

SCD2021GUIAPRSEM.pdf



danielsp10



Sistemas Concurrentes y Distribuidos



2º Grado en Ingeniería Informática



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



¿Qué ha sido lo más
escuchado este verano?

Ponte la de Quevedo y Biza Ponla otra vez QUÉÉÉÉDATE

Quevedo: Bzrp Music Sessions, Vol. 52 de Quevedo y Bizarra: +715 millones de reproducciones en 2022.

#SPOTIFYWRAPPED

MÁS INFORMACIÓN



 Spotify®



¿Quién te conoce mejor?

Tu madre

Tus amigos

Tu Wrapped

Si dudas, echa un ojo a tu Wrapped

#SPOTIFYWRAPPED



SISTEMAS CONCURRENTES Y DISTRIBUIDOS

GUÍA DE PRÁCTICAS Y SEMINARIOS RESUELTOS

Autor: DanielsP

0. MOTIVACIÓN

En este documento se encuentra la solución a los ejercicios presentados en la asignatura de SISTEMAS CONCURRENTES Y DISTRIBUIDOS (SCD) de la UNIVERSIDAD DE GRANADA (UGR) realizado en el DOBLE GRADO DE INGENIERÍA INFORMÁTICA Y MATEMÁTICAS (DGIIM) en el curso 2020/21. Los problemas que se han propuesto por parte de los profesores de la asignatura son comunes a los grados de Ingeniería Informática y Doble Grado de Ingeniería Informática y ADE.

1. SEMINARIO1: INTRODUCCIÓN A SCD

En este primer seminario se introduce la Programación Multihebra en C++ (estándar 11) y también se empieza a ver una estructura de datos muy importante a la hora de realizar correctamente trabajos con más de una hebra: los semáforos.

El único ejercicio disponible en esta sección es el de realizar el cálculo de una integral definida sobre el intervalo $[0, 1]$ de una función $f(x)$ que permite aproximar el valor decimal de π . La idea consta de utilizar la aproximación a la integral utilizando sumas de áreas de rectángulos tomando previamente una partición de n partes sobre el intervalo indicado anteriormente. Es por eso, por lo que entra en juego la programación multihebra, a mayor valor de n , mayor tiempo y esfuerzo se requerirá a la hora de computar toda esa suma si lo hacemos de la manera tradicional. Utilizando las hebras de C++, podemos paralelizar el trabajo y dedicar a hacer sumas parciales a cada una de ellas, y por último juntar todo el trabajo, reduciendo así los costes de computación.

La solución al ejercicio es el que se encuentra en el fichero `integral.cpp`:

```
/*
 * @file integralConcurrente.cpp
 * @author Daniel Pérez Ruiz
 */

#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')
```

Quéééédate
y descubre
tu resumen
del año.

MÁS INFORMACIÓN



#SPOTIFYWRAPPED

Reservados todos los derechos. Queda permitida la impresión en su totalidad.
No se permite la explotación económica ni la transformación de esta obra.

WUOLAH

```

// -----
// 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
    }
    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 sumaParcial = 0.0;
    for( long j = i ; j < m; j+=n){
        const double xj = double(j+0.5)/m;
        sumaParcial += f( xj );
    }

    return sumaParcial;
}

// -----
// calculo de la integral de forma concurrente
double calcular_integral_concurrente( ){
    future<double> futuros[n];

    for(int i=0; i<n; i++){
        futuros[i] = async(launch::async, funcion_hebra,i);
    }

    double resultado = 0.0;
    for(int i=0; i<n; i++){
        resultado += futuros[i].get();
    }

    return resultado/m;
}

// -----
int main(int narg, char* argv[])
{
    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() ;

```



LA ÚLTIMA

AITANA OCAÑA · MIGUEL BERNARDEAU
· UNA HISTORIA SOBRE EMPEZAR ·

Temporada completa
Ya disponible solo en



```

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 double pi = 3.141592653589793238461 ;

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;
}

```

2. PRÁCTICA1: SEMÁFOROS

En esta sección se aborda de manera más específica la implementación de soluciones a problemas multihebra con semáforos como estructuras de control. Hay dos problemas principales (que se seguirán viendo en otras prácticas) que serán los que servirán como práctica.

2.1 PROBLEMA DE LOS PRODUCTORES Y CONSUMIDORES

La motivación de este problema es la siguiente: hay una cantidad n de hebras creadas en el sistema tal que existen dos roles para ellas, en las que una hebra i -ésima adoptará uno y sólo uno de ellos en tiempo de ejecución. Las hebras *productoras* se encargan de producir una serie de números empezando por el 1 y terminando por otro que llamaremos *tope*. Cuando lo producen, lo almacenan en un buffer que en nuestro caso será un array estático de C++ (con un tamaño predefinido en tiempo de compilación), siempre que éste se encuentre con capacidad suficiente como para almacenar más información. A continuación, vienen las hebras *consumidoras*, que se encargan de acceder al buffer de datos, recoger uno de los elementos que se han producido, y "consumirlo", que en nuestro caso bastará con imprimirla por pantalla. Se debe verificar que los datos que se han producido se han consumido una Y SÓLO una vez.

Puesto que estamos accediendo a un recurso compartido (entre hebras productoras y consumidoras y entre los mismos tipos de rol), es necesario gestionar las entradas y salidas al buffer mediante una estructura de control, en este caso un semáforo, que impida acceso simultáneo de hebras provocando fallos en datos o en ejecución del propio programa. Además de eso, también es importante definir una política de entrada al buffer para una buena gestión de los datos. Se plantea por parte de los profesores las siguientes:

- Política LIFO (Last In First Out): Se utilizará un único índice que irá recorriendo el vector utilizado como buffer desde la posición 0 hasta la posición $\text{tam_vec}-1$, que se verá

incrementado cuando se realice una inserción en el buffer, y se verá decrementado cuando se produzca una extracción. Esto simula el comportamiento de una Pila (*stack*), ya que hay un puntero apuntando al frente de la pila constantemente.

- Política FIFO (First In First Out): Se utilizará dos índices que irán recorriendo el vector utilizado como buffer desde la posición 0 hasta la posición tam_vec-1, de manera cíclica. El primer índice al que llamaremos *primeraOcupada* se encarga de señalar dentro del buffer cuál es la primera posición que está siendo ocupada por un dato que ha sido producido previamente por una hebra *productora*, por tanto, la primera extracción que se debe de realizar por una hebra *consumidora* debe ser la del dato que es apuntado por dicho índice. El segundo índice al que llamaremos *primeraLibre* se encarga de señalar dentro del buffer cuál es la primera posición que está libre dentro del vector para realizar una inserción de un dato por parte de las hebras *productoras*. Esto simula el comportamiento de una Cola Circular (*Circular Queue*), en el que hay dos punteros que apuntan al tope y al final de la cola respectivamente. Cabe destacar que, al ser una cola circular, la posición de los índices *primeraLibre* y *primeraOcupada* puede ser CUALQUIERA, es decir, puede darse el caso de que haya posiciones al comienzo del buffer que estén libres, y sin embargo el puntero *primeraLibre* apunte a otra posición (naturalmente, contigua a la de otra que está ocupada en caso de que el vector no esté vacío).

