# 7. Programación Concurrente

## 1. ¿Qué es la programación concurrente?

Se conoce por programación concurrente a la rama de la informática que trata de las técnicas de programación que se usan para expresar el paralelismo entre tareas y para resolver los problemas de comunicación y sincronización entre procesos.

El principal problema de la programación concurrente corresponde a no saber en que orden se ejecutan los programas (en especial los programas que se comunican). Se debe tener especial cuidado en que este orden no afecte el resultado de los programas.

## 2. Cobegin y Coend.

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 sentencia coend. Veamos un ejemplo:

```
S1;
COBEGIN
        S2;
        S3;
COEND
S4;
```

Esto quiere decir que:

- Primeramente se debe ejecutar S1, y no se puede ejecutar en paralelo con nada.
- S2 y S3 se pueden ejecutar en paralelo, luego de ejecutado S1
- S4 se ejecutará al terminar S2 y S3, y no se puede ejecutar en paralelo con nada.

Veamos un ejemplo de un programa no consistente: el orden de ejecución de las sentencias afecta el resultado.

```
J= 10;
COBEGIN
        Print J;
        J = 1000;
COEND
Print J;
```

Si se ejecutara primero la sentencia *Print J* entonces se imprimiría "10 1000". Pero si se ejecutara primero la sentencia *J=1000* entonces se imprimiría "1000 10". El resultado no es el mismo, por lo cual el programa es inconsistente.

## 3. Sección crítica y mutua exclusión

El método más sencillo de comunicación entre los procesos de un programa concurrente es el uso común de unas variables de datos. Esta forma tan sencilla de comunicación puede llevar, no obstante, a errores en el programa ya que el acceso concurrente puede hacer que la acción de un proceso interfiera en las acciones de otro de una forma no adecuada.

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. Se denomina **Sección Crítica** a aquellas partes de los procesos concurrentes que no pueden ejecutarse de forma concurrente o, también, que desde otro proceso se ven como si fueran una única instrucción. Esto quiere decir que si un proceso entra a ejecutar una sección crítica en la que se accede a unas variables compartidas, entonces otro proceso no puede entrar a ejecutar una región crítica en la que acceda a variables compartidas con el anterior.

Las secciones críticas se pueden **mutuo excluir.** Para conseguir dicha exclusión se deben implementar protocolos software que impidan o el acceso a una sección crítica mientras está siendo utilizada por un proceso.

## 4. Semáforos: Concepto

Dijkstra dio en 1968 una solución al problema de la exclusión mutua con la introducción del concepto de semáforo binario. Está técnica permite resolver la mayoría de los problemas de sincronización entre procesos y forma parte del diseño de muchos sistemas operativos.

Un semáforo binario es un indicador (S) de condición que registra si un recurso está disponible o no. Un semáforo binario sólo puede tomar dos valores: 0 y 1. Si, para un semáforo binario, S = 1 entonces el recurso está disponible y la tarea lo puede utilizar; si S = 0 el recurso no está disponible y el proceso debe esperar.
Los semáforos se implementan con una cola de tareas a la cual se añaden los procesos que están en espera del recurso.

Así pues, un semáforo binario se puede definir como un tipo de datos especial que sólo puede tomar los valores 0 y 1, con una cola de tareas asociada y con sólo tres operaciones para actuar sobre él:

- INIT (S, val): inicializa al semáforo en el valor val (0 ó 1)

- P(S):
  **if** S = 1 **then**
  S := 0
  **else**
  Suspender la tarea que hace la llamada y ponerla en la cola de tareas

- V(S):
  **if** la cola de tareas está vacía **then**
  S := 1
  **else**
  Reanudar la primera tarea de la cola de tareas

La operación **INIT** se debe llevar a cabo antes de que comience la ejecución concurrente de los procesos ya que su función exclusiva es dar un valor inicial al semáforo.

Un proceso que corre la operación **P** y encuentra el semáforo a 1, lo pone a 0 y prosigue su ejecución. Si el semáforo está a 0 el proceso queda en estado de *bloqueado* hasta que el semáforo se libera.

