Práctica 2. Monitores.

Jesús Sánchez de Lechina Tejada

Contents

Productor Consumidor con FIFO

Variables utilizadas

- int **pos_lectura:** Usada para darle la funcionalidad FIFO a nuestro programa, permitiendo el control de acceso sobre nuestro buffer con dos variables.
- int **n_escrituras:** Al usar números de acceso entre 0 y 9 la condición de escritura/lectura es que no estén en la misma posición, esto puede pasar en dos ocasiones: Cuando la lectura haya alcanzado a la posición de escritura o cuando la escritura haya escrito el máximo de elementos posibles en el buffer. Esta variable permite distinguir estos dos casos.
- bool **DEBUG_MODE:** Booleana que imprime mensajes de depuración en caso de estar activada.

Código Fuente

La principal diferencia radica en que ahora la posición de lectura se aumenta una vez leída la posición.

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Seminario 2. Introducción a los monitores en C++11.
//
// archivo: prodcons_1.cpp
// Ejemplo de un monitor en C++11 con semántica SC, para el problema
// del productor/consumidor, con un único productor y un único consumidor.
// Opcion FIFO
//
// -----------------------------------------------------------------------------


#include <iostream>
#include <iomanip>
#include <cassert>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <random>

using namespace std ;

bool DEBUG_MODE = false;
constexpr int
num_items  = 40 ;     // número de items a producir/consumir

mutex
```

```
mtx ;                 // mutex de escritura en pantalla
unsigned
cont_prod[num_items], // contadores de verificación: producidos
  cont_cons[num_items]; // contadores de verificación: consumidos

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

//**********************************************************************
// 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 )
{
  if ( num_items <= dato )
    {
      cout << " dato === " << dato << ", num_items == " << num_items << endl ;
      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 ini_contadores()
{
  for( unsigned i = 0 ; i < num_items ; i++ )
    {  cont_prod[i] = 0 ;
      cont_cons[i] = 0 ;
    }
}

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

void test_contadores()
{
  bool ok = true ;
  cout << "comprobando contadores ...." << flush ;

  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 << "solución (aparentemente) correcta." << endl << flush ;
}

// *****************************************************************************
// clase para monitor buffer, version LIFO, semántica SC, un prod. y un cons.

class ProdCons1SC
{
private:
  static const int           // constantes:
  num_celdas_total = 10;   //  núm. de entradas del buffer
  int                        // variables permanentes
  buffer[num_celdas_total],//  buffer de tamaño fijo, con los datos
    primera_libre ;          //  indice de celda de la próxima inserción

  // Variable pos_lectura
  int pos_lectura;
  int n_escrituras;
```

```
  mutex
  cerrojo_monitor ;        // cerrojo del monitor
  condition_variable         // colas condicion:
  ocupadas,                //  cola donde espera el consumidor (n>0)
    libres ;                //  cola donde espera el productor  (n<num_celdas_total)

public:                    // constructor y métodos públicos
  ProdCons1SC(  ) ;           // constructor
  int  leer();                // extraer un valor (sentencia L) (consumidor)
  void escribir( int valor ); // insertar un valor (sentencia E) (productor)
} ;
// -----------------------------------------------------------------------------

ProdCons1SC::ProdCons1SC(  )
{
  primera_libre = 0 ;
  pos_lectura = 0;
  n_escrituras = 0;
}
// -----------------------------------------------------------------------------
// función llamada por el consumidor para extraer un dato

int ProdCons1SC::leer(  )
{
  // ganar la exclusión mutua del monitor con una guarda
  unique_lock<mutex> guarda( cerrojo_monitor );

  // esperar bloqueado hasta que 0 < num_celdas_ocupadas
  if(DEBUG_MODE)
    cout << "Intento leer: Primera libre = " << primera_libre << ", pos_lectura = " << pos_lect
  if ( primera_libre == pos_lectura )
    ocupadas.wait( guarda );

  if(DEBUG_MODE)
    cout << "Se ha cumplido la condición de espera de la hebra consumidora" << endl;
  // hacer la operación de lectura, actualizando estado del monitor
  assert( primera_libre != pos_lectura  );
  const int valor = buffer[pos_lectura] ;
  pos_lectura = (pos_lectura + 1) % num_celdas_total  ;
  n_escrituras--;

  if(DEBUG_MODE)
    cout << "Se consume buffer[" << pos_lectura << "] = " << buffer[pos_lectura] << endl;


  // señalar al productor que hay un hueco libre, por si está esperando
  libres.notify_one();
```

```
  // devolver valor
  return valor ;
}
// -----------------------------------------------------------------------------

void ProdCons1SC::escribir( int valor )
{
  // ganar la exclusión mutua del monitor con una guarda
  unique_lock<mutex> guarda( cerrojo_monitor );

  // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total

  if(DEBUG_MODE)
    cout << "Intento escribir el " << valor << ": Primera libre = " << primera_libre << ", pos
  if ( primera_libre == num_celdas_total  && n_escrituras == 9)
    libres.wait( guarda );

  //cout << "escribir: ocup == " << num_celdas_ocupadas << ", total == " << num_celdas_total <
  assert( !(primera_libre == num_celdas_total  && n_escrituras == 9) );

  // hacer la operación de inserción, actualizando estado del monitor
  buffer[primera_libre] = valor ;
  if(DEBUG_MODE)
    cout << "buffer[" << primera_libre << "] = " << buffer[primera_libre] << endl;
  primera_libre = (primera_libre + 1) % num_celdas_total ;
  n_escrituras++;

