SISTEMAS CONCURRENTES Y DISTRIBUIDOS

PRACTICA 2

Jose Antonio Padial Molina
josepadial@correo.ugr.es

Contenido

PRODUCTOR-CONSUMIDOR SC LIFO

```
mutex m;      // mutex de escritura en pantalla
const int num_productor = 2, num_consumidor = 4;
```

Solo declaramos un variable mutex para controlar las salidas por pantalla. Indicamos con dos variables globales el número de productores y el de consumidores.

```
class ProdConsdScLifo{
  private:
    static const int num_celdas_total = 10;
    int buffer[num_celdas_total], primera_libre ;
    mutex cerrojo_monitor ;
    condition_variable ocupadas, libres ;
  public:
    ProdConsdScLifo();
    int leer(int num);
    void escribir(int valor, int num);
};
```

La clase del monitor va a contener en el privado al ser de tipo SC un cerrojo y dos variables de tipo conition_variable que las usaremos en las funciones. En la parte publica declaramos en constructor por defecto y las variables para leer valores y escribir valores.

```
ProdConsdScLifo::ProdConsdScLifo(){
  primera_libre = 0;
}
```

Lo que hace el constructor por defecto es poner la variable que indica en el vector el número de elementos usados a 0.

```
int ProdConsdScLifo::leer(int num){
  unique_lock<mutex> guarda( cerrojo_monitor );

  cout << "Llega hebra consumidora " <<setw(2) << num << endl ;

  if ( primera_libre == 0 )
      ocupadas.wait( guarda );

  assert( 0 < primera_libre  );
  primera_libre-- ;
  const int valor = buffer[primera_libre] ;

  libres.notify_one();

  cout << "              Sale hebra consumidora " <<setw(2) << num << endl;

  return valor ;
}
```

Creamos un cerrojo llamado guarda de tipo unique_lock<mutex> que le pasamos por parámetro en cerrojos declarado en el private de la clase. Vamos a definir unique_lock, es un objeto que administra un objeto de exclusión mutua con propiedad única en ambos estados:

bloqueado y desbloqueado. Como estamos en LIFO (ultima en entrar primera en salir) devolvemos el valor apuntado por la variable primera_libre menos uno.

```
void ProdConsdScLifo::escribir(int valor, int num){
  unique_lock<mutex> guarda( cerrojo_monitor );

  cout << "Llega hebra productora " <<setw(2) << num << endl ;

  if ( primera_libre == num_celdas_total )
     libres.wait( guarda );

  assert( primera_libre < num_celdas_total );

  buffer[primera_libre] = valor ;
  primera_libre++ ;

  ocupadas.notify_one();

  cout << "              Sale hebra productora " <<setw(2) << num << endl;
}
```

La función de escritura sigue el mismo esquema que la función de lectura. Solo que en vez de sacar el valor de primera_libre menos uno, añade un valor en la posición de buffer de primera_libre y esta variable la incrementa en una unidad.

```
void  funcion_hebra_productora( ProdConsdScLifo* monitor , int num)
{
   for( unsigned i = 0 ; i < num_items/num_productor ; i++ )
   {
      int valor = producir_dato() ;
      monitor->escribir( valor , num);
   }
}
```

La función de la hebra productora recibe los parámetros en número de productor y un monitor del tipo de la clase que hemos declarado antes. La cual produce un valor y llama a la función de la clase escribir para guardarlo.

```
void funcion_hebra_consumidora( ProdConsdScLifo* monitor , int num)
{
   for( unsigned i = 0 ; i < num_items/num_consumidor ; i++ )
   {
      int valor = monitor->leer( num );
      consumir_dato( valor ) ;
   }
}
```

La función de la hebra del consumidor sigue la estructura de la función de la hebra productuora.

