Sincronización

4

Sistemas Operativos y Distribuidos

Mg. Javier Echaiz
D.C.I.C. – U.N.S.
http://cs.uns.edu.ar/~jechaiz
je@cs.uns.edu.ar

Sincronización de procesos

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.
- Supóngase que se modifica el código del productor-consumidor agregando una variable *contador*, inicializado a 0 e incrementado en cada momento que un nuevo item es agregado al buffer

Buffer Limitado

- Datos compartidos

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

- Proceso Productor

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;

Buffer Limitado (cont.)

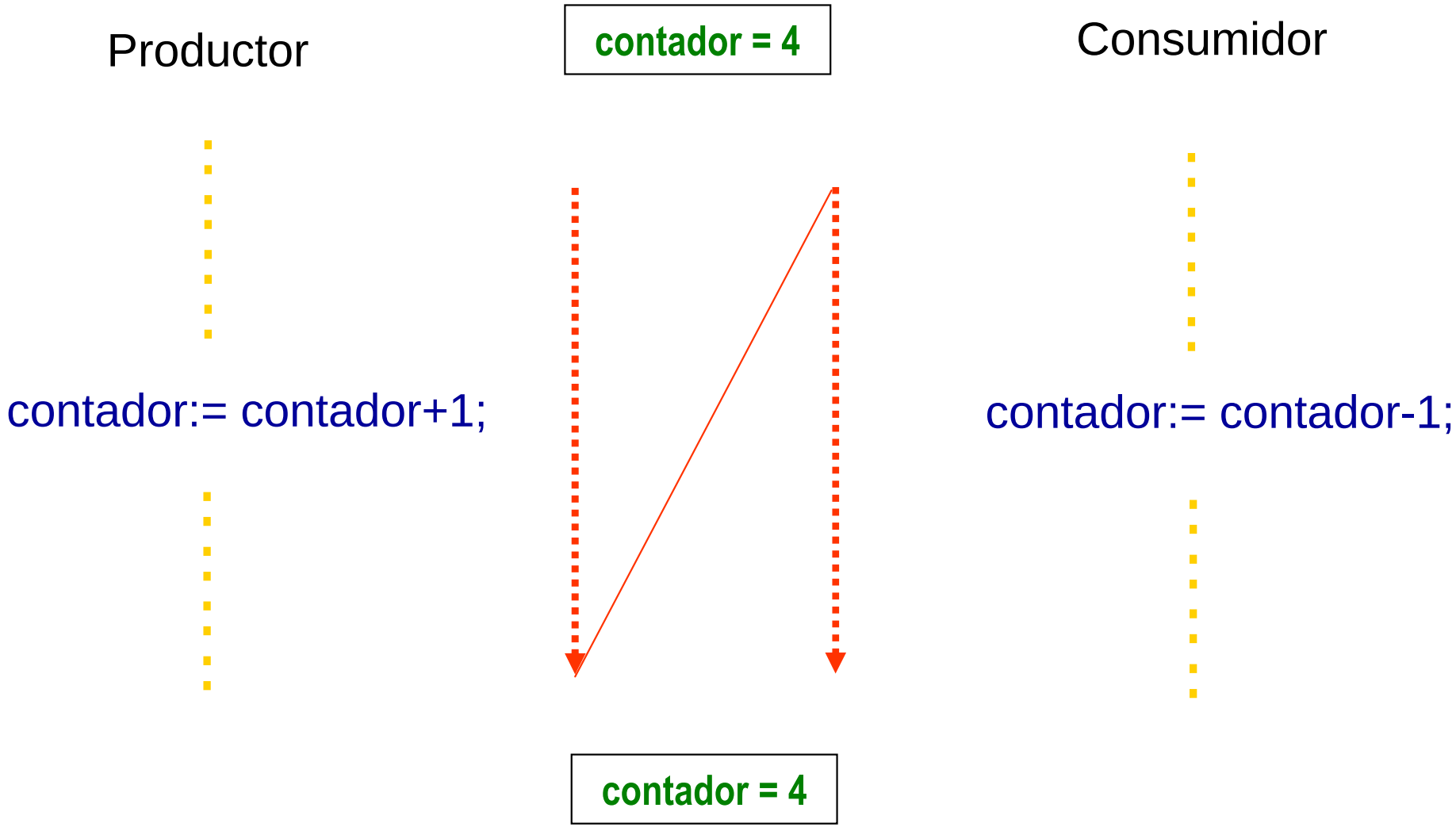
Proceso Consumidor

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*;

- Las sentencias:
  - *contador* := *contador* + 1;
  - *contador* := *contador* - 1;

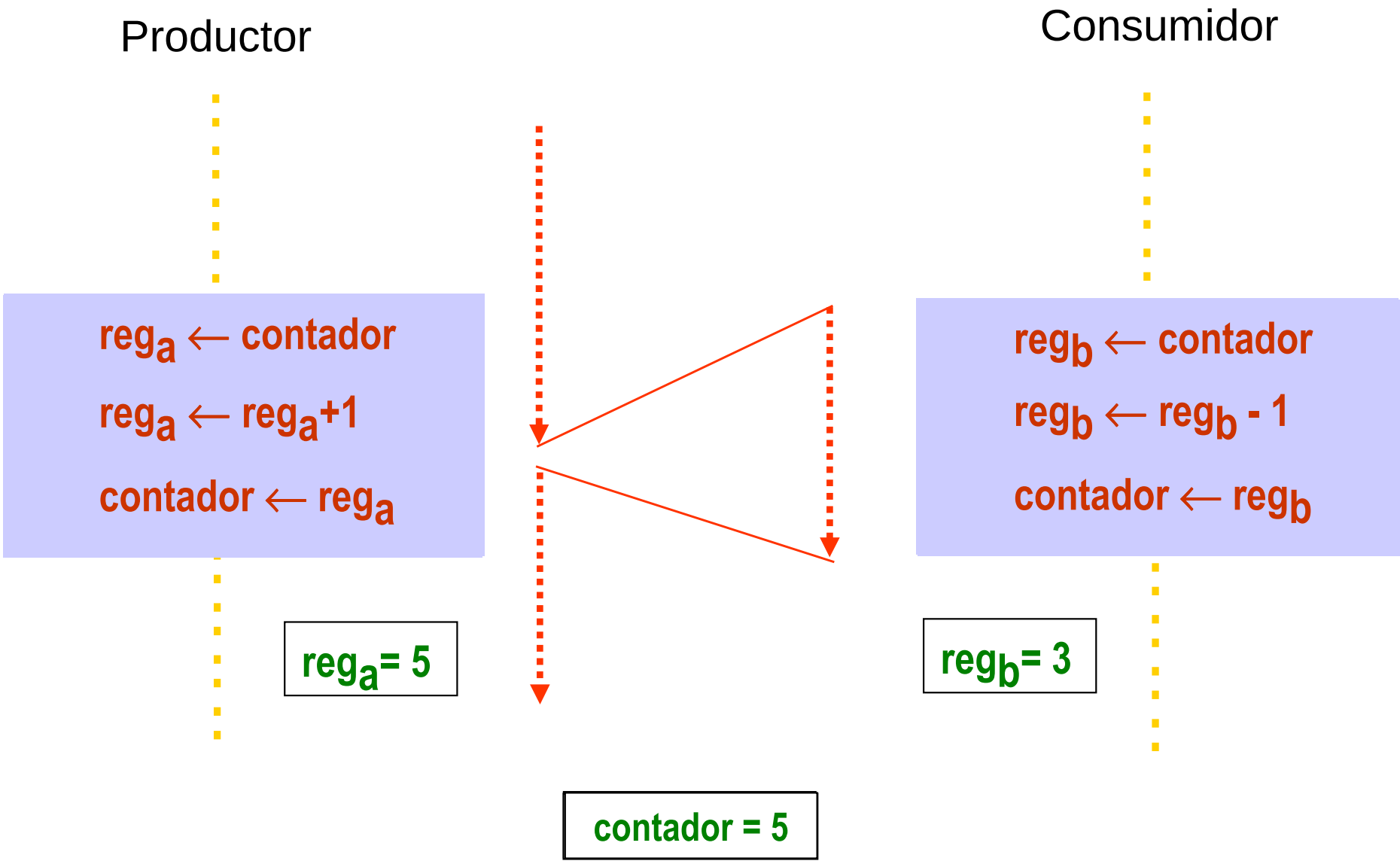
  deben ser ejecutadas *atómicamente*.

El Problema

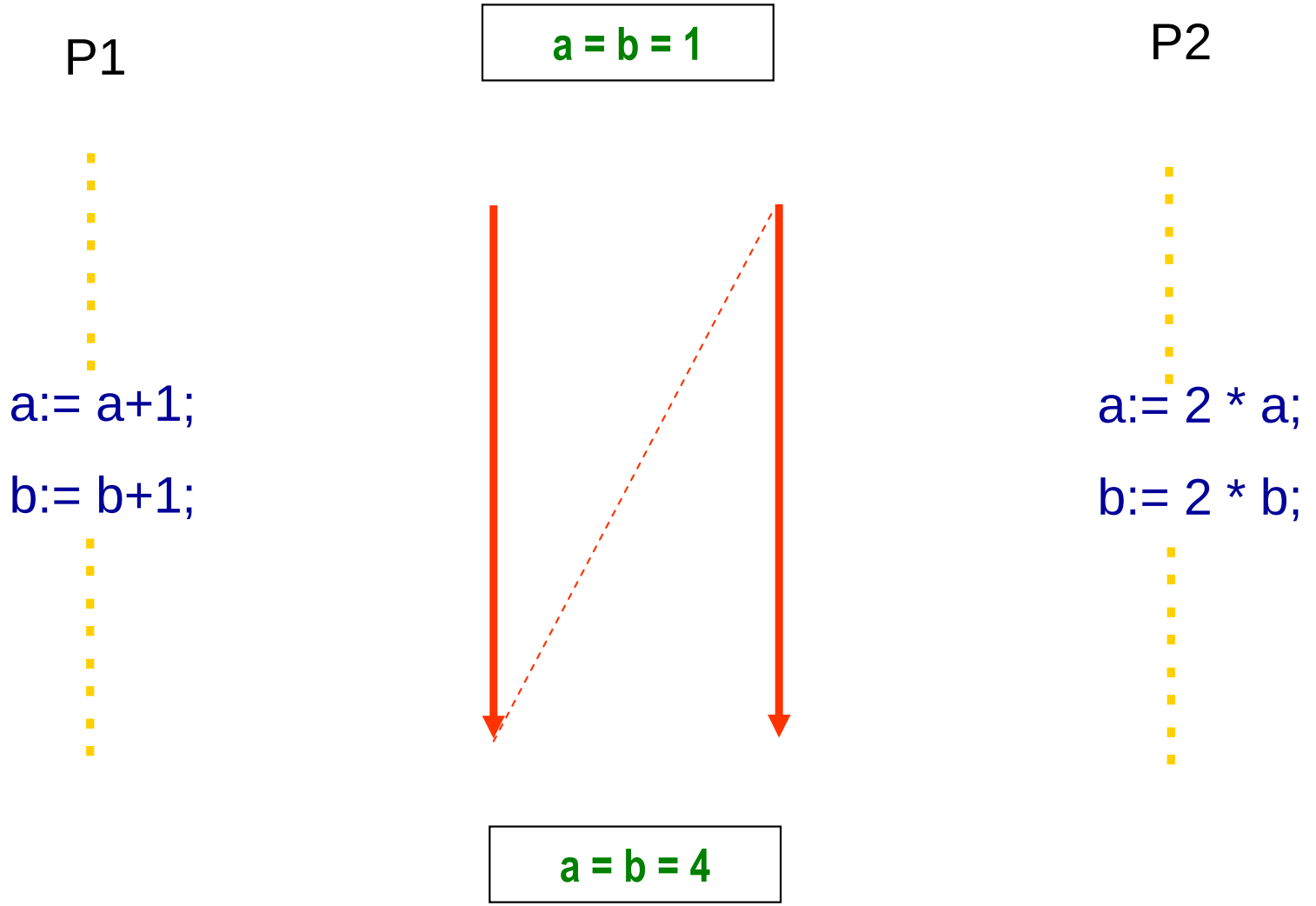


Luego de una operación de Productor y otra de Consumidor … contador permanece invariante

El Problema

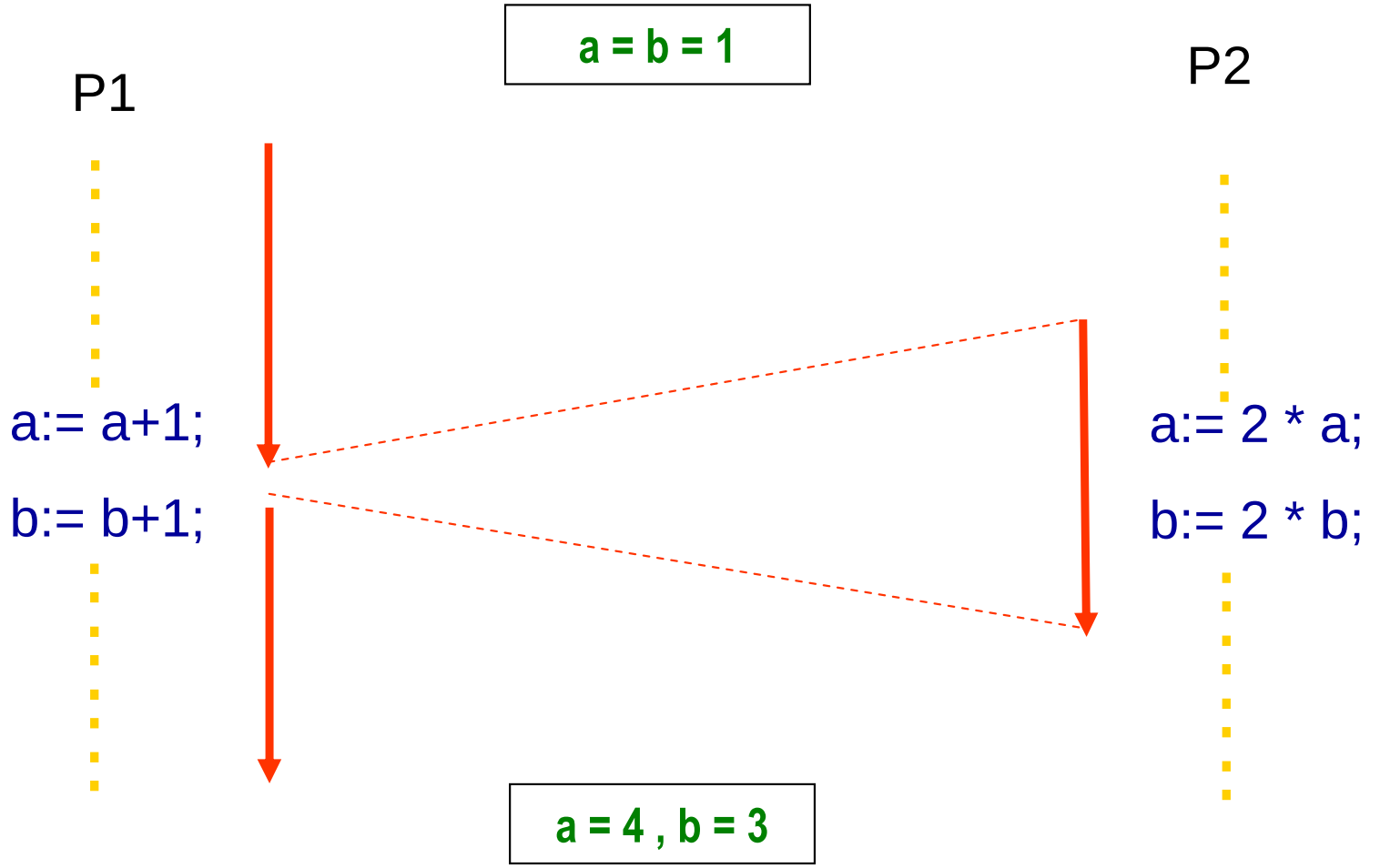

Productor
Consumidor
$reg_a \leftarrow contador$
$reg_a \leftarrow reg_a+1$
$contador \leftarrow reg_a$
$reg_b \leftarrow contador$
$reg_b \leftarrow reg_b - 1$
$contador \leftarrow reg_b$
$reg_a = 5$
$reg_b = 3$
contador = 5

El Problema



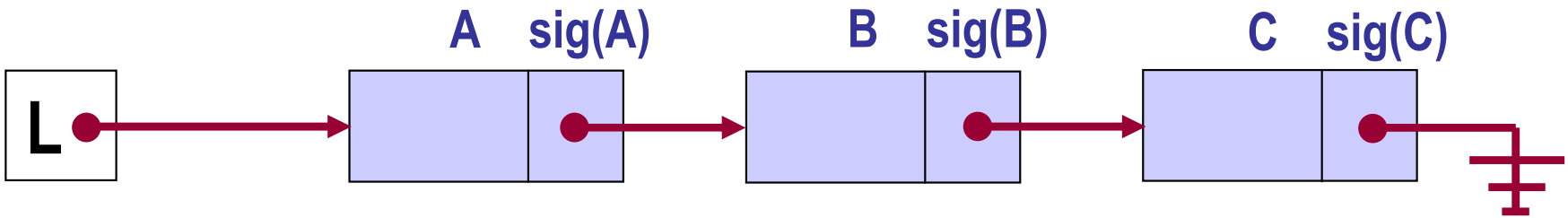
Luego de la ejecución de P1 seguida de P2 los valores de *a* y *b* deben ser iguales entre sí

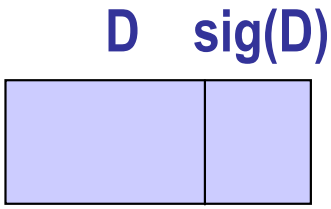
El Problema

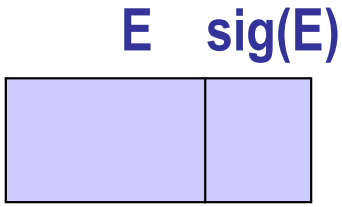


Luego de la ejecución de P1 y P2 los valores de *a* y *b* deberían ser iguales entre sí, *pero no lo son.*