En conclusión, aquí se aportan las soluciones al problema de **Productores y Consumidores** utilizando ambas políticas de gestión, y utilizando los Semáforos como estructura de control de acceso exclusivo a un recurso compartido.

Solución al problema **ProdCons** con LIFO en el fichero `prodconsLIFO.cpp`:

```
/**  
 * @file prodconsLIFO.cpp  
 * @author Daniel Pérez Ruiz  
 * @brief PRACTICA 1 - PRODUCTOR CONSUMIDOR CONCURRENTE  
 */  
  
#include <iostream>  
#include <cassert>  
#include <thread>  
#include <mutex>  
#include <random>  
#include "Semaphore.h"  
  
using namespace std;  
using namespace SEM;  
  
//=====  
// VARIABLES COMPARTIDAS  
//=====  
  
const unsigned int num_items = 40; // número de items  
const unsigned int tam_vec = 10 ; // tamaño del buffer  
  
unsigned cont_prod[num_items] = {0}; // contadores de verificación: producidos  
unsigned cont_cons[num_items] = {0}; // contadores de verificación: consumidos  
  
Semaphore libres(tam_vec); //Semaforo que cuenta las posiciones libres del buffer  
Semaphore ocupadas(0); //Semaforo que cuenta las posiciones ocupadas del  
buffer  
Semaphore gestionBuffer(1);
```



¿Quién te conoce mejor?

Tu madre Tus amigos Tu Wrapped

#SPOTIFYWRAPPED

Si dudas, echa un ojo a tu Wrapped

¿Tu madre,
tus amigos
o tu
Wrapped?

Descubre
quién te conoce
mejor en tu
resumen del año.



MÁS INFORMACIÓN



```
unsigned int buffer[tam_vec] = {0}; //Buffer para insertar datos
unsigned int primeraLibre = 0; //Puntero al primer hueco libre [LIFO]

//=====

/** 
 * @brief Genera un entero aleatorio uniformemente distribuido entre dos
 * valores enteros, ambos incluidos
 * @param<T,U> : Números enteros min y max, respectivamente
 * @return un numero aleatorio generado entre min,max
 */
template<int min, int max> int aleatorio(){
    static default_random_engine generador( random_device()() );
    static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
    return distribucion_uniforme( generador );
}

//=====

// FUNCIONES COMUNES A LAS SOLUCIONES LIFO Y FIFO
//=====

/** 
 * @brief Produce un dato aleatorio (con un retardo de thread)
 * @return dato aleatorio
 */
int producir_dato(){
    static int contador = 0 ;
    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));

    cout << "producido: " << contador << endl << flush;

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

//-----

/** 
 * @brief Consumir el dato producido con anterioridad
 * @param dato : Objeto 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;
}

//-----

/** 
 * @brief Comprueba si cada dato se ha producido y consumido una única vez
 */
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 ;
}

//-----

/***
 * @brief Funcion realizada por la hebra. Produce los valores y los inserta
 * en un buffer temporal
 */
void funcion_hebra_productora(){
    for(unsigned i = 0 ; i < num_items ; i++){
        int dato = producir_dato();

        //>>>> INICIO SECCION CRITICA<<<<

        libres.sem_wait();
        //INSERCIÓN DEL DATO
        gestionBuffer.sem_wait();
        assert(0 <= primeraLibre && primeraLibre < tam_vec);

        buffer[primeraLibre++] = dato;
        cout << "insertado de buffer: " << buffer[primeraLibre-1] << endl;
        gestionBuffer.sem_signal();

        ocupadas.sem_signal();

        //>>>> FIN SECCION CRITICA<<<<
    }
}

//-----

/***
 * @brief Funcion realizada por la hebra. Consume los valores y los inserta
 * en un buffer temporal
 */
void funcion_hebra_consumidora(){
    for(unsigned i = 0 ; i < num_items ; i++){
        //>>>> INICIO SECCION CRITICA<<<<

        ocupadas.sem_wait();
        //EXTRACCION DEL DATO

```

```

gestionBuffer.sem_wait();
assert(0 <= primeraLibre && primeraLibre < tam_vec);

int dato = buffer[--primeraLibre];
cout << "extraido de buffer: " << dato << endl;

libres.sem_signal();
gestionBuffer.sem_signal();

//>>>>> FIN SECCION CRITICA<<<<<

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() ;

    test_contadores();
}

```

Solución al problema **ProdCons** con FIFO en el fichero `prodconsFIFO.cpp`:

La solución es exactamente la misma que la proporcionada en el otro fichero. Lo único que varía es la gestión de los índices dentro de las funciones `funcion_hebra_productora()` y `funcion_hebra_consumidora()`.

```

/**
 * @brief Función realizada por la hebra. Produce los valores y los inserta
 * en un buffer temporal
 */
void funcion_hebra_productora(){
    for(unsigned i = 0 ; i < num_items ; i++){
        int dato = producir_dato();

        //>>>>> INICIO SECCION CRITICA<<<<<

        libres.sem_wait();

        gestionBuffer.sem_wait();
        //INSERCIÓN DEL DATO
        assert(0 <= primeraLibre && primeraLibre < tam_vec);

        int index = primeraLibre;

```

```

        buffer[primeraLibre++] = dato;
        if(primeraLibre >= tam_vec) primeraLibre = 0;

        cout << "insertado de buffer: " << buffer[index] << endl;
        gestionBuffer.sem_signal();

        ocupadas.sem_signal();

        //>>>>> FIN SECCION CRITICA<<<<<
    }
}

//-----


/** 
 * @brief Funcion realizada por la hebra. Consume los valores y los inserta
 * en un buffer temporal
 */
void funcion_hebra_consumidora(){
    for(unsigned i = 0 ; i < num_items ; i++){
        //>>>>> INICIO SECCION CRITICA<<<<<

        ocupadas.sem_wait();

        gestionBuffer.sem_wait();
        //EXTRACCION DEL DATO
        assert(0 <= primeraOcupada && primeraOcupada < tam_vec);

        int dato = buffer[primeraOcupada++];
        if(primeraOcupada >= tam_vec) primeraOcupada = 0;

        cout << "extraido de buffer: " << dato << endl;
        gestionBuffer.sem_signal();

        libres.sem_signal();

        //>>>>> FIN SECCION CRITICA<<<<<

        consumir_dato(dato);
    }
}

```

2.2 PROBLEMA DE LOS FUMADORES

Otro de los problemas que sirven para practicar con la utilización de semáforos para controlar recursos compartidos por varias hebras es el que se indica en esta subsección con la siguiente motivación: Se considera un *estanco* en el que hay 1 hebra con un rol determinado que llamaremos *estanquero*, cuya misión es proveer de ciertos ingredientes a sus clientes, que serán n hebras a las que llamaremos *fumadores*.

Los fumadores, para poder realizar la acción de *fumar*, necesitan de una serie de ingredientes que serán proporcionadas por el estanquero a través de la ventanilla del propio estanco. Enumerando a los fumadores con números naturales $0, 1, 2, \dots, n$, los ingredientes que necesitan estos propios fumadores también irán numerados con la misma identificación.