PRODUCTOR-CONSUMIDOR SC FIFO

En este problema hemos mantenido todo idéntico a PRODUCTOR-CONSUMIDOR SC LIFO . La diferencia que poseen es la gestión de los datos en el buffer. Al ser de tipo FIFO (Primo en llegar primero en salir), solo es modificada la función leer de la clase del monitor.

```
int ProdConsdScFifo::leer(int num){
  unique_lock<mutex> guarda( cerrojo_monitor );

  cout << "Llega hebra consumidora " <<setw(2) << num << endl ;

  if ( primera_libre == 0 )
      ocupadas.wait( guarda );

  assert( 0 < primera_libre  );
  primera_libre-- ;
  const int valor = buffer[0] ;

  for(int i=0; i<primera_libre; i++){
    buffer[i] = buffer[i+1];
  }

  libres.notify_one();

  cout << "              Sale hebra consumidora " <<setw(2) << num << endl;

  return valor ;
}
```

Lo que hacemos es devolver siempre el prime valor del buffer y con un bucle for vamos pasando todos los elementos del buffer una posición atrás.

PRODUCTOR-CONSUMIDOR SU LIFO

Al usar un monitor SU (señala y espera urgente) incluimos su librería y su using namaspace

```
#include "HoareMonitor.h"

using namespace std;
using namespace HM ;
```

```
class ProdConsdSuLifo:public HoareMonitor{
  private:
    static const int num_celdas_total = 10;
    int buffer[num_celdas_total], primera_libre;
    CondVar puede_producir;
    CondVar puede_escribir;
  public:
    ProdConsdSuLifo();
    int leer(int num);
    void escribir(int valor, int num);
};
```

La clase del monitor de tipo SU es diferente de SC, lo primero que hacemos es indicar que hay una herencia simple entre la clase del monitor que estamos declarando y la de HoareMonitor. Donde en sus datos privados nos encontramos dos variables de tipo CondVar las que van a controlar si se puede leer y si se puede escribir. Y en su parte pública nos encontramos el constructor por defecto y las funciones para leer y escribir datos.

```
ProdConsdSuLifo::ProdConsdSuLifo(){
  primera_libre = 0;
  puede_producir = newCondVar();
  puede_escribir = newCondVar();
}
```

Lo que hace el constructor por defecto es poner la variable que indica en el vector el número de elementos usados a 0. Y las variables CondVar las inicializamos un un new.

Ahora las funciones no van a depender un unique_lock para hacer la sección crítica. En los monitores de tipo SU solo nos hace falta hacer signal o wait a las variables del tipo CondVar.

En este caso como estamos en el modo LIFO, gestionamos los datos en el buffer igual que en PRODUCTOR-CONSUMIDOR SC LIFO

```
int ProdConsdSuLifo::leer(int num){
  cout << "Llega hebra consumidora " <<setw(2) << num << endl ;

  if ( primera_libre == 0 )
     puede_escribir.wait( );

  assert( 0 < primera_libre  );
  primera_libre-- ;
  const int valor = buffer[primera_libre] ;

  puede_producir.signal();

  cout << "              Sale hebra consumidora " <<setw(2) << num << endl;

  return valor ;
}
// -----------------------------------------------------------------------------

void ProdConsdSuLifo::escribir(int valor, int num){
  cout << "Llega hebra productora " <<setw(2) << num << endl ;

  if ( primera_libre == num_celdas_total )
     puede_producir.wait();

  assert( primera_libre < num_celdas_total );

  buffer[primera_libre] = valor ;
  primera_libre++ ;

  puede_escribir.signal();

  cout << "              Sale hebra productora " <<setw(2) << num << endl;
}
```

Las hebras del productor y del consumidor son iguales a las ya explicadas en [PRODUCTOR-CONSUMIDOR SC LIFO]

PRODUCTOR-CONSUMIDOR SU FIFO

En este problema hemos mantenido la estructura de los monitores SU explicados en PRODUCTOR-CONSUMIDOR SU LIFO pero hacemos la gestión de datos en el buffer igual que en PRODUCTOR-CONSUMIDOR SC FIFO.

