Lección 6: Sincronización de procesos mediante semáforos

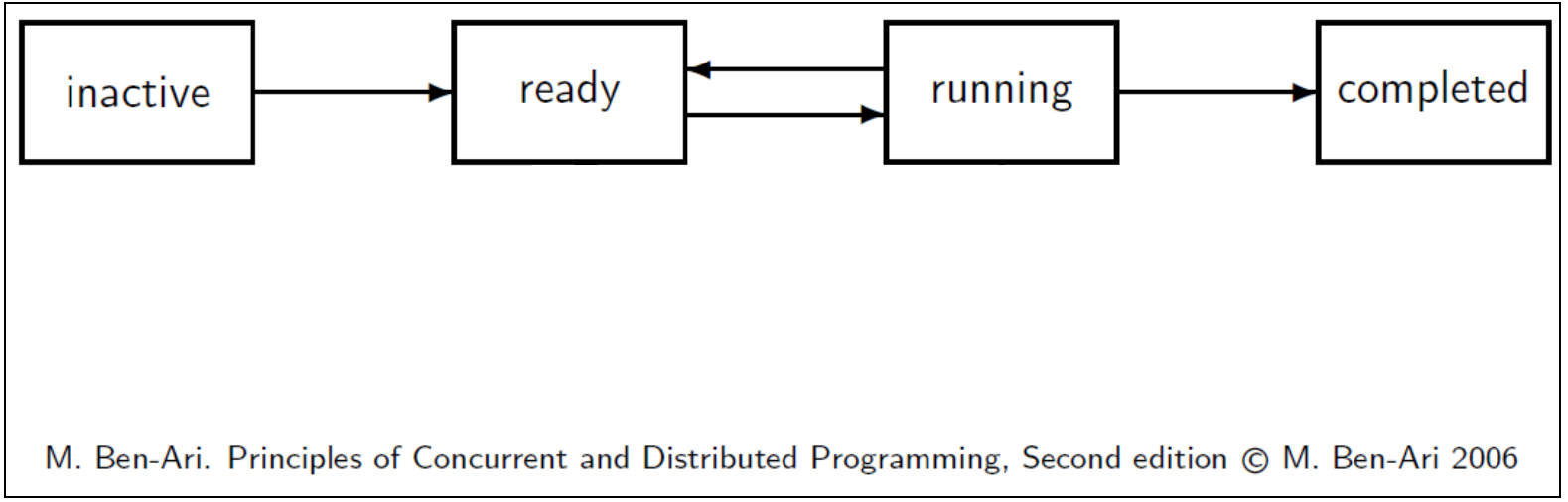
- Introducción
- Estados de un proceso
- Semáforos: sintaxis y semántica
- Técnicas de programación con semáforos:
  - el problema de la sección crítica
  - el problema de la cena de los filósofos
- La técnica del paso del testigo
  - El problema de la cena de los filósofos

Introducción

- La sincronización por espera activa tiene inconvenientes:
  - los algoritmos son difíciles de diseñar y verificar
  - se mezclan variables del problema a resolver y del método de resolución
  - en los casos de multi-programación, se ocupa innecesariamente el procesador "haciendo nada"
- Los semáforos son una de las primeras soluciones
  - son una solución conceptual, que puede implementarse de diferentes formas
  - propuestos por Dijkstra en 1968