El Problema







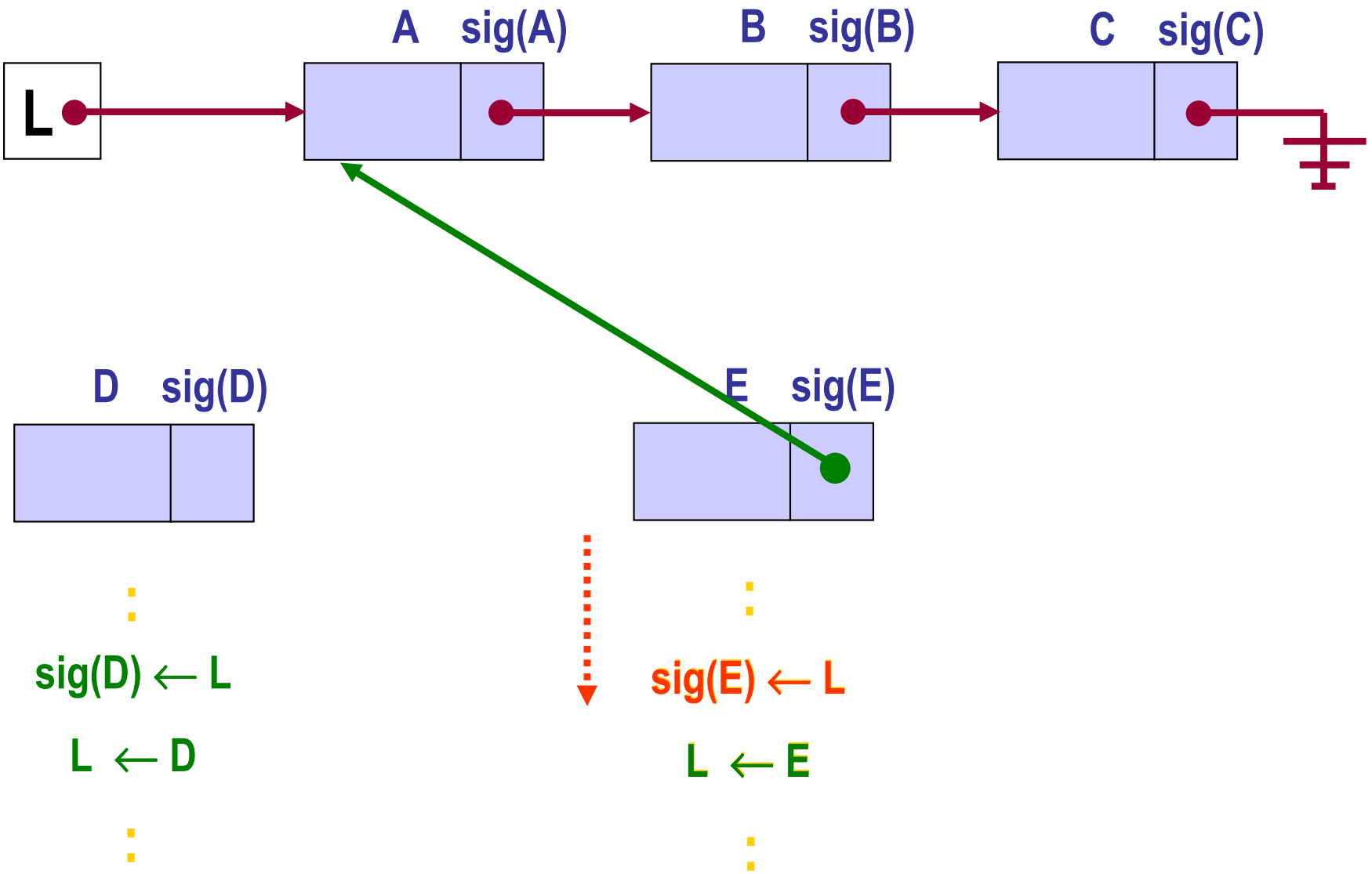
sig(D) ← L

L ← D

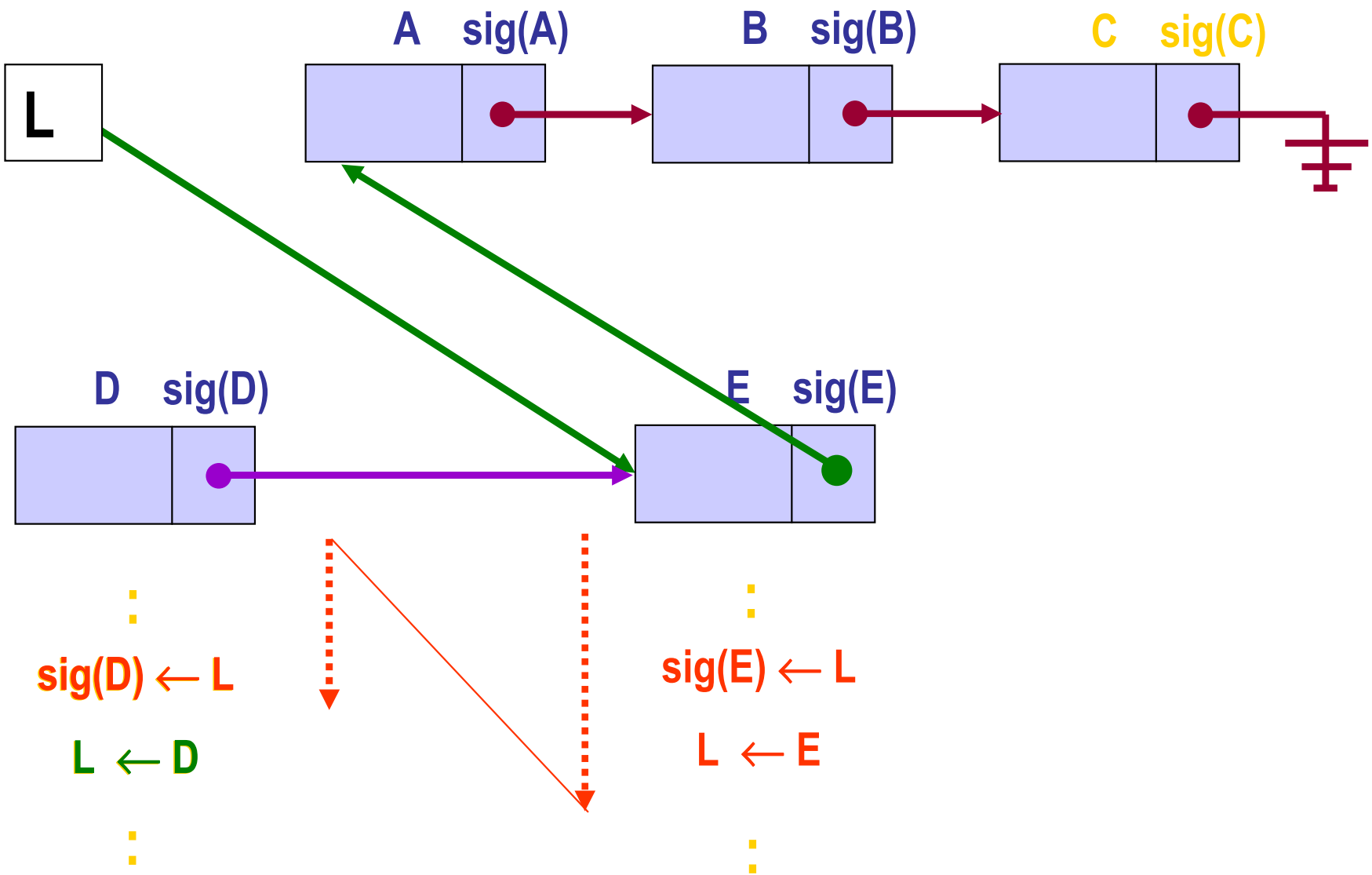
sig(E) ← L

L ← E

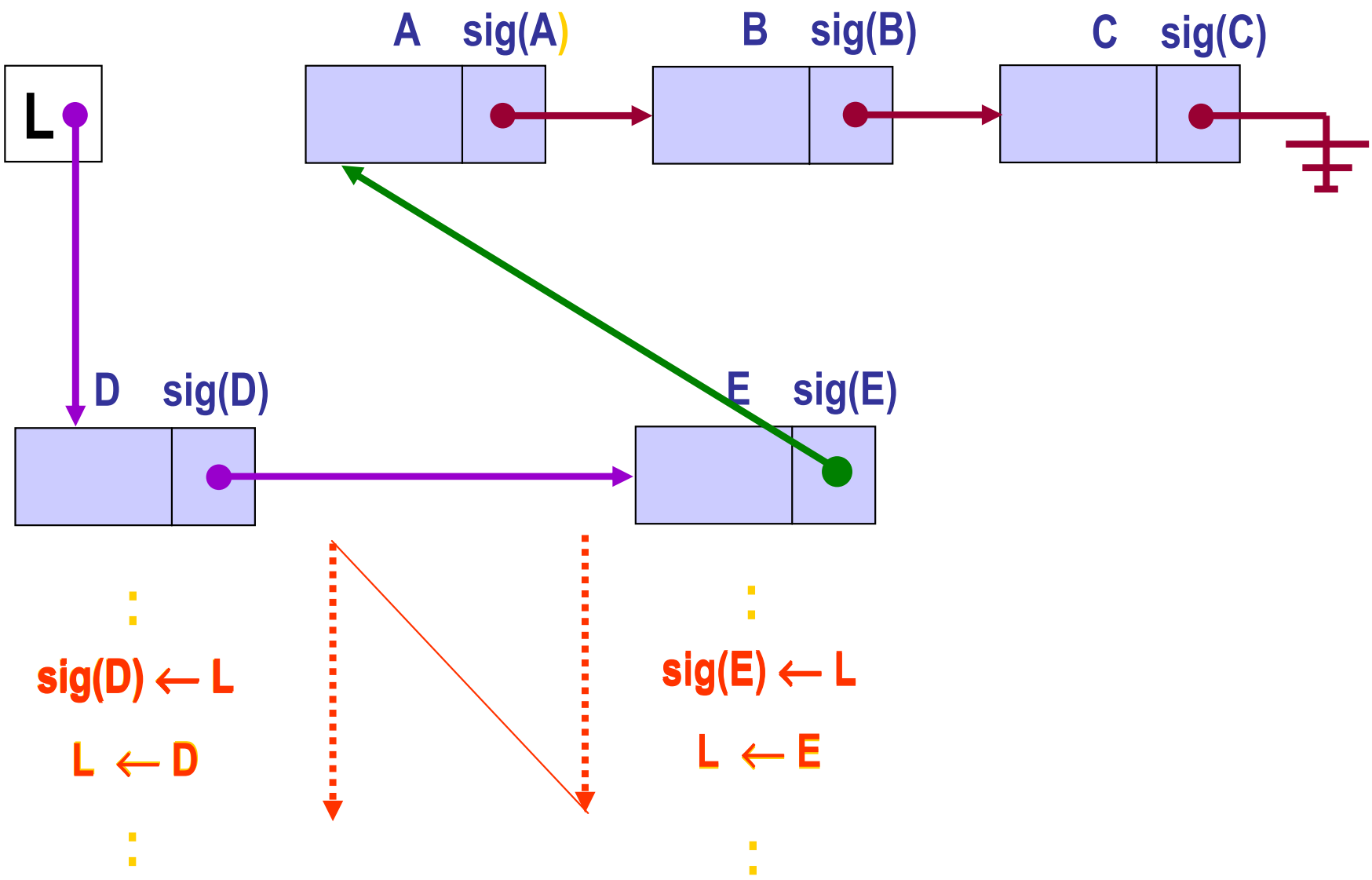
El Problema



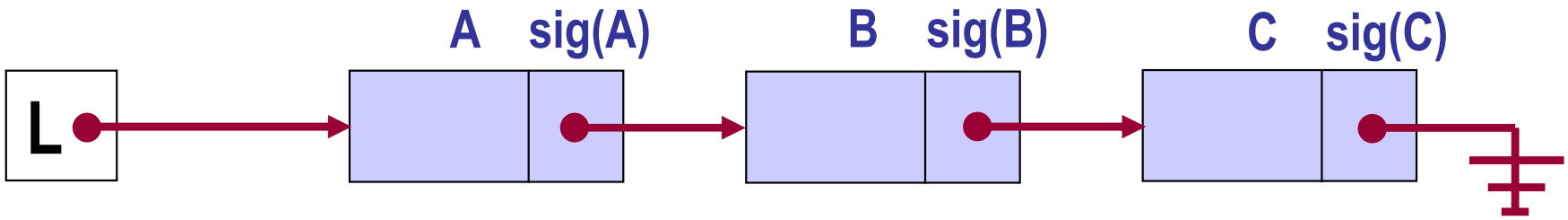
El Problema

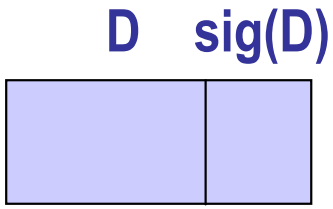


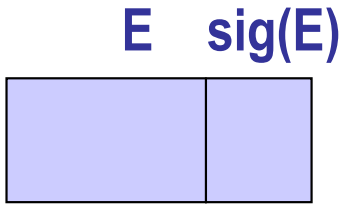
El Problema



El Problema







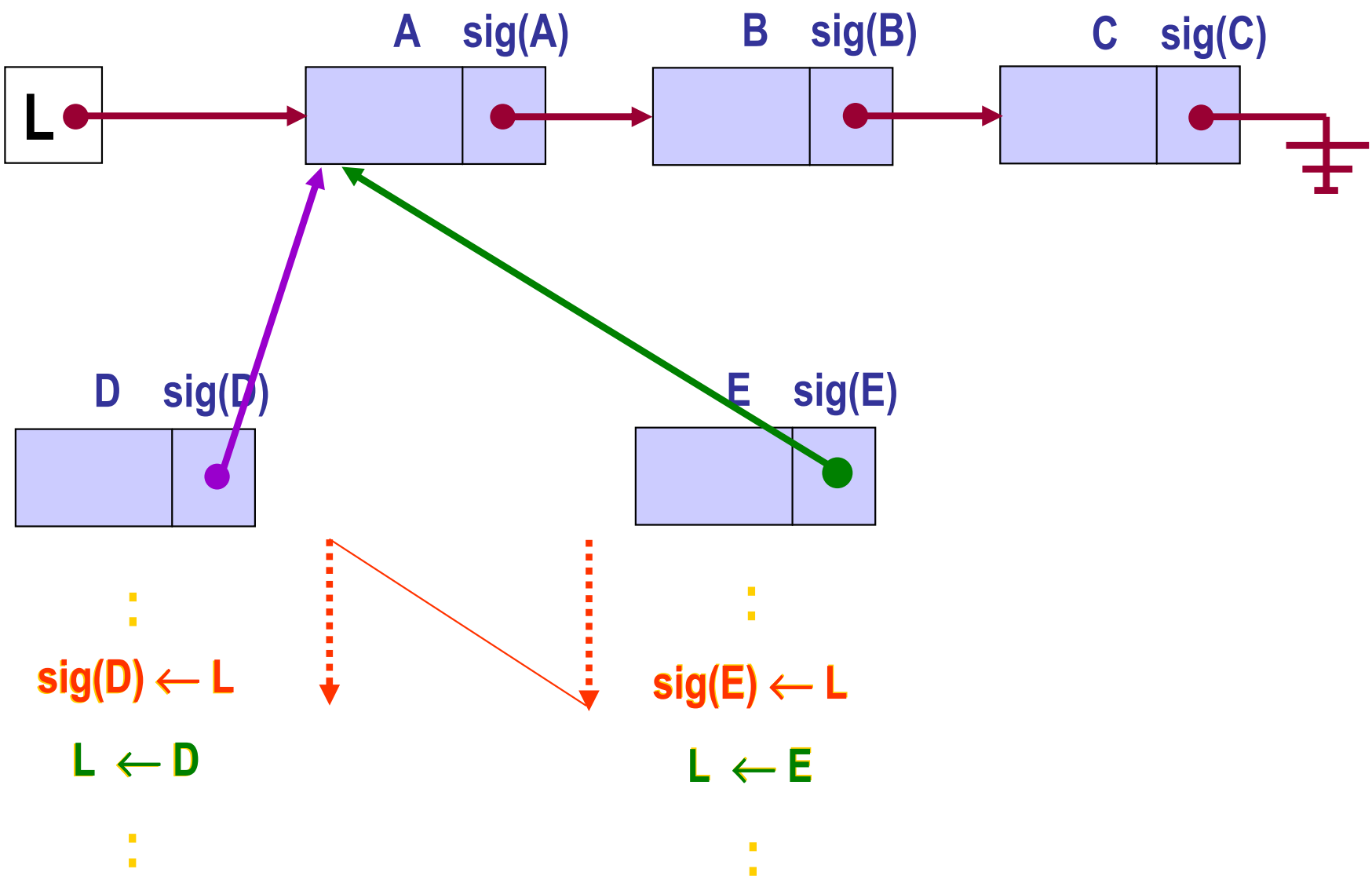
sig(D) ← L

L ← D

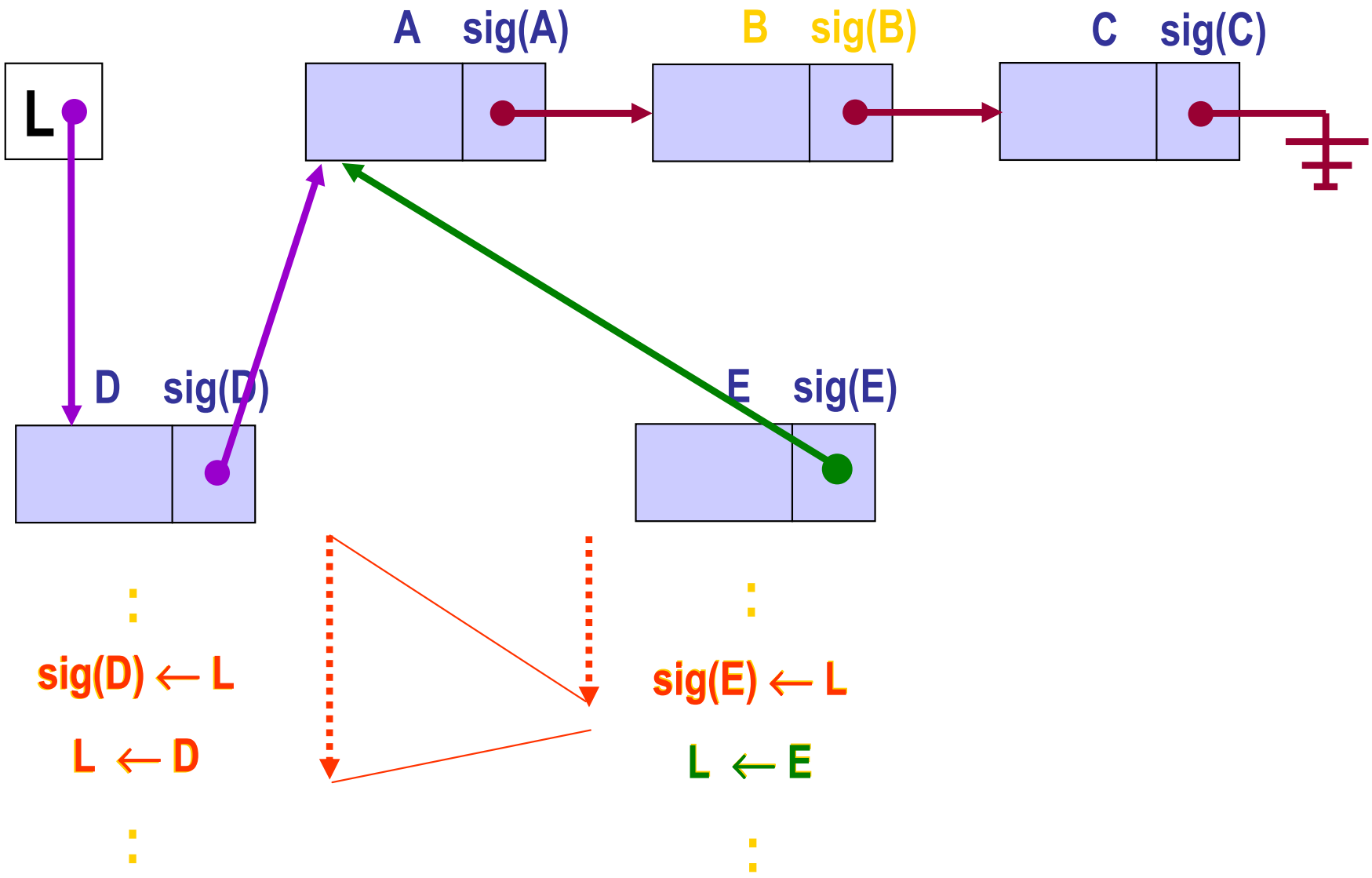
sig(E) ← L

L ← E

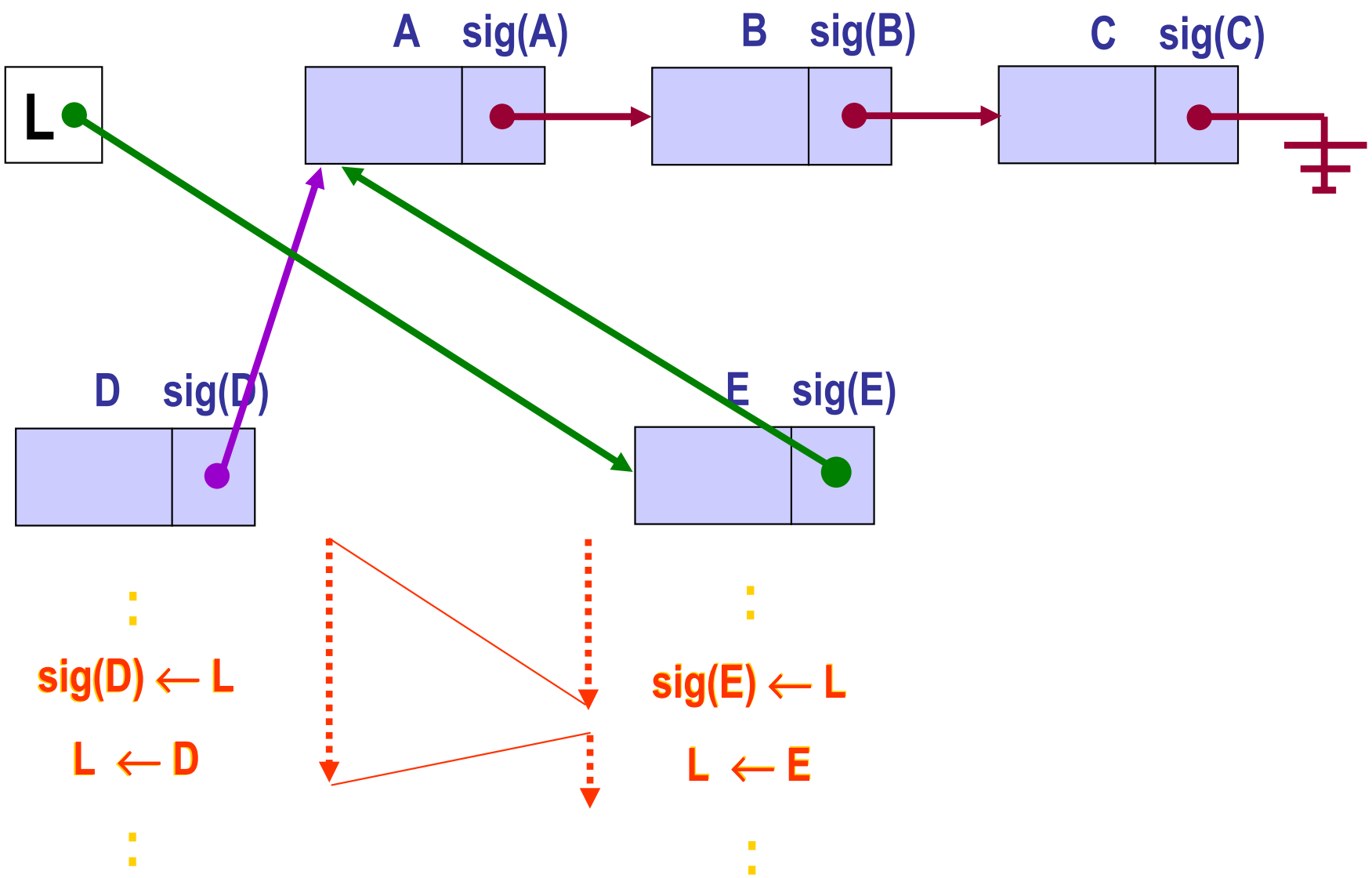
El Problema



El Problema



El Problema



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**.

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

1. **Exclusión Mutua**. Si el proceso P_j está ejecutando en su sección crítica, entonces ningún otro proceso puede estar ejecutando en sus secciones críticas.
2. **Progreso**. Si ningún proceso está ejecutando en su sección crítica y existe algunos procesos que desean entrar en sus secciones criticas, entonces la selección de procesos que desean entrar en la sección crítica no puede ser pospuesta indefinidamente.
3. **Espera Limitada**. Debe existir un límite en el número de veces que a otros procesos les está permitido entrar en sus secciones críticas después que un proceso ha hecho un requerimiento para entrar en su sección crítica y antes que ese requerimiento sea completado.
   - Asuma que cada proceso ejecuta a velocidad distinta de cero.
   - No se asume nada respecto a la velocidad relativa de los *n* procesos.

Intentos Iniciales para resolver el Problema

- Sólo 2 procesos, P_0 y P_1
- Estructura general del proceso P_i

repeat

protocolo de entrada

sección crítica

protocolo de salida

resto de sección

until *falso*;

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

Algoritmo

- Combina las variables compartidas de los algoritmos 1 y 2.
- Proceso P_i

```
repeat
    flag [i] := true;
    turn := j;
    while (flag [j] and turn = j) do no-op;
        sección crítica
    flag [i] := false;
        resto de sección
until false;
```

- Alcanza los tres requerimientos; resuelve el problema de la sección crítica para dos 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 mas 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 servido 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 del "Panadero"(Cont.)

- Notación orden lexicográfico (ticket #,id proceso #)
  - (a,b) < c,d) si a < c o si a = c y b < d
  - max (a_0,…, a_{n-1}) es un número k, tal que k ≥ a_i para i = 0, …, n – 1
- Dato compartido

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 del "Panadero"(Cont.)

```
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;
       resto de sección
