Programación concurrente

Master de Computación

I Conceptos y recursos para la programación concurrente:

I.3 Problemas específicos de la programación concurrente.

J.M. Drake

M. Aldea

Problemas específicos de la programación concurrente

- Modelos de interacción entre procesos.
- Problemas de sincronización y exclusión mutua.
  - Actualizaciones concurrentes de variables compartidas
  - Sincronización en la ejecución de tareas
- Solución mediante técnicas de programación secuencial

Tipos básicos de interacción entre procesos

Los procesos de una aplicación concurrente pueden interaccionar entre sí de acuerdo con los siguientes esquemas:

- **Independientes entre sí**: Interfieren por compartir el procesador.
- **Cooperan entren sí**: Uno genera una información o realiza algún servicio que el segundo necesita.
- **Compiten entre sí**: Requieren usar recursos comunes en régimen exclusivo.

Interacciones entre procesos en programación OO.

Dentro de la metodología orientada a objetos, una aplicación resulta de componer tres tipos de objetos:

- **Objetos activos:** Tienen capacidad de realizar actividades autónomas propias. Son objetos con thread propio.

- **Objetos neutros:** Son objetos que prestan servicios a los objetos activos. Pueden ser usados simultáneamente por los objetivos activos sin representar interacción entre ellos. Son librerías pasivas.

- **Objetos pasivos:** Son objetos que prestan servicios a los objetos activos. Cuando varios objetos activos tratan de hacer uso de él simultáneamente arbitran el acceso de acuerdo con una estrategia propia. Representan recursos de interacción entre procesos.

Implementación de objetos pasivos.

- Los objetos pasivos se construyen en base a procesos internos que implementan su control:
  - El thread de los mismos arbitra el acceso de los objetos activos.
  - No requieren introducir nuevos componentes de programación.
  - Son poco eficientes ya que requieren múltiples cambios de contexto.

- Los objetos pasivos se construyen a partir de componentes de sincronización pasivos:
  - Requieren la definición de nuevas primitivas de sincronización.
  - Introducen mayor complejidad ya que utilizan componentes de muy diferentes niveles de abstracción.
  - Son muy eficientes.

Problemas específicos de programación concurrentes.

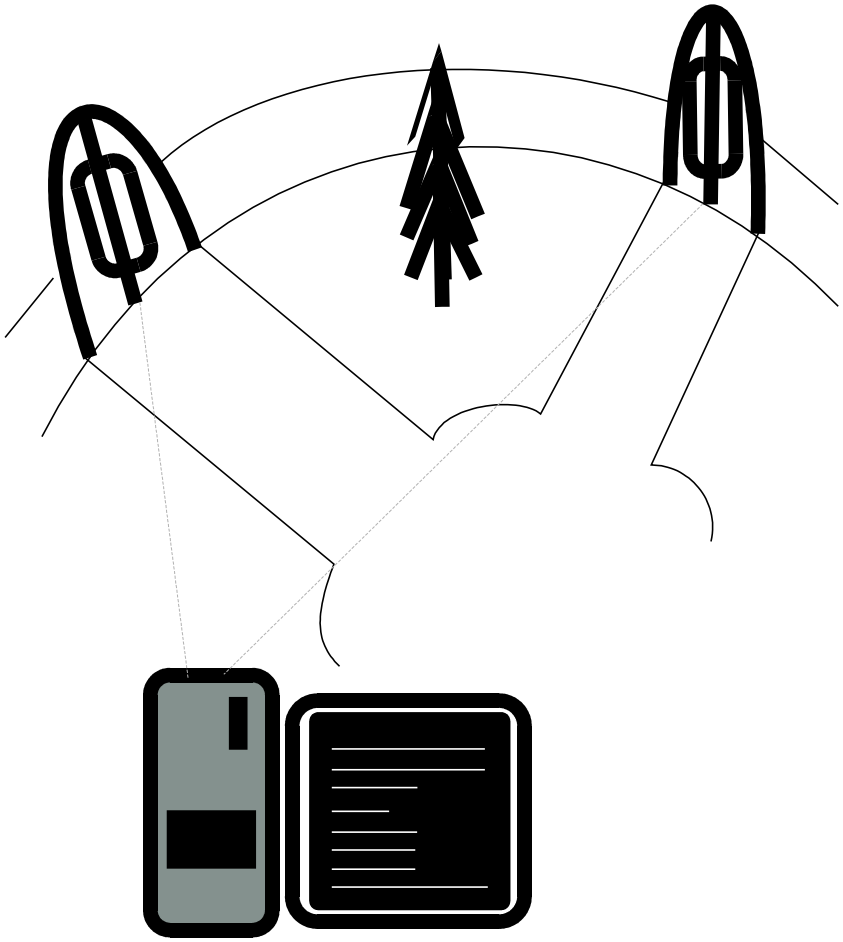
- **Actualizaciones concurrentes de variables compartidas**:

  "Cuando un proceso modifica una misma variable compartida que no es atómica, mientras que otro proceso concurrente la lee o escribe, el resultado que se obtiene no es seguro."