Ábrete la Cuenta Online de BBVA y
llévate 1 año de Wuolah PRO

cómo???



1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

Es por tanto que la secuencia natural del programa sería la siguiente: en primer lugar el *estanquero* produce un ingrediente de manera aleatoria que colocará para su retirada dentro del mostrador/ventanilla. A continuación, el fumador que necesite ese ingrediente (ingrediente *i*-ésimo ↔ fumador *i*-ésimo), lo recogerá, y se pondrá a fumar un periodo de tiempo aleatorio. El estanquero no puede producir más ingredientes hasta que el mostrador esté vacío, lo que quiere decir que un fumador fue a por su ingrediente, y tampoco puede acceder al mostrador un fumador cuyo ingrediente no le corresponde, siendo aquí donde se tendrá que garantizar la exclusión mutua. Sin embargo, para una simulación más real del problema, Sí que se debe permitir que los fumadores fumen simultáneamente.

Una estrategia para afrontar este problema es utilizar un vector de semáforos (convenientemente usar un `std::vector` frente a un array estático de C++) en las que la posición *i*-ésima del vector de semáforos corresponde a la disponibilidad del ingrediente *i*-ésimo para el fumador al que le corresponde dicho ingrediente, estando en estado bloqueado cuando su ingrediente no se encuentre disponible, y estando libre cuando Sí lo esté.

La solución al problema de los **Fumadores** con semáforos es la del fichero `fumadoresSem.cpp`:

```
/*
 * @file fumadores.cpp
 * @author Daniel Pérez Ruiz
 * @brief PRÁCTICA P1 - SISTEMAS CONCURRENTES Y DISTRIBUIDOS
 */

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

using namespace std;
using namespace SEM;

//=====
// VARIABLES COMPARTIDAS
//=====

const int num_fumadores = 3;

vector<Semaphore> ingr_disp;
Semaphore mostr_vacio(1);

//=====

/** 
 * @brief Inicializa los vectores de semafotos especificados
 */
void iniciarSemaforos(){
    assert(num_fumadores > 0);
    for(int i=0; i<num_fumadores; i++)
        ingr_disp.push_back(Semaphore(0));
}
```

ventajas

PRO



Dí adiós a la publi
en los apuntes y
en la web



Participa gratis
en todos los
sorteos



Descarga
carpetas
completas

estudia sin publi

WUOLAH PRO

```

/**
 * @brief Genera un entero aleatorio uniformemente distribuido entre dos
 * valores enteros, ambos incluidos
 * @param<T,U> : Números enteros min y max, respectivamente
 * @return un numero aleatorio generado entre min,max
 */
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);
}

/**
 * @brief Simula la acción de producir un ingrediente, con un retardo aleatorio
 * de hebra
 * @return 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;
}

/**
 * @brief Función ejecutada por la hebra de estanquero
 */
void funcion_hebra_estanquero(){
    int i;

    while(true){
        i = producir_ingrediente();

        //>>> INICIO SECCION CRITICA <<<
        mostr_vacio.sem_wait();
        cout << "Puesto ingr.: " << i << endl;
        ingr_disp[i].sem_signal();
        //>>> FIN SECCION CRITICA <<<
    }
}
/**
 * @brief Función que simula la acción de fumar, con un retardo aleatorio de
 * hebra

```

```

 * @param num_fumador : Número de fumador que realiza la acción
 */
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;
}

/**
 * @brief Función ejecutada por la hebra del fumador
 */
void funcion_hebra_fumador(int num_fumador){
    assert(0 <= num_fumador && num_fumador < num_fumadores);
    while(true){
        //>>> INICIO SECCION CRITICA <<<
        ingr_disp[num_fumador].sem_wait();
        cout << "Retirado ingr.: " << num_fumador << endl;
        mostr_vacio.sem_signal();
        //>>> FIN SECCION CRITICA <<<

        fumar(num_fumador);
    }
}

//-----
int main(){
    //COMPROBAR CONDICIONES
    assert(num_fumadores > 0);

    //INICIAR SEMAFOROS
    iniciarSemaforos();

    //PARTE 0: DECLARACION DE HEBRAS
    thread estanquero;                      //HEBRA DEL ESTANQUERO
    thread fumadores[num_fumadores];          //VECTOR DE HEBRAS DE LOS FUMADORES

    string border =
"=====";
    cout << border << endl << " PROBLEMA DE LOS FUMADORES " << endl << border <<
endl;

    //PARTE 1: LANZAMIENTO DE LAS HEBRAS
    estanquero = thread(funcion_hebra_estanquero);
    for(int i=0; i<num_fumadores; i++){

```

```

        fumadores[i] = thread(funcion_hebra_fumador,i);
    }

//PARTE 2: SINCRONIZACION ENTRE LAS HEBRAS
estanquero.join();
for(int i=0; i<num_fumadores;i++){
    fumadores[i].join();
}

return 0;
}

```

3. SEMINARIO2: MONITORES

En este seminario se introduce el concepto de Monitores y su implementación en C++11. El objetivo fundamental se basa en entender y comprender los diferentes medios para garantizar exclusión mutua a través de los diferentes tipos de monitores que se pueden usar, principalmente se verán las modalidades de *Señalar* y *Continuar* y *Señalar* y *Espera Urgente*, siendo éste último el que se utilizará de forma exclusiva en las prácticas asociadas a esta sección con la implementación del `Hoare Monitor`.

3.1 EL PROBLEMA DE LOS PRODUCTORES Y CONSUMIDORES

Se vuelve a tratar el problema asociado a Productores y Consumidores indicado en la sección anterior, pero esta vez se le da un enfoque con Monitores. Recordando lo visto en teoría lo que se necesita para implementar un monitor son una serie de variables llamadas *Variables Permanentes*, unas colas donde se encontrarán las hebras llamadas *Variables Condición*, y unos métodos que son los que se utilizarán las propias hebras una vez consigan la exclusión mutua.

- *Uso de Monitores Señalar y Continuar:* Recordemos que en este tipo de monitores se basan en señalar una hebra que estaba bloqueada con anterioridad y se prepara en la cola de entrada del monitor para adquirir el cerrojo (y por tanto, tener garantizada la exclusión mutua), mientras que la hebra que señala continúa la ejecución tras esa llamada que ha provocado. Para implementar un monitor SC adecuado para este problema necesitamos el siguiente esquema:

- **Variables Permanentes:**

- buffer[num_celdas_total] (int) : Buffer de tamaño fijo, con los datos
- primeraLibre (int) : Índice que indica la siguiente posición donde realizar una inserción de datos.
- primeraOcupada (int) [Caso FIFO] : Índice que indica la siguiente posición donde realizar una extracción de datos.
- numCeldasOcupadas (int) : Variable que cuenta el total de valores almacenados en el buffer. Es necesario para poder utilizar esperas en las variables condición.
- cerrojoMonitor (mutex) : Objeto mutex necesario para garantizar exclusión mutua en un monitor SC.

