PROGRAMACION CONCURRENTE

I.3. Interacción entre Procesos Concurrentes.

J.M. Drake

Notas:

Interacción entre Procesos Concurrentes

- Modelos de interacción entre procesos.
- Problemas de sincronización y exclusión mutua.

Notas:

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.

Notas:

Las relaciones básicas que se pueden presentar entre dos procesos concurrentes de un mismo programa, son:

- Son independientes entre sí.
- Son procesos cooperantes.
- Son procesos competidores.

En el primer caso, entre los procesos no se requiere que exista ningún tipo de interrelación. Si embargo, en el segundo y en el tercer caso, se necesita disponer de componentes que hagan posible establecer acciones de sincronización y de comunicación entre los procesos.

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 procesos 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.

Notas:

Dentro del paradigma de **programación orientada al objeto**, cualquier aplicación resulta como composición de tres tipos de objetos:

•**Objetos activos**: son aquellos entes que requieren tener capacidad de llevar a cabo acciones espontáneas autónomas. En programación concurrente estos objetos corresponden siempre a componentes tipo proceso.

•**Objetos neutros**: son aquellos objetos no activos que prestan servicios a los objetos activos que lo requieren, sin necesidad de introducir ningún tipo de actividad especifica por su parte. En programación concurrente, estos objetos se construyen mediante estructuras de datos pasivas, y cuando convenga se las dota con un interfaz de procedimientos que permita abstraer el acceso a ellos.

•**Objetos pasivos**: son aquellos objetos no activos que solo prestan servicios a los objetos activos que lo requieren, pero introduciendo por su parte una actividad de control o arbitraje efectivo, entre los sucesivos o simultáneos servicios que le son requeridos. Estos objetos, que se pueden identificar con el concepto de servidor, se introducen los aspectos más significativos de la programación concurrente

Implementación de objetos pasivos.

- Los objetos pasivos se construyen en base a procesos internos que implementan su control:
  - El thread de los mismos arbitran 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.
  - Introduce mayor complejidad ya que utiliza componentes de muy diferentes niveles de abstracción.
  - Son muy eficientes.

Notas:

Para construir los objetos pasivos se utiliza una de las dos siguientes estrategias:

•Los objetos pasivos se implementan como **procesos**, que incluyen los recursos del objeto, y que con la actividad que le es propia, controlan y arbitran el acceso a los mismos. Es la solución más simple ya que no implica introducir nuevos componentes de programación, sin embargo presenta el inconveniente de que incrementa el número de procesos del programa, así como los cambios de contexto.

•Introducir **nuevos componentes primitivos** (tales como semáforos, señales, zonas de exclusión mutua, etc.) que permitan representar explícitamente y directamente los objetos pasivos. Con ellos se consigue que la programación concurrente sea mucho más eficiente, aunque más compleja y proclive a errores.

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."

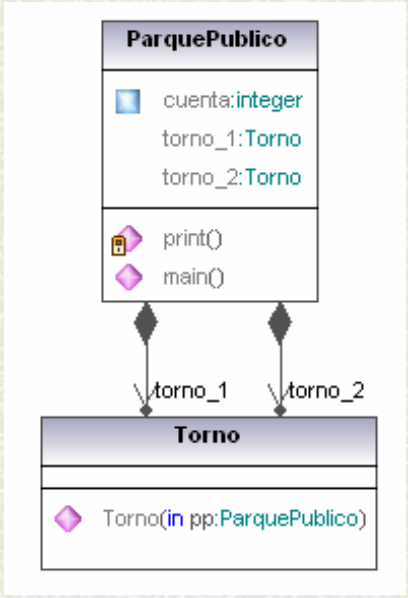
Notas:

En programación concurrente se presentan problemas específicos que pueden conducir a errores que no existen en programación secuencial. Los dos problemas básicos propios de programación concurrente son:

•La actualización concurrente de variables compartidas. Si dos procesos concurrentes tratan de actualizar una variable compartida, y para ello cada uno de ellos debe realizar múltiples operaciones de lectura y escritura de los campos de la variable la operación no es segura. Puede producirse que la operación concurrente termine corrompiendo la variable que actualizan.

