Práctica 1. Sincronización de hebras con semáforos.

1.Productor-Consumidor con buffer FIFO

```
// ---------------------------------------------------------------------
// prodcons-FIFO.cpp
//
// Se completa el problema del productor-consumidor con un buffer para guardar
// los datos. Se realiza mediante FIFO(primero en entrar es el primero en salir).
// ---------------------------------------------------------------------

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

using namespace std ;
using namespace scd ;

//**********************************************************************
// Variables globales

const unsigned
num_items = 40 ,   // número de items
tam_vec   = 10 ;   // tamaño del buffer
unsigned
cont_prod[num_items] = {0}, // contadores de verificación: para cada dato, número de veces que se ha producido.
cont_cons[num_items] = {0}, // contadores de verificación: para cada dato, número de veces que se ha consumido.
siguiente_dato       = 0 ;  // siguiente dato a producir en 'producir_dato' (solo se usa ahí)

int
vec[tam_vec],
primera_libre   = 0,  // índice de primera celda libre del vector. ++ al escribir.
primera_ocupada = 0;  // índice de primera celda ocupada del vector. ++ al leer.
// Al hacer el programa de modo FIFO(cola circular) hay que comprobar que
// no nos salimos del vector, para ello hacemos modulo tam_vec.

Semaphore
```

```
libres   = tam_vec, // num. entradas libres { k + #L + #E }
ocupadas = 0;       // num. entradas ocupadas { #E + #L }
//**********************************************************************
// funciones comunes a las dos soluciones (fifo y lifo)
//----------------------------------------------------------------------

unsigned producir_dato()
{
this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
const unsigned dato_producido = siguiente_dato ;
siguiente_dato++ ;
cont_prod[dato_producido] ++ ;
cout << "producido: " << dato_producido << endl << flush ;
return dato_producido ;
}
//----------------------------------------------------------------------

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

cout << "                consumido: " << dato << endl ;

}


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

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

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

void  funcion_hebra_productora(  )
{
for( unsigned i = 0 ; i < num_items ; i++ )
{
int dato = producir_dato() ;
// completar .....
sem_wait(libres);
vec[primera_libre] = dato;
cout << "\nInsertado '" << dato << "' en el vector. \n";
primera_libre = (primera_libre + 1) % tam_vec;   // cola circular.
sem_signal(ocupadas);
}
}

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

void funcion_hebra_consumidora(  )
{
for( unsigned i = 0 ; i < num_items ; i++ )
{
int dato ;
// completar ......
sem_wait(ocupadas);
dato = vec[primera_ocupada];
cout << "\n                Extraído '" << dato << "' del vector. \n";
primera_ocupada = (primera_ocupada + 1) % tam_vec;  // cola circular
sem_signal(libres);
consumir_dato( dato ) ;
}
}
//----------------------------------------------------------------------

int main()
{
```

```
cout << "--------------------------------------------------------------------" << endl
<< "Problema de los productores-consumidores (solución FIFO)." << endl
<< "--------------------------------------------------------------------" << endl
<< flush ;

thread hebra_productora ( funcion_hebra_productora ),
hebra_consumidora( funcion_hebra_consumidora );

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

cout << "\nFin.\n" << endl;

test_contadores();
}
```

2. Productor-Consumidor con buffer LIFO

```
// -----------------------------------------------------------------------------
// prodcons-LIFO.cpp
//
// Se completa el problema del productor-consumidor con un buffer para guardar
// los datos. Se realiza mediante LIFO(último en entrar es el primero en salir).
// -----------------------------------------------------------------------------

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

using namespace std ;
using namespace scd ;

//**********************************************************************
// Variables globales

const unsigned
num_items = 40 ,   // número de items
tam_vec   = 10 ;   // tamaño del buffer
unsigned
cont_prod[num_items] = {0}, // contadores de verificación: para cada dato, número de veces que se ha producido.
cont_cons[num_items] = {0}, // contadores de verificación: para cada dato, número de veces que se ha consumido.
siguiente_dato       = 0 ;  // siguiente dato a producir en 'producir_dato' (solo se usa ahí)

int
vec[tam_vec],
primera_libre = 0;  // primera celda libre del vector.

Semaphore
libres   = tam_vec, // num. entradas libres { k + #L + #E }
ocupadas = 0;       // num. entradas ocupadas { #E + #L }
```