- **Variables Condición:**

- ocupadas: Cola donde esperan las hebras *consumidoras* (esperan si no hay datos en el buffer).
- libres: Cola donde esperan las hebras *productoras* (esperan si el buffer está lleno).

- **Métodos:**

Encuentra el trabajo de tus sueños



Escanéame y obtén más info!!



- Constructor
- Leer(int n) (int): Extrae un dato del buffer.
- Escribir(int n) (int): Lee un dato del buffer.

Por tanto, la solución al problema de **ProdCons** para 1 hebra productora, y 1 hebra consumidora, utilizando un monitor **Señalar** y **Continuar**, con política LIFO para el buffer, es la del fichero **prodcons1SCLifo.cpp**:

```
// -----
// 
// Sistemas concurrentes y Distribuidos.
// Seminario 2. Introducción a los monitores en C++11.
//
// archivo: prodcons1_SC_Lifo.cpp
// Ejemplo de un monitor en C++11 con semántica SC, para el problema
// del productor/consumidor, con un único productor y un único consumidor.
// Opcion LIFO (stack)
//
// Historial:
// Creado en Julio de 2017
// -----
// -----
// BIBLIOTECAS A UTILIZAR
// -----
#include <iostream>
#include <iomanip>
#include <cassert>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <random>

using namespace std;

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

mutex mtx;                      // mutex de escritura en pantalla

unsigned cont_prod[num_items],    // contadores de verificación: producidos
        cont_cons[num_items];   // contadores de verificación: consumidos

// -----
/** 
 * @brief Genera un entero aleatorio uniformemente distribuido entre dos
 * valores enteros, ambos incluidos
 * @param<T,U> : Números enteros min y max, respectivamente
 * @return un numero aleatorio generado entre min,max
 */
template<int min, int max> int aleatorio(){
    static default_random_engine generador( (random_device())() );
    uniform_int_distribution<int> dist(min, max);
    return dist(generador);
}
```



WUOLAH

```

        static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
        return distribucion_uniforme( generador );
    }

//=====
// FUNCIONES COMUNES A LAS SOLUCIONES LIFO Y FIFO
//=====

/***
 * @brief Produce un dato aleatorio (con un retardo de thread)
 * @return dato aleatorio
 */
int producir_dato(){
    static int contador = 0 ;
    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));

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

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

//-----

/***
 * @brief Consume el dato producido con anterioridad
 * @param dato : Objeto producido
 */
void consumir_dato(unsigned dato){
    if(num_items <= dato){
        cout << " dato === " << dato << ", num_items == " << num_items << endl ;
        assert( dato < num_items );
    }

    cont_cons[dato]++;
    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
    mtx.lock();
    cout << "                consumido: " << dato << endl ;
    mtx.unlock();
}

//-----

/***
 * @brief Inicializa los contadores de items
 */
void ini_contadores(){
    for( unsigned i = 0 ; i < num_items ; i++ ){
        cont_prod[i] = 0 ;
        cont_cons[i] = 0 ;
    }
}

//-----

/***

```

```

 * @brief Comprueba si cada dato se ha producido y consumido una única vez
 */
void test_contadores(){
    bool ok = true ;
    cout << "comprobando contadores ...." << flush;
    for(unsigned i = 0 ; i < num_items ; i++){
        if (cont_prod[i] != 1){
            cout << "error: valor " << i << " producido " << cont_prod[i] << "
veces." << endl ;
            ok = false ;
        }
        if(cont_cons[i] != 1){
            cout << "error: valor " << i << " consumido " << cont_cons[i] << "
veces" << endl ;
            ok = false ;
        }
    }

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

//=====================================================================
//=====================================================================
// MONITOR PRODUCTOR Y CONSUMIDOR
//=====================================================================

/***
 * @brief Esta clase representa un monitor con las siguientes características
 * -> Posee Buffer : Versión FIFO
 * -> Semántica SC
 * -> Num Productores : 1
 * -> Num Consumidores : 1
 */
class ProdCons1SC{
private:
    //VARIABLES CONSTANTES
    static const int num_celdas_total = 10;      // númer. de entradas del buffer

    //VARIABLES PERMANENTES
    int buffer[num_celdas_total], //buffer de tamaño fijo, con los datos
        primera_libre;           //indice de celda de la próxima inserción

    mutex cerrojo_monitor;          //cerrojo del monitor

    //COLAS DE CONDICION
    condition_variable ocupadas,   //cola donde espera el consumidor (n>0)
        libres;                 //cola donde espera el productor
    (n<num_celdas_total)

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


```

```

// ----

ProdCons1SC::ProdCons1SC( )
{
    primera_libre = 0 ;
}

/**
 * @brief Función llamada por el consumidor para extraer un dato
 * @return dato extraido
 */
int ProdCons1SC::leer()
{
    // ganar la exclusión mutua del monitor con una guarda
    unique_lock<mutex> guarda( cerrojo_monitor );

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

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

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

    // devolver valor
    return valor ;
}

// ----

/**
 * @brief Función llamada por el consumidor para insertar un dato
 */
void ProdCons1SC::escribir( int valor )
{
    // ganar la exclusión mutua del monitor con una guarda
    unique_lock<mutex> guarda( cerrojo_monitor );

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

    //cout << "escribir: ocup == " << num_celdas_ocupadas << ", total == " <<
    num_celdas_total << 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.notify_one();
}

```

Encuentra el trabajo de tus sueños

Participa en retos y competiciones de programación



Escanéame y
obtén más info!!

```
}

//=====
// FUNCIONES DE HEBRAS
//=====

/***
 * @brief Función de la hebra que produce datos
 * @param monitor : Puntero al monitor ProdCons
 */
void funcion_hebra_productora( ProdCons1SC * monitor )
{
    for( unsigned i = 0 ; i < num_items ; i++ )
    {
        int valor = producir_dato();
        monitor->escribir( valor );
    }
}

// ----

/***
 * @brief Función de la hebra que consume datos
 * @param monitor : Puntero al monitor ProdCons
 */
void funcion_hebra_consumidora( ProdCons1SC * monitor )
{
    for( unsigned i = 0 ; i < num_items ; i++ )
    {
        int valor = monitor->leer();
        consumir_dato( valor ) ;
    }
}

//=====
// FUNCION MAIN
//=====

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

    ProdCons1SC monitor ;

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

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

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



```
}
```

En el caso de que utilicemos una política FIFO, lo único que varía es la adición de una nueva variable permanente *primeraOcupada*, que será el índice que indique en qué posición dentro del buffer se debe realizar la extracción del dato.

La solución al problema **ProdCons** con 1 hebra productora y 1 hebra consumidora, utilizando un monitor *Señalar* y *Continuar*, con política FIFO para el buffer, es la del fichero

`prodcons1SCFifo.cpp`:

```
//=====
// MONITOR PRODUCTOR Y CONSUMIDOR
//=====

/** 
 * @brief Esta clase representa un monitor con las siguientes características
 * -> Posee Buffer : Versión FIFO
 * -> Semántica SC
 * -> Num Productores : 1
 * -> Num Consumidores : 1
 */
class ProdCons1SC{
private:
    //VARIABLES CONSTANTES
    static const int num_celdas_total = 10;      // númer. de entradas del buffer

