DOCUMENTACIÓN EJERCICIOS

SISTEMAS CONCURRENTES Y DISTRIBUIDOS

David Gómez Hernández 2ºB Grupo B1

FUMADORES CON SEMÁNTICA SU

```
// ----------------------------------------------------------------------------
//David Gómez Hernández 2ºB
// Sistemas concurrentes y Distribuidos.
//
// archivo: fumadores_su.cpp
// Ejemplo de un monitor 'Fumadores' parcial, con semántica SU
//
// ----------------------------------------------------------------------------


#include <iostream>
#include <iomanip>
#include <random>
#include "HoareMonitor.h"

using namespace std ;
using namespace HM ;

Incluimos la variable global que indica el número de fumadores

const int num_fumadores=7;

Incluimos un mutex para que nos proteja las salidas por consola
mutex Mutex;

La función aleatorio es un generador de número entre 2 enteros pasados por parámetro
template< int min, int max > int aleatorio()
{
  static default_random_engine generador( (random_device())() );
  static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
  return distribucion_uniforme( generador );
}

// *****************************************************************************
// clase para monitor Estanco,  semántica SU
```

Definimos la clase Estanco con un monitor

```
class Estanco : public HoareMonitor
{
  private:
```

Definimos la variable de clase que irá cambiando en función de los fumadores

```
  int     num_ingrediente;
```

Definimos las variables condicionales:

- mostrador_vacio indica si el mostrador está vacío o no
- ingrediente_disponible es un vector de variables condicionales cuyo tamaño varía en funcion del número de fumadores. Cada una de estas condiciones regulan si el ingrediente está disponible o no.

```
  CondVar  mostrador_vacio,
           ingrediente_disponible[num_fumadores];
  public:
```

Definimos el constructor por defecto y las funciones que se explican más abajo.

```
  Estanco() ; // constructor
  //funciones
  void obtenerIngredientes(int ingrediente);
  void ponerIngrediente(int ingrediente);
  void esperarRecogidaIngrediente();
} ;
// ---------------------------------------------------------------------------
```

Constructor por copia. Iguala el num_ingrediente a -1 de tal forma que no interfiera con ningún fumador. Luego inicializa todas las variables condicionales de monitor.

```
Estanco::Estanco()
{
  num_ingrediente = -1;
  mostrador_vacio=newCondVar();
  for(unsigned i=0;i<num_fumadores;i++)
          ingrediente_disponible[i]=newCondVar();
}
// ---------------------------------------------------------------------------
```

La función obtenerIngredientes compara si el ingrediente pasado por parámetro es igual al que supuestamente ese fumador tiene que coger. Si no es así se pone a esperar. Luego iguala num_ingrediente a -1, dando a indicar que el ingrediente ya ha sido retirado por el fumador correspondiente. Finalmente realiza una salida por consola indicando lo anterior.

```
void Estanco::obtenerIngredientes(int ingrediente)
{
  if(num_ingrediente!=ingrediente)
          ingrediente_disponible[ingrediente].wait();

  num_ingrediente=-1;
```

```
  Mutex.lock();
  cout<<"        Ingrediente "<<ingrediente<<" retirado"<<endl;
  Mutex.unlock();
  mostrador_vacio.signal();
}
```

La función Fumar recibe como parámetro el fumador que va a fumar. La hebra realiza una espera en función de un tiempo aleatorio simulando la acción de fumar.

```
void Fumar(int num_fumador)
{
  chrono::milliseconds fumar(aleatorio<200,600>());
  Mutex.lock();
  cout<<"        Fumador "<<num_fumador<<" comienza a fumar"<<endl;
  Mutex.unlock();
  this_thread::sleep_for(fumar);
  Mutex.lock();
  cout<<"        Fumador "<<num_fumador<<" termina de fumar"<<endl;
  Mutex.unlock();
}
```

La función ProducirIngrediente simula la acción de producir un ingrediente mediante un tiempo de espera y posteriormente se lo asigna a un fumador al azar.

```
int ProducirIngrediente()
{
  chrono::milliseconds poner(aleatorio<200,600>());
  this_thread::sleep_for(poner);
  int ingrediente = aleatorio<0,num_fumadores-1>();
  return ingrediente;
}
```

