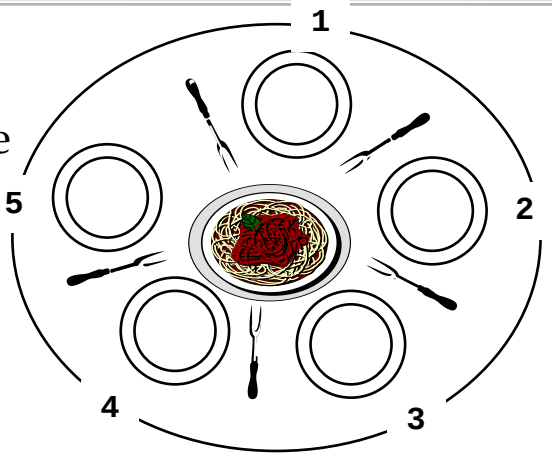
Lección 10: Ejemplos de programación con semáforos

- El problema de la cena de los filósofos
- El problema de los lectores y escritores
- Ejercicios
- Gestión de concurrencia mediante paso de testigo (implementación con semáforos)

El problema de la cena de los filósofos

- El problema:
  - cada filósofo, alternativamente, piensa y come
  - se necesitan dos tenedores para comer
    - el de la izquierda y el de la derecha
  - la cantidad de espaguetis ¡es infinita!
  - hay que evitar que se mueran de hambre
    - por la cabezonería de los filósofos
- Salta a la vista que:
  - dos filósofos vecinos no pueden comer a la vez
  - no más de dos filósofos pueden comer a la vez
- Objetivo: desarrollar un programa concurrente que simule el sistema



Filósofo::
Mq TRUE
coge tenedores
come
deja tenedores
piensa
FMq

El problema de la cena de los filósofos

```
Vars cogido,dejado:vector(1..n) de Ent:=(1..5,0)

Filosofo(i:1..5)::
Mq TRUE
  <cogido(i) := cogido(i)+1>
  <cogido(i⊕1) := cogido(i⊕1)+1>
  come
  <dejado(i) := dejado(i)+1>
  <dejado(i⊕1) := dejado(i⊕1)+1>
  piensa
FMq
```

I: $\forall \alpha \in \{1..5\}$. dejado($\alpha$)<=cogido($\alpha$)<=dejado($\alpha$)+1

El problema de la cena de los filósofos

```
Vars cogido,dejado:vector(1..n) de Ent:=(1..5,0)

Filosofo(i:1..5)::
Mq TRUE
  <await cogido(i)<dejado(i)+1
    cogido(i) := cogido(i)+1
  >
  <await cogido(i⊕1)<dejado(i⊕1)+1
    cogido(i⊕1) := cogido(i⊕1)+1
  >
  come
  <dejado(i) := dejado(i)+1>
  <dejado(i⊕1) := dejado(i⊕1)+1>
  piensa
FMq
```

I: $\forall \alpha \in \{1..5\}.$ $dejado(\alpha) <= cogido(\alpha) <= dejado(\alpha)+1$

El problema de la cena de los filósofos

```
Vars tenedor:vector(1..n)
           de sem:=(1..5,1)

Filosofo(1..5)::
Mq TRUE
  wait(tenedor(i))
  wait(tenedor(i⊕1))
  come
  send(tenedor(i))
  send(tenedor(i⊕1))
  piensa
FMq
```

- **Problema**: se llega a bloquear
- Bloqueo:
  - recursos de uso en exclusión mutua
  - estados "hold-and-wait"
  - "no preemption"
  - ciclo "wait-for"
- Proponer una solución

El problema de los lectores/escritores

- **Problema**:
  - dos tipos de procesos para acceder a una base de datos:
    - lectores: consultan la BBDD
    - escritores: consultan y modifican la BBDD
  - cualquier transacción aislada mantiene la consistencia de la BBDD
  - cuando un escritor accede a la BBDD, es el único proceso que la puede usar
  - varios lectores pueden acceder simultáneamente

El problema de los lectores/escritores

```
Vars leyendo:=0, numLec:=0,
     escribiendo:vector(1..n) de Ent := (1..n,0)
Lector(i:1..m)::
Mq true
  ....
  <numLec:=numLec+1
   Si numLec=1
      leyendo:=leyendo+1
   FSi>
  leeBaseDatos
  <numLec:=numLec-1
   Si numLec=0
      leyendo:=leyendo-1
   FSi>
   ....
FMq
```