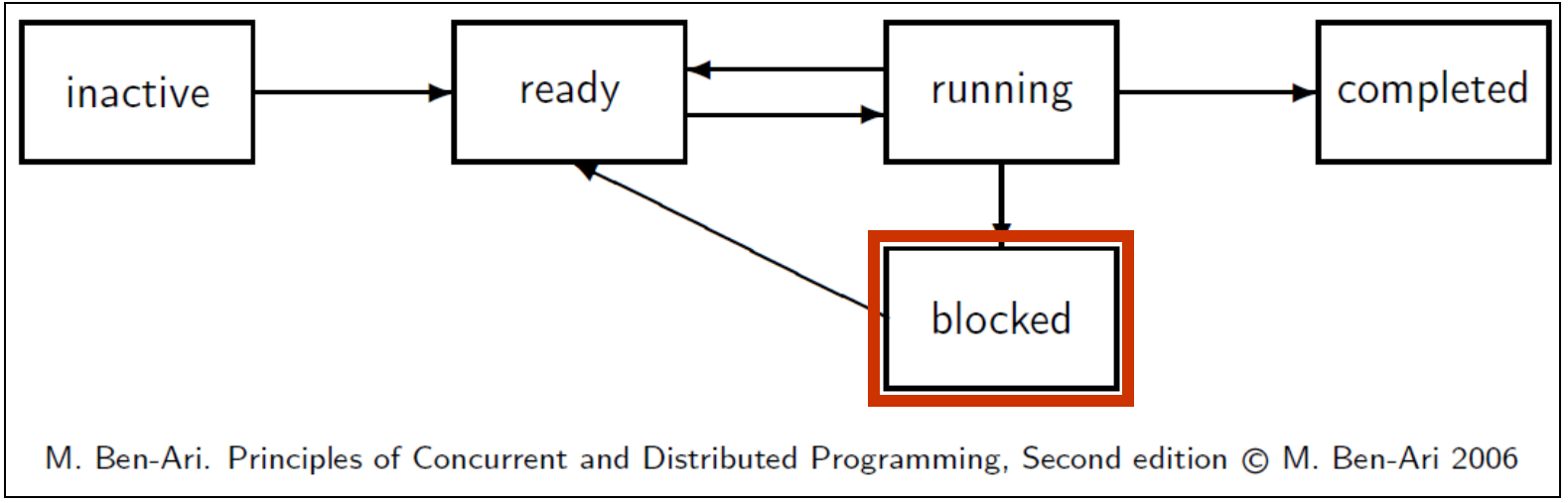
Estados de un proceso

- Para multitarea, y un proceso P, **`P.state`** es su estado



Estados de un proceso

- Para multitarea, y un proceso P, **`P.state`** es su estado



M. Ben-Ari. Principles of Concurrent and Distributed Programming, Second edition © M. Ben-Ari 2006

Semáforos: sintaxis y semántica

- **Semáforo**: instancia de un tipo abstracto de dato
  - **`S=(v,L)`**
- tal que ...
  - v es un natural
  - L es un conjunto de procesos
- con dos instrucciones atómicas...
  - **`wait`**
  - **`signal`**

Semáforos: sintaxis y semántica

- Declaración **semaphore S := (S_0,{})**

- Semántica de las operaciones (atómicas) cuando la invoca P:

wait(S)

```
< if S.v > 0
    S.v := S.v - 1
  else
    S.L := S.L + {P}
    P.state := blocked
>
```

Semáforos: sintaxis y semántica

- Semántica de las operaciones (atómicas) cuando la invoca P:

signal(S)

```
< if S.L = {}
    S.v := S.v + 1
  else
    S.L := S.L - {Q}
    Q.state := ready
>
```

- Propiedad: un semáforo S cumple el siguiente invariante

$$S.v = S_0 + \#signal(S) - \#wait(S)$$

- Existen también los "strong semaphores": "L" es una cola

El problema de la sección crítica

- Ejemplo: la sección crítica

	semaphore mutex := (1,{})		
	Process P	*Process Q*	
	while true	while true	
	SNC	SNC	
p1	wait(mutex)	wait(mutex)	q1
	SC	SC	
p2	signal(mutex)	signal(mutex)	q2

Semáforos: sintaxis y semántica

- Semántica alternativa para semáforos: versión "***busy-wait***"

```
semaphore S := (S0,{})
```

wait(S)

```
< await S.v > 0
   S.v := S.v - 1
>
```

signal(S)

```
< S.v := S.v + 1 >
```

Semáforos: sintaxis y semántica

- En realidad, S.L no hace falta, y podemos identificar S con S.v

```
semaphore S := S0
```

wait(S)

```
< await S > 0
    S := S - 1
>
```

signal(S)

```
< S := S + 1 >
```

Semáforos: sintaxis y semántica

- Existe un caso especial: los semáforos binarios

- Declaración

```
binary semaphore S := S0
```

  - con `S0: 0..1`

- Semántica de las operaciones (atómicas) cuando la invoca P:

wait(S)

```
< await S = 1
   S := 0 >
```

signal(S)

```
< await S = 0
   S := 1 >
```

Semáforos: sintaxis y semántica

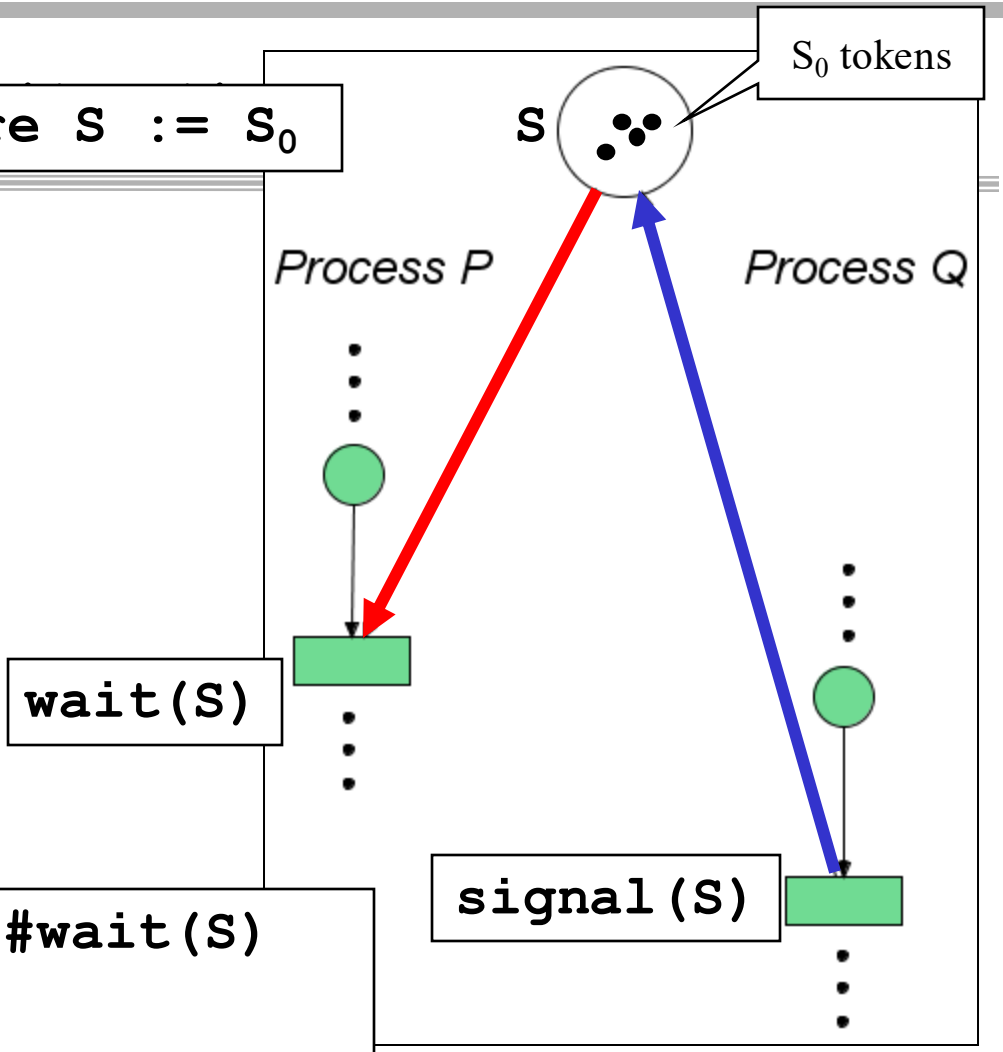
- Para binarios, existe una versión con una semántica alternativa (mutex):

signal(S)

```
< if S.v = 1
    ??????????
  else if S.L = {}
    S.v := 1
  else
   S.L := S.L - {Q}
   Q.state := ready
>
```