La función ponerIngrediente de la clase Estanco recibe como parámetro el ingrediente producido, simula la acción de ponerlo en el mostrador y realiza una señal al fumador que le corresponde dicho ingrediente.

```
void Estanco::ponerIngrediente(int ingrediente)
{
  Mutex.lock();
  cout<<"Poniendo ingrediente "<<ingrediente<<endl;
  Mutex.unlock();

  ingrediente_disponible[ingrediente].signal();
}
```

La función de la clase Estanco esperarRecogidaIngrediente espera a que el mostrador se quede vació si hay un ingrediente en este ya que un fumador no lo ha recogido, es decir, que num_ingrediente sea distinto de -1.

```
void Estanco::esperarRecogidaIngrediente(){
  if (num_ingrediente!=-1)
        mostrador_vacio.wait();
}
```

```
// -----------------------------------------------------------------------------
```

La funcion_hebra_fumador recibe como parámetros el monitor y el ingrediente que tiene que obtener llamando a la función obtenerIngredientes. Finalmente se fuma dicho ingrediente.

```
void funcion_hebra_fumador( MRef<Estanco> monitor, int ingrediente )
{
  while( true )
  {
        monitor->obtenerIngredientes(ingrediente);
        Fumar(ingrediente);;
  }
}
```

La función_hebra_estanquero recibe como paŕametro el monitor y simula lo que haría el estanquero: produce el ingrediente con la función ProducirIngrediente, dicho ingrediente lo pone con ponerIngrediente y finalmente se va a dormir hasta que el fumador correspondiente recoja el ingrediente.

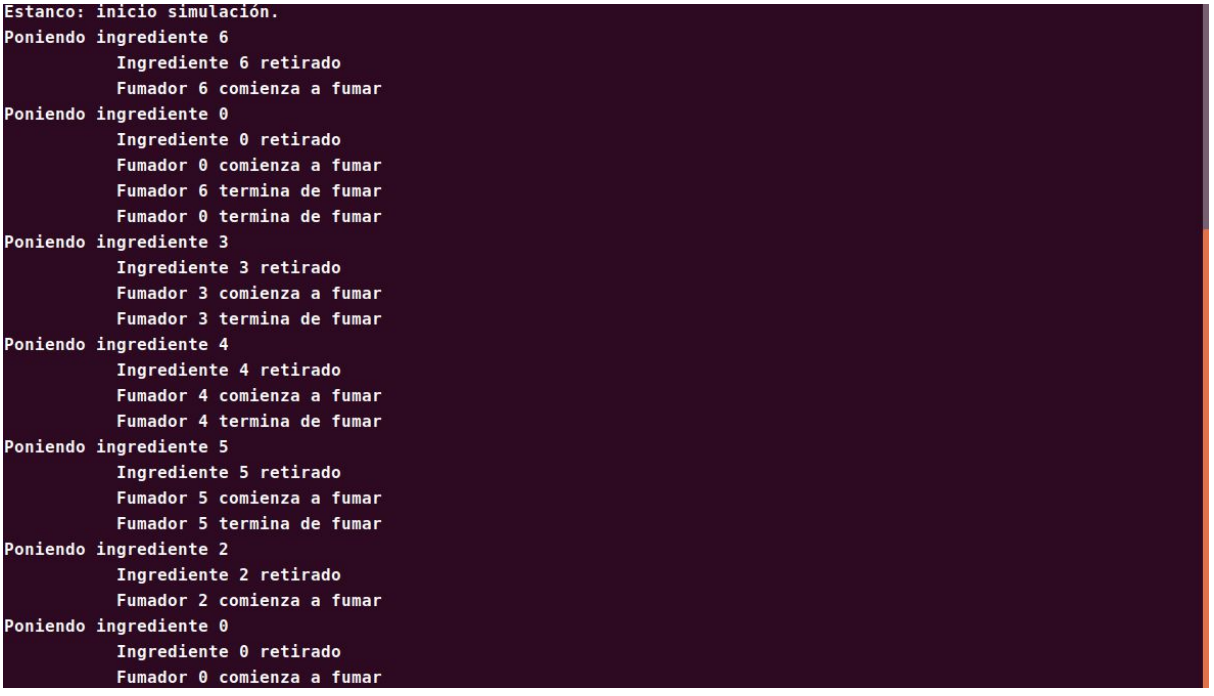
```
void funcion_hebra_estanquero(MRef<Estanco> monitor)
{
  while( true )
  {
        int ingrediente=ProducirIngrediente();
        monitor->ponerIngrediente(ingrediente);
        monitor->esperarRecogidaIngrediente();
  }
}
```

```
// *****************************************************************************
```

```
int main()
{
  cout <<  "Estanco: inicio simulación." << endl ;
```

```
  // crear monitor
  MRef<Estanco> monitor = Create<Estanco>();

  // crear y lanzar hebras
  thread hebra_estanquero(funcion_hebra_estanquero,monitor);;
  thread hebras_fumador[num_fumadores];

  for( unsigned i = 0 ; i < num_fumadores ; i++ )
        hebras_fumador[i] = thread( funcion_hebra_fumador, monitor, i );

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

  hebra_estanquero.join();
}
```

