TODOS LOS SEMINARIOS Y PRÁCTICAS DE SCD.

SISTEMAS CONCURRENTES Y DISTRIBUIDOS

2ºINGENIERÍA INFORMÁTICA

Este archivo contiene los siguientes seminarios/prácticas:

Seminario 1. Programación multihebra y semáforos.

1. Cálculo concurrente de una integral (de forma contigua)
2. Cálculo concurrente de una integral (de forma entrelazada)

Práctica 1. Sincronización de hebras con semáforos.

1. Productor-Consumidor con buffer FIFO
2. Productor-Consumidor con buffer LIFO
3. Múltiples Productores-Consumidores con buffer FIFO
4. Múltiples Productores-Consumidores con buffer LIFO
5. Fumadores.

Seminario 2. Introducción a los monitores en C++11.

1. monitor_em
2. Productor-Consumidor con monitor SU y buffer FIFO.

Práctica 2. Casos prácticos de monitores en C++11.

1. Múltiples Productores-Consumidores con monitores y buffer FIFO
2. Múltiples Productores-Consumidores con monitores y buffer LIFO.
3. Fumadores con monitores SU.
4. Lectores-Escritores con monitores.

Práctica 3. Implementación de algoritmos distribuidos con MPI.

1. Múltiples Productores-Consumidores con buffer FIFO por paso de mensajes.
2. Cena de filósofos (con interbloqueo).
3. Cena de filósofos con camarero.

Práctica 4. Implementación de Sistemas de Tiempo Real.

1. Modificando ejecutivo1.cpp.
2. Ejecutivo2.cpp

Seminario 1: Programación multihebra y semáforos.

1. Cálculo concurrente de una integral (de forma contigua)

```
// ---------------------------------------------------------------------
// Sistemas concurrentes y Distribuidos.
// Seminario 1. Programación Multihebra y Semáforos.
//
// Ejemplo 9 (ejemplo9.cpp)
// Calculo concurrente de una integral.
//
// Se completa el problema de la integral para calcular PI, de forma contigua.
// ---------------------------------------------------------------------
#include <iostream>
#include <iomanip>
#include <chrono>  // incluye now, time\_point, duration
#include <future>
#include <vector>
#include <cmath>

using namespace std ;
using namespace std::chrono;

const long m  = 1024l*1024l*1024l, // número de muestras (del orden de mil millones)
n  = 4  ;              // número de hebras concurrentes (divisor de 'm')


// ---------------------------------------------------------------------
// evalua la función $f$ a integrar ($f(x)=4/(1+x^2)$)
double f( double x )
{
return 4.0/(1.0+x*x) ;
}
// ---------------------------------------------------------------------
// calcula la integral de forma secuencial, devuelve resultado:
double calcular_integral_secuencial(  )
{
double suma = 0.0 ;                    // inicializar suma
for( long j = 0 ; j < m ; j++ )          // para cada $j$ entre $0$ y $m-1$:
{  const double xj = double(j+0.5)/m ;     //      calcular $x_j$
suma += f( xj );                    //      añadir $f(x_j)$ a la suma actual
}
```

```
return suma/m ;                         // devolver valor promedio de $f$
}

// ---------------------------------------------------------------------------
// función que ejecuta cada hebra: recibe $i$ ==índice de la hebra, ($0\leq i<n$)
double funcion_hebra( long i )
{
double suma = 0.0;
int puntos = m/n;    // Nº puntos que calcula cada hebra
//    if(i == 0){
//        for(int j = i; j < puntos; j++){
//            const double xj = double(j+0.5)/m;
//            suma += f( xj );
//        }
//   }else{
//        for(int j = (i-1)+ puntos; j < puntos+i; j++){
//            const double xj = double(j+0.5)/m;
//            suma += f( xj );
//   }
// Más eficiente:
int inicio = puntos*i;  //A partir de que punto comienza a calcular.
for(int j = inicio; j < puntos+inicio; j++){
const double xj = double(j+0.5)/m;
suma += f( xj );
}

return suma/m;
}

// ---------------------------------------------------------------------------
// calculo de la integral de forma concurrente
/*
El vector futuros debe estar declarado con el tipo de dato correcto.
En este caso, debería ser std::future<double> ya que funcion_hebra
devuelve un double.
*/
double calcular_integral_concurrente( )
{
double suma = 0.0;
future<double> futuros[n];
for (long i = 0; i < n; i++){
futuros[i] = async( launch::async, funcion_hebra, i);
```

```
}

for (int i = 0; i < n; i++)
suma += futuros[i].get();

return suma;

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

int main()
{

time_point<steady_clock> inicio_sec  = steady_clock::now() ;
const double             result_sec  = calcular_integral_secuencial(  );
time_point<steady_clock> fin_sec     = steady_clock::now() ;
double x = sin(0.4567);
time_point<steady_clock> inicio_conc = steady_clock::now() ;
const double             result_conc = calcular_integral_concurrente(  );
time_point<steady_clock> fin_conc    = steady_clock::now() ;
duration<float,milli>    tiempo_sec  = fin_sec  - inicio_sec ,
tiempo_conc = fin_conc - inicio_conc ;
const float              porc        = 100.0*tiempo_conc.count()/tiempo_sec.count() ;


constexpr long double pi = 3.14159265358979323846l ;

cout << "Número de muestras (m)   : " << m << endl
<< "Número de hebras (n)     : " << n << endl
<< setprecision(18)
<< "Valor de PI              : " << pi << endl
<< "Resultado secuencial     : " << result_sec  << endl
<< "Resultado concurrente    : " << result_conc << endl
<< setprecision(5)
<< "Tiempo secuencial        : " << tiempo_sec.count()  << " milisegundos. " << endl
<< "Tiempo concurrente       : " << tiempo_conc.count() << " milisegundos. " << endl
<< setprecision(4)
<< "Porcentaje t.conc/t.sec. : " << porc << "%" << endl;
}
```

Seminario 1: Programación multihebra y semáforos.

