TEMA 4

Monitores

Monitores

1. Introducción
2. Definición de Monitor
3. Condición de sincronización en monitores
4. Resolución de problemas usando monitores
5. Semántica de la operación **`signal`**
6. Monitores en Java

1. Introducción

- Hemos visto que los **semáforos** permiten solucionar tanto el problema de la exclusión mutua como el de la condición de sincronización.
- No obstante, presentan algunos **inconvenientes** que las hacen difíciles de usar en problemas complejos.
- Usan **variables globales**, lo que impide un diseño modular de la aplicación.
- **No diferencian** entre las variables usadas para **sincronización** o **exclusión mutua**. Además están dispersos por el código y sus errores no son detectables en tiempo de compilación.

1. Introducción

- Para solucionar estos problemas introducimos una **nueva herramienta**, que permite un **control estructurado** de la exclusión mutua y un mecanismo de control de la sincronización igual de **versátil** que los semáforos.
- Además, permite **encapsular datos**, protegiendo el uso de indebido de las variables usadas.

1. Introducción

- En 1974 C. A. R. Hoare describe el funcionamiento de una nueva herramienta para el control de la concurrencia, el **monitor**.
- Un monitor es un mecanismo de **abstracción de datos** que nos permite representar de forma abstracta un recurso compartido mediante variables que definen su estado.
- El acceso a dichas variables sólo es posible mediante el conjunto de funciones exportadas por el monitor al exterior.

1. Introducción

- Los programas que usan monitores tienen **dos tipos de procesos**
  - Procesos pasivos: son aquellos que implementan el monitor y esperan a que usen las operaciones que exportan. Los datos solo pueden ser manipulados por ellos.
  - Procesos activos: son los que realizan las llamadas a los procedimientos exportados por el monitor.
- Esto presenta dos ventajas:
  - Los procesos activos no se preocupan de la implementación, sólo del uso de las operaciones.
  - El programador del monitor no debe preocuparse de donde o como se usa el monitor, sólo de que éste funcione.

2. Definición del monitor

- El monitor contiene las **variables** que representan el **estado del recurso** y los **procedimientos** que implementan las **operaciones**.
- El monitor **garantiza la exclusión mutua** de los procedimientos que contiene. Un único proceso puede estar ejecutando código del monitor.
- Un monitor se compone de los siguientes **elementos**:
  - Un conjunto de variables de estado que son “permanentes”.
  - Un código de inicialización de las variables “permanentes”.
  - Un conjunto de procedimientos internos.
  - Una declaración de procedimientos que son exportados.

2. Definición del monitor

```
monitor nombre_monitor;
  var variables_locales;
  export procedimientos_exportados;

  procedure Proc1(parametros);
  var variables locales;
  begin
    {Codigo del procedimiento}
  end;

  procedure Proc2(parametros);
  var variables locales;
  begin
    {Codigo del procedimiento}
  end;

begin
  {Código de inicialización}
end
```

2. Definición del monitor

- Desde el punto de vista de los procesos que usan el monitor, los “procesos activos”, el monitor sólo se ve como un conjunto de acciones que se pueden invocar.
- La forma de invocar los procedimientos del monitor es:

```
nombre_monitor.nombre_procedimiento(parametros)
```

2. Definición del monitor

- El control de la **exclusión mutua** sobre el monitor se basa en una **cola** asociada al monitor. La forma de gestionar la cola se realiza de la siguiente forma:
  - Cuando un proceso activo está ejecutando código monitor y otro proceso intenta ejecutar otro de los procedimientos del monitor (o el mismo), éste queda a la espera en la cola.
  - Cuando un proceso activo abandona el monitor, el primer proceso esperando en la cola obtiene acceso al procedimiento que provocó el bloqueo.
  - Si un proceso activo abandona el monitor y la cola está vacía, el monitor queda libre para ser usado.

2. Definición del monitor

- Ejemplo. Desarrolle un monitor que permita que varios procesos incrementen o consulten el valor de una variable.

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

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

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

begin
  i:=0;
end;
```

3. Condición de sincronización en monitores

- La exclusión mutua está garantizada, pero se necesita un mecanismo para detener la ejecución de un proceso cuando no se cumpla una determinada condición.
- En 1974 Hoare propone las **variables de condición**.
- Las variables condición representan colas FIFO, que están inicialmente vacías.
- Las operaciones que permiten que los procesos se bloqueen o salgan de estas colas son **`await`** y **`signal.`**

3. Condición de sincronización en monitores

- Operación **`await(c)`.** Siendo **`c`** una variable condición hace que el proceso que la ejecute se bloquee al final de la cola asociada a la variable. Para permitir que otros procesos puedan acceder al monitor, antes de bloquearse, el proceso libera la exclusión mutua del monitor, permitiendo que otros entren.
- Operación **`signal(c)`**. Si existe, desbloquea al primer proceso de la cola asociada a **`c`** . **El proceso que realiza la acción `signal` abandona el monitor, que es tomado por el proceso que ha sido desbloqueado.**
  El proceso que ha realizado **`signal`**, tendrá preferencia para entrar al monitor una vez finalice el proceso desbloqueado.

3. Condición de sincronización en monitores

- Operación **empty(c)**. Devuelve cierto o falso según la cola asociada a la variable **c** esté vacía o no.
- Una variable condición se declara dentro de un monitor de la siguiente forma:

```
var
   cond: condition;
   vector_condiciones: array [1..N] of condition;
```

- Para ilustrar el uso de monitores y variables de condición, vamos a ver como implementar un semáforo binario.

3. Condición de sincronización en monitores

```
monitor semaforo_binario;
  var sem: boolean;
      csem: condition;
  export wait, signal;

  procedure wait;
  begin
    if sem then await(csem);
    sem := true;
  end;

  procedure signal;
  begin
    if not empty(csem) then signal(csem);
    else sem:=false;
  end;

begin
  sem:=false;
end;
```

3. Condición de sincronización en monitores

- BARRERAS

```
monitor barrera;
  var esperar: condition; llegan:integer;
  export barrera;

  procedure barrera(n:integer)
  begin
    llegan:=llegan+1;
    if llegan = n then begin
        llegan = 0;
        for i = 1 to n do signal(esperar);
    end
    else await(esperar);
  end;

begin
  sem:=false;
end;
```

4. Solución de problemas con monitores

- El problema del productor-consumidor
- El problema de los lectores y los escritores (Prioridad en lectura)
- El problema de los lectores y los escritores (Prioridad en escritura)
- El problema de los filósofos

5. Semántica de la operación `signal`

- Dado que la operación signal despierta a uno de los procesos que esperan en el monitor, la implementación debe decidir qué hacer tras la operación signal.
- Existen diversas opciones:
  1. **Desbloquear y continuar**. El proceso que es desbloqueado no comienza su ejecución hasta que el que ha realizado el signal no sale del monitor (o se bloquea en una condición).

     La desventaja es que el proceso que desbloquea puede volver a modificar el estado de las variables del monitor, por tanto, el desbloqueado debe volver a comprobar la condición una vez ha sido desbloqueado.

5. Semántica de la operación `signal`

- Existen diversas opciones:

  2. **Retorno Forzado**. Obliga a que el proceso que realiza la operación signal finalice, de tal forma que el proceso *desbloqueador* sale del monitor cediéndolo al proceso desbloqueado.
     De esta forma el proceso desbloqueado no debe volver a comprobar la condición que lo llevó al bloqueo.

  3. **Desbloquear y esperar**. El proceso que hace signal cede el monitor al proceso desbloqueado. El proceso *desbloqueador* deberá competir por la exclusión mutua del monitor.
     No es necesario que el proceso desbloqueado evalúe la condición que lo bloqueó. Por contra, se perjudica al proceso *desbloqueador.*

5. Semántica de la operación `signal`

- Existen diversas opciones:

  4. **Desbloquear y espera urgente**. Evita la injusticia de la situación anterior asociando al monitor una nueva cola de "cortesía". El proceso *desbloqueador* espera en la cola de cortesía que tiene preferencia para la entrada al monitor. Esta es la usada por Pascal-FC, y la que hemos asumido en este tema.

6. Monitores en Java

- Los métodos de sincronización de Java son prácticamente iguales a un monitor.
- Si tenemos un objeto en Java con todos sus métodos sincronizados tenemos un monitor.
- La principal diferencia radica en que la operación **`wait()`** no diferencia por la condición de espera. Es decir, tenemos una única variable condición.
- Esto provoca que al despertar a los hilos bloqueados, todos deban volver a reevaluar su condición, con la perdida de eficiencia que ello conlleva.

6. Monitores en Java

- Para ilustrar la situación veamos el problema del productor-consumidor

```
public synchronized void Acola(Object elemento) {
    while (colallena()) {
        wait();
    }
    datos[tail] = elemento;
    tail = (tail + 1) % capacidad;
    numelementos++;
    notifyAll();
}

public synchronized Object Desacola() {
    while (colavacia()) {
        wait();
    }
    Object valor = datos[head];
    head = (head + 1) % capacidad;
    numelementos--;
    notifyAll();
    return valor;
}
```

6. Monitores en Java

- Una posible solución es sincronizar sobre objetos distintos.

