Capítulo 6
Monitores

Anteriormente hemos analizado el mecanismo de Semáforos que nos permite resolver los problemas de sincronización y exclusión mutua que nos podemos encontrar cuando abordamos la resolución de problemas en los que es necesaria la utilización de varios procesos concurrentes. Sin embargo, a pesar de que si utilizamos de manera correcta los mismos nos permite alcanzar soluciones eficientes, presentan una serie de inconvenientes que hacen difícil su aplicación a problemas más complejos.
Observamos que en semáforos el uso de las variables no es explícito, es decir, analizando el código de un programa que utilice semáforos no podemos determinar que semáforos están siendo utilizados para la sincronización de procesos y que semáforos se utilizan para la exclusión mutua en el acceso a secciones críticas. Añadiendo que las operaciones wait y signal están dispersas por el código y que cualquier supresión/adición de una instrucción puede provocar interbloqueos, el mantenimiento de los programas puede resultar costoso.
En 1974 Hoare publicó un trabajo donde describía el funcionamiento de un **monitor**. Los monitores son mecanismos de abstracción de datos, que encapsulan las representaciones de recursos y proporcionan un conjunto de operaciones que son la únicas que se pueden realizar sobre dichos recursos, es decir, contiene las variables que representan el estado del recurso y los procedimientos que implementan operaciones sobre el recurso.
Un proceso sólo puede acceder a las variables del monitor usando los procedimientos exportados por el monitor. La exclusión mutua en el acceso a los procedimientos del monitor está garantizada por el mismo.

Componentes

Un monitor se compone con los siguientes elementos:

- Un conjunto de variables locales, las mismas se utilizan para almacenar el estado interno del recurso que representa.
- Un código de inicio que se ejecuta antes de utilizar el monitor y tiene como objetivo iniciar las variables locales.
- Conjunto de procedimientos internos que manipulan las variables locales. Obviamente los procedimientos del monitor pueden tener parámetros y variables locales.
- Una declaración de procedimientos que son exportados.

Como podemos observar el monitor se puede implementar como un ***objeto*** siendo las variables locales sus atributos, el código de inicio su constructor, métodos privados y públicos con sus respectivos parámetros.

Funcionamiento

El control de la exclusión mutua en un monitor se basa en la existencia de una cola asociada al monitor que denominaremos **cola del monitor**. Cuando un proceso activo está ejecutando uno de los procedimientos del monitor “el proceso está dentro del monitor”, y otro proceso activo intenta ejecutar otro o el mismo procedimiento, el código de acceso al monitor (generado por el compilador) bloquea el proceso que realiza la llamada y lo inserta el la cola del monitor (siguiendo una política FIFO), impidiendo a dos procesos activos que estén al mismo tiempo dentro del monitor. Cuando un proceso activo abandona el monitor, el monitor selecciona el proceso que está al frente de la cola del monitor y lo desbloquea. Si la cola del monitor está vacía el monitor quedará libre y el primer proceso que lo requiera entrará al monitor.

Bueno, veamos un ejemplo en pseudo código utilizando la teoría de monitores:
Supongamos que varios procesos deben incrementar el valor de una variable compartida y poder examinar su valor en cualquier momento. Como ya sabemos, el acceso a dicha variable debe realizarse en exclusión muta. Entonces definiremos un monitor que encapsulará dicha variable compartida.
Para implementar dicho monitor necesitaremos:

- Una variable local que representa el valor de la variable compartida.
- Un procedimiento para incrementar la variable y otro para obtener su valor.
- Iniciar la variable compartida desde el código de inicio.

A continuación el psedo-código:

```
monitor incremento;
   var i:integer;
   export inc,leer;

   procedure inc;
   begin
      i:=1+1;
   end;

   procedure leer(var valor: integer);
   begin
      valor:=i;
   end;
begin
   i:=0;
end;
```

La política de acceso al monitor nos **garantiza** la ejecución de diferentes procesos invocando a los procedimientos exportados en **mutua exclusión**.