2. Cálculo concurrente de una integral (de forma entrelazada)

```
// ----------------------------------------------------------------------------
// Sistemas concurrentes y Distribuidos.
// Seminario 1. Programación Multihebra y Semáforos.
//
// Ejemplo 9 (ejemplo9.cpp)
// Calculo concurrente de una integral.
//
// Se completa el problema de la integral para calcular PI, de forma entrelazada.
// ----------------------------------------------------------------------------
#include <iostream>
#include <iomanip>
#include <chrono>  // incluye now, time\_point, duration
#include <future>
#include <vector>
#include <cmath>

using namespace std ;
using namespace std::chrono;

const long m  = 1024l*1024l*1024l, // número de muestras (del orden de mil millones)
n  = 4  ;              // número de hebras concurrentes (divisor de 'm')



// ----------------------------------------------------------------------------
// evalua la función $f$ a integrar ($f(x)=4/(1+x^2)$)
double f( double x )
{
return 4.0/(1.0+x*x) ;
}
// ----------------------------------------------------------------------------
// calcula la integral de forma secuencial, devuelve resultado:
double calcular_integral_secuencial(  )
{
double suma = 0.0 ;                    // inicializar suma
for( long j = 0 ; j < m ; j++ )          // para cada $j$ entre $0$ y $m-1$:
{  const double xj = double(j+0.5)/m ;     //      calcular $x_j$
suma += f( xj );                    //      añadir $f(x_j)$ a la suma actual
}
```

```
return suma/m ;                    // devolver valor promedio de $f$
}

// ---------------------------------------------------------------------
// función que ejecuta cada hebra: recibe $i$ ==índice de la hebra, ($0\leq i<n$)
double funcion_hebra( long i )
{
double suma = 0.0;
for(int j = i; j < m; j+=n){
const double xj = double(j+0.5)/m;
suma += f( xj );
}
return suma/m;
}

// ---------------------------------------------------------------------
// calculo de la integral de forma concurrente
/*
El vector futuros debe estar declarado con el tipo de dato correcto.
En este caso, debería ser std::future<double> ya que funcion_hebra
devuelve un double.
*/
double calcular_integral_concurrente( )
{
double suma = 0.0;
future<double> futuros[n];
for (long i = 0; i < n; i++){
futuros[i] = async( launch::async, funcion_hebra, i);
}

for (int i = 0; i < n; i++)
suma += futuros[i].get();

return suma;

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

int main()
{

time_point<steady_clock> inicio_sec  = steady_clock::now() ;
```

```
const double             result_sec  = calcular_integral_secuencial(  );
time_point<steady_clock> fin_sec     = steady_clock::now() ;
double x = sin(0.4567);
time_point<steady_clock> inicio_conc = steady_clock::now() ;
const double             result_conc = calcular_integral_concurrente(  );
time_point<steady_clock> fin_conc    = steady_clock::now() ;
duration<float,milli>    tiempo_sec  = fin_sec  - inicio_sec ,
tiempo_conc = fin_conc - inicio_conc ;
const float              porc        = 100.0*tiempo_conc.count()/tiempo_sec.count() ;


constexpr long double pi = 3.14159265358979323846l ;

cout << "Número de muestras (m)   : " << m << endl
<< "Número de hebras (n)     : " << n << endl
<< setprecision(18)
<< "Valor de PI              : " << pi << endl
<< "Resultado secuencial     : " << result_sec  << endl
<< "Resultado concurrente    : " << result_conc << endl
<< setprecision(5)
<< "Tiempo secuencial        : " << tiempo_sec.count()  << " milisegundos. " << endl
<< "Tiempo concurrente       : " << tiempo_conc.count() << " milisegundos. " << endl
<< setprecision(4)
<< "Porcentaje t.conc/t.sec. : " << porc << "%" << endl;
}
```

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();
```

```
}
```

Seminario 2. Introducción a los monitores en C++11.

1. monitor_em

```
// ---------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Seminario 2. Introducción a los monitores en C++11.
//
// archivo: monitor_em.cpp
// Ejemplo de monitores en C++11 sin variables condición
// (solo con encapsulamiento y exclusión mutua)
//
//  -- MContador1 : sin E.M., únicamente encapsulamiento
//  -- MContador2 : con E.M. mediante clase base 'HoareMonitor' y MRef
//
// Historial:
// Julio 2017: creado
// Sept 2022 : se quita MContador3 antiguo y se adapta MContador2 para usar HoareMonitor
// ---------------------------------------------------------------------------

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

using namespace std ;
using namespace scd ;

const int num_incrementos = 10000 ;

// ****************************************************************************
// clase contador, sin exclusión mutua

class MContador1
{
private:
int cont ;

public:
```

```
MContador1( int valor_ini ) ;
void incrementa();
int leer_valor();
} ;
// ----------------------------------------------------------------------------

MContador1::MContador1( int valor_ini )
{
cont = valor_ini ;
}
// ----------------------------------------------------------------------------

void MContador1::incrementa()
{
cont ++ ;
}
// ----------------------------------------------------------------------------

int MContador1::leer_valor()
{
return cont ;
}

// *****************************************************************************
// clase contador, con exclusión mutua mediante herencia de 'HoareMonitor'

class MContador2 : public HoareMonitor
{
private:
int cont ;

public:
MContador2( int valor_ini ) ;
void incrementa();
int leer_valor();
} ;

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

MContador2::MContador2( int valor_ini )
{
cont = valor_ini ;
```

```
}
// -----------------------------------------------------------------------

void MContador2::incrementa()
{
cont ++ ;
}
// -----------------------------------------------------------------------

int MContador2::leer_valor()
{
return cont ;
}

// *****************************************************************************

void funcion_hebra_M1( MContador1 & monitor )
{
for( int i = 0 ; i < num_incrementos ; i++ )
monitor.incrementa();
}
// -----------------------------------------------------------------------

void test_1()
{
MContador1 monitor(0) ;

thread hebra1( funcion_hebra_M1, ref(monitor) ),
hebra2( funcion_hebra_M1, ref(monitor) );

hebra1.join();
hebra2.join();

cout  << "Monitor contador (sin exclusión mutua):" << endl
<< endl
<< "  valor esperado == " << 2*num_incrementos << endl
<< "  valor obtenido == " << monitor.leer_valor() << endl
<< endl;
}
// *****************************************************************************

void funcion_hebra_M2( MRef<MContador2>  monitor )
```

```
{
for( int i = 0 ; i < num_incrementos ; i++ )
monitor->incrementa();
}
// -----------------------------------------------------------------------------

void test_2()
{
MRef<MContador2> monitor = Create<MContador2>(0) ;

thread hebra1( funcion_hebra_M2, monitor ),
hebra2( funcion_hebra_M2, monitor );

hebra1.join();
hebra2.join();

cout  << "Monitor contador (EM usando clase derivada de HoareMonitor):" << endl
<< endl
<< "  valor esperado == " << 2*num_incrementos << endl
<< "  valor obtenido == " << monitor->leer_valor() << endl
<< endl ;
}
// *****************************************************************************

int main()
{
test_1();
test_2();
}
```

Seminario 2: Introducción a los monitores en C++11.

2. Productor-Consumidor con monitor SU y buffer FIFO

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Seminario 2. Introducción a los monitores en C++11.
//
// Archivo: prodcons1_su.cpp
//
// Ejemplo de un monitor en C++11 con semántica SU, para el problema
// del productor/consumidor, con productor y consumidor únicos.
// Opcion LIFO
//
// Historial:
// Creado el 30 Sept de 2022. (adaptado de prodcons2_su.cpp)
// 20 oct 22 --> paso este archivo de FIFO a LIFO, para que se corresponda con lo que dicen las transparencias
// -----------------------------------------------------------------------------


// Se realiza mediante FIFO(primero en entrar es el primero en salir).
// -----------------------------------------------------------------------------


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

using namespace std ;
using namespace scd ;

constexpr int
num_items = 15 ;   // número de items a producir/consumir
int
siguiente_dato = 0 ; // siguiente valor a devolver en 'producir_dato'

constexpr int
```

```
min_ms    = 5,     // tiempo minimo de espera en sleep_for
max_ms    = 20 ;   // tiempo máximo de espera en sleep_for


mutex
mtx ;                 // mutex de escritura en pantalla
unsigned
cont_prod[num_items] = {0}, // contadores de verificación: producidos
cont_cons[num_items] = {0}; // contadores de verificación: consumidos

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

int producir_dato(  )
{

this_thread::sleep_for( chrono::milliseconds( aleatorio<min_ms,max_ms>() ));
const int valor_producido = siguiente_dato ;
siguiente_dato ++ ;
mtx.lock();
cout << "hebra productora, produce " << valor_producido << endl << flush ;
mtx.unlock();
cont_prod[valor_producido]++ ;
return valor_producido ;
}
//----------------------------------------------------------------------

void consumir_dato( unsigned valor_consumir )
{
if ( num_items <= valor_consumir )
{
cout << " valor a consumir === " << valor_consumir << ", num_items == " << num_items <<
endl ;
assert( valor_consumir < num_items );
}
cont_cons[valor_consumir] ++ ;
this_thread::sleep_for( chrono::milliseconds( aleatorio<min_ms,max_ms>() ));
mtx.lock();
cout << "                hebra consumidora, consume: " << valor_consumir << endl ;
mtx.unlock();
}
```

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

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

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 SC, multiples prod/cons

class ProdConsSU1 : public HoareMonitor
{
private:
static const int           // constantes ('static' ya que no dependen de la instancia)
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
primera_ocupada         ,//   índice de celda de la próxima extracción
n                      ;//   número de celdas ocupadas.

