SINCRONIZACIÓN DE PROCESOS

DEPARTAMENTO DE CIENCIAS E INGENIERÍA DE LA COMPUTACIÓN

UNIVERSIDAD NACIONAL DEL SUR

AGENDA

1. Fundamentos
2. El Problema de la Sección Crítica
3. Solución a la sección crítica para 2 procesos (Algoritmo de Peterson) y para n procesos (Algoritmo del Panadero)
4. Soluciones por Hardware
5. Soluciones por Software
    1. Locks
    2. Semáforos
    3. Monitores
6. Problemas Clásicos
7. Ejemplos de Sincronización en los Sistemas Operativos

AGENDA

1. **Fundamentos**
2. El Problema de la Sección Crítica
3. Solución a la sección crítica para 2 procesos (Algoritmo de Peterson) y para n procesos (Algoritmo del Panadero)
4. Soluciones por Hardware
5. Soluciones por Software
    1. Locks
    2. Semáforos
    3. Monitores
6. Problemas Clásicos
7. Ejemplos de Sincronización en los Sistemas Operativos

FUNDAMENTOS

- El acceso concurrente a datos compartidos puede resultar en inconsistencias.
- Mantener la consistencia de datos requiere mecanismos para asegurar la ejecución ordenada de procesos cooperativos.
- Caso de análisis: problema del buffer limitado. Una solución, donde todos los N buffers son usados, no es simple.
  - Considere la siguiente solución

DATOS COMPARTIDOS

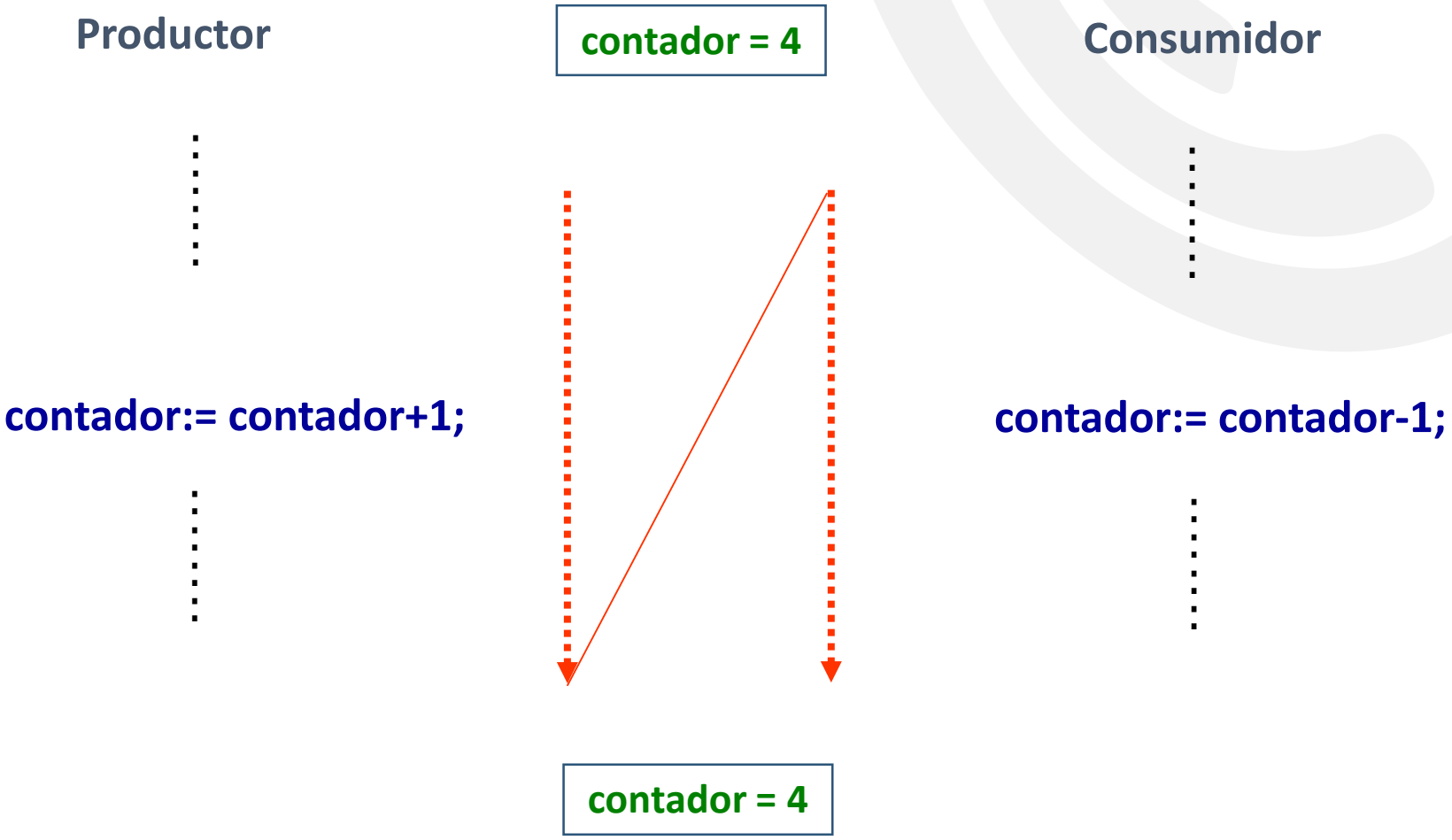
```
type item = … ;
var buffer array [0..n-1] of item;
in, out: 0..n-1;
contador : 0..n;
in, out, contador := 0;
```