Condición de sincronización en Monitores:
Vimos que hay situaciones donde en las que un proceso activo no puede continuar su ejecución (ejemplo productor/consumidor). Para ello los monitores utilizan **variables de condición**. Las variables de condición no son accesibles para el programador y representan colas (FIFO). Estas variables nos permitirán bloquear procesos activos que no pueden continuar su ejecución dentro del monitor, y desbloquearlos cuando cuando la situación que provocó su bloqueo ya no se dé debido a que otro proceso haya modificado el estado del monitor. Para esto es necesario definir al menos dos operaciones sobre las variables de condición: *delay* para bloquear a un proceso y *resume* para desbloquearlo.
Bloqueo de procesos, delay: La ejecución de la operación *delay(C)*, siendo C una variable de condición hace que el proceso que la ejecuta se bloquee al final de la cola asociada a la variable de condición C.
La diferencia con la operación *wait(sem)* de semáforos es clara, *delay(C)* provoca el bloqueo incondicional del proceso, mientras que *wait(sem)* solo bloquea el proceso si el valor del semáforo es cero. Además debemos observar que si un proceso adquiere el acceso exclusivo al monitor y ejecuta la operación *delay* el proceso es bloqueado reteniendo la exclusión mutua con lo que ningún otro proceso podrá acceder al monitor para desbloquear al proceso. Para evitar esto, antes que un

proceso sea bloqueado en la cola de asociada a la variable de condición tiene que liberar la exclusión mutua, permitiendo a otros procesos acceder al monitor. Por lo tanto la operación *delay(C)* se puede interpretar como:
liberar la exclusión mutua del monitor
bloquear el proceso que realiza la llamada al final de la cola C

Desbloqueo de procesos, resume: Cuando un proceso ha sido bloqueado mediante la ejecución de la operación *delay(C)* sólo puede ser desbloqueado por otro proceso que ejecute la operación *resume(C)* sobre la misma variable de condición. Básicamente, la operación *resume(C)* extrae el proceso que se encuentra en el frente de la correspondiente cola y lo prepara para su ejecución. En el caso que la cola C esté vacía, la operación *resume(C)* se transforma en un operación nula. Podemos observar aquí la principal diferencia respecto a la operación *signal(sem)* ya que si la cola asociada al semáforo está vacía la operación no es nula sino que incrementa la variable entera asociada al semáforo.
Ahora bien, si un proceso A desbloquea a un proceso B, el proceso B no puede continuar su ejecución sino tendríamos dos procesos en el monitor, rompiendo la exclusividad de ejecución de procesos dentro del monitor, por lo tanto, si la operación *resume(C)* desbloquea un proceso, el proceso que ejecuta la operación abandona el monitor liberando la exclusión mutua que es cedida al proceso desbloqueado, pudiendo éste continuar su ejecución dentro del monitor. El proceso que cede el paso al proceso desbloqueado tendrá preferencia sobre el resto de los procesos que esperan por acceder al monitor a través de una cola de cortesía.

Resolución de problemas a través de monitores

El problema del productor/consumidor
A la hora de resolver un problema utilizando monitores, lo primero que debemos hacer es identificar el recurso que va a ser compartido y los procesos activos que van a utilizar dicho recurso. En el problema del productor/consumidor estos conceptos están muy claros: el *buffer* es el recurso compartido y el productor y el consumidor son los procesos.
Bueno vamos a definir los componentes de nuestro monitor, como variables locales del monitor necesitaremos: el array que utilizaremos de *buffer* (recurso), un puntero que indique la posición para insertar elementos *"cola"*, un puntero que indique la posición para extraer elementos *"frente"* y el número de elementos que existen en el *buffer "tam"* y una constante para indicar el tamaño del *buffer*.
Teniendo en cuenta la lógica de la solución implementada con semáforos necesitaremos dos variables de condición: una para bloquear al productor cuando el *buffer* esté lleno *"nolleno"* y otra para bloquear al consumidor cuando el *buffer* este vacío *"novacio"*. Por último necesitaremos que nuestro monitor exporte los procedimientos insertar y extraer (elementos del buffer).