    //VARIABLES PERMANENTES
    int 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 inserción
        num_celdas_ocupadas;      //comprueba el número de celdas ocupadas en el
    buffer

    mutex cerrojo_monitor ;      //cerrojo del monitor

    //COLAS DE CONDICION
    condition_variable ocupadas, //cola donde espera el consumidor (n>0)
                        libres;   //cola donde espera el productor
    (n<num_celdas_total)

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

// ----

ProdCons1SC::ProdCons1SC()
{
    primera_libre = 0 ;
    primera_ocupada = 0;
    num_celdas_ocupadas = 0;
}

// -----
```

```

/**
 * @brief Función llamada por el consumidor para extraer un dato
 * @return dato extraido
 */
int ProdCons1SC::leer()
{
    // ganar la exclusión mutua del monitor con una guarda
    unique_lock<mutex> guarda( cerrojo_monitor );

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

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

    primera_ocupada = (primera_ocupada + 1) % num_celdas_total;
    num_celdas_ocupadas--;

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

    // devolver valor
    return valor ;
}

// -----
/** 
 * @brief Función llamada por el consumidor para insertar un dato
 */
void ProdCons1SC::escribir( int valor )
{
    // ganar la exclusión mutua del monitor con una guarda
    unique_lock<mutex> guarda( cerrojo_monitor );

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

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

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

    primera_libre = (primera_libre + 1) % num_celdas_total;
    num_celdas_ocupadas++;

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

```

Para adaptar este problema y permitir que se puedan utilizar múltiples hebras productoras y múltiples hebras consumidoras, lo único que hay que realizar es declarar variables estáticas y constantes especificando cuántas hay de cada rol; asegurarnos de que el total de números que se van a producir y consumir es divisible por el número de hebras productoras y consumidoras para equilibrar el trabajo adecuadamente, y añadir en los métodos un valor identificativo entero, para indicar qué hebra es la que está produciendo o consumiendo el dato.

Por tanto, la solución al problema de **ProdCons** con múltiples hebras productoras y consumidoras, utilizando un monitor *Señalar y Continuar*, con política LIFO para el buffer, es la del fichero `prodconsMULTSCLifo.cpp`:

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



/***
 * =====
 * MODIFICACIONES REALIZADAS PARA MÚLTIPLOS PROD/CONS
 * --> Variables compartidas:
 *      -> Crear num_productores y num_consumidores
 *
 * --> producirDatos(int i):
 *      -> Se pone parámetro para identificar la hebra que la invoca
 *      -> Se produce uniformemente cada dato (en orden)
 *
 * --> consumirDatos(int valor, int numHebra):
 *      -> Se pone param numHebra para identificar la hebra que la invoca
 *
 * --> funcion_hebra_productora(int numHebra):
 *      -> Bucle hasta datos_producidos_hebra
 *      -> Se llama a la nueva función
 *
 * --> funcion_hebra_consumidora(int numHebra):
 *      -> Bucle hasta datos_consumidos_hebra
 *      -> Se llama a la nueva función
 *
 * --> Main:
 *      -> Assert para comprobar que los num_productores y num_consumidores
 *          son correctos
 *      -> Vector de hebras
 *
 * =====
 */
//=====
// BIBLIOTECAS A UTILIZAR
//=====
```



¿Quién te conoce mejor?

Tu madre

Tus amigos

Tu Wrapped

Si dudas, echa un ojo a tu Wrapped

#SPOTIFYWRAPPED



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

using namespace std;

//=====
// VARIABLES COMPARTIDAS
//=====

constexpr int num_items = 40;      // número de items a producir/consumir
constexpr int num_productores = 10, // número de productores totales
              num_consumidores = 5; // número de consumidores totales

constexpr int datos_producidos_hebra = num_items / num_productores,
            datos_consumidos_hebra = num_items / num_consumidores;

mutex mtx;                         // mutex de escritura en pantalla

unsigned cont_prod[num_items],     // contadores de verificación: producidos
        cont_cons[num_items];       // contadores de verificación: consumidos

unsigned producciones_hebras[num_productores] = {0},   //Cuenta las veces
producidas por cada hebra
        consumiciones_hebras[num_consumidores] = {0}; //Cuenta las veces
consumidas por cada hebra

//=====

/**
 * @brief Genera un entero aleatorio uniformemente distribuido entre dos
 * valores enteros, ambos incluidos
 * @param<T,U> : Números enteros min y max, respectivamente
 * @return un numero aleatorio generado entre min,max
 */
template<int min, int max> int aleatorio(){
    static default_random_engine generador( (random_device())() );
    static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
    return distribucion_uniforme( generador );
}

//=====
// FUNCIONES COMUNES A LAS SOLUCIONES LIFO Y FIFO
//=====

/**
 * @brief Produce un dato aleatorio (con un retardo de thread)
 * @param i : Número de thread
 * @return dato aleatorio
 */
int producir_dato(int i){
    int dato = i * datos_producidos_hebra + producciones_hebras[i];
    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ) );
}
```

Quéééédate
y descubre
tu resumen
del año.

MÁS INFORMACIÓN



#SPOTIFYWRAPPED

Reservados todos los derechos. Queda permitida la impresión en su totalidad.
No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

WUOLAH

```

    mtx.lock();
    cout << "[Hebra: " << i << "] producido: " << dato << endl << flush ;
    mtx.unlock();

    producciones_hebras[i]++;
    cont_prod[dato]++;
    return dato;
}

//-----

/***
 * @brief Consume el dato producido con anterioridad
 * @param dato : Objeto producido
 */
void consumir_dato(unsigned dato, int numHebra){
    if(num_items <= dato){
        cout << " dato === " << dato << ", num_items == " << num_items << endl ;
        assert( dato < num_items );
    }

    cont_cons[dato]++;
    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ) );
    mtx.lock();
    cout << "                               [Hebra: " << numHebra << "] consumido: " << dato <<
endl ;
    mtx.unlock();
}

//-----

/***
 * @brief Inicializa los contadores de items
 */
void ini_contadores(){
    for( unsigned i = 0 ; i < num_items ; i++ ){
        cont_prod[i] = 0 ;
        cont_cons[i] = 0 ;
    }
}

//-----

/***
 * @brief Comprueba si cada dato se ha producido y consumido una única vez
 */
void test_contadores(){
    bool ok = true ;
    cout << "comprobando contadores ...." << flush;
    for(unsigned i = 0 ; i < num_items ; i++){
        if (cont_prod[i] != 1){
            cout << "error: valor " << i << " producido " << cont_prod[i] << "
veces." << endl ;
            ok = false ;
        }
        if(cont_cons[i] != 1){

```

```

        cout << "error: valor " << i << " consumido " << cont_cons[i] << "
veces" << endl ;
        ok = false ;
    }

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

//=====================================================================
//=====================================================================
// MONITOR PRODUCTOR Y CONSUMIDOR
//=====================================================================

