Sistemas Operativos II

Procesos conceptos avanzados: Sincronización

Universidad Arturo Jauretche
Ingeniería Informática

Docentes:

Coordinador: Ing. Jorge Osio

Profesor: Ing. Eduardo Kunysz

Procesos cooperativos

Son aquellos que puede afectar o verse afectado por otros procesos que estén ejecutándose en el sistema.

- Pueden compartir espacio de direcciones
- Compartir datos a través de mensajes

Razones para procesos cooperativos

- **Compartir información:** varios usuarios pueden estar interesados en compartir información.
- **Acelerar cálculos:** atomizar tareas y ejecución en paralelo.
- **Modularidad:** construcción del sistema en forma modular.
- **Convivencia:** múltiples tareas o procesos ejecutándose al mismo tiempo.

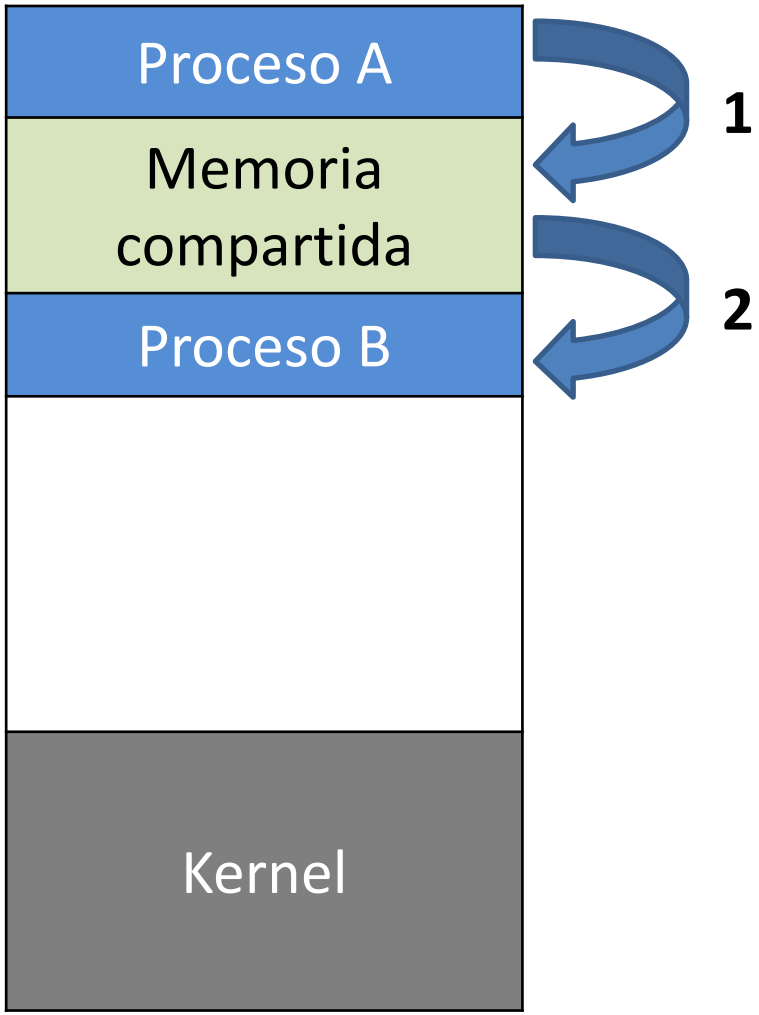
Memoria compartida

- Los procesos **intercambian** información leyendo y escribiendo en **zonas compartidas**.

- Es **más rápida** que el paso de mensajes. (tiempos de acceso a memoria).

- Las llamadas al kernel sólo son necesarias para establecer las zonas de memoria compartida.

Productor /Consumidor



- Un proceso **productor** genera información que consume un proceso **consumidor**.
- Debemos tener disponible un **buffer** de elementos que pueda llenar el **productor** y vaciar el **consumidor**.
- Deben estar **sincronizados** para que el consumidor no intente consumir si el productor no escribió aun

Productor / Consumidor
Tipo de buffer

- **Buffer no limitado:** el productor puede escribir todo el tiempo sin esperar por el consumidor

- **Buffer limitado:** el productor tiene que esperar si el buffer está lleno.

Buffer limitado

```
#define BUFFER_SIZE 10

typedef struct {
…
} item;

item
buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Se implementa como buffer circular.
- Punteros lógicos:
  - **in**: apunta a la siguiente posición libre.
  - **out**: apunta a la primera posición ocupada.

in == out → El buffer está **vacío**

((in+1)%BUFFER_SIZE) == out → El buffer está **lleno**

Permite tener como máximo **BUFFER_SIZE – 1** elementos

Proceso productor

- Se añade variable `counter` , inicializada en 0.
- `counter` se **incrementa** cada vez que se **añade** elemento al buffer
- `counter` se **decrementa** cada vez que se **elimina** elemento al buffer

```
while (true)
        {
        /* produce un elemento en nextProduced */
        while(counter == BUFFER_SIZE); //no hacer nada
                buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
        }
```

Proceso consumidor

