3. SISTEMAS CON MEMORIA COMPARTIDA

3.1. EXCLUSIÓN MUTUA

Consiste en asegurar que dos procesos concurrentes no accedan a un mismo recurso común al mismo tiempo, ya que ello puede llevar a incoherencia en la información que se procesa. Un problema típico de la programación concurrente donde la exclusión mutua se hace presente es el ya conocido:

Problema de los jardines: Se desea controlar el número de visitantes a unos jardines. La entrada y la salida a los jardines se pueden realizar por dos puntos que disponen de 2 puertas. Se desea poder conocer en cualquier momento el número de visitantes a los jardines, por lo que se dispone de un dispositivo de cálculo con conexión en cada uno de los dos puntos de entrada que le informan cada vez que se produce una entrada o una salida.

Si analizamos el problema y lo llevamos al terreno de la programación concurrente, podemos asociar el proceso *P1* a un punto de entrada y el proceso *P2* al otro punto de entrada. Ambos procesos se ejecutan concurrentemente y utilizan una única variable, digamos ***x***, para llevar la cuenta del número de visitantes. El incremento o decremento de la variable se produce cada vez que un visitante entra o sale por una de las puertas. De esta forma, la entrada de un visitante por una de las puertas hace que se ejecute la instrucción:

x := x + 1

mientras que la salida de un visitante hace que se ejecute la instrucción:

x := x - 1

Aunque el proceso *P1* y el proceso *P2* se supongan ejecutados en procesadores distintos (lo cual no necesariamente es cierto), ambos utilizan la misma posición de memoria para guardar el valor de la variable compartida, es decir, de ***x***. Se puede dar la situación de que el planificador de procesos del sistema permita el entrelazado (interfoliación) de las operaciones elementales anteriores de cada uno de los procesos, lo cual inevitablemente producirá errores que son difíciles de detectar mediante una prueba del programa, ya que el que se produzcan depende de la temporización de dos procesos independientes. Éste ejemplo muestra la clara necesidad de sincronizar (poner de acuerdo) de alguna manera la actuación de ambos procesos de forma que no se produzcan interferencias entre ellos.

Para evitar este tipo de errores se pueden identificar aquellas regiones de los procesos que acceden a variables compartidas y dotarlas de la posibilidad de ejecución como si fueran una única instrucción. Surge entonces el concepto de ***región crítica***.

Región Crítica: Es la zona de código de un proceso concurrente donde se utiliza un recurso compartido. La exclusión mutua se consigue impidiendo que dos o más procesos concurrentes ejecuten a la vez su región crítica. Una región crítica solo puede ser ejecutada por un único proceso concurrente siguiendo una única secuencia de interfoliación. Si algún otro proceso desea ejecutar su región crítica, deberá esperar a que aquel que actualmente ejecuta la suya termine y lo indique. Se necesita por tanto algún mecanismo (*protocolos software*) que indique a los procesos cuando uno de ellos entra o sale de la región crítica[1].

Esto se consigue con dos protocolos de software, uno de entrada y otro de salida (lock o semáforo binario) que todo proceso debe forzosamente ejecutar antes y después de ejecutar su región crítica, y que constituyen un mecanismo de seguridad (*bloqueo*) que además sobrecarga la ejecución. Por ejemplo:

```
Task body P is
Begin
   Loop
        Resto de código;
        Pre_Protocolo;
              Región Crítica;
        Post-Protocolo;
   End Loop;
End P.
```

Candado o Semáforo Binario que produce la exclusión mutua

El Pre_Protocolo y Post_Protocolo utilizan una variable común2 a todos los procesos concurrentes la cual controla el acceso a la Región Crítica. El Pre_Protocolo o Protocolo de Entrada consultará el valor de esa variable y en función de dicho valor dejará entrar o bloqueará al proceso que desea entrar en su región crítica. El Post_Protocolo o Protocolo de Salida modificará el valor de la variable común para indicar que el recurso ya esta libre. La variable común se utiliza como cauce de comunicación entre los distintos procesos, con el objeto de acceder éstos a sus regiones críticas con "cierto orden".

[1] Es claro que el requerimiento de exclusión mutua incrementa el determinismo de los programas concurrentes.
[2] Que no se considera variable compartida.