•Cuando dos procesos concurrentes necesitan colabora de forma sincrona, se necesita poder evitar que un proceso no ejecute una cierta actividad hasta que el otro no haya alcanzado el punto adecuado para que la colaboración sea eficaz.

Parque público: Actualización concurrente de variable



Notas:

Ejemplo "Control de parque público": Considérese el caso de un parque público que dispone de varias puertas de acceso. El acceso por cada una de las puertas del parque, está controlada por un torno independiente, que envía un evento propio a una aplicación de computador que debe contarlas y proporcionar en cualquier instante el número total de visitantes que han entrado en el parque.

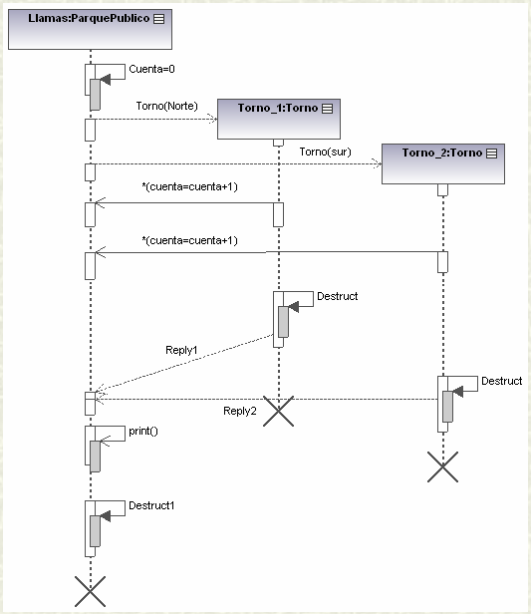
La solución concurrente inicial que se propone, se basa en las siguientes ideas:

- Los eventos que genera cada torno van a ser gestionados por un proceso independiente.
- Los procesos de control de los tornos se ejecutan de forma concurrente.
- En el programa existe una variable global entera "cuenta" que representa el número el numero de visitantes que ha entrado en el parque.

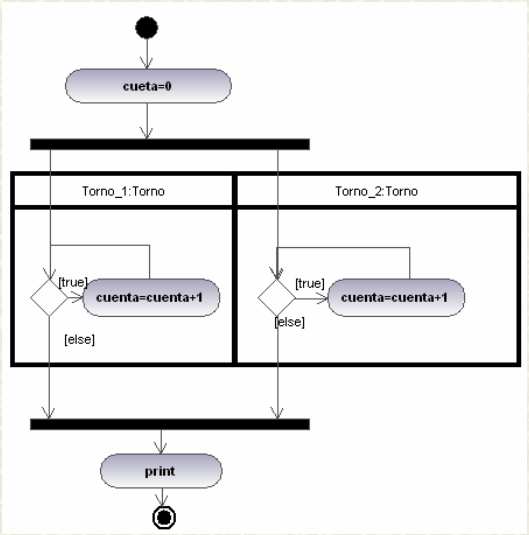
  Cuando la actividad de los tornos ha concluido proporciona la información sobre el número de visitantes que se han producidos.

Descripción de la aplicación parque público

Interacción entre procesos



Lógica del procedimiento main()



Notas:

El diagrama de secuencias que muestra las interacciones entre los tres objetos que constituyen la aplicación, y el diagrama de actividad que representa las actividades que genera la ejecución del procedimiento main() que lanza la apliacación contienen una misma información:

•Inicialmente la variable cuanta se inicializa a cero.

•Se crean y lanzan los dos procesos que gestionan cada uno de los totos.

