UNIVERSIDAD
DE GRANADA

Sistemas Concurrentes y Distribuidos: Seminario 2. Introducción a los monitores en C++11.

Carlos Ureña / Jose M. Mantas / Pedro Villar / Manuel Noguera

Curso 2022-23 (archivo generado el 28 de octubre de 2022)

Grado en Ingeniería Informática,
Grado en Informática y Matemáticas,
Grado en Informática y Administración de Empresas.
Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Seminario 2. Introducción a los monitores en C++11.

Índice.

Introducción

El objetivo básico de este seminario es introducir las herramientas que usaremos en C++11 para implementar monitores con semántica *Señalar y Espera Urgente*, junto con un ejemplo sencillo. En concreto:

- Hacemos un breve repaso de los monitores con semántica **Señalar y Espera Urgente** (en adelante SU)
- Vemos como el mecanismo de las clases puede usarse para implementar monitores, en general.
- Introducimos la clase **`HoareMonitor`**, que sirve para implementar monitores SU en C++11.
- Vemos cómo se puede adaptar el ejemplo del productor y consumidor para monitores SU con la citada clase.

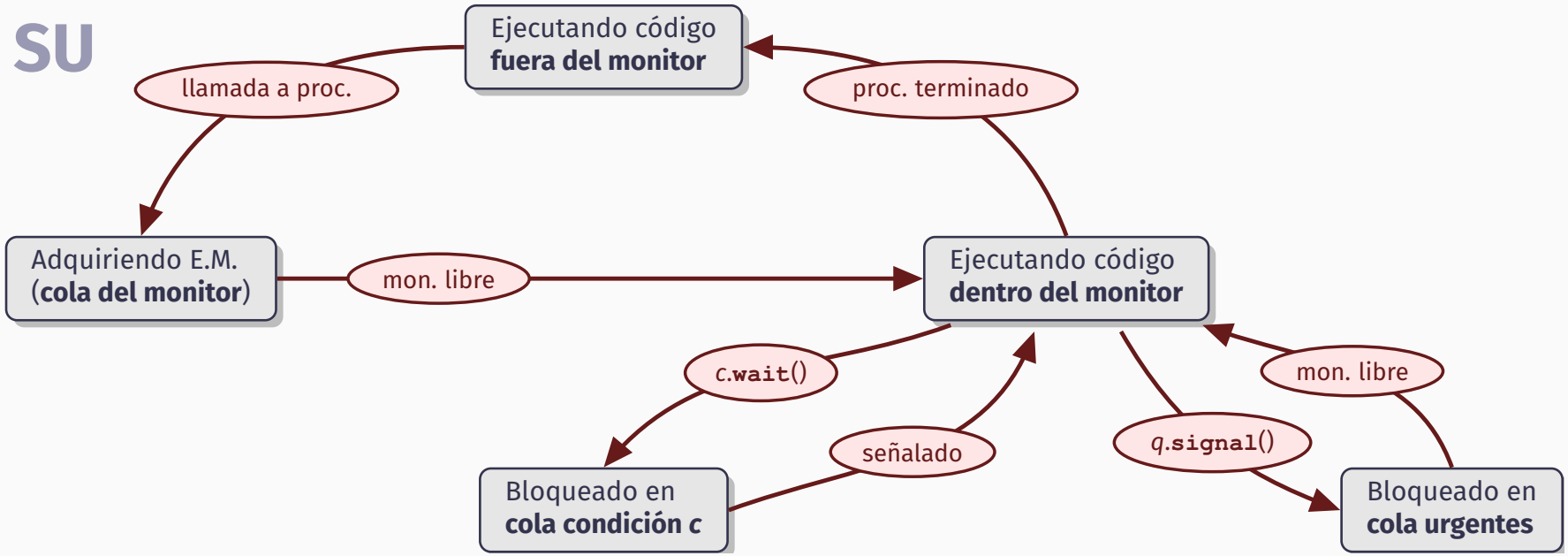
Sección 1.
La semántica SU (monitores Hoare).

La semántica *Señalar y Espera Urgente*. Características

El término **Señalar y Espera Urgente** (SU en adelante) hace referencia a una de las estrategias (semánticas) de implementación de monitores. En ella, si un proceso *señalador* hace **`signal`** sobre un proceso *señalado* (que estaba bloqueado en un **`wait`**):

- El señalador se bloquea justo al ejecutar la operación **`signal`** en una cola de procesos que esperan para acceder al monitor, que llamamos *cola de procesos urgentes*.
- El señalado sale del **`wait`** y entra de forma inmediata en el monitor, ejecutando las sentencias posteriores a **`wait`**. Por tanto, encuentra el monitor exactamente en el mismo estado en que lo dejo el señalador al hacer **`signal`**.
- Los procesos en la cola de procesos urgentes tienen preferencia para acceder al monitor frente a los procesos que esperan en la cola del monitor.

Semántica SU: diagrama de estados de un proceso



- **Señalador:** se bloquea en la **cola de urgentes** hasta readquirir la E.M. y ejecutar código del monitor tras **`signal`**. Para readquirir E.M. tiene más prioridad que los procesos en la cola del monitor.
- **Señalado:** reanuda inmediatamente la ejecución de código del monitor tras **`wait`**.

Ventajas de la semántica SU

La semántica SU presenta diversas ventajas frente a las demás:

- Es posible incluir sentencias después de **`signal`** (el proceso señalador no abandona el monitor tras **`signal`**).
- Las sentencias tras **`signal`** se ejecutan con prioridad frente las hebras de la cola del monitor (el proceso señalador no tiene que esperar en dicha cola otra vez para ejecutar lo que siga a **`signal`**).
- Si un estado del monitor permite desbloquear una hebra con **`signal`**, esa hebra señalada se desbloquea en ese mismo estado y puede avanzar inmediatamente. Se evita que la señalada tenga que volver a comprobar el estado, en bucle.

Sección 2.
Monitores en C++11..

2.1. Los monitores como clases C++

2.2. Encapsulamiento.

Subsección 2.1.
Los monitores como clases C++.

Clases y monitores

El mecanismo de definición de clases de C++ es la forma más natural de implementar el concepto de monitor en este lenguaje:

- Los procedimientos exportados son **métodos publicos**. Son los únicos que se pueden invocar desde fuera.
- Las variables permanentes del monitor se implementan como **variables de instancia no públicas**.
- La inicialización ocurre en los **métodos constructores** de la clase.

Los mecanismos de concurrencia de C++11 permiten:

- Asegurar la exclusión mutua.
- Implementar las variables condición.

Implementación de monitores en C++11

El lenguaje C++11 y la librería estándar no incluye la posibilidad de definir monitores directamente (no hay una clase para eso).

- En C++11, para implementar monitores se pueden usar diversas clases o tipos *nativos* (proporcionados por el lenguaje) como son: *objetos mutex* (tipo **`mutex`**), *guardas de cerrojo* (tipo **`lock_guard`**) y *variables condición* (tipo **`condition_variable`**).
- La implementación puede tener una semántica tipo
  - *Señalar y Continuar* (SC), usando esos tipos nativos directamente.
  - *Señalar y Espera Urgente* (SU, o monitores tipo *Hoare*), de forma indirecta mediante una biblioteca de funciones o clases, las cuales a su vez usan los tipos nativos.

Subsección 2.2.
Encapsulamiento..

Ejemplo sencillo

Partimos de un diseño de un monitor sencillo para garantizar EM en los accesos a una variable permanente entera:

```
monitor MContador1 ;                  { declaración del nombre del monitor}
 var cont : integer;                  { variable permanente (no accesible) }
 export incrementa, leer_valor;       { nombres de métodos que podemos llamar}

 procedure incrementa( );             { procedimiento: incrementa valor actual}
 begin
   cont := cont+1 ;                   {    (añade 1 al valor actual) }
 end;
 function leer_valor() : integer;     { función: devuelve el valor: }
 begin
   return cont ;                      {    el resultado es el valor actual  }
 end;
begin                                 { código de inicialización:  }
  cont := 0 ;                         {    pone la variable a cero }
end
```

En este ejemplo, el código del monitor se ejecuta en exclusión mutua. No hay variables condición.

Clase `Mcontador1` para un monitor: declaración

En el archivo `monitor_em.cpp` vemos la declaración de la clase:

```
class MContador1 // nombre de la clase: MContador1
{
  private:       // elementos privados (usables internamente):
    int cont ;   // variable de instancia (contador)
  public:        // elementos públicos (usables externamente):
    MContador1( int valor_ini ); // declaración del constructor
    void incrementa();           // método que incrementa el valor actual
    int  leer_valor() ;          // método que devuelve el valor actual
} ;
MContador1::MContador1( int valor_ini )
{
  cont = valor_ini ;
}
void MContador1::incrementa()
{
   cont ++ ;
}
int MContador1::leer_valor()
{
  return cont ;
}
```

Clase `Mcontador1` para un monitor: uso

El monitor se usa por dos hebras concurrentes (en **`test_1`**)

```
const int num_incrementos = 10000; // número de incrementos por hebra

void funcion_hebra_M1( MContador1 & monitor ) // recibe referencia al monitor
{
   for( int i = 0 ; i < num_incrementos ; i++ )
      monitor.incrementa();
}
void test_1( ) // (se invoca desde main)
{
  // declarar instancia del monitor (inicialmente, cont==0)
  MContador1 monitor(0) ;
  // lanzar las hebras
  thread hebra1( funcion_hebra_M1, ref(monitor) ),
         hebra2( funcion_hebra_M1, ref(monitor) );
  // esperar que terminen las hebras
  hebra1.join();
  hebra2.join();
  // imprimir el valor esperado y el obtenido:
  cout<< "Valor obtenido: " << monitor.leer_valor() << endl // valor final
      << "Valor esperado: " << 2*num_incrementos << endl ;  // valor o.k.
}
```

Ausencia de exclusión mutua

Al ejecutar el programa anterior, vemos que el valor obtenido no es igual al esperado, ya que:

- Las dos hebras acceden concurrentemente a la variable compartida.
- Cada una puede sobreescribir incrementos realizados por la otra (el valor obtenido es menor que el esperado, generalmente).

Para solucionar el problema, usaremos una biblioteca que incluye una clase base para monitores:

- Los métodos se ejecutan en exclusión mutua (el valor final será el esperado).
- Se pueden declarar variables condición.
- Se implementa la semántica *Señalar y Espera Urgente* (SU).

Sección 3.
La clase `HoareMonitor` para monitores SU.

La clase `HoareMonitor` para monitores SU en C++11

La clase `HoareMonitor`

- Se implementa usando las características nativas de C++11.
- Es una clase base de la cual derivamos (con herencia) otras clases para implementar diseños concretos de monitores.
- Los métodos de la clase se ejecutan en exclusión mutua, de forma prácticamente transparente al programador.
- Se proporciona una clase para las variables condición (`CondVar`), con los métodos para señalar y para esperar.
- Garantiza que siempre habrá orden FIFO en la cola del monitor, la de urgentes, y en las colas de las variables condición.

Uso de la clase `HoareMonitor`

Para definir un monitor tipo Hoare (SU) con esta clase, debemos:

- Hacer **`#include`** del archivo **`scd.h`**
- Definir la clase del monitor como derivada (tipo **`public`**) de **`HoareMonitor`**.
- Declarar los procedimientos exportados y el constructor como públicos (el resto de elementos son privados)
- Al inicio del programa, se debe crear una instancia del monitor con la función **`Create`**.
- Guardamos una referencia o puntero al monitor en una variable de tipo **`MRef`** (por *Monitor **Ref**erence*).
- Hacer que las hebras usen esa referencia para invocar los métodos del monitor.

Subsección 3.1.
Exclusión mutua.

Declaración de la clase MContador2

La clase **MContador2** es similar, pero derivada de **HoareMonitor**

```
class MContador2 : public HoareMonitor
{
  private:       // elementos privados (usables internamente):
    int cont;    // variable de instancia (contador)
  public:        // elementos públicos (usables externamente):
    MContador1( int valor_ini ); // declaración del constructor
    void incrementa();           // método que incrementa el valor actual
    int  leer_valor() ;          // método que devuelve el valor actual
} ;
MContador2::MContador2( int valor_ini )
{
  cont = valor_ini ;
}
void MContador2::incrementa()
{
   cont ++ ;
}
int MContador2::leer_valor()
{
  return cont ;
}
```

Exclusión mutua implícita mediante referencias `MRef`

La exclusión mutua en los métodos está asegurada implícitamente. Si M es una clase derivada de **`HoareMonitor`**, entonces:

- Se debe usar la función **`Create`**<M> para crear una instancia del monitor (de tipo **`MRef`**<M>), debemos de pasar como argumentos los requeridos por el constructor de M:

```
MRef<M> monitor = Create<M>( ... );
```

- Las funciones que ejecutan las hebras reciben como parámetro un objeto `r` de tipo **`MRef`**<M>, y usan el operador `->` para invocar los métodos del monitor, con `r->metodo(...)`.
- **Nunca** debemos usar directamente un objeto `m` de tipo M mediante `m.metodo(...)`, ni tampoco un puntero `p` de tipo M`*`, mediante `p->metodo(...)` (lo métodos no se ejecutarían en EM).

Creación y uso de una instancia de MContador2

Código que usa **MContador2** (en **test_2**):

```
void funcion_hebra_M2( MRef<MContador2>  monitor )
{
   for( int i = 0 ; i < num_incrementos ; i++ )
      monitor->incrementa();
}
void test_2()
{
   // declarar instancia del monitor (inicialmente, cont==0)
   MRef<MContador2> monitor = Create<MContador2>( 0 );
   // lanzar las hebras
   thread hebra1( funcion_hebra_M2, monitor ),
          hebra2( funcion_hebra_M2, monitor );
   // esperar que terminen las hebras
   hebra1.join();
   hebra2.join();
   // imprimir el valor esperado y el obtenido:
     cout<< "Valor obtenido: " << monitor->leer_valor() << endl // valor fi-
nal
         << "Valor esperado: " << 2*num_incrementos << endl ;  // valor o.k.
}
```

Actividad

Realiza estas actividades:

- Compila y ejecuta `monitor_em.cpp`. Verás que ejecuta las funciones `test_1` y`test_2`, cada una de ellas usa cada uno de los dos monitores descritos.
- Verifica que el valor obtenido es distinto del esperado en el caso del monitor sin exclusión mutua. Verifica que en el otro monitor (con EM), el valor obtenido coincide con el esperado.
- Añade código para conocer los tiempos que tardan cada par de hebras desde que se lanzan hasta que terminan, sin contar la creación del monitor ni los `cout`. Justifica los resultados que obtienes.

Subsección 3.2.
Variables condición.

El tipo `CondVar` para variables condición

En las clases derivadas de **`HoareMonitor`** se puede usar el tipo **`CondVar`** para las *variables condición*

- Deben ser variables de instancia privadas de la clase monitor (nunca se pueden usar fuera de los métodos de dicha clase).
- En el constructor del monitor, **cada una** de esas variables debe crearse explícitamente usando la función **`newCondVar`** (deben inicializarse después de que ya exista el objeto monitor, ya que guardan un puntero al mismo).
- Si alguna variable de este tipo se declara pero no se inicializa en el constructor, usarla es un error que aborta el programa.
- En estas colas condición el orden de salida es siempre FIFO (si hay más de una hebra bloqueda, la primera en salir será la primera que entró).

Operaciones sobre variables condición

Las variables condición (de tipo **`CondVar`**) tienen definidas estas operaciones (métodos), que se invocan desde los métodos del monitor:

- **`wait`**`()`: la hebra que invoca espera bloqueada hasta que otra hebra haga **`signal`** sobre la cola.
- **`signal`**`()`: si la cola está vacía, no se hace nada, en otro caso se libera una hebra de las que esperan, y la hebra que invoca se bloquea en cola de urgentes. La hebra liberada **será siempre la primera que llamó** a **`wait`** en esa cola (si hay más de una esperando).
- **`get_nwt`**`()`: devuelve el número de hebras esperando en la cola.
- **`empty`**`()`: devuelve **`true`** si no hay hebras esperando, o **`false`** si hay al menos una.

Sección 4.
Productor/Consumidor únicos en monitores SU.

El productor/consumidor en monitores

En estas sub-sección veremos cómo diseñar e implementar una solución al problema del productor/consumidor (con una hebra en cada rol), usando un monitor SU. Para ello nos basamos en la solución que vimos con semáforos en la práctica 1:

- Las funciones para producir un dato y consumir un dato son exactamente iguales que en la versión de semáforos.
- Diseñamos un monitor SU que encapsula el buffer y define métodos de acceso.
- La función que ejecuta la hebra de productor invocan el método del monitor **`insertar`** para añadir un nuevo valor en el buffer.
- La función que ejecuta la hebra consumidora invoca el método (función) del monitor **`extraer`** para leer un valor del buffer y eliminarlo del mismo.
- El tamaño o capacidad del buffer es un valor constante conocido, que llamamos k.

Hebras productora y consumidora

La hebra productora produce un total de m items, los mismos que consume la consumidora. El valor m es una constante cualquiera, superior al tamaño del buffer ($k < m$).

El pseudo-código de los procesos es como sigue:

```
Monitor ProdConsSU1
  .......
end
```

```
Process Productor ;
  var dato : integer ;
begin
  for i := 1 to m do begin
    dato := ProducirDato();
    ProdConsSU1.insertar( dato );
  end
end
```

```
Process Consumidor ;
  var dato : integer ;
begin
  for i := 1 to m do begin
    dato := ProdConsSU1.extraer();
    ConsumirDato( dato );
  end
end
```

Diseño del monitor: condiciones de espera

Llamamos n al número de entradas del buffer ocupadas con algún valor pendiente de leer. El diseño del monitor debe de asegurar que:

- La hebra productora espera (en **`insertar`**) hasta que hay al menos un hueco para insertar el valor (es decir, espera hasta que $n < k$)
- La hebra consumidora espera (en **`extraer`**) hasta que hay al menos una celda ocupada con un valor pendiente de leer (es decir, hasta que $0 < n$).

Como consecuencia, en el monitor debemos incluir:

- Una variable permanente que contenga el valor de n
- Una cola condición llamada **`libres`**, cuya condición asociada es $n < k$, y donde espera la hebra productora cuando $n = k$.
- Otra, llamada **`ocupadas`**, cuya condición asociada es $0 < n$, y donde espera la hebra consumidora cuando $n = 0$.

Diseño del monitor: accesos al buffer

Los accesos al buffer se pueden hacer usando el esquema LIFO o el FIFO que ya vimos para el diseño con semáforos. Hay que tener en cuenta que:

- Si se usa la opción LIFO, basta con tener la variable permanente **`primera_libre`**, cuyo valor coincide con n. Esta variable sirve tanto para acceder al buffer como para comprobar las condiciones de espera.
- Si se usa la opción FIFO, son necesarias las variables **`primera_libre`** y **`primera_ocupada`** para acceder al buffer. Estas dos variables no permiten saber cuántas celdas hay ocupadas, por tanto necesitamos una variable adicional con el valor de n.

Pseudocódigo del monitor

Por todo lo dicho, el monitor SU (opcion LIFO) puede tener, por tanto, este diseño:

```
Monitor ProdConsSU1

var
  { array con los datos insertados pendientes extraer }
  buffer : array[ 0..k-1 ] of integer ;

  { variables permanentes para acceso y control de ocupación }
  primera_libre   : integer := 0;  { celda de siguiente inserción (== n) }

  { colas condición }
  libres          : Condition ;    { cola de espera hasta n < k (prod.) }
  ocupadas        : Condition ;    { cola de espera hasta n > 0 (cons.) }

  { procedimientos exportados del monitor }
  procedure insertar( dato : integer ) begin ..... end
  function  extraer ( ) : integer      begin .... end

end
```

Operaciones de insertar y extraer

Su diseño es de esta forma:

```
procedure insertar( dato : integer )
begin
  if primera_libre == k then          { si buffer lleno (n == k) }
    libres.wait() ;                   {  esperar que haya celdas libres }
  buffer[primera_libre]:= dato ;      { escribir dato en buffer }
  primera_libre:=  primera_libre+1 ;  { incrementa n (ahora n > 0) }
  ocupadas.signal();                  { despertar consum., si esperaba }
end
```

```
function extraer( ) : integer
   var result ;
begin
  if primera_libre == 0 then          { si buffer vacío (n == 0) }
    ocupadas.wait() ;                 {  esperar que haya celdas ocupadas }
  primera_libre := primera_libre-1 ;  { decrementa n (ahora n < k) }
  result := buffer[primera_libre] ;   { leer del buffer (antes de signal) }
  libres.signal();                    { despertar produ., si esperaba }
  return result ;                     { devolver valor del buffer }
end
```

Implementación en C++11. Actividad.

En el archivo `prodcons1_su.cpp` puedes encontrar una implementación del monitor **`ProdConsSU1`** descrito, con semántica SU (versión LIFO).

- Esta implementación, al finalizar, comprueba que cada valor entero producido está entre 0 y $m-1$, ambos incluidos, y además que es producido y consumido una y solo una sola vez. Si esto no ocurre, se produce un mensaje de error al final.
- Modifica la implementación para usar la opción FIFO. Verifica que funciona correctamente. Describe razonadamente en tu portafolio los cambios que esto supone.

Fin de la presentación.