until false;
```

Sincronización por Hardware

- Verifica y modifica el contenido de una palabra atómicamente.

función *Test-and-Set* (**var** target: *boolean*): *boolean*;

begin

Test-and-Set := *target*;

target := *true*;

end;

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*;

resto de sección

until *false*;

Semáforos

- Es una herramienta de sincronización.
- Semáforo *S* – variable entera
- Puede ser accedido solo por dos operaciones indivisibles (atómicas):

wait (*S*): **while** *S*≤ 0 **do** *no-op*;
S := *S* – 1;

signal (*S*): *S* := *S* + 1;

Ejemplo: Sección Crítica de *n* Procesos

- Variables compartidas
  - **var** *mutex* : *semaphore*
  - inicialmente *mutex* = 1
- Proceso P_i

repeat

wait(*mutex*);

sección crítica

signal(*mutex*);

resto de sección

until *false*;

Implementación del Semáforo

- Se define un semáforo como un registro

  type *semaphore* = **record**

  valor: *integer*

  L: **list of** *procesos*;

  end;

- Se supone dos operaciones simples:
  - **block**: suspende el proceso que la invoca.
  - **wakeup(*P*)**: reactiva la ejecución de un proceso bloqueado P.

Implementación (Cont.)

- 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;

El Semáforo como Herramienta General de 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*

Dos Tipos de Semáforos

- *Semáforos de Cuenta* – valor entero sobre un rango de dominio irrestricto.
- *Semáforo Binario* – valor entero que puede valor solo 0 o 1; puede ser simple de implementar.
- Se puede implementar un semáforo de cuenta *S* como un semáforo binario. Cómo?

Problemas Clásicos de Sincronización

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

Monitores

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

```
type monitor-name = monitor
        declaraciones de variables
        procedure entry P1 :(…);
                begin … end;
        procedure entry P2(…);
                begin … end;
                    :
        procedure entry Pn (…);
                begin…end;
        begin
                código de inicialización
        end
```

Monitores (Cont.)

- Se debe declarar una variable de condición para permitir que un proceso espere dentro del monitor:

  var *x, y: condition*

- La variable de condición puede ser solamente usada con las operaciones *wait* y *signal*.
  - La operación

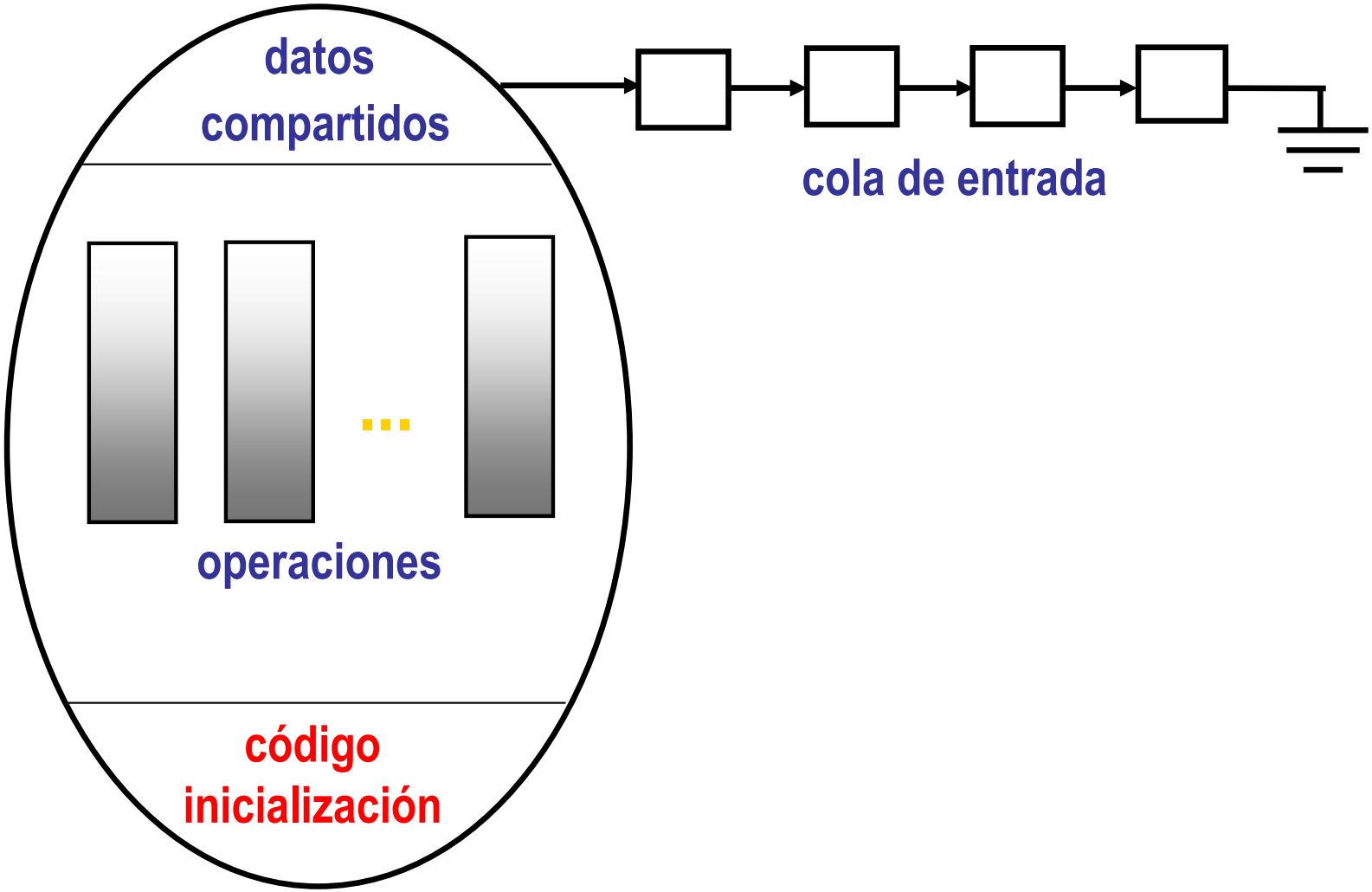
    x.wait;

    significa que el proceso que invoca esta operación es suspendido hasta que otro proceso invoque

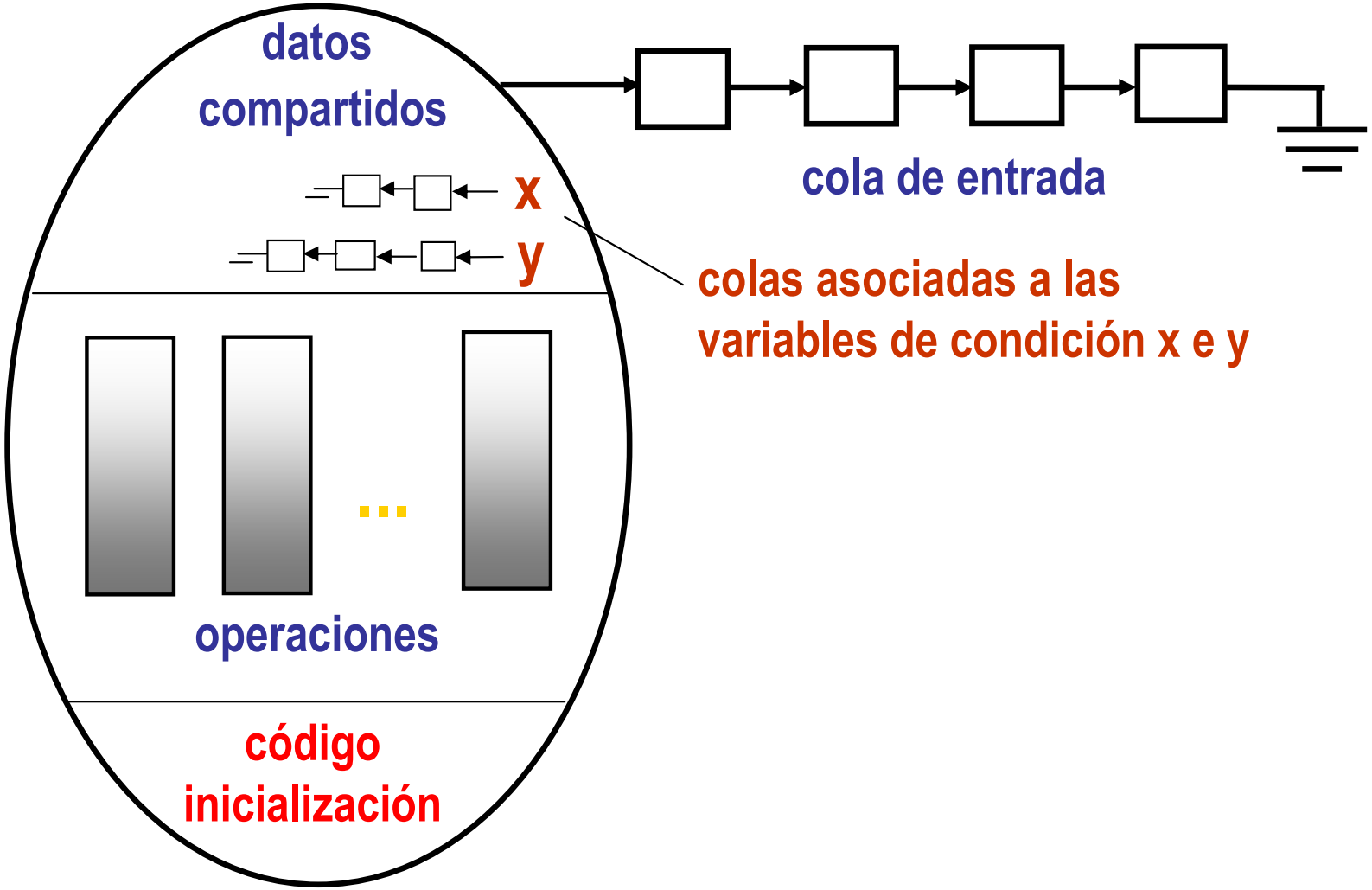
    x.signal;

  - La operación *x.signal* reactiva exactamente un proceso suspendido. Si no hay procesos suspendidos, entonces la operación no tiene efecto.