  // señalar al consumidor que ya hay una celda ocupada (por si esta esperando)
  ocupadas.notify_one();
}
// *****************************************************************************
// funciones de hebras

void funcion_hebra_productora( ProdCons1SC * monitor )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
    {
      int valor = producir_dato() ;
      monitor->escribir( valor );
    }
}
// -----------------------------------------------------------------------------

void funcion_hebra_consumidora( ProdCons1SC * monitor )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
    {
```

```
      int valor = monitor->leer();
      consumir_dato( valor ) ;
    }
}
// -----------------------------------------------------------------------------

int main()
{
  cout << "-------------------------------------------------------------------------------" <<
       << "Problema de los productores-consumidores (1 prod/cons, Monitor SC, buffer FIFO). " <
       << "-------------------------------------------------------------------------------" <<
       << flush ;

  ProdCons1SC monitor ;

  thread hebra_productora ( funcion_hebra_productora, &monitor ),
    hebra_consumidora( funcion_hebra_consumidora, &monitor );

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

  // comprobar que cada item se ha producido y consumido exactamente una vez
  test_contadores() ;
}
```

Múltiples productores y consumidores LIFO

Variables utilizadas

- const int **n__productoras:** Número de hebras productoras, simularemos que hay más consumidores que productores.
- const int **n__consumidoras:** Número de hebras consumidoras
- int **id:** Como parámetro para las funciones de las hebras productoras y consumidoras le indicamos un ID para que se repartan la carga de trabajo. Esto lo conseguimos inicializando el bucle la primera iteración sobre el elemento en la posición ID y aumentando en función del número de hebras consumidoras/productoras segun corresponda.

Código fuente

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Seminario 2. Introducción a los monitores en C++11.
//
// archivo: prodcons_1.cpp
```

```cpp
// Ejemplo de un monitor en C++11 con semántica SC, para el problema
// del productor/consumidor, con múltiples productores y consumidores.
// Opcion LIFO (stack)
//
// Historial:
// Creado en Julio de 2017
// -----------------------------------------------------------------------------


#include <iostream>
#include <iomanip>
#include <cassert>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <random>

using namespace std ;

constexpr int
   num_items  = 40 ;     // número de items a producir/consumir

const int n_productoras = 2;
const int n_consumidoras = 4;

mutex
   mtx ;                 // mutex de escritura en pantalla
unsigned
   cont_prod[num_items], // contadores de verificación: producidos
   cont_cons[num_items]; // contadores de verificación: consumidos

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

//**********************************************************************
// 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 )
{
   if ( num_items <= dato )
   {
      cout << " dato === " << dato << ", num_items == " << num_items << endl ;
      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 ini_contadores()
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {  cont_prod[i] = 0 ;
      cont_cons[i] = 0 ;
   }
}

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

void test_contadores()
{
   bool ok = true ;
   cout << "comprobando contadores ...." << flush ;

   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 << "solución (aparentemente) correcta." << endl << flush ;
}

// *****************************************************************************
// clase para monitor buffer, version LIFO, semántica SC, un prod. y un cons.

class ProdCons1SC
{
 private:
 static const int           // constantes:
   num_celdas_total = 10;   //  núm. de entradas del buffer
 int                        // variables permanentes
   buffer[num_celdas_total],//  buffer de tamaño fijo, con los datos
   primera_libre ;          //  indice de celda de la próxima inserción
 mutex
   cerrojo_monitor ;        // cerrojo del monitor
 condition_variable         // colas condicion:
   ocupadas,                //  cola donde espera el consumidor (n>0)
   libres ;                 //  cola donde espera el productor  (n<num_celdas_total)

 public:                    // constructor y métodos públicos
   ProdCons1SC(  ) ;           // constructor
   int  leer();                // extraer un valor (sentencia L) (consumidor)
   void escribir( int valor ); // insertar un valor (sentencia E) (productor)
} ;
// -----------------------------------------------------------------------------

ProdCons1SC::ProdCons1SC(  )
{
   primera_libre = 0 ;
}
// -----------------------------------------------------------------------------
// función llamada por el consumidor para extraer un dato

int ProdCons1SC::leer(  )
{
   // ganar la exclusión mutua del monitor con una guarda
   unique_lock<mutex> guarda( cerrojo_monitor );

   // esperar bloqueado hasta que 0 < num_celdas_ocupadas
```

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

   // hacer la operación de lectura, actualizando estado del monitor
   assert( 0 < primera_libre  );
   primera_libre-- ;
   const int valor = buffer[primera_libre] ;


   // señalar al productor que hay un hueco libre, por si está esperando
   libres.notify_one();

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

void ProdCons1SC::escribir( int valor )
{
   // ganar la exclusión mutua del monitor con una guarda
   unique_lock<mutex> guarda( cerrojo_monitor );

   // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total
   if ( primera_libre == num_celdas_total )
      libres.wait( guarda );

   //cout << "escribir: ocup == " << num_celdas_ocupadas << ", total == " << num_celdas_total
   assert( primera_libre < num_celdas_total );

   // hacer la operación de inserción, actualizando estado del monitor
   buffer[primera_libre] = valor ;
   primera_libre++ ;