<**
 * @brief Esta clase representa un monitor con las siguientes características
 * -> Posee Buffer : Versión FIFO
 * -> Semántica SC
 * -> Num Productores : Múltiples
 * -> Num Consumidores : Múltiples
 */
class ProdConsMultSC{
private:
    //VARIABLES CONSTANTES
    static const int num_celdas_total = 10;    // númer. de entradas del buffer

    //VARIABLES PERMANENTES
    int buffer[num_celdas_total], //buffer de tamaño fijo, con los datos
        primera_libre;           //índice de celda de la próxima inserción

    mutex cerrojo_monitor ;      //cerrojo del monitor

    //COLAS DE CONDICION
    condition_variable ocupadas, //cola donde espera el consumidor (n>0)
                      libres;   //cola donde espera el productor
    (n<num_celdas_total)

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

// -----
ProdConsMultSC::ProdConsMultSC()
{
    primera_libre = 0 ;
}

<**
 * @brief Función llamada por el consumidor para extraer un dato
 * @return dato extraído
 */

```

```

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

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

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

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

    // devolver valor
    return valor ;
}

// -----
/** 
 * @brief Función llamada por el consumidor para insertar un dato
 */
void ProdConsMultSC::escribir( int valor )
{
    // ganar la exclusión mutua del monitor con una guarda
    unique_lock<mutex> guarda( cerrojo_monitor );

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

    //cout << "escribir: ocup == " << num_celdas_ocupadas << ", 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.notify_one();
}

//=====
// FUNCIONES DE HEBRAS
//=====

/** 
 * @brief Función de la hebra que produce datos
 * @param monitor : Puntero al monitor ProdCons
 */
void funcion_hebra_productora( ProdConsMultSC * monitor, int numHebra )
{

```



¿Quién te conoce mejor?

Tu madre Tus amigos Tu Wrapped

#SPOTIFYWRAPPED

Si dudas, echa un ojo a tu Wrapped

¿Tu madre,
tus amigos
o tú
Wrapped?

Descubre
quién te conoce
mejor en tu
resumen del año.



MÁS INFORMACIÓN



```
for( unsigned i = 0 ; i < datos_producidos_hebra ; i++ )
{
    int valor = producir_dato(numHebra) ;
    monitor->escribir( valor );
}

// -----
/** 
 * @brief Función de la hebra que consume datos
 * @param monitor : Puntero al monitor ProdCons
 */
void funcion_hebra_consumidora( ProdConsMultSC * monitor, int numHebra )
{
    for( unsigned i = 0 ; i < datos_consumidos_hebra ; i++ )
    {
        int valor = monitor->leer();
        consumir_dato( valor, numHebra ) ;
    }
}

//=====
// FUNCION MAIN
//=====

int main()
{
    //COMPROBACIÓN DE PARÁMETROS CORRECTOS
    assert((num_items % num_productores == 0) && (num_items % num_consumidores == 0));

    cout << "-----"
        << endl
        << "Problema de los productores-consumidores (MULT prod/cons, Monitor SC,
buffer LIFO). " << endl
        << "-----"
        << endl
        << flush ;

    ProdConsMultSC monitor;

    thread hebras_productoras[num_productores];
    thread hebras_consumidoras[num_consumidores];

    for(int i=0; i<num_productores; i++){
        hebras_productoras[i] = thread(funcion_hebra_productora, &monitor, i);
    }

    for(int i=0; i<num_consumidores; i++){
        hebras_consumidoras[i] = thread(funcion_hebra_consumidora, &monitor, i);
    }

    for(int i=0; i<num_productores; i++){
        hebras_productoras[i].join();
    }

    for(int i=0; i<num_consumidores; i){

```

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

WUOLAH

```

        hebras_consumidoras[i].join();
    }

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

```

Igual que en el caso 1&1, para permitir la política FIFO del buffer lo único que hay que realizar es añadir la variable permanente *primeraOcupada* señalando la primera extracción dentro del buffer.

La solución al problema de **ProdCons** con múltiples hebras productoras y consumidoras, utilizando un monitor *Señalar y Continuar*, con política FIFO para el buffer, se basa en una combinación de los dos ejemplos anteriores, por lo que basta reemplazar en el fichero anterior el monitor con política FIFO para el caso de 1 hebra prod/cons.

- *Uso de monitores Señalar y Espera Urgente:* La diferencia con respecto los otros monitores radica en lo que sucede cuando una hebra que se encuentra en exclusión mutua dentro del monitor hace una llamada a `signal()` y señala a otra hebra que se encuentre esperando dentro de la cola. La hebra señaladora adquiere el cerrojo (obteniendo así la exclusión mutua), y continúa ejecutando sus sentencias después del wait en el que se quedó esperando; mientras que la hebra señaladora se queda esperando en una cola especial llamada *cola de urgentes*. Si la cola de urgentes no se encuentra vacía cuando una hebra ejecuta una llamada a signal, ésta tiene prioridad sobre las demás a la hora de adquirir la exclusión mutua, en caso contrario, cualquier hebra que espere en la cola de bloqueados puede entrar al monitor.

Ya no será necesario utilizar un mutex para la exclusión mutua ya que contamos con la implementación de `HoareMonitor`, además de tener un tipo de datos especial llamado `CondVar` con el que representaremos las variables condición. Para crear un monitor SU, crearemos una clase con los mismos atributos y métodos que necesitaba SC (salvando los cambios que hemos comentado anteriormente), y además, la clase creada será una clase hija de `HoareMonitor`.

Se mostrará cómo debe quedar la solución completa en el siguiente fichero, especialmente para que el lector comprenda cómo se debe utilizar esta herramienta proporcionada por el profesor a la hora de instanciarlo, crearlo, y referenciarlo para pasarlo por funciones.

La solución para el problema de **ProdCons** con una hebra productora y una consumidora, utilizando un monitor *Señalar y Espera Urgente* con política LIFO para el buffer de datos es la del fichero `prodconsSU_LIFO.cpp`:

```

// -----
//
// Sistemas concurrentes y Distribuidos.
// Seminario 2. Introducción a los monitores en C++11.
//
// archivo: prodcons1_SU_Lifo.cpp
// Ejemplo de un monitor en C++11 con semántica SU, para el problema
// del productor/consumidor, con un único productor y un único consumidor.
// Opcion LIFO (stack)
//
// Historial:

```

```

// Creado en Julio de 2017
// -----
//=====//
// BIBLIOTECAS A UTILIZAR
//=====//

#include <iostream>
#include <iomanip>
#include <cassert>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <random>
#include "HoareMonitor.h"

using namespace std;
using namespace HM;      //NECESARIO PARA UTILIZAR HOAREMONITOR

//=====
// VARIABLES COMPARTIDAS
//=====

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

mutex mtx;                      // mutex de escritura en pantalla

unsigned cont_prod[num_items],    // contadores de verificación: producidos
        cont_cons[num_items];    // contadores de verificación: consumidos