FUMADORES

Vamos a modificar el problema de los fumadores de la practica 1. Ahora va a pasar a tener un monitor de tipo SU. Este problema va a contener una variable global donde vamos a indicar el número de fumadores, donde nosotros lo vamos a dejar a tres fumadores. Igual que los problemas anteriores vamos a tener una función denominada aleatorio de tipo template que nos va a devolver un valor entre dos pasados a la función. La función que nos simula el fumar va a dormir la hebra por un tiempo aleatorio.

```
void fumar( int num_fumador )
{

   // calcular milisegundos aleatorios de duración de la acción de fumar)
   chrono::milliseconds duracion_fumar( aleatorio<20,200>() );

   // informa de que comienza a fumar

    cout << "Fumador " << num_fumador << "  :"
         << " empieza a fumar (" << duracion_fumar.count() << " milisegundos)" << endl;

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

   // informa de que ha terminado de fumar

    cout << "Fumador " << num_fumador << "  : termina de fumar, comienza espera de ingrediente." << endl;

}
```

Declaramos la función producir que lo que nos va a hacer en generar un entero entre 0, 1 y 2.

Al crear la clase del monitor de fumador hay que indicar que tenemos una herencia simple entre clases. Esta clase en parte privada posee dos variables de tipo CondVar, la primera va a ser un vector del tamaño según el número de fumadores que nos va a indicar el sí se puede fumar según el fumador; y la otra es la condición del estanquero. En la parte publica tenemos el constructor por defecto, la función para poner ingrediente, la función para recoger ingredientes y la función para esperar la recogida.

```
class Monitorfumador : public HoareMonitor{
    private:
        int ingrediente_disponible;
        CondVar cond_fumadores[num_fumadores], cond_estanquero;
        bool vacio, lleno;
    public:
        Monitorfumador();
        void PonerIngrediente(int ingrediente);
        void ObtenerIngrediente(int ingrediente);
        void EsperarRecogida();
};
```

En el constructor por defecto inicializamos las variables CondVar a new y la variable de ingrediente disponible a -1.

```
Monitorfumador::Monitorfumador(){
    ingrediente_disponible = -1;
    for(int i=0; i<num_fumadores; i++)
        cond_fumadores[i] = newCondVar();

    cond_estanquero = newCondVar();
}
```

La función poner ingrediente les da ingredientes a los fumadores. Y la función obtener ingredientes hace que el fumador coja un ingrediente. La función de espera de recogida hace esperar al estanquero para que los fumadores recojan los ingredientes.

```
void Monitorfumador::PonerIngrediente(int ingrediente){
    ingrediente_disponible = ingrediente;
    cond_fumadores[ingrediente].signal();
}

void Monitorfumador::ObtenerIngrediente(int ingrediente){
    if(ingrediente_disponible != ingrediente)
        cond_fumadores[ingrediente].wait();

    ingrediente_disponible = -1;
    cond_estanquero.signal();
}

void Monitorfumador::EsperarRecogida(){
    if(ingrediente_disponible != -1)
        cond_estanquero.wait(); // Si hay un ingrediente esperamos a que se retire para poner otro
}
```

La función de la hebra del estanquero solo se le pasa el monitor, lo que hace es producir un ingrediente llama al monitor para poner el ingrediente en el mostrador y espera a que un fumador lo recoja.

La función de la hebra fumador se le pasa el monitor y el número de fumador, la cual obtiene un ingrediente con la función del monitor y usa la función de fumar para simular que fuma.

```
void funcion_hebra_estanquero(MRef<Monitorfumador> monitor){
    while(true){
        int ingrediente = producir();
        cout << "\tEstanquero: " << ingrediente << endl;
        monitor -> PonerIngrediente(ingrediente);
        monitor -> EsperarRecogida();
    }
}

//----------------------------------------------------------------------
// función que ejecuta la hebra del fumador

void funcion_hebra_fumador(MRef<Monitorfumador> monitor, int num_fumador){
    while(true){
        monitor -> ObtenerIngrediente(num_fumador);
        fumar(num_fumador);
    }
}
```

BARBERO DURMIENTE

Realizamos un problema desde 0, en el cual tenemos un barbero y un número de clientes el cual será una variable global en el programa. En este problema vamos a gestionar una barbería con un monitor de tipo SU. Vamos a indicar en este ejemplo que vamos a tener 10 clientes, lo que viene siendo 10 hebras de tipo cliente.