```
while (true)
      {
      while(counter == 0); // no hacer nada
            nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;
      /*Consume el elemento que hay en
      nextConsumed*/
      }
```

Por separados los dos procesos son correctos, pero **no pueden funcionar concurrentemente**.

Problema concurrencia productor /consumidor

`counter++` en lenguaje de máquina

registro$_1$ = `counter`

registro$_1$ = registro$_1$ + 1

`counter` = registro$_1$

`Counter--` en lenguaje de máquina

registro$_2$ = `counter`

registro$_2$ = registro$_2$ - 1

`counter` = registro$_2$

Problema concurrencia productor /consumidor

La ejecución concurrente se realiza de forma secuencial:

T0:	productor	registro$_1$ = `counter`	[registro$_1$ = 5]
T1:	productor	registro$_1$ = registro$_1$ + 1	[registro$_1$ = 6]
T2:	consumidor	registro$_2$ = `counter`	[registro$_2$ = 5]
T3:	consumidor	registro$_2$ = registro$_2$ - 1	[registro$_2$ = 4]
T4:	productor	`counter` = registro$_1$	[`counter` = 6]
T5:	consumidor	`counter` = registro$_2$	[`counter` = 4]

Se llega al estado incorrecto con **`counter`** diferentes. Si cambia el orden de ejecución en T4 y T5 se invertiría el resultado. Este fenómeno se conoce como:

condición de carrera → **sincronización**

Sección crítica

- Denominamos sección crítica al conjunto de **recursos** a los que **se quiere acceder en exclusión mutua** (lo que significa que sólo puede acceder a un recurso un único proceso al mismo tiempo).

Ejemplo: contar el **numero de coches** que pasan por los dos carriles de una autopista.

El problema surge cuando los procesos-contadores intentan acceder a una variable compartida.

Sección crítica

- Solución: → **exclusión mutua**

"Cuando un proceso está en su sección crítica, ningún otro puede estar en la suya."

Para ello, adoptamos las **restricciones de Djkstra:**

- La solución debe ser independiente del HW o del número de procesos.
- No se puede realizar ninguna suposición sobre la velocidad relativa de los procesos.
