LosCocos
www.wuolah.com/student/LosCocos

Practica-1-RESUELTA.pdf

Practica 1 RESUELTA

2º Sistemas Concurrentes y Distribuidos

Grado en Ingeniería Informática

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

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

using namespace std ;
using namespace SEM ;


const int num_fumadores=3;
Semaphore mostrador=1; //1 si mostrador vacÃo, 0 si ocupado
std::vector<Semaphore> ingredientes; //1 si ingrediente i estÃ¡ disponible, 0 si no.
Inicializados a 0 para solo poder entrar en la funcion estanquero
mutex mtx; //Para proteger la pantalla;

//**********************************************************************
// 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 ejecuta la hebra del estanquero

void funcion_hebra_estanquero(  )
{
        int num_fumador;
        while(true){

                sem_wait(mostrador);
                num_fumador=aleatorio<0,num_fumadores-1>();
                mtx.lock();
                cout << "Se ha puesto el ingrediente nÃºmero: " << num_fumador << endl;
                mtx.unlock();

                sem_signal(ingredientes[num_fumador]);
        }
}

//----------------------------------------------------------------------
// 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();
}

//----------------------------------------------------------------------
// funciÃ³n que ejecuta la hebra del fumador
void  funcion_hebra_fumador( int num_fumador )
{
   while( true )
   {
               sem_wait(ingredientes[num_fumador]);

               mtx.lock();
               cout << "Retirando ingrediente nÃºmero: " << num_fumador << endl;
               mtx.unlock();

               sem_signal(mostrador);
               fumar(num_fumador);
   }
}

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

int main()
{
        for(int i=0; i<num_fumadores; i++){
               ingredientes.push_back(0);
        }

   thread hebra_estanquero(funcion_hebra_estanquero);
   thread hebra_fumador[num_fumadores];


   for(unsigned long i=0; i<num_fumadores; i++){
       hebra_fumador[i]=thread(funcion_hebra_fumador,i);
   }
   hebra_estanquero.join();

   for(unsigned long i=0; i<num_fumadores; i++){
       hebra_fumador[i].join();
   }

}
```

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

using namespace std ;
using namespace SEM ;

//**********************************************************************
// variables compartidas

const int num_items = 40 ,   // nÃºmero de items
              tam_vec   = 10 ;   // tamaÃ±o del buffer
unsigned  cont_prod[num_items] = {0}, // contadores de verificaciÃ³n: producidos
          cont_cons[num_items] = {0}; // contadores de verificaciÃ³n: consumidos

//SemÃ¡foros compartidos

Semaphore puede_escribir = 1;
Semaphore puede_leer = 0;
mutex mtx;

int entrada=0, salida=0; //Salida aumenta al escribir y Entrada aumenta al leer
int buffer[tam_vec];


//**********************************************************************
// 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 );
}

//**********************************************************************
// funciones comunes a las dos soluciones (fifo y lifo)
//----------------------------------------------------------------------

int producir_dato()
{
   static int contador = 0 ;
   this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));

   mtx.lock();
   cout << "producido: " << contador << endl << flush ;
   mtx.unlock();

   cont_prod[contador] ++ ;
   return contador++ ;
}
//----------------------------------------------------------------------

void consumir_dato( unsigned dato )
{
```

```
   assert( dato < num_items );
   cont_cons[dato] ++ ;
   this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));

   mtx.lock();
   cout << "                  consumido: " << dato << endl ;
   mtx.unlock();
}


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

void test_contadores()
{
   bool ok = true ;
   cout << "comprobando contadores ...." ;
   for( unsigned i = 0 ; i < num_items ; i++ )
   {  if ( cont_prod[i] != 1 )
      {  cout << "error: valor " << i << " producido " << cont_prod[i] << " veces." << endl
;
         ok = false ;
      }
      if ( cont_cons[i] != 1 )
      {  cout << "error: valor " << i << " consumido " << cont_cons[i] << " veces" << endl
;
         ok = false ;
      }
   }
   if (ok)
      cout << endl << flush << "soluciÃ³n (aparentemente) correcta." << endl << flush ;
}

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

void  funcion_hebra_productora(  )
{

   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato = producir_dato() ;
      // completar ........
       sem_wait(puede_escribir);

       mtx.lock();
       buffer[entrada]=dato;
       entrada=(entrada+1)%tam_vec;
       mtx.unlock();

       sem_signal(puede_leer);
   }
}

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

void funcion_hebra_consumidora(  )
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato ;
      // completar ......
       sem_wait(puede_leer);
```

```
        mtx.lock();
        dato=buffer[salida];
        salida=(salida+1)%tam_vec;
        mtx.unlock();

        sem_signal(puede_escribir);
        consumir_dato( dato ) ;

    }
}
//----------------------------------------------------------------------

int main()
{
   cout << "--------------------------------------------------------" << endl
        << "Problema de los productores-consumidores (soluciÃ³n FIFO)." << endl
        << "--------------------------------------------------------" << endl
        << flush ;

   thread hebra_productora ( funcion_hebra_productora ),
          hebra_consumidora( funcion_hebra_consumidora );

   hebra_productora.join() ;
   hebra_consumidora.join() ;

   test_contadores();
}
```