   // señalar al consumidor que ya hay una celda ocupada (por si esta esperando)
   ocupadas.notify_one();
}
// *****************************************************************************
// funciones de hebras

void funcion_hebra_productora( ProdCons1SC * monitor, int id )
{
   for( unsigned i = id ; i < num_items ; i+=n_productoras)
   {
      int valor = producir_dato() ;
      //cout << "soy hebra productora " << id << " produzco: " << valor<< endl;
      monitor->escribir( valor );
   }
}
```

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

void funcion_hebra_consumidora( ProdCons1SC * monitor, int id )
{
   for( unsigned i = id ; i < num_items ; i+=n_consumidoras )
   {
      int valor = monitor->leer();
      //cout << "Soy hebra consumidora " << id << " consumo: " << valor<< endl;
      consumir_dato( valor ) ;
   }
}
// ---------------------------------------------------------------------

int main()
{
   cout << "-------------------------------------------------------------------------------" <<
        << "Problema de los productores-consumidores (multiples prod/cons, Monitor SC, buffer l
        << "-------------------------------------------------------------------------------" <<
        << flush ;

   ProdCons1SC monitor ;

   thread hebra_productora0 ( funcion_hebra_productora, &monitor,0 ),
     hebra_productora1 ( funcion_hebra_productora, &monitor,1 ),
     hebra_consumidora0( funcion_hebra_consumidora, &monitor,0 ),
     hebra_consumidora1( funcion_hebra_consumidora, &monitor,1 ),
     hebra_consumidora2( funcion_hebra_consumidora, &monitor,2 ),
     hebra_consumidora3( funcion_hebra_consumidora, &monitor,3 );


   hebra_productora0.join() ;
   hebra_productora1.join() ;

   hebra_consumidora0.join() ;
   hebra_consumidora1.join() ;
   hebra_consumidora2.join() ;
   hebra_consumidora3.join() ;

   // comprobar que cada item se ha producido y consumido exactamente una vez
   test_contadores() ;
}
```

Múltiples productores y consumidores FIFO

Variables utilizadas

Partiendo de la previa versión de LIFO para un único consumidor-productor, con las variables de posición de escritura/lectura del buffer intermedio y la booleana de depuración, añadimos las siguientes variables, idénticas al la versión FIFO:

- const int **n_productoras:** Número de hebras productoras, de nuevo, habrá menos productores que consumidores (2 y 4).
- const int **n_consumidoras:** Número de hebras consumidoras.
- int **id:** Como parámetro para las funciones de las hebras productoras y consumidoras le indicamos un ID para que se repartan la carga de trabajo. Esto lo conseguimos inicializando el bucle la primera iteración sobre el elemento en la posición ID y aumentando en función del número de hebras consumidoras/productoras segun corresponda.

Código fuente

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Seminario 2. Introducción a los monitores en C++11.
//
// archivo: prodcons_1.cpp
// Ejemplo de un monitor en C++11 con semántica SC, para el problema
// del productor/consumidor, con múltiples productores y consumidores.
// Opcion FIFO
//
// -----------------------------------------------------------------------------


#include <iostream>
#include <iomanip>
#include <cassert>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <random>

using namespace std ;

const int n_productoras = 2;
const int n_consumidoras = 4;
bool DEBUG_MODE = false;
constexpr int
num_items  = 40 ;     // número de items a producir/consumir
```

```
mutex
mtx ;                 // mutex de escritura en pantalla
unsigned
cont_prod[num_items], // contadores de verificación: producidos
  cont_cons[num_items]; // contadores de verificación: consumidos

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

//**********************************************************************
// 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 )
{
  if ( num_items <= dato )
    {
      cout << " dato === " << dato << ", num_items == " << num_items << endl ;
      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 ini_contadores()
{
  for( unsigned i = 0 ; i < num_items ; i++ )
    {  cont_prod[i] = 0 ;
      cont_cons[i] = 0 ;
    }
}

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

void test_contadores()
{
  bool ok = true ;
  cout << "comprobando contadores ...." << flush ;

  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 << "solución (aparentemente) correcta." << endl << flush ;
}

// *****************************************************************************
// clase para monitor buffer, version LIFO, semántica SC, un prod. y un cons.

class ProdCons1SC
{
private:
  static const int           // constantes:
  num_celdas_total = 10;   //  núm. de entradas del buffer
  int                        // variables permanentes
  buffer[num_celdas_total],//  buffer de tamaño fijo, con los datos
    primera_libre ;          //  indice de celda de la próxima inserción

  // Variable pos_lectura
```

```
  int pos_lectura;
  int n_escrituras;

  mutex
  cerrojo_monitor ;        // cerrojo del monitor
  condition_variable         // colas condicion:
  ocupadas,                //  cola donde espera el consumidor (n>0)
    libres ;                 //  cola donde espera el productor  (n<num_celdas_total)

public:                    // constructor y métodos públicos
  ProdCons1SC(  ) ;           // constructor
  int  leer();                // extraer un valor (sentencia L) (consumidor)
  void escribir( int valor ); // insertar un valor (sentencia E) (productor)
} ;
// -----------------------------------------------------------------------------

ProdCons1SC::ProdCons1SC(  )
{
  primera_libre = 0 ;
  pos_lectura = 0;
  n_escrituras = 0;
}
// -----------------------------------------------------------------------------
// función llamada por el consumidor para extraer un dato

int ProdCons1SC::leer(  )
{
  // ganar la exclusión mutua del monitor con una guarda
  unique_lock<mutex> guarda( cerrojo_monitor );

  // esperar bloqueado hasta que 0 < num_celdas_ocupadas
  if(DEBUG_MODE)
    cout << "Intento leer: Primera libre = " << primera_libre << ", pos_lectura = " << pos_lect
  if ( primera_libre == pos_lectura )
    ocupadas.wait( guarda );

  if(DEBUG_MODE)
    cout << "Se ha cumplido la condición de espera de la hebra consumidora" << endl;
  // hacer la operación de lectura, actualizando estado del monitor
  assert( primera_libre != pos_lectura  );
  const int valor = buffer[pos_lectura] ;
  pos_lectura = (pos_lectura + 1) % num_celdas_total  ;
  n_escrituras--;

  if(DEBUG_MODE)
    cout << "Se consume buffer[" << pos_lectura << "] = " << buffer[pos_lectura] << endl;
```