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
ProdConsSU1() ;         // constructor
int  leer();            // extraer un valor (sentencia L) (consumidor)
void escribir( int valor ); // insertar un valor (sentencia E) (productor)
} ;
// -----------------------------------------------------------------------

ProdConsSU1::ProdConsSU1(  )
{
primera_libre    = 0 ;
primera_ocupada  = 0;
n              = 0;
ocupadas       = newCondVar();
libres         = newCondVar();
}
// -----------------------------------------------------------------------
// función llamada por el consumidor para extraer un dato

int ProdConsSU1::leer(  )
{
// esperar bloqueado hasta que 0 < primera_libre
if ( n == 0 )    // Si el número de celdas ocupadas es 0 .....
ocupadas.wait();

cout << "leer: ocup == " << n << ", total == " << num_celdas_total << endl ;
assert( 0 < n  );

// hacer la operación de lectura, actualizando estado del monitor
const int valor = buffer[primera_ocupada] ;
n--;
primera_ocupada = (primera_ocupada + 1) % num_celdas_total;

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

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

void ProdConsSU1::escribir( int valor )
{
```

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

cout << "escribir: ocup == " << n << ", total == " << num_celdas_total << endl ;
assert( primera_libre < num_celdas_total );

// hacer la operación de inserción, actualizando estado del monitor
buffer[primera_libre] = valor ;
primera_libre = (primera_libre + 1) % num_celdas_total ;
n++;

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

void funcion_hebra_productora( MRef<ProdConsSU1> monitor )
{
for( unsigned i = 0 ; i < num_items ; i++ )
{
int valor = producir_dato(  ) ;
monitor->escribir( valor );
}
}
// -----------------------------------------------------------------------------

void funcion_hebra_consumidora( MRef<ProdConsSU1>  monitor )
{
for( unsigned i = 0 ; i < num_items ; i++ )
{
int valor = monitor->leer();
consumir_dato( valor ) ;
}
}
// -----------------------------------------------------------------------------

int main()
{
cout << "------------------------------------------------------------" << endl
<< "Problema del productor-consumidor únicos (Monitor SU, buffer LIFO). " << endl
```

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

// crear monitor  ('monitor' es una referencia al mismo, de tipo MRef<...>)
MRef<ProdConsSU1> monitor = Create<ProdConsSU1>() ;

// crear y lanzar las hebras
thread hebra_prod( funcion_hebra_productora, monitor ),
hebra_cons( funcion_hebra_consumidora, monitor );

// esperar a que terminen las hebras
hebra_prod.join();
hebra_cons.join();

test_contadores() ;
}
```

Práctica 2. Casos prácticos de monitores en C++11.

1. Múltiples Productores-Consumidores con monitores y buffer FIFO.

```
// -----------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Seminario 2. Introducción a los monitores en C++11.
//
// Archivo: prodcons1_mu_FIFO.cpp
//
// Ejemplo de un monitor en C++11 con semántica SU, para el problema
// del productor/consumidor, con productor y consumidor únicos.
//
//
// Se realiza el problema de los múltiples productores consumidores con monitores.
// Se hace con FIFO(primero en entrar es el primero en salir).
// -----------------------------------------------------------------------


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

using namespace std ;
using namespace scd ;

constexpr int
num_items = 15 ,   // número de items a producir/consumir.
np        = 5  ,   // número de hebras productoras.
nc        = 5  ;   // número de hebras consumidoras.
int
siguiente_dato = 0 , // siguiente valor a devolver en 'producir_dato'
p = num_items/np,    // número de items por cada hebra productora.
c = num_items/nc,    // número de items por cada hebra consumidora.
items_producidos[np] = {0};     // contador de items producidos por cada hebra

constexpr int
min_ms    = 5,     // tiempo minimo de espera en sleep_for
```

```
max_ms    = 20 ;   // tiempo máximo de espera en sleep_for


mutex
mtx ;                // mutex de escritura en pantalla
unsigned
cont_prod[num_items] = {0}, // contadores de verificación: producidos
cont_cons[num_items] = {0}; // contadores de verificación: consumidos

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

int producir_dato(int ih)
{

assert (ih < np);   // Verifico que el índice de la hebra es menor que el núm. de hebras
productoras.
this_thread::sleep_for( chrono::milliseconds( aleatorio<min_ms,max_ms>() ));
const int inicio = ih*p;
const int valor_producido = items_producidos[ih] + inicio;
items_producidos[ih]++;
mtx.lock();
cout << "hebra productora nº" << ih << " produce " << valor_producido << endl << flush ;
mtx.unlock();
cont_prod[valor_producido]++ ;
return valor_producido ;
}
//----------------------------------------------------------------------

void consumir_dato( unsigned valor_consumir, int ih)
{
assert(ih < nc);
if ( num_items <= valor_consumir )
{
cout << " valor a consumir === " << valor_consumir << ", num_items == " << num_items <<
endl ;
assert( valor_consumir < num_items );
}
cont_cons[valor_consumir] ++ ;
this_thread::sleep_for( chrono::milliseconds( aleatorio<min_ms,max_ms>() ));
mtx.lock();
```

```
cout << "                hebra consumidora nº"' << ih << "' consume: " << valor_consumir << endl
;
mtx.unlock();
}
//----------------------------------------------------------------------

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

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 SC, multiples prod/cons

class ProdConsMu : public HoareMonitor
{
private:
static const int           // constantes ('static' ya que no dependen de la instancia)
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
primera_ocupada,         //   indice de celda de la próxima extracción.
n;                       //   ( == número de celdas ocupadas)
```

```
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
ProdConsMu() ;           // constructor
int  leer();             // extraer un valor (sentencia L) (consumidor)
void escribir( int valor ); // insertar un valor (sentencia E) (productor)
} ;
// -----------------------------------------------------------------------------

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

int ProdConsMu::leer(  )
{
// esperar bloqueado hasta que 0 < primera_libre
if ( n == 0 )
ocupadas.wait();

cout << "leer: ocup == " << n << ", total == " << num_celdas_total << endl ;
assert( 0 < n  );

// hacer la operación de lectura, actualizando estado del monitor
const int valor = buffer[primera_ocupada] ;
n--;
primera_ocupada = (primera_ocupada+1) % num_celdas_total;

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

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

```
void ProdConsMu::escribir( int valor )
{
// esperar bloqueado hasta que primera_libre < num_celdas_total
if ( primera_libre == num_celdas_total )
libres.wait();

cout << "escribir: ocup == " << n << ", total == " << num_celdas_total << endl ;
assert( primera_libre < num_celdas_total );

// hacer la operación de inserción, actualizando estado del monitor
buffer[primera_libre] = valor ;
primera_libre = (primera_libre+1) % num_celdas_total ;
n++;

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

void funcion_hebra_productora( MRef<ProdConsMu> monitor, int ih )
{
for( unsigned i = ih*p; i < ((ih*p)+p) ; i++ )
{
int valor = producir_dato(ih) ;
monitor->escribir( valor );
}
}
// ----------------------------------------------------------------------------

void funcion_hebra_consumidora( MRef<ProdConsMu>  monitor, int ih )
{
for( unsigned i = ih*c; i < ((ih*c)+c) ; i++ )
{
int valor = monitor->leer();
consumir_dato( valor, ih ) ;
}
}
// ----------------------------------------------------------------------------

int main()
{
```

```
cout << "------------------------------------------------------------------" << endl
<< "Problema del productor-consumidor únicos (Monitor SU, buffer FIFO). " << endl
<< "------------------------------------------------------------------" << endl
<< flush ;

// crear monitor  ('monitor' es una referencia al mismo, de tipo MRef<...>)
MRef<ProdConsMu> monitor = Create<ProdConsMu>() ;



// crear y lanzar las hebras
thread hebras_prod[np],
hebras_cons[nc];
for(int i = 0; i < np; i++)
hebras_prod[i] = thread( funcion_hebra_productora, monitor, i);

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

// esperar a que terminen las hebras
for(int i = 0; i < np; i++)
hebras_prod[i].join();

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

test_contadores() ;
}
```

Práctica 2. Casos prácticos de monitores en C++11.