El problem

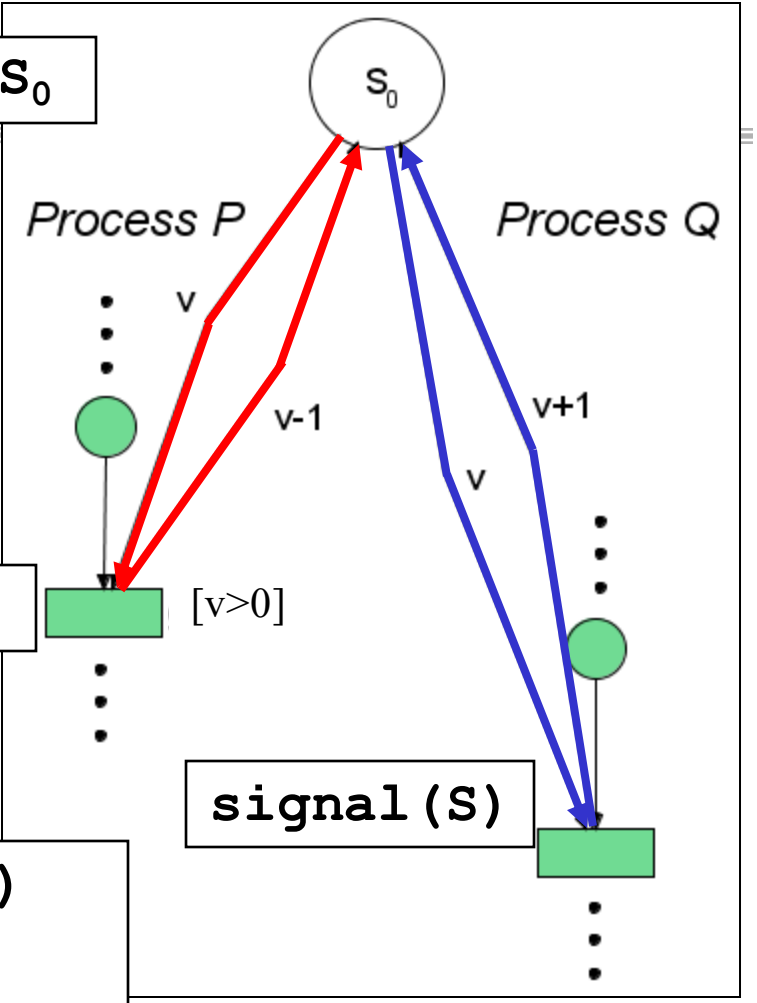
```
semaphore S := S0
```




```
M(S) = S0+#signal(S)-#wait(S)
M(S)>=0
```

El problem

semaphore S := S_0



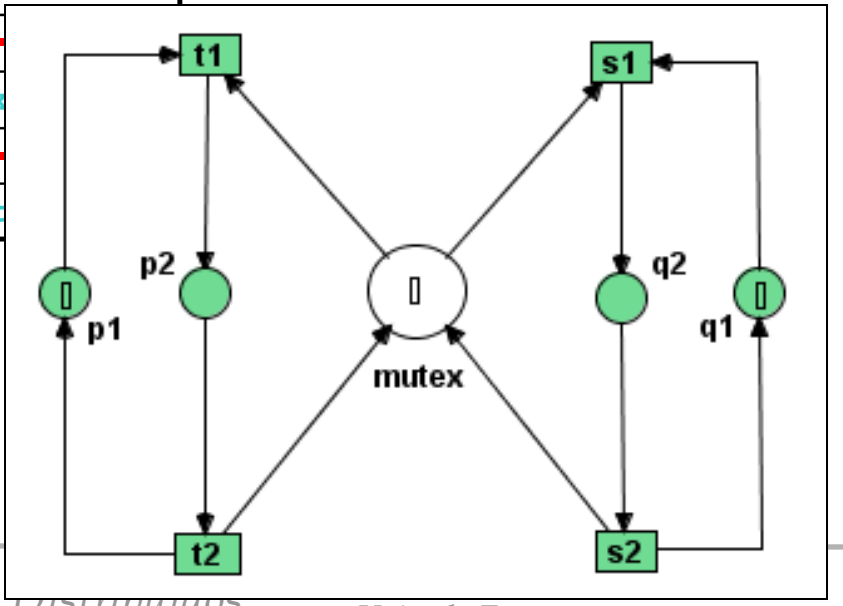
M(S) = S_0+#signal(S)-#wait(S)

M(S)>=0

El problema de la sección crítica

- Ejemplo: la sección crítica para dos procesos

	semaphore mutex := (1,{})	
	Process P	*Process Q*
	while true	while true
	~~SNC~~	~~SNC~~
p1	wait(mutex)	wait(mutex
	~~SC~~	~~SC~~
p2	signal(mutex)	signal(mut



- ¿Y para n procesos?

Ejemplos de diseño: máximo de un vector

- Considérese el código que se muestra a continuación, en el que se lanzan M procesos concurrentes con el objetivo de que entre todos encuentren el valor máximo de un vector
  - cada proceso se encarga de un trozo del vector.
- Una vez todos han acabado, el proceso informador muestra el valor máximo del vector
- Se pide completar el diseño de acuerdo a la especificación.

```
constant integer N := ... //lo que sea, >= 1
                 M := ... //lo que sea, >= 1
                 NM := N*M
integer array[1..NM] val := ... //lo que sea, ya inicializado
integer max
// código a completar

Process P(i: 1..M)::
    // código a completar
    // al terminar todos los procesos, "max" contiene el
    // valor máximo del vector "val"
end

Process informador::
    // código a completar
    write('Max= ', max)
end
```

```
//Pre:  1 <= i1 <= i2 <= NM
//Post: maxTrozo() = el valor máximo de v[i1],...,v[i2]
operation maxTrozo(integer array[1..NM] v, integer i1,i2): integer
```