Vista esquemática de un Monitor



Monitor con Variables de Condición



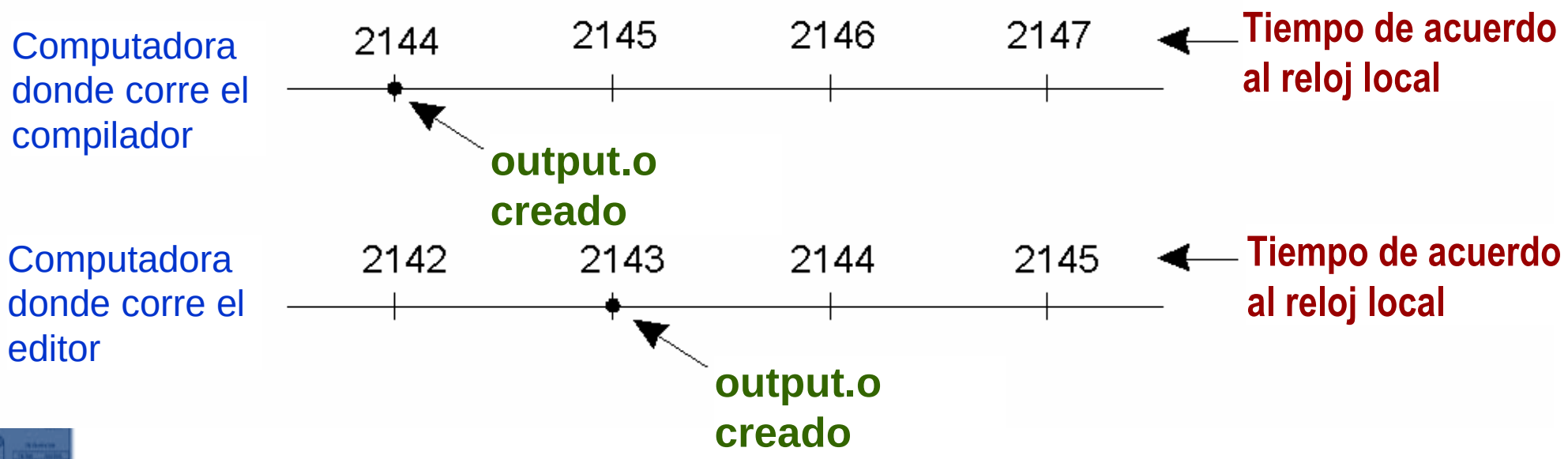
Sincronización en Sistemas Distribuidos

Sincronización en Sistemas Distribuidos

- Sincronización de Relojes.
- Exclusión Mutua.
- Algoritmos de Elección.
- Fallas de Comunicación y Procesos.
- Interbloqueos.

Sincronización de Reloj

Sincronización de Reloj



Cuando cada máquina tiene su propio reloj, un evento que ocurre después de otro no puede ser asignado en tiempo anterior.

Sincronización de Reloj (Cont.)

Sincronización de Reloj

No es posible lograr ordenamiento de eventos si no existe una sincronización entre los relojes de diferentes sitios.

¿Cómo se implementan los relojes de las computadoras?

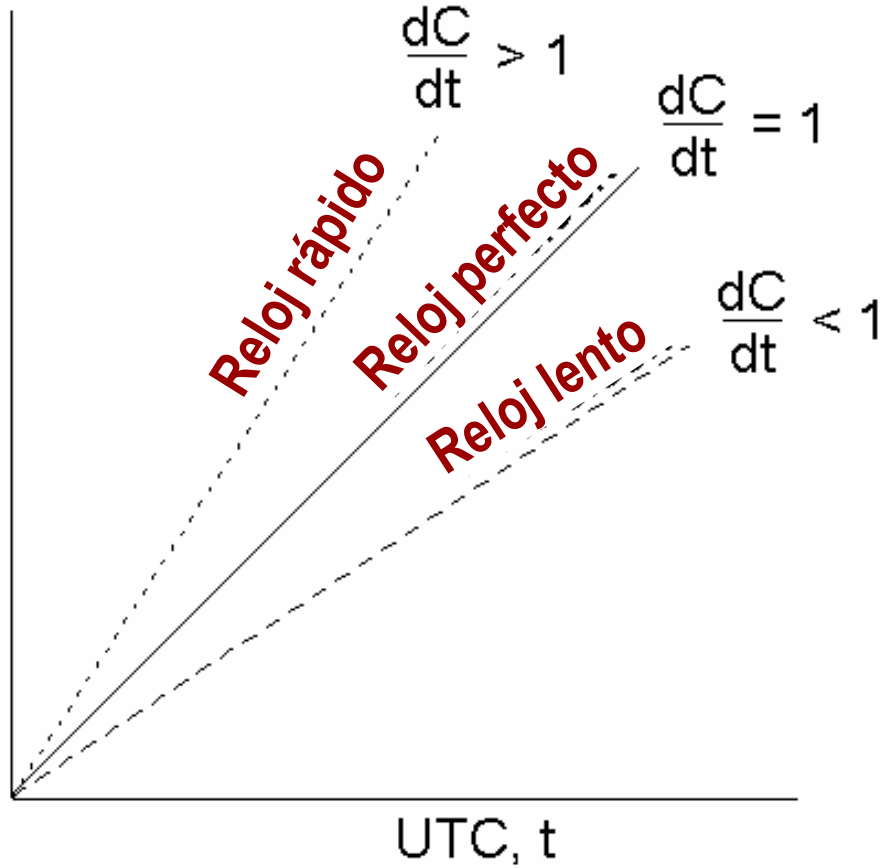
Se tiene un oscilador de cristal de cuarzo, un registro *contador* y un registro *constante*.

Sincronización de Reloj (Cont.)

- El registro *constante* almacena un valor dependiente de la frecuencia de oscilación del cristal. Cuando este registro llega a *cero* produce un *tick de reloj* que se computa en el registro *contador* (el registro constante es reiniciado).
- El valor del registro constante es tal que 60 *ticks* ocurren en un segundo.
- El reloj de la computadora debe sincronizarse con el tiempo real (relojes externos).

Sincronización de Reloj (Cont.)

Deriva de relojes



Sincronización de Reloj (Cont.)

Deriva de relojes

Con el pasaje del tiempo, el reloj de una computadora puede derivar respecto al tiempo real.

La deriva aproximada de un cristal es de:

1 seg cada 1000000 seg (11.6 días)

El valor del tiempo para un reloj *p* es $C_p(t)$.

Si todos los relojes están sincronizados (caso ideal) entonces:

$$C_p(t)=t \ \forall p \text{ y } \forall t \Rightarrow dC/dt=1$$

Sincronización de Reloj (Cont.)

Si el *máximo ritmo de deriva* permitido es ρ, entonces se dice que el *reloj funciona sin falla* si:

$1-\rho \leq dC/dt \leq 1+\rho$

Un sistema distribuido consiste de varios nodos, cada uno con su propio reloj corriendo a su propia velocidad. Esto significa que cada nodo debe sincronizar su reloj respecto a los demás.

Sincronización de Reloj (Cont.)

Esta sincronización puede hacerse de dos maneras:

- Sincronización de los relojes de las computadoras con relojes (externos) de tiempo real. Se utilizan fuentes externas de información como el UTC (*Universal Time Coordinated*).
- Sincronización mutua de los relojes de los diferentes nodos del sistema. Se hace en aquellas aplicaciones que requieren una visión consistente del tiempo en todos los nodos.

Sincronización de Reloj (Cont.)

Debe tenerse en cuenta que:

- Los retardos y cargas de las redes hacen impredecibles el cálculo del costo de los mensajes de actualización.
- El tiempo nunca debe ir hacia atrás porque puede causar serios problemas.

Sincronización de Reloj (Cont.)

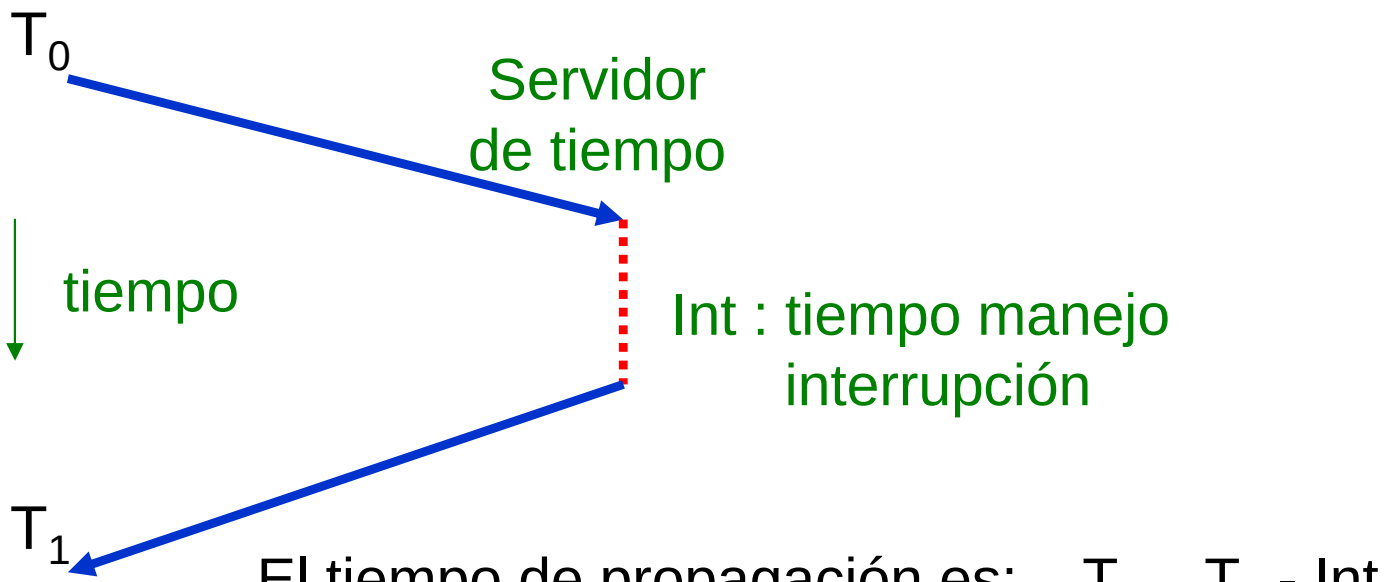
Algoritmos de Sincronización de Reloj

Algoritmos Centralizados

En este tipo de algoritmo, un nodo tiene el tiempo real (de cualquier fuente).

Sincronización de Reloj (Cont.)

Algoritmo Centralizado con Servidor de Tiempo Pasivo (Cristian, 1989)



Sincronización de Reloj (Cont.)

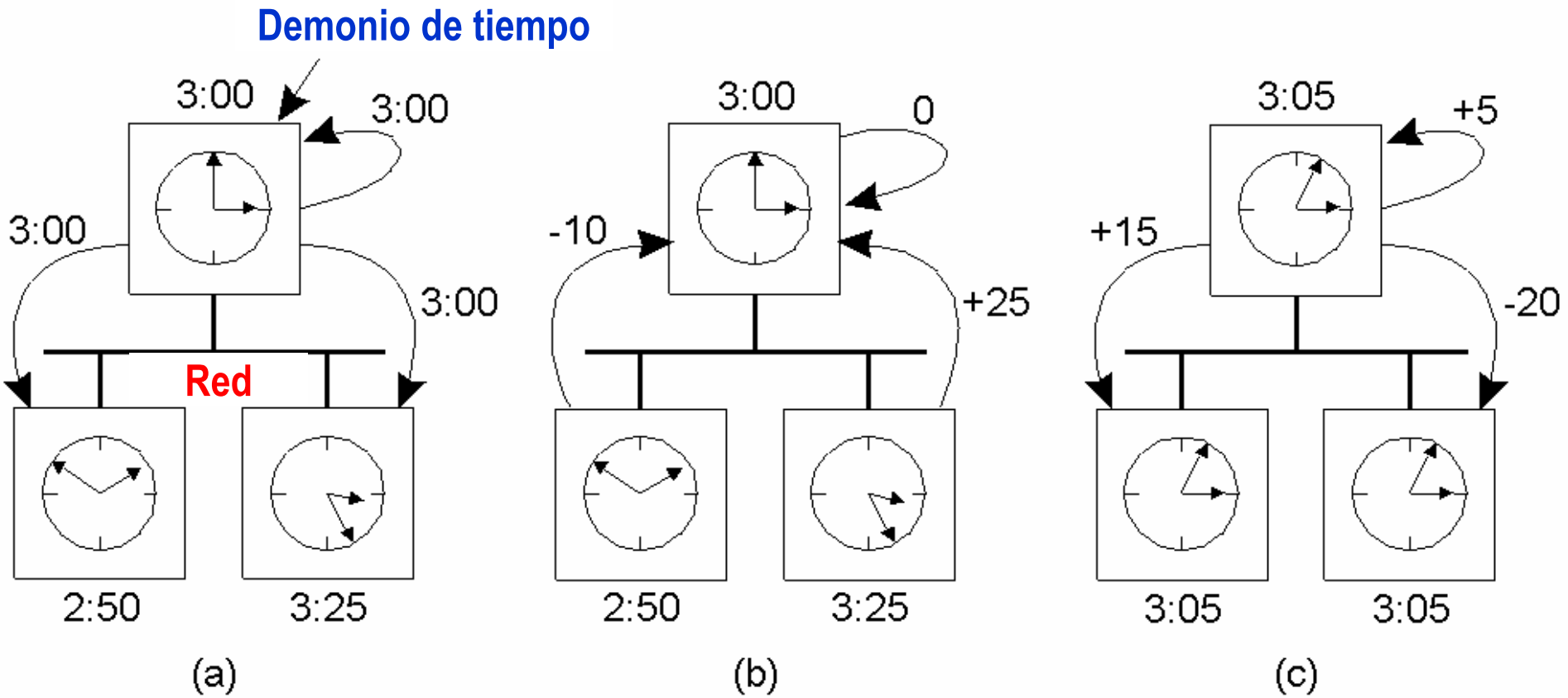
Algoritmo Centralizado con Servidor de Tiempo Activo

El servidor de tiempo activo, periódicamente hace un *broadcast* con su valor de tiempo y los nodos que lo reciben ajustan sus relojes considerando un retardo tipo.

Tiene algunos desajustes.

Sincronización de Reloj (Cont.)

El **algoritmo Berkeley** mejora el anterior:



Sincronización de Reloj (Cont.)

El servidor envía un mensaje a cada uno de los nodos.

Estos le envían al servidor sus tiempos.

El servidor promedia todos (dentro de determinado intervalo, incluido el propio). Este es el valor al cual se tienen que ajustar todos los relojes. El servidor envía solo la diferencia que los otros nodos deben usar para corregir sus relojes.

Sincronización de Reloj (Cont.)

Los algoritmos centralizados sufren de:

Único punto de falla

Malo para la escalabilidad

Sincronización de Reloj (Cont.)

Algoritmos Distribuidos

Algoritmos distribuidos con promedio global.

Cada nodo hace un broadcast con su **tiempo local** a todos los demás, cuando el tiempo local es T_0+iR para algún entero i, donde T_0 es un tiempo fijo en el pasado y R es un parámetro del sistema que depende de factores tales como el número de nodos, la máxima deriva permitida, etc.

Cada nodo promedia y establece su tiempo.

Sincronización de Reloj (Cont.)

Algoritmos distribuidos con promedio localizado.

Los globales no escalan bien.

Se agrupan nodos y se calculan tiempos en ellos como si fueran globales.

La agrupación puede ser en anillo, solamente se consulta a los vecinos.

Sincronización de Reloj (Cont.)

Algoritmo de Lamport

Problema : no hay reloj común

Solución: relojes lógicos

Ordenamiento: Lamport: *"sucede antes"*

1) Si ***a*** y ***b*** son eventos en el mismo proceso y ***a*** ocurre antes que ***b***; entonces

$$a \rightarrow b$$ es verdadero

2) Si ***a*** es el evento de envío de mensaje por un proceso y ***b*** el de recepción del mismo mensaje por otro proceso; entonces:

$$a \rightarrow b$$ es verdadero

Sincronización de Reloj (Cont.)

Algoritmo de Lamport (cont.)