//=====

/**
 * @brief Genera un entero aleatorio uniformemente distribuido entre dos
 * valores enteros, ambos incluidos
 * @param<T,U> : Números enteros min y max, respectivamente
 * @return un numero aleatorio generado entre min,max
 */
template<int min, int max> int aleatorio(){
    static default_random_engine generador( random_device()());
    static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
    return distribucion_uniforme( generador );
}

//=====
// FUNCIONES COMUNES A LAS SOLUCIONES LIFO Y FIFO
//=====

/**
 * @brief Produce un dato aleatorio (con un retardo de thread)
 * @return dato aleatorio
 */
int producir_dato(){
    static int contador = 0 ;
    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));

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

```

```

    mtx.unlock();

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

//-----

/***
 * @brief Consume el dato producido con anterioridad
 * @param dato : Objeto producido
 */
void consumir_dato(unsigned dato){
    if(num_items <= dato){
        cout << " dato === " << dato << ", num_items == " << num_items << endl ;
        assert( dato < num_items );
    }

    cont_cons[dato]++;
    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ) );
    mtx.lock();
    cout << "                consumido: " << dato << endl ;
    mtx.unlock();
}

//-----

/***
 * @brief Inicializa los contadores de items
 */
void ini_contadores(){
    for( unsigned i = 0 ; i < num_items ; i++ ){
        cont_prod[i] = 0 ;
        cont_cons[i] = 0 ;
    }
}

//-----

/***
 * @brief Comprueba si cada dato se ha producido y consumido una única vez
 */
void test_contadores(){
    bool ok = true ;
    cout << "comprobando contadores ...." << flush;
    for(unsigned i = 0 ; i < num_items ; i++){
        if (cont_prod[i] != 1){
            cout << "error: valor " << i << " producido " << cont_prod[i] << "
veces." << endl ;
            ok = false ;
        }
        if(cont_cons[i] != 1){
            cout << "error: valor " << i << " consumido " << cont_cons[i] << "
veces" << endl ;
            ok = false ;
        }
    }
}

```



Ábrete la Cuenta Online de BBVA y
llévate 1 año de Wuolah PRO

cómo??



1/6

Este número es
indicativo del
riesgo del
producto, siendo
1/6 indicativo de
menor riesgo y
6/6 de mayor
riesgo.

BBVA está
adherido al
Fondo de
Garantía de
Depósitos de
Entidades de
Crédito de
España.
La cantidad
máxima
garantizada es
de 100.000 euros
por la totalidad
de los depósitos
constituidos
en BBVA por
persona.

ventajas

PRO



Dí adiós a la publi
en los apuntes y
en la web



Participa gratis
en todos los
sorteos



Descarga
carpetas
completas

estudia sin publi
WUOLAH PRO

```
    if(ok)
        cout << endl << flush << "solución (aparentemente) correcta." << endl <<
flush ;
}

//=====
//===== MONITOR PRODUCTOR Y CONSUMIDOR =====
//=====

/** 
 * @brief Esta clase representa un monitor con las siguientes características
 * -> Posee Buffer : Versión LIFO
 * -> Semántica SU
 * -> Num Productores : 1
 * -> Num Consumidores : 1
 */
class ProdCons1SU : public HoareMonitor{
private:
    //VARIABLES CONSTANTES
    static const int num_celdas_total = 10;    // núm. de entradas del buffer

    //VARIABLES PERMANENTES
    int buffer[num_celdas_total], //buffer de tamaño fijo, con los datos
        primera_libre;           //indice de celda de la próxima inserción

    //COLAS DE CONDICION
    CondVar ocupadas, //cola donde espera el consumidor (n>0)
               libres;   //cola donde espera el productor (n<num_celdas_total)

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

// -----
ProdCons1SU::ProdCons1SU()
{
    primera_libre = 0 ;
    ocupadas = newCondVar();
    libres = newCondVar();
}

/** 
 * @brief Función llamada por el consumidor para extraer un dato
 * @return dato extraido
 */
int ProdCons1SU::leer()
{
    // esperar bloqueado hasta que 0 < num_celdas_ocupadas
    if ( primera_libre == 0 )
        ocupadas.wait();

    // hacer la operación de lectura, actualizando estado del monitor
}
```

```

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

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

// devolver valor
return valor ;
}

// -----
/***
 * @brief Función llamada por el consumidor para insertar un dato
 */
void ProdCons1SU::escribir( int valor )
{
    // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total
    if ( primera_libre == num_celdas_total )
        libres.wait();

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

/***
 * @brief Función de la hebra que produce datos
 * @param monitor : Puntero al monitor ProdCons
 */
void funcion_hebra_productora(MRef<ProdCons1SU> monitor)
{
    for( unsigned i = 0 ; i < num_items ; i++ )
    {
        int valor = producir_dato() ;
        monitor->escribir( valor );
    }
}

// -----
/***
 * @brief Función de la hebra que consume datos
 * @param monitor : Puntero al monitor ProdCons
 */

```

```

void funcion_hebra_consumidora(MRef<ProdCons1SU> monitor)
{
    for( unsigned i = 0 ; i < num_items ; i++ )
    {
        int valor = monitor->leer();
        consumir_dato( valor ) ;
    }
}

//=====================================================================
// FUNCION MAIN
//=====================================================================

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

//DECLARACIÓN DEL MONITOR SU
MRef<ProdCons1SU> monitor = Create<ProdCons1SU>();

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

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

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

```

La solución para el problema de **ProdCons** con una hebra productora y una consumidora, utilizando un monitor *Señalar y Espera Urgente* con política FIFO para el buffer de datos es la del fichero `prodconsSUFIFO.cpp`:

```

//=====================================================================
// MONITOR PRODUCTOR Y CONSUMIDOR
//=====================================================================

/**
 * @brief Esta clase representa un monitor con las siguientes características
 * -> Posee Buffer : Versión FIFO
 * -> Semántica SU
 * -> Num Productores : 1
 * -> Num Consumidores : 1
 */
class ProdCons1SU : public HoareMonitor {
private:
    //VARIABLES CONSTANTES
    static const int num_celdas_total = 10;    // númer. de entradas del buffer

    //VARIABLES PERMANENTES

```

```

int 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 inserción
    num_celdas_ocupadas;     //comprueba el número de celdas ocupadas en el
buffer

//COLAS DE CONDICION
CondVar ocupadas, //cola donde espera el consumidor (n>0)
    libres; //cola donde espera el productor (n<num_celdas_total)

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

// ----

ProdCons1SU::ProdCons1SU()
{
    primera_libre = 0 ;
    primera_ocupada = 0;
    num_celdas_ocupadas = 0;

    ocupadas = newCondVar();
    libres = newCondVar();
}

// ----

/***
 * @brief Función llamada por el consumidor para extraer un dato
 * @return dato extraido
 */
int ProdCons1SU::leer()
{
    // esperar bloqueado hasta que 0 < num_celdas_ocupadas
    if ( num_celdas_ocupadas == 0 )
        ocupadas.wait();

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

    primera_ocupada = (primera_ocupada + 1) % num_celdas_total;
    num_celdas_ocupadas--;

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

    // devolver valor
    return valor ;
}

// ----

/***

```

Encuentra el trabajo de tus sueños



Escanéame y obtén más info!!

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