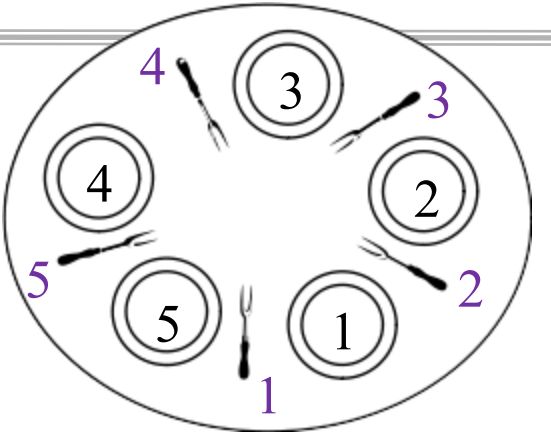
```
constant integer N := ... //lo que sea, >= 1
                 M := ... //lo que sea, >= 1
                 NM := N*M
integer array[1..NM] val := ... //lo que sea, ya inicializado
integer max
integer numHT := 0
Semaphore mutHT := 1
Semaphore todosHT := 0

Process P(i: 1..M)::
   integer maxLocal := maxTrozo(val,(i
   wait(mutHT)

   signal(muHT)
end
```

```
numHT++
if numHT=1
   max := maxLocal
else
   if maxLocal > max
      max := maxLocal
   end
end
if numHT=M
   signal(todosHT)
end
```

```
Process informador()::
   wait(todosHT)
   write("Max=",max)
end
```

Ejemplos de diseño: el problema de los filósofos

- Dijkstra 65, Hoare 85
  - Prototipo de sistema en que los procesos comparten recursos conservativos
- Hay 5 filósofos sentados alrededor de una mesa para comer espaguetis
- Hay 5 tenedores, e infinita pasta
- Hacen falta dos tenedores para comer
  - Cada filósofo puede usar los que tiene más cerca
  - Cada filósofo coge primero el de su izda., y luego el de su dcha.
- Se pide un programa que simule el comportamiento del sistema




```
Process filosofo(i:1..5):
   while true
      //piensa
      //coge tenedores
      //come
      //deja tenedores
   end
end
```

El problema de la cena de los filósofos

- Solución con semáforos:
  - Los recursos compartidos (tenedores) los representamos por medio de semáforos
  - El uso que hacen los procesos de esos recursos (coger/dejar) se "mapea" a las operaciones ofrecidas por los semáforos correspondientes (wait/signal)

```
Semaphore array[1..numFil] tDisponible := (1..numFil, 1)

Process filosofo(i:1..numFil)
  while true
    // Pensando
    wait(tDisponible[i])
    wait(tDisponible[1 + i mod numFil])
    // Comiendo
    signal(tDisponible[1 + i mod numFil])
    signal(tDisponible[i])
  end
end loop
```

Semáforos: sintaxis y semántica

- ¿Cuál es el matiz entre las distintas versiones de semáforos?
  - "normal", "strong", "busy-wait"
  - Pista: estudiar el comportamiento desde el punto de vista de la equidad en el problema de la sección crítica

signal(S)

```
if S.L = {}
   S.v := S.v + 1
else
   S.L := S.L - {Q}
   Q.state := ready
```

signal(S)

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

Un ejercicio sencillo

- Consideremos un esquema productor/consumidor en el que:
  - el productor produce un mensaje, que debe depositar en un buffer
  - el consumidor lee el mensaje del buffer
- Requisitos:
  - no se sobrescribe un mensaje
  - no se lee dos veces el mismo mensaje
  - el buffer tiene capacidad para un mensaje

```
                    string buffer

Process Productor::                    Process Consumidor::
   string paraEnviar                      string mensaje
   while true                             while true
      paraEnviar := produceMensaje()         mensaje := buffer
      buffer := paraEnviar                   write(mensaje)
   end                                    end
end                                    end
```

El paso del testigo

- Resolver problemas de sincronización complejos con semáforos es una tarea complicada
  - difícil obtener una solución
  - difícil asegurar la corrección
- La técnica del "paso del testigo" permite desarrollar, a partir de un diseño mediante instrucciones "**await**", una solución mediante semáforos
  - Diseñar con "**await**" (mucho más fácil y seguro)
  - Implementar con semáforos