- Cuando un proceso está fuera de su sección crítica no puede impedir a otros procesos entrar en sus respectivas secciones criticas.
- La selección del proceso que debe entrar en la sección crítica no puede posponerse indefinidamente.

Posibles soluciones al contador de coches

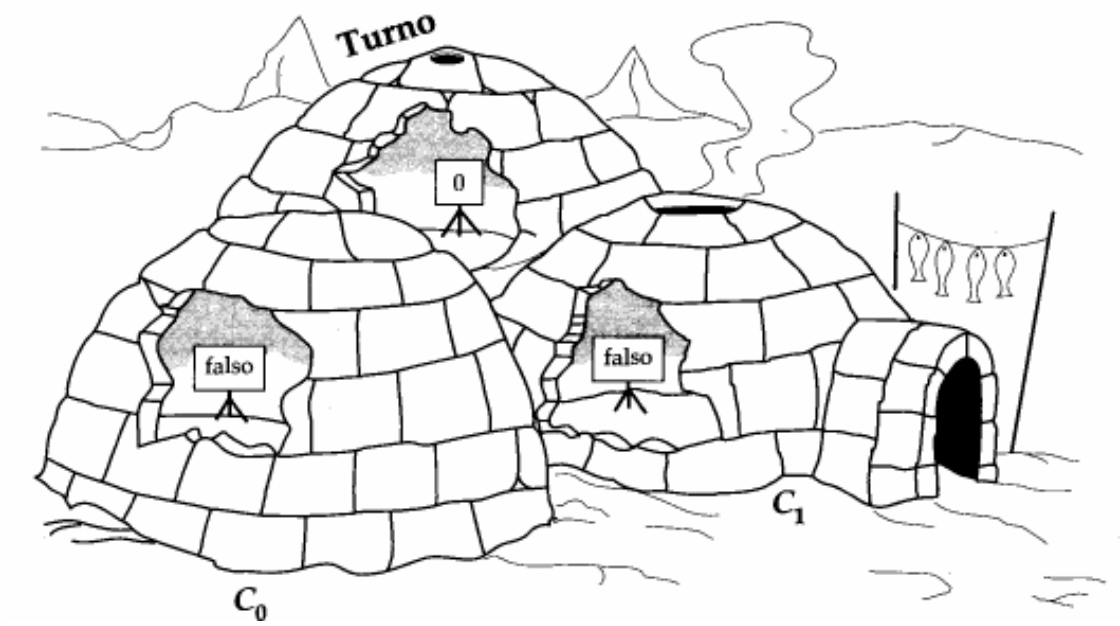
1. Asociar una variable booleana *libre* al acceso al recurso común (la sección crítica). El proceso accede si *libre=TRUE*
   - *No sirve porque si se pierde el control en libre=FALSE, los dos procesos accederán a la vez al recurso => indeterminismo*
2. *Un proceso pasa cuando libre= TRUE y el otro cuando libre = FALSE.*
   - *Solo funciona si la velocidad de ambos procesos está acompasada: si pasan dos coches por el carril izquierdo, sólo se podrá contar el segundo cuando pase un coche por el carril derecho (y ponga libre=TRUE).*
   - *Viola la tercera restricción de Djkstra*

Posibles soluciones al contador de coches

3. Si el segundo proceso quiere acceder a la sección crítica, le dejamos; en caso contrario, accede el primer proceso.
   - Se garantiza exclusión mutua.
   - Es posible que los dos procesos se den el turno el uno al otro y ninguno acceda al recurso común => viola la 4ta restricción de Djkstra.
4. La solución válida es el **algorítmo de Dekker**

Algoritmo de Dekker

- Se tiene tres iglúes:
- Uno **arbitro** con una pizzarra llamada **turno**
- Dos (C0 y C1) pertenecientes a cada proceso.

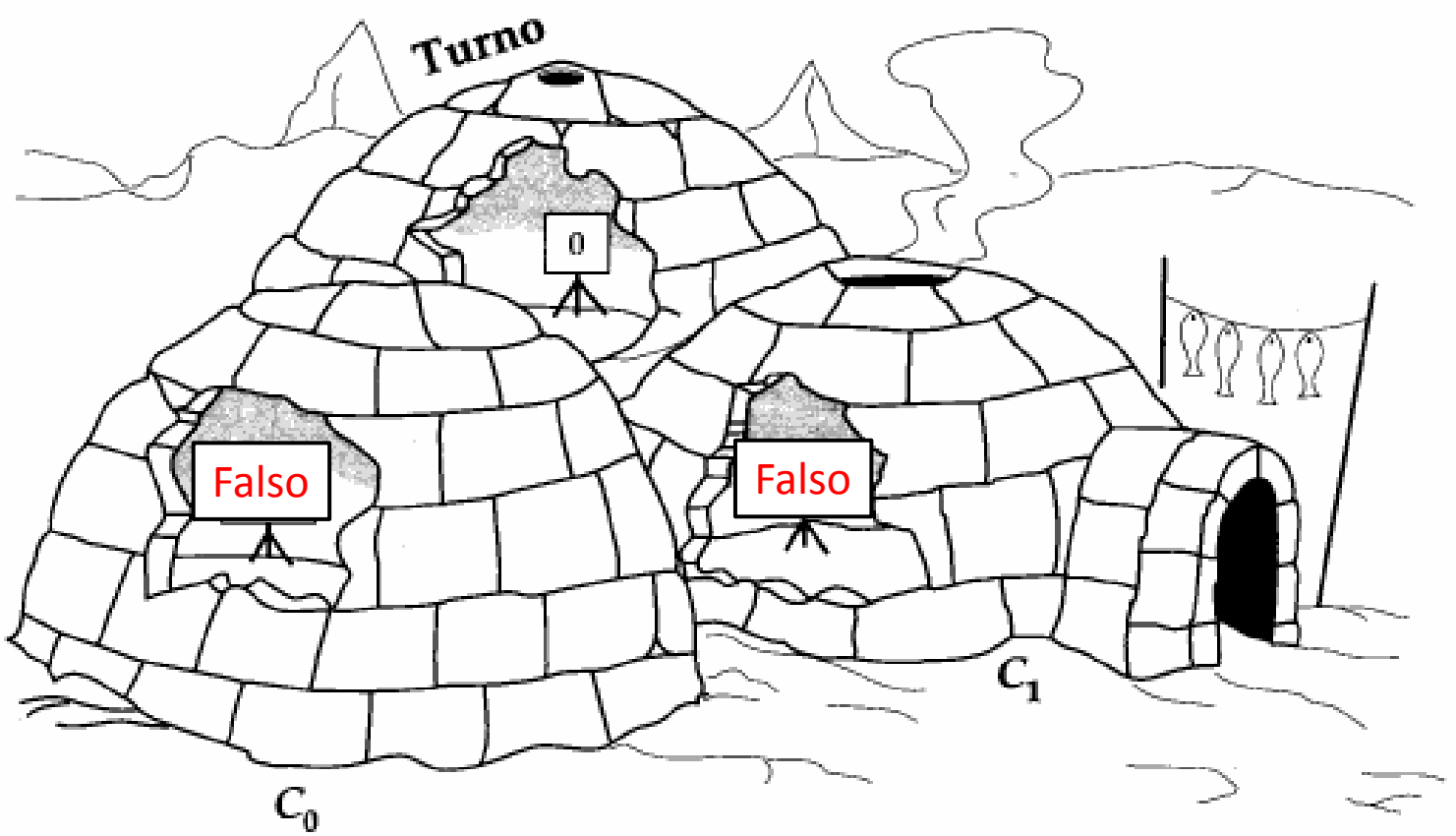


Cada proceso pondrá en sus pizarra :

Cierto: quiere acceder a la sección critica

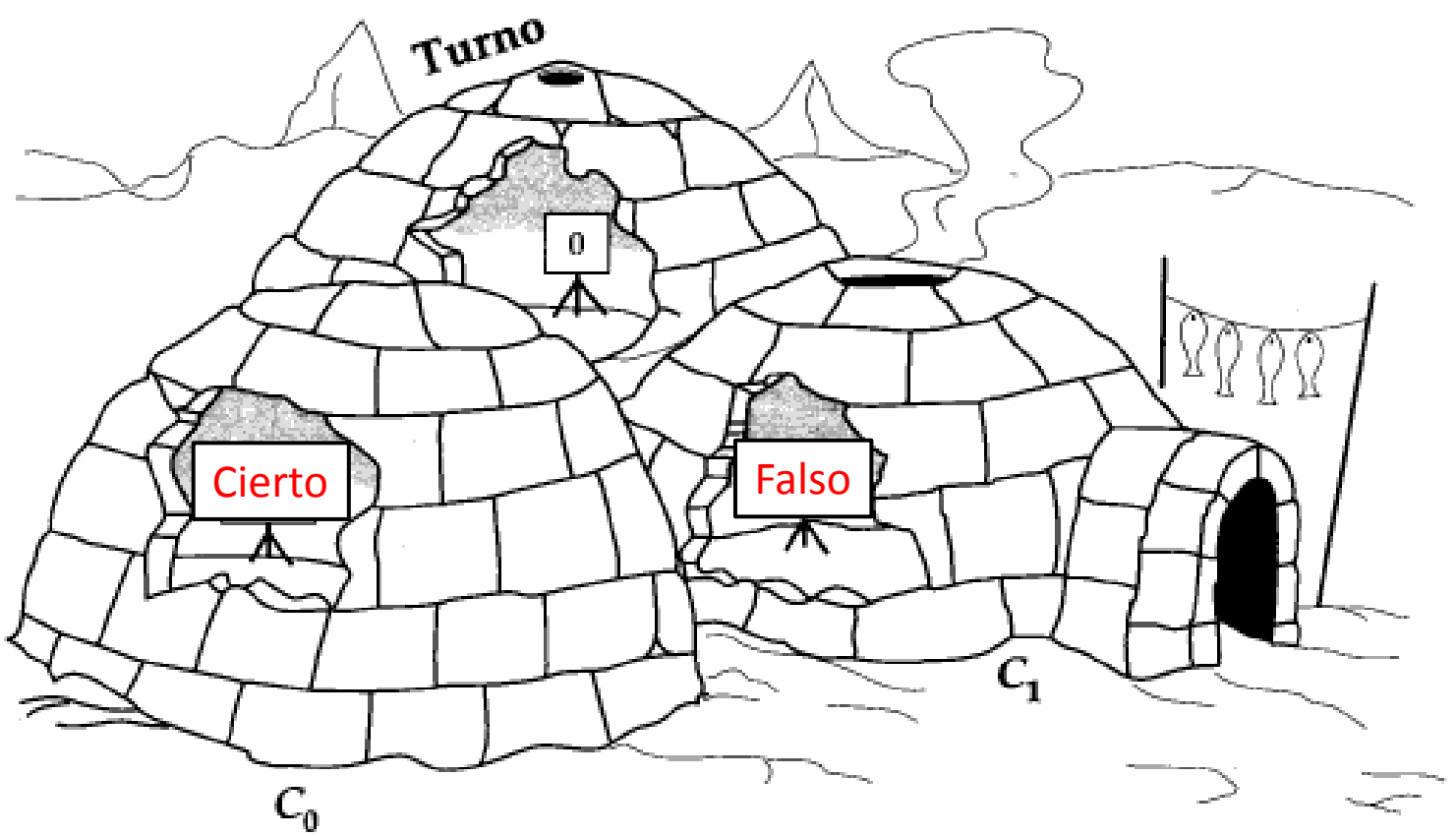
Falso: no quiere acceder a la sección crítica

Algoritmo de Dekker



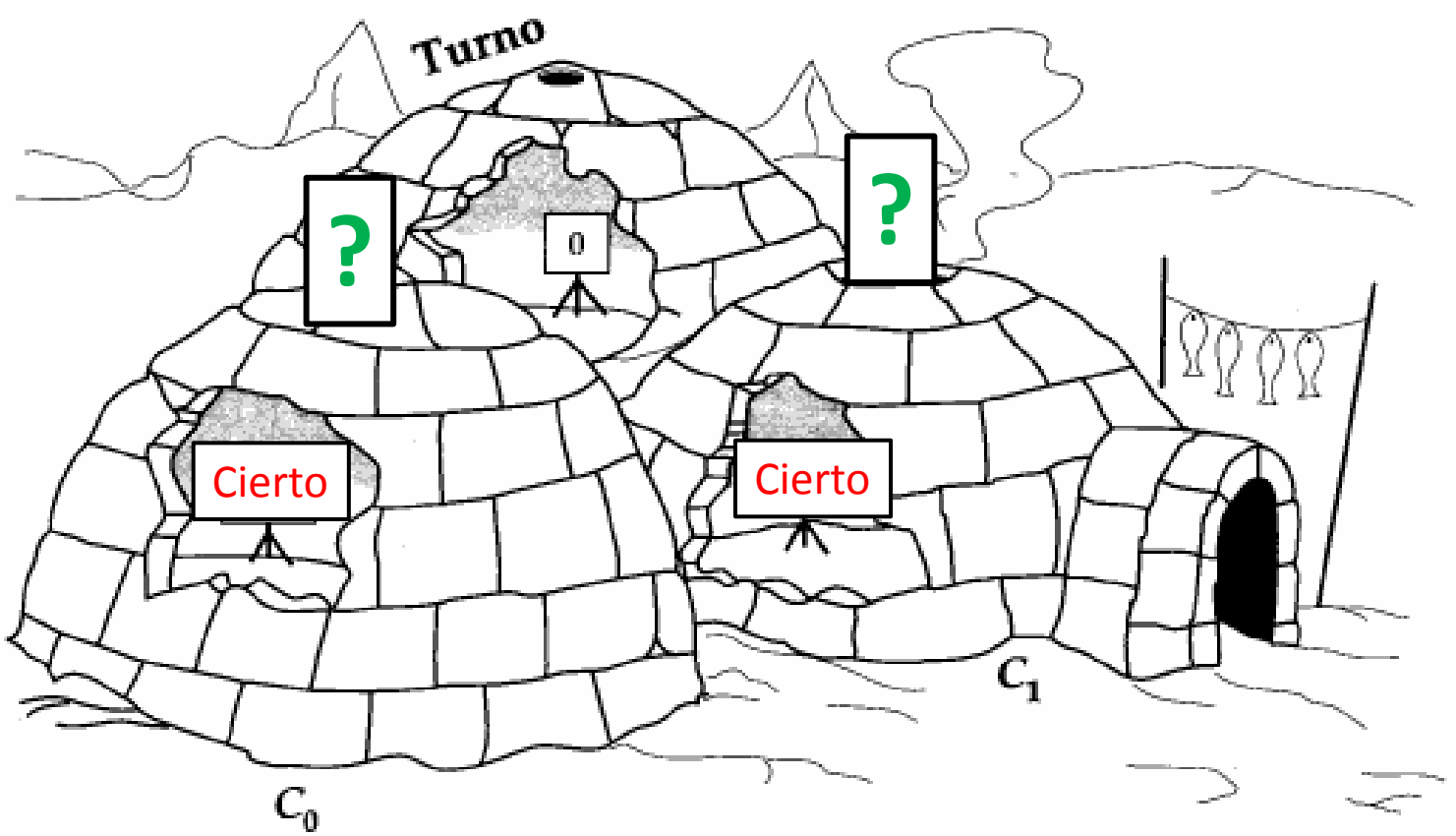
- Ninguno quiere entrar en la sección crítica

Algoritmo de Dekker



- C0 mira que C1 dice falso, entonces entra en la sección crítica. Cuando sale pone **falso**

Algoritmo de Dekker



- En este caso se debe ver el valor del arbitro. C1 en algún momento se enterará que el arbitro dice 0 y tendrá que poner **falso**

Algoritmo de Peterson

- Solución simple y elegante

