



## TEMA 3

### Semáforos



1. Introducción
2. Definición de Semáforo
3. Resolución de problemas con semáforos
4. Implementación de semáforos en JAVA
5. Problemas clásicos de concurrencia
6. Inconvenientes de los semáforos



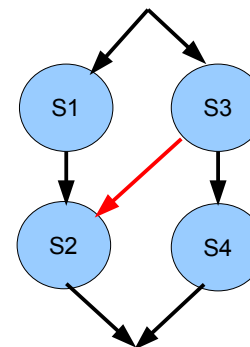
## 1. Introducción

- En 1965, a partir del algoritmo de Dijkstra se plantea el desarrollo de mecanismos eficientes y fiables para que los procesos puedan **cooperar**.
- El principio fundamental es que dos o más procesos puedan cooperar por medio de simples **señales**, de forma que un proceso pueda detenerse en una posición determinada hasta que reciba una señal.
- Cualquier situación de coordinación, aunque sea complicada puede resolverse con las señales adecuadas.
- Para la señalización se usan variables especiales llamadas **semáforos**.



## 1. Introducción

- Tenemos cuatro procesos S1, S2, S3 y S4. Los vamos a ejecutar concurrentemente, tal que satisfagan el siguiente grafo de precedencias:



```
cobegin
begin
  S1
  S2
end
begin
  S3
  S4
end
coend
```



## 2. Definición de Semáforo

- Un semáforo se puede definir como una variable tipo **semaphore**. Por tanto, puede ser usada para la definición de arrays y registros.

```
var
  sem: semaphore;
  sem_array: array[1..10] of semaphore;
type
  sem_record = record of
    s: semaphore;
    i: integer;
  end;
```

5



## 2. Definición de Semáforo

- Un semáforo es un **tipo abstracto de datos**. Consiste en unas estructuras de datos y en un conjunto de operaciones asociadas a tales estructuras.
- Estructuras de datos:**
  - Un **contador** entero positivo en el rango 0...infinito (teóricamente, ya que un ordenador tiene una capacidad limitada para representar números enteros)
  - Una **cola de procesos** esperando por ese semáforo.
- Operaciones.** Si **s** es una variable de tipo semáforo:
  - acquire(s)**
  - release(s)**
  - initial(s, valor)**

6



## 2. Definición de Semáforo

- Las operaciones **acquire** y **release** se excluyen mutuamente en el tiempo.
- La operación **initial** tan sólo está permitida en el cuerpo principal del programa (la parte que no es concurrente).
- Por el contrario, las otras dos operaciones sólo se permiten en procesos concurrentes.

7



## 2. Definición de Semáforo

### **acquire(s)**

- Si el contador del semáforo **s** es mayor que 0, se decrementa en 1 dicho contador y el proceso que hizo la llamada continúa ejecutándose.
- Si el contador del semáforo **s** es igual a 0, se lleva el proceso que hizo la operación a la cola asociada con el semáforo **s**. Esto significa que se suspende su ejecución, abandonando el procesador a favor de otro proceso.
- Observe que estos semáforos, por definición, **no pueden tener valores negativos**.

8



## 2. Definición de Semáforo

### release(s)

- Si el contador del semáforo **s** es mayor que 0, significa que no hay ningún proceso en la cola del semáforo **s**, y por tanto, incrementa en 1 dicho contador y **el proceso que ejecutó esta operación continua**.
- Si el contador del semáforo **s** es igual a 0 y hay procesos esperando en la cola del semáforo, se toma a uno de ellos y se le pone en un estado de preparado para ejecutarse. **El proceso que ejecutó la operación continúa** ejecutándose.
- Si el contador del semáforo **s** es igual a 0 y no hay procesos esperando en la cola del semáforo, se incrementa en 1 **el contador**. **El proceso que ejecutó la operación continúa** ejecutándose.

9



## 2. Definición de Semáforo

### initial(s,valor)

Esta operación pone el contador del semáforo **s** al valor indicado por el parámetro **valor**.