A continuación nuestro pseudo código del monitor:

```
monitor buffer;
   const N=10;
   var tam,frente,cola:integer;
      nolleno,novacio: condition;
      recurso: array[0..N-1] of item;
   export insertar,extraer;

   procedure insertar(elemento:item)
   begin
      if tam=N then delay(nolleno); (* bloquearse si no hay espacio.*)
      recurso[cola]:=elemento;
      cola:=(cola+1) mod N;
      tam:=tam+1;
      resume(novacio); (* desbloquear al consumidor.*)
   end;

   procedure extraer(var elemento:item);
   begin
      if tam=0 then delay(novacio); (* bloquearse si no hay
                                                   elementos.*)
      elemento:=recurso[frente];
      frente:=(frente+1) mod N;
      tam:=tam-1;
      resume(nolleno); (* desbloquear a un productor. *)
   end;
begin
   frente:=0;
   cola:=0;
   tam:=0;
end;
```

Solución de monitor en Java

Se puede observar que en la unidad anterior (Primitivas de sincronización en Java) realizamos un monitor con una sola condición (el mismo objeto que hace de “monitor”) ya que podemos observar que todos sus métodos son sincronizados y el mismo objeto hace de condición ya que se hace el Lock sobre dicho objeto . De ahí podemos decir que las primitivas de Java apuntaban a que el programador resuelva problemas de concurrencia “utilizando monitores”, las comillas y cursiva no son casualidad, las primitivas de Java no se corresponden a la definición de un monitor, solo se acercan al concepto. Entonces, si solo puedo implementar una condición en el monitor, ¿no puedo resolver problemas con más de una condición como productores y consumidores?.

Tranquilos, Java se apiadó de nosotros y nos brinda “Lock and Condition”, para que trabajemos con el concepto de monitores al cien por cien.

Lock and Condition

En Java no hay una palabra clave o clase para crear un Monitor. Para implementar un monitor se debe crear una nueva clase y usar las clases: *Lock* y *Condition*. Lock es una interface y ReentrantLock es la implementación más utilizada (buscar algo mini de reentrantlock). Los Lock tienen la misma semántica que los bloques synchronized, y un bloque Lock se lo marca con lock()/unlock(). Cabe preguntarse porque existe el bloque Lock si tiene la misma semántica que el bloque synchronized, bien, con synchronized no podemos utilizar variables de condición. Podemos decir que si no necesitamos variables de condición utilizaremos directamente synchronized.
Podemos crear condiciones usando el método newCondition sobre el lock. Una condición es una variable del tipo Condition. Por lo tanto podemos hacer que el "current thread" haga *wait* sobre la condición usando el método *await*, y a su vez podemos hacer el mismo haga *signal* y *signalAll*. La operación *signalAll* levanta todos los hilos esperando sobre la variable de condición, mientras que *signal* solo levanta un hilo.

Ahora bien, veamos la solución de problema de Productor / Consumidor en Java:

```
/*
 *  Monitor
 */
public class Buffer {

    public static final int CAPACIDAD = 5;

    private int frente, cola, tamanio;
    private Item[] buffer;

    private Lock lock;
    private Condition noLleno, noVacio;

    public Buffer() {
        super();
        this.lock = new ReentrantLock();
        this.noLleno = this.lock.newCondition(); // hilos que esperan que el buffer no este lleno
        this.noVacio = this.lock.newCondition(); // hilos que esperan que el buffer no este vacio
        this.buffer = new Item[CAPACIDAD];
        this.tamanio = 0;
    }
```