•Por cada visitante del parque que llega por un torno el proceso que lo gestiona ejecuta la sentencia cuenta=cuenta+1 sobre una misma variable global.

•Cuando los procesos que gestionan los tornos terminan (por temporización: ha llegado la ora de cerrar; o por cuenta: el parque está lleno) se imprime el valor de la variable cuenta.

El ejemplo presenta problemas porque dos procesos concurrentes actualizan de forma no segura la misma variable global cuenta.

Código de la aplicación Parque Público.

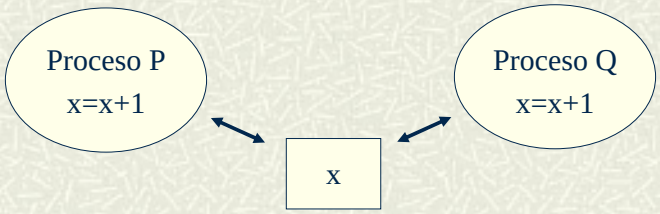
```
program Control_parque;
var Cuenta : integer;

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;
```


Notas:

A continuación se muestra un programa simple que emula el funcionamiento de este programa. Se consideran dos tornos, cada uno de ellos representado mediante un proceso independiente que recibe un número predeterminado (20) de visitantes y por cada uno de ellos incrementa la variable global. Una sentencia writeln final emula la gestión del monitor.

Este programa da soluciones erroneas. No siempre escribe 40 como debiera.

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.

Notas:

Problema de las actualizaciones concurrentes: consideremos el caso de dos procesos P y Q que ejecutan de forma independiente, pero concurrentemente la sentencia x := x +1;

•Esta sentencia no es atómica. Para su realización, la CPU debe llevar a cabo las siguiente secuencia de instrucciones básicas: (1) Carga el valor de x en un acumulador de la CPU, (2) Incrementa el valor del acumulador y (3) Almacena el valor del acumulador en la variable x.

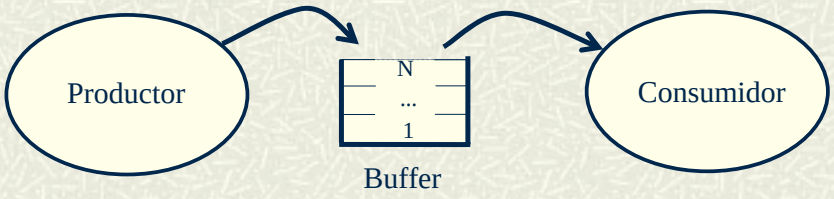
•Cuando la sentencias correspondientes a los procesos P y Q se ejecutan concurrentemente, las instrucciones básicas pueden resultar entrelazadas en cualquier combinación.

•Es fácil en este caso encontrar secuencias validas de entrelazado que no conducen a un incremento en 2 el valor de la variable x, como debe ser en un funcionamiento correcto.

	Inicialmente:	**x = 4**	
(P1)	P carga el valor de x en su acumulador P_{AC}.		P_{AC} = 4
(Q1)	Q carga el valor de x en su acumulador Q_{AC}.		Q_{AB}= 4
(Q2)	Q Incrementa el valor de su acumulador Q_{AC}.	Q_{AB} = 5	
(P2)	P Incrementa el valor de su acumulador P_{AC}.	P_{AB} = 5	
(Q3)	Q almacena el valor de su acumulador Q_{AC} en x.	x = 5	
(P3)	P almacena el valor de su acumulador Q_{AC} en x.	x = 5	
	Resulta:	**x = 5**	

•El valor que resulta para la variable x es 5, y no 6 como debería ser si la funcionalidad hubiese sido correcta.

Sincronización : Problema productor-consumidor.