- Si el semáforo sólo admite como posibles valores el 1 y el 0 estamos hablando de un **semáforo binario**, en otro caso estamos hablando de un **semáforo general**.
- En *Pascal-FC* no hay diferencias entre semáforos binarios y generales. Es responsabilidad del programador que use un semáforo binario controlar que éste no tome valores mayores a 1.

10



## 2. Definición de Semáforo

- Las operaciones **acquire** y **release** pueden ser expresadas de la siguiente forma:

```
acquire(s) :
  if s > 0 then
    s := s - 1
  else
    bloquear proceso;
```

```
release(s) :
  if hay procesos bloqueados then
    desbloquear proceso
  else
    s := s + 1;
```

- Es evidente que **ambas** deben ejecutarse de forma atómica y **deben ser mutuamente excluyentes**.
- El bloqueo de los procesos se realiza con los mecanismos del sistema operativo.
- La **cola** de procesos bloqueados se puede gestionar de varias formas, pero lo normal es que sea una **FIFO**.

11



## 3. Resolución de problemas con semáforos

### Exclusión mutua

Usamos un semáforo binario **s** inicializado a **1**.

```
process P1;
begin
  ...
  acquire(s);
  Sección Crítica;
  release(s);
  ...
end;
```

```
process Pn;
begin
  ...
  acquire(s);
  Sección Crítica;
  release(s);
  ...
end;
```

12



### 3. Resolución de problemas con semáforos

#### Condición de Sincronización

- Asignamos un semáforo general a cada condición.
- Las **esperas** por la condición se realizan con **acquire**.
- Los **avisos** de disponibilidad se realizan con **release**.
- El **valor inicial** del semáforo será el de los **recursos disponibles inicialmente**.
- El **valor del semáforo** en cada instante será el de los **recursos disponibles en ese momento**.

13



### 3. Resolución de problemas con semáforos

#### Condición de sincronización

- Supongamos que **P2** no puede ejecutar **d** hasta que **P1** ejecuta **a**.

```
process P1;
begin
  a;
  b;
end;
```

```
process P2;
begin
  c;
  d;
end;
```

- Usamos un semáforo **s** inicializado a 0

```
process P1;
begin
  a;
  release(s);
  b;
end;
```

```
process P2;
begin
  c;
  acquire(s);
  d;
end;
```

14



### 3. Resolución de problemas con semáforos

#### Condición de sincronización

- Supongamos ahora tres procesos, y **P2** solo puede ejecutar **d** si **P1** ha ejecutado **a** o **P3** ha ejecutado **e**.

```
process P1;
begin
  a;
  b;
end;
```

```
process P2;
begin
  c;
  d;
end;
```

```
process P3;
begin
  e;
  f;
end;
```

- Usamos un semáforo **s** inicializado a 0

```
process P1;
begin
  a;
  release(s);
  b;
end;
```

```
process P2;
begin
  c;
  acquire(s);
  d;
end;
```

```
process P3;
begin
  e;
  release(s);
  f;
end;
```

15



### 3. Resolución de problemas con semáforos

#### Condición de sincronización

- Supongamos ahora tres procesos, y **P2** solo puede ejecutar **d** si **P1** ha ejecutado **a** y **P3** ha ejecutado **e**.
- Usamos un semáforo **s** inicializado a 0

```
process P1;
begin
  a;
  release(s);
  b;
end;
```

```
process P2;
begin
  c;
  acquire(s);
  acquire(s);
  d;
end;
```

```
process P3;
begin
  e;
  release(s);
  f;
end;
```

16



### 3. Resolución de problemas con semáforos

#### Condición de sincronización

- No obstante, si **P1** o **P3** se ejecutasen muy rápido la solución anterior podría fallar. Es más seguro usar dos semáforos **s** y **t** inicializados a 0.

```
process P1;
begin
  a;
  release(s);
  b;
end;
```

```
process P2;
begin
  c;
  acquire(s);
  acquire(t);
  d;
end;
```

```
process P3;
begin
  e;
  release(t);
  f;
end;
```

17



### 3. Resolución de problemas con semáforos

#### Condición de sincronización. Barreras

- Es un mecanismo de sincronización que obliga a procesos concurrentes (o distribuidos) a esperar a que **todos hayan llegado a un punto determinado**.
- Solo podrán continuar cuando todos los procesos hayan llegado a una barrera. El conjunto de los puntos de sincronización se denomina **barrera**.