Métodos insertar/exportar

```
public void insertar(Item item) {
    this.lock.lock(); // obtengo lock del objeto
    try {
        if (this.tamanio == CAPACIDAD) {
            this.noLleno.await(); // buffer lleno, hilo espera a que se cumpla esta condición (que no esté lleno)
        }
        this.buffer[this.cola] = item;
        this.cola = (this.cola + 1) % CAPACIDAD;
        this.tamanio++;
        this.noVacio.signal(); // se inserto un item en el buffer, se emite un signal por si hay algún hilo
                               // esperando esta condición (que no esté vacío)
                               // IMPORTANTE: Recordar la diferencia con Semaphore.
        System.out.println(Thread.currentThread().getName() + " inserto: " + item.getPalabra());
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        this.lock.unlock(); // libero lock del objeto
    }
}
```

```
public Item extraer() {
    try {
        this.lock.lock(); // obtengo lock del objeto
        if (this.tamanio==0) {
            this.noVacio.await(); // buffer vacío, hilo espera a que se cumpla esta condición (que no esté vacío)
        }
        Item item = this.buffer[this.frente];
        this.frente = (this.frente + 1) % CAPACIDAD;
        this.tamanio--;
        this.noLleno.signal(); // se extrajo un item del buffer, se emite un signal por si hay algún hilo
                               // esperando esta condición (que no esté lleno)
                               // IMPORTANTE: Recordar la diferencia con Semaphore.
        System.out.println(Thread.currentThread().getName() + " extrajo: " + item.getPalabra());
        return item;
    } catch (InterruptedException e) {
        e.printStackTrace();
        return null;
    } finally {
        this.lock.unlock(); // libero lock del objeto
    }
}
```

Código completo en proyecto: monitor-productor-consumidor (pedir a Ariel).

El problema de los lectores/escritores

Como ya vimos anteriormente este problema admite dos soluciones dependiendo de la política de acceso elegida, es decir si se le da prioridad a los lectores o a los escritores.
Independientemente de la política elegida debemos saber que:

- Existen dos tipos de procesos activos, lectores y escritores.
- El monitor será el encargado de controlar el acceso al recurso (el mismo no está dentro del monitor).
- Necesitamos cuatro operaciones para acceder al recurso: abrirLectura, cerrarLectura, abrirEscritura y cerrarEscritura.

Prioridad Lectores

En este caso, los lectores pueden acceder siempre al recurso a no ser que exista un escritor que está escribiendo en el recurso. Los escritores deben esperar hasta que no existan lectores accediendo al recurso.
Para implementar la solución necesitaremos una variable entera *n1* para tener la cantidad de lectores accediendo al recurso, una variable booleana *escribiendo* que nos indique si hay al algún escritor accediendo al recurso y por último dos variables de condición: una para bloquear a los lectores *lector* y otra para los escritores *escritor*.
Teniendo en cuenta el análisis anterior observamos la solución:

```
process type lector;
begin
   ...
   L_E.abrir_lectura;
   Leer del recurso;
   L_E.cerrar_lectura;
   ...
end;
```

```
process type escritor;
begin
   ...
   L_E.abrir_escritura;
   Escribir en el recurso;
   L_E.cerrar_escritura;
   ...
end;
```

Observamos que tanto la lectura como la escritura en el recurso se realizan en los procesos lector y escritor respectivamente y no el Monitor.

```
monitor L_E;
var
   nl:integer;
   escribiendo:boolean;
   lector,escritor:condition;
   export abrir_lectura,cerrar_lectura,
      abrir_escritura,cerrar_escritura;

   procedure abrir_lectura;
   begin
      if (escribiendo) then delay(lector);
      (* bloquearse si un proceso escritor está accediendo.*)
      nl:=nl+1;
      resume(lector); (* desbloquear en cadena al resto de lectores.*)
   end;

   procedure cerrar_lectura;
   begin
      nl:=nl-1;
      if (nl=0) then resume(escritor)
      (* desbloquear a un escritor si no quedan lectores *)
   end;
   procedure abrir_escritura;
   begin
      if (nl<>0) or escribiendo then delay(escritor);
      (* bloquearse si el recurso está ocupado. *)
      escribiendo:=true
   end;

   procedure cerrar_escritura;
   begin
      escribiendo:=false;
      if empty(lector) then resume(escritor)
      (* desbloquear a un escritor si no hay lectores esperando.*)
      else resume(lector)
      (* sino desbloquear a un lector. *)
   end;
begin
   nl:=0;
   escribiendo:=false;
end;
```

Se puede observar que la concurrencia en los lectores la asegura el *resume(lector)* al final del procedimiento *abrirLectura*, también se garantiza el acceso exclusivo para los escritores ya que ningún lector ni escritor pueden acceder al recurso si *escribiendo=true*.

