Barbero_P2.pdf

mma66

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

```
#include <iostream>
#include <iomanip>
#include <cassert>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include "HoareMonitor.h"

using namespace std;
using namespace HM;

const int num_clientes = 3; // número de clientes
mutex mtx;

//**************************************************************************
// plantilla de función para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
//--------------------------------------------------------------------------

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 Barrera, version 2,  semántica SU
class MBarreraSU : public HoareMonitor{

   private:
      int clientes_sala;      // 0sala vacía (1, 2, 3)clientes
      CondVar silla;    //cola silla
      CondVar colabarbero;    //cola barbero
      CondVar colaespera;   //cola clientes
   public:
      MBarreraSU(); //constructor
      void siguienteCliente();         //metodo coger cliente
      void pelarCliente(int i);        //metodo pelar cliente
      void finCliente();         //metodo fin cliente
};
//----------------------------------------------------------------------------
MBarreraSU::MBarreraSU(){
      //Inicio sala de espera con el total de clientes
      clientes_sala = 0;
      //Creacion cola barbero
      colabarbero=newCondVar();
      //Creacion cola clientes
      colaespera=newCondVar();
      //Creacion cola silla
      silla=newCondVar();
}

//----------------------------------------------------------------------------
void MBarreraSU::siguienteCliente(){
      if(clientes_sala == 0)
         colabarbero.wait();
      if(clientes_sala > 0 && silla.empty())
         colaespera.signal();
}
//----------------------------------------------------------------------------
```

```
void MBarreraSU::pelarCliente(int i){
      clientes_sala++;
      if(!silla.empty() || clientes_sala > 1){
         mtx.lock();
         cout << "Cliente " << i << " pasa a sala de espera." << endl;
         mtx.unlock();

         colaespera.wait();
      }
      if(silla.empty()){
         mtx.lock();
         cout << "Cliente " << i << " pasa a pelarse." << endl;
         mtx.unlock();
         colabarbero.signal();
         silla.wait();
      }
}
//----------------------------------------------------------------------------
void MBarreraSU::finCliente(){
      mtx.lock();
      cout << "Barbero termina de pelar al cliente. " << endl;
      mtx.unlock();
      clientes_sala--;
      silla.signal();
}

//----------------------------------------------------------------------------
// Función que simula la acción de pelar, como un retardo aleatorio de la hebra

void cortarPeloACliente( ){
  // calcular milisegundos aleatorios de duración de la acción de cortar el
pelo)
  chrono::milliseconds duracion_pelar(aleatorio<20,200>());

  // espera bloqueada un tiempo igual a 'duracion_cortar' milisegundos
this_thread::sleep_for(duracion_pelar);
 cout << "Barbero pela y tarda: (" << duracion_pelar.count() << "
milisegundos)." << endl;
}
//----------------------------------------------------------------------------
// Función que simula la acción de esperar fuera de la barbería, como un retardo
aleatorio de la hebra

void esperarFueraBarberia( int i ){


  // calcular milisegundos aleatorios de duración de la acción de cortar el
pelo)
  chrono::milliseconds duracion_esperar(aleatorio<20,200>());

    mtx.lock();
    cout << "Cliente " << i << " :" << " espera fuera (" <<
duracion_esperar.count() << " milisegundos)" << endl;
    mtx.unlock();

  // espera bloqueada un tiempo igual a 'duracion_cortar' milisegundos
this_thread::sleep_for(duracion_esperar);

 cout << "Cliente " << i << " : termina de esperar fuera." << endl;
}
//----------------------------------------------------------------------------
// función que ejecuta la hebra del barbero

void funcion_hebra_barbero(MRef<MBarreraSU> monitor){
```

```
      while(true){
         monitor->siguienteCliente();
         cortarPeloACliente();
         monitor->finCliente();
      }

}
//----------------------------------------------------------------------
// función que ejecuta la hebra del cliente
void  funcion_hebra_cliente(MRef<MBarreraSU> monitor, int num_cliente)
{

      while(true){
            mtx.lock();
            cout << "Cliente " << num_cliente << " entra a la barbería" << endl;
            mtx.unlock();

            monitor->pelarCliente(num_cliente);

            mtx.lock();
            cout << "Cliente " << num_cliente << " sale de la barbería" << endl;
            mtx.unlock();
            esperarFueraBarberia(num_cliente);
      }
}

//----------------------------------------------------------------------

int main()
{

   MRef<MBarreraSU> monitor = Create<MBarreraSU>();

      // crear y lanzar todas las hebras (se les pasa ref. a monitor)
      thread clientes[num_clientes];
      for( unsigned i = 0 ; i < num_clientes ; i++ )
         clientes[i] = thread( funcion_hebra_cliente, monitor, i );
      thread barbero = thread(funcion_hebra_barbero, monitor);
      // esperar a que terminen las hebras (no pasa nunca)
      for( unsigned i = 0 ; i < num_clientes ; i++ )
         clientes[i].join();
      barbero.join();
}
```