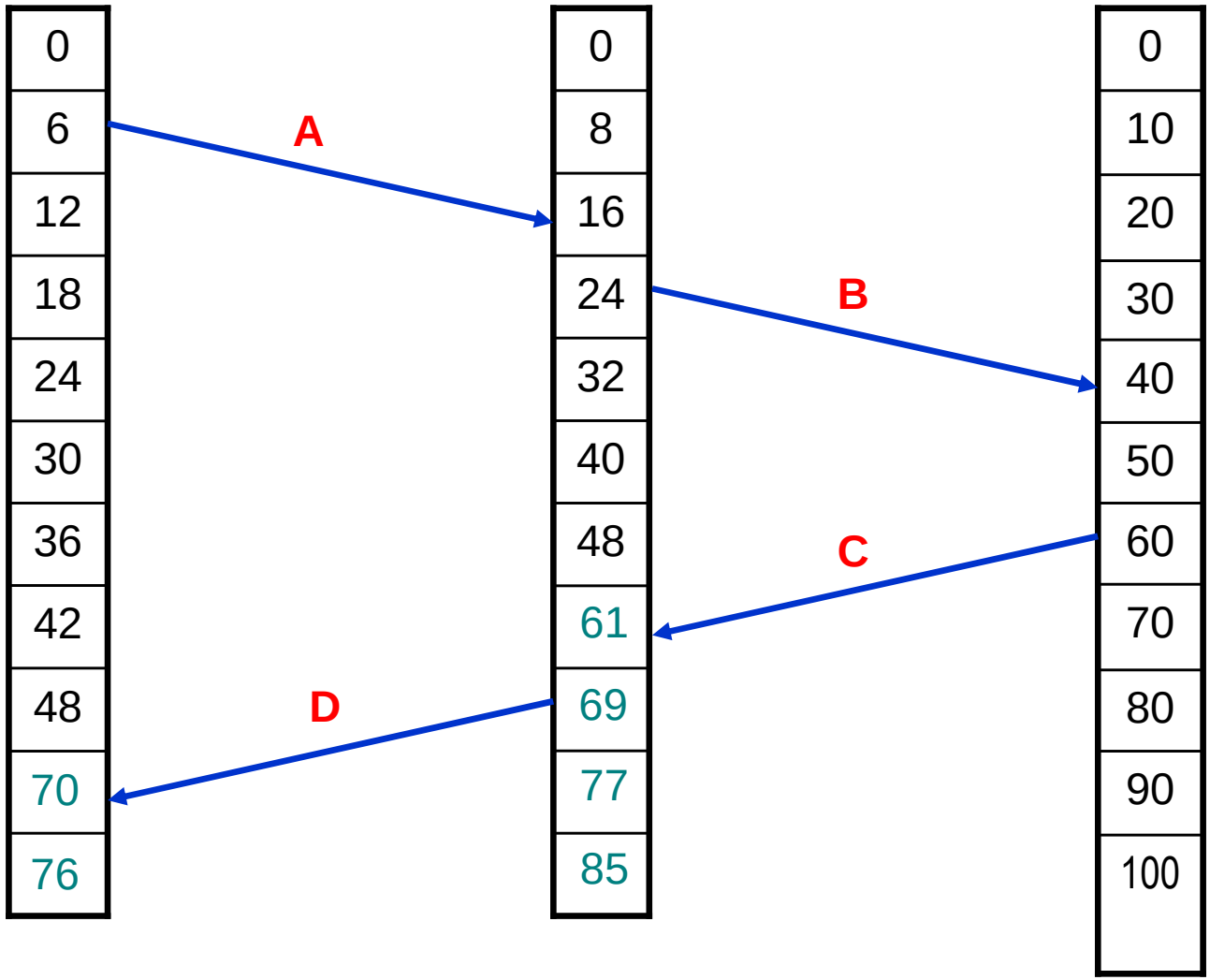
Se asocia un contador C a un evento *a* o *b*:

C(a), C(b)

1. Si *a* sucede antes que *b* en el mismo proceso, C(a) < C(b)
2. Si a y b representan el envío y recepción de un mensaje, C(a) < C(b)
3. Para otros eventos *a* y *b*, C(a) ≠ C(b)

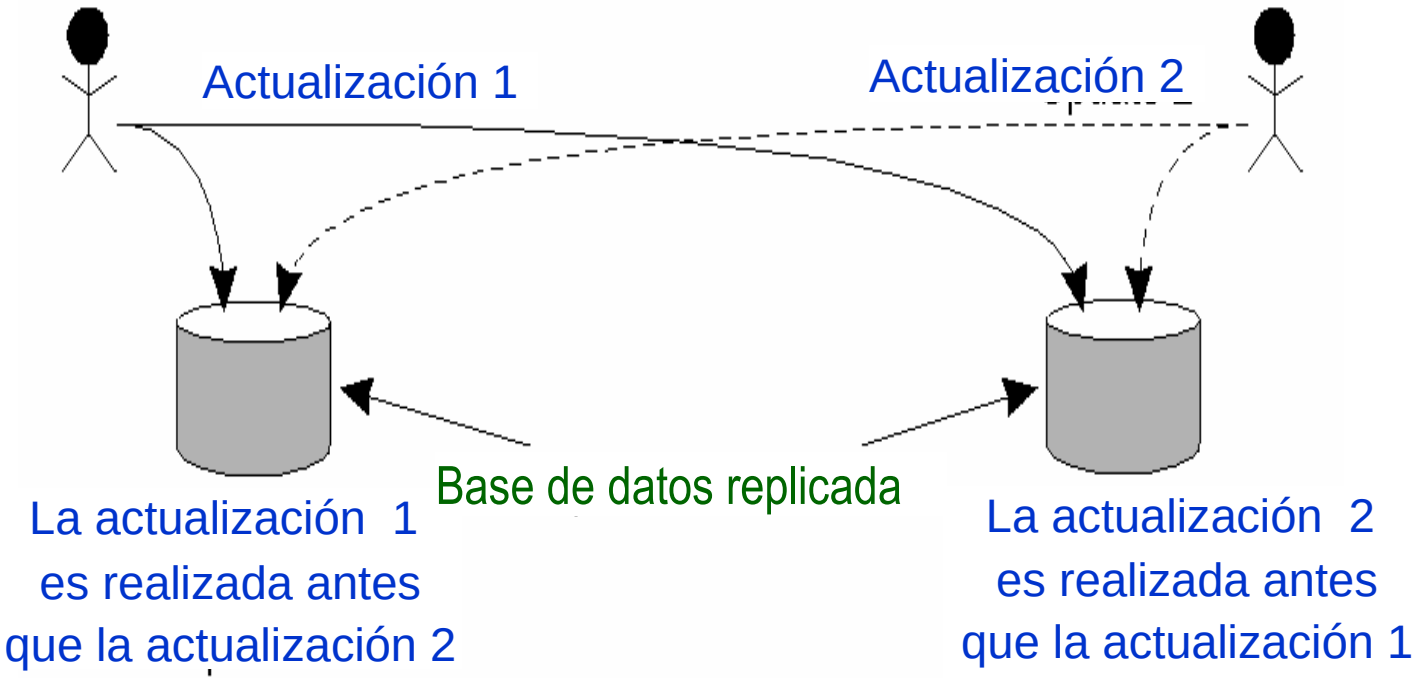
Sincronización de Reloj (Cont.)

Algoritmo de Lamport (cont.)



Sincronización en Sistemas Distribuidos

Ejemplo: Multicast Totalmente Ordenado



Actualizando una base de datos replicada dejándola inconsistente

Estado Global Distribuido

Algunos Términos

- Canal
  - Existe entre dos procesos si intercambian mensajes.
- Estados
  - Secuencia de mensajes que han sido enviados y recibidos por canales incidentes en el proceso.
- Instantánea (Snapshot)
  - Registra el estado de un proceso.
- Estado Global
  - La combinación de estados de todos los procesos
- Instantánea Distribuida (Distributed Snapshot)
  - Una colección de instatáneas, una por cada proceso.

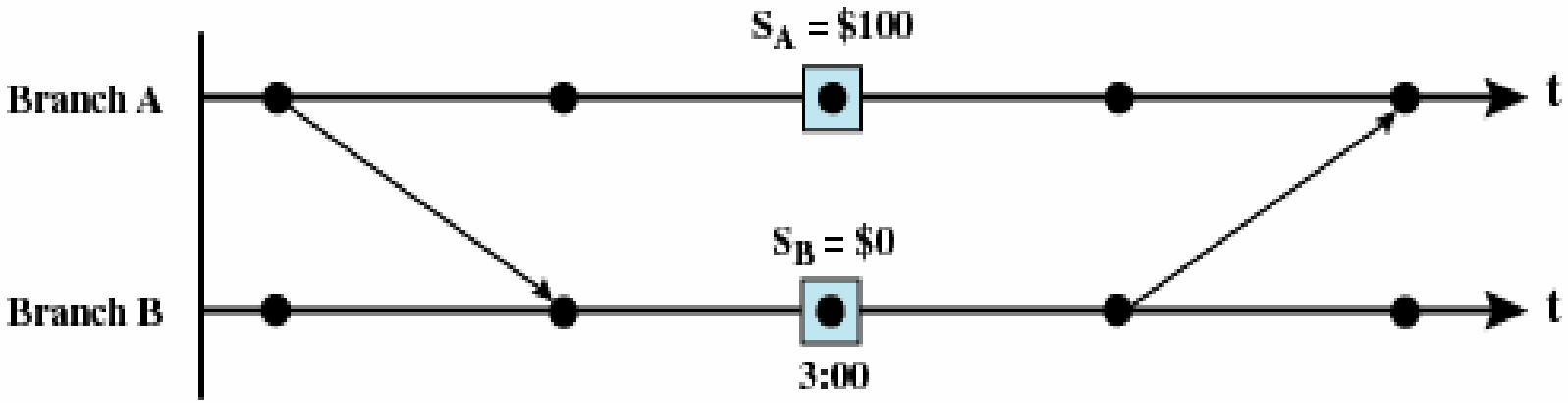
Estado Global Distribuido

- El sistema operativo no puede conocer el estado actual (corriente) de todos los procesos en el sistema distribuido.
- Un proceso solo puede conocer los estados actuales de todos los procesos en el sistema local.
- Los procesos remotos solo conocen la información sobre estados que es proporcionada por los mensajes recibidos.
  - Generalmente estos mensajes presentan información de estado del pasado.

Estado Global Distribuido

Ejemplo

- Una cuenta de banco está distribuida sobre dos sucursales.
- La cantidad total de la cuenta es la suma de cada sucursal.
- El balance de la cuenta es determinado a las 3 PM.
- Para requerir la información se envían mensajes

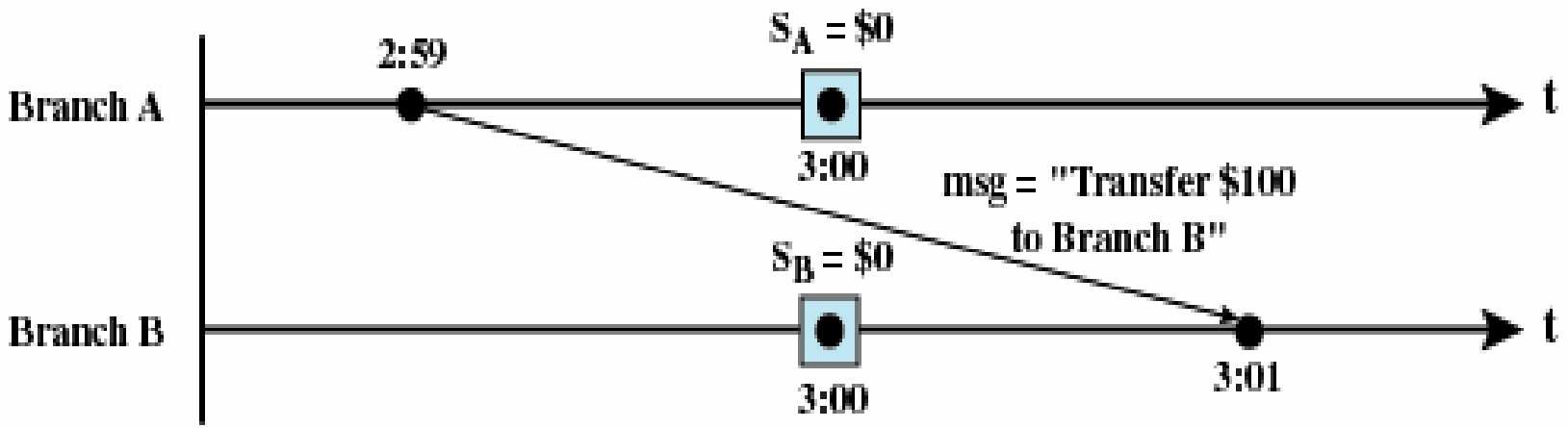


(a) Total = $100

Estado Global Distribuido

Ejemplo (cont.)

- Si al momento de la determinación del balance, el balance de la sucursal A está en tránsito a la sucursal B,
- El resultado es una lectura falsa.



(b) Total = $0

Estado Global Distribuido

Ejemplo (cont.)

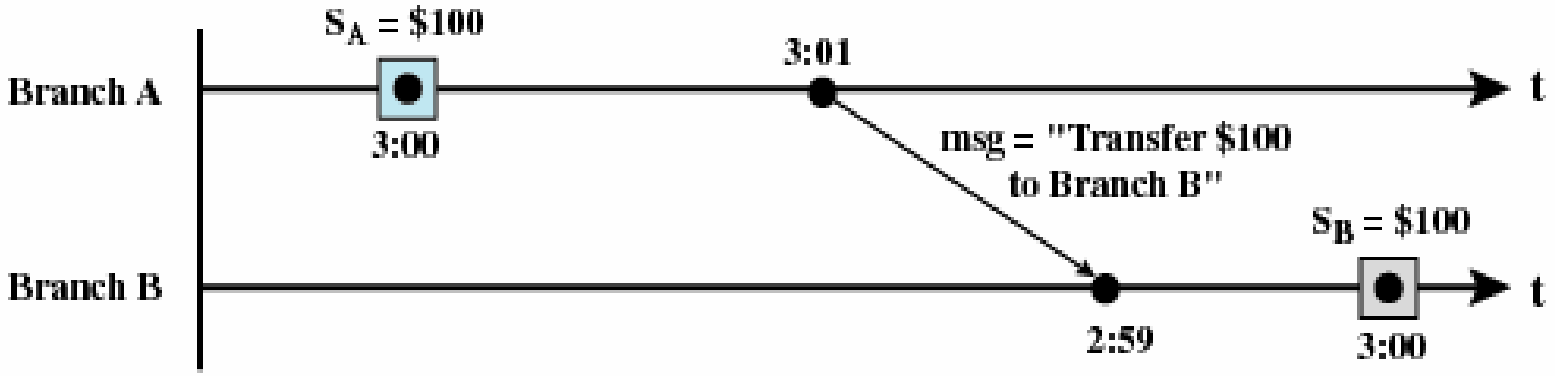
- Todos los mensajes en tránsito deben ser examinados en el tiempo de observación.
- Debe haber consistencia total del balance de ambas sucursales y la cantidad en el mensaje.

Estado Global Distribuido

Ejemplo (cont.)

Esta estrategia no es a prueba de “tontos”.

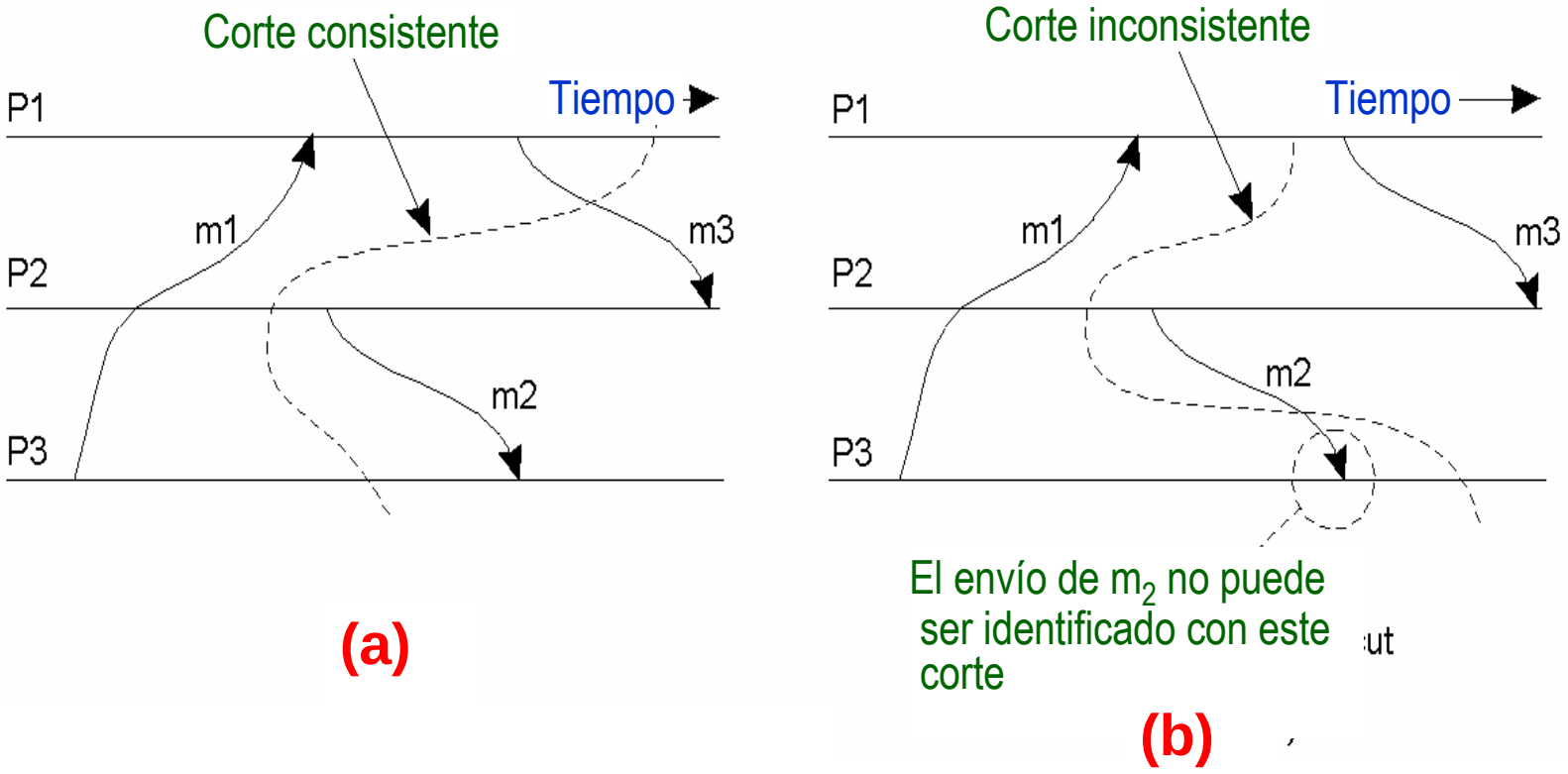
- Si los relojes de las dos sucursales no están perfectamente sincronizados.
- Desde la sucursal A se transfiere el monto a las 3:01.
- El monto llega a la sucursal B a las 2:59 (hora de B)
- A las 3:00 la cantidad es contada dos veces.