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

unsigned producir_dato()
{
this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
const unsigned dato_producido = siguiente_dato ;
siguiente_dato++ ;
cont_prod[dato_producido] ++ ;
cout << "producido: " << dato_producido << endl << flush ;
return dato_producido ;
}
//----------------------------------------------------------------------

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

cout << "                consumido: " << dato << endl ;

}



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

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

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

void  funcion_hebra_productora(  )
{
for( unsigned i = 0 ; i < num_items ; i++ )
{
int dato = producir_dato() ;
// completar .....
sem_wait(libres);
vec[primera_libre] = dato;
primera_libre++;
cout << "\nInsertado "' << dato << "' en el vector. \n";
sem_signal(ocupadas);
}
}

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

void funcion_hebra_consumidora(  )
{
for( unsigned i = 0 ; i < num_items ; i++ )
{
int dato ;
// completar ......
sem_wait(ocupadas);
primera_libre--;
dato = vec[primera_libre];
cout << "\n                Extraído "' << dato << "' del vector. \n";
sem_signal(libres);
consumir_dato( dato ) ;
}
}
//----------------------------------------------------------------------

int main()
{
cout << "------------------------------------------------------------" << endl
<< "Problema de los productores-consumidores (solución LIFO)." << endl
```

```
<< "------------------------------------------------------------" << endl
<< flush ;

thread hebra_productora ( funcion_hebra_productora ),
hebra_consumidora( funcion_hebra_consumidora );

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

cout << "\nFin.\n" << endl;

test_contadores();
}
```

3. Múltiples Productores-Consumidores con buffer FIFO

```
// ---------------------------------------------------------------------
// prodcons-multi-FIFO.cpp
//
// Se completa el problema de los múltiples productores-consumidores con un
// buffer para guardar los datos. Se realiza mediante FIFO(primero en entrar es
// el primero en salir).
// ---------------------------------------------------------------------

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

using namespace std ;
using namespace scd ;

//**********************************************************************
// Variables globales

const unsigned
num_items   = 40 ,          // número de items
tam_vec     = 10 ,          // tamaño del buffer
np          = 4  ,          // número de hebras productoras
nc          = 4  ,          // número de hebras consumidoras
p           = num_items/np, // número de items a producir por cada hebra
c           = num_items/nc; // número de items a consumir por cada hebra
unsigned
cont_prod[num_items] = {0}, // contadores de verificación: para cada dato, número de veces que se ha producido.
cont_cons[num_items] = {0}, // contadores de verificación: para cada dato, número de veces que se ha consumido.
siguiente_dato       = 0  , // siguiente dato a producir en 'producir_dato' (solo se usa ahí)
vec[tam_vec]              , // vector para escribir/leer los datos.
primera_libre        = 0  , // primera celda libre del vector. ++ al escribir
primera_ocupada      = 0  , // índice de primera celda ocupada del vector. ++ al leer.
items_producidos[np] = {0}; // nº intems producidos por cada hebra productora.
```

```
Semaphore
libres   = tam_vec, // num. entradas libres { k + #L + #E }
ocupadas = 0;       // num. entradas ocupadas { #E + #L }