3.2. La Sincronización

Supongamos que tenemos dos procesos concurrentes ocupados uno de ellos en llenar un buffer circular de datos y el otro en vaciarlo. Puesto que el primer proceso no puede poner más datos en el buffer si está lleno, y el segundo no puede tomar datos del buffer cuando está vacío, es necesario sincronizar la ejecución de ambos procesos en función del estado del buffer, consiguiendo:

1. Que el proceso productor se bloquee cuando el buffer esta lleno
2. Que el proceso consumidor se bloquee cuando el buffer esta vacío
3. que el productor desbloquee al consumidor
4. que el consumidor desbloquee al productor

Existen diversos mecanismos software utilizados para conseguir la sincronización, uno de ellos es el candado (lock) o semáforo binario, ya comentado en el apartado de la exclusión mutua, otro de ellos son las variables de condición (monitores) que son variables comunes a los procesos concurrentes que comparten recursos, donde dependiendo del valor de la variable de condición se produce una sincronización del tipo wait-notify, espera-notificación, de manera que los procesos se comunican y sincronizan al acceder a sus recursos compartidos mediante mensajes que indican el bloqueo o desbloqueo de uno a otro u otros procesos vía la variable de condición.

3.3. EL PROBLEMA DE LOS ESQUIMALES

Intentaremos una manera de implementar el bloqueo a una región crítica mediante el uso de una variable compartida en el problema denominado “de los esquimales”. Iremos refinando el algoritmo de manera que cada versión de dicho refinamiento presente el tipo de problemas que suele surgir en la implementación de la concurrencia entre procesos. El problema de los esquimales mostrará el uso de las variables compartidas para implementar la exclusión mutua entre dos procesos.

La ejecución concurrente de los procesos la indicaremos mediante la estructura cobegin/coend. La palabra cobegin indica el comienzo de la ejecución concurrente de los procesos que se señalan hasta la palabra coend. La acción de bloqueo se realiza con la activación de la variable común (indicador) y la de desbloqueo con su desactivación.

Primer Intento: ALTERNANCIA

Imaginemos 2 esquimales y un agujero para pescar como recurso compartido. Existe un iglú con un pizarrón. Solo uno de los esquimales puede acceder a la vez al pizarrón a la vez. Cuando uno de los esquimales quiere acceder al agujero para pescar debe consultar si tiene permiso para hacerlo en el pizarrón, si en el pizarrón se indica que el permiso lo tiene el otro esquimal espera un tiempo y lo vuelve a probar de nuevo más tarde. Si en el pizarrón se indica que tiene permiso, irá a pescar. Una vez que termine con el agujero, irá al pizarrón y cederá el permiso al otro esquimal. De manera más formal la codificación del algoritmo es:

```
PROGRAM PrimerIntento;
   VAR turno [1..2];

   PROCEDURE P1;
     BEGIN
          REPEAT
            WHILE turno = 2 DO ;      (* pasea *)
            ******  usa agujero de pesca  ******
            turno := 2;
            otras cosas
           FOREVER
        END;

    PROCEDURE P2;
        BEGIN
           REPEAT
             WHILE turno = 1 DO ;            (* pasea *)
             ******  usa agujero de pesca  ******
             turno := 1;
             otras cosas
           FOREVER
        END;

BEGIN
   turno := 1
   COBEGUIN
        P1; P2
   COEND
END
```

El inconveniente de este algoritmo es la alternancia obligada en el uso del recurso, si uno de los procesos es cancelado mientras usa el recurso nunca podrá ceder su uso de nuevo.

Segundo Intento: FALTA DE EXCLUSION

Siguiendo con el paradigma anterior, haremos uso de dos iglúes con su correspondiente pizarrón. Los pizarrones tienen dos indicadores, pescando o NOpescando, cada uno de los pizarrones corresponden a cada uno de los esquimales. Si queremos hacer uso del agujero para pescar deberemos consultar en el pizarrón del

otro esquimal, si indica que está pescando saldremos y esperaremos para volverlo a intentar más tarde. Si en el pizarrón pone NOpescando iremos a nuestro pizarrón y pondremos pescando. Cuando terminemos de usar el agujero de pescar iremos a nuestro pizarrón y pondremos NOpescando. De manera más formal podríamos codificar el algoritmo como:

```
PROGRAM SegundoIntento;
  VAR pizarra1, pizarra2: (pescando NOpescando);

  PROCEDURE P1;
    BEGIN
       REPEAT
         WHILE pizarra2 = pescando DO;  (* pasea *)
         pizarra1 := pescando;
         *** usa agujero de pesca ***
         pizarra1 := NOpescando;
         otras cosas
      FOREVER
    END;

  PROCEDURE P2;
    BEGIN
       REPEAT
         WHILE pizarra1 = pescando DO;   (* pasea *)
         pizarra2 := pescando;
         *** usa agujero de pesca ***
         pizarra2 := NOpescando;
         otras cosas
      FOREVER
    END;

BEGIN
  pizarra1 := NOpescando;
  pizarra2 := NOpescando;
  COBEGIN
       P1; P2
  COEND
END.
```

Este algoritmo puede generar falta de exclusión mutua si los dos procesos tienen libre el recurso, a la vez consultan el estado de los oponentes y al comprobar que está libre los dos indican estado de uso del recurso y lo usan.