```
  // señalar al productor que hay un hueco libre, por si está esperando
  libres.notify_one();

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

void ProdCons1SC::escribir( int valor )
{
  // ganar la exclusión mutua del monitor con una guarda
  unique_lock<mutex> guarda( cerrojo_monitor );

  // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total

  if(DEBUG_MODE)
    cout << "Intento escribir el " << valor << ": Primera libre = " << primera_libre << ", pos
  if ( primera_libre == num_celdas_total  && n_escrituras == 9)
    libres.wait( guarda );

  //cout << "escribir: ocup == " << num_celdas_ocupadas << ", total == " << num_celdas_total <
  assert( !(primera_libre == num_celdas_total  && n_escrituras == 9) );

  // hacer la operación de inserción, actualizando estado del monitor
  buffer[primera_libre] = valor ;
  if(DEBUG_MODE)
    cout << "buffer[" << primera_libre << "] = " << buffer[primera_libre] << endl;
  primera_libre = (primera_libre + 1) % num_celdas_total ;
  n_escrituras++;

  // señalar al consumidor que ya hay una celda ocupada (por si esta esperando)
  ocupadas.notify_one();
}
// *****************************************************************************
// funciones de hebras

void funcion_hebra_productora( ProdCons1SC * monitor, int id )
{
  for( unsigned i = id ; i < num_items ; i+=n_productoras )
    {
      int valor = producir_dato() ;
      monitor->escribir( valor );
    }
}
// -----------------------------------------------------------------------------

void funcion_hebra_consumidora( ProdCons1SC * monitor, int id )
{
```

```
  for( unsigned i = id ; i < num_items ; i+=n_consumidoras )
    {
      int valor = monitor->leer();
      consumir_dato( valor ) ;
    }
}
// -----------------------------------------------------------------------------

int main()
{
  cout << "-------------------------------------------------------------------------------" <<
       << "Problema de los productores-consumidores (multiples prod/cons, Monitor SC, buffer F
       << "-------------------------------------------------------------------------------" <<
       << flush ;

  ProdCons1SC monitor ;

   thread hebra_productora0 ( funcion_hebra_productora, &monitor,0 ),
     hebra_productora1 ( funcion_hebra_productora, &monitor,1 ),
     hebra_consumidora0( funcion_hebra_consumidora, &monitor,0 ),
     hebra_consumidora1( funcion_hebra_consumidora, &monitor,1 ),
     hebra_consumidora2( funcion_hebra_consumidora, &monitor,2 ),
     hebra_consumidora3( funcion_hebra_consumidora, &monitor,3 );


   hebra_productora0.join() ;
   hebra_productora1.join() ;

   hebra_consumidora0.join() ;
   hebra_consumidora1.join() ;
   hebra_consumidora2.join() ;
   hebra_consumidora3.join() ;

  // comprobar que cada item se ha producido y consumido exactamente una vez
  test_contadores() ;
}
```

Múltiples productores y consumidores con semántica SU LIFO

Variables usadas

En este caso hay que destacar, más que variables utilizadas, los cambios que se han producido, pues con la definición del nuevo tipo de monitor con semántica SU ahora todas las variables de tipo ProdCons1SC que usábamos o que esperábamos como argumento son de la nueva clase que hemos creado: **ProdCons1SU**.

Eliminamos el cerrojo del monitor.

Las colas las sustituimos por colas del tipo `CondVar`, y en el constructor las inicializaremos con `newCondVar()`. Esto implica que ahora los "*wait()*" se llamarán sin guardas, y en lugar de usar "*notify_one()*" se llamará a "*signal()*" para indicar a la otra cola que puede liberar a una hebra para realizar su función.

`primera_libre`: Vuelve a ser el índice de nuestra pila (LIFO).

Hemos sustituido el monitor en el main por una referencia a este. Esto lo conseguimos declarándolo así:

```
MRef<ProdCons1SU> monitor = Create<ProdCons1SU>() ;
```

Esto implica la necesidad de realizar ciertos cambios. Primero la hebra habrá que lanzarla pasándole esta variable monitor, en lugar de lanzarla mediante una referencia (ahora no es necesario pues la variable monitor ya es una referencia).

Además en las funciones de las hebras habrá que cambiar el tipo de dato del argumento, que ahora serán del tipo