I: leyendo+$\sum_{\alpha \in \{1..n\}}$.escribiendo($\alpha$)<=1

```
Escritor(i:1..n)::
Mq true
  ....
  <escribiendo(i):=
     escribiendo(i)+1>
  escribeBaseDatos
  <escribiendo(i):=
     escribiendo(i)-1>
   ....
FMq
```

El problema de los lectores/escritores

```
vars leyendo:=0, numLec:=0,
     escribiendo:vector(1..n) de Ent := (1..n,0)
Lector(i:1..m)::
Mq true
  ....
  <numLec:=numLec+1
   Si numLec=1
     <await SUM<=0
       leyendo:=leyendo+1
     >
   FSi>
  leeBaseDatos
  <numLec:=numLec-1
   Si numLec=0
      leyendo:=leyendo-1
   FSi>
   ....
FMq
```

SUM: leyendo+$\sum_{\alpha \in \{1..n\}}$.escribiendo($\alpha$)
I: SUM<=1

```
Escritor(i:1..n)::
Mq true
  ....
  <await SUM<=0
     escribiendo(i):=
        escribiendo(i)+1>
  escribeBaseDatos
  <escribiendo(i):=
     escribiendo(i)-1>
   ....
FMq
```

```
vars numLec:=0, mutexLec:sem:=1,permiso:sem:=1
Lector(i:1..m)::
Mq true
  ....
  wait(mutexLect)
  numLec:=numLec+1
  Si numLec=1
    wait(permiso)
  FSi
  send(mutexLect)
  leeBaseDatos
  wait(mutexLect)
  numLec:=numLec-1
  Si numLec=0
    send(permiso)
  FSi
  send(mutexLect)
   ....
FMq
```

```
Escritor(i:1..n)::
Mq true
  ....
  wait(permiso)
  escribeBaseDatos
  send(permiso)
   ....
FMq
```

El problema de los lectores/escritores. Prioridad

```
Lector(i:1..n)::
Mq true
  ....
  Si escritor escribiendo o
        esperando
    Ent esperarParaLeer
  Fsi
  avisar "sig." lector espera
  leeBaseDatos
  Si último y escritor espera
    Ent avisarle
  Fsi
  ....
FMq
```

```
Escritor(i:1..m)::
Mq true
  ....
  Si alguien escribiendo o
        leyendo
    Ent esperarParaEsc
  Fsi
  escribeBaseDatos
  Si escritor esperando
    Ent avisarle
    SiNo Si lector esperando
           Ent avisarle
         FSi
  Fsi
  ....
FMq
```

```
vars numLec: Ent:=0, mutex:sem:=1; puedesLeer,
    puedesEscribir:sem:=0;numLE,numEE: Ent:=0
Lector(i:1..m)::
Mq true
  ....
  wait(mutex)
  Si escribiendo ∨ numEE>0
    numLE:=numLE+1
    send(mutex)
    wait(puedesLeer)
    numLE:=numLE-1
  FSi
  numLec:=numLec+1
  Si numLE>0
     send(puedesLeer)
  Si_No
     send(mutex)
  Fsi

FMq
```

Prioridad a escritores

```
....
  leeBaseDeDatos
  wait(mutex)
  numLec:=numLec-1
  Si numLec=0 ∧ numEE>0
     send(puedesEscribir)
  Si_No
     send(mutex)
  FSi
....
```

escritores

Prioridad a escritores

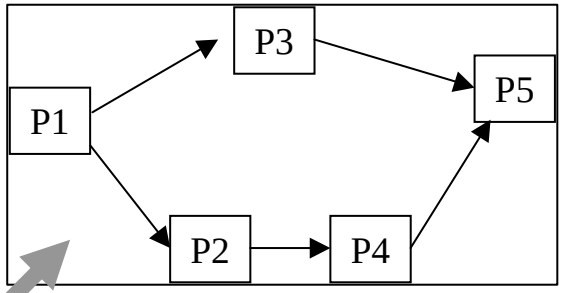
```
...
Escritor(i:1..n)::
Mq true
   ....
   wait(mutex)
   Si numLec>0 ∨ escribiendo
      numEE:=numEE+1
      send(mutex)
      wait(puedesEscribir)
      numEE:=numEE-1
   FSi
   escribiendo:=TRUE
   send(mutex)
   escribirBBDD

FMq
```

```
....
   wait(mutex)
   Si numEE>0 Ent
      send(puedesEscribir)
   Si_No Si nLE>0 Ent
            send(puedesLeer)
         Si_No
            send(mutex)
         FSi
   FSi
```