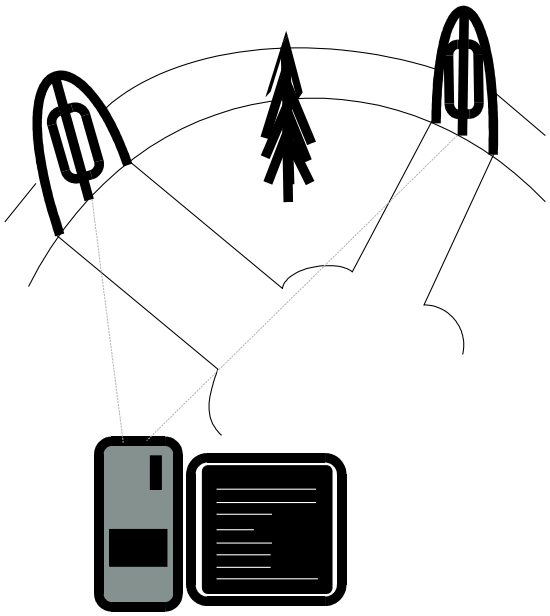
- **Sincronización en la ejecución de tareas**:

  "La lógica de la aplicación requiere que un proceso no pueda ejecutar una sentencia determinada hasta que otro proceso haya ejecutado una sentencia de la que depende."

Parque público: Actualización concurrente de variable



Código de la aplicación Parque Público.



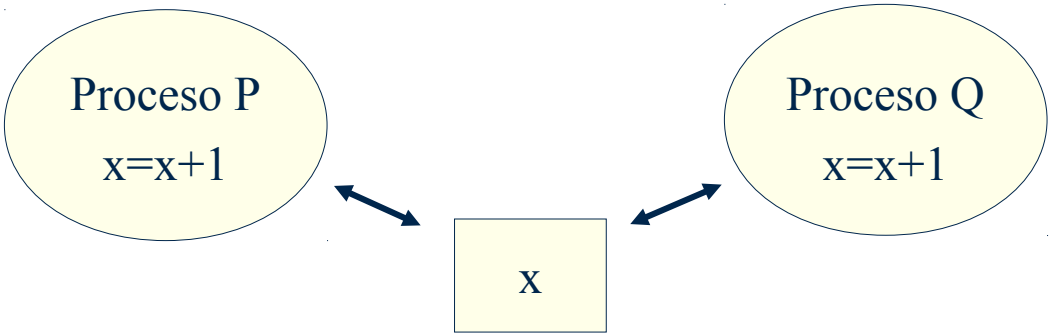
```
program Control_parque;
  var Cuenta : integer;

 process Torno_1;
    var n: Integer;
 begin
    for n:=1 to 20 do Cuenta:=Cuenta +1;
 end;

 process Torno_2;
    var n: Integer;
 begin
    for n:=1 to 20 do Cuenta:=Cuenta +1;
 end;

begin (* Cuerpo del programa principal Control_parque *)
        Cuenta:=0;
        cobegin  Torno_1; Torno_2; coend;
        writeln ("Total de visitantes: ", Cuenta);
end;
```

Actualización concurrente de una variable compartida.



(X1) Carga x en acumulador
(X2) Incrementa acumulador
(X3) Almacena acumulador en x

Entrelazado de operaciones concurrentes que conduce a error.

Valor inicial de x	x = 4	
(P1) P carga x en su acumulador	AcP = 4	x = 4
(Q1)Q carga x en su acumulador	AcQ = 4	x = 4
(Q2)Q incrementa su acumulado	AcQ = 5	x = 4
(P2) P incrementa su acumulador	AcP = 5	x = 4
(Q3)Q almacena el acumulador en x	AcQ=5	x = 5
(P3) P almacena el acumulador en x	AcP = 5	x = 5

El resultado es incorrecto ya que el resultado debería ser 6 y no 5.

Ejemplo de código de programa productor consumidor.

```
program Productor_consumidor;
var Buffer: integer;

process Productor;
  var Dato_entregado: Integer;
begin
  .....
  Buffer:= Dato_Entregado;
  .....
end;

process Consumidor;
  var Dato_recibido: Integer;
begin
  ....
  Dato_recibido:= Buffer;
  ....
end.

begin
    cobegin
        Productor;
        Consumidor;
     coend;
end.
```

Necesidad de primitivas para programas concurrentes.