```
(MRef<ProdCons1SU> monitor, int id)
```

Código fuente

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Seminario 2. Introducción a los monitores en C++11.
//
// archivo: prodcons_1.cpp
// Ejemplo de un monitor en C++11 con semántica SC, para el problema
// del productor/consumidor, con múltiples productores y consumidores.
// Opcion LIFO (stack)
//
// Historial:
// Creado en Julio de 2017
// -----------------------------------------------------------------------------


#include <iostream>
#include <iomanip>
#include <cassert>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <random>
#include "HoareMonitor.hpp"
using namespace HM;
using namespace std ;

constexpr int
```

```
   num_items  = 40 ;     // número de items a producir/consumir

const int n_productoras = 2;
const int n_consumidoras = 4;

mutex
   mtx ;                 // mutex de escritura en pantalla
unsigned
   cont_prod[num_items], // contadores de verificación: producidos
   cont_cons[num_items]; // contadores de verificación: consumidos

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

//**********************************************************************
// 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 )
{
   if ( num_items <= dato )
   {
      cout << " dato === " << dato << ", num_items == " << num_items << endl ;
      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 ini_contadores()
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {  cont_prod[i] = 0 ;
      cont_cons[i] = 0 ;
   }
}

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

void test_contadores()
{
   bool ok = true ;
   cout << "comprobando contadores ...." << flush ;

   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 << "solución (aparentemente) correcta." << endl << flush ;
}

// *****************************************************************************
// clase para monitor buffer, version LIFO, semántica SU, varios prodcons.

class ProdCons1SU:public HoareMonitor
{
 private:
 static const int           // constantes:
   num_celdas_total = 10;   //  núm. de entradas del buffer
 int                        // variables permanentes
```

```
   buffer[num_celdas_total],//  buffer de tamaño fijo, con los datos
   primera_libre ;          //  indice de celda de la próxima inserción
  //mutex
  //cerrojo_monitor ;        // cerrojo del monitor
 CondVar          // colas condicion:
   ocupadas,                //  cola donde espera el consumidor (n>0)
   libres ;                 //  cola donde espera el productor  (n<num_celdas_total)

 public:                    // constructor y métodos públicos
   ProdCons1SU(  ) ;           // constructor
   int  leer();                // extraer un valor (sentencia L) (consumidor)
   void escribir( int valor ); // insertar un valor (sentencia E) (productor)
} ;
// -----------------------------------------------------------------------------

ProdCons1SU::ProdCons1SU(  )
{
   primera_libre = 0 ;
   ocupadas = newCondVar();
   libres = newCondVar();
}
// -----------------------------------------------------------------------------
// función llamada por el consumidor para extraer un dato

int ProdCons1SU::leer(  )
{

   // esperar bloqueado hasta que 0 < num_celdas_ocupadas
   if ( primera_libre == 0 )
      ocupadas.wait(  );

   // hacer la operación de lectura, actualizando estado del monitor
   assert( 0 < primera_libre  );
   primera_libre-- ;
   const int valor = buffer[primera_libre] ;


   // señalar al productor que hay un hueco libre, por si está esperando
   libres.signal();

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

void ProdCons1SU::escribir( int valor )
{
   // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total
```

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

   //cout << "escribir: ocup == " << num_celdas_ocupadas << ", total == " << num_celdas_total
   assert( primera_libre < num_celdas_total );

   // hacer la operación de inserción, actualizando estado del monitor
   buffer[primera_libre] = valor ;
   primera_libre++ ;

   // señalar al consumidor que ya hay una celda ocupada (por si esta esperando)
   ocupadas.signal();
}
// ****************************************************************************
// funciones de hebras

void funcion_hebra_productora( MRef<ProdCons1SU> monitor, int id )
{
   for( unsigned i = id ; i < num_items ; i+=n_productoras)
   {
      int valor = producir_dato() ;
      //cout << "soy hebra productora " << id << " produzco: " << valor<< endl;
      monitor->escribir( valor );
   }
}
// ----------------------------------------------------------------------------

void funcion_hebra_consumidora( MRef<ProdCons1SU> monitor, int id )
{
   for( unsigned i = id ; i < num_items ; i+=n_consumidoras )
   {
      int valor = monitor->leer();
      //cout << "Soy hebra consumidora " << id << " consumo: " << valor<< endl;
      consumir_dato( valor ) ;
   }
}
// ----------------------------------------------------------------------------

int main()
{
   cout << "------------------------------------------------------------------------------" <<
        << "Problema de los productores-consumidores (multiples prod/cons, Monitor SU, buffer
        << "------------------------------------------------------------------------------" <<
        << flush ;

   MRef<ProdCons1SU> monitor = Create<ProdCons1SU>() ;

   thread hebra_productora0 ( funcion_hebra_productora, monitor,0 ),
```

```
      hebra_productora1 ( funcion_hebra_productora, monitor,1 ),
      hebra_consumidora0( funcion_hebra_consumidora, monitor,0 ),
      hebra_consumidora1( funcion_hebra_consumidora, monitor,1 ),
      hebra_consumidora2( funcion_hebra_consumidora, monitor,2 ),
      hebra_consumidora3( funcion_hebra_consumidora, monitor,3 );


   hebra_productora0.join() ;
   hebra_productora1.join() ;

   hebra_consumidora0.join() ;
   hebra_consumidora1.join() ;
   hebra_consumidora2.join() ;
   hebra_consumidora3.join() ;

   // comprobar que cada item se ha producido y consumido exactamente una vez
   test_contadores() ;
}
```

Múltiples productores y consumidores con semántica SU FIFO

Variables usadas

Para este ejercicio hemos partido del apartado anterior, múltiples productores/consumidores con semántica SU LIFO.

El primer cambio que hemos hecho se aprecia en la función main. Se ha sustituido la inicialización individual de cada hebra por una inicialización en un array para así dar cabida a problemas de escalado sin tener que modificar más que el valor de las variables que indican el número de hebras productoras y consumidoras.

Posteriormente hemos modificado el funcionamiento para adaptar el uso del buffer a una cola (FIFO).

Hemos añadido dos variables booleanas para comprobar si el buffer está lleno o si tiene todas sus posiciones libres:

```
bool todas_ocupadas;
bool todas_libres;
```

En un primer momento todas libres se incializará a `true` y todas ocupadas a `false`. La utilidad de estas variables es poder discernir entre el caso en el que la posición de lectura alcance a la de escritura del caso contrario.

Código fuente

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

```
// Sistemas concurrentes y Distribuidos.
// Seminario 2. Introducción a los monitores en C++11.
//
// archivo: prodcons_1.cpp
// Ejemplo de un monitor en C++11 con semántica SC, para el problema
// del productor/consumidor, con múltiples productores y consumidores.
// Opcion LIFO (stack)
//
// Historial:
// Creado en Julio de 2017
// -----------------------------------------------------------------------------