Cuando se ejecuta la operación **V** puede haber varios procesos en la lista o cola. El proceso que la dejará para pasar al estado listo dependerá del esquema de gestión de la Cola. Si no hay ningún proceso en espera del semáforo este se deja libre (S := 1) para el primero que lo requiera.

El semáforo binario resulta adecuado cuando hay que proteger un recurso que pueden compartir varios procesos, pero cuando lo que hay que proteger es un conjunto de recursos similares, se puede usar una versión más general de semáforo que lleve la cuenta del número de recursos disponibles. En este caso el semáforo se inicializa con el número total de recursos disponibles (N) y las operaciones P y V se diseñan de modo que se impida el acceso al recurso protegido por el semáforo cuando el valor de éste es menor o igual que cero. Cada vez que se solicita y obtiene un recurso, el semáforo se decrementa y se incrementa cuando que uno de ellos se libera.
Las operaciones que tenemos son las mismas, con algunas diferencias en su semántica:

- INIT (S, val): inicializa al semáforo en el valor val (puede ser cualquier valor)

- P(S): **if** S > 0 **then**
  S = S -1;
  **else**
  Suspender la tarea que hace la llamada y ponerla en la cola de tareas

- V(S): **if** la cola de tareas está vacía **then**
  S = S +1;
  **else**
  Reanudar la primera tarea de la cola de tareas

### 5. Semáforos: mutua exclusión

La exclusión mutua se realiza fácilmente utilizando semáforos. La operación **P** se usará como procedimiento de bloqueo antes de acceder a una sección crítica y la operación **V** como procedimiento de desbloqueo. Se utilizarán tantos semáforos como clases de secciones críticas se establezcan.

```
process P1:
        P (S) ;
        Sección Crítica
        V (S) ;
        (* resto del proceso *)
end P1;
```

```
process P2:
        P (S) ;
        Sección Crítica
        V (S) ;
        (* resto del proceso *)
end P2;

process Principal:
        INIT(S,1);
        COBEGIN
                P1;
                P2;
        COEND
end Principal;
```

Con esta estructura, el primer proceso que ejecute P(S) pone a S en cero, pasa a ejecutar la sección crítica y generará que el otro proceso, cuando quiera acceder a la sección critica, se bloquee.
Cuando el primer proceso termine la sección crítica, ejecutará V(S), liberando al otro proceso bloqueado y permitiéndole ejecutar su sección crítica.

Si tuviéramos n procesos, y no quisiéramos que se ejecuten mas de m secciones críticas simultáneamente, basta con cambiar INIT(S,m).

### 6. Semáforos: Problema Productor – Consumidor

Este problema aparece en distintos lugares de un sistema operativo y caracteriza a aquellos problemas en los que existe un conjunto de procesos que *producen* información que otros procesos *consumen,* siendo diferentes las velocidades de producción y consumo de la información. Este desajuste en las velocidades, hace necesario que se establezca una sincronización entre los procesos de manera que la información no se pierda ni se duplique, consumiéndose en el orden en que es producida.

Para ello se considera un buffer común, del que el consumidor toma datos y donde el productor los coloca. Se debe mutuoexcluir, porque ambos, productor y consumidor, pueden modificarlo.
Cuando el productor desea colocar un dato en el buffer y este está lleno, el productor se quedará bloqueado hasta que exista espacio disponible en el buffer.
Cuando el consumidor desea tomar un dato en el buffer y este está vacío, el consumidor se quedará bloqueado hasta que exista algún dato disponible en el buffer.

```
process Productor:
        repeat
                d= producirDato();
                P(libres);
                P(S);
                AgregarBuffer(d);
                V(S);
                V(ocupados);
        Forever;
end Productor;
```

```
process Consumidor:
        repeat
                P(ocupados);
                P(S);
                d = SacarDato();
                V(S);
                V(libres);
                consumirDato(d);
        Forever;
end Consumidor;
```

```
process Principal:
        INIT(libres, Tamaño_Buffer);
        INIT(ocupados, 0);
        INIT(S,1);
        COBEGIN
                Productor
                Consumidor
        COEND
end Principal;
```

Se necesitarán como variables compartidas los semáforos *S, ocupados* y *libres*, que representarán:

- S será el semáforo para mutuoexcluir la sección critica correspondiente a modificar el buffer.
- *Ocupados* será el semáforo que contabilizará la cantidad de datos que hay en el buffer.
- *Libres* será el semáforo que contabilizará la cantidad de lugares libres que le restan al buffer.