Un productor y un consumidor de un dato, lo intercambian a través de un buffer de capacidad limitada a N elementos. En este caso se requieren las dos siguientes condiciones de sincronización:

$\forall$ e • (producción^{i} $\rightarrow$ consumisión^{i})

$\forall$ e • (consumisión^{i} $\rightarrow$ producción^{i+N})

Notas:

En este caso, el problema difiere del problema de "actualización concurrente" en que solo uno de los procesos modifica la variable global, mientras que el otro se reduce a leerlo. Sin embargo, para que este proceso opere correctamente es necesario que entre productor y consumidor exista una sincronización para que el consumidor no lea el dato de buffer, antes de que el consumidor lo haya depositado.

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.
```


Notas:

Problema productor consumidor

Ejemplo "Productor consumidor con buffer sencillo": Considérese el caso de dos procesos que denominamos Productor y Consumidor. El primero proceso genera un dato, y se lo transfiere al segundo que debe recibirlo y consumirlo. Se desea que ambos proceso se puedan llevar a cabo de forma concurrente, para lo cual se dispone de un buffer, donde el productor lo deja cuando lo genera, y de donde el consumidor lo recoge cuando esta dispuesto a aceptar un nuevo dato. Considérese que inicialmente el buffer tenga cabida para solo un dato.

Este problema no opera razonablemente ya que el proceso Consumidor lee el dato antes de que Productor lo haya generado.

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 si, pero solo de forma poco eficiente.

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

Notas:

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;

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

begin
Dato_Creado:= False;
cobegin Productor; Consumidor; coend;
end;
```


Notas:

Solución del problema de sincronización:

La necesidad de sincronización del problema productor-consumidor (condición (a)) se puede resolver introduciendo una segunda variable Dato_creado que retrasa al consumidor hasta que el dato está producido.

En la solución que se propone,el consumidor permanece en un bucle hasta que la variable Dato_creado, le avisa de que el dato esta disponible.

Esta solución es muy poco eficiente, ya que implica que la espera por parte del consumidor consume procesador porque el proceso esta plenamente activo, es muy poco eficiente. La sentencia sleep(0) permite que una política imprevista de planificación dé siempre procesador al consumidor, y en consecuencia el productor no generaría el dato y el programa permanecería permanentemente bloqueado.

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;

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

begin
Dato_Creado:= False;
cobegin Productor; Consumidor; coend;
end;
```


Notas:

Solución del problema de sincronización:

La necesidad de sincronización del problema productor-consumidor (condición (a)) se puede resolver introduciendo una segunda variable Dato_creado que retrasa al consumidor hasta que el dato está producido.

En la solución que se propone,el consumidor permanece en un bucle hasta que la variable Dato_creado, le avisa de que el dato esta disponible.

Esta solución es muy poco eficiente, ya que implica que la espera por parte del consumidor consume procesador porque el proceso esta plenamente activo, es muy poco eficiente. La sentencia sleep(0) permite que una política imprevista de planificación dé siempre procesador al consumidor, y en consecuencia el productor no generaría el dato y el programa permanecería permanentemente bloqueado.

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;
```


Notas:

Solución de la exclusión mutua: Soportar mediante variables ordinarias la exclusión mutua es más complicado. Se expone la solución de Peterson (1981):

•Las soluciones que se plantean se basan en las siguientes suposiciones:

•Operaciones concurrentes de lectura de una variable son siempre correctas.

•Si dos procesos P y Q realizan concurrentemente una operación de escritura sobre una variable, el resultado es la escritura correcta, bien la correspondiente a P o la correspondiente a Q, pero no una mezcla incorrecta de ambas.

•Si un proceso escribe una variable y otro concurrentemente lee la misma variable, el valor leído, es o bien el antiguo, o bien el recién escrito, pero no una mezcla incorrecta de ambos.