(c) Total = $200

Estado Global Distribuido

Estado Global: Problema

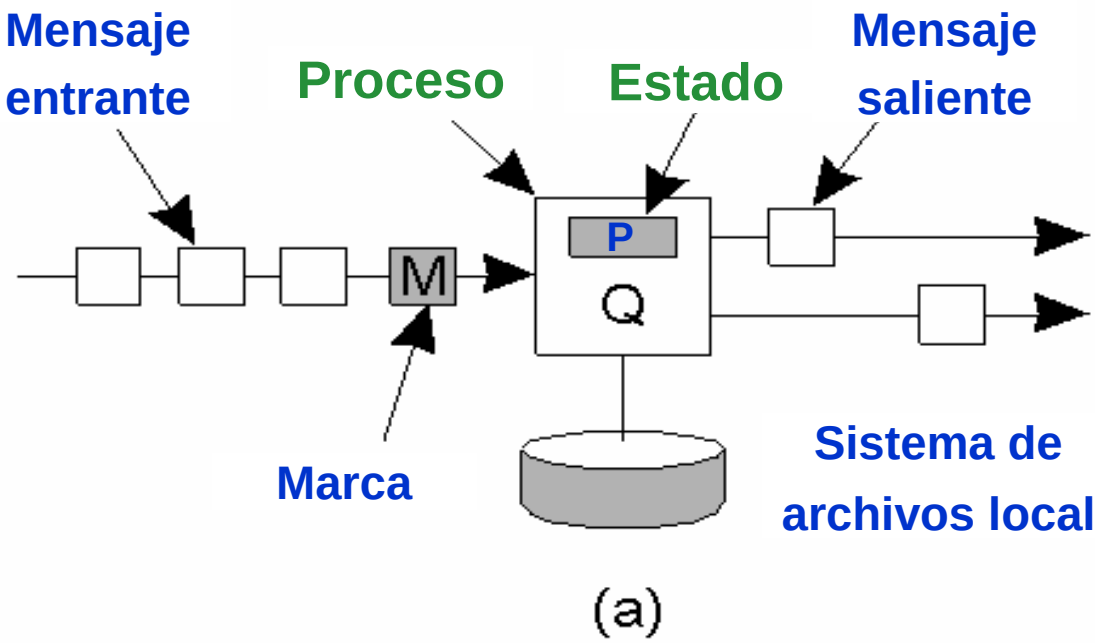


a) Un corte consistente

b) Un corte inconsistente

Estado Global Distribuido

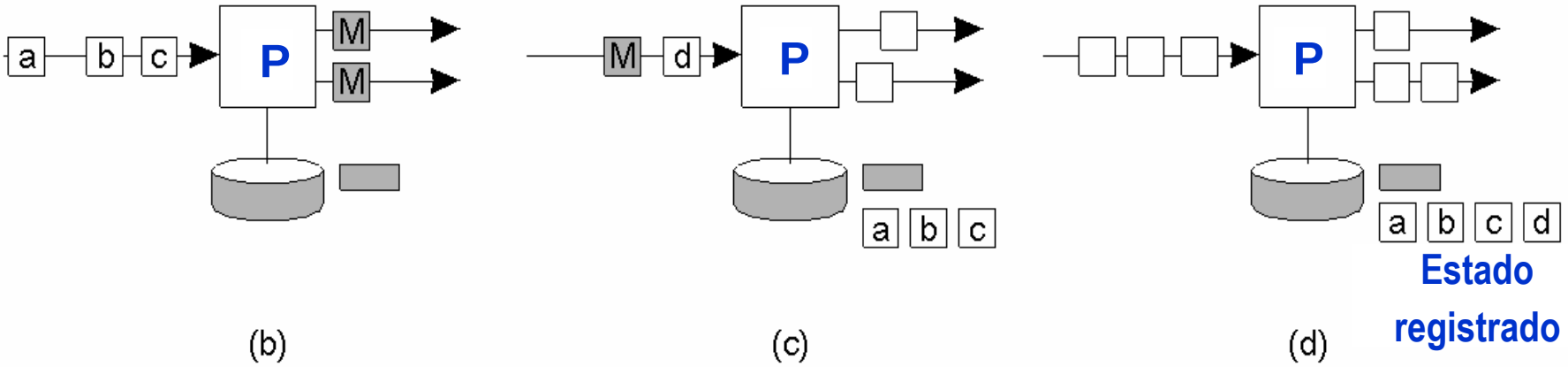
Algoritmo de Instantánea Distribuida



a) Organización de un proceso y canales para una "instantánea" distribuida.

Estado Global Distribuido

Algoritmo de Instantánea Distribuida (Cont.)



b) El proceso P recibe una marca la primera vez y registra su estado local.

c) P registra todos los mensajes entrantes.

d) P recibe una marca para su canal de entrada y termina de registrar el estado del canal entrante.

Estado Global Distribuido

Algoritmo de Instantánea Distribuida (Cont.)

Chandy en el ’85 presentó un algoritmo para la obtención de una instantánea distribuida. El algoritmo supone que los mensajes son recibidos en el orden en que son enviados y no hay pérdidas.
Usa mensajes especiales llamados *marker*.
Algunos procesos inician el algoritmo registrando su estado y enviando un *marker* por todos los canales de salida antes de que sean enviados más mensajes.
Cada proceso *p* procede de la siguiente manera:
Luego de la recepción del primer *marker* (p.e. del proceso *q*) el proceso *p* hace lo siguiente:

1. *p* registra su estado local S_p.
2. *p* registra el estado de los canales entrantes de *q* a *p* como vacío.
3. *p* propaga el *marker* a todos sus vecinos por los canales salientes.

Estado Global Distribuido

Algoritmo de Instantánea Distribuida (Cont.)

Estos pasos deben ser realizados atómicamente, es decir, *p* no puede enviar o recibir mensajes mientras está ejecutando el paso 3.

En algún momento luego de registrar su estado, cuando *p* recibe un marker de otro canal de entrada (p.e. del proceso *r*), hace lo siguiente:

- *p* registra el estado del canal de *r* a *p* como la secuencia de mensajes que *p* ha recibido de *r* desde el momento que *p* registró su estado local S_p hasta el momento que recibió el *marker* de *r*.

El algoritmo termina en el proceso una vez que se han recibido el marker de todos los canales entrantes.

Exclusión Mutua Distribuida

- Suposiciones
  - El sistema consiste de *n* procesos; cada proceso P_i reside en un procesador diferente.
  - Cada proceso tiene una sección crítica que requiere exclusión mutua.

- Dificultades que deben enfrentarse cuando se diseña el protocolo:
  - Interbloqueo o “abrazo mortal” (***deadlock***).
  - Inanición (***starvation***).

Exclusión Mutua Distribuida

- Requerimiento
  - Si P_i está ejecutando en su sección crítica (SC), entonces no hay otro proceso ejecutando en su SC.
  - Si varios procesos están esperando para entrar en la SC, mientras ninguno de ellos está en la misma, alguno de ellos deberá entrar en un tiempo finito.
  - El comportamiento de un proceso fuera de la SC o del protocolo que gobierna el acceso, no tiene influencia sobre el protocolo de exclusión mutua (hay independencia).
  - No existe proceso privilegiado.

Exclusión Mutua Distribuida

La exclusión mutua en un ambiente distribuido puede ser conseguida mediante dos familias de algoritmos:

- Basados en permisos.
- Basados en fichas (*tokens*).

Como caso especial se considera la forma centralizada dentro de un ambiente distribuidos.

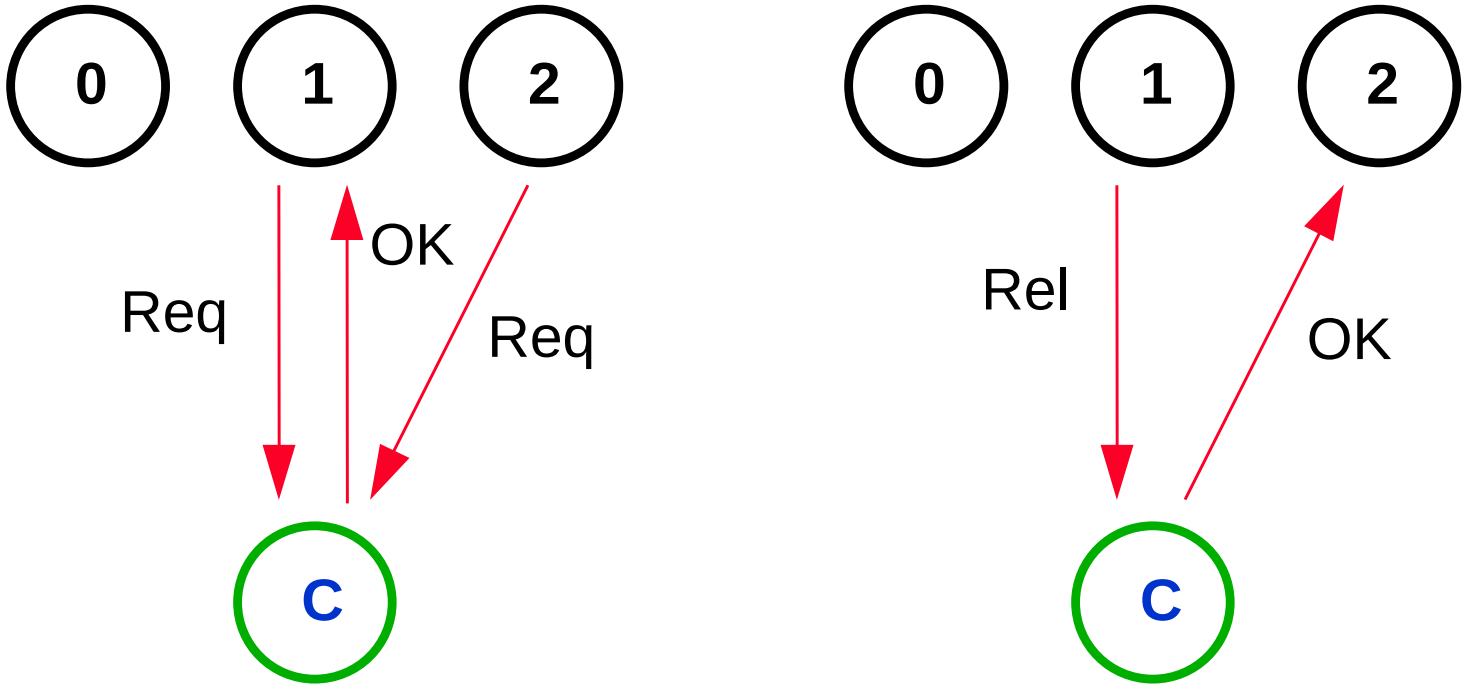
Exclusión Mutua Distribuida

Algoritmos centralizados para Exclusión Mutua

- Un nodo es asignado como nodo de control.
- Este nodo de control accede a todos los objetos compartidos.
- Solo el nodo de control toma decisiones sobre la asignación de los recursos compartidos.
- Toda la información necesaria es concentrada en el nodo de control.
- Si el nodo de control falla, la exclusión mutua se cae.

Exclusión Mutua Distribuida

Algoritmo Centralizado



Exclusión Mutua Distribuida

Ventajas:

- No hay inanición.
- No hay bloqueo.
- Fácil de implementar.

Desventajas:

- Único punto de falla.
- El coordinador es un “cuello de botella”.

Complejidad:

- Tres mensaje por entrada en la sección crítica .

Exclusión Mutua Distribuida

Algoritmos Distribuidos

En general todos los algoritmos distribuidos debieran cumplir con las siguientes pautas:

- Todos los nodos tiene igual cantidad de información, en promedio.
- Cada nodo tiene solo una visión parcial del sistema total y debe tomar decisiones en base a esta información.
- Todos los nodos tienen igual responsabilidad sobre la decisión final.
- Todos los nodos realizan el mismo esfuerzo, en promedio, para llegar a la decisión final.
- Falla en un nodo, en general, no resulta en un colapso total del sistema.
- No existe un reloj común con el cual regular el tiempo de los eventos.

Exclusión Mutua (Cont.)

Algoritmo Distribuido

El proceso que quiere entrar en su ***sección crítica*** envía un mensaje a los demás *n-1* procesos y a sí mismo.

Posibles respuestas de los *n-1* procesos restantes:

- Si el receptor no está en su sección crítica envía un OK al proceso emisor.
- Si el receptor está en la sección crítica no contesta. Pone el pedido en cola.
- Si quiere entrar en la sección crítica compara las estampillas de los mensajes, la menor (más antigua) gana.

Exclusión Mutua Distribuida

Algoritmo Distribuido

El número de mensajes es ***2(n-1)***

Desventaja: hay *n* puntos de fallas.

Exclusión Mutua (cont.)

Comportamiento

- Está libre de interbloqueos.
- Está libre de inanición dado que la entrada en la sección crítica está planificada de acuerdo a un ordenamiento basado en estampillas de tiempo. Este tipo de ordenamiento asegura que los procesos son atendidos en orden primero en llegar-primero en ser servido (FIFO).
- El número de mensajes por entrada a la sección crítica es:

$$2(n-1)$$

Este es el mínimo número de mensajes requeridos por entrada a la sección crítica cuando los procesos actúan independientemente y concurrentemente.

Exclusión Mutua (Cont.)

Consecuencias indeseables

- Los procesos necesitan conocer la identidad de todos los otros procesos en el sistema, lo cual hace el agregado y remoción dinámica de procesos más complejo.
- Si uno de los procesos falla el esquema completo colapsa. Esto puede manejarse con un continuo monitoreo del estado de todos los procesos del sistema.
- Los procesos que no han entrado en su sección crítica deben pausar frecuentemente para asegurar a otros procesos la entrada en la sección crítica. Este protocolo es adecuado para conjuntos pequeños y estables de procesos cooperativos.

Exclusión Mutua (Cont.)

Algoritmos de Pasaje de Ficha

- Se pasan una ficha (*token*) entre los procesos participantes.
- La ficha es una entidad que en algún momento es retenida por un proceso.
- El proceso que retiene la ficha puede entrar a su sección crítica sin pedir permiso.
- Cuando un proceso deja su sección crítica, pasa la ficha a otro proceso.

Exclusión Mutua (Cont.)

Ejemplo de Algoritmo de Pasaje de Ficha

Fue propuesto por Suzuki en 1982. Para este algoritmo se necesitan dos estructuras de datos:

La ficha que pasa de proceso en proceso es un arreglo cuyo k-ésimo elemento registra la estampilla de tiempo de la última vez que la ficha visitó el proceso P_k.

Cada proceso mantiene un arreglo de requerimientos, cuyo j-ésimo elemento registra la estampilla de tiempo del último requerimiento de P_j.

Inicialmente la ficha se asigna arbitrariamente a cualquier proceso. Cuando un proceso quiere entrar en su SC, puede hacerlo si posee la ficha, sino hace un broadcast de requerimiento con su estampilla de tiempo a todos los demás procesos y espera hasta recibir la ficha.

Exclusión Mutua (Cont.)

Ejemplo de Algoritmo de Pasaje de Ficha (cont)

Cuando el proceso P_j deja su SC , debe transmitir la ficha a algún otro proceso. Elige el próximo proceso para recibir la ficha buscando en el arreglo de requerimientos en el orden j+1,j+2,….,1,2,….,j-1 por el primer entrada en arreglo de requerimientos [k] tal que la estampilla de tiempo del último requerimiento de P_k por la ficha sea mas grande que el valor registrado en la ficha para P_k (requerimiento[k]>ficha[k]).

El algoritmo requiere:

- N mensajes (N-1 para hacer broadcast del requerimiento y 1 para transferir la ficha) cuando el proceso no posee la ficha.
- No tiene costo en mensajes si el proceso tiene la ficha.

Exclusión Mutua (Cont.)

Algoritmo de Pasaje de Ficha en Anillo

Los procesos tienen una conexión lógica en anillo.

Una ficha recorre los procesos en un solo sentido en forma circular.

El proceso que quiere entrar en su ***sección crítica*** espera tener la ficha y la retiene mientras procesa su sección crítica.

Exclusión Mutua (Cont.)

Desventaja: puede perderse la ficha o caer un proceso.

Ventaja: no hay inanición.

Para detectar la pérdida de la ficha circulan dos fichas.

Exclusión Mutua (Cont.)

Sean las dos fichas *ping* y *pong*

Se genera un invariante tal que

nping + *npong* = 0

Inicialmente *nping* = 1 y *npong* = -1

En cada proceso existe una variable entera local *m* que memoriza la última ficha que vió pasar.

Exclusión Mutua (Cont.)

Entonces para el proceso P_i :

Si recibe (*ping,nping*) **entonces**
  Si *m=nping* **entonces** <pong está perdido>
    nping:=*nping*+1;
    npong:=-*nping*;
  sino
    m:=*nping*;

Si recibe (*pong,npong*) ..... es simétrico

Exclusión Mutua (Cont.)

Algoritmo Distribuido para Exclusión Mutua basado en Topología Arbórea

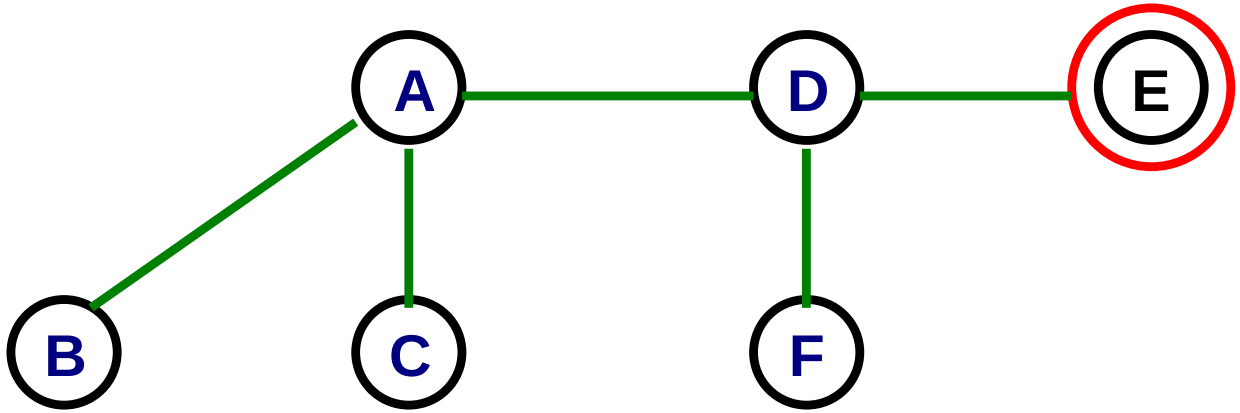
(Kenny-Raimond, 1981)

La posesión del *ticket* implica permiso para entrar en la sección crítica.

Cada nodo se comunica con el vecino solamente.

Cada nodo tiene la ubicación del ticket H_i.

Exclusión Mutua (Cont.)