Tercer Intento: INTERBLOQUEO (Espera Infinita)

Ahora actuaremos a la inversa para tratar de evitar la falta de exclusión mutua, cuando un proceso quiera utilizar el recurso, primero indica que quiere hacerlo y luego espera a que esté libre. De manera más formal podríamos codificar el algoritmo como:

```
PROGRAM TercerIntento;
 VAR pizarra1, pizarra2: (pescando NOpescando);

 PROCEDURE P1;
   BEGIN
  REPEAT
    pizarra1 := pescando;
```

```
    WHILE pizarra2 = pescando DO;  (* pasea *)
    *** usa el agujero de pesca ***
    pizarra1 := NOpescando;
    otras cosas
      FOREVER
  END;

PROCEDURE P2;
  BEGIN
 REPEAT
    pizarra2 := pescando;
    WHILE pizarra1 = pescando DO;  (* pasea *)
    *** usa el agujero de pesca ***
    pizarra2 := NOpescando;
    otras cosas
      FOREVER
  END;

BEGIN
  pizarra1 := NOpescando;
  pizarra2 := NOpescando;
  COBEGIN
        P1; P2
  COEND
END.
```

En el caso que los dos esquimales estén sin pescar (NOpescando) y los dos pretendan acceder al agujero al mismo instante, los dos activarán en su pizarrón el estado de intención de pescar (pescando), al ir a consultar la pizarra del oponente comprobarán que está pescando (o en intención de hacerlo), en este caso ninguno de los dos esquimales podrá acceder al recurso ni podrá cambiar su estado. Hemos llegado a un estado de exclusión mutua (los dos procesos quieren acceder al mismo recurso, pero para que este les sea concedido los dos procesos deben realizar una acción que depende mutuamente de la acción del oponente para poderse realizar).

Cuarto Intento: ESPERA INFINITA

Para solucionar el problema anterior añadiremos el trato de cortesía, si un proceso ve que su oponente quiere hacer uso del recurso, se lo cede. De manera más formal podríamos codificar el algoritmo como:

```
PROGRAMA CuartoIntento;
  VAR pizarra1, pizarra2: (pescando, NOpescando);

  PROCEDURE P1;
    BEGIN
 REPEAT
   pizarra1 := pescando;
   WHILE pizarra2 = pescando DO
     BEGIN
        (* tratamiento de cortesía *)
         pizarra1 := NOpescando;
        (* date una vuelta *)
         pizarra1 := pescando
     END;
```

```
  *** usa el agujero de pesca ***
  pizarra1 := NOpescando;
  otras cosas
 FOREVER
   END;

 PROCEDURE P2;
   BEGIN
 REPEAT
   pizarra2 := pescando;
   WHILE pizarra1 = pescando DO
       BEGIN
         (* tratamiento de cortesía *)
          pizarra2 := NOpescando;
         (* date una vuelta *)
          pizarra2 := pescando
       END;
   *** usa el agujero de pesca ***
   pizarra2 := NOpescando;
   otras cosas
 FOREVER
END;

BEGIN
  pizarra1 := NOpescando;
  pizarra2 := NOpescando;
  COBEGIN
  P1; P2
  COEND
END.
```

Este tratamiento de cortesía puede conducir a que los procesos se queden de manera indefinida cediéndose mutuamente el paso. Esta solución no asegura que se acceda al recurso en un tiempo finito.