2. Múltiples Productores-Consumidores con monitores y buffer LIFO.

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos
//
//
// Archivo: prodcons1_mu-LIFO.cpp
//
// Ejemplo de un monitor en C++11 con semántica SU, para el problema
// del productor/consumidor, con productor y consumidor únicos.
// Opcion LIFO
//
// Se realiza el problema de los múltiples productores consumidores con monitores.
// Se hace con LIFO(último en entrar es el primero en salir).
// -----------------------------------------------------------------------------


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

using namespace std ;
using namespace scd ;

constexpr int
num_items = 15 ,   // número de items a producir/consumir.
np        = 5  ,   // número de hebras productoras.
nc        = 5  ;   // número de hebras consumidoras.
int
siguiente_dato = 0 , // siguiente valor a devolver en 'producir_dato'
p = num_items/np,    // número de items por cada hebra productora.
c = num_items/nc,    // número de items por cada hebra consumidora.
items_producidos[np] = {0};     // contador de items producidos por cada hebra

constexpr int
min_ms    = 5,     // tiempo minimo de espera en sleep_for
```

```
max_ms    = 20 ;   // tiempo máximo de espera en sleep_for


mutex
mtx ;                // mutex de escritura en pantalla
unsigned
cont_prod[num_items] = {0}, // contadores de verificación: producidos
cont_cons[num_items] = {0}; // contadores de verificación: consumidos

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

int producir_dato(int ih)
{

assert (ih < np);   // Verifico que el índice de la hebra es menor que el núm. de hebras
productoras.
this_thread::sleep_for( chrono::milliseconds( aleatorio<min_ms,max_ms>() ));
const int inicio = ih*p;
const int valor_producido = items_producidos[ih] + inicio;
items_producidos[ih]++;
mtx.lock();
cout << "hebra productora nº" << ih << " produce " << valor_producido << endl << flush ;
mtx.unlock();
cont_prod[valor_producido]++ ;
return valor_producido ;
}
//----------------------------------------------------------------------

void consumir_dato( unsigned valor_consumir, int ih)
{
assert(ih < nc);
if ( num_items <= valor_consumir )
{
cout << " valor a consumir === " << valor_consumir << ", num_items == " << num_items <<
endl ;
assert( valor_consumir < num_items );
}
cont_cons[valor_consumir] ++ ;
this_thread::sleep_for( chrono::milliseconds( aleatorio<min_ms,max_ms>() ));
mtx.lock();
```

```
cout << "                hebra consumidora nº"' << ih << "' consume: " << valor_consumir << endl
;
mtx.unlock();
}
//----------------------------------------------------------------------

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

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 SC, multiples prod/cons

class ProdConsMu : public HoareMonitor
{
private:
static const int           // constantes ('static' ya que no dependen de la instancia)
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 ( == número de celdas
ocupadas)

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
ProdConsMu() ;          // constructor
int  leer();            // extraer un valor (sentencia L) (consumidor)
void escribir( int valor ); // insertar un valor (sentencia E) (productor)
} ;
// -----------------------------------------------------------------------------

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

int ProdConsMu::leer(  )
{
// esperar bloqueado hasta que 0 < primera_libre
if ( primera_libre == 0 )
ocupadas.wait();

cout << "leer: ocup == " << primera_libre << ", total == " << num_celdas_total << endl ;
assert( 0 < primera_libre  );

// hacer la operación de lectura, actualizando estado del monitor
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 ProdConsMu::escribir( int valor )
{
```

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

cout << "escribir: ocup == " << primera_libre << ", total == " << num_celdas_total << endl ;
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<ProdConsMu> monitor, int ih )
{
for( unsigned i = ih*p; i < ((ih*p)+p) ; i++ )
{
int valor = producir_dato(ih) ;
monitor->escribir( valor );
}
}
// ----------------------------------------------------------------------------

void funcion_hebra_consumidora( MRef<ProdConsMu>  monitor, int ih )
{
for( unsigned i = ih*c; i < ((ih*c)+c) ; i++ )
{
int valor = monitor->leer();
consumir_dato( valor, ih ) ;
}
}
// ----------------------------------------------------------------------------

int main()
{
cout << "--------------------------------------------------------------------" << endl
<< "Problema del productor-consumidor únicos (Monitor SU, buffer LIFO). " << endl
<< "--------------------------------------------------------------------" << endl
```

```
<< flush ;

// crear monitor  ('monitor' es una referencia al mismo, de tipo MRef<...>)
MRef<ProdConsMu> monitor = Create<ProdConsMu>() ;



// crear y lanzar las hebras
thread hebras_prod[np],
hebras_cons[nc];
for(int i = 0; i < np; i++)
hebras_prod[i] = thread( funcion_hebra_productora, monitor, i);

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

// esperar a que terminen las hebras
for(int i = 0; i < np; i++)
hebras_prod[i].join();

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

test_contadores() ;
}
```

Práctica 2. Casos prácticos de monitores en C++11.

3. Fumadores con monitores SU.

```
// -----------------------------------------------------------------------------
// fumadores-monitores.cpp
// FECHA: 13/11/23
// 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 ;

//---------------------------------------------------------------------------
// 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 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;

}

//----------------------------------------------------------------------
// clase para monitor buffer, version FIFO, semántica SU, fumadores.
class Estanco : public HoareMonitor
{
private:
int                    // variables permanentes
ingre_mostrador;       // número de ingrediente que hay en el mostrador
CondVar                 // colas condicion:
mostr_vacio,             //  cola donde espera el estanquero.
ingr_disp[num_fumadores];   //  cola donde esperan los fumadores.
```

```
public:                 // constructor y métodos públicos
Estanco() ;             // constructor
void obtenerIngrediente(int ingre);
void ponerIngrediente(int ingre);
void esperarRecogidaIngrediente();
} ;

//------------------------------------------------------------------------
Estanco::Estanco(){
ingre_mostrador = -1;
for(int i = 0; i < num_fumadores; i++)
ingr_disp[i] = newCondVar();

mostr_vacio = newCondVar();
}

//------------------------------------------------------------------------
void Estanco::obtenerIngrediente(int ingre){
if(ingre_mostrador != ingre)
ingr_disp[ingre].wait();
ingre_mostrador = -1;
cout << "\nFumador '" << ingre << "' retira su ingrediente del mostrador\n" <<endl;
mostr_vacio.signal();
}

//------------------------------------------------------------------------
void Estanco::ponerIngrediente(int ingre){
if(ingre_mostrador != -1) // si el mostrador no está vacio...
mostr_vacio.wait();
ingre_mostrador = ingre;
cout << "\nEstanquero produce ingrediente '" << ingre << "'." << endl;
ingr_disp[ingre].signal();
}

//------------------------------------------------------------------------
void Estanco::esperarRecogidaIngrediente(){
if(ingre_mostrador != -1) // si el mostrador no está vacio...
mostr_vacio.wait();
}

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

```
void  funcion_hebra_fumador( MRef<Estanco> monitor, int num_fumador )
{
while( true )
{
monitor->obtenerIngrediente(num_fumador);
cout << "\nFumador '" << num_fumador << "' retira su ingrediente del mostrador\n" <<endl;
fumar(num_fumador);
}
}

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

void funcion_hebra_estanquero( MRef<Estanco> monitor )
{
while( true )
{
int ingre = producir_ingrediente();
monitor->ponerIngrediente(ingre);
monitor->esperarRecogidaIngrediente();
}
}

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

int main()
{
cout << "----------------------------------------------------------------------" << endl
<< "        Problema del los fumadores (Monitor SU).                    " << endl
<< "----------------------------------------------------------------------" << endl
<< flush ;

// crear monitor  ('monitor' es una referencia al mismo, de tipo MRef<...>)
MRef<Estanco> monitor = Create<Estanco>() ;

thread hebra_estanquero,
hebras_fumadoras[num_fumadores];

hebra_estanquero = thread(funcion_hebra_estanquero, monitor);

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

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

Práctica 2. Casos prácticos de monitores en C++11.