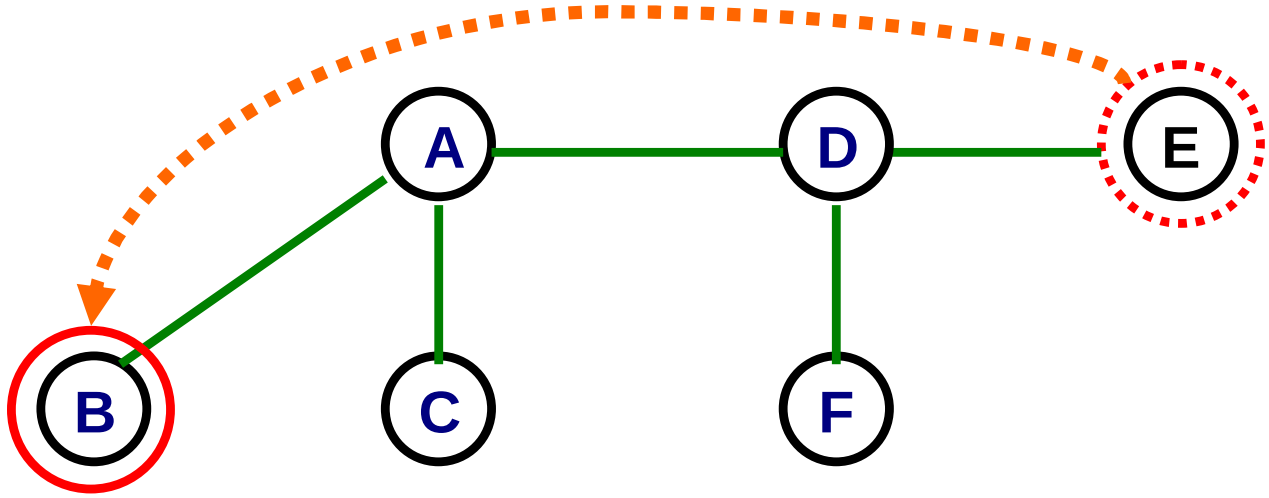


$H_A \leftarrow D$ $H_C \leftarrow A$ $H_E \leftarrow$ "self"

$H_B \leftarrow A$ $H_D \leftarrow E$ $H_F \leftarrow D$

Exclusión Mutua (Cont.)

Ejemplo 1: B pide el “token”

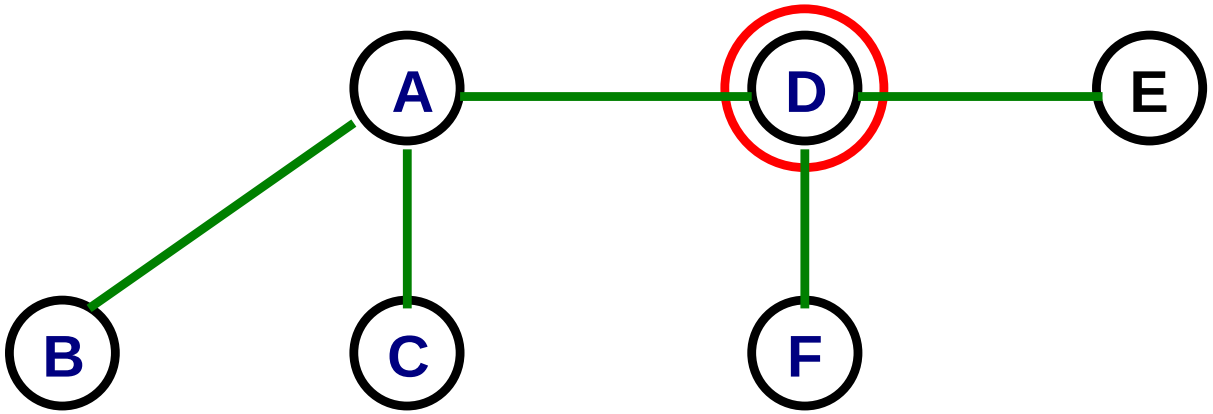


Estado Final

$H_A \leftarrow B$ $H_C \leftarrow A$ $H_E \leftarrow D$

$H_B \leftarrow$ “self” $H_D \leftarrow A$ $H_F \leftarrow D$

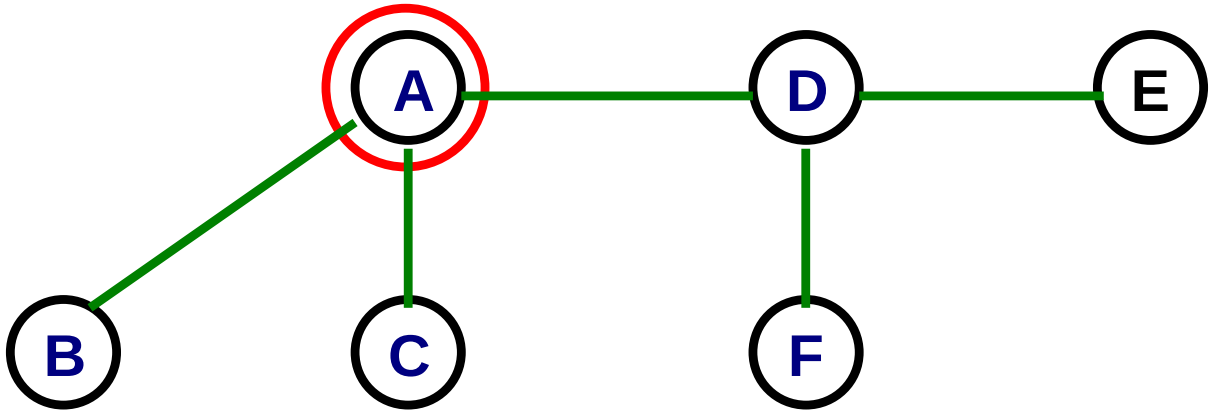
Exclusión Mutua (Cont.)



$H_A \leftarrow D$ $H_C \leftarrow A$ $H_E \leftarrow D$

$H_B \leftarrow A$ $H_D \leftarrow$ "self" $H_F \leftarrow D$

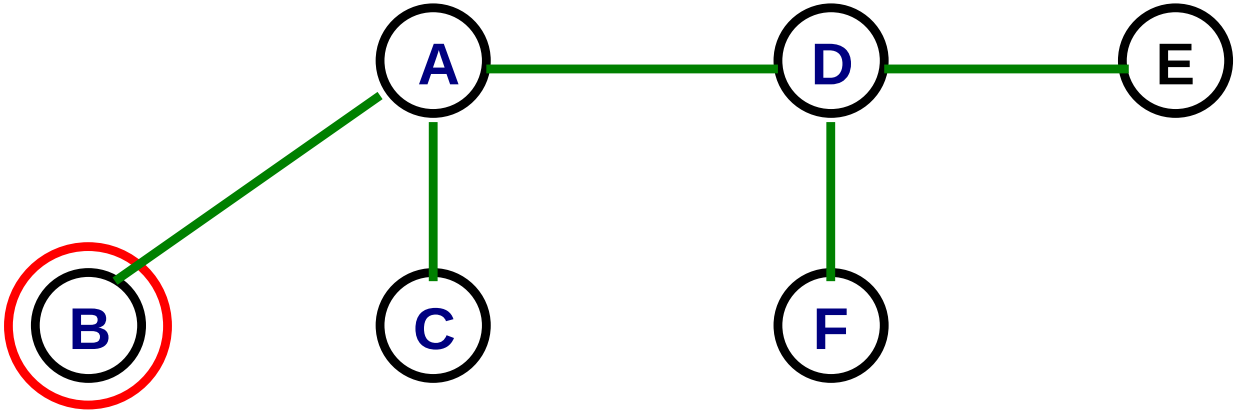
Exclusión Mutua (Cont.)



$H_A \leftarrow$ "self" $H_C \leftarrow A$ $H_E \leftarrow D$

$H_B \leftarrow A$ $H_D \leftarrow A$ $H_F \leftarrow D$

Exclusión Mutua (Cont.)



$H_A \leftarrow B$ $H_C \leftarrow A$ $H_E \leftarrow D$

$H_B \leftarrow$ "self" $H_D \leftarrow A$ $H_F \leftarrow D$

Algoritmos de Elección

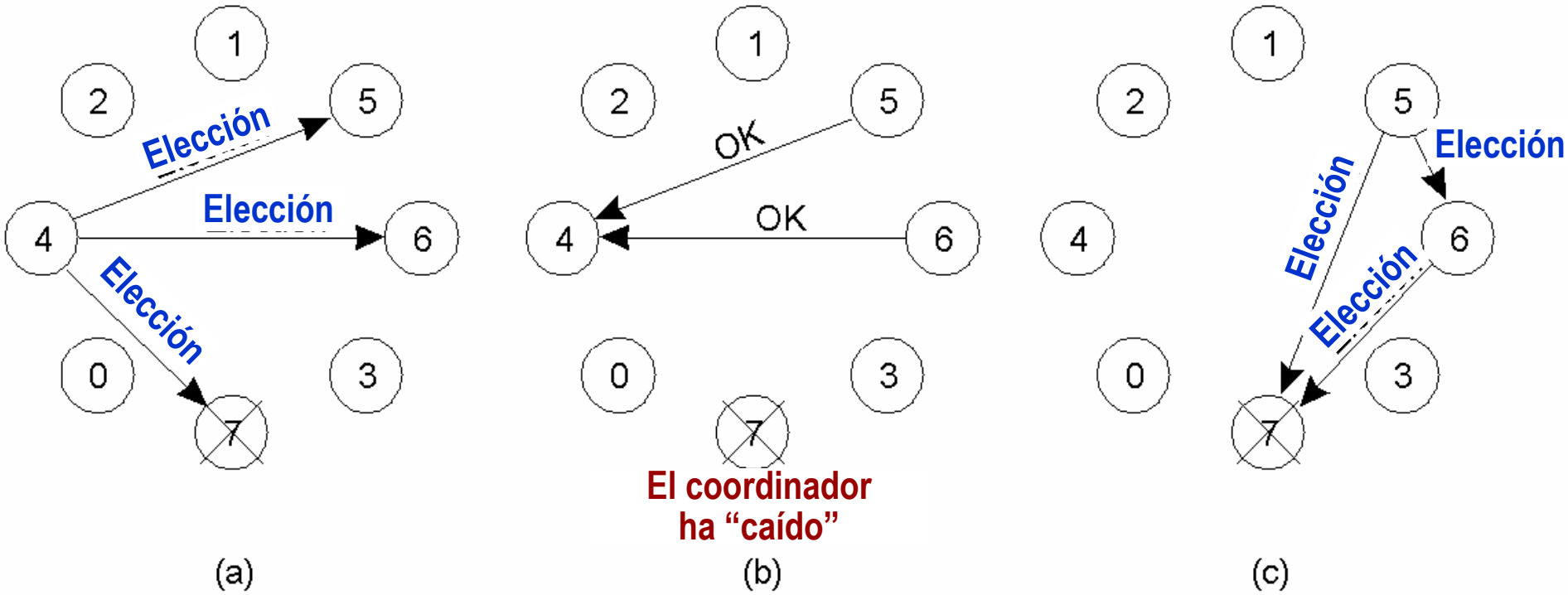
Algoritmos de Elección

Cuando el coordinador no responde a los requerimientos, se inicia un proceso de elección.

Algoritmo "bully"

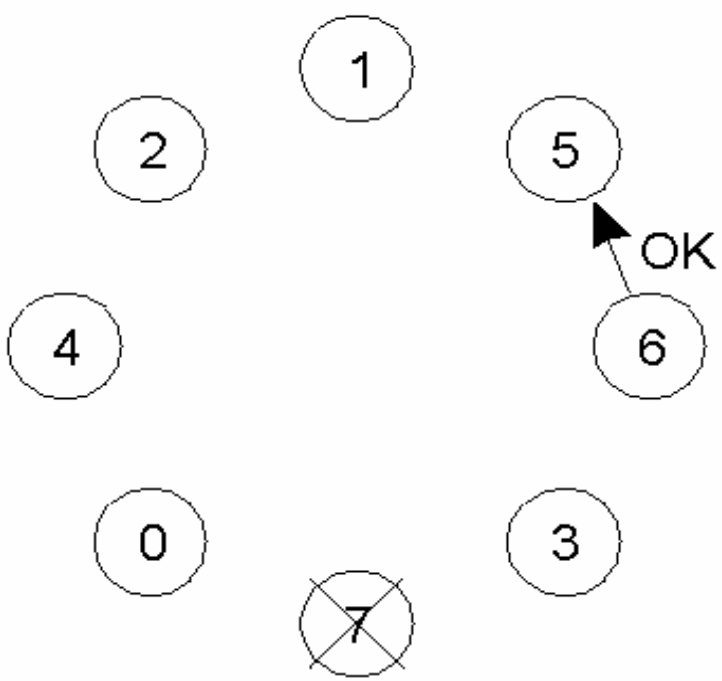
- P_i envía un mensaje ELECCION a cada proceso con número más grande.
- Si no responde nadie P_i gana la elección.
- Si un proceso mayor P_j responde, P_i queda afuera y P_j *broadcast* que él es el coordinador.

Algoritmos de Elección (Cont.)

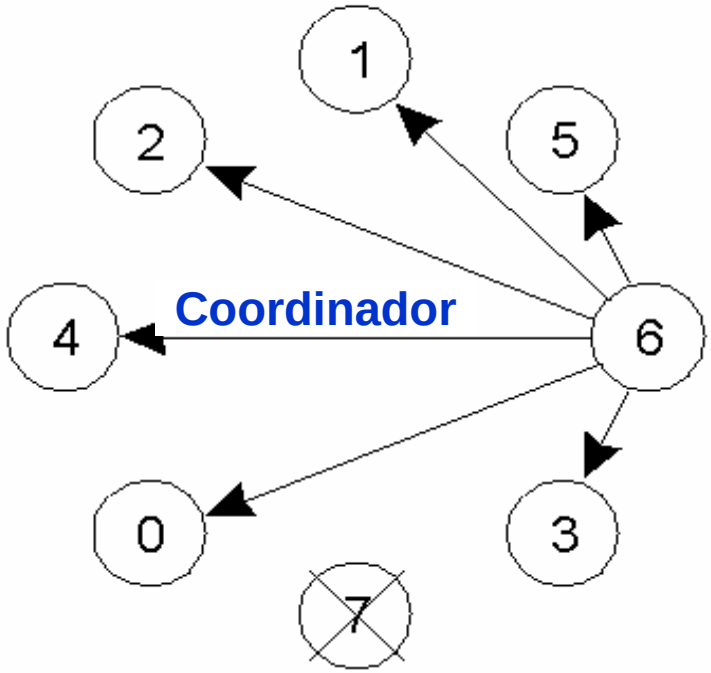


a) El proceso 4 inicia una elección.
b) Los procesos 5 y 6 responden, entonces 4 para.
c) Ahora 5 and 6, cada uno, inicia una elección.

Algoritmos de Elección (Cont.)



(d)



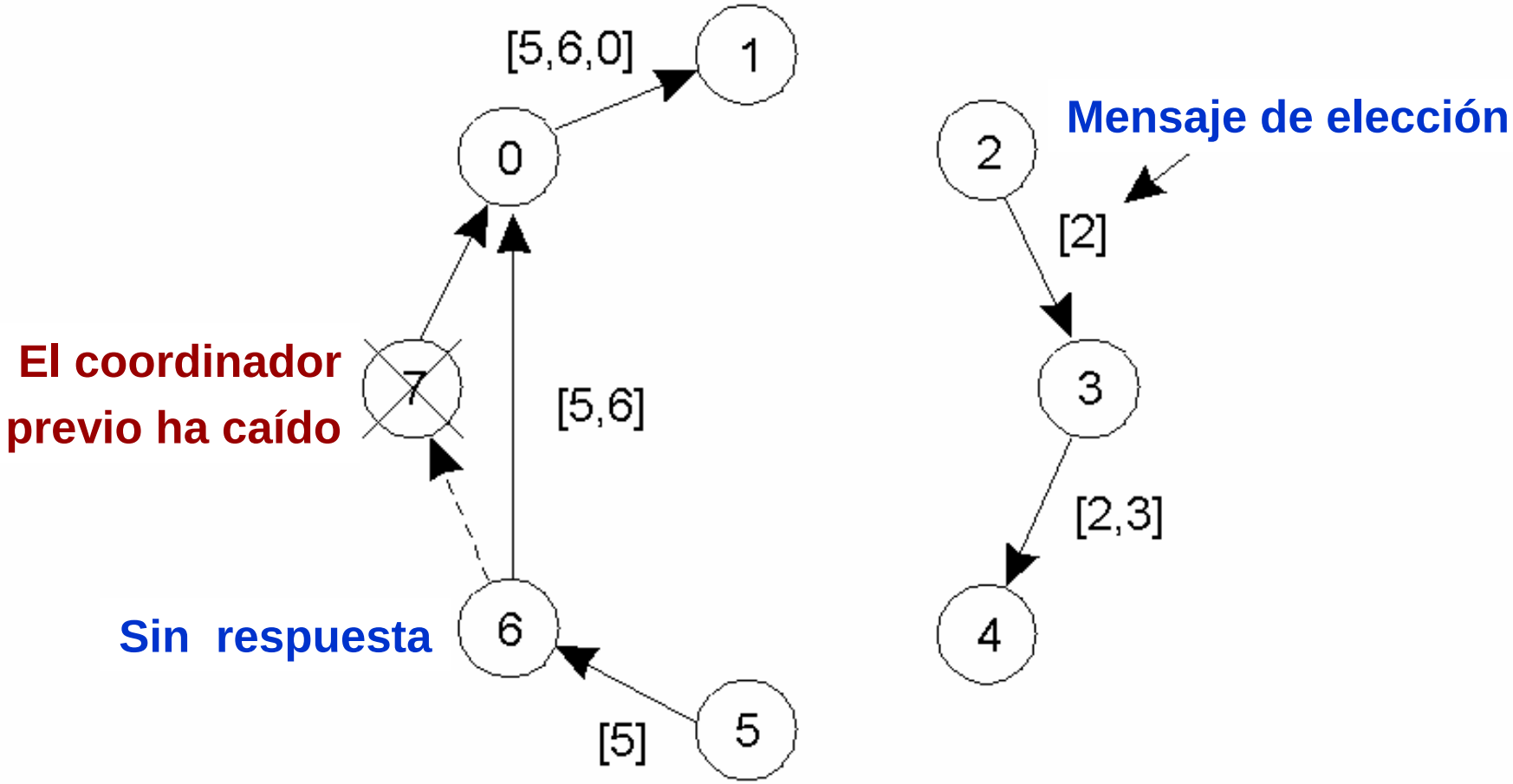
(e)

d) El proceso 6 le dice al 5 que pare.

e) El proceso 6 gana y le avisa a todos.

Algoritmos de Elección (Cont.)

Algoritmo de anillo



Algoritmos de Elección (Cont.)