•El planteamiento de algoritmo que se busca, es desarrollar dos procesos P1 y P2, que se ejecuten concurrentemente, cada uno de los cuales tenga una zona crítica, en las que bajo ninguna situación pueden estar simultáneamente. El esqueleto de los procesos que se utilizarán es:

```
process P;
begin
   repeat
      Protocolo_de_entrada;  Sección_critica;  Protocolo_de_salida
     Sección_no_crítica;
    forever;
end;
```

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

Fallo bloqueo

Notas:

Se introducen dos variables booleanas Flag1 y Flag2 que están inicializada al valor false. Flag1 solo es modificado por P1 y representa el anuncio por parte de P1 de su intención de acceder a la zona crítica. Flag2 guarda la misma relación respecto de P2.

Este algoritmo no es válido porque permite que ambos proceso queden en un bloqueo cruzado indefinido. La secuencia que conduce al bloqueo es:

P1 establece Flag1:= true
P2 establece Flag2 := true
P2 chequea el valor de Flag1 y permanece haciendo la sentencia null
P1 chequea el valor de Flag2 y permanece haciendo la sentencia null

esta situación se mantiene indefinidamente bloqueada.

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

Fallo seguridad

Notas:

Similar al intento 1, salvo que ahora se invierte el orden de las acciones de anunciar el acceso a la sección crítica y comprobar el estado del otro proceso.

Este algoritmo falla porque posibilita el acceso de ambos procesos a sus respectivas secciones críticas, si se produce la siguiente secuencia:

P1 y P2 están en su sección no crítica (Flag1 =Flag2 = false)
P1 chequea Flag2 (que aún false)
P2 chequea Flag1 (que aún es false)
P2 pone Flag2 a true
P2 entra en su SC
P1 pone Flag1 a true
P1 entra en su sección crítica

el algoritmo ha fallado porque ambos están simultáneamente en su sección crítica.

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;
```


Notas:

El problema de los intentos anteriores es que no se puede realizar de forma atómica el chequeo de la situación del otro y la confirmación de acceso propia. Por ello el tercer intento se basa en utilizar una sola variable que sea la que establezca cual es el proceso que puede entrar en la zona crítica. Esta variable la denominamos Proc, y puede tomar los valores 1.. 2.

Esta solución no es admisible porque exige que el acceso a la sección sea alternativa por parte de los dos procesos.

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;                    process P2
begin                          begin
  repeat                         repeat
     Flag1 := True;                 Flag2 := True;
     Proc :=2 ;                     Proc :=1 ;
     while Flag2  and (Proc = 2)    while Flag1  and (Proc = 1)
                    do sleep(0);                   do sleep(0);
        (Sección crítica)              (Sección crítica )
     Flag1:= false;                 Flag2:= false;
     (Sección no crítica )          (Sección no crítica)
   forever;                       forever;
end;                           end;
```


Notas:

olución de Peterson

Se basa en utilizar dos variables booleanas Flag1 y Flag2 que anuncia la intención de entrar en su zona crítica, y una variable Proc que puede tomar los valores 1..2 y que indica el procesador al que se da prioridad de acceso, y que solo es útil cuando hay conflicto en el acceso a la zona crítica.

Si solo un proceso desea entrar en la sección crítica, el Flag contrario será false y el acceso es inmediato con independencia del valor de Proc. Si ambos tratan de acceder y ambos Flag son true, solo entra la que indique Proc.

Algoritmo bakery: Programa principal

```
program ControlParquePublico  (* Exclusión mutua utilizando el algoritmo bakery *)
  const nTornos=50;
  var  ticket: array[1..nTornos] of integer;
       cogiendoTicket: array[1.nTornos] of boolean;
       cuenta: integer;
       tornoID: integer;
   process type TipoTorno(id: integer) ..... end; (* Se define en la siguiente transparencia....*)
    var torno: array[1..nTornos] of TipoTorno;
begin
  cuenta:=0; (* se inicializa cuenta a 0 *)
  for tornoID=1 to nTornos do begin   (* Se inicializan las variables asociadas a cada torno*)
    ticket[tornoID]:=0;
    choosing[tornoID]:=false;
  end;
  cobegin  (* Se ejecutan concurrentemente los procesos torno*)
     for tornoID:= 1 to nTornos do torno[tornoID](tornoID);
  coend;
  writeln("Total de clientes admitidos: ", cuenta);
end;
```


Notas:

Para el caso general de exclusión mutua, en el que hay que sincronizar a un número ilimitado de cliente, se han propuesto algunos algoritmos (Dijkstra 1968, Eisenberg and McGuire, 1972, etc.) En este ejemplo se muestra el algoritmo de bakery propuesto por Lamport, 1974.