SALIDA FUMADORES



Podemos observar que la salida es correcta ya que ningún fumador se interpone con otro y cada ingrediente es recogido por su respectivo cliente.

BARBERÍA CON SEMÁNTICA SU

```
// ---------------------------------------------------------------------------
//David Gómez Hernández 2ºB
// Sistemas concurrentes y Distribuidos.
//
// archivo: barberia_su.cpp
// Ejemplo de un monitor Barberia, con semántica SU
//
// ---------------------------------------------------------------------------


#include <iostream>
#include <iomanip>
#include <random>
#include "HoareMonitor.h"

using namespace std ;
using namespace HM ;
```

Incluimos un mutex para que nos proteja las salidas por consola

```
mutex Mutex;
```

La función aleatorio es un generador de número entre 2 enteros pasados por parámetro

```
template< int min, int max > int aleatorio()
{
  static default_random_engine generador( (random_device())() );
  static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
  return distribucion_uniforme( generador );
}

// *****************************************************************************
// clase para monitor Barbería, semántica SU

class Barberia : public HoareMonitor
{
   private:
```

Definimos la variable de clase que irá cambiando en función del cliente

```
   int     cliente;
```

Definimos las variables condicionales:

- sala_espera que comprueba si hay clientes esperando.
- ocupado que comprueba si el barbero puede irse a dormir.
- cortando que comprueba si un cliente se está cortando el pelo.

```
   CondVar  sala_espera,
```

```
        ocupado,
        cortando;

  public:
```

Definimos el constructor por defecto y las funciones que se explican más abajo.

```
  Barberia() ; // constructor
  //funciones
  void CortarPelo(int cliente);
  void siguienteCliente();
  void finCliente();
} ;
// ----------------------------------------------------------------------------
```

Constructor por copia. Iguala el cliente a 0 de tal forma que no interfiera con ningún cliente. Luego inicializa todas las variables condicionales de monitor.

```
Barberia::Barberia()
{
  cliente = 0;
  sala_espera=newCondVar();
  ocupado=newCondVar();
  cortando=newCondVar();
}
// ----------------------------------------------------------------------------
```

La función de la clase Barbería CortarPelo recibe por parámetro el cliente que se va a cortar el pelo. Primero comprueba que la sala de espera esté vacía, ya que si lo está el barbero se va a dormir y si no lo está realiza una señal. Finalmente simula mediante una salida por consola que el cliente se está cortando y realiza una espera mientras esto ocurre.

```
void Barberia::CortarPelo(int cliente)
{
  Mutex.lock();
  cout<<"El cliente "<<cliente<<" llega a la barberia"<<endl;
  Mutex.unlock();

  if(sala_espera.empty()){
        ocupado.wait();
  }
  else{
        sala_espera.signal();
  }

  Mutex.lock();
  cout<<"El cliente "<<cliente<<" se está cortando el pelo"<<endl;
```

```
  Mutex.unlock();

  cortando.wait();
}
```