4. Lectores-Escritores con monitores

```
// -----------------------------------------------------------------------
// Archivo lec-esc.cpp
// Se completa el problema de los lectores-escritores con monitores.
// -----------------------------------------------------------------------

#include <iostream>
#include <iomanip>
#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 ;

const unsigned
ne = 3,         // número de hebras escritoras.
nl = 2;         // número de hebras lectoras.
int valor = 0;              // para ir mostrando en pantalla el valor

mutex mtx;                  // mutex para la escritura en pantalla.
//----------------------------------------------------------------------
// clase para monitor, semántica SU, lectores_escritores.

class LecEsc : public HoareMonitor{

private:
int n_lec;      // { número de lectores leyendo }
bool escrib;    // { true si hay algún escritor escribiendo }

CondVar         //  colas condicion:
lectura,    //  cola donde esperan los lectores cuando hay un escritor escribiendo.
escritura;  //  cola donde esperan los escritores cuando otros están escribiendo ->
//  -> escrib == true o n_lec > 0 {no hay lec. ni esc. escritura posible}
public:
LecEsc();
```

```
// { invocados por lectores }
void ini_lectura();
void fin_lectura();

//{ invocados por escritores }
void ini_escritura();
void fin_escritura();
};

//---------------------------------------------------------------------
LecEsc::LecEsc(){
n_lec = 0;
escrib = false;
lectura = newCondVar();
escritura = newCondVar();
}

//---------------------------------------------------------------------
void LecEsc::ini_lectura(){
if(escrib)  // si hay escritor.
lectura.wait();
n_lec++;    // registrar un lector más.
cout << "\n Lector comienza a leer, número de lectores: " << n_lec << endl;
lectura.signal();   // desbloqueo en cadena los posibles lectores bloqueados.
}

//---------------------------------------------------------------------
void LecEsc::fin_lectura(){
n_lec--;    // registro un lector menos.
if(n_lec == 0)          // si es el último lector.
escritura.wait();   // desbloqueo un  escritor.
}

//---------------------------------------------------------------------
void LecEsc::ini_escritura(){
if(n_lec > 0 || escrib) // si hay otros, esperar.
escritura.wait();
escrib = true;          // registrar que hay un escritor.
}

//---------------------------------------------------------------------
void LecEsc::fin_escritura(){
```

```
escrib = false;     // registrar que ya no hay escritor
if(!lectura.empty()) // si hay lectores, despertar uno
lectura.signal();
else                 // si no hay, despertar un escritor
escritura.signal();
}

//----------------------------------------------------------------------
//función que ejecutan las hebras escitoras
void funcion_hebra_escritora(MRef<LecEsc> monitor, int ih){
while( true ){
monitor->ini_escritura();   // se inicia la escritura.
valor++;
mtx.lock();
cout << "\nHebra "' << ih << "' escribe "' << valor << "'." << endl;
mtx.unlock();
this_thread::sleep_for((chrono::milliseconds) aleatorio<100,250>() ); // retraso aleatorio
simulando escritura
monitor->fin_escritura();   // se finaliza la escritura
this_thread::sleep_for((chrono::milliseconds) aleatorio<30,100>() ); // retraso aleatorio
}
}

//----------------------------------------------------------------------
//función que ejecutan las hebras lectoras
void funcion_hebra_lectora(MRef<LecEsc> monitor, int ih){
while( true ){
monitor->ini_lectura();   // se inicia la lectura.
mtx.lock();
cout << "\nHebra "' << ih << "' lee "' << valor << "'." << endl;
mtx.unlock();
this_thread::sleep_for((chrono::milliseconds) aleatorio<100,250>() );// retraso aleatorio
simulando lectura
monitor->fin_lectura();   // se finaliza la lectura
this_thread::sleep_for((chrono::milliseconds) aleatorio<30,100>() ); // retraso aleatorio
}
}

//----------------------------------------------------------------------
int main(){
cout << "----------------------------------------------------------------------" << endl
<< "        Problema del los lectores-escritores (Monitor SU).          " << endl
```

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

// creo el monitor
MRef<LecEsc> monitor = Create<LecEsc>();

// creo y lanzo las hebras.
thread hebra_escritora[ne], hebra_lectora[nl];

for(int i = 0; i < ne; i++)
hebra_escritora[i] = thread(funcion_hebra_escritora, monitor, i);

for(int i = 0; i < nl; i++)
hebra_lectora[i] = thread(funcion_hebra_lectora, monitor, i);

// espero a que terminen las hebras.
for(int i = 0; i < ne; i++)
hebra_escritora[i].join();

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

Práctica 3. Implementación de algoritmos distribuidos con MPI.

1. Múltiples Productores-Consumidores con buffer FIFO por paso de mensajes.

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 3. Implementación de algoritmos distribuidos con MPI
//
// Archivo: prodcons2-mu.cpp
// Implementación del problema del productor-consumidor con
// un proceso intermedio que gestiona un buffer finito y recibe peticiones
// en orden arbitrario
//
// Se realiza el problema de los múltiples productores consumidores con paso de
// mensajes MPI. Se hace con FIFO (primero en entrar es el primero en salir).
// -----------------------------------------------------------------------------

#include <iostream>
#include <thread> // this_thread::sleep_for
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include <mpi.h>

using namespace std;
using namespace std::this_thread ;
using namespace std::chrono ;

const int
tam_vector           = 10,
// Añadido a la plantilla:
np = 4,              // num. de prods.
nc = 5,              // num. de cons.
m = 20,              // num. de items a producir/consumir
id_min_prod = 0,     // id del primer prod.
id_max_prod = np - 1,   // id del ultimo prod.
id_buffer = np,      // id del buffer.
id_min_cons = np + 1,   // id del primer cons.
id_max_cons = np + nc,  // id del ultimo cons.
num_procesos_esperado = np + nc + 1, // prod + cons + buffer
k = m/np,            // num. de items a producir por cada prod
c = m/nc,            // num. de items a consumir por cada cons
```

```
etiq_prod = 0,          // etiqueta para los mensajes de los prods.
etiq_cons = 1;          // etiqueta para los mensajes de los cons.