Quinto Intento: ALGORITMO DE DEKKER

La solución al problema de la exclusión mutua que sigue, se atribuye al matemático holandés T. Dekker y fue presentada por Dijkstra en 1968. Se utiliza al igual que en otro algoritmo llamado Algoritmo de Peterson, una variable turno que sirve para establecer la prioridad relativa de los dos procesos, es decir, en caso de conflicto ésta variable servirá para saber, a quien se le concede el recurso y su actualización se realiza en la región crítica, lo que evita que pueda haber interferencias entre los procesos. De manera más formal la codificación del algoritmo es:

```
PROGRAMA AlgoritmoDeDekker;
  VAR pizarra1, pizarra1: (pescando, NOpescando);
  Turno:[1..2];

  PROCEDURE P1;
    BEGIN
       REPEAT
         pizarra1 := pescando
         WHILE pizarra2 = pescando DO
           IF turno = 2 THEN
           BEGIN
```

```
            (* tratamiento de cortesía *)
             pizarra1 := NOpescando;
              WHILE turno = 2 DO;  (* date una vuelta *)
              pizarra1 := pescando
          END
        *** usa el agujero de pesca ***
        turno := 2;
        pizarra1 := NOpescando;
        otras cosas
      FOREVER
  END;

PROCEDURE P2;
  BEGIN
      REPEAT
        pizarra2 := pescando
        WHILE pizarra1 = pescando DO
          IF turno = 1 THEN
          BEGIN
             (* tratamiento de cortesía *)
             pizarra2 := NOpescando;
             WHILE turno = 1 DO;   (* date una vuelta *)
             Pizarra2 := pescando
          END
        *** usa el agujero de pesca ***
        turno := 1;
        pizarra2 := NOpescando;
        otras cosas
       FOREVER
  END;

 BEGIN
      pizarra1 := NOpescando;
      pizarra2 := NOpescando;
      turno := 1;
      COBEGIN
         P1; P2
      COEND
 END.
```

Este algoritmo asegura la exclusión mutua y está libre de interbloqueos.

3.4. SEMÁNTICA DE LA PROGRAMACIÓN CONCURRENTE

En ocasiones, será necesario definir un medio de expresar el contenido semántico o significado de un programa concurrente. Existen 2 formas de expresar la semántica de un programa concurrente.

Bajo el enfoque operacional, el significado de un programa concurrente viene descrito por una secuencia de interfoliación de instrucciones atómicas de los distintos procesos que integran el sistema concurrente. Como ya se ha mencionado, la exclusión mutua y la sincronización restringen el conjunto de interfoliaciones posible, incrementando el determinismo y la secuencialidad del programa concurrente. Este enfoque define el significado de un programa por lo que hace, pero no ayuda a verificar si esta o no correcto.

Ello da origen al enfoque axiomático, que sirve para probar la correccción de programas concurrentes. En este enfoque, cada sentencia del programa adopta la forma:

$$\{ P \} S \{ Q \}$$

donde P es una precondición que debe cumplirse para que la sentencia S sea ejecutable, y Q es una postcondición que se cumple siempre que se ejecute S. Demostrar que un programa es correcto equivale entonces a probar un teorema de la lógica de programas, que indica que existe una secuencia de sentencias que pueden ejecutarse correctamente.

3.5. CORRECCIÓN EN LOS SISTEMAS CONCURRENTES

Cuando se trabaja con programas puramente secuenciales, la corrección de éstos es evidente: a una entrada dada deberá corresponder una salida deseada. Por el contrario, cuando se trabaja con programas o sistemas concurrentes, es adecuado en muchas de las veces disponer de un modelo formal para demostrar la corrección, debido a que el no-determinismo y la ambigüedad están presentes.

Sea una Formula(x) una propiedad sobre las variables de entrada y una Relación(x,y) sobre las variables de entrada y las de salida. Entonces, para todos los valores, digamos x e y se define un programa como:

a) parcialmente correcto si la Formula(p) es cierta, si el programa comenzó con x==a, el programa termina y la Relación(a,b) es cierta siendo y==b cuando el programa acaba (es decir, el programa puede no terminar).
b) Totalmente correcto si es parcialmente correcto y el programa además termina siempre.

A continuación definimos una serie de propiedades importantes que terminan de completar la correctitud de un programa concurrente.

Propiedad de Seguridad (Safety)

Decimos que un sistema o programa concurrente es seguro cuando dicho programa o sistema preserva la exclusión mutua. Esta propiedad se relaciona con el comportamiento estático del sistema concurrente. Un programa concurrente con esta propiedad es parcialmente correcto.

Propiedad de Vivacidad (Liveness)

Un programa concurrente tiene la propiedad de vivacidad cuando éste previene los interbloqueos (deadlocks), que se producen cuando una serie de recursos protegidos por exclusión mutua están distribuidos de tal manera entre los procesos que, todo proceso posee un recurso y espera a otro que esta en posesión de un proceso distinto.

Propiedad de No Inanición (Not Lockout)

Es aquella propiedad que asegura que el conjunto de procesos que integran al sistema o programa concurrente, siempre accederán al procesador. En otras palabras, la inanición se da cuando hay un conjunto de procesos que por ciertas circunstancias nunca acceden al procesador. La inanición (lockout) es menos grave que un interbloqueo (deadlock), ya que al menos otros procesos siguen en ejecución.

Propiedad de Equidad (Fairness)

Un sistema sin interbloqueos y sin inaniciones es equitativo cuando todos los procesos disponen de una fracción proporcionada de tiempo de procesador y no hay procesos marginados para acceder a dicha fracción de tiempo.