PRODUCTOR-CONSUMIDOR

Proceso Productor	Proceso Consumidor

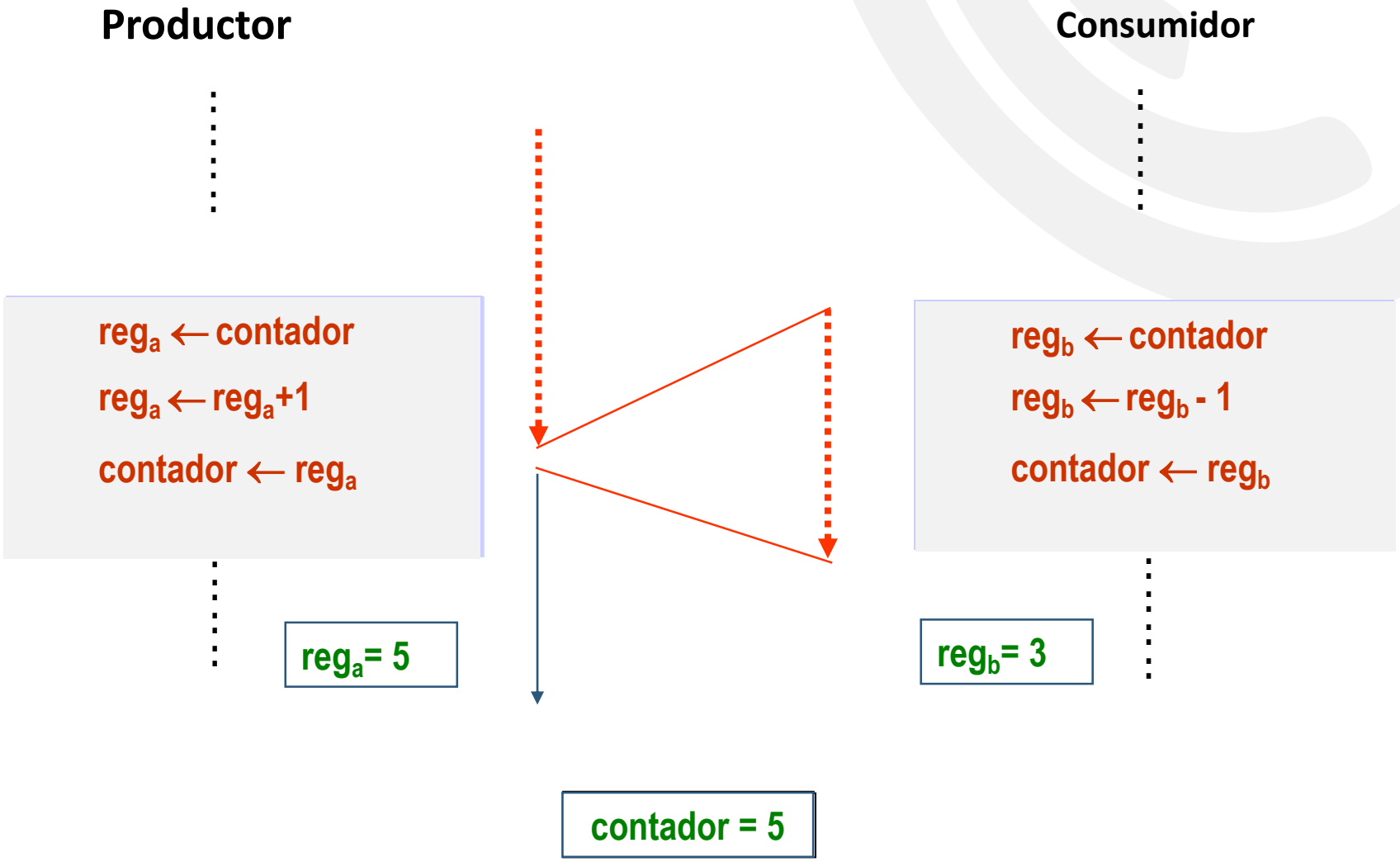
repeat
...
produce un item en *nextp*
...
while *contador* = n **do** no-op;
buffer [*in*] := *nextp*;
in := *in* + 1 **mod** *n*;
contador := *contador* +1;
 	
until false;
	repeat
	
	while *contador* = 0 **do** *no-op*;
	nextc := *buffer* [*out*];
	out := *out* + 1 **mod** *n*;
	contador := *contador* – 1;
	
	...
	consume el item en *nextc*
	...
	
	until *false*;

PROBLEMA



Luego de una operación de Productor y otra de Consumidor …
contador permanece invariante

PROBLEMA


Productor
Consumidor
$reg_a \leftarrow contador$
$reg_a \leftarrow reg_a+1$
$contador \leftarrow reg_a$
$reg_b \leftarrow contador$
$reg_b \leftarrow reg_b - 1$
$contador \leftarrow reg_b$
$reg_a = 5$
$reg_b = 3$
contador = 5

PROBLEMA

Situaciones dónde puede ocurrir este problema.