//**********************************************************************
// 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 );
}
// ---------------------------------------------------------------------
// producir produce los numeros en funcion del numero de orden del productor
// y lleva espera aleatorio
int producir(int orden_prod)
{
static int
contador[np] = {0} ;
int
inicio = orden_prod * k,
valor_producido = 0;
sleep_for( milliseconds( aleatorio<10,100>()) );
contador[orden_prod]++ ;
valor_producido = inicio + contador[orden_prod];
cout << "Productor " << orden_prod << " ha producido valor " << valor_producido
<< endl << flush;
return valor_producido;
}
// ---------------------------------------------------------------------

void funcion_productor(int orden_prod)
{
for ( unsigned int i= 0 ; i < k ; i++ )
// Poniendo esta condicion, la funcion producir nunca llegara a producir > ik + k -1
{
// producir valor
int valor_prod = producir(orden_prod);
// enviar valor
```

```
cout << "Productor " << orden_prod << " va a enviar valor " << valor_prod
<< endl << flush;
MPI_Ssend( &valor_prod, 1, MPI_INT, id_buffer, etiq_prod, MPI_COMM_WORLD );
}
}
// ----------------------------------------------------------------------

void consumir( int valor_cons, int orden_cons )
{
// espera bloqueada
sleep_for( milliseconds( aleatorio<110,200>()) );
cout << "Consumidor " << orden_cons << " ha consumido valor " << valor_cons << endl <<
flush ;
}
// ----------------------------------------------------------------------

void funcion_consumidor(int orden_cons)
{
int         peticion,
valor_rec = 1 ;
MPI_Status  estado ;

for( unsigned int i=0 ; i < c; i++ )
{
MPI_Ssend( &peticion,  1, MPI_INT, id_buffer, etiq_cons, MPI_COMM_WORLD);
MPI_Recv ( &valor_rec, 1, MPI_INT, id_buffer, etiq_cons, MPI_COMM_WORLD,&estado );
cout << "Consumidor " << orden_cons << " ha recibido valor " << valor_rec << endl << flush ;
consumir( valor_rec, orden_cons );
}
}
// ----------------------------------------------------------------------

void funcion_buffer()
{
int        buffer[tam_vector],      // buffer con celdas ocupadas y vacías
valor,                  // valor recibido o enviado
primera_libre       = 0, // índice de primera celda libre
primera_ocupada     = 0, // índice de primera celda ocupada
num_celdas_ocupadas = 0, // número de celdas ocupadas
etiq_aceptable ;         // etiqueta del emisor aceptable
MPI_Status estado ;                // metadatos del mensaje recibido
```

```
for( unsigned int i=0 ; i < m*2 ; i++ )
{
// 1. determinar si puede enviar solo prod., solo cons, o todos

if ( num_celdas_ocupadas == 0 )              // si buffer vacío
etiq_aceptable = etiq_prod ;              // $~~~$ solo prod.
else if ( num_celdas_ocupadas == tam_vector ) // si buffer lleno
etiq_aceptable = etiq_cons ;              // $~~~$ solo cons.
else                                      // si no vacío ni lleno
etiq_aceptable = MPI_ANY_TAG ;            // $~~~$ cualquiera

// 2. recibir un mensaje del emisor o emisores aceptables

MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_aceptable, MPI_COMM_WORLD,
&estado );

// 3. procesar el mensaje recibido

switch( estado.MPI_TAG ) // leer emisor del mensaje en metadatos
{
case etiq_prod: // si ha sido el productor: insertar en buffer
buffer[primera_libre] = valor ;
primera_libre = (primera_libre+1) % tam_vector ;
num_celdas_ocupadas++ ;
cout << "Buffer ha recibido valor " << valor << endl ;
break;

case etiq_cons: // si ha sido el consumidor: extraer y enviarle
valor = buffer[primera_ocupada] ;
primera_ocupada = (primera_ocupada+1) % tam_vector ;
num_celdas_ocupadas-- ;
cout << "Buffer va a enviar valor " << valor << endl ;
MPI_Ssend( &valor, 1, MPI_INT, estado.MPI_SOURCE, etiq_cons, MPI_COMM_WORLD);
break;
}
}
}

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

int main( int argc, char *argv[] )
{
```

```
int id_propio, num_procesos_actual;

// inicializar MPI, leer identif. de proceso y número de procesos
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );

if ( num_procesos_esperado == num_procesos_actual )
{
// ejecutar la operación apropiada a 'id_propio'
if ( id_propio >= id_min_prod && id_propio <= id_max_prod )
funcion_productor(id_propio);
else if ( id_propio == id_buffer )
funcion_buffer();
else
funcion_consumidor(id_propio - id_min_cons);
}
else
{
if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
{ cout << "el número de procesos esperados es:    " << num_procesos_esperado << endl
<< "el número de procesos en ejecución es: " << num_procesos_actual << endl
<< "(programa abortado)" << endl ;
}
}

// al terminar el proceso, finalizar MPI
MPI_Finalize( );
return 0;
}
```

Práctica 3. Implementación de algoritmos distribuidos con MPI.

2. Cena de filósofos (con interbloqueo)

```
// -----------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 3. Implementación de algoritmos distribuidos con MPI
//
// Archivo: filosofos-interb.cpp
// Implementación del problema de los filósofos (sin camarero).
// Plantilla para completar.
//
// Implementar una solución distribuida al problema de los filósofos de acuerdo
// con el esquema descrito en las plantillas. Usar la operación síncrona de envío MPI_Ssend.
// -----------------------------------------------------------------------

#include <mpi.h>
#include <thread> // this_thread::sleep_for
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include <iostream>

using namespace std;
using namespace std::this_thread ;
using namespace std::chrono ;

const int
num_filosofos = 5 ,              // número de filósofos
num_filo_ten  = 2*num_filosofos, // número de filósofos y tenedores
num_procesos  = num_filo_ten ,   // número de procesos total (por ahora solo hay filo y ten)
// Añadido a la plantilla:
etiq_fi_coge = 0,                // etiqueta de que el filosofo coge el tenedor
etiq_fi_suelta = 1;              // etiqueta de que el filosofo suelta el tenedor


//**********************************************************************
// 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 funcion_filosofos( int id )
{
int id_ten_izq = (id+1)              % num_filo_ten, //id. tenedor izq.
id_ten_der = (id+num_filo_ten-1) % num_filo_ten, //id. tenedor der.
valor;

while ( true )
{
cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
// ... solicitar tenedor izquierdo (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, etiq_fi_coge, MPI_COMM_WORLD);

cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
// ... solicitar tenedor derecho (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_der, etiq_fi_coge, MPI_COMM_WORLD);

cout <<"Filósofo " <<id <<" comienza a comer" <<endl ;
sleep_for( milliseconds( aleatorio<10,100>() ) );

cout <<"Filósofo " <<id <<" suelta ten. izq. " <<id_ten_izq <<endl;
// ... soltar el tenedor izquierdo (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, etiq_fi_suelta, MPI_COMM_WORLD);

cout<< "Filósofo " <<id <<" suelta ten. der. " <<id_ten_der <<endl;
// ... soltar el tenedor derecho (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_der, etiq_fi_suelta, MPI_COMM_WORLD);

cout << "Filosofo " << id << " comienza a pensar" << endl;
sleep_for( milliseconds( aleatorio<10,100>() ) );
}
}
// ---------------------------------------------------------------------
```

```
void funcion_tenedores( int id )
{
int valor, id_filosofo ;  // valor recibido, identificador del filósofo
MPI_Status estado ;       // metadatos de las dos recepciones

while ( true )
{
// ..... recibir petición de cualquier filósofo (completar)
MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_fi_coge, MPI_COMM_WORLD,
&estado );

// ..... guardar en 'id_filosofo' el id. del emisor (completar)
id_filosofo = estado.MPI_SOURCE;
cout <<"Ten. " <<id <<" ha sido cogido por filo. " <<id_filosofo <<endl;

// ..... recibir liberación de filósofo 'id_filosofo' (completar)
MPI_Recv( &valor, 1, MPI_INT, id_filosofo, etiq_fi_suelta, MPI_COMM_WORLD, &estado);
cout <<"Ten. "<< id<< " ha sido liberado por filo. " <<id_filosofo <<endl ;
}
}
// ----------------------------------------------------------------------

int main( int argc, char** argv )
{
int id_propio, num_procesos_actual ;

MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );


if ( num_procesos == num_procesos_actual )
{
// ejecutar la función correspondiente a 'id_propio'
if ( id_propio % 2 == 0 )          // si es par
funcion_filosofos( id_propio ); //   es un filósofo
else                               // si es impar
funcion_tenedores( id_propio ); //   es un tenedor
}
else
{
if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
```

```
{ cout << "el número de procesos esperados es:    " << num_procesos << endl
<< "el número de procesos en ejecución es: " << num_procesos_actual << endl
<< "(programa abortado)" << endl ;
}
}

MPI_Finalize( );
return 0;
}

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

Práctica 3. Implementación de algoritmos distribuidos con MPI.