```
#define FALSE 0
#define TRUE 1
#define N 2                 // número de procesos
int turno;                  // ¿de quién es el turno?
int interesado[N]; // arranca todos 0 (FALSE)
void entrar_region(int proceso); // el proceso es 0 o 1
{
  int otro;               // número del otro proceso
  otro = 1 - proceso; // el opuesto del proceso
  interesado[proceso] = TRUE; // muestra interés
  turno = proceso; // establece la bandera
  while (turno == proceso && interesado[otro] == TRUE)
}
void salir_region(int proceso) // proceso: sale RC
{
interesado[proceso] = FALSE; // indica que salió de RC
}
```

Algoritmo de Peterson

- Cada proceso llama a **entrar_region** con su propio número como parámetro.
- Esta función hará que espere si es necesario hasta que sea seguro.
- Una vez que termina de utilizar la región, llama a **salir_region** para indicar que ha terminado.

Herramientas de sincronización

- Funciones primitivas, implementadas de forma SW o HW, que ayudan a controlar la interacción entre procesos concurrentes:
  - **Sincronización:** Los procesos intercambian señales que controlan su avance.
  - **Comunicación:** Los procesos intercambian información.

Características de sincronización

- Características deseables:
  - Desde el punto de vista conceptual:
    - Simplicidad.
    - Generalidad.
    - Verificabilidad.
  - Desde el punto de vista de implementación:
    - Eficiencia.
- Tipos de herramientas de sincronización:
  - Nivel HW: acceso a recursos HW compartidos asegurando uso en exclusión mutua (por ejemplo, memoria y buses).
  - Nivel SW: ***TSL, XCHG***.

La instrucción TSL

- Solución SW con ayuda de hardware
- Instrucción de assembler indivisible