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

unsigned producir_dato(unsigned ind_p)
{
/*Cada hebra debe de producir todos los valores de un rango, que dicho rango
* vendrá dado por el total de items entre el número de hebras (se repartirá
* el trabajo). Cada hebra los calculará de forma CONTIGUA, como se vio en el
* Seminario 1.
* Por eso, en la constante dato_producido, en vez de ir de 1 en 1 como antes,
* ahora se cogerá el valor anterior, que es el que está en items_producidos[ind_p]
* (que empieza en 0, y se va incrementando de 1 en 1) y se le va sumando un
* valor contante que será el índice de la hebra productora por el número de
* items a producir por cada hebra
* Por ejemplo, como está ahora, si el indide de la hebra es 1, y hay 8 items,
* por lo que si hay dos hebras, cada una producira 4 items, p = 4, tenemos:
* (Iteración 1) = dato_producido = items_producidos[ind_p](0) + ind_p(1) + p(4) = 5;
*                item_producidos[ind_p]++;
* (Iteración 2) = dato_producido = items_producidos[ind_p](1) + ind_p(1) + p(4) = 6;
*                 item_producidos[ind_p]++;
* .
* .
* .
*/
assert (ind_p < np);   // Verifico que el índice de la hebra es menor que el núm. de hebras
productoras.
this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
const unsigned inicio = ind_p * p;  // valor inicial.
const unsigned dato_producido = items_producidos[ind_p] + inicio;
items_producidos[ind_p]++;
cont_prod[dato_producido] ++ ;
cout << "\nproducido: " << dato_producido << " por la hebra nº" << ind_p << endl << flush ;
return dato_producido;
}
//----------------------------------------------------------------------
```

```
void consumir_dato( unsigned dato, unsigned ind_c )
{
assert ( ind_c < nc);
assert( dato < num_items );
cont_cons[dato] ++ ;
this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
cout << "\n                consumido: " << dato << " por la hebra nº " << ind_c << endl ;
}


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

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

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

void  funcion_hebra_productora( int ind_p )
{

for( unsigned i = 0 ; i < p ; i++ ) // Condición: mientras que i no se supere el número
// máximo de items que tiene que producir cada hebra. Si lo supera, estaría
// calculando items que corresponden a otra hebra.
{
int dato = producir_dato(ind_p) ;
// completar .....
sem_wait(libres);
```

```
vec[primera_libre] = dato;
cout << "\nInsertado ''' << dato << ''' en el vector, por la hebra nº" << ind_p << "\n";
primera_libre = (primera_libre + 1) % tam_vec;
sem_signal(ocupadas);
}
}

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

void funcion_hebra_consumidora( unsigned ind_c )
{
for( unsigned i = 0 ; i < p ; i++ )
{
int dato ;
// completar .....
sem_wait(ocupadas);
dato = vec[primera_ocupada];
cout << "\n                Extraído ''' << dato << ''' del vector, por la hebra nº"
<< ind_c << "\n";
primera_ocupada = (primera_ocupada + 1) % tam_vec;
sem_signal(libres);
consumir_dato( dato, ind_c ) ;
}
}
//----------------------------------------------------------------------

int main()
{
cout << "----------------------------------------------------------------------" << endl
<< "Problema de los múltiples productores-consumidores (solución FIFO)." << endl
<< "----------------------------------------------------------------------" << endl
<< flush ;
thread hebras_productoras[np],
hebras_consumidoras[nc];
for(int i = 0; i < np; i++)  // Lanzo las hebras productoras.
hebras_productoras[i] = thread( funcion_hebra_productora, i );

for(int i = 0; i < nc; i++)  // Lanzo las hebras consumidoras.
hebras_consumidoras[i] = thread( funcion_hebra_consumidora, i);

for(int i = 0; i < np; i++)  // Finalizo las hebras productoras.
hebras_productoras[i].join();
```

```
for(int i = 0; i < nc; i++)  // Finalizo las hebras consumidoras.
hebras_consumidoras[i].join();

cout << "\nFin.\n" << endl;

test_contadores();
}
```

4. Múltiples Productores-Consumidores con buffer LIFO

```
// ---------------------------------------------------------------------
// prodcons-multi-LIFO.cpp
//
// Se completa el problema de los múltiples productores-consumidores con un
// buffer para guardar los datos. Se realiza mediante LIFO(último en entrar es
// el primero en salir).
// ---------------------------------------------------------------------

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

using namespace std ;
using namespace scd ;

//**********************************************************************
// Variables globales

const unsigned
num_items   = 40 ,          // número de items
tam_vec     = 10 ,          // tamaño del buffer
np          = 4  ,          // número de hebras productoras
nc          = 4  ,          // número de hebras consumidoras
p           = num_items/np, // número de items a producir por cada hebra
c           = num_items/nc; // número de items a consumir por cada hebra
unsigned
cont_prod[num_items] = {0}, // contadores de verificación: para cada dato, número de veces que se ha producido.
cont_cons[num_items] = {0}, // contadores de verificación: para cada dato, número de veces que se ha consumido.
siguiente_dato       = 0  , // siguiente dato a producir en 'producir_dato' (solo se usa ahí)
vec[tam_vec]              , // vector para escribir/leer los datos.
primera_libre = 0         , // primera celda libre del vector.
items_producidos[np] = {0}; // nº intems producidos por cada hebra productora.
```

```
Semaphore
libres   = tam_vec, // num. entradas libres { k + #L + #E }
ocupadas = 0;       // num. entradas ocupadas { #E + #L }

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

unsigned producir_dato(unsigned ind_p)
{
/*Cada hebra debe de producir todos los valores de un rango, que dicho rango
* vendrá dado por el total de items entre el número de hebras (se repartirá
* el trabajo). Cada hebra los calculará de forma CONTIGUA, como se vio en el
* Seminario 1.
* Por eso, en la constante dato_producido, en vez de ir de 1 en 1 como antes,
* ahora se cogerá el valor anterior, que es el que está en items_producidos[ind_p]
* (que empieza en 0, y se va incrementando de 1 en 1) y se le va sumando un
* valor contante que será el índice de la hebra productora por el número de
* items a producir por cada hebra
* Por ejemplo, como está ahora, si el indide de la hebra es 1, y hay 8 items,
* por lo que si hay dos hebras, cada una producira 4 items, p = 4, tenemos:
* (Iteración 1) = dato_producido = items_producidos[ind_p](0) + ind_p(1) + p(4) = 5;
*                 item_producidos[ind_p]++;
* (Iteración 2) = dato_producido = items_producidos[ind_p](1) + ind_p(1) + p(4) = 6;
*                  item_producidos[ind_p]++;
* .
* .
* .
*/
assert (ind_p < np);   // Verifico que el índice de la hebra es menor que el núm. de hebras
productoras.
this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
const unsigned inicio = ind_p * p;  // valor inicial.
const unsigned dato_producido = items_producidos[ind_p] + inicio;
items_producidos[ind_p]++;
cont_prod[dato_producido] ++ ;
cout << "\nproducido: " << dato_producido << " por la hebra nº" << ind_p << endl << flush ;
return dato_producido;
}
//------------------------------------------------------------------------

void consumir_dato( unsigned dato, unsigned ind_c )
```

```
{
assert ( ind_c < nc);
assert( dato < num_items );
cont_cons[dato] ++ ;
this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
cout << "\n                consumido: " << dato << " por la hebra nº " << ind_c << endl ;
}


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

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

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

void  funcion_hebra_productora( int ind_p )
{

for( unsigned i = 0 ; i < p ; i++ ) // Condición: mientras que i no se supere el número
// máximo de items que tiene que producir cada hebra. Si lo supera, estaría
// calculando items que corresponden a otra hebra.
{
int dato = producir_dato(ind_p) ;
// completar .....
sem_wait(libres);
vec[primera_libre] = dato;
```

```
cout << "\nInsertado ''' << dato << ''' en el vector, por la hebra nº" << ind_p << "\n";
primera_libre++;
sem_signal(ocupadas);
}
}

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

void funcion_hebra_consumidora( unsigned ind_c )
{
for( unsigned i = 0 ; i < p ; i++ )
{
int dato ;
// completar .....
sem_wait(ocupadas);
primera_libre--;
dato = vec[primera_libre];
cout << "\n                Extraído ''' << dato << ''' del vector, por la hebra nº"
<< ind_c << "\n";
sem_signal(libres);
consumir_dato( dato, ind_c ) ;
}
}
//----------------------------------------------------------------------

int main()
{
cout << "----------------------------------------------------------------------" << endl
<< "Problema de los múltiples productores-consumidores (solución LIFO)." << endl
<< "----------------------------------------------------------------------" << endl
<< flush ;
thread hebras_productoras[np],
hebras_consumidoras[nc];
for(int i = 0; i < np; i++)  // Lanzo las hebras productoras.
hebras_productoras[i] = thread( funcion_hebra_productora, i );

for(int i = 0; i < nc; i++)  // Lanzo las hebras consumidoras.
hebras_consumidoras[i] = thread( funcion_hebra_consumidora, i);

for(int i = 0; i < np; i++)  // Finalizo las hebras productoras.
hebras_productoras[i].join();
```

```
for(int i = 0; i < nc; i++)  // Finalizo las hebras consumidoras.
hebras_consumidoras[i].join();

cout << "\nFin.\n" << endl;

test_contadores();
}
```

5. Fumadores

```
// -----------------------------------------------------------------------------
// fumadores.cpp
// Se completa el problema de las hebras fumadores y la hebra estanquero.
// -----------------------------------------------------------------------------

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

using namespace std ;
using namespace scd ;

// numero de fumadores

const int num_fumadores = 3 ;

Semaphore
mostr_vacio = 1, // 1 si mostrador está vacío, 0 si no. {0 <= 1 + #Ri - #Pi}
ingr_disp[num_fumadores] = {0, 0, 0}; // 1 si ingrediente está disponible, 0 si no. {#Pi - #Ri}
/*
* Cada semáforo tiene un índice i, y cada índice se corresponde con un fumador i.
* Para cada i, el número de veces que el fumador i ha retirado el
* ingrediente i no puede ser mayor que el número de veces que se ha
* producido el ingrediente i, es decir #Ri  #Pi , o lo que es lo mismo:
* 0  #Pi  #Ri
*/

//----------------------------------------------------------------------
// Función que simula la acción de producir un ingrediente, como un retardo
// aleatorio de la hebra (devuelve número de ingrediente producido)

int producir_ingrediente()
{
// calcular milisegundos aleatorios de duración de la acción de fumar)
```

```
chrono::milliseconds duracion_produ( aleatorio<10,100>() );

// informa de que comienza a producir
cout << "Estanquero : empieza a producir ingrediente (" << duracion_produ.count() << "
milisegundos)" << endl;

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

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

// informa de que ha terminado de producir
cout << "Estanquero : termina de producir ingrediente " << num_ingrediente << endl;

return num_ingrediente ;
}

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

void funcion_hebra_estanquero(  )
{
while( true )
{
int i;
i = producir_ingrediente();
sem_wait(mostr_vacio);
cout << "\nEstanquero produce ingrediente '" << i << "'." << endl;
sem_signal(ingr_disp[i]);
}
}

//----------------------------------------------------------------------
// 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." << endl;

}

//----------------------------------------------------------------------
// función que ejecuta la hebra del fumador
void  funcion_hebra_fumador( int num_fumador )
{
while( true )
{
sem_wait(ingr_disp[num_fumador]);
cout << "\nFumador "' << num_fumador << "' retira su ingrediente del mostrador\n" <<endl;
sem_signal(mostr_vacio);
fumar(num_fumador);
}
}

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

int main()
{
thread hebra_estanquero,
hebras_fumadoras[num_fumadores];

hebra_estanquero = thread(funcion_hebra_estanquero);

for(int i = 0; i < num_fumadores; i++)
hebras_fumadoras[i] = thread(funcion_hebra_fumador, i);

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

```
}
```