3. Cena de los filósofos con camarero

```
// -----------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 3. Implementación de algoritmos distribuidos con MPI
//
// Archivo: filosofos-camarero.cpp
// Implementación del problema de los filósofos (sin camarero).
// Plantilla para completar.
//
// Implementar una solución distribuida al problema de los filósofos de basado
// en un proceso camarero con espera selectiva. La espera selectiva se consigue
// con el uso de etiquetas.
// -----------------------------------------------------------------------

#include <mpi.h>
#include <thread> // this_thread::sleep_for
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include <iostream>

using namespace std;
using namespace std::this_thread ;
using namespace std::chrono ;

const int
num_filosofos = 5 ,              // número de filósofos
num_filo_ten  = 2*num_filosofos, // número de filósofos y tenedores
num_procesos  = num_filo_ten + 1,// número de procesos total (filo, ten y cam)
// Añadido a la plantilla:
etiq_fi_coge = 0,                // etiqueta de que el filosofo coge el tenedor
etiq_fi_suelta = 1,              // etiqueta de que el filosofo suelta el tenedor
// Añadido a la solución con el camarero
etiq_sentarse = 2,               // etiqueta de que el filosofo se sienta en la mesa
etiq_levantarse = 3,             // etiqueta de que el filosofo se levanta de la mesa
id_camarero = 10;                // identificador del camarero.

//**********************************************************************
// 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 funcion_filosofos( int id )
{
int id_ten_izq = (id+1)              % num_filo_ten, //id. tenedor izq.
id_ten_der = (id+num_filo_ten-1) % num_filo_ten, //id. tenedor der.
valor;

while ( true ){
// 1

cout <<"Filósofo " <<id << " solicita sentarse."<<endl;
// Solicitar si puede sentarse en la mesa.
MPI_Ssend( &valor, 1, MPI_INT, id_camarero, etiq_sentarse, MPI_COMM_WORLD);

// 2

cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
// ... solicitar tenedor izquierdo (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, etiq_fi_coge, MPI_COMM_WORLD);

cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
// ... solicitar tenedor derecho (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_der, etiq_fi_coge, MPI_COMM_WORLD);

// 3

cout <<"Filósofo " <<id <<" comienza a comer" <<endl ;
sleep_for( milliseconds( aleatorio<10,100>() ) );

// 4
```

```
cout <<"Filósofo " <<id <<" suelta ten. izq. " <<id_ten_izq <<endl;
// ... soltar el tenedor izquierdo (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, etiq_fi_suelta, MPI_COMM_WORLD);

cout<< "Filósofo " <<id <<" suelta ten. der. " <<id_ten_der <<endl;
// ... soltar el tenedor derecho (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_der, etiq_fi_suelta, MPI_COMM_WORLD);

// 5

cout <<"Filósofo " <<id << " solicita levantarse."<<endl;
// Solicitar si puede levantarse de la mesa.
MPI_Ssend( &valor, 1, MPI_INT, id_camarero, etiq_levantarse, MPI_COMM_WORLD);

// 6

cout << "Filosofo " << id << " comienza a pensar" << endl;
sleep_for( milliseconds( aleatorio<10,100>() ) );
}
}
// ----------------------------------------------------------------------

void funcion_tenedores( int id )
{
int valor, id_filosofo ;  // valor recibido, identificador del filósofo
MPI_Status estado ;       // metadatos de las dos recepciones

while ( true )
{
// ..... recibir petición de cualquier filósofo (completar)
MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_fi_coge, MPI_COMM_WORLD,
&estado );

// ..... guardar en 'id_filosofo' el id. del emisor (completar)
id_filosofo = estado.MPI_SOURCE;
cout <<"Ten. " <<id <<" ha sido cogido por filo. " <<id_filosofo <<endl;

// ..... recibir liberación de filósofo 'id_filosofo' (completar)
MPI_Recv( &valor, 1, MPI_INT, id_filosofo, etiq_fi_suelta, MPI_COMM_WORLD, &estado);
cout <<"Ten. "<< id<< " ha sido liberado por filo. " <<id_filosofo <<endl ;
}
```

```
}
// ----------------------------------------------------------------------

void funcion_camarero()
{
int etiq_puede = 999;   // etiqueta de lo que puede hacer el filosofo
int s = 0,  // numero de filosofos sentados.
valor,  // valor recibido
id_filo;//id del filosofo que llama al camarero
MPI_Status estado ;       // metadatos de las dos recepciones

while ( true )
{
if(s == num_filosofos-1)   // si s == 4, sólo puede levantarse
etiq_puede = etiq_levantarse;
else if (s == 0)                 // si s == 0, solo puede sentarse
etiq_puede = etiq_sentarse;
else                             // si no, puede las dos
etiq_puede = MPI_ANY_TAG;

// ..... recibir petición de cualquier filósofo
MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_puede, MPI_COMM_WORLD,
&estado );

id_filo = estado.MPI_SOURCE;

if(estado.MPI_TAG == etiq_sentarse){
s++;
cout <<"Filósofo " <<id_filo << " se ha sentado."<<endl;
} else if (estado.MPI_TAG == etiq_levantarse){
s--;
cout <<"Filósofo " <<id_filo << " se ha levantado."<<endl;
}
}
}
// ----------------------------------------------------------------------ç

int main( int argc, char** argv )
{
int id_propio, num_procesos_actual ;

MPI_Init( &argc, &argv );
```

```
MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );


if ( num_procesos == num_procesos_actual )
{
// ejecutar la función correspondiente a 'id_propio'
if( id_propio == 10)                  // si es 10
funcion_camarero();                 // es el camarero
else if ( id_propio % 2 == 0 )          // si es par
funcion_filosofos( id_propio );     //   es un filósofo
else                                  // si es impar
funcion_tenedores( id_propio );     //   es un tenedor
}
else
{
if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
{ cout << "el número de procesos esperados es:    " << num_procesos << endl
<< "el número de procesos en ejecución es: " << num_procesos_actual << endl
<< "(programa abortado)" << endl ;
}
}

MPI_Finalize( );
return 0;
}

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

Práctica 4. Implementación de Sistemas de Tiempo Real.

1. Modificando ejecutivo1.cpp

```
// ---------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 4. Implementación de sistemas de tiempo real.
//
// Archivo: ejecutivo1-compr.cpp
// Implementación del primer ejemplo de ejecutivo cíclico:
//
//   Datos de las tareas:
//   ------------
//   Ta.  T    C
//   ------------
//   A  250  100
//   B  250   80
//   C  500   50
//   D  500   40
//   E 1000   20
//  -------------
//
//  Planificación (con Ts == 250 ms)
//  *---------*----------*---------*--------*
//  | A B C   | A B D E  | A B C   | A B D  |
//  *---------*----------*---------*--------*
//
// ---------------------------------------------------------------------
// Actividad 1: nueva funcionalidad
// En la simulación (en ejecutivo1.cpp) cada tarea es una simple espera bloqueada
// de duración igual a su tiempo de cómputo. También hay una espera al final del
// ciclo secundario.
// - Sabemos que, en la práctica, en una ejecución el tiempo de duración actual
//   de cada una de esas esperas puede ser algo mayor que el argumento de sleep_for.
// - Copia el código en ejecutivo1-compr.cpp y ahí extiéndelo de forma que, cada
//   vez que acaba un ciclo secundario, se informe del retraso del instante final
//   actual respecto al instante final esperado.
// - La comprobación se hará al final del bucle, inmediatamente después de sleep_until.
// ---------------------------------------------------------------------