Prioridad en la escritura

Recordar que la diferencia de esta política con la anterior radica en el momento en que haya escritores bloqueados en espera por acceder al recurso, los lectores deben ser bloqueados hasta que los escritores puedan ser atendidos. Para poder implementar un monitor que resuelva este problema se requieren las mismas variables que en el caso anterior.

```
monitor L_E;
var
   nl:integer;
   escribiendo:boolean;
   lector,escritor:condition;
   export abrir_lectura,cerrar_lectura,abrir_escritura,cerrar_escritura;

   procedure abrir_lectura;
   begin
      if (escribiendo) or not empty(escritor) then delay(lector);
      (* bloquearse si un proceso escritor está accediendo
                                                        o bloqueado. *)
      nl:=nl+1;
      resume(lector); (* desbloquear al resto de lectores. *)
   end;

   procedure cerrar_lectura;
   begin
      nl:=nl-1;
      if (nl=0) then resume(escritor)
      (* desbloquear a un escritor si no quedan lectores *)
   end;

   procedure abrir_escritura;
   begin
      if (nl<>0) or escribiendo then delay(escritor);
      (* bloquearse si el recurso está ocupado. *)
      escribiendo:=true
   end;

   procedure cerrar_escritura;
   begin
      escribiendo:=false;
      if not empty(lector) then resume(lector)
      (* si hay lectores bloqueados despertar al primero. *)
      else resume(escritor)
      (* sino desbloquear a un escritor. *)
   end;
```

El problema de la comida de los filósofos:
Para solucionar el problema vamos a considerar que los filósofos sólo pueden tomar los dos palillos, y por lo tanto comer, siempre que los dos filósofos que tenga a su lado no estén comiendo, con lo que los dos palillos están libres. Para solucionar el problema necesitaremos los siguientes elementos: una array estado para indicar los siguientes estados en los que puede estar un filósofo: *pensando, comiendo y hambriento* (de esta forma, un filósofo sólo puede estar comiendo si sus dos vecinos no lo están); un array de variables de condición dormir, en la que se bloquearán los filósofos que no puedan acceder a los palillos, y por último necesitaremos dos procedimientos *tomarPalillos* y *soltarPalillos*.

```
monitor comida_filosofos;
   const N=5;
   var estado:array[0..N-1] of (pensando,comiendo,hambriento);
      dormir:array[0..N-1] of condition;
      i:integer
   export coger_palillos,soltar_palillos;

   procedure coger_palillos(i:integer);
   begin
      estado[i]:=hambriento; (* Avisa de que quiere comer.     *)
      test(i);               (* Comprueba si puede comer.      *)
                             (* Si no puede comer se bloquea. *)
      if estado[i]<>comiendo then delay(dormir[i])
   end;

   procedure soltar_palillos(i:integer);
   begin
      estado[i]:=pensando; (* Deja de comer e intenta desbloquear       *)
                           (* a los filósofos que tiene a ambos lados.*)
      test((i+4) mod N);
      test((i+1) mod N)
   end;

   procedure test(k:integer);
   begin
      (* Para poder comer, ninguno de los fil. de los lados *)
      (* puede estar comiendo.                              *)
      if (estado[(k+N-1) mod N]<>comiendo) and
         (estado[(k+1) mod N]<>comiendo) and(estado[lx]=hambriento)then
      begin (* Si puede comer desbloquea al correspondiente filósofo *)
         estado[k]:=comiendo;
         resume(dormir[k])
      end
   end;
begin
   for i:=0 to N-1 do
      estado[i]:=pensando;
end.
```

La clave está en el procedimiento *test*. El mismo permite que un filósofo cambie su estado a *comiendo* sólo en el caso de que esté *hambriento* (esperando para tomar los palillos) y sus dos vecinos no estén comiendo.
Para que los filósofos interacciones a través de este monitor:

```
process type filosofo(i:integer);
begin
   repeat
      piensa;
      comida_filosofos.coger_palillos(i);
      come;
      comida_filosofos.soltar_palillos(i)
   forever
end;
```