UNIVERSIDAD
DE GRANADA

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

Carlos Ureña / Jose M. Mantas / Pedro Villar

2018-19

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

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

Introducción

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, implementando la cola del monitor como una variable de instancia (privada) de tipo `mutex` o *cerrojo* (lo llamamos *cerrojo del monitor*)
- Implementar las variables condición usando el tipo `condition_variable`

Estructura del seminario

En este seminario veremos:

- Un ejemplo de un monitor sencillo (sin variables condición), que ofrece **encapsulamiento y exclusión mutua**.
- Monitores basados directamente en las características de C++11, que permite colas condición con semántica **señalar y continuar (SC)**.
- Monitores basados en una biblioteca que permiten usar la semántica **señalar y espera urgente (SU)**

En la práctica 2 veremos ejemplos adicionales de monitores SU.

Sección 1.
Encapsulamiento y exclusión mutua.

1.1. Encapsulamiento mediante clases C++
1.2. Exclusión mutua mediante uso directo de cerrojos
1.3. Exclusión mutua mediante guardas de cerrojo

Subsección 1.1.
Encapsulamiento mediante clases C++.

Ejemplo sencillo

Ejemplo de un monitor que permite accesos concurrentes a una variable permanente de tipo entero.

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

 procedure incrementa( );         { procedimiento: incrementa valor actual}
 begin
   cont := cont+1 ;               {    (añade 1 al valor actual) }
 end;
 function valor() : integer ;     { función: devuelve el valor: }
 begin
   result := 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 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 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 puntero 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, &monitor ),
         hebra2( funcion_hebra_M1, &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.
}
```

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, y por tanto, cada una puede sobreescribir incrementos realizados por la otra (el valor obtenido es menor que el esperado por regla general).

- Para solucionar el problema, usaremos un objeto **`mutex`** asociado a cada instancia del monitor (una variable de instancia privada de tipo **`std::mutex`**)
- A esa variable le llamamos **cerrojo del monitor**.
- Al inicio de cada método público, adquirimos el cerrojo (con **`lock`**), y al final lo liberamos (con **`unlock`**).
- De esta forma, el código del monitor se ejecuta en exclusión mutua.

Subsección 1.2.
Exclusión mutua mediante uso directo de cerrojos.

Declaración de la clase con un cerrojo

Declaramos una clase (**MContador2**) similar a la anterior, pero que ahora incorpora como variable de instancia el cerrojo del monitor (también en el archivo `monitor_em.cpp`):

```
class MContador2 // nombre de la clase: MContador2
{
  private:        // elementos privados (usables internamente):
    int   cont ;         // variable de instancia (contador)
    mutex cerrojo_mon ; // cerrojo del monitor (tipo mutex)

  public:         // elementos públicos (usables externamente):
    MContador2( 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 )  // el constructor es similar
{
  cont = valor_ini ;
}
```

Métodos con exclusión mutua

En los dos métodos públicos, llamamos a **lock** y **unlock**

```
void MContador2::incrementa()
{
  cerrojo_mon.lock();    // accede a exclusión mutua
  cont ++ ;              // incrementar variable
  cerrojo_mon.unlock(); // liberar exclusión mutua
}