El paso del testigo

- Veamos una forma sistemática de implementar exclusiones mutuas y sincronización por condición con semáforos (binarios)
- Necesitamos gestionar un conjunto de instrucciones que deban sincronizar, del siguiente tipo:

  I_i: <S_i> I_j:<await B_j S_j>

- Para ello:
  - Un semáforo (binario) para asegurar la ejecución en exclusión mutua de las instrucciones:

    Semaphore testigo := 1

  - Para cada :<await B_j S_j>

    Semaphore b_j := 0

    integer d_j := 0

El paso del testigo

- Cambiar **<S_i>** por

```
I_i: wait(testigo)
     S_i
     pasarTestigo(...)
```

- Cambiar **<await B_j S_j>** por

```
I_j: wait(testigo)
     if no B_j
        d_j := d_j+1
        signal(testigo)
        wait(b_j)
     end
     S_j
     pasarTestigo(...)
```

El paso del testigo

- Donde

```
Operation pasarTestigo(...)
    switch
        B1 AND d1>0: d1:=d1-1;signal(b1)
        ...
        Bn AND dn>0: dn:=dn-1;signal(bn)
        otherwise: signal(testigo)
    end
end
```

El paso del testi

- Ejercicio: implementar usando semáforos

```
constant integer N := ... //lo que sea
             M := ... //lo que sea
             NM := N*M
integer array[1..NM] val := ... //lo que sea
integer max

integer numHanAcabado := 0 //# procs han acabado

Process P(i: 1..M)::
   integer maxLocal

   maxLocal := maxTrozo(val,(i-1)*N+1,i*N)
   //proteger numHanAcabado y max
   ⟨
      numHanAcabado := numHanAcabado+1
      if numHanAcabado=1
         max := maxLocal
        else
           if maxLocal > max
              max := maxLocal
           end
      end
   ⟩
end

Process informador::
   ⟨await numHanAcabado = M
      write('Max= ', max)
   ⟩
end
```

El paso del testigo

- Ejercicio: implementar usando semáforos

```
integer s := 2

Process P1
    while true
        <await s >= 1
           s := s-1
        >
        //en la zona crít
        < s := s+1 >
    end
end

Process P2
    while true
        <await s >= 1
           s := s-1
        >
        //en la zona crítica
        < s := s+1 >
    end
end
```

```
Process P3
    while true
        <await s >= 2
           s := s-2
        >
        //en la zona crítica
        < s := s+2 >
    end
end

Process P4
    while true
        <await s >= 2
           s := s-2
        >
    end
end
```

```
bool array[1..n] libre := (1..n,TRUE)

Process filosofo(i:1..n)::
  while true
    //pensando
    <await libre[i] AND
          libre[1+i mod n]
       libre[i] := FALSE
       libre[1+i mod n] := FALSE
    >
    //comiendo
    < libre[i] := TRUE
       libre[1+i mod n] := TRUE
    >
    end
end
```

```
bool array[1..n] libre := (1..n,TRUE)
int array[1..n] d := (1..n,0)
Semaphore array[1..n] b := (1..n,0)
Semaphore testigo := 1
bool array[1..n] libre := (1..n,TRUE)
```

```
wait(testigo)
if NOT (libre[i] AND libre[1+i mod n])
   d[i]++
   signal(testigo)
   wait(b[i])
end
libre[i] := FALSE
libre[1+i mod n] := FALSE
pasarTestigo(i)
```

```
wait(testigo)
libre[i] := TRUE
libre[1+i mod n] := TRUE
pasarTestigo(i)
```

```
bool array[1..n] libre := (1..n,TRUE)
int array[1..n] d := (1..n,0)
Semaphore array[1..n] b := (1..n,0)
Semaphore testigo := 1
bool array[1..n] libre := (1..n,TRUE)
```

```
wait(testigo)
if NOT (libre[i] AND libre[1+i mod n])
   d[i]++
   signal(testigo)
   wait(b[i])
end
libre[i] := FALSE
libre[1+i mod n] := FALSE
pasarTestigo(i)
```

```
wait(testigo)
libre[i] := TRUE
libre[1+i mod n] := TRUE
pasarTestigo(i)
```

```
operation pasarTestigo(int i)
   bool testigoPasado = false
   int j := i+1
   while j<>i AND NOT testigoPasado
      if libre[j]
         AND libre[1+j mod n]
         AND d[j]>0
            d[j]--
            testigoPasado := TRUE
            signal(b[j])
      else
            j++
      end
   end
   if NOT testigoPasado
       signal(testigo)
   end
end
```