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

using namespace std ;
using namespace SEM ;

//**********************************************************************
// variables compartidas

const int num_items = 40 ,   // nÃºmero de items
          tam_vec   = 10 ;   // tamaÃ±o del buffer
unsigned  cont_prod[num_items] = {0}, // contadores de verificaciÃ³n: producidos
          cont_cons[num_items] = {0}; // contadores de verificaciÃ³n: consumidos

//SemÃ¡foros compartidos

Semaphore puede_escribir = 10; //CorresponderÃ-a al nÃºmero de huecos libres
Semaphore puede_leer = 0;
mutex mtx; //Mutex para proteger la pantalla

int cima=0; //Esta variable se incrementa al escribir y se decrementa al leer
int buffer[tam_vec];


//**********************************************************************
// 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 );
}

//**********************************************************************
// funciones comunes a las dos soluciones (fifo y lifo)
//----------------------------------------------------------------------

int producir_dato()
{
   static int contador = 0 ;
   this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));

   mtx.lock();
   cout << "producido: " << contador << endl << flush ;
   mtx.unlock();

   cont_prod[contador] ++ ;
   return contador++ ;
}
//----------------------------------------------------------------------

void consumir_dato( unsigned dato )
{
```

```
   assert( dato < num_items );
   cont_cons[dato] ++ ;
   this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));

   mtx.lock();
   cout << "                  consumido: " << dato << endl ;
   mtx.unlock();
}


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

void test_contadores()
{
   bool ok = true ;
   cout << "comprobando contadores ...." ;
   for( unsigned i = 0 ; i < num_items ; i++ )
   {  if ( cont_prod[i] != 1 )
      {  cout << "error: valor " << i << " producido " << cont_prod[i] << " veces." << endl
;
         ok = false ;
      }
      if ( cont_cons[i] != 1 )
      {  cout << "error: valor " << i << " consumido " << cont_cons[i] << " veces" << endl
;
         ok = false ;
      }
   }
   if (ok)
      cout << endl << flush << "soluciÃ³n (aparentemente) correcta." << endl << flush ;
}

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

void  funcion_hebra_productora(  )
{

   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato = producir_dato() ;
      // completar ........
       sem_wait(puede_escribir);

       mtx.lock();
       buffer[cima]=dato;
       cima++;
       mtx.unlock();

       sem_signal(puede_leer);
   }
}

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

void funcion_hebra_consumidora(  )
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato ;
      // completar ......
       sem_wait(puede_leer);
```

```
        mtx.lock();
        dato=buffer[cima-1];
        cima--;
        mtx.unlock();

        sem_signal(puede_escribir);
         consumir_dato( dato ) ;
    }
}
//----------------------------------------------------------------------

int main()
{
   cout << "--------------------------------------------------------" << endl
        << "Problema de los productores-consumidores (soluciÃ³n LIFO)." << endl
        << "--------------------------------------------------------" << endl
        << flush ;

   thread hebra_productora ( funcion_hebra_productora ),
          hebra_consumidora( funcion_hebra_consumidora );

   hebra_productora.join() ;
   hebra_consumidora.join() ;

   test_contadores();
}
```