```
TSL registro, candado
```

- «candado»: variable compartida
  - 0: permite paso (cualquiera lo puede poner a 1)
  - 1: bloquea
- En microinstrucciones:

T0 : Bloquea el bus de memoria
T1: candado → Rx
T2: candado ← valor ≠ 0

- Para volverlo a poner a 0 se usa **`MOVE`**

Ejemplo de TSL

```
entrar_region:
TSL REGISTRO,CANDADO |copia candado Rx y  lo fija a 1
CMP REGISTRO,#0      |¿era candado cero?
JNE entrar_region    |si era distinto de cero, el
                     candado está cerrado, y se repite
RET                  |regresa al llamador; entra a
                     región crítica

salir_region:
MOVE CANDADO,#0      |almacena 0 en candado
RET                  |regresa al llamador
```

La Instrucción XCHG

- Variante a TSL, intercambia el contenido de dos ubicaciones en forma atómica.

```
entrar_region:
MOVE REGISTRO,#1          |coloca 1 en el registro
XCHG REGISTRO,CANDADO     |intercambia el contenido del
                          registro y la variable candado
CMP REGISTRO,#0           |¿era candado cero?
JNE entrar_region         |si era distinto de cero, el
                          candado está cerrado, y se repite
RET                       |regresa al que hizo la llamada;
                          entra a región crítica

salir_region:
MOVE CANDADO,#0           |almacena 0 en candado
RET                       |regresa al que hizo la llamada
```

Semáforos

- Herramienta de sincronización
- Es una variable (S) entera no negativa a la que, aparte de la inicialización, sólo se accede mediante dos operaciones atómicas estándar:
  - **Wait(S)** → P (Probar): **Si S > 0 → S=S - 1**
    Si S ≤ 0 espera
  - **Signal(S)** → V (Intercambiar): **S=S + 1**

Semáforos: propiedades

- Consta de tres partes:
  - Una variable entera interna **(S)** con un valor máximo N (No accesible para los procesos).
  - Una **cola de procesos** (no accesible para los procesos)
  - Wait y signal son **atómicas**
- Semáforos pueden ser:
  - Semáforo contador (dominio no restringido)
  - Semaforo binario (0 o 1) → **MUTEX**

Implementación de WAIT

- **WAIT(s):**

```
s := s – 1
if s < 0 then
        begin
                estado-proceso = espera;
                poner-proceso-en-cola_espera_semáforo;
        end;
```

Implementación SIGNAL

- **SIGNAL(s):**

```
s := s + 1
if s ≤ 0 then
        begin
        poner-proceso-en-cola_preparados;
        estado-proceso = activo;
end;
```

Si $s \leq 0 \Rightarrow$ se bloquean todos los procesos; si $s \geq 1 \Rightarrow$ no exclusión mutua.

El valor inicial y máximo de la variable s determinan la funcionalidad del semáforo.

MUTEX

- Semáforos binarios: (0 y 1) .
- Solución al problema de la exclusión mutua:
  - Un semáforo binario con ***s = 1***.
  - Antes de acceder a la sección crítica el proceso ejecuta ***wait(sem)***.
  - Al finalizar la sección crítica el proceso ejecuta ***signal(sem)***.

Semáforos de Paso

- Permiten implementar grafos de precedencia.
  - Para cada punto de sincronización entre dos procesos: semáforo binario con ***s = 0***.
  - El proceso que debe esperar ejecuta ***wait(sem)***.
  - Al alcanzar un punto de sincronización el otro proceso ejecuta **signal(sem)**.

Semáforos enteros y de condición

- ***N*** procesos accediendo a la sección crítica:
  - Semáforo con ***s = N***, siendo ***N*** el valor máximo permitido.
  - Antes de acceder a la sección crítica el proceso ejecuta ***wait(sem)***.
  - Al finalizar la sección crítica el proceso ejecuta ***signal(sem)***.

Semáforos enteros y de condición

- Sincronización entre procesos en función de una variable entera:
  - Semáforo cuya variable ***s*** está inicializada a ***N*** ó ***0*** (siendo ***N*** el valor máximo).
  - El proceso que debe esperar ejecuta ***wait(sem).***
  - Al alcanzar un punto de sincronización el otro proceso ejecuta ***signal(sem)***.
  - Ejemplo clásico: problema del productor-consumidor (con búfer limitado e ilimitado).

Solución Productor-Consumidor con buffer no limitado

- Programa principal que lanza dos procesos hijos y prepara semáforos:

```
PROGRAM P-C;
VAR
  buffer: ARRAY [N] of datos;
  s: SEMAFORO //en exclusión mutua
  vacio: SEMAFORO //de condición
BEGIN
  s=1;
  vacio=0;
  COBEGIN
    P; C;
  COEND
END
```

Solución Productor-Consumidor con buffer no limitado

```
PROCEDURE P;
BEGIN
 REPEAT
  Producir_Dato;
  WAIT(s);
  Dejar_Dato_en_Buffer;
  SIGNAL(s);
  SIGNAL(vacio);
 FOREVER;
END;
```

```
PROCEDURE C;
BEGIN
 REPEAT
  WAIT(vacio);
  WAIT(s);
  Extraer_Dato_del_Buffer;
  SIGNAL(s);
  Consumir_Dato;
 FOREVER;
END;
```

Solución Productor-Consumidor con buffer limitado (tamaño N)

- Programa principal que lanza dos procesos hijos y prepara semáforos:

```
PROGRAM P-C;
VAR
  buffer: ARRAY [N] of datos;
  s: SEMAFORO //en exclusión mutua
  vacio, lleno: SEMAFORO //de condición
BEGIN
  s=1; // exclusión mutua
  vacio=0; //de condición
  lleno=N; //de condición
  COBEGIN
   P; C;
  COEND
END
```

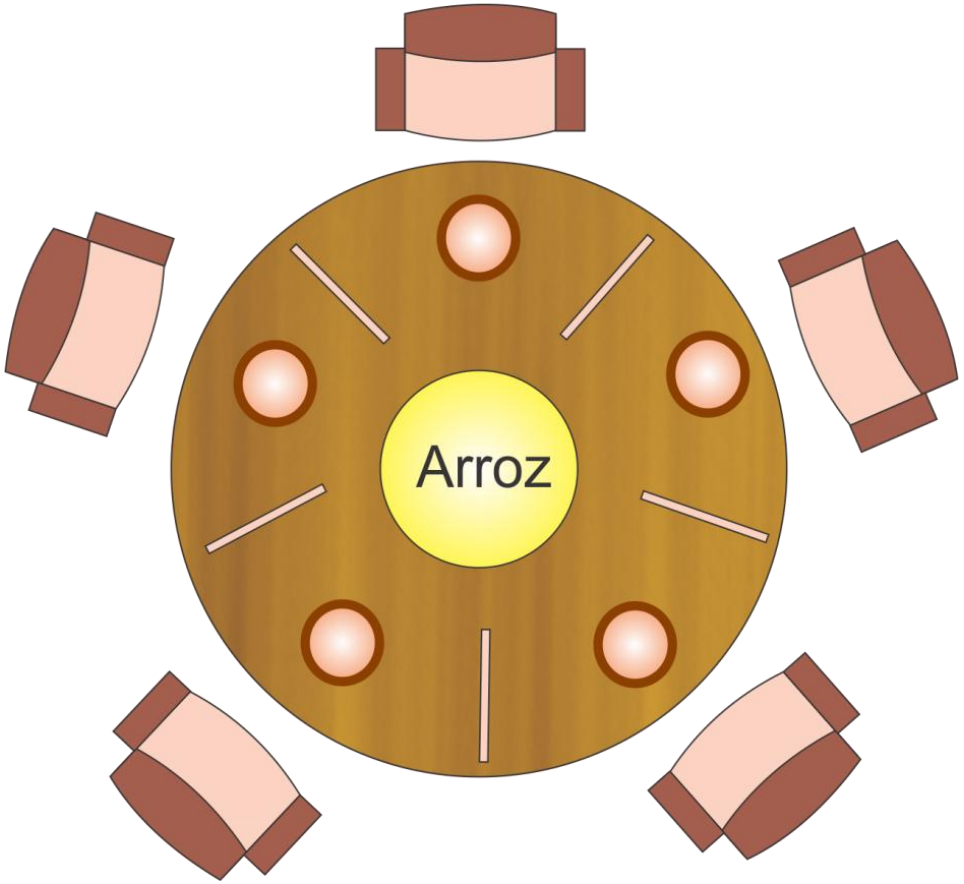
Solución Productor-Consumidor con buffer limitado (tamaño N)

```
PROCEDURE P;
BEGIN
  REPEAT
   Producir_Dato;
   WAIT(lleno);
   WAIT(s);
   Dejar_Dato_en_Buffer;
   SIGNAL(s);
   SIGNAL(vacio);
  FOREVER;
END;
```

```
PROCEDURE C;
BEGIN
  REPEAT
   WAIT(vacio);
   WAIT(s);
   Extraer_Dato_del_Buffer;
   SIGNAL(s);
   SIGNAL(lleno);
   Consumir_Dato;
  FOREVER;
END;
```

Problema de la cena de los filósofos

- Cinco filósofos chinos, comparten una mesa, cada uno con un plato y un palillo a cada lado.
- Cuando un filósofo piensa, no se relaciona con sus colegas.
- Cuando siente hambre y trata de tomar los palillos mas próximos a el.
- Un filósofo puede tomar solo un palillo a la vez.
- Pueden comer cuando agarran dos palillos
- Cuando termina, coloca de nuevo los palillos sobre la mesa.



Recurso compartido: los palillos

¿Posible? solución al problema de los filósofos

```
PROGRAM CENA-FILOSOFOS;
VAR
 palillos: ARRAY [0:4] of SEMAFORO;
BEGIN
for i=0:4 palillos[i]=1; // exclusión mutua
COBEGIN
for i=0:4 FILOSOFO[i];
COEND
END
```

Se lanzan 5 procesos FILOSOFO[i] y se generan 5 semáforos: uno por cada palillo

¿Posible? solución al problema de los filósofos

- Un bloque de varios *WAIT* no es indivisible; un solo *WAIT* sí lo es.
- Solución **inválida**: 5 filósofos pueden bloquear un solo palillo y ¡nadie come!