## 7. Semáforos: Problema Lector - Escritor

Otro problema famoso es el de los lectores y escritores, que modela el acceso a una base de datos. Supongamos una base de datos, con muchos procesos que compiten por leer y escribir en ella. Se puede permitir que varios procesos lean de la base de datos al mismo tiempo, pero si uno de los procesos está escribiendo (es decir, modificando) la base de datos, ninguno de los demás debería tener acceso a ésta, ni siquiera los lectores.

En esta solución, el primer lector que obtiene el acceso a la base de datos realiza un P sobre el semáforo *wrt.*. Los lectores siguientes sólo incrementan un contador, *Cant_Lectores*. Al salir los lectores, éstos decrementan el contador, y el último en salir realiza un *V* sobre el semáforo, lo que permite entrar a un escritor bloqueado, si existe.

Una hipótesis implícita en esta solución es que los lectores tienen prioridad sobre los escritores. Si surge un escritor mientras varios lectores se encuentran en la base de datos el escritor debe esperar. Pero si aparecen nuevos lectores, y queda al menos un lector accediendo a la base de datos, el escritor deberá esperar hasta que no haya más lectores interesados en la base de datos.

```
process Escritor:
        repeat
                P(wrt);
                Escribir();
                V(wrt);
        Forever;
end Escritor;
```

```
process Lector:
        repeat
                P(S);
                Cant_Lectores ++;
                If (Cant_Lectores == 1)
                        P(wrt);
                V(S);

                Leer();

                P(S);
                Cant_Lectores --;
                If (Cant_Lectores == 0)
                        V(wrt);
                V(S);
        Forever;
end Lector;

process Principal:
        INIT(wrt, 1);
        INIT(S,1);
        Int Cant_Lectores = 0;
        COBEGIN
                Lector;
                ….
                Lector;
                Escritor;
                ….
                Escritor;
        COEND
end Principal;
```

Se necesitarán como variables compartidas los semáforos *S* y *wrt*, que representarán:

- S será el semáforo para mutuoexcluir la sección critica correspondiente a modificar la variable compartida *Cant_Lectores*.
- *Wrt* será el semáforo que indicará cuando se habilita a escribir y cuando se debe esperar.

### 8. Semáforos: Problema de los filósofos que cenan

Cinco filósofos se sientan alrededor de una mesa y pasan su vida cenando 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.

Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer.

Si todos los filósofos cogen el tenedor que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Entonces los filósofos se morirán de hambre. Este bloqueo mutuo se denomina interbloqueo o *deadlock*.

El problema consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de hambre.

Veamos una posible Solución:

```
process Principal:
        INIT(Comedor, 4);
        For (int i=0; i<4; i++)
                INIT(tenedor[i], 1);
        COBEGIN
                Filósofo (0);
                Filósofo (1);
                Filósofo (2);
                Filósofo (3);
                Filósofo (4);
        COEND
end Principal;
```

```
process Filósofo(int identidad):
        int izq, der;
        izq = identidad;
        der = (identidad + 1) mod 5
        repeat
                Pensar();

                P(Comedor);
                P(tenedor[izq]);
                P(tenerdor[der]);

                Comer();

                V(tenedor[der]);
                V(tenedor[izq]);
                V(Comedor);
        forever
end Filósofo;
```

Se necesitarán como variables compartidas los semáforos *Comedor* y el arreglo de Semáforos *tenedor* (de largo 5), que representarán:

- *Comedor* se utiliza para permitir que estén el la mesa a lo sumo 4 filósofos. Si todos estuvieran en la mesa a la vez, y a todos les da hambre, podrían generarse deadlocks.
- cada semáforo del arreglo representa si el tenedor está o no siendo usado por algún filósofo (el de su izquierda o derecha).

Notar que los filósofos reciben como parámetro su identidad para saber que tenedores deben tomar.