Algoritmo de anillo

Cada proceso conoce su sucesor (vecino en el anillo)

Algún proceso detecta que el coordinador no funciona entonces envía un mensaje *elección* a su sucesor, incorporando su número.

En cada paso cada proceso agrega su número.

El mensaje vuelve (lo reconoce porque está su propio número).

El siguiente mensaje es con el nombre del coordinador.

Alcance de Acuerdo

- Hay aplicaciones donde un conjunto de procesos desea acordar sobre un “valor” común.
- Tales acuerdos pueden no tener lugar debido a:
  - Fallas en el medio de comunicación
  - Procesos que fallan:
    - Los procesos pueden enviar mensajes incorrectos o mal escritos a otros procesos.
    - Un subconjunto de procesos puede colaborar con otro en un intento de derrotar al esquema.

Falla en las Comunicaciones

- El proceso P_i en el sitio A, ha enviado un mensaje al proceso P_j de B; para proceder, P_i necesita saber si P_j ha recibido el mensaje.
- Detección de fallas usando un esquema de time-out:
  - Cuando P_i envía un mensaje, también especifica un intervalo de tiempo durante el cual espera recibir un mensaje de ACK de P_j.
  - Cuando P_j recibe el mensaje, inmediatamente envía ACK a P_i.
  - Si P_i recibe el ACK dentro del tiempo especificado en el intervalo de tiempo, concluye que P_j ha recibido su mensaje. Si un time-out ocurre, P_j necesita retransmitir su mensaje y esperar por su ACK.
  - Continúe hasta que P_i reciba un ACK, o sea notificado por el sistema que B está caído.

Falla en las Comunicaciones (Cont.)

- Suponga que P_j también necesita saber que P_i ha recibido su mensaje de ACK, para decidir sobre como proceder.
  - En la presencia de falla, no es posible cumplir con esta tarea.
  - No es posible en un medio distribuido por los procesos P_i y P_j estar de acuerdo completamente sobre sus respectivos estados.

Fallas de Procesos

Problema de los Generales Bizantinos

- El medio de comunicación es confiable, pero los procesos pueden fallar de modo impredecible.
- Considere un sistema de **n** procesos, de los cuales no más de **m** están fallados. Suponga que cada proceso P_i tiene un valor privado V_i.
- Se corre un algoritmo que permite que cada P_i no fallado construya un vector $X_i = (A_{i,1}, A_{i,2}, \ldots, A_{i,n})$ tal que:
  - Si P_j es un proceso no fallado, entonces $A_{ij} = V_j$.
  - Si P_i y P_j son procesos no fallados, entonces $X_i = X_j$.
- La solución comparte las siguientes propiedades.
  - Un algoritmo correcto puede ser corrido solo si $n \geq 3m + 1$.
  - El retardo del peor caso para el alcance de acuerdo es proporcional a $m+1$ mensajes.

Fallas de Procesos (Cont.)

- Un algoritmo para el caso donde m=1 y n=4 requiere dos rondas de intercambio de información:
  - Cada proceso envía su valor privado a los otros 3 procesos.
  - Cada proceso envía la información obtenida en la primer ronda a todos los otros procesos.
- Si un proceso fallado no envía mensajes, un proceso no fallado puede elegir un valor arbitrario y pretender que ese valor fue enviado por ese proceso.
- Completadas las dos rondas, un proceso no fallado puede construir su vector $X_i = (A_{i,1}, A_{i,2}, A_{i,3}, A_{i,4})$ como sigue:
  - $A_{i,j} = V_j$.
  - Para $j \neq i$, si al menos dos de los tres valores reportados por el proceso P_j coinciden, entonces se usa el valor de mayoría para iniciar el valor A_{ij}, sino se usa ***nil***.

Interbloqueos Distribuidos

Interbloqueos (Deadlocks)

El uso de un recurso implica:

- Requerimiento del recurso.
- Asignación del recurso.
- Des-asignación del recurso.

Los recursos pueden ser *reusables* o *consumibles*.

Interbloqueos Distribuidos (Cont.)

Condiciones *Necesarias* para un Interbloqueo:

- Exclusión Mutua.
- Retención y espera.
- No apropiación.
- Espera circular.

Si alguna de ellas no se cumple no hay interbloqueo.

Interbloqueos Distribuidos (Cont.)

Modelado de Interbloqueos

Grafo dirigido: es un par (N,E) donde N es un conjunto no vacío de nodos y E es un conjunto de lados dirigidos. Un lado dirigido es un par (*a,b*) donde $a,b \in$ N.

Camino: Un camino es una secuencia de nodos (*a,b,c,…,i,j*) de un grafo dirigido tales que (*a,b*), (*b,c*), …, (*i,j*) son lados dirigidos.

Ciclo: Un ciclo es un camino donde el primer y último nodo son los mismos.

Interbloqueos Distribuidos (Cont.)

Conjunto alcanzable: el conjunto alcanzable de un nodo *a* es el conjunto de todos los nodos *b* tales que existe un camino de *a* a *b*.

Nudo: Un nudo es un conjunto no vacío K de nodos tales que el conjunto alcanzable de cada nodo en K es exactamente el conjunto K.

Interbloqueos Distribuidos (Cont.)

Estrategias

- Prevención.
- Evasión.
- Detección.

La prevención se hace naturalmente.

La evasión es muy costosa.

Ambas son estrategias “pesimistas”, solo se intenta la detección y la recuperación.

Interbloqueos Distribuidos (Cont.)

Prevención de interbloqueos

- Adquiriendo todos los recursos que necesita el proceso para proceder.
- Asignación de números a los recursos, el proceso puede tomar los recursos con números mayores a los que tiene; si requiere uno menor debe soltar los que mantiene.
- Se les asigna número de prioridad a los procesos.

Interbloqueos Distribuidos (Cont.)

Ejemplo de prevención de interbloqueos: BDD

T_1: lock A;
lock B;
comienza la transacción ;
unlock B;
unlock A;

Si hay otra transacción T_2 sobre A y B y cada una de ellas adquiere un lock se puede producir una situación de interbloqueo.

Interbloqueos Distribuidos (Cont.)

Otros esquemas basados en estampillas de tiempo:

Wait-die: se basa en un método no apropiativo.

Cuando un proceso P_i requiere un recurso que tiene P_j, P_i espera ssi $e(P_i) < e(P_j)$ (P_i es más **viejo** que P_j).

De otra manera P_i es abortado.

$LC_i < LC_j \rightarrow$ halt P_i (*wait*)

$LC_i \geq LC_j \rightarrow$ kill P_i (*die*)

Interbloqueos Distribuidos (Cont.)

Wound-wait: se basa en un método apropiativo.

Cuando un proceso P_i requiere un recurso que tiene P_j, P_i espera sssi $e(P_i) > e(P_j)$ (P_i es mas **jóven** que P_j).

De otra manera P_i es abortado (P_j es "herido" por P_i).

$$LC_i < LC_j \rightarrow \text{kill } P_j \text{ (}\textit{wound}\text{)}$$

$$LC_i \geq LC_j \rightarrow \text{halt } P_i \text{ (}\textit{wait}\text{)}$$

Ambos esquemas evitan inanición.

En ambos esquemas el proceso retrasado recibe luego la misma estampilla.

Interbloqueos Distribuidos (Cont.)

Detección y recuperación de interbloqueos

Se mantiene el grafo de espera se busca la presencia de ciclos.

- Centralizados.
- Distribuidos.
- Jerárquicos.

Interbloqueos Distribuidos (Cont.)

Centralizado:

Debe construirse el grafo en diferentes puntos en tiempo:

1. Siempre que aparezca un borde nuevo o desaparezca uno.
2. Periódicamente.
3. Cuando el coordinador busque por ciclos

Problema: se pueden producir interbloqueos “fantasmas”.

Interbloqueos Distribuidos (Cont.)

Distribuido:

1. Cada sitio tiene una copia global.
2. Pasaje de mensajes de prueba.

Jerárquico:

Se arma un árbol de nodos con una jerarquía establecida.

Interbloqueos Distribuidos (Cont.)

Modelos de requerimiento

- Interbloqueos de recursos
- Interbloqueos de comunicaciones

La diferencia real entre ambos tipos es que el primero usa la condición AND y la segunda la condición OR.

Condición **AND**: Un proceso que requiere recursos para su ejecución puede proceder cuando ha adquirido todos.

Interbloqueos Distribuidos (Cont.)

Condición **OR**: un proceso que requiere recursos para su ejecución puede proceder cuando ha adquirido alguno de ellos.

Se usa OR para interbloqueo de comunicaciones porque un proceso puede estar esperando un mensaje de más de una fuente (proceso). No es determinístico.

El modelo AND-OR es otra generalización.

Interbloqueos Distribuidos (Cont.)

Condiciones de Interbloqueo

Interbloqueos de recursos única instancia (condición AND)

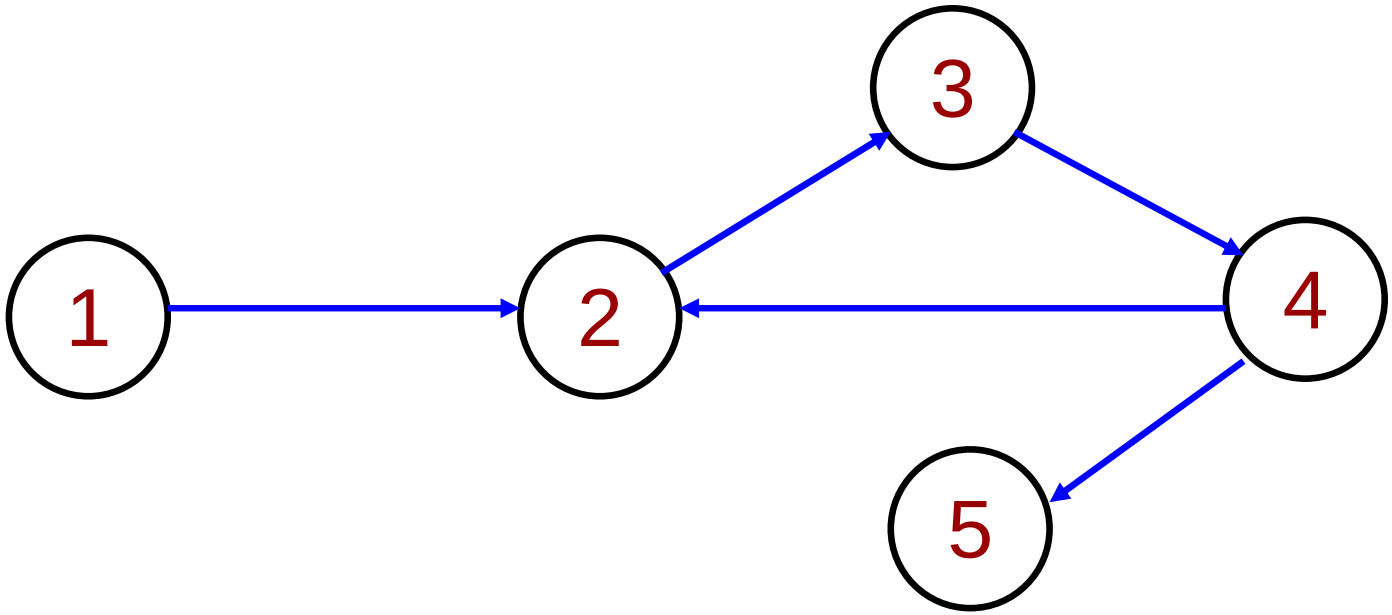
- Con detectar ciclo es suficiente.

Interbloqueos de comunicaciones (condición OR). Con la detección de ciclos no es suficiente

- Es necesario detectar nudos.

Problema: interbloqueos fantasmas.

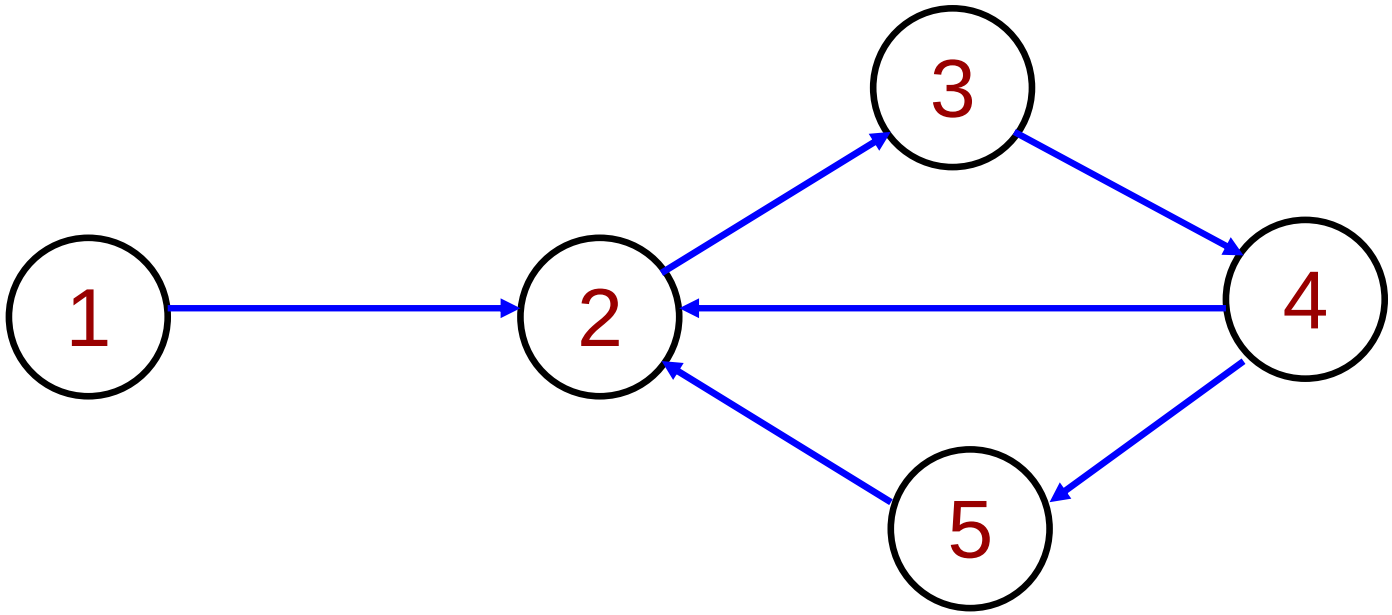
Interbloqueos Distribuidos (Cont.)



No hay interbloqueo

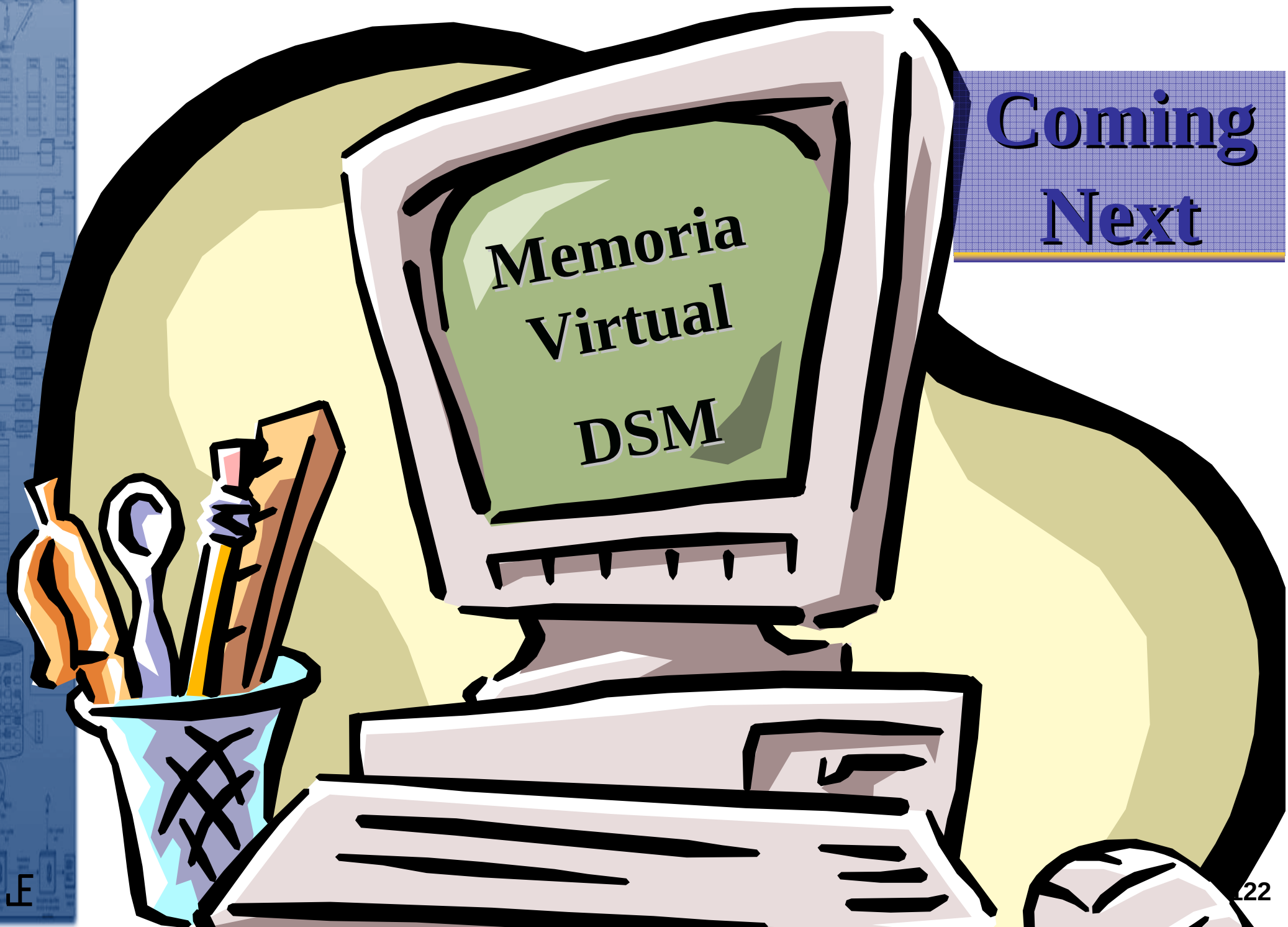
(no hay nudo)

Interbloqueos Distribuidos (Cont.)



Hay interbloqueo

(nudo: {2,3,4,5})



Coming Next