Creamos una función de esperaFueraBarberia donde va a dormir la hebra por un tiempo aleatorio. Y la función cortarPeloCliente donde se va a dormir la hebra un tiempo aleatorio para simular el tiempo que tarda el barbero en cortar el pelo.

```
void esperaFueraBarberia(){
    // calcular milisegundos aleatorios de duración de la espera
   chrono::milliseconds espera( aleatorio<200,2000>() );

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

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

void cortarPeloCliente(){
    // calcular milisegundos aleatorios de duración de la espera
   chrono::milliseconds espera( aleatorio<200,2000>() );

   // espera bloqueada un tiempo igual a ''espera' milisegundos
   this_thread::sleep_for( espera );
}
```

Al crear la clase del monitor de la barbería hay que indicar que tenemos una herencia simple entre clases. Esta clase en parte privada posee tres variables de tipo CondVar, la silla; el barbero; el cliente. En la parte publica tenemos el constructor por defecto, la función para cortar pelo, la función para llamar al siguiente cliente y la función para terminar con un cliente.

```
class Barberia : public HoareMonitor{
    private:
        CondVar barbero, silla, clientes;
    public:
        Barberia();
        void cortarPelo(int cliente);
        void siguienteCliente();
        void finCliente();
};
```

En el constructor por defecto inicializamos las variables CondVar a new.

```
Barberia::Barberia(){
    barbero = newCondVar();
    silla = newCondVar();
    clientes = newCondVar();
}
```

La función para cortar pelo son condicionales anidados. Si no hay cliente sentado tenemos dos opciones sentarnos a cortarnos el pelo en el caso que este el barbero despierto, sino lo despertamos primero. Si hay cliente sentado en la silla esperamos a que se vaya.

```
void Barberia::cortarPelo(int cliente){
    cout << "LLega el cliente: " << cliente << endl;

    if(clientes.empty()){
        if(barbero.empty()){
            cout << "El cliente: " << cliente << " espera al barbero" << endl;
            clientes.wait();
        }
        else{
            cout << "El cliente: " << cliente << " despierta al barbero" << endl;
            barbero.signal();
        }
    }
    else{
        cout << "El cliente: " << cliente << " se pone a la cola" << endl;
        clientes.signal();
    }

    cout << "El cliente: " << cliente << " se sienta en la silla del barbero" << endl;
    silla.wait();
}
```

La función del siguiente cliente lo que hace es si hay cliente esperando el barbero lo coge sino se duerme.

```
void Barberia::siguienteCliente(){
    if(clientes.empty()){
        cout << "\t\t\tEl babero se duerme, no hay clientes" << endl;
        barbero.wait();
    }
    else{
        cout << "\t\t\tEl babero llama al siguiente cliente" << endl;
        clientes.signal();
    }
}
```

La función de fin de cliente lo que hace es indicar que le cliente deja libre la silla del barbero.

```
void Barberia::finCliente(){
    cout << "\t\t\tEl babero termina de con el cliente" << endl;
    silla.signal();
}
```

La función hebra del barbero lo que hace es llamar al siguiente cliente, cortarle el pelo e indicar que acaba con él. La función hebra del cliente lo que hace es cortarse el pelo y esperar a la siguiente vez.

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

//----------------------------------------------------------------------
// función que ejecuta la hebra del cliente

void funcion_hebra_cliente(MRef<Barberia> monitor, int num_clie){
    while(true){
        monitor -> cortarPelo(num_clie);
        cout << "El cliente: " << num_clie << "espera fuera" << endl;
        esperaFueraBarberia();
    }
}
```