18



### 3. Resolución de problemas con semáforos

#### Condición de sincronización. Barreras

- Si queremos que ambos se esperen en el punto central usamos dos semáforos binarios **s** y **t** inicializados a 0.

```
process P1;
begin
  a;
  release(s);
  acquire(t);
  b;
end;
```

```
process P2;
begin
  c;
  release(t);
  acquire(s);
  d;
end;
```

- ¿Qué pasaría si intercambiamos el orden de los **acquire** y los **release**?

19



### 3. Resolución de problemas con semáforos

#### Condición de sincronización. Barreras para N procesos

- La intención de uso de barreras genéricas para n procesos es poder implementar **sincronizaciones cíclicas** como la siguiente:

```
process Pi;
begin
  repeat
    tareas_previas_i
    barrier(n)
    resto_i
  forever
end;
```

- Cada proceso esperará a que los demás hayan llegado al mismo punto, solo así podrán continuar con la siguiente.
- La misma barrera puede ser **reusada cíclicamente**.

20



### 3. Resolución de problemas con semáforos

#### Condición de sincronización. Barreras para N procesos

- Estas barreras no pueden implementarse igual que las binarias.
- No tiene sentido tener un array de N semáforos y hacer N operaciones de **acquire** y **release**.
- Hay que solucionarlo con un **número limitado de semáforos**, y que no requiera que el número de operaciones de cada proceso sea proporcional al número de procesos concurrentes.

21



### 3. Resolución de problemas con semáforos

#### Condición de sincronización. Barreras para N procesos

```
program barreras
var
  contador:integer;
  llegar, salir: semaphore;

procedure barrier(n : integer);
begin
  acquire(llegar);
  Contador++; → debe ser atómico
  if contador<n then release(llegar)
  else release(salir)
  acquire(salir);
  Contador--; → debe ser atómico
  if contador>0 then release(salir)
  else release(llegar)
end;
```

```
process pi()
...

begin
  contador=0;
  initial(llegar,1);
  initial(salir,0);
  cobegin
    p0; p1 ;...
  coend
end.
```

22



### 3. Resolución de problemas con semáforos

#### Condición de sincronización. Barreras para N procesos

```
procedure barrier(n : integer);
begin
  acquire(llegar);
  acquire(mutex);
  contador++;
  if contador<n then release(llegar)
  else release(salir)
  release(mutex);
  acquire(salir);
  acquire(mutex);
  contador--;
  if contador>0 then release(salir)
  else release(llegar)
  release(mutex);
end;
```

```
process pi()
...
  mutex: semaphore;
begin
  Contador=0;
  initial(mutex,1);
  initial(llegar,1);
  initial(salir,0);
  cobegin
    p0; p1 ;...
  coend
end.
```

23



### 4. Implementación de semáforos en JAVA

```
package MiSemaforo;

public class SemaforoBinario {

  protected volatile int contador;

  public SemaforoBinario(int inicial) throws Exception{
    if(inicial!=0 && inicial!=1){
      throw(new Exception( message:"Imposible inicializar semaforo binario"));
    }
    contador=inicial;
  }

  public synchronized void ACQUIRE() throws InterruptedException {
    while (contador==0) {
      wait();
    }
    contador=0;
  }

  public synchronized void RELEASE() {
    contador = 1;
    notify();
  }
}
```

24



## 4. Implementación de semáforos en JAVA

```
package MiSemaforo;

public class SemaforoGeneral {
    protected volatile int contador;

    public SemaforoGeneral(int inicial) throws Exception {
        if(inicial<0){
            throw(new Exception( message:"Imposible inicializar semaforo general"));
        }
        contador=inicial;
    }

    public synchronized void ACQUIRE() throws InterruptedException {
        while (contador==0) {
            wait();
        }
        contador--;
    }

    public synchronized void RELEASE() {
        contador++;
        notify();
    }
}
```

25



## 4. Implementación de semáforos en JAVA

### Package java.util.concurrent.semaphore

```
import java.util.concurrent.semaphore;

Semaphore sem = new Semaphore(valor_inicial);

sem.acquire(); //acquire

sem.release(); //release
```

26



## 4. Implementación de semáforos

### Algoritmo de Barz

- Con el algoritmo de Barz se pueden simular semáforos generales, mediante semáforos binarios. Este algoritmo requiere dos semáforos binarios (mutex y gate) y una variable entera (value).
- Las funciones **generalAcquire** y **generalRelease** son las emulaciones genéricas de **acquire** y **release** respectivamente; k es el valor inicial del semáforo. El semáforo mutex asegura exclusión mutua para el acceso a value. El semáforo gate se usa para controlar qué procesos deben bloquearse o desbloquearse según el valor de value.

27



## 4. Implementación de semáforos

### Algoritmo de Barz

```
mutex, gate :semaphore;
valor:integer;

initial(mutex,1);
initial(gate,1);
valor=k;

Procedure generalAcquire()
begin
    acquire(gate);
    acquire(mutex);
    valor := valor-1;
    if valor > 0 then release(gate);
    release(mutex)
end

Procedure generalRelease()
begin
    acquire(mutex);
    valor := valor+1;
    if valor=1 then release(gate);
    release(mutex);
end
```

28



## 5. Problemas clásicos de concurrencia

### El problema del productor-consumidor

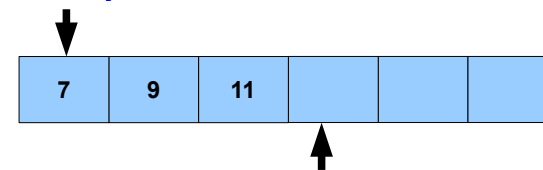
- Un proceso productor genera información que es usada por otro proceso consumidor.
- La comunicación se realiza a través de un *buffer* compartido.
- Se debe sincronizar el proceso para que el consumidor no intente consumir elementos que no se han producido.
- Existen dos posibilidades, usar un **buffer ilimitado**, o usar un **buffer limitado**. En este último caso, el productor debe esperar si no hay sitio libre en el *buffer*.

29



## 5. Problemas clásicos de concurrencia

### El problema del productor-consumidor



```
process Productor;
begin
  repeat
    producir elemento;
    protocolo de entrada;
    insertar elemento;
    protocolo de salida;
  forever
end;
```

```
process Consumidor;
begin
  repeat
    protocolo de entrada;
    extraer elemento;
    protocolo de salida;
    consumir elemento;
  forever
end;
```

30



## 5. Problemas clásicos de concurrencia

### El problema del productor-consumidor

```
program prodcon;
const
  buffmax = 4;
var
  buffer: array[0..buffmax] of char;
  nextin, nextout: integer;
  spacesleft, itemsready: semaphore;
  mutex: semaphore;
```

31



## 5. Problemas clásicos de concurrencia

### El problema del productor-consumidor

```
process producer;
var
  data: char;
begin
  for data := 'a' to 'z' do
    begin
      acquire(spacesleft);
      acquire(mutex);
      buffer[nextin] := data;
      nextin := (nextin + 1) mod (buffmax + 1);
      release(mutex);
      release(itemsready)
    end
  end;
end;
```

32





## 5. Problemas clásicos de concurrencia

### El problema del productor-consumidor

```

process consumer;
var
  data: char;
begin
  repeat
    begin
      acquire(itemsready);
      acquire(mutex);
      data := buffer[nextout];
      nextout := (nextout + 1) mod (buffmax + 1);
      release(mutex);
      release(spacesleft);
      write(data);
    end
  until data = 'z';
end;

```

33



## 5. Problemas clásicos de concurrencia

### El problema del productor-consumidor

```

begin
  initial(spacesleft, buffmax + 1);
  initial(itemsready, 0);
  initial(mutex, 1);
  nextin := 0;
  nextout := 0;
  cobegin
    producer;
    consumer
  coend
end.

```

34



## 5. Problemas clásicos de concurrencia

### El problema de los lectores y los escritores

- Existe un recurso que debe ser compartido por varios procesos concurrentes (B.D. o fichero)
- Hay una serie de procesos que solo quieren leer la información del recurso. Estos son los **procesos lectores**.
- Existe otra serie de procesos que desean actualizarlo, esto es leer y escribir en él. Estos procesos son los **escritores**.
- Los lectores pueden realizar acceso simultaneo.
- Los escritores necesitan acceso exclusivo.

35



## 5. Problemas clásicos de concurrencia

### El problema de los lectores y los escritores

- Según qué proceso tenga prioridad podemos tener dos versiones:
  - Prioridad en lectura. Ningún lector espera salvo que haya un escritor accediendo.
  - Prioridad en escritura. Una vez que un escritor muestra su necesidad de actualizar, ningún lector debe comenzar su lectura.

36



## 5. Problemas clásicos de concurrencia

### El problema de los lectores y los escritores

```
process type lector;
begin
  ...
  protocolo de entrada;
  leer del recurso;
  protocolo de salida;
  ...
end;
```

```
process type escritor;
begin
  ...
  protocolo de entrada;
  escribir en el recurso;
  protocolo de salida;
  ...
end;
```

```
var
  Lectores: array[1..NLEC] of lector;
  Escritores: array[1..NESC] of escritor;
cobegin
  for i := 1 to NLEC do Lectores[i];
  for i := 1 to NESC do Escritores[i];
coend
```

37



## 5. Problemas clásicos de concurrencia

### El problema de los lectores y los escritores

```
(* Prioridad en la lectura *)
program LectEscr;
const
  NLEC = 5;
  NESC = 2;
var
  nl: integer;
  mutex, escritura: semaphore;
```

38



## 5. Problemas clásicos de concurrencia

### El problema de los lectores y los escritores

```
process type Lectores(id:integer);
begin
  repeat
    acquire(mutex);
    nl:=nl+1;
    if nl=1 then acquire(escritura);
    release(mutex);
    {SECCIÓN CRÍTICA}
    acquire(mutex);
    nl:=nl-1;
    if nl=0 then release(escritura);
    release(mutex);
  forever
end;
```

39



## 5. Problemas clásicos de concurrencia

### El problema de los lectores y los escritores

```
process type Escritores(id:integer);
begin
  repeat
    acquire(escritura);
    {SECCIÓN CRÍTICA}
    release(escritura);
  forever
end;
```

40



## 5. Problemas clásicos de concurrencia

### El problema de los lectores y los escritores

```

var
  Escritor: array[1..NESC] of Escritores;
  Lector: array[1..NLEC] of Lectores;
  i:integer;

begin
  nl:=0;
  initial(mutex,1);
  initial(escritura,1);
  cobegin
    for i := 1 to NLEC do Lector[i](i);
    for i := 1 to NESC do Escritor[i](i)
  coend
end.

```

41



## 5. Problemas clásicos de concurrencia

### El problema de los lectores y los escritores. V2

```

(* Prioridad en la escritura *)
program LectEscr;

const
  NLEC = 5;
  NESC = 2;

var
  nl, ne, nle, nee : integer;
  escribiendo: boolean;
  mutex, slector, sescritor: semaphore;

```

42



**E**

```

process type Lectores(id:integer);
begin
  repeat
    acquire(mutex);
    while escribiendo or (nee > 0) do begin
      nle:=nle+1;
      release(mutex);
      acquire(slector);
      nle:=nle-1;
    end;
    nl:=nl+1;
    if nle > 0 then release(slector) {desbloqueo
encadenado}
    else release(mutex);
    {SECCIÓN CRÍTICA}
    acquire(mutex);
    nl:=nl-1;
    if (nl=0) and (nee>0) then release(sescritor)
    else release(mutex);
  forever

```



**E**

```

process type Escritores(id:integer);
var veces: integer;
begin
  repeat
    acquire(mutex);
    if escribiendo or (nl > 0) then begin
      nee:=nee+1;
      release(mutex);
      acquire(sescritor);
      nee:=nee-1;
    end;
    escribiendo:=true;
    release(mutex);
    {SECCIÓN CRÍTICA}
    acquire(mutex);
    escribiendo:=false;
    if nee>0 then release(sescritor)
    else if nle>0 then release(slector)
    else release(mutex)
  forever
end;

```



## 5. Problemas clásicos de concurrencia

### El problema de los lectores y los escritores. V2

```
var
  Escritor: array[1..NESC] of Escritores;
  Lector: array[1..NLEC] of Lectores;
  i: integer;
begin
  nle:=0;  nee:=0;  nl:=0;  ne:=0;
  escribiendo:=false;
  initial(mutex,1);
  initial(sescriptor,0);
  initial(slector,0);
  cobegin
    for i := 1 to NLEC do Lector[i](i);
    for i := 1 to NESC do Escritor[i](i)
  coend
end.
```

45



## 5. Problemas clásicos de concurrencia

### El problema de la comida de los filósofos

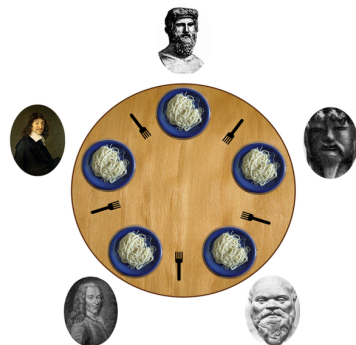
- Es un problema clásico, propuesto por Dijkstra, que ilustra el problema del interbloqueo y la postergación indefinida (*starvation, inanición, hambre, ...*)
  - Cinco filósofos se sientan alrededor de una mesa y pasan su vida comiendo y pensando. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato. Para comer los fideos son necesarios dos tenedores y cada filósofo sólo puede tomar los que están a su izquierda y derecha. Si cualquier filósofo coge un tenedor y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda coger el otro tenedor, para luego empezar a comer.

46



## 5. Problemas clásicos de concurrencia

### El problema de la comida de los filósofos



```
process type filosofo;
begin
  repeat...
    piensa;
    protocolo de entrada;
    come;
    protocolo de salida;
  forever
end;
```

```
var
  Filósofos: array[1..N] of filosofo;
cobegin
  for i := 1 to N do Filósofos[i];
coend
```

47



## 5. Problemas clásicos de concurrencia

### El problema de la comida de los filósofos

```
(* Sin control de interbloqueos *)
program Filósofos;
const
  N = 5;
var
  tenedor : array [1..N] of semaphore;
```

48



## 5. Problemas clásicos de concurrencia

### El problema de la comida de los filósofos

```
process type Filosofo(id : integer);
begin
  repeat
    (* PENSANDO *)
    acquire(tenedor[id]);
    acquire(tenedor[(id mod N) + 1]);
    (* COMIENDO *)
    release(tenedor[id]);
    release(tenedor[(id mod N) + 1]);
  forever
end;
```

49



## 5. Problemas clásicos de concurrencia

### El problema de la comida de los filósofos

```
var
  Filo: array[1..N] of Filosofo;
  i : integer;

begin
  for i := 1 to N do initial(tenedor[i],1);
  cobegin
    for i := 1 to N do Filo[i](i);
  coend
end.
```