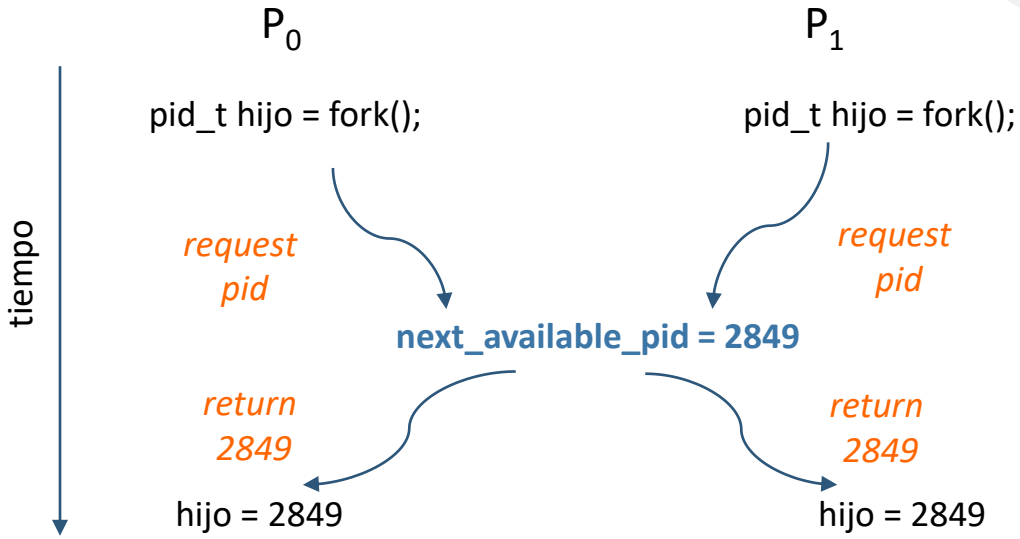
- Utilizando una variable compartida
- Ejecutando un conjunto de sentencias sobre variables compartidas
- Actualizando una estructura de datos, como por ejemplo una lista enlazada

PROBLEMA

¿QUÉ OCURRE CON EL KERNEL?

PROBLEMA

- Procesos P_0 y P_1 están creando un proceso hijo utilizando la llamada al sistema **fork()**.
- El kernel tiene la variable **next_available_pid** la cual representa el próximo identificador disponible para un proceso (pid).



- Podría ocurrir que el mismo pid sea asignado a dos procesos diferentes.

CONDICIÓN DE CARRERA

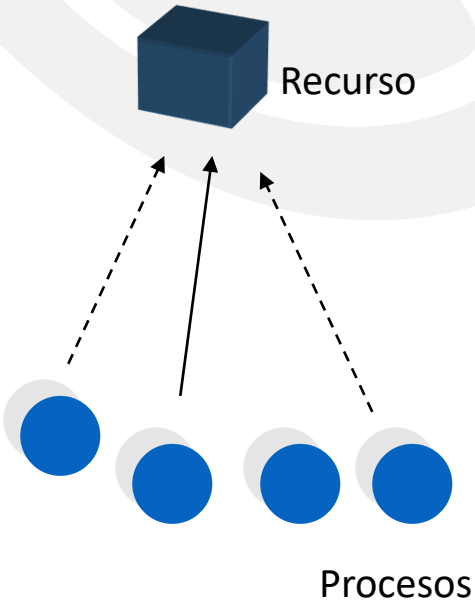
- **Condición de carrera** – Es la situación donde varios procesos acceden y manejan datos compartidos concurrentemente. El valor final de los datos compartidos depende de que proceso termina último.
- Para prevenir las condiciones de carrera, los procesos concurrentes cooperativos deben ser **sincronizados**.

AGENDA

1. Fundamentos
2. **El Problema de la Sección Crítica**
3. Solución a la sección crítica para 2 procesos (Algoritmo de Peterson) y para n procesos (Algoritmo del Panadero)
4. Soluciones por Hardware
5. Soluciones por Software
    1. Locks
    2. Semáforos
    3. Monitores
6. Problemas Clásicos
7. Ejemplos de Sincronización en los Sistemas Operativos

PROBLEMA DE LA SECCIÓN CRÍTICA

- n procesos todos compitiendo para usar datos compartidos
- Cada proceso tiene un segmento de código llamado ***sección crítica***, en la cual los datos compartidos son accedidos
- Problema – asegurar que cuando un proceso está ejecutando en su sección crítica, no se le permite a otro proceso ejecutar en su respectiva sección crítica.



SOLUCIÓN AL PROBLEMA DE LA SECCIÓN CRÍTICA

CONDICIONES PARA UN BUEN ALGORITMO

1. **Exclusión Mutua**
2. **Progreso**
3. **Espera Limitada**

- Asuma que cada proceso ejecuta a velocidad distinta de cero.
- No se asume nada respecto a la velocidad relativa de los n procesos.

RESOLUCIÓN DEL PROBLEMA

Estructura general del proceso P_i

repeat

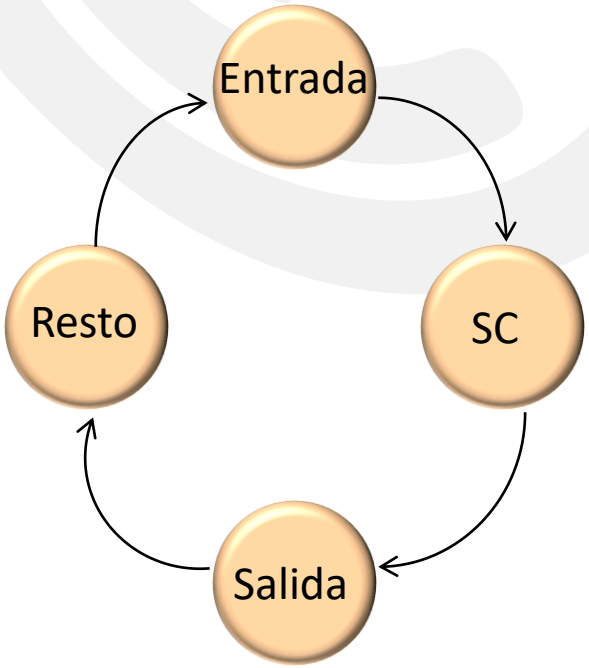
protocolo de entrada

sección crítica (SC)

protocolo de salida

sección resto

until *falso*



- Los procesos pueden compartir algunas variables comunes para sincronizar sus acciones.

