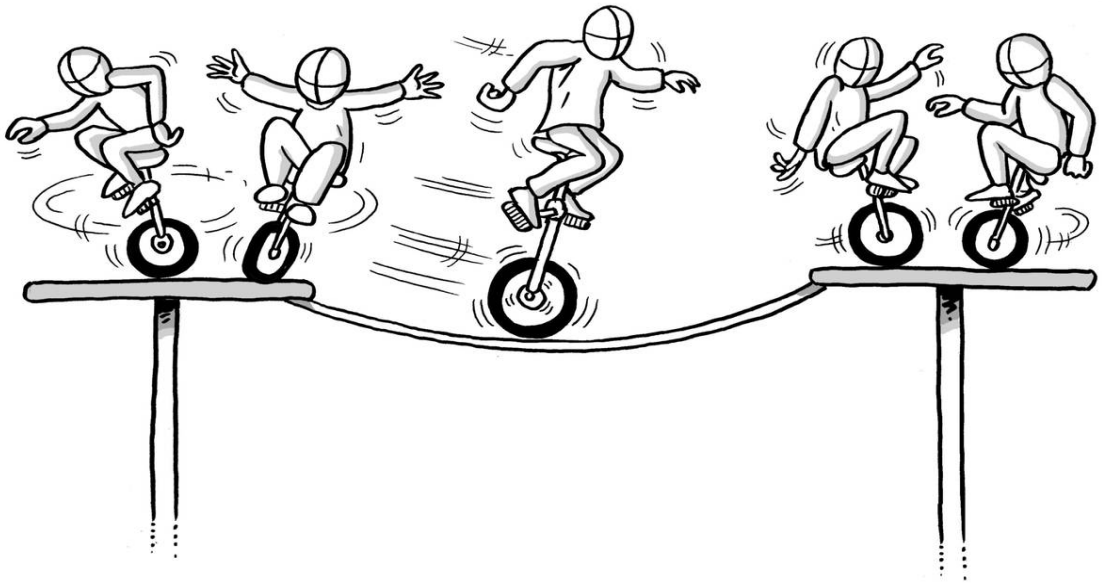
```
#define N  20
int  buffer[N];
int  entrada, salida, contador;
```

```
void *func_prod(void *p) {
  int item;

  while(1) {
    item = producir();

    while (test_and_set(&llave))
      /*bucle vacio*/ ;

    while (contador == N)
      /*bucle vacio*/ ;
    buffer[entrada] = item;
    entrada = (entrada + 1) % N;
    contador = contador + 1;

    llave = 0;
  }
}
```

```
void *func_cons(void *p) {
  int item;

  while(1) {
    while (test_and_set(&llave))
      /*bucle vacio*/ ;

    while (contador == 0)
      /*bucle vacio*/ ;
    item = buffer[salida];
    salida = (salida + 1) % N;
    contador = contador - 1;

    llave = 0;

    consumir(item);
  }
}
```

  - Este código tiene problemas y no funciona
    - ¿Dónde aparecen los problemas?

- **Contenido**
  - El problema de la Sección Crítica
  - Soluciones software
  - Soluciones hardware
  - Espera activa
  - **Ejercicios**



Ejercicios UT06.1:

Razone adecuadamente si el siguiente código es una buena solución al problema de sección crítica para dos hilos (hilo_0 e hilo_1).

Determine si cumplen las tres condiciones del protocolo de la sección crítica

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

¿ Exclusión mutua
Progreso
Espera limitada ?

```
hilo_i(void) {
  while ( 1 ) {

    sección_restante;

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

    sección_crítica;

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

Ejercicio UT06.2:

Compare las tres soluciones propuestas con test and set al problema de sección crítica y diga si son una solución al problema de la espera activa

```
/* Solución a */
void *hilo(void *p) {

  while(1) {
   while (test_and_set(&llave));
   /* Sección crítica */
   llave = 0;
   /* Sección restante */
  }
}
```

```
/* Solución b */
void *hilo(void *p) {

 while(1) {
 while (test_and_set(&llave)) usleep(1000);
   /* Sección crítica */
   llave = 0;
   /* Sección restante */
  }
}
```

```
#include <stdio.h>
/**Compartidas **/
  int llave=0;
```

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

 while(1) {
 while (test_and_set(&llave)) yield();
   /* Sección crítica */
   llave = 0;
   /* Sección restante */
  }
}
```

Nota: La llamada **yield()** permite a un proceso ceder lo que le resta de su cuanto de tiempo de CPU . De esta forma, el proceso que invoca yield() pasa a PREPARADO y libera la CPU, y da al planificador la oportunidad de seleccionar otro proceso para su ejecución.

Ejercicios UT06.3:

Observe el siguiente fragmento de código correspondiente a dos hilos que pertenecen al mismo proceso y se ejecutan concurrentemente. Indique qué recursos compartidos aparecen en el código y qué mecanismos se emplean para evitar las condiciones de carrera.

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

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

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

   tmp=V;
   tmp--;
   V=tmp;
   llave=0;
  }

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

Nota: Las variables y funciones que no están definidas dentro de las funciones agrega y resta son definidas como globales