```
* @brief Función llamada por el consumidor para insertar un dato
*/
void ProdCons1SU::escribir( int valor )
{
    // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total
    if ( num_celdas_ocupadas == num_celdas_total )
        libres.wait();

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

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

    primera_libre = (primera_libre + 1) % num_celdas_total;
    num_celdas_ocupadas++;

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

La solución para el problema de **ProdCons** con múltiples hebras productoras y consumidoras, utilizando un monitor *Señalar y Espera Urgente* con política LIFO para el buffer de datos es la del fichero `prodconsMULTSULIFO.cpp`:

```
// -----
// 
// Sistemas concurrentes y Distribuidos.
// Seminario 2. Introducción a los monitores en C++11.
// 
// archivo: prodcons1_SU_Lifo.cpp
// Ejemplo de un monitor en C++11 con semántica SU, para el problema
// del productor/consumidor, con un único productor y un único consumidor.
// Opcion LIFO (stack)
// 
// Historial:
// Creado en Julio de 2017
// -----


/**
 * =====
 * MODIFICACIONES REALIZADAS PARA MÚLTIPLOS PROD/CONS
 * --> Variables compartidas:
 *      -> Crear num_productores y num_consumidores
 *
 * --> producirDatos(int i):
 *      -> Se pone parámetro para identificar la hebra que la invoca
 *      -> Se produce uniformemente cada dato (en orden)
 *
 * --> consumirDatos(int valor, int numHebra):
 *      -> Se pone param numHebra para identificar la hebra que la invoca
 *
 * --> funcion_hebra_productora(int numHebra):
 *      -> Bucle hasta datos_producidos_hebra
 *      -> Se llama a la nueva función
 *
```

```

* --> funcion_hebra_consumidora(int numHebra):
*     -> Bucle hasta datos_consumidos_hebra
*     -> Se llama a la nueva función
*
* --> Main:
*     -> Assert para comprobar que los num_productores y num_consumidores
*        son correctos
*     -> Vector de hebras
*
* =====
*/
// -----
//=====//
// BIBLIOTECAS A UTILIZAR
//=====//

#include <iostream>
#include <iomanip>
#include <cassert>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <random>
#include "HoareMonitor.h"

using namespace std;
using namespace HM;      //NECESARIO PARA UTILIZAR HOAREMONITOR

//=====
// VARIABLES COMPARTIDAS
//=====

constexpr int num_items = 40;          // número de items a producir/consumir
constexpr int num_productores = 5,    // número de productores totales
            num_consumidores = 10; // número de consumidores totales

constexpr int datos_producidos_hebra = num_items / num_productores,
            datos_consumidos_hebra = num_items / num_consumidores;

mutex mtx;                          // mutex de escritura en pantalla

unsigned cont_prod[num_items],      // contadores de verificación: producidos
         cont_cons[num_items];      // contadores de verificación: consumidos

unsigned producciones_hebras[num_productores] = {0},   //Cuenta las veces
producidas por cada hebra
        consumiciones_hebras[num_consumidores] = {0}; //Cuenta las veces
consumidas por cada hebra

//=====

/***
 * @brief Genera un entero aleatorio uniformemente distribuido entre dos
 * valores enteros, ambos incluidos
 * @param<T,U> : Números enteros min y max, respectivamente
 * @return un numero aleatorio generado entre min,max
*/

```

```

*/
template<int min, int max> int aleatorio(){
    static default_random_engine generador( (random_device())() );
    static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
    return distribucion_uniforme( generador );
}

//=====================================================================
// FUNCIONES COMUNES A LAS SOLUCIONES LIFO Y FIFO
//=====================================================================

/***
 * @brief Produce un dato aleatorio (con un retardo de thread)
 * @param i : Índice de thread
 * @return dato aleatorio
 */
int producir_dato(int i){
    int dato = i * datos_producidos_hebra + producciones_hebras[i];
    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));

    mtx.lock();
    cout << "[Hebra: " << i << "] producido: " << dato << endl << flush ;
    mtx.unlock();

    producciones_hebras[i]++;
    cont_prod[dato]++;
    return dato;
}

//-----

/***
 * @brief Consume el dato producido con anterioridad
 * @param dato : Objeto producido
 * @param numHebra : Número de Hebra
 */
void consumir_dato(unsigned dato, int numHebra){
    if(num_items <= dato){
        cout << "dato === " << dato << ", num_items == " << num_items << endl ;
        assert( dato < num_items );
    }

    cont_cons[dato]++;
    consumiciones_hebras[numHebra]++;

    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));

    mtx.lock();
    cout << "                                [Hebra: " << numHebra << "] consumido: " << dato <<
endl ;
    mtx.unlock();
}

//-----

/***
 * @brief Inicializa los contadores de items
 */
void ini_contadores(){

}

```

```

        for( unsigned i = 0 ; i < num_items ; i++ ){
            cont_prod[i] = 0 ;
            cont_cons[i] = 0 ;
        }

    //}

    //-----



/** 
 * @brief Comprueba si cada dato se ha producido y consumido una única vez
 */
void test_contadores(){
    bool ok = true ;
    cout << "comprobando contadores ...." << flush;
    for(unsigned i = 0 ; i < num_items ; i++){
        if (cont_prod[i] != 1){
            cout << "error: valor " << i << " producido " << cont_prod[i] << "
veces." << endl ;
            ok = false ;
        }
        if(cont_cons[i] != 1){
            cout << "error: valor " << i << " consumido " << cont_cons[i] << "
veces" << endl ;
            ok = false ;
        }
    }

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

//=====

//=====

// MONITOR PRODUCTOR Y CONSUMIDOR
//=====

/** 
 * @brief Esta clase representa un monitor con las siguientes características
 * -> Posee Buffer : Versión LIFO
 * -> Semántica SU
 * -> Num Productores : Múltiples
 * -> Num Consumidores : Múltiples
 */
class ProdConsMultSU : public HoareMonitor{
private:
    //VARIABLES CONSTANTES
    static const int num_celdas_total = 10;    // númer. de entradas del buffer

    //VARIABLES PERMANENTES
    int buffer[num_celdas_total], //buffer de tamaño fijo, con los datos
        primera_libre;           //índice de celda de la próxima inserción

    //COLAS DE CONDICION
    CondVar ocupadas, //cola donde espera el consumidor (n>0)
        libres;      //cola donde espera el productor (n<num_celdas_total)
}

```

Encuentra el trabajo de tus sueños

Participa en retos y competiciones de programación



Escanéame y
obtén más info!!

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

// ----

ProdConsMultSU::ProdConsMultSU()
{
    primera_libre = 0 ;
    ocupadas = newCondVar();
    libres = newCondVar();
}

/**
 * @brief Función llamada por el consumidor para extraer un dato
 * @return dato extraido
 */
int ProdConsMultSU::leer()
{
    // esperar bloqueado hasta que 0 < num_celdas_ocupadas
    if ( primera_libre == 0 )
        ocupadas.wait();

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

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

    // devolver valor
    return valor ;
}

// ----

/**
 * @brief Función llamada por el consumