LosCocos
www.wuolah.com/student/LosCocos

Practica-2-RESUELTA.pdf

Practica 2 RESUELTA

2º Sistemas Concurrentes y Distribuidos

Grado en Ingeniería Informática

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

```
#include <iostream>
#include <cassert>
#include <thread>
#include <mutex>
#include <random>
#include <chrono>
#include "Semaphore.h"
#include "HoareMonitor.h"

using namespace std;
using namespace SEM;
using namespace HM;

const int num_fumadores=3;
mutex mtx;


//-------------------------------------------------------------------------
// plantilla de funciÃ³n para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilaciÃ³n)
//-------------------------------------------------------------------------

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 );
}

//-------------------------------------------------------------------------
// FunciÃ³n que simula la acciÃ³n de fumar, como un retardo aleatoria de la hebra

void fumar( int num_fumador )
{

   // calcular milisegundos aleatorios de duraciÃ³n de la acciÃ³n de fumar)

   chrono::milliseconds duracion_fumar( aleatorio<20,200>() );

   // informa de que comienza a fumar
    mtx.lock();
    cout << "Fumador " << num_fumador << "  :"
          << " empieza a fumar (" << duracion_fumar.count() << " milisegundos)" << endl;
    mtx.unlock();
   // espera bloqueada un tiempo igual a ''duracion_fumar' milisegundos
   this_thread::sleep_for( duracion_fumar );

   // informa de que ha terminado de fumar
    mtx.lock();
    cout << "Fumador " << num_fumador << "   : termina de fumar, comienza espera de ingrediente."
<< endl;
    mtx.unlock();
}

int ProducirIngrediente()
{
       mtx.lock();
       cout << "Estanquero empieza a producir " << endl;
       mtx.unlock();
```

```
        chrono::milliseconds tiempo( aleatorio<20,200>() );
        this_thread::sleep_for( tiempo );

        int ingr=aleatorio<0,num_fumadores-1>();

        mtx.lock();
        cout << "Estanquero termina de producir ingrediente" << endl;
        mtx.unlock();
        return ingr;
}

//-------------------------------------------------------------------------
//Monitor

class Estanco: public HoareMonitor{
        private:
                CondVar fumador[num_fumadores], mostrador;
                int ingrediente;
        public:
                Estanco(){
                        ingrediente=-1;
                        mostrador= newCondVar();
                        for(int i=0; i<num_fumadores; i++)
                                fumador[i]= newCondVar();
                }

                void obtenerIngredienteFumador(int num_fumador){
                        if(num_fumador==this->ingrediente){
                                mostrador.signal();
                                fumador[num_fumador].wait();
                                ingrediente=-1;
                        }
                        else
                                fumador[num_fumador].wait();
                }

                void ponerIngredienteMostrador(int ingr){
                        this->ingrediente=ingr;
                        fumador[this->ingrediente].signal();
                }

                void esperarIngredienteMostrador(){
                        if(ingrediente!=-1)
                                mostrador.wait();
                }
};

void funcion_hebra_estanquero( MRef<Estanco> monitor){

        while(true){
                const int ingr= ProducirIngrediente();
                monitor->ponerIngredienteMostrador(ingr);
                monitor->esperarIngredienteMostrador();

        }
}

void funcion_hebra_fumador( MRef<Estanco> monitor, int num_fumador){
        while(true){
                monitor->obtenerIngredienteFumador(num_fumador);
                fumar(num_fumador);
```

```
        }
}

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

int main(){
        MRef<Estanco> monitorEstanco = Create<Estanco>();
        thread hebra_fumador[num_fumadores];
        thread hebra_estanquero(funcion_hebra_estanquero, monitorEstanco);

        for(int i=0; i<num_fumadores; i++)
                hebra_fumador[i]= thread(funcion_hebra_fumador, monitorEstanco, i);
        hebra_estanquero.join();

        for(int i=0; i<num_fumadores; i++)
                hebra_fumador[i].join();
}
```

```cpp
#include <iostream>
#include <cassert>
#include <thread>
#include <mutex>
#include <random>
#include <chrono>
#include "Semaphore.h"
#include "HoareMonitor.h"

using namespace std;
using namespace SEM;
using namespace HM;

const int num_clientes=5;

//**************************************************************************
// plantilla de funciÃ³n para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilaciÃ³n)
//--------------------------------------------------------------------------

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 );
}

//----------------------------------------------------------------------
//Monitor

class Barberia: public HoareMonitor{
	private:
		CondVar barbero, cliente, salaDeEspera;
	public:
		Barberia(){
			barbero= newCondVar();
			cliente= newCondVar();
			salaDeEspera= newCondVar();
		}

		void cortarPelo(int cl){
			if(!barbero.empty() && cliente.empty()){
				cout << "Cliente "<< cl << ": inicia su corte de pelo" << endl;
				barbero.signal();
				salaDeEspera.wait();
			}
			else{
				cliente.wait();
				salaDeEspera.wait();
			}
		}

		void siguienteCliente(){
			if(!cliente.empty())
				cliente.signal();
			else
				barbero.wait();
		}

		void finCliente(){
```

```
                        salaDeEspera.signal();
                }
};


void EsperarFueraBarberia(int cl){
        chrono::milliseconds tiempo(aleatorio<20,200>());
        cout << "Cliente  " << cl << ": espera fuera durante " << tiempo.count() <<
"milisegundos" << endl;
        this_thread::sleep_for(tiempo);
        cout << "Cliente " << cl << ": acaba la espera fuera" << endl;
}

void cortarPeloACliente(){
        chrono::milliseconds tiempo(aleatorio<20,200>());
        cout << "Barbero: corta durante " << tiempo.count() << "milisegundos" << endl;
        this_thread::sleep_for(tiempo);
        cout << "Barbero: termina de cortar" << endl;
}

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

void funcion_hebra_cliente(MRef<Barberia> monitor, int cl){
        while(true){
                EsperarFueraBarberia(cl);
                monitor->cortarPelo(cl);
        }
}

int main(){
        MRef<Barberia> monitorBarberia= Create<Barberia>();
        thread hebra_cliente[num_clientes];
        thread hebra_barbero(funcion_hebra_barbero, monitorBarberia);

        for(int i=0; i< num_clientes; i++)
                hebra_cliente[i]=thread(funcion_hebra_cliente, monitorBarberia, i);

        hebra_barbero.join();
        for(int i=0; i<num_clientes; i++)
                hebra_cliente[i].join();
}
```