#include <string>
```

```
#include <iostream> // cout, cerr
#include <thread>
#include <chrono>   // utilidades de tiempo
#include <ratio>    // std::ratio_divide

using namespace std ;
using namespace std::chrono ;
using namespace std::this_thread ;

// tipo para duraciones en segundos y milisegundos, en coma flotante:
//typedef duration<float,ratio<1,1>>    seconds_f ;
typedef duration<float,ratio<1,1000>> milliseconds_f ;

// ---------------------------------------------------------------------
// tarea genérica: duerme durante un intervalo de tiempo (de determinada duración)

void Tarea( const std::string & nombre, milliseconds tcomputo )
{
cout << "   Comienza tarea " << nombre << " (C == " << tcomputo.count() << " ms.) ... " ;
sleep_for( tcomputo );
cout << "fin." << endl ;
}

// ---------------------------------------------------------------------
// tareas concretas del problema:

void TareaA() { Tarea( "A", milliseconds(100) );  }
void TareaB() { Tarea( "B", milliseconds( 80) );  }
void TareaC() { Tarea( "C", milliseconds( 50) );  }
void TareaD() { Tarea( "D", milliseconds( 40) );  }
void TareaE() { Tarea( "E", milliseconds( 20) );  }

// ---------------------------------------------------------------------
// implementación del ejecutivo cíclico:

int main( int argc, char *argv[] )
{
// Ts = duración del ciclo secundario (en unidades de milisegundos, enteros)
const milliseconds Ts_ms( 250 );

// ini_sec = instante de inicio de la iteración actual del ciclo secundario
time_point<steady_clock> ini_sec = steady_clock::now();
```

```
while( true ) // ciclo principal
{
cout << endl
<< "--------------------------------------" << endl
<< "Comienza iteración del ciclo principal." << endl ;

for( int i = 1 ; i <= 4 ; i++ ) // ciclo secundario (4 iteraciones)
{
cout << endl << "Comienza iteración " << i << " del ciclo secundario." << endl ;

switch( i )
{
case 1 : TareaA(); TareaB(); TareaC();          break ;
case 2 : TareaA(); TareaB(); TareaD(); TareaE(); break ;
case 3 : TareaA(); TareaB(); TareaC();          break ;
case 4 : TareaA(); TareaB(); TareaD();          break ;
}

// calcular el siguiente instante de inicio del ciclo secundario
ini_sec += Ts_ms ;

// esperar hasta el inicio de la siguiente iteración del ciclo secundario
sleep_until( ini_sec );

// ACTIVIDAD 1----------------------------------------------------------

// fin_sec = instante final
time_point<steady_clock> fin_sec  = steady_clock::now();

// Calcular la diferencia entre el instante final actual y el instante final esperado
milliseconds_f retraso = fin_sec - ini_sec;
milliseconds_f duracion_real = Ts_ms + retraso;

// Imprimir resultados después de sleep_until
cout << "Duracion esperada:    " << Ts_ms.count() << " milisegundos." << endl;
cout << "Duracion real:        " << duracion_real.count() << " milisegundos." << endl;
cout << "Ocurre un retraso de: " << retraso.count() << " milisegundos." << endl;

}
}
}
```

Práctica 4. Implementación de Sistemas de Tiempo Real.

2. ejecutivo2.cpp

```
// ---------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 4. Implementación de sistemas de tiempo real.
//
// Archivo: ejecutivo2.cpp
// Implementación del segundo ejemplo de ejecutivo cíclico:
//
//   Datos de las tareas:
//   ------------
//   Ta.  T    C
//   ------------
//   A  500  100
//   B  500  150
//   C  1000 200
//   D  2000 240
//  -------------
//
//  Hiperperiodo Tm = 2000ms
//
//  Planificación (con Ts == 500 ms). Se cumple que maxC(240) <= Ts <= minD(500)
//  *---------*----------*---------*--------*
//  | A B C   | A B D    | A B C   | A B    |
//  *---------*----------*---------*--------*
//
// ---------------------------------------------------------------------------
/* Actividad 2: Responde en tu portafolios a estas cuestiones:
*
* PREGUNTA: ¿ cual es el mínimo tiempo de espera que queda al final de las
* iteraciones del ciclo secundario con tu solución ?
*
* RESPUESTA: El mínimo tiempo que queda es 10 ms y ocurre en la 2a iteracion
* del ciclo secundario.
*
* PREGUNTA: ¿ sería planificable si la tarea D tuviese un tiempo cómputo de 250 ms ?
*
* RESPUESTA: En teoría si podría ser planificable, aunque debemos de tener en cuenta que
* habrá iteraciones en las que no haya tiempo de espera al final del ciclo secundario,
```

```
* como es el caso de la iteracion 2, que entre la 2 y la 3 no habría tiempo de espera
* debido a que las tareas A B y D tardan 500ms, que es tiempo del ciclo secundario.
*
*/



#include <string>
#include <iostream> // cout, cerr
#include <thread>
#include <chrono>   // utilidades de tiempo
#include <ratio>    // std::ratio_divide

using namespace std ;
using namespace std::chrono ;
using namespace std::this_thread ;

// tipo para duraciones en segundos y milisegundos, en coma flotante:
//typedef duration<float,ratio<1,1>>    seconds_f ;
typedef duration<float,ratio<1,1000>> milliseconds_f ;

// ---------------------------------------------------------------------
// tarea genérica: duerme durante un intervalo de tiempo (de determinada duración)

void Tarea( const std::string & nombre, milliseconds tcomputo )
{
cout << "   Comienza tarea " << nombre << " (C == " << tcomputo.count() << " ms.) ... " ;
sleep_for( tcomputo );
cout << "fin." << endl ;
}

// ---------------------------------------------------------------------
// tareas concretas del problema:

void TareaA() { Tarea( "A", milliseconds(100) );  }
void TareaB() { Tarea( "B", milliseconds(150) );  }
void TareaC() { Tarea( "C", milliseconds(200) );  }
void TareaD() { Tarea( "D", milliseconds(240) );  }

// ---------------------------------------------------------------------
// implementación del ejecutivo cíclico:

int main( int argc, char *argv[] )
```

```
{
// Ts = duración del ciclo secundario (en unidades de milisegundos, enteros)
const milliseconds Ts_ms( 500 );

// ini_sec = instante de inicio de la iteración actual del ciclo secundario
time_point<steady_clock> ini_sec = steady_clock::now();

while( true ) // ciclo principal
{
cout << endl
<< "--------------------------------------" << endl
<< "Comienza iteración del ciclo principal." << endl ;

for( int i = 1 ; i <= 4 ; i++ ) // ciclo secundario (4 iteraciones)
{
cout << endl << "Comienza iteración " << i << " del ciclo secundario." << endl ;

switch( i )
{
case 1 : TareaA(); TareaB(); TareaC();          break ;
case 2 : TareaA(); TareaB(); TareaD();          break ;
case 3 : TareaA(); TareaB(); TareaC();          break ;
case 4 : TareaA(); TareaB();                    break ;
}

// calcular el siguiente instante de inicio del ciclo secundario
ini_sec += Ts_ms ;

// esperar hasta el inicio de la siguiente iteración del ciclo secundario
sleep_until( ini_sec );

// ACTIVIDAD 1----------------------------------------------------

// fin_sec = instante final
time_point<steady_clock> fin_sec  = steady_clock::now();

// Calcular la diferencia entre el instante final actual y el instante final esperado
milliseconds_f retraso = fin_sec - ini_sec;
milliseconds_f duracion_real = Ts_ms + retraso;

// Imprimir resultados después de sleep_until
cout << "Duracion esperada:    " << Ts_ms.count() << " milisegundos." << endl;
```

```
cout << "Duracion real:         " << duracion_real.count() << " milisegundos." << endl;
cout << "Ocurre un retraso de: " << retraso.count() << " milisegundos." << endl;

}
}
}
```