La función EsperarFueraBarbería simula la salida de un cliente de la barbería y realiza una espera.

```
void EsperarFueraBarberia(int cliente)
{
  chrono::milliseconds espera(aleatorio<200,600>());
  Mutex.lock();
  cout<<"El cliente "<<cliente<<" sale de la barberia"<<endl;
  Mutex.unlock();
  this_thread::sleep_for(espera);
}
```

La función de la clase Barbería siguienteCliente comprueba si el barbero está ocupado. Si lo está realiza una espera en la sala de espera y si no lo está despierta al barbero.

```
void Barberia::siguienteCliente()
{
  if(ocupado.empty()){
        sala_espera.wait();
  }
  else{
        ocupado.signal();
  }
}
```

La función CortarPeloCliente simula un corte de pelo al cliente mediante una espera y una salida en pantalla.

```
void CortarPeloCliente()
{
  chrono::milliseconds cortando(aleatorio<30,150>());
  Mutex.lock();
  cout<<"El barbero corta el pelo al cliente"<<endl;
  Mutex.unlock();
  this_thread::sleep_for(cortando);
}
```

La función de la clase Barbería finCliente avisa de que ya ha terminado de cortar el pelo al cliente.

```
void Barberia::finCliente()
{
```

```
   cortando.signal();
}

// ----------------------------------------------------------------------------
```

La funcion_hebra_cliente recibe por parámetro y el número del cliente. Este se corta el pelo y espera fuera de la barbería hasta que se lo vaya a cortar otra vez.

```
void funcion_hebra_cliente( MRef<Barberia> monitor, int num_cliente )
{
   while( true )
   {
         monitor->CortarPelo(num_cliente);
         EsperarFueraBarberia(num_cliente);
   }
}
```

La funcion_hebra_barbero recibe por parámetro el monitor y simula lo que haría el barbero: Primero llama a un cliente, le corta el pelo y lo “echa” de la barbería.

```
void funcion_hebra_barbero(MRef<Barberia> monitor)
{
   while( true )
   {
         monitor->siguienteCliente();
         CortarPeloCliente();
         monitor->finCliente();
   }
}

// ****************************************************************************

int main()
{
   cout <<  "Barbería: inicio simulación." << endl ;

   // declarar el número total de hebras
   const int num_clientes=7;

   // crear monitor
   MRef<Barberia> monitor = Create<Barberia>();

   // crear y lanzar hebras
   thread hebra_barbero(funcion_hebra_barbero,monitor);;
   thread hebras_cliente[num_clientes];
```

```
   for( unsigned i = 0 ; i < num_clientes ; i++ )
         hebras_cliente[i] = thread( funcion_hebra_cliente, monitor, i );

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

   hebra_barbero.join();
}
```

SALIDA BARBERÍA

```
Barberia: inicio simulación.
El cliente 0 llega a la barberia
El cliente 0 se está cortando el pelo
El barbero corta el pelo al cliente
El cliente 3 llega a la barberia
El cliente 1 llega a la barberia
El cliente 2 llega a la barberia
El cliente 5 llega a la barberia
El cliente 4 llega a la barberia
El cliente 6 llega a la barberia
El cliente 0 sale de la barberia
El cliente 3 se está cortando el pelo
El barbero corta el pelo al cliente
El cliente 3 sale de la barberia
El cliente 1 se está cortando el pelo
El barbero corta el pelo al cliente
El cliente 1 sale de la barberia
El cliente 2 se está cortando el pelo
El barbero corta el pelo al cliente
El cliente 2 sale de la barberia
El cliente 5 se está cortando el pelo
El barbero corta el pelo al cliente
El cliente 0 llega a la barberia
El cliente 5 sale de la barberia
```

Como vemos la salida es satisfactoria ya que el barbero corta el pelo a los clientes según el orden de llegada y los clientes no se superponen a la hora de cortarse el pelo.