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

Introducción

Vamos a usar **clases**, los **métodos son públicos**, las **variables de instancia son privadas** y la inicialización se hace en los **constructores**.

Encapsulamiento y exclusión mutua

Encapsulamiento mediante clases C++

Declaración de la clase Monitor:

```
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;
}
```

NO hay exclusión mutua.

Exclusión mutua mediante uso directo de cerrojos

Usaremos un objeto `mutex` asociado a cada instancia del monitor y la llamamos **cerrojo del monitor**. Al inicio de cada método público adquirimos el cerrojo (`lock`) y al final lo liberamos (`unlock`)

```
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){
    cont = valor_ini ;
}
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
} ;
```

Hay que usar la variable `resultado` ya que si se liberara el cerrojo y luego hiciéramos `return cont`, se leería `cont` fuera de exclusión mutua.

Sin embargo, si se produce una **excepción** durante la ejecución de un método, no se libraría el cerrojo.

Exclusión mutua mediante guardas de cerrojo

Para solventar dicho problema, usaremos **guardas de cerrojo (lock guards)** que son del tipo `unique_lock<mutex>`.

- Es una variable local a cada método exportado del monitor.
- Contiene una referencia al cerrojo del monitor.
- Se declara al inicio del método (`lock`).
- Se destruye automáticamente al final del método (`unlock`).

NO hay que escribir `unlock`.

```
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
};
```

Monitores nativos tipo Señalar y Continuar (SC)

Veremos cómo añadir variables condición `variable_condition` para implementar monitores con semántica **señalar y continuar**. Estas variables tienen una **lista de hebras bloqueadas en espera**. Hay

métodos para **esperar y señalar**. Se inicializan automáticamente en su declaración. Métodos:

- `wait(guarda)`: para hacer espera bloqueada y libera durante la espera el cerrojo.
- `notify_one()`: despierta a una hebra que espera (va a la cola del monitor) y continua su ejecución.
- `notify_all()`: despierta a todas las hebras que esperan (van a la cola del monitor) y continua su ejecución.

Monitor de barrera simple

Implementaremos un monitor `Barrera` con una cola condición.

Debe haber `n` procesos usando el monitor que ejecutan un bucle y en cada iteración realizan una actividad, luego llaman a `cita`. Por último, en su iteración `k`, no terminará la llamada antes de que todas hayan hecho su llamada número `k`. Por lo que las hebras avanzan de forma síncrona.

```
#include ..... // includes varios (iostream,thread,mutex,random,...)
#include <condition_variable> // tipo std::condition_variable
using namespace std ;
class MBarreraSC{
    private:
        int cont,
            num_hebras;
        mutex cerrojo_monitor;
        condition_variable cola;
    public:
        MBarreraSC( int p_num_hebras ) ;
        void cita( int num_hebra );
};

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 ;
```

```
}

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 );
    }
}

int main(){
    const int num_hebras = 10 ; // número total de hebras
    MBarreraSC monitor( num_hebras ); // crear el monitor
    thread hebra[num_hebras]; // crear y lanzar 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();
}
```

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).

Monitor de barrera parcial

Vamos a hacer algunos cambios:

- La espera de la cita en cada iteración será una espera de un subconjunto de las hebras.
- Hay `n`hebras en ejecución y en `cita` solo se espera que lleguen `m`.
- El constructor recibe como parámetro `m`.
- Sustituimos `num_hebras` por `num_hebras_cita`.

Por lo que las salidas de un grupo no son consecutivas: se alternan con las entradas de otras hebras que no son del grupo.

Solución del Productor/Consumidor con monitores SC

Ahora queremos que el orden de salida de `cita` debe coincidir con el de entrada. Hasta que no terminen las `m` hebras no entrará otro grupo de `m` hebras.

```
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
```

- `n`: número de entradas del buffer ocupadas.
- Hebra productora espera (`insertar`) hasta que hay algún hueco (`n < k`).
- Hebra consumidora espera (`extraer`) hasta que hay alguna celda ocupada (`0 < n`).
- Hay que incluir
    - Una variable permanente que contenga `n`.
    - Una cola condición `libres` con `n < k` donde la espera de la productora es si `n = k`.
    - Una cola condición `ocupadas` con `0 < n` donde la espera de la consumidora es si `n = 0`.
- Para acceder al buffer:
    - LIFO: hay que tener la variable permanente `primera_libre` cuyo valor coincide con `n`.
    - FIFO: hay que tener `primera_libre` y `primera_ocupada`.

```
//LIFO
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
```

```
        if primera_libre == k then
        libres.wait() ;
        buffer[primera_libre]:= dato ;
        primera_libre:= primera_libre+1 ;
        ocupadas.signal();
    end

    function extraer( ) : integer
        var dato ;
    begin
        if primera_libre == 0 then
        ocupadas.wait() ;
        primera_libre := primera_libre-1 ;
        result := buffer[primera_libre] ;
        libres.signal();
    end
end
```

Ver ejemplo `prodcons1_sc.cpp` y modificarlo:

- 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).
- 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.
- 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.

En la versión de múltiples productores y consumidores:

- 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`.

Monitores tipo Señalar y Espera Urgente (SU)

Cuando una hebra señaladora hace `signal` en una cola donde hay una hebra esperando:

- La hebra señalada adquiere el cerrojo del monitor y continua ejecutando las sentencias que siguen al `wait`.
- La hebra señaladora espera en una cola especial **cola de urgentes**.

Cuando una hebra libera el cerrojo del monitor:

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

Propiedades del monitor SU:

- 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.

Para construir monitores SU hay que:

- `#include HoareMonitor.h`
- Definir la clase del monitor como derivada de `HoareMonitor`.
- Procedimientos y constructor públicos.
- Variables de condición `CondVar`.
- En el constructor, inicializar cada variable condición con `newCondVar` e inicializar.
- En `main` crear una instancia del monitor.

Operaciones sobre variables condición:

- `wait()`: la hebra que invoca espera hasta que otra haga `signal`.
- `signal()`: se libera la hebra que lleva más tiempo y la hebra que invoca se bloquea en cola de urgentes.
- `get_nwt()`: devuelve el número de hebras esperando en la cola.
- `empty()`: `true` si no hay hebras esperando.

Monitor de barrera parcial con semántica SU

```
#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,
        num_hebras_cita ;
    CondVar cola ;

    public:
    MBarreraParSU( int p_num_hebras_cita );
    void cita( int num_hebra );
};
```

```
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
}

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
    }
}

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 ->
    }
}

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();
}
```

Para compilar:

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

Productor/Consumidor con semántica SU

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.