Un número arbitrario de tornos actualizan concurrentemente una variable cuenta, incrementándola en 1.

El algoritmo se basa en que antes de actualizar la variable, el proceso cliente toma un ticket con un entero que es único. El proceso no actualiza la variable si existe algún otro cliente con un número en el ticket inferior al suyo. Como la búsqueda del ticket puede fallar porque no es seguro, en el caso de que tengan igual número se decide por el valor del indentificador del proceso. Esto supone que el algoritmo no es imparcial, sino que favorece al que tiene identificadores mas bajos.

Significado de las variables definidas en el programa principal:

•nTornos: integer=50 Número de tornos que tiene el parque.

•ticket: Array que contiene el ticket que es establecido por cada proceso torno antes de realizar la actualización concurrente

•cogiendo: array de booleanos que se utilizan para proteger cada proceso mientras que coge el valor del tickets

•cuenta: Acumula el número de visitantes que ha tenido el parque.

•torno: Array de procesos tipo TipoTorno.

Algoritmo bakery: Proceso TipoTorno (1)

```
process type TipoTorno (id: integer);
  var otroTorno:integer;
      tornoID: integer;
      cliente: integer;
  function max: integer;
   (* Retorna el número del ticket *)
    var i, mayor:integer;
   begin
    mayor:=0;
    for i:=1 to nTornos do
      if ticket[i]>mayor
      then mayor:= ticket[i];
    max:=mayor+1;
   end;

begin
  ....
end;
```

```
function elegido(mio,otro:integer): boolean;
 (* Compara el ticket de los tornos otro y yo *)
begin
 if (ticket[mio]=0) or (ticket[mio]>ticket[otro])
 then elegido:=false;
 else
      if (ticket[mio]<ticket[otro])
      then elegido:=true;
      else elegido=(mio<otro);
end;
```


Notas:

Describe los procesos que se instancian por cada torno:

•function max(): integer => Retorna el entero que corresponde al ticket, buscando el entero mas alto asignado a los ticket actualmente asignados.

•function elegido(otro,yo:integer): boolean => Retorna True si el torno de identificador yo tiene el ticket mas bajo que el identificado con otro.

Algoritmo bakery: Proceso TipoTorno (2)

```
...
begin
  for cliente :=1 to 20 do begin   (* Simula la entrada de 20 visitantes *)
     eligiendoTicket[id]:=true;     (* Requiere un ticket *)
     ticket[id]:=max;
     eligiendoTicket[id]:=false;
     for otroTorno:=1 to nTornos do begin  (* Espera a que ningún otro torno esté accediendo *)
        while eligiendoTicket[otroTorno] do null; (* Ninguno está cogiendo ticket *)
        while elegido(otroTorno,id) do null;       (* Ningún otro tiene un ticket mas preferente *)
     end;
     cuenta:=cuenta+1;          (* Incrementa la variable cuenta *)
     ticket[id]:=0;             (* Anula el ticket *)
   end; (* for cliente *)
end; (* TipoTorno *)
```


Notas:

La actividad que ejecuta cada torno es:

- El loop simula la entrada de 20 visitante por el bucle.
- Para cada cliente:
  - Elige un ticket
  - Espera hasta que ningún otro torno esté cogiendo un ticket
  - Espera hasta que ningún otro torno tenga un ticket mas preferente que el propio
  - Incrementa la cuenta
  - Anula el ticket