int MContador2::leer_valor()
{
  cerrojo_mon.lock();    // accede a exclusión mutua
  int resultado = cont; // copiar cont en el resultado
  cerrojo_mon.unlock(); // liberar a exclusión mutua
  return resultado ;     // devolver el resultado
} ;
```

Es necesario usar la variable local **resultado** en **leer_valor**, ya que si se liberara el cerrojo y luego hicieramos **return cont**, se leería **cont** fuera de la exclusión mutua.

Inconvenientes de lock/unlock

El esquema que hemos visto antes presenta varios inconvenientes:

- En los casos de **`return`** que devuelven un valor, hay que hacer la copia previa en una variable local.
- En métodos con varias sentencias **`return`**, en varios puntos de su código, hay que repetir la llamada a **`unlock`** antes de cada **`return`**.
- Si se produce una excepción durante la ejecución del método (en E.M.), no se liberaría el cerrojo (no se ejecuta **`unlock`**), dejando al monitor en un estado incorrecto que lleva después a interbloqueo (ninguna hebra está ejecutando código pero ninguna puede entrar y todas esperan).

El librería estándar de C++11 incluye un mecanismo para solventar estas dificultades.

Subsección 1.3.
Exclusión mutua mediante guardas de cerrojo.

Guardas de cerrojo

C++11 incluye las denominadas **guardas de cerrojo** (*lock guards*):

- Una guarda de cerrojo es una variable local a cada método exportado del monitor, variable que contiene una referencia al cerrojo del monitor.
- La guarda se declara al inicio del método, y en la creación (en su constructor) se gana la exclusión mutua (se llama a **`lock`**)
- Al ser una variable local, la guarda **se destruye automáticamente al final del método** (independientemente de si el método se acaba por llegar el flujo de control al final, por **`return`** o por una excepción).
- Durante la destrucción de la guarda, se invoca automáticamente **`unlock`**

De esta forma, el programador no tiene que escribir explícitamente sentencias **`unlock`**, que se ejecutan siempre antes de acabar el método.

Uso de guardas de cerrojo

Las guardas son variables del tipo **unique_lock<mutex>**. El constructor de esas variables recibe como parámetro una referencia al cerrojo del monitor. La implementación de `Contador3` es similar a `Contador2` excepto que los métodos públicos quedan así:

```
void MContador3::incrementa()
{
   unique_lock<mutex> guarda( cerrojo_mon ); // gana exclusión mutua
   cont ++ ;    // incrementar variable, después liberar exclusión mutua
}
int MContador3::valor()
{
   unique_lock<mutex> guarda( cerrojo_mon ); // gana exclusión mutua
   return cont ;     // devolver valor de x, después liberar exclusión mutua
} ;
```

Tambien se puede usar **lock_guard** en lugar de **unique_lock**, sería totalmente equivalente en este ejemplo, pero usamos **unique_lock** ya que **unique_lock** es compatible con las variables condición, pero **lock_guard** no.

Actividad

Realiza estas actividades:

- Compila y ejecuta `monitor_em.cpp`. Veras que ejecuta las funciones `test_1`,`test_2` y `test_3`, cada una de ellas usa uno de los tres monitores descritos.
- Verifica que el valor obtenido es distinto del esperado en el caso del monitor sin exclusión mutua. Verifica que los otros dos monitores (con EM), el valor obtenido coincide con el esperado.
- Prueba a quitar el `unlock` de `MContador2::incrementa`. Describe razonadamente que ocurre.

Sección 2.
Monitores nativos tipo *Señalar y Continuar* (SC).

2.1. Monitor de barrera simple
2.2. Monitor de barrera parcial
2.3. Solución del Productor/Consumidor con monitores SC

Introducción

En esta sección veremos como añadir variables condición a las clases que implementan monitores. Junto con las guardas, estas variables condición permiten implementar monitores con semántica **señalar y continuar**.

- El tipo de datos para las variables condición se denomina `variable_condition`.
- Una variable condición contiene una lista (posiblemente vacía) de hebras bloquedas en espera de que sea cierta una determinada condición (una función lógica de las variables permanentes).
- Existen métodos de `variable_condition` para esperar y señalar.
- Las variables de este tipo se inicializan automáticamente en su declaración, es decir, al declarar `variable_condition v`, la variable `v` es inmediatamente usable.

Operaciones sobre variables condición

La clase `variable_condition` tiene los siguientes métodos:

- **`wait`**`(`**`guarda`**`)`, para hacer espera bloqueada, liberando durante la espera el cerrojo del monitor (se pasa como parámetro una guarda del cerrojo del cerrojo del monitor). La hebra que llama a **`wait`** debe poseer el cerrojo al llamar.
- **`notify_one`**`()`, para despertar una hebra de las que esperan (si hay alguna). La hebra señaladora (la que llama a **`notify_one`**) continua la ejecución tras esa llamada, si hay alguna hebra señalada se pone en espera en la cola del monitor para readquirir el cerrojo.
- **`notify_all`**`()`, para despertar todas las hebras que esperan. La hebra señaladora continua la ejecución, y las hebras señaladas esperan en la cola del monitor.

Características de las operaciones

Si hay más de una hebra esperando en una v.c., entonces una llamada a **`notify_one`** despierta a **una cualquiera de ellas**

- El estándar C++11 no especifica ninguna política concreta para seleccionar la hebra que reanuda la ejecución.
- El programador no debe asumir que se está usando ningún criterio concreto, y en particular no puede asumir que se selecciona la más antigua (respecto del orden de entrada en el wait).
- Las implementaciones son libres de usar cualquier criterio equitativo.
- El diseño debe hacerse de forma que todas las hebras esperando en una v.c. deben ser igualmente capaces de realizar el trabajo computacional que sigue a la reanudación.

Subsección 2.1.
Monitor de barrera simple.

Un ejemplo sencillo: barrera simple

Vamos a diseñar una solución para un monitor sencillo llamado `Barrera` con una única cola condición. Los requerimientos son:

- Hay n procesos usando el monitor, los procesos ejecutan un bucle, en cada iteración realizan una actividad (un retraso aleatorio) y luego llaman al único procedimiento exportado del monitor, llamado `cita`.
- Cuando una hebra llama a `cita` en su iteración número k, no terminará la llamada antes de que todas y cada una de las otras hebras hayan hecho también su llamada número k.

Esta especificación implica que todos las hebras se esperan entre ellas tras cada iteración de su bucle, es decir, avanzan de forma síncrona.

Diseño de la solución

En cada iteración, cada hebra debe de esperar a que todas las demás invoquen a la cita, excepto la última en llegar, que debe despertar a las demás. Por tanto, necesitamos una variable permanente del monitor que nos indique cuantas hebras han llegado a la cita en la presente iteración. La llamamos c:

- Debe estar inicializada a 0.
- Cuando una hebra llega a la cita, c debe ser incrementado en una unidad (ya que hay una nueva hebra que ha llegado).
- Tras incrementar, si $c < n$ entonces la hebra debe esperar a las demás que faltan por llegar (faltan $n - c > 0$). Sin embargo, si $c = n$ entonces la hebra es la última y debe despertar a todas las demás.

Por tanto, debemos usar claramente una cola condición cuya condición asociada es $c == n$.

Pseudo-código de la solución

Así que el monitor puede tener esta estructura:

```
monitor MonitorBarrera1 ;

   var c    : integer;          { número de hebras que han llegado a cita }
       n    : integer ;         { núm. total de hebras usando el monitor }
       cola : Condition ;       { cola de hebras esperando c == n }

   procedure cita( )
   begin
      c := c+1 ;                { modificar estado del monitor }
      if c < n then             { comprobar condición, si no es cierta: }
         cola.wait();           {   esperar a que lo sea }
      else begin                { si es cierta (c == n): }
         for i := 0 to n-1 do   {   para cada una de las otras hebras }
            cola.signal();      {     despertar a la hebra que esperaba }
         c := 0 ;               {   reinicializar c para siguiente iter. }
      end
   end
begin                           { inicialización: }
  c := 0 ;                      {    hay 0 hebras en la cita }
  n := ..... ;                  {    número de hebras, arbitrario, (n > 1) }
end
```

Implementación en C++11: declaración de la clase

Ahora veremos como implementar esta solución en C++11, usando guardas de cerrojo para E.M. y variables condición (en el archivo `barrera1_sc.cpp`)

La declaración de la clase tiene esta forma:

```
#include ......                  // includes varios (iostream,thread,mutex,random,...)
#include <condition_variable>    // tipo std::condition_variable
using namespace std ;

class MBarreraSC
{
   private:                                  // variables privadas:
   int                 cont,                 // contador de hebras en cita
                       num_hebras ;          // número total de hebras
   mutex               cerrojo_monitor;      // cerrojo del monitor
   condition_variable  cola ;                // cola de hebras esperando en cita

   public:                                   // metodos públicos:
   MBarreraSC( int p_num_hebras ) ;          // constructor (inicialización)
   void cita( int num_hebra );               // método de cita
} ;
```

Implementación en C++11: constructor y cita

El constructor y el método **cita** se implementan así:

```
MBarreraSC::MBarreraSC( int p_num_hebras )
{
   num_hebras = p_num_hebras ;
   cont       = 0 ;
}
void MBarreraSC::cita( int num_hebra )
{
   unique_lock<mutex> guarda( cerrojo_monitor ); // ganar E.M.

   cont ++ ;
   const int orden = cont ; // copia local del contador (para la traza)
   cout <<"Llega hebra " <<num_hebra <<" (" <<orden <<")." <<endl ;
   if ( cont < num_hebras )
      cola.wait( guarda );  // wait accede al cerrojo del monitor
   else
   {  for( int i = 0 ; i < num_hebras-1 ; i++ )
         cola.notify_one() ;
      cont = 0 ;
   }
   cout <<"      Sale hebra " <<num_hebra <<" (" <<orden <<")." <<endl ;
}
```

Implementación en C++11: código de las hebras

La función de las hebras recibe como parámetros

- un puntero al monitor (**mon_barrera1**)
- el número de hebra (comenzando en 0, hasta **num_hebras-1**)

```
void funcion_hebra( MBarreraSC * monitor, int num_hebra )
{
   while( true )
   {
      const int ms = aleatorio<10,100>();
      this_thread::sleep_for( chrono::milliseconds( ms ) );
      monitor->cita( num_hebra );
   }
}
```

Implementación en C++11: hebra principal

La hebra principal crea el monitor y pone en marcha el resto de hebras:

```
int main()
{
  const int num_hebras = 10 ; // número total de hebras

  // crear el monitor
  MBarreraSC monitor( num_hebras );

  // crear y lanzar hebras
  thread hebra[num_hebras];
  for( unsigned i = 0 ; i < num_hebras ; i++ )
    hebra[i] = thread( funcion_hebra, &monitor, i );

  // esperar a que terminen las hebras (no ocurre nunca)
  for( unsigned i = 0 ; i < num_hebras ; i++ )
    hebra[i].join();
}
```

Características de los monitores SC

Hay que tener en cuenta estos aspectos relativos los monitores SC en general:

- El método **`wait`** de **`condition_variable`** accede al cerrojo del monitor a través de la variable local **`guarda`** (que tiene dentro una referencia a dicho cerrojo).
- El método **`wait`** libera temporalmente el cerrojo del monitor (mientras que la hebra espera). Cuando la hebra es señalada, espera en la cola del monitor hasta que pueda readquirir el cerrojo, entonces continua la ejecución de las sentencias que haya después de la llamada a **`wait`**.
- La hebra señaladora, tras ejecutar **`notify_one`**, continua su ejecución y tiene el cerrojo hasta que sale del monitor (termina el método que está ejecutando).

Traza del programa

La salida del programa (para $n = 6$) en una iteración de todas las hebras es típicamente así:

```
Llega hebra 5 (1).
Llega hebra 1 (2).
Llega hebra 3 (3).
Llega hebra 2 (4).
Llega hebra 0 (5).
Llega hebra 4 (6).
              Sale hebra 4 (6).
              Sale hebra 5 (1).
              Sale hebra 3 (3).
              Sale hebra 2 (4).
              Sale hebra 1 (2).
              Sale hebra 0 (5).
```

Los números entre paréntesis expresan los números de orden de entrada a la cita.

Actividades relativas al monitor Barrera1 (SC)

Describe razondamente en tu portafolio el motivo de cada uno de estos tres hechos:

1. La hebra que entra la última al método **`cita`** (la hebra señaladora) es siempre la primera en salir de dicho método.
2. El orden en el que las hebras señaladas logran entrar de nuevo al monitor no siempre coincide con el orden de salida de **`wait`** (se observa porque los números de orden de entrada no aparecen ordenados a la salida).
3. El constructor de la clase no necesita ejecutarse en exclusión mutua.

Prueba a usar **`notify_all`** en lugar de **`notify_one`**. Describe razonadamente en tu portafolio si se observa o no algún cambio importante en la traza del programa.

Subsección 2.2.
Monitor de barrera parcial.

Requerimientos

Para ilustrar mejor las características de la semántica SC, vamos a analizar una modificción del monitor barrera simple que acabamos de ver:

- En este nuevo monitor, vamos a suponer que la espera de la cita (en cada iteración) no será una espera de todas las hebras en ejecución, sino solo de un subconjunto de ellas (por eso es una barrera *parcial*)
- Suponemos ahora que hay un total de n hebras en ejecución, mientras que en la cita solo se espera a que lleguen m hebras ($m < n$). En las pruebas haremos $m = 10$ y $n = 100$.
- Por tanto, cada conjunto de m hebras que llegan de forma consecutiva a la cita se esperan entre ellas.

Diseño de la solución

Por tanto, la solución con un monitor es similar a la barrera simple que ya hemos visto, solo que substituyendo n (número total de hebras) por m (número de hebras en la cita)

- Ahora creamos una nueva versión del monitor, en la cual el constructor del monitor recibe como parámetro el valor de m, en lugar de n. La cola tiene asociada la condición $c = m$, en lugar de $c = n$.
- En la implementación, sustituimos **`num_hebras`** por **`num_hebras_cita`** en la declaración de la clase monitor y en la implementación de sus métodos.

Como actividad, puedes probar a compilar y ejecutar el monitor **`MBarreraParSC`**, en el archivo `barrera2_sc.cpp`.

Traza de la barrera parcial

Podemos analizar la traza de la barrera parcial, observamos esto:

- Al igual que en la barrera simple, la hebra señaladora (última de cada grupo), es siempre la primera en salir (de ese grupo)
- Igualmente, el orden de salida no coincide necesariamente con el de entrada (en cada grupo)
- Además, ahora las hebras señaladas compiten por el cerrojo con otras hebras que quieren comenzar a ejecutar `cita`, así que las salidas de un grupo no son consecutivas: se alternan con las entradas de otras hebras que no son del grupo.

Propiedades adicionales

Supongamos que añadimos nuevas propiedades a la barrera parcial:

- El orden de salida de **`cita`** debe coincidir con el orden de entrada a la misma.
- Hasta que no termina de salir un grupo completo de m hebras, no se permitirá que ninguna otra hebra fuera del grupo pueda comenzar a ejecutar **`cita`** (las entradas y salidas de un grupo se hacen consecutivas)

Esto se puede conseguir fácilmente, usando el mismo diseño de solución, **pero con un monitor con semántica SU** (lo veremos más adelante)

Subsección 2.3.
Solución del Productor/Consumidor con monitores SC.

El productor/consumidor en monitores

En estas sub-sección veremos como diseñar e implementar una solución al problema del productor/consumidor (con una hebra en cada rol), usando un monitor SC. 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 SC 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 ProdConsSC
  .......
end
```

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

```
Process Consumidor ;
  var dato : integer ;
begin
  for i := 1 to m do begin
    dato := ProdConsSC.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 cuantas celdas hay ocupadas, por tanto necesitamos una variable adicional con el valor de n.

Pseudocódigo del monitor

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

```
Monitor ProdConsSC

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 dato ;
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] ;   { resultado es dato del buffer }
  libres.signal();                    { despertar produ., si esperaba }
end
```

Implementación en C++11. Actividad.

En el archivo `prodcons1_sc.cpp` puedes encontrar una implementación del monitor **`ProdConsSC`** descrito, con semántica SC (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.

Actividad: múltiples productores y consumidores

Copia el archivo `prodcons1_sc.cpp` en `prodcons2_sc.cpp`, y en este nuevo archivo adapta la implementación para permitir múltiples productores y consumidores.

Ten en cuenta estos requerimientos:

- El número de hebras productoras es una constante n_p, (> 0). El número de hebras consumidoras será otra constante n_c (> 0). Ambos valores deben ser divisores del número de items a producir m, y no tienen que ser necesariamente iguales. Se definen en el programa como dos constantes arbitrarias.
- Cada productor produce $p == m/n_p$ items. Cada consumidor consume $c == m/n_c$ items.
- Cada entero entre 0 y $m-1$ es producido una única vez (igual que antes).

Implementación

Para poder cumplir los requisitos anteriores:

- La función `producir_dato` tiene ahora como argumento el número de hebra productora que lo invoca (un valor i entre 0 y $n_p - 1$, ambos incluidos).
- La hebra productora número i produce de forma consecutiva los p números enteros que hay entre el número ip y el número $ip + p - 1$, ambos incluidos.
- Para esto, debemos tener un array compartido con n_p entradas que indique, en cada momento, para cada hebra productora, cuantos items ha producido ya. Este array se consulta y actualiza en `producir_dato`. Debe estar inicializado a 0.

Semántica SC en múltiples prod/cons: implicaciones

La versión de múltiples productores y consumidores mantiene la verificación final de que cada item es consumido una y solo una vez:

- Para que el programa sea correcto, debes de cambiar las sentencias **`if`** en **`extraer`** e **`insertar`**, por bucles **`while`** (manteniendo la condición).
- Comprueba que esto es realmente así, es decir, observa que con el uso de **`if`** se produce un error en las verificaciones que el programa hace, pero con **`while`** se ejecuta correctamente.
- Describe razonadamente en tu portafolio a que se debe que (con semántica SC), la versión para múltiples productores y consumidores deba usar **`while`**, mientras que la versión para un único productor y consumidor puede usar simplemente **`if`**.

Sección 3.
Monitores tipo *Señalar y Espera Urgente* (SU) .

Monitores *Señalar y Espera Urgente* (*SU*)

En los monitores con semántica **Señalar y Espera Urgente** (**SU**), cuando una hebra señaladora hace **`signal`** en una cola donde hay una hebra señalada esperando:

- La hebra señalada adquiere el cerrojo del monitor, y continua ejecutando las sentencias que siguen al **`wait`**.
- La hebra señaladora pasa a esperar a una cola especial, llamada **cola de urgentes**.

Cuando una hebra libera el cerrojo del monitor (bien por entrar en **`wait`**, bien por salir de un procedimiento del monitor):

- Si hay hebras esperando en la cola de urgentes, la que antes entró se libera y continua ejecutando código tras el signal.
- Si no hay hebras en la cola de urgentes, pero hay en la cola del monitor, una de ellas puede acceder al mismo.

Monitores SU: propiedades

Los monitores SU suelen permitir diseños más simples en muchos casos:

- La hebra señalada no tiene que competir con otras hebras para adquirir el cerrojo del monitor y reanudar la ejecución.
- Cada hebra espera en la cola del monitor como mucho una vez como consecuencia de una llamada a un procedimiento del monitor.
- Está garantizado que la hebra señalada reanuda su ejecución inmediatamente tras **`signal`**, ninguna otra puede acceder.
- Como consecuencia, la hebra señalada tiene garantizado que, al salir de **`wait`**, se cumple la condición que espera.

Todo esto permite en muchos casos diseños más simples. Veremos el ejemplo del monitor de barrera parcial.

Monitores SU en C++11

El lenguaje C++11 y la librería estándar no incluye la posibilidad de definir monitores SU directamente. Pero se pueden usar los cerrojos y las variables condición para construir estos monitores

- Se ha construido una clase base para monitores SU.
- La clase usa variables condición nativas de C++11 para implementar la cola del monitor, la cola de urgentes, y las variables condición definidas por los usuarios (junto con un cerrojo para gestión y acceso a esas colas)
- La adquisición y liberación de la E.M. se hace de forma transparente al programador.

La implementación de esta clase se basa en la de los semáforos, según se describe aquí:

Clases para monitores SU

Para construir monitores SU hay que:

- Hacer **`#include`** del archivo **`HoareMonitor.`**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)
- Declarar las variables condición como variables de instancia de tipo **`CondVar`**
- En el constructor, inicializar cada variable condición llamando a **`newCondVar`** e inicializar cada variable permanente al valor adecuado.
- En **`main`**, crear una instancia del monitor.

Veremos el ejemplo de la barrera parcial usando un monitor SU.

Operaciones sobre variables condición

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

- Método **`wait()`** (sin argumentos): la hebra que invoca espera bloqueada hasta que otra hebra haga **`signal`** sobre la cola.
- Método **`signal()`**: se libera la hebra que más tiempo lleva en la cola y la hebra que invoca se bloquea en cola de urgentes (si la cola está vacía, no se hace nada).
- Método **`get_nwt()`**: devuelve el número de hebras esperando en la cola.
- Método **`empty()`**: devuelve **`true`** si no hay hebras esperando, o **`false`** si hay al menos una.

Respecto a la creación de las variables condición:

- Se usa el método **`newCondVar`** (en el constructor del monitor), para crear una nueva variable condición.

Subsección 3.1.
Monitor de barrera parcial con semántica SU.

Declaración de la clase

Veremos el ejemplo del monitor de barrera parcial (en el archivo `barrera2_su.cpp`): partimos del diseño usado para SC y lo trasladamos al monitor SU. La clase se declara usando la clase base **`HoareMonitor`** y la clase **`CondVar`**:

```
#include ..... // iostream, random, etc...
#include "HoareMonitor.hpp"

using namespace std ;
using namespace HM ;   // namespace para monitores SU (monitores Hoare)

class MBarreraParSU : public HoareMonitor  // clase derivada (pública)
{
   private:
   int     cont,              // contador de hebras en cita
           num_hebras_cita ;  // número total de hebras en cita
   CondVar cola ;             // cola de hebras esperando en cita

   public:
   MBarreraParSU( int p_num_hebras_cita ); // constructor
   void cita( int num_hebra );              // método de cita
} ;
```

Constructor y métodos

El constructor usa **newCondVar** para inicializar **cola**:

```
MBarreraParSU::MBarreraParSU( int p_num_hebras_cita )
{
   num_hebras_cita = p_num_hebras_cita ; // total de hebras en cita (> 1)
   cont            = 0 ;                 // hebras actualmente en cita
   cola            = newCondVar();       // cola de espera
}
```

El método **cita** usa los métodos **wait** y **signal** de **CondVar**:

```
void MBarreraParSU::cita( int num_hebra )
{
   cont ++ ;                          // una hebra más ha llegado a la cita
   const int orden = cont ;           // guarda numero de orden de llegada
   if ( cont < num_hebras_cita )      // si no han llegado todas la hebras:
      cola.wait();                    //     espera bloqueado en la cola
   else                               // si ya han llegado todas las demás:
   {  for( int i = 0 ; i < num_hebras_cita-1 ; i++ ) // para cada una:
         cola.signal() ;                             //     reanudar hebra
      cont = 0 ;                                     // ini. contador
   }
}
```

Función que ejecutan las hebras

La función que ejecutan las hebras recibe como parámetro una referencia a la instancia del monitor que van a usar (se usa el tipo **MRef**: encapsula un puntero o referencia al monitor).

Esta referencia permite gestionar automáticamente la exclusión mutua del monitor (el código de **cita** no menciona el cerrojo del monitor)

```
void funcion_hebra( MRef<MBarreraParSU> monitor, int num_hebra )
{
 while( true )
 { const int ms = aleatorio<0,30>();                    // duración aleatoria
   this_thread::sleep_for( chrono::milliseconds(ms) );// espera bloqueada
   monitor->cita( num_hebra );                          // invocar cita con ->
 }
}
```

Hebra principal

La hebra principal crea una instancia del monitor, y usa un objeto de tipo **MRef** que referencia a dicha instancia (y que se pasa a las hebras como parámetro)

```
int main()
{
 const int num_hebras      = 100, // número total de hebras
           num_hebras_cita = 10 ; // número de hebras en cita

 // crear monitor ('monitor' es una referencia al mismo, de tipo MRef<...>)
 MRef<MBarreraParSU> monitor = Create<MBarreraParSU>( num_hebras_cita );

 // crear y lanzar todas las hebras (se les pasa ref. a monitor)
 thread hebra[num_hebras];
 for( unsigned i = 0 ; i < num_hebras ; i++ )
   hebra[i] = thread( funcion_hebra, monitor, i );

 // esperar a que terminen las hebras (no pasa nunca)
 for( unsigned i = 0 ; i < num_hebras ; i++ )
   hebra[i].join();
}
```

Compilar programas con monitores SU

Para compilar en la línea de órdenes programas con monitores SU, se hace igual que antes, solo que ahora añadimos la unidad de compilación `HoareMonitor.cpp`. Para compilar el ejemplo de la barrera parcial (archivo `barrera2_su.cpp`) escribimos (en una sola línea de órdenes)

```
g++ -std=c++11 -pthread -o barrera2_su_exe barrera2_su.cpp
                                    HoareMonitor.cpp Semaphore.cpp
```

Esta orden genera el archivo ejecutable `barrera2_su_exe`.

En la misma carpeta del programa es necesario tener disponibles los archivos:

- `HoareMonitor.h`
- `HoareMonitor.cpp`
- `Semaphore.h`
- `Semaphore.cpp`

Traza de la barrera parcial con semántica SU

Si ejecutamos `barrera2_su_exe`, ahora vemos una salida como esta:

```
Llega hebra 88 ( 1).
Llega hebra 49 ( 2).
Llega hebra 31 ( 3).
Llega hebra 52 ( 4).
Llega hebra 32 ( 5).
Llega hebra 94 ( 6).
Llega hebra 35 ( 7).
Llega hebra 86 ( 8).
Llega hebra 25 ( 9).
Llega hebra 66 (10).
              Sale hebra 88 ( 1).
              Sale hebra 49 ( 2).
              Sale hebra 31 ( 3).
              Sale hebra 52 ( 4).
              Sale hebra 32 ( 5).
              Sale hebra 94 ( 6).
              Sale hebra 35 ( 7).
              Sale hebra 86 ( 8).
              Sale hebra 25 ( 9).
              Sale hebra 66 (10).
```

Propiedades de la barrera parcial con semántica SU

De la traza se deduce lo siguiente:

- El orden de salida de la cita coincide siempre con el orden de entrada
- Hasta que todas las hebras de un grupo han salido de la cita, ninguna otra hebra que no sea del grupo logra entrar.

Describe razondamente en tu portafolio a que se debe que ahora, con la semántica SU, se cumplan las dos propiedades descritas.

Subsección 3.2.
Productor/Consumidor con semántica SU.

Actividad

A modo de actividad, puedes partir de tu implementación de la solución al problema del productor/consumidor (con múltiples productores/consumidores y con semántica SC), y construir una solución equivalente con semántica SU:

- Adapta la versión SC para SU, usando como referencia el código de la barrera parcial SU.
- Implementa la solución LIFO y la FIFO.
- Comprueba que la verificación final es correcta en los dos casos.
- Verifica si en el caso de la semántica SU es también necesario poner las operaciones **`wait`** dentro de un bucle **`while`**, o bien podemos sustituir dichos bucles por sentencias **`if`**.
- Describe razonadamente en tu portafolio a que se debe el resultado que has obtenido en el punto anterior.

Fin de la presentación.