AGENDA

1. Fundamentos
2. El Problema de la Sección Crítica
3. **Solución a la sección crítica para 2 procesos (Algoritmo de Peterson) y para n procesos (Algoritmo del Panadero)**
4. Soluciones por Hardware
5. Soluciones por Software
    1. Locks
    2. Semáforos
    3. Monitores
6. Problemas Clásicos
7. Ejemplos de Sincronización en los Sistemas Operativos

SOLUCIÓN DE PETERSON PARA 2 PROCESOS

DATOS COMPARTIDOS

int turno;
boolean flag[2]; incializado en *false*

Proceso P_i

repeat

flag [*i*] := *true*;
turno := *j*;
while (*flag* [*j*] and *turno* = *j*) **do** *no-op*;

sección crítica

flag [*i*] := *false*;

sección resto

until *false*;

- Alcanza las propiedades de un buen algoritmo; resuelve el problema de la sección crítica para dos procesos.

ALGORITMO PARA N PROCESOS – ALGORITMO DEL PANADERO

Sección crítica para *n* procesos

- Antes de entrar en su sección crítica, el proceso recibe un número. El poseedor del número más chico entra en su sección crítica.
- Si los procesos P_i y P_j reciben el mismo número, si i < j, entonces P_i es atendido primero; sino lo es P_j .
- El esquema de numeración siempre genera números en orden incremental de enumeración;

  p.e., 1,2,3,3,3,3,4,5...

ALGORITMO PARA N PROCESOS – ALGORITMO DEL PANADERO

Notación orden lexicográfico (ticket #,id proceso #)

- (a,b) < (c,d) si a < c o si a = c y b < d
- max (a0,…, an-1) es un número k, tal que k ≥ ai para i = 0, …, n – 1

DATOS COMPARTIDOS

var *choosing*: **array** [0..*n* – 1] **of** *boolean*;

number: **array** [0..*n* – 1] **of** *integer*,

Las estructuras de datos son inicializadas a false y 0 respectivamente.

ALGORITMO PARA N PROCESOS – ALGORITMO DEL PANADERO

```
repeat
    choosing[i] := true;
    number[i] := max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] := false;
    for j := 0 to n – 1
        do begin
            while choosing[j] do no-op;
            while number[j] ≠ 0
                and (number[j],j) < (number[i], i) do no-op;
        end;
        sección crítica
    number[i] := 0;
        sección resto
until false;
```

AGENDA

1. Fundamentos
2. El Problema de la Sección Crítica
3. Solución a la sección crítica para 2 procesos (Algoritmo de Peterson) y para n procesos (Algoritmo del Panadero)
4. **Soluciones por Hardware**
5. Soluciones por Software
    1. Locks
    2. Semáforos
    3. Monitores
6. Problemas Clásicos
7. Ejemplos de Sincronización en los Sistemas Operativos

SINCRONIZACIÓN POR HARDWARE

- Monoprocesador – pueden deshabilitarse las interrupciones

- Las computadoras modernas proveen instrucciones especiales que se ejecutan atómicamente

  Atómico = no-interrumpible

  - Por verificación de una palabra de memoria y su inicialización. Por ejemplo: ***test-and-set***
  - Por intercambio de dos palabras de memoria. Por ejemplo: **swap**

EXCLUSIÓN MUTUA CON TEST-AND-SET

DATO COMPARTIDO

var *lock*: *boolean* (*inicialmente* ***false***)

Proceso P_i

repeat

while *Test-and-Set* (*lock*) **do** *no-op*;

sección crítica

lock := *false*;

sección resto

until *false*;

función ***Test-and-Set*** (**var** target: *boolean*): *boolean*;
begin
Test-and-Set := *target*;
target := *true*;
end;

EXCLUSIÓN MUTUA CON SWAP

La variable booleana compartida es inicializada en ***false***; Cada proceso tiene una variable local booleana *key*

- Solución:

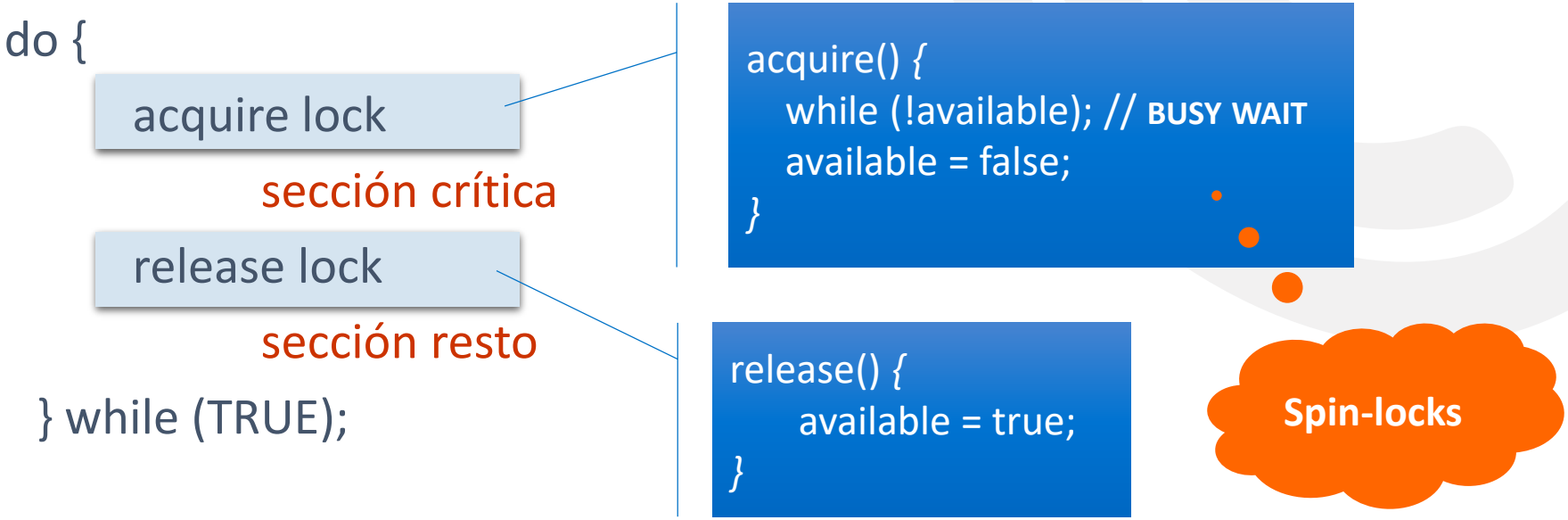
```
while (true)  {
        key = true;
        while ( key == true)
                Swap (&lock, &key );

                sección crítica

        lock = false;

                sección restante

}
```

```
void Swap (boolean *a, boolean *b)
    {
        boolean temp = *a;
        *a = *b;
        *b = temp:
    }
```

AGENDA

1. Fundamentos
2. El Problema de la Sección Crítica
3. Solución a la sección crítica para 2 procesos (Algoritmo de Peterson) y para n procesos (Algoritmo del Panadero)
4. Soluciones por Hardware
5. **Soluciones por Software**
    1. **Locks**
    2. **Semáforos**
    3. **Monitores**
6. Problemas Clásicos
7. Ejemplos de Sincronización en los Sistemas Operativos

EXCLUSIÓN MUTUA USANDO LOCKS



mutex locks – se utilizan específicamente para la exclusión mutual

SEMÁFOROS

Semáforo

Es una herramienta de sincronización.

- Semáforo ***S*** se define como una variable entera
- Dos operaciones standard modifican el valor de un semáforo ***S***
  wait() y **signal()**
- Puede ser accedido solo por dos operaciones indivisibles (*deben ser* atómicas):

wait (***S***)

```
while S≤ 0 do no-op;
S := S – 1;
```

signal (***S***)

```
S := S + 1;
```

SEMÁFORO - HERRAMIENTA GENERAL DE SINCRONIZACIÓN

- Semáforo de **Cuenta (Contador)** – el valor entero puede tener un rango sobre un dominio sin restricciones.
- Semáforo **Binario** – el valor entero puede tener un rango solo entre 0 y 1.
- Un semáforo puede utilizarse para alcanzar exclusión mutua

```
Semáforo S;    //  inicializado en 1
wait (S);
        Sección Crítica
 signal (S);
```

SEMÁFORO - HERRAMIENTA GENERAL DE SINCRONIZACIÓN

- Para sincronización
  - Ejecute *B* en P_j solo después que *A* ejecute en P_i
  - Use el semáforo *flag* inicializado a 0
  - Código:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

IMPLEMENTACIÓN DEL SEMÁFORO

- Debe garantizar que dos procesos no puedan ejecutar **wait ()** y **signal ()** sobre el mismo semáforo al mismo tiempo

- La implementación se convierte en el problema de la sección crítica donde el **código del wait** y **el signal** son la sección crítica.
  - Podemos tener ahora espera ocupada en la implementación de la sección crítica porque:
    - El código de implementación es corto
    - Poca espera ocupada si la sección crítica está raramente invocada

- Note que las aplicaciones pueden pasar y gastar mucho tiempo en secciones críticas, entonces no es una buena solución utilizar semáforos implementados con espera ocupada.

IMPLEMENTACIÓN DE SEMÁFORO SIN ESPERA OCUPADA

- Con cada semáforo hay asociada una cola de espera.
  Cada entrada en dicha cola tiene dos datos:
  - valor (de tipo entero)
  - puntero al próximo registro en la lista

- Dos operaciones:
  - **block** – ubica el proceso invocando la operación en la apropiada cola de espera.
  - **wakeup** – remueve uno de los procesos en la cola de espera y lo ubica en la cola de listos.