```
public void acola(Object elemento) throws Exception {
    synchronized (lleno) {
        espacios--;
        while (espacios < 0) lleno.wait();
        datos[tail] = elemento;
        tail = (tail + 1) % capacidad;
    }
    synchronized (vacio) {
        numelementos++;
        if (numelementos <= 0) vacio.notify();
    }
}

public Object desacola() throws Exception {
    Object valor;
    synchronized (vacio) {
        numelementos--;
        while (numelementos < 0) vacio.wait();
        valor = datos[head];
        head = (head + 1) % capacidad;
    }
    synchronized (lleno) {
        espacios++;
        if (espacios <= 0) lleno.notify();
        return valor;
    }
}
```

6. Monitores en Java

- Otra opción es usar **Locks** y **Conditions**
- Un **java.util.concurrent.locks.Lock** es un mecanismo de sincronización de hilos semejante a los bloques sincronizados. Un **Lock** es, sin embargo, más flexible y más sofisticado que un bloque sincronizado.
- Dado que **Lock** es una interfaz, es necesario utilizar alguna de sus implementaciones para utilizar un bloqueo, como **ReentrantLock**

```
Lock bloqueo = new ReentrantLock();
bloqueo.lock();
// sección crítica
bloqueo.unlock();
```

6. Monitores en Java

- Las principales diferencias entre un **`Lock`** y un bloque sincronizado son:
  - Un bloque sincronizado no ofrece ninguna garantía acerca de la secuencia en la que se les conceda acceso hilos esperando para entrar en ella. El constructor **`ReentrantLock(boolean fair)`** si lo permite.
  - No se puede pasar ningún parámetro a la entrada de un bloque sincronizado. Por lo tanto, tener un tiempo de espera de tratar de obtener acceso a un bloque sincronizado no es posible. Con **`tryLock()`** y **`tryLock(long timeout, TimeUnit timeUnit)`** si es posible.
  - El bloque sincronizado debe estar completamente contenido dentro de un solo método. Un **`Lock`** puede tener su llamadas para bloquear y desbloquear en métodos separados.

6. Monitores en Java

- Otra opción es usar **`Locks`** y **`Conditions`**

```
class BoundedBuffer {

   final Lock mutex = new ReentrantLock();
   final Condition lleno = mutex.newCondition();
   final Condition vacio = mutex.newCondition();

   public void Acola(Object elemento) throws InterruptedException {
       mutex.lock();
       try {
           while (colallena()) lleno.await();
           datos[tail] = elemento;
           tail = (tail + 1) % capacidad;
           numelementos++;
           vacio.signal();
       } finally {mutex.unlock();}
   }

   public Object Desacola() throws InterruptedException {
       mutex.lock();
       try {
           while (colavacia()) vacio.await();
           Object valor = datos[head];
           head = (head + 1) % capacidad;
           numelementos--;
           lleno.signal();
           return valor;
       } finally {mutex.unlock();}
   }
}
```

6. Monitores en Java

- ¿ Que significa que un bloqueo es **reentrante** ?
- Los bloques sincronizados y los **`ReentrantLock`** en Java son **reentrantes**. Esto quiere decir, que si un hilo entra en un bloque sincronizado (por lo tanto bloquea el objeto del bloque en que se sincroniza el hilo) puede entrar en otros bloques de código sincronizados en el mismo objeto.

```
public class Reentrant{
  public synchronized externo(){
    interno();
  }
  public synchronized interno(){
    // otras acciones
  }
}
```

Implementación de monitores con semáforos

- Para implementar un monitor usando semáforos necesitaremos los siguientes semáforos y variables:
  - Para la exclusión mutua un semáforo inicializado a 1 al que llamaremos **`m_mutex`**. (**`m`** será el nombre del monitor)
  - Para la cola se cortesía, un semáforo inicializado a 0 al que llamaremos **`m_cortesia`**. Asociado a dicho semáforo tendremos un contador inicializado a 0 al que llamaremos **`m_cortesia_count`**.

Implementación de monitores con semáforos

- Con dichas variables, cada procedimiento del monitor debe contener las siguientes instrucciones:

```
procedure nombre_del_procedimiento;
  begin
    wait(m_mutex);
    Cuerpo del procedimiento
    if m_cortesia_count>0 then
      signal(m_cortesia)
    else
      signal(m_mutex)
  end;
```

Implementación de monitores con semáforos

- Para cada variable condición **m_vc** necesitamos un semáforo al que llamaremos **m_vc_sem** y un contador **m_vc_count** para conocer el número de procesos que esperan en la variable condición. Ambos inicializados a 0.
- La operación await **(m_vc)** quedaría así:

```
m_vc_count := m_vc_count + 1;
if m_cortesia_count <> 0 then
  signal(m_cortesia)
else
  signal(m_mutex);
wait(m_vc_sem);
m_vc_count := m_vc_count - 1;
```

Implementación de monitores con semáforos

- La operación signal **(m_vc)** quedaría así:

```
if m_vc_count <>0 then
begin
  m_cortesia_count := m_cortesia_count + 1;
  signal(m_vc_sem);
  wait(m_cortesia);
  m_cortesia_count := m_cortesia_count - 1;
end;
```

- Y la operación **empty(m_vc)** como

```
if m_vc_count = 0 then
  return(true)
else
  return(false);
```