```
PROCEDURE FILOSOFO[i];
BEGIN
 REPEAT
  Pensar;
  WAIT(palillos[i]);
  WAIT(palillos[i+1 mod 5]);
  Servirse_y_comer;
  SIGNAL(palillos[i]);
  SIGNAL(palillos[i+1 mod 5]);
 FOREVER;
END;
```

- **<u>Solución correcta</u>**:
  - «comedor virtual» con 4 comensales (1 filósofo siempre come).
  - Semáforos utilizados: en exclusión mutua (*palillos*, inicializados a 1) y de condición (*comedor*, inicializado a 4).

Monitores

- Las técnicas vistas hasta ahora de sincronización, tiene **riesgos de interbloqueos** y errores de **condiciones de carrera**.

- Un monitor es una **colección** de **procedimientos, variables y estructuras** de datos que se agrupan en un tipo especial de **módulo** o paquete

- Son objetos con características especiales. (Programados en alto nivel: Pascal, Modula-2,3).

Monitores: Características

- Las **variables locales** solo accesibles para procedimientos del monitor y no para procedimientos externos.

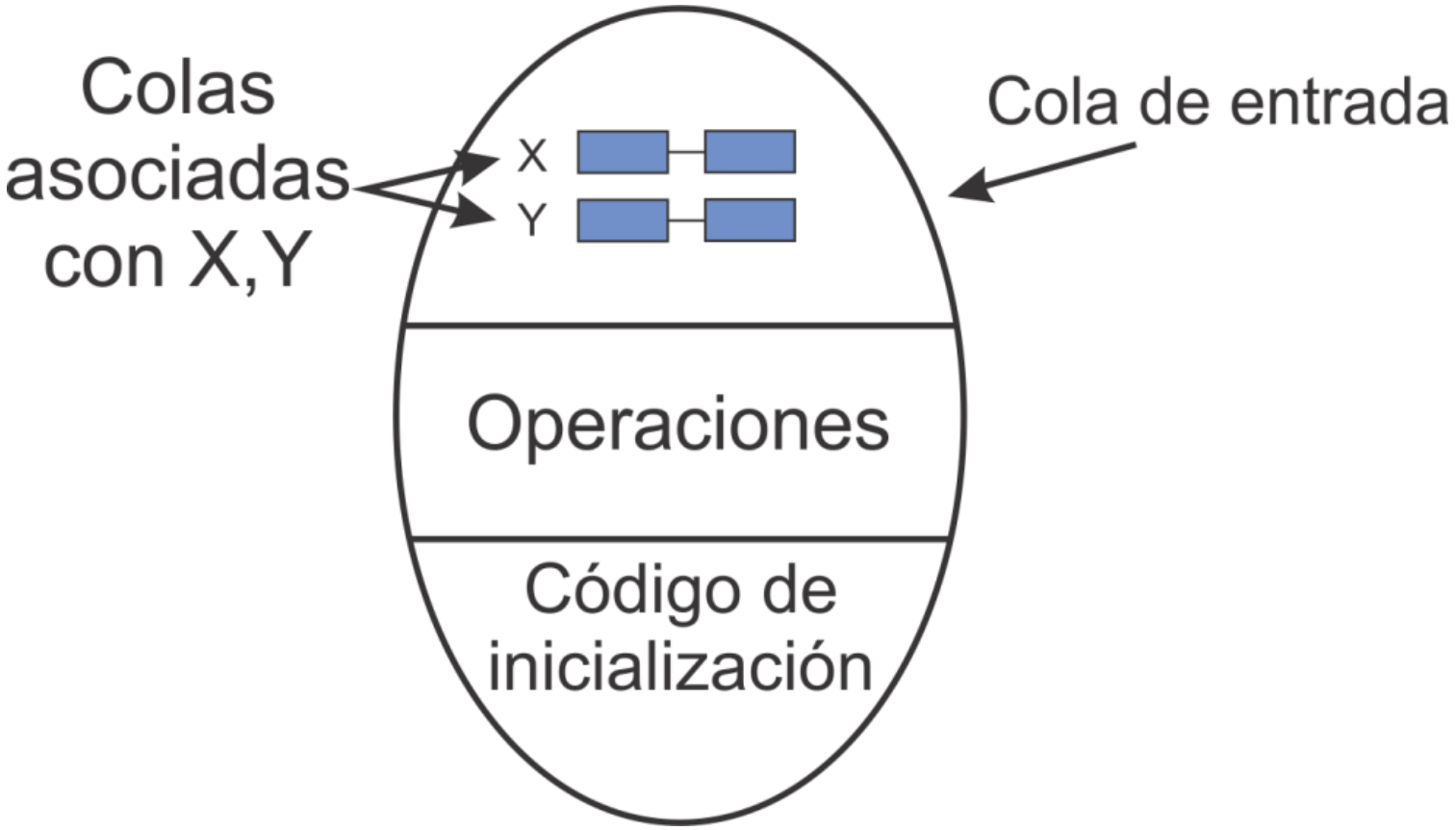
- Un proceso entra en el monitor invocando a uno de sus procedimientos.

- Solo **un proceso** puede **invocar al monitor**. Los demás quedan suspendidos hasta que el monitor esté disponible

Sintaxis de un monitor