IMPLEMENTACIÓN DE SEMÁFORO SIN ESPERA OCUPADA

- Las operaciones del semáforo se definen como

***wait*(*S*):**

```
S.value := S.value – 1;
if S.value < 0
   then begin
          agregue este proceso a S.L;
          block;
   end;
```

***signal*(*S*):**

```
S.value := S.value + 1;
if S.value ≤ 0
   then begin
          remueva un proceso P de S.L;
          wakeup(P);
    end;
```

IMPLEMENTACIÓN DE *S* CON SEMÁFOROS BINARIOS

- Estructuras de datos:

  var *S1: binary-semaphore; S2: binary-semaphore;*
  S3: binary-semaphore; C: integer;

- Inicialización:

  S1 = *S3* = 1; *S2* = 0; *C* = valor inicial del semáforo *S*

operación ***wait***

```
wait(S3);
wait(S1);
C := C – 1;
if C < 0
then begin
        signal(S1);
        wait(S2);
end
else signal(S1);
signal(S3);
```

operación ***signal***

```
wait(S1);
C := C + 1;
if C ≤ 0 then signal(S2);
signal(S1);
```

Cuidado con el mal uso de los semáforos

INTERBLOQUEO E INANICIÓN

- **INTERBLOQUEO** – dos o más procesos están esperando indefinidamente por un evento que puede ser causado por solo uno de los procesos que esperan.
- Sean S y Q dos semáforos inicializados a 1

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
⋮	⋮
signal(S);	signal(Q);
signal(Q);	signal(S);

- **INANICIÓN** – bloqueo indefinido. Un proceso no puede ser removido nunca de la cola del semáforo en el que fue suspendido.
- **INVERSIÓN DE PRIORIDADES**

PROBLEMA

- Realizar la sincronización de la siguiente secuencia de ejecución. Cada proceso está asociado con una letra diferente.

ABCABCABCABCABCABCABC…….

```
semaphore semA =1
semaphore semB = 0
semaphore semC = 0
```

La solución está asociada con la inicialización

Proceso A	Proceso B	Proceso C
repeat	repeat	repeat
wait(semA)	wait(semB)	wait(semC)
Tarea A	Tarea B	Tarea C
signal(semB)	signal(semC)	signal(semA)
until false	until false	until false

SEMÁFOROS

- Incorrecto uso de las operaciones sobre semáforos:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitir el wait (mutex) o el signal (mutex) (o ambos)

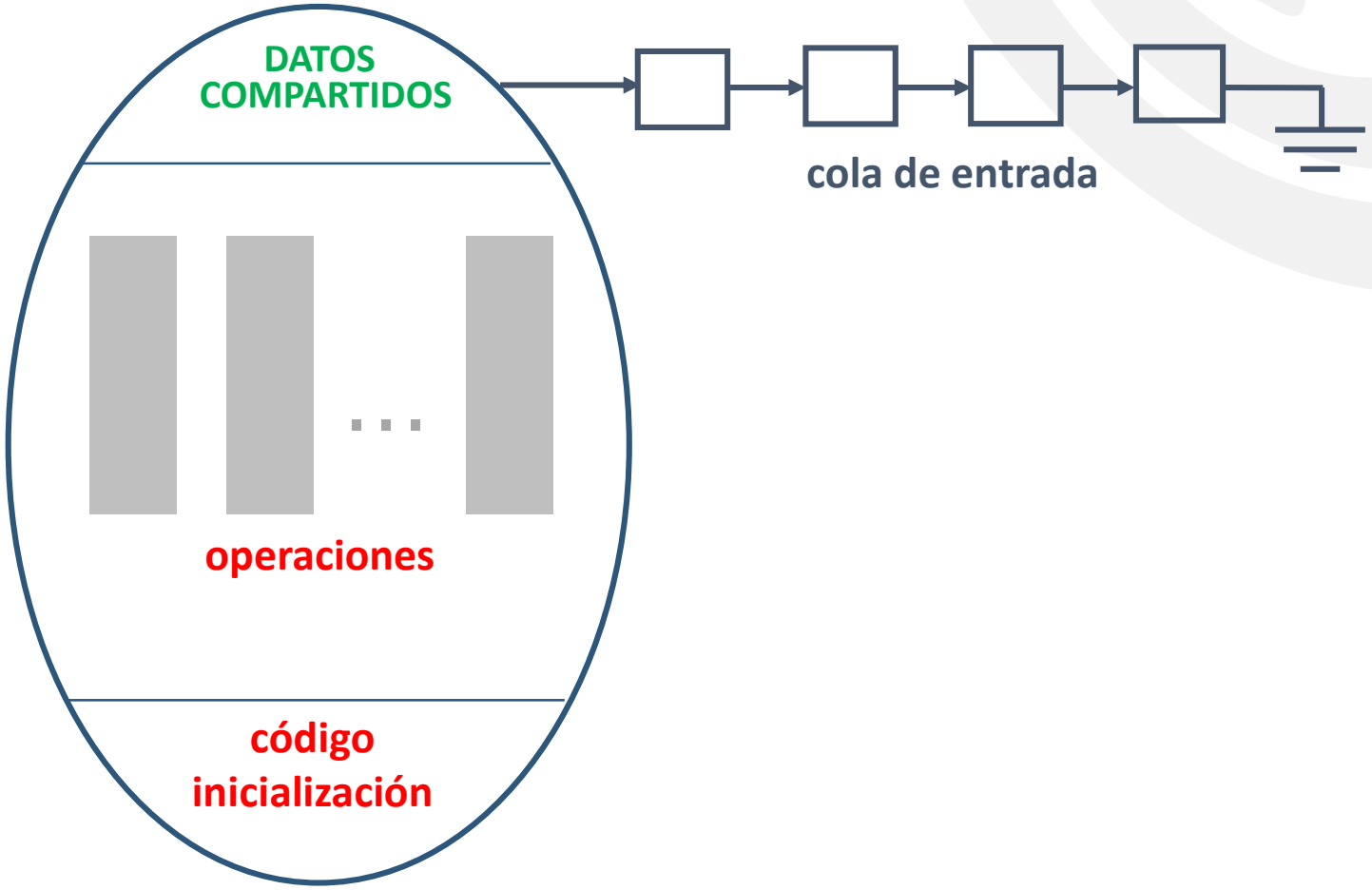
- Extensión
  - Incorporación de la operación ***wait-try*** sobre un semáforo.