#include <iostream>
#include <iomanip>
#include <cassert>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <random>
#include "HoareMonitor.hpp"
using namespace HM;
using namespace std ;

constexpr int
   num_items  = 40 ;     // número de items a producir/consumir

const int n_productoras = 2;
const int n_consumidoras = 4;

mutex
   mtx ;                 // mutex de escritura en pantalla
unsigned
   cont_prod[num_items], // contadores de verificación: producidos
   cont_cons[num_items]; // contadores de verificación: consumidos

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

```
//**********************************************************************
// 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 )
{
   if ( num_items <= dato )
   {
      cout << " dato === " << dato << ", num_items == " << num_items << endl ;
      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 ini_contadores()
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {  cont_prod[i] = 0 ;
      cont_cons[i] = 0 ;
   }
}

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

void test_contadores()
{
   bool ok = true ;
   cout << "comprobando contadores ...." << flush ;

   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 << "solución (aparentemente) correcta." << endl << flush ;
}

// *****************************************************************************
// clase para monitor buffer, version FIFO, semántica SU, varios prodcons.

class ProdCons1SU:public HoareMonitor
{
 private:
 static const int           // constantes:
   num_celdas_total = 10;   //  núm. de entradas del buffer
 int                        // variables permanentes
   buffer[num_celdas_total],//  buffer de tamaño fijo, con los datos
   primera_libre ;          //  indice de celda de la próxima inserción

  int primera_ocupada;
  bool todas_ocupadas, todas_libres;
  //mutex
  //cerrojo_monitor ;         // cerrojo del monitor
 CondVar         // colas condicion:
   ocupadas,                //  cola donde espera el consumidor (n>0)
   libres ;                 //  cola donde espera el productor  (n<num_celdas_total)

 public:                    // constructor y métodos públicos
   ProdCons1SU(  ) ;           // constructor
   int  leer();                // extraer un valor (sentencia L) (consumidor)
   void escribir( int valor ); // insertar un valor (sentencia E) (productor)
} ;
// -----------------------------------------------------------------------------

ProdCons1SU::ProdCons1SU(  )
{
  todas_ocupadas = false;
  todas_libres = true;
   primera_libre = 0 ;
```

```
   primera_ocupada = 0;
   ocupadas = newCondVar();
   libres = newCondVar();
}
// -----------------------------------------------------------------------------
// función llamada por el consumidor para extraer un dato

int ProdCons1SU::leer(  )
{

   // esperar bloqueado hasta que 0 < num_celdas_ocupadas
   if ( todas_libres )
      ocupadas.wait(  );

   // hacer la operación de lectura, actualizando estado del monitor
   const int valor = buffer[primera_ocupada] ;
   primera_ocupada = (primera_ocupada + 1) % num_celdas_total  ;
   if(primera_ocupada == primera_libre)
     todas_libres = true;
   todas_ocupadas = false;


   // señalar al productor que hay un hueco libre, por si está esperando
   libres.signal();

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

void ProdCons1SU::escribir( int valor )
{
   // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total
   if ( todas_ocupadas)
      libres.wait(  );

   //cout << "escribir: ocup == " << num_celdas_ocupadas << ", total == " << num_celdas_total

   // hacer la operación de inserción, actualizando estado del monitor
   buffer[primera_libre] = valor ;
  primera_libre = (primera_libre + 1) % num_celdas_total ;
  if(primera_ocupada == primera_libre)
    todas_ocupadas = true;
  todas_libres = false;


   // señalar al consumidor que ya hay una celda ocupada (por si esta esperando)
   ocupadas.signal();
```

```
}
// *****************************************************************************
// funciones de hebras

void funcion_hebra_productora( MRef<ProdCons1SU> monitor, int id )
{
   for( unsigned i = id ; i < num_items ; i+=n_productoras)
   {
      int valor = producir_dato() ;
      //cout << "soy hebra productora " << id << " produzco: " << valor<< endl;
      monitor->escribir( valor );
   }
}
// ----------------------------------------------------------------------------

void funcion_hebra_consumidora( MRef<ProdCons1SU> monitor, int id )
{
   for( unsigned i = id ; i < num_items ; i+=n_consumidoras )
   {
      int valor = monitor->leer();
      //cout << "Soy hebra consumidora " << id << " consumo: " << valor<< endl;
      consumir_dato( valor ) ;
   }
}
// ----------------------------------------------------------------------------

int main()
{
   cout << "--------------------------------------------------------------------------------" <<
        << "Problema de los productores-consumidores (multiples prod/cons, Monitor SU, buffer
        << "--------------------------------------------------------------------------------" <<
        << flush ;

   MRef<ProdCons1SU> monitor = Create<ProdCons1SU>() ;

   thread hebras_prod[n_productoras];
   thread hebras_cons[n_consumidoras];

   for(int i = 0; i < n_productoras;i++)
     hebras_prod[i] = thread (funcion_hebra_productora, monitor, i);

   for(int i = 0; i < n_consumidoras;i++)
     hebras_cons[i] = thread (funcion_hebra_consumidora,monitor,i);


   for(int i = 0; i < n_productoras;i++)
     hebras_prod[i].join();
```

```
   for(int i = 0; i < n_consumidoras;i++)
     hebras_cons[i].join();

   // comprobar que cada item se ha producido y consumido exactamente una vez
   test_contadores() ;
}
```

Fumadores con SU

Para este ejercicio extenderemos de nuevo la funcionalidad de la clase HoareMonitor para permitir la implementación de nuestro monitor con semántica SU.

Métodos y variables utilizadas

Variable global:

`n_fumadores`: Refleja el número de fumadores.

Clase `MonitorFumadores`: Nuestro monitor que gestiona el acceso a los recursos para los fumadores y el estanquero.

Para implementarla necesitamos las siguientes **variables privadas**:

`int ingrediente_listo`: Ingrediente que hay en el mostrador que se necesita para fumar. Cuando no hay ninguno se representará como -1.

`CondVar c_fumadores[int n_fumadores]`: Conjunto de colas para los recursos de los fumadores.

`CondVar c_estanquero`: Única cola para el estanquero, para que espere cuando el ingrediente no ha sido retirado.

Métodos públicos:

`Monitorfumadores()`: Constructor que inicializa los atributos de la clase según corresponda.

`void PonerIngrediente(int ingrediente)`: Usada por la hebra estanquera para poner un ingrediente en el mostrador.

`void CogerIngrediente(int ingrediente)`: Usada por las hebras fumadoras para tomar un ingrediente del mostrador.

`void EsperaEstanquero()`: El estanquero espera hasta que vuelvan a extraer un ingrediente y el mostrador esté vacío.

Otras funciones:

Hemos reciclado las funciones `aleatorio` y `fumar` de las prácticas anteriores.

Además hemos realizado adaptaciones a las funciones de las hebras estanquera y fumadoras. Para que hagan uso de los métodos del monitor.

Main:

En el programa principal simplemente declaramos el monitor SU como hicimos en los ejercicios anteriores de SU, luego declaramos e inicializamos cada hebra con la función correspondiente y, aunque nunca se llegue a dar el caso dada la naturaleza de los bucles infinitos de ambas hebras, se esperan con un join en cada una.

Código fuente

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 2. Introducción a los monitores en C++11.
//
// archivo: fumadores.cpp
//
// Historial:
// Creado en Noviembre de 2017
// -----------------------------------------------------------------------------


#include <iostream>
#include <iomanip>
#include <cassert>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <random>
#include "HoareMonitor.hpp"
using namespace HM;
using namespace std;

const int n_fumadores = 3;


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

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

```
// Función que simula la acción de fumar, como un retardo aleatoria de la hebra

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

}



//**************************************************************************
class MonitorFumadores : public HoareMonitor {
private:
  int ingrediente_listo;
  CondVar c_fumadores[n_fumadores], c_estanquero;
 public:
   MonitorFumadores();
   void PonerIngrediente(int ingrediente);
   void CogerIngrediente(int ingrediente);
   void EsperaEstanquero();
} ;
// -----------------------------------------------------------------------------


MonitorFumadores::MonitorFumadores(){

  ingrediente_listo = -1;

  // Como queremos controlar el acceso  a los n_fumadores recursos y como cada
  // fumador requiere un recurso tomamos ese número de colas
  for(int i = 0; i < n_fumadores; i++)
    c_fumadores[i] = newCondVar();

  c_estanquero = newCondVar();
```

```
}

void MonitorFumadores::PonerIngrediente(int ingrediente){
  ingrediente_listo = ingrediente;
  c_fumadores[ingrediente].signal(); // Avisamos a la cola donde estén
                    // esperando al recurso producido
}

void MonitorFumadores::CogerIngrediente(int ingrediente){

  // Esta hebra se ejecutará hasta aquí desde el primer momento,
  // entonces nos interesa que si su ingrediente no está, que espere
  // en la cola correspondiente y que el estanquero le avise una vez
  // haya sido producido el ingrediente que necesite.
  if(ingrediente_listo != ingrediente)
    c_fumadores[ingrediente].wait();

  // Posteriormente ponemos ingrediente a -1 y avisamos al estanquero
  // para que produzca

  ingrediente_listo = -1;
  c_estanquero.signal();
}

void MonitorFumadores::EsperaEstanquero(){
  if(ingrediente_listo != -1)
    c_estanquero.wait();
}

//------------------Fin-de-la-implementación-del-monitor----------------


void funcion_hebra_fumador(MRef<MonitorFumadores> monitor, int id_fumador){
  while(true){
    monitor->CogerIngrediente(id_fumador);
    fumar(id_fumador);
  }
}

void funcion_hebra_estanquero(MRef<MonitorFumadores> monitor){
  while(true){
    int ingrediente_generado = aleatorio<0,n_fumadores-1>();
    cout << "\\tEl estanquero ha producido el ingrediente " << ingrediente_generado << endl;
    monitor->PonerIngrediente(ingrediente_generado);
    monitor->EsperaEstanquero();
  }
}
```

```
int main(){

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

  thread fumadores[n_fumadores];
  thread estanquero (funcion_hebra_estanquero,monitor);

  for(int i = 0; i < n_fumadores; i++)
    fumadores[i] = thread(funcion_hebra_fumador,monitor, i);

  for(int i = 0; i < n_fumadores; i++)
    fumadores[i].join();

  estanquero.join();
}
```

Barbero SU

Igual que en el ejercicio de los fumadores, usaremos una clase monitor que herede de HoareMonitor para dar la funcionabilidad de SU a nuestro monitor.

Métodos y variables utilizadas

Variables de ámbito global:

`const int n_clientes`: Indica el número de clientes de la barbería.

Funciones de ámbito global:

Reutilizamos el código para generar números aleatorios.

Utilizaremos dos métodos que representen tanto la espera que realizan los clientes fuera de la barbería como el tiempo que se tarda en cortar el pelo (`EsperaCortarPelo` y `EsperaCliente`). Esto lo haremos con la intención de facilitar la lectura del código, pues ambas harán uso de un tercer método (`EsperaAleatoria`), que será una adaptación de la función `fumar` del ejercicio anterior, para representar esta espera.

Atributos privados del monitor:

`CondVar c_clientes, c_barbero, c_silla`: Necesitaremos las colas de condición para que el barbero duerma, para el acceso al recurso silla por parte de los clientes y luego otra cola para esperar si la silla no está disponible.

`bool libre`: Indica si la silla está libre o no. Esencialmente útil cuando no haya clientes en la cola pero sí alguien sentado, para que no intente acceder al recurso.

Métodos públicos del monitor:

El constructor del monitor que inicializa las colas e indica que la silla en un primer momento está libre.

`void CortarPelo()`: Simula la acción del cliente que entra a la barbería, comprueba si puede pasar a ser atendido o se pone en cola. Cuando puede pasar, avisa al barbero y este le atiende, ocupando la silla. Finalmente, cuando el barbero le avisa abandona la barbería.

`void SiguienteCliente()`: Acción del barbero, comprueba si hay clientes para dormirse. Si hay clientes en cola, avisa al primero para atenderle.

`void TerminarCliente()`: Es la acción del barbero al terminar de cortarle el pelo a un cliente, indica que la silla está libre y libera al cliente que estuviese esperando en la silla a terminar de ser pelado.

Funciones de las hebras:

- Hebra barbero: Se le pasa una referencia al monitor y se encarga de gestionar el siguiente cliente, realizar la espera aleatoria correspondiente al corte de pelo y de despachar al cliente.
- Hebra cliente: Se le pasa una referencia al monitor y el número de cliente para poder dar información sobre la situación de la barbería. Llama a CortarPelo y luego simula la espera que realizan fuera de la barbería los clientes.

Programa principal:

En la función main se repite el procedimiento del ejercicio anterior. Se inicializa el monitor, se inicializan las hebras con sus correspondientes funciones y argumentos y por último, se espera a que se junten con un join. Pero, una vez más, esto no sucederá dada la naturaleza de las hebras de bucles infinitos.

Código fuente

```
  // -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 2. Introducción a los monitores en C++11.
//
// archivo: Barbero.cpp
//
// Historial:
// Creado en Noviembre de 2017
// -----------------------------------------------------------------------------


#include <iostream>
#include <iomanip>
#include <cassert>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <random>
#include "HoareMonitor.hpp"
using namespace HM;
```

```
using namespace std;

const int n_clientes = 5;

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

void EsperaAleatoria(){
   // calcular milisegundos aleatorios de duración de la espera
   chrono::milliseconds espera( aleatorio<20,200>() );

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


void EsperaCliente(){
  EsperaAleatoria();
}
void EsperaCortarPelo(){
  EsperaAleatoria();
}


//------------------Definición-de-nuestro-monitor--------------------

class MonitorBarbero : public HoareMonitor {
private:
  CondVar c_clientes, c_barbero, c_silla;
  bool libre;

 public:
   MonitorBarbero ();
   void CortarPelo(int cliente);
   void SiguienteCliente();
   void TerminarCliente();
} ;
```

```
MonitorBarbero::MonitorBarbero (){
  libre = true;
  c_clientes = newCondVar();
  c_barbero = newCondVar();
  c_silla = newCondVar();
}

void MonitorBarbero::CortarPelo(int cliente){
  cout << "El cliente " << cliente << " ha entrado" << endl;

  // Cuando el cliente entra y hay alguien sentado o algún cliente en
  // la cola esperando este tiene que ponerse a la cola
  if(!c_clientes.empty() || !libre){
    c_clientes.wait();
  }
  // Una vez sale de la cola es porque la silla se ha quedado libre,
  // entonces se sienta y avisa al barbero de que está listo.
  libre = false;
  c_barbero.signal();

  cout << "El cliente " << cliente << " ha avisado al barbero y ha ocupado la silla" << endl;

  c_silla.wait();
  // Cuando el barbero le haya avisado el cliente podrá irse y
  // desocupará la silla
  cout << "El cliente " << cliente << " ha dejado la silla libre" << endl;
}

void MonitorBarbero::SiguienteCliente(){
  // Si no hay clientes en la cola, el barbero duerme, en caso
  // contrario se encarga de proceder con el siguiente cliente.
  if(c_clientes.empty()){
    cout << "\\tNo hay nadie esperando, el barbero se duerme." << endl;
    c_barbero.wait();
  } else
    c_clientes.signal();

  cout << "\\tEl barbero recibe al siguiente cliente" << endl;
}


void MonitorBarbero::TerminarCliente(){
  cout << "\\tEl barbero termina con el cliente" << endl;
  libre = true;
  c_silla.signal();
}
```

```
//--------------------Fin-de-la-implementación-del-monitor--------------


// Funciones Hebras:

void funcion_hebra_barbero(MRef<MonitorBarbero> monitor){
  while(true){
    monitor->SiguienteCliente();
    EsperaCortarPelo();
    monitor->TerminarCliente();
  }
}

void funcion_hebra_cliente( MRef<MonitorBarbero> monitor, int cliente ){
   while( true ){
     monitor->CortarPelo(cliente);
     EsperaCliente();
   }
}

int main(){

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

  thread clientes[n_clientes];
  thread barbero (funcion_hebra_barbero,monitor);

  for(int i = 0; i < n_clientes; i++)
    clientes[i] = thread(funcion_hebra_cliente,monitor, i);

  for(int i = 0; i < n_clientes; i++)
    clientes[i].join();

  barbero.join();
}
```