- ¿Es posible superar los problemas sincronización y exclusión mutua de la programación concurrente utilizando lenguajes de programación secuenciales?

  La respuesta es sí, pero sólo de forma poco eficiente.

- Por ello todos los lenguajes de programación concurrentes introducen primitivas específicas que permiten resolver los problemas de forma mas eficiente.

Solución del problema de sincronización (mal).

```
program Productor_consumidor_correcto;
var    Buffer : Integer;
       Dato_Creado: Boolean;

process Productor;
  var Dato_producido: Integer;
begin
    ....
    Buffer:=Dato_producido;
    Dato_Creado:= True
    .....
end;
begin
  Dato_Creado:= False;
  cobegin Productor; Consumidor; coend;
end;
```

```
process Consumidor;
   var Dato_recibido: Integer;
begin
   ....
   while not Dato_Creado  do  null;
   Dato_recibido:= Buffer;
   ....
end;
```

Solución del problema de sincronización.

```
program Productor_consumidor_correcto;
var    Buffer : Integer;
       Dato_Creado: Boolean;

process Productor;
  var Dato_producido: Integer;
begin
   ....
   Buffer:=Dato_producido;
   Dato_Creado:= True
   .....
end;
begin
   Dato_Creado:= False;
   cobegin Productor; Consumidor; coend;
end;
```

```
process Consumidor;
  var Dato_recibido: Integer;
begin
   ....
   while not Dato_Creado  do sleep(0);
   Dato_recibido:= Buffer;
   ....
end;
```

Solución de la exclusión mútua

- La **solución de Peterson** (1981), presupone:
  - Lecturas concurrentes de una variable son siempre correctas.
  - Si dos procesos P y Q escriben una variable, una de las dos escrituras es correcta (la última que se hizo).
  - Si un proceso escribe y otro lee concurrentemente una variable, se lee el valor escrito o el valor previo a la escritura, pero no una mezcla de ambos.
- Estas suposiciones son razonables para una variable **atómica** en un sistema monoprocesador.
- Esqueleto

```
repeat
   Protocolo_de_entrada;
      Sección_crítica;
   Protocolo_de_salida
   Sección_no_crítica;
forever;
```

Primer intento (No válido)

....

var

Flag1 : Boolean; *(* P1 anuncia su intención de entrar en S.C. *)*

Flag2 : Boolean; *(* P2 anuncia su intención de entrar en S.C. *)*


```
process P1;
begin
  repeat
     Flag1:= True;
     while Flag2 do sleep(0);
     (Sección crítica)
     Flag1:= False;
     (Sección no crítica)
   forever;
end;
```

```
process P2;
begin
  repeat
     Flag2:= True;
     while Flag1 do sleep(0);
      (Sección crítica)
      Flag2:= False;
      (Sección no crítica)
  forever;
end;
```

Segundo intento (No válido)

```
var
Flag1: Boolean:=False;   (* P1 anuncia su entrada en S.C.*)
Flag2: Boolean:=False;   (* P2 anuncia su entrada en S.C.*)
```

```
process P1;
begin
  repeat
     while Flag2 do sleep(0);
     Flag1:= True;
     (Sección crítica)
     Flag1:=False;
     (Sección no crítica)
   forever;
end;
```

```
process P2;
begin
    repeat
     while Flag1 do sleep(0);
     Flag2:= True;
      (Sección crítica)
      Flag2:= False;
     (Sección no crítica)
   forever;
end;
```

Tercer intento (No válido)

```
.....
var Proc: 1..2;        (* Procesador que ha accedido a S.C.*)

process P1;                          process P2;
begin                                begin
  repeat                               repeat
    while Proc=2 do sleep(0);            while Proc=1 do sleep(0);
     (Sección crítica)                    (Sección crítica)
     Proc:=2;                             Proc:=1;
     (Sección no crítica)                 (Sección no crítica)
  forever;                             forever;
end;                                 end;
```

- NO válido: exige accesos alternativos

Solución de Peterson.

```
Var
    Flag1: Boolean:= False; (*P1 anuncia su intención de entrar en SC*)
    Flag2: Boolean:= False; (*P2 anuncia su intención de entrar en SC*)
    Proc:1..2;   (*Procesador que tiene preferencia para acceder a S.C.*)
```

```
process P1;
begin
   repeat
      Flag1 := True;
      Proc :=2 ;
      while Flag2 and (Proc = 2)
         do sleep(0);
      (Sección crítica)
      Flag1:= false;
      (Sección no crítica )
    forever;
end;
```

```
process P2
begin
   repeat
      Flag2 := True;
      Proc :=1 ;
      while Flag1  and (Proc = 1)
         do sleep(0);
      (Sección crítica )
      Flag2:= false;
      (Sección no crítica)
    forever;
end;
```