MONITORES

Es un constructor de sincronización de alto nivel que permite compartir en forma segura un tipo de dato abstracto entre procesos concurrentes.

```
monitor monitor-nombre
{
  // declaración de variables compartidas

  procedure P1 (…) { …. }

  procedure Pn (…) {……}

  código de inicialización(…) { … }
}
```

VISTA ESQUEMÁTICA DE UN MONITOR



VARIABLES DE CONDICIÓN

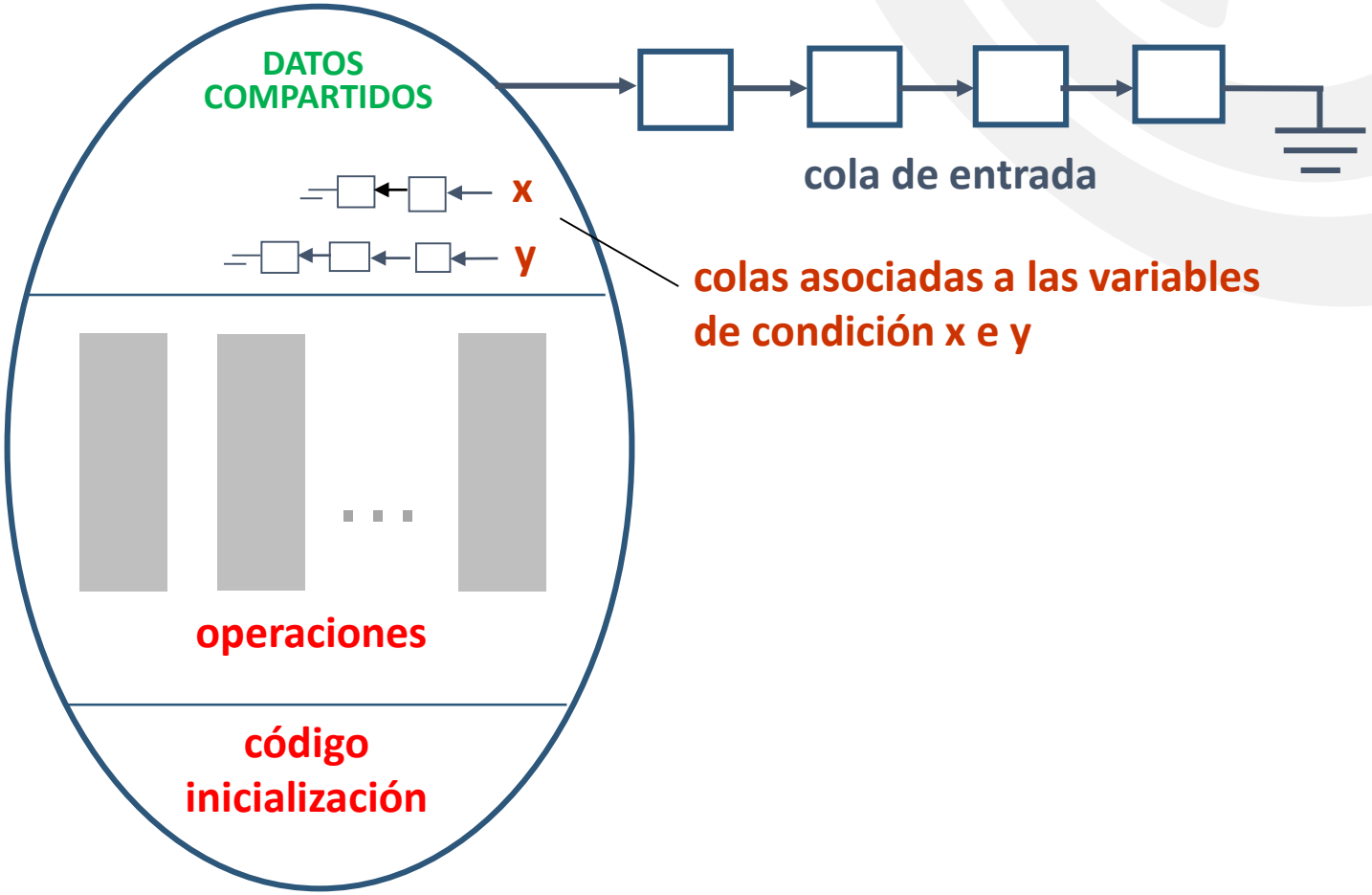
Las variables de condición son un mecanismo que se utiliza para la sincronización.

Por ejemplo para definir dos variables de condición

Condition x, y

- Dos operaciones sobre una variable de condición
  - **x.wait ()** – el proceso que invoca esa operación es suspendido
  - **x.signal ()** – reinicia uno de los procesos (si hay alguno) que invocó **x.wait ()**

MONITOR CON VARIABLES DE CONDICIÓN



AGENDA

1. Fundamentos
2. El Problema de la Sección Crítica
3. Solución a la sección crítica para 2 procesos (Algoritmo de Peterson) y para n procesos (Algoritmo del Panadero)
4. Soluciones por Hardware
5. Soluciones por Software
    1. Locks
    2. Semáforos
    3. Monitores
6. **Problemas Clásicos**
7. Ejemplos de Sincronización en los Sistemas Operativos

PROBLEMAS CLÁSICOS DE SINCRONIZACIÓN

- Problema del Buffer Limitado
- Problema de Lectores y Escritores
- Problema de los Filósofos Cenando

PROBLEMA DEL BUFFER LIMITADO

DATOS COMPARTIDOS

type *item* = …

var *buffer* = …

full, empty : semáforo de conteo;

mutex: semáforo binario; (o como mutex-lock)

nextp, nextc: item;

INICIALIZACIÓN SEMÁFOROS

full :=0; *empty* := *n*; *mutex* :=1;

PROBLEMA DEL BUFFER LIMITADO

Proceso Productor	Proceso Consumidor
repeat … produce un ítem en *nextp* … *wait*(*empty*); ***wait*(*mutex*);** … agregue *nextp* al buffer … ***signal*(*mutex*);** *signal*(*full*); **until** *false*;	**repeat** *wait*(*full*) ***wait*(*mutex*);** … remueve un ítem de *buffer* a *nextc* … ***signal*(*mutex*);** *signal*(*empty*); … consume el ítem en *nextc* … **until** *false*;

PROBLEMA DEL BUFFER LIMITADO

```
monitor ProdCons
    condition full, empty;
    integer contador;

procedure insertar(ítem: integer)
begin
    if contador == N then full.wait();
    Insertar_ítem(ítem);
    contador := contador + 1;
    If contador == 1 then empty.signal()
end;

function remover: integer
begin
    If contador == 0 then empty.wait();
    Remover = remover_ítem;
    contador := contador - 1;
    if contador == N-1 then full.signal()
end;

    contador := 0;
end monitor;
```

PROBLEMA DEL BUFFER LIMITADO

```
procedure productor
begin
    while true do
    begin
     ítem = produce_ítem;
     ProdCons.insertar(ítem);
    end
end;
```

```
procedure consumidor
begin
    while true do
    begin
        ítem = ProdCons.remover;
        Consume_ítem(ítem);
    end
end;
```

AGENDA

1. Fundamentos
2. El Problema de la Sección Crítica
3. Solución a la sección crítica para 2 procesos (Algoritmo de Peterson) y para n procesos (Algoritmo del Panadero)
4. Soluciones por Hardware
5. Soluciones por Software
    1. Locks
    2. Semáforos
    3. Monitores
6. Problemas Clásicos
7. **Ejemplos de Sincronización en los Sistemas Operativos**

EJEMPLOS DE SINCRONIZACIÓN

- Windows
- Linux
- Solaris
- Pthreads

SINCRONIZACIÓN WINDOWS

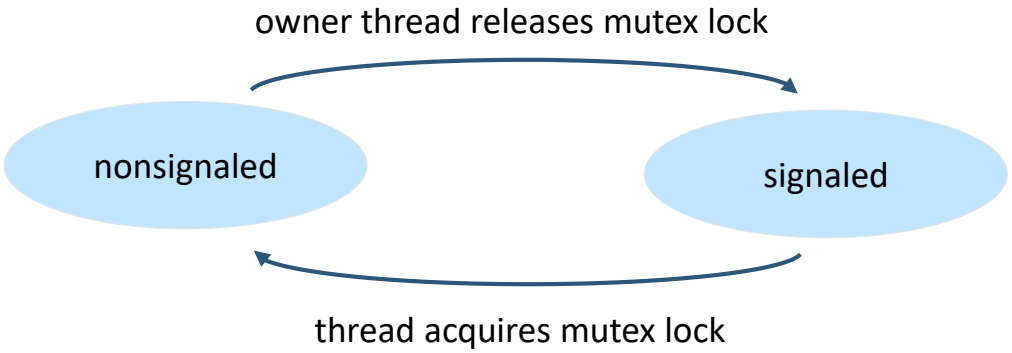
En el kernel

- Usa máscaras de interrupción para proteger el acceso a recursos globales en sistemas mono procesador
- Usa **spinlocks** en sistemas multiprocesador

Sincronización hilos

- También provee dispatcher objects los cuales actúan como mutex locks, semáforos, eventos y timers.
- Los *Dispatcher objects* pueden proveer eventos
  - Un evento actúa como una variable de condición

Los estados del ***MUTEX DISPATCHER OBJECT***



SINCRONIZACIÓN EN LINUX

- Linux:
  - Antes de la versión 2.6 no era un kernel totalmente apropiativo.
  - Deshabilita las interrupciones para implementar secciones críticas cortas
- Linux provee:
  - mutex-locks
  - semáforos
  - spin locks

Un procesador	Múltiples procesadores
Deshabilitar apropiación kernel	Acquire spin lock
Habilitar apropiación kernel	Release spin lock

SINCRONIZACIÓN EN SOLARIS

Implementa una variedad de locks para soportar multitasking, multithreading (incluyendo threads en tiempo real), y multiprocesamiento.

- Lock Exclusión Mutua
- Semáforos
- Lock Lectores-Escritor
- Variables de condición

SINCRONIZACIÓN EN EN POSIX APIS

- APIs Pthreads son independientes de los SOs
- Proveen:
  - Locks mutex
  - Variables de condición
- Extensiones no portables incluyen:
  - Locks lector-escritor
  - spin locks

Bibliografía:

- Silberschatz, A., Gagne G., y Galvin, P.B.; "Operating System Concepts", 7ma Edición 2009, 9na Edición 2012, 10ma Edición 2018.
- Stallings, W. "Operating Systems: Internals and Design Principles", Prentice Hall, 7ma Edición 2011, 8va Edición 2014, 9na Edición 2018.
- Tanenbaum, A.; "Modern Operating Systems", Addison-Wesley, 3ra. Edición 2008, 4ta. Edición 2014.