50



## 5. Problemas clásicos de concurrencia

### El problema de la comida de los filósofos. V2

```
(* Con control de interbloqueos *)
program Filósofos;
const
  N = 5;
var
  tenedor : array [1..N] of semaphore;
  sillalibre : semaphore;
```

51



## 5. Problemas clásicos de concurrencia

### El problema de la comida de los filósofos. V2

```
process type Filosofo(id : integer);
begin
  repeat
    (* PENSANDO *)
    acquire(sillalibre);
    acquire(tenedor[id]);
    acquire(tenedor[(id mod N) + 1]);
    (* COMIENDO *)
    release(tenedor[id]);
    release(tenedor[(id mod N) + 1]);
    release(sillalibre);
  forever
end;
```

52



## 5. Problemas clásicos de concurrencia

### El problema de la comida de los filósofos. V2

```
var
  Filo: array[1..N] of Filosofo;
  i : integer;

begin
  for i := 1 to N do initial(tenedor[i],1);
  initial(sillalibre,N - 1);
  cobegin
    for i := 1 to N do Filo[i](i);
  coend
end.
```

53



## 5. Problemas clásicos de concurrencia

### El problema de la comida de los filósofos. V3

```
(* Con control de interbloqueos *)
program Filósofos;
const
  N = 5;
var
  libres : array [1..N] of boolean;
  mutex : semaphore;
```

54



## 5. Problemas clásicos de concurrencia

```
process type Filosofo(id : integer);
begin
  repeat
    (* PENSANDO *)
    acquire(mutex);
    while not (libres[id] and libres[(id mod N)+1]) do begin
      release(mutex);
      acquire(mutex);
    end;
    libres[id]:=false;
    libres[(id mod N)+1]:=false;
    release(mutex);
    (* COMIENDO *)
    acquire(mutex);
    libres[id]:=true;
    libres[(id mod N)+1]:=true;
    release(mutex);
  forever
end;
```



## 5. Problemas clásicos de concurrencia

### El problema de la comida de los filósofos. V3

```
var
  Filo: array[1..N] of Filosofo;
  i : integer;

begin
  for i := 1 to N do libres[i]:=true;
  initial(mutex,1);
  cobegin
    for i := 1 to N do Filo[i](i);
  coend
end.
```

56



## 6. Inconvenientes de los semáforos

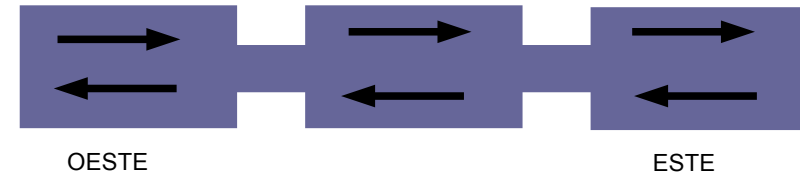
- Es un mecanismo de bajo nivel, no estructurado que fácilmente conduce a errores.
- No es posible restringir el tipo de operaciones realizadas sobre los recursos.
- Es fácil olvidar bloquear todas las instrucciones de la sección crítica.
- Se usan las mismas primitivas tanto para realizar exclusión mutua como condición de sincronización.
- Los programas con semáforos son difíciles de mantener, al estar el código de sincronización disperso entre todos los procesos.

57



## Problemas Propuestos

- Una carretera cruza dos puentes de una sola vía como se muestra en la figura. Programar el comportamiento de los coches del Oeste y del Oeste de forma que la solución no presente interbloqueos. [Palma et al. 2008]



58



## Problemas Propuestos

- Una tribu de  $N$  salvajes cenan en comunidad una gran olla que contiene  $M$  misioneros cocinados. Cuando un salvaje quiere comer, él mismo se sirve de la olla un misionero, a menos que esté vacía. Si la olla está vacía, el salvaje despierta al cocinero y espera a que éste llene la olla. Desarrollar el código de los salvajes y el cocinero. [Palma et al. 2008]

59