```
monitor ejemplo
      integer i;
      condition c;
      procedure productor();
      .
      .
      .
      end;
      procedure consumidor();
      .          .          .
      end;
end monitor;
```

Estructura de monitor



Consideraciones sobre monitores

- Asegura que solo un proceso a la vez puede estar activo dentro del monitor.
- El programador no necesita codificar explícitamente.
- Se agrega el constructor "condition" para sincronizar (es un método y una estructura)

Monitor sencillo
Ejemplo: Simulación de semáforos

Monitor simulación de semáforos

```
ocupado     : boolean;
no-ocupado  : condición;
procedure P (v);
        begin
        if ocupado then Wait (no-ocupado);
        ocupado = V;
        end;
procedure V;
        begin
        ocupado = F;
        Signal (no-ocupado);
        end;
begin (* Cuerpo del monitor *)
        ocupado = F;
end; (* fin monitor *)
```

Monitor sencillo
Ejemplo: Simulación de semáforos

```
tarea T1;
      begin
      P;
      Región Crítica;
      V;
      end;
tarea T2;
      begin
      P;
      Región Crítica;
      V;
      end;
Programa-Principal
      Begin;
            cobegin T1, T2; coend
      end.
```

Monitores
Ejemplo: Productor Consumidor

```
MAX = ......;
monitor M;
buffer : Array (0..MAX-1);
in, out, n; enteros;
buff_lleno, buff_vacío: condición;
procedure Almacenar (v);
      begin
      if n = MAX then Wait (buff_vacío);
      buffer (in) = v;
      in = (in + 1) mod MAX;
      n = n + 1;
      Signal (buff_lleno)
      end;
```

Monitores
Ejemplo: Productor Consumidor

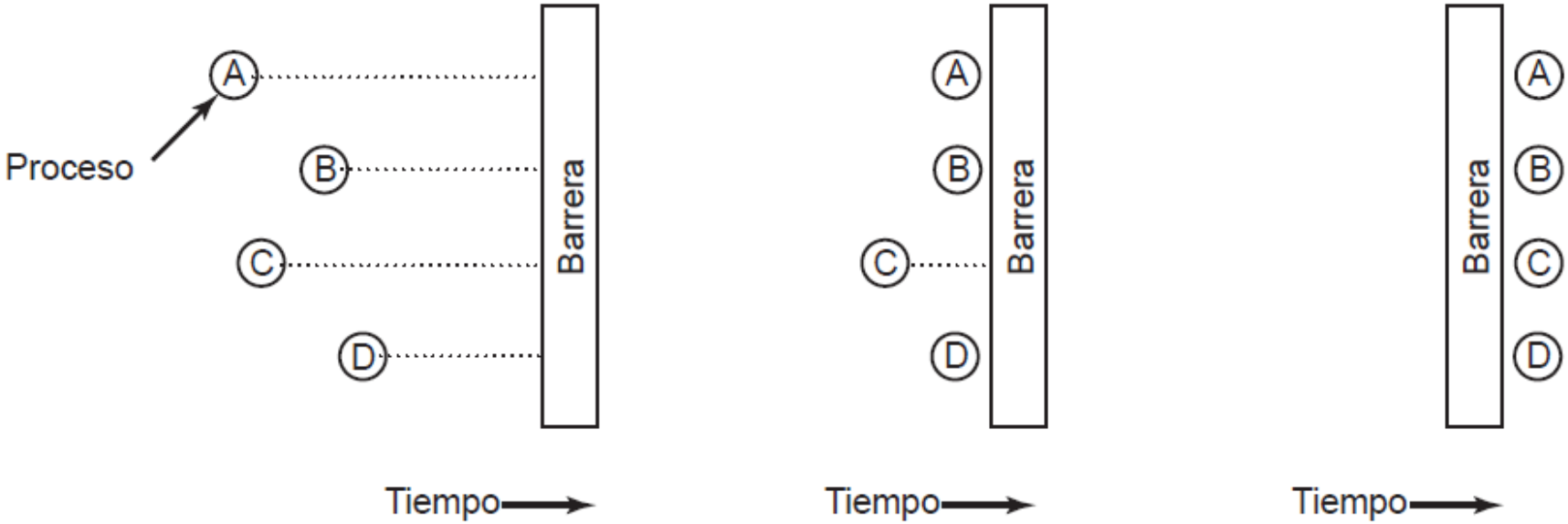
```
procedure Retirar (v);
begin
      if n = 0 then Wait (buff_lleno);
      v = buffer (out);
      out = (out + 1) mod MAX;
      n = n - 1;
      Signal (buff_vacio)
      end;
begin (* Cuerpo del monitor *)
      in, out, n = 0;
end; (* fin monitor *)
```

Monitores
Ejemplo: Productor Consumidor

```
procedure Productor;
      begin
      v = "dato producido"
      Almacenar (v)
      end;
procedure Consumidor;
      begin
      Retirar (v);
      Hacer algo con v
      end;
begin Programa-Principal
      Begin;
            cobegin
            Productor;
            Consumidor
            coend
      end.
```

Barreras

- Mecanismo para sincronizar grupos de procesos
- Cuando un proceso llega a la barrera, se bloquea hasta que todos los procesos lleguen



Bibliografía

- Cap 3, 6 Silberschatz "Fundamentos de sistemas operativos"
- Cap 2, Tanembaum "Sistemas Operativos modernos"
- Cap 4, William Stallings "Sistemas Operativos"