Ejercicios

- **Ejercicio 1:** Un grafo de precedencias de tareas es un grafo dirigido acíclico que establece un orden de ejecución de tareas de un programa concurrente, de manera que una tarea no puede empezar a ejecutarse hasta que las que sean anteriores en el grafo hayan terminado. Por ejemplo, si consideramos el grafo de la figura 1, la tarea P4 no puede empezar a ejecutarse hasta que P2 y P1 terminen, mientras que P5 no puede empezar hasta que P1, P2, P3 y P4 hayan terminado. Asumamos que cada tarea ejecuta el siguiente código:

```
espera a que las tareas anteriores terminen
ejecuta el cuerpo de la tarea
avisa de su terminación a quien corresponda
```



1) Usando semáforos, escribir el programa concurrente correspondiente al diagrama de la figura

2) Esquematizar un método general de sincronización para un grafo de precedencias general. Calcular el número de semáforos que el método utilizaría. Este número se puede poner, por ejemplo, como función del número de tareas, de arcos, etc.

Ejercicios

- **Ejercicio 2**: Considerar el siguiente programa concurrente:

  Calcular para él el conjunto de los posibles valores finales para la variable x

```
Vars x: Ent := 0;
     s1: sem := 1;
     s2: sem := 0
P1::          |  P2::          |  P3::
              |                |
wait(s2)      |  wait(s1)      |  wait(s1)
wait(s1)      |  x := x*x      |  x := x+3
x := 2*x      |  send(s1)      |  send(s2)
send(s1)      |                |  send(s1)
```

- **Ejercicio 3:** Implementar un semáforo general en base a uno (o varios) semáforo binarios

Ejercicios

- **Ejercicio 4**: Problema de uso de recursos: **n** procesos compiten por el uso de un recurso, del que se disponen de k unidades. Las peticiones de uso del recurso son del tipo:

  `reserva(r)`: --necesito se me concedan, "de golpe", r unidades del recurso

  `libera(l)`: --libero, "de golpe", l unidades del recurso, que previamente se me habían concedido

- Programar dichas operaciones usando semáforos binarios

Ejercicios

- **Ejercicio 5**: Considerar el siguiente programa, que presenta un problema: nada impide que se use una variable en una expresión con un valor todavía indefinido.
- Se pide lo siguiente. Utilizando semáforos, completar el código de dicho programa de manera que nunca una variable sea usada en una expresión antes de que se le haya asignado un valor, ya sea mediante una instrucción `leer` o una asignación.
- Explicar cómo se generalizaría dicha solución para cualquier programa. ¿Podría un programa escrito de acuerdo a la propuesta anterior llegar a bloquearse? Si es así, escribir un ejemplo sencillo de bloqueo. Si no es posible, justificar "de manera convincente" por qué es así.

```
Vars a,b,c,d: Ent;
P1::
   leer(a);
   c := a+b
   b := 2*d
P2::
   leer(c)
   leer(d)
   b := a+d+c
```

El paso del testigo

- Veamos una forma (eficiente) de implementar exclusiones mutuas y sincronización por condición con semáforos binarios
- Necesitamos:

E_1: <S_i> E_2:<await B_j S_j>

- Un semáforo binario para asegurar la ejecución en exclusión mutua: ***mutex*** (valor inicial 1)
- Para cada j, un semáforo binario b_j y un contador d_j, a cero

El paso del testigo

- Cambiar

```
E1: wait(mutex)
    Si
    AVISAR
```

```
E2: wait(mutex)
    Si no Bj
      Ent dj:=dj+1
          send(mutex)
          wait(bj)
    FSi
    Sj
    AVISAR
```

- donde

```
AVISAR:
--mutex=0
  Sel
    B1 y d1>0: d1:=d1-1;send(b1)
    ....
    Bn y dn>0: dn:=dn-1;send(bn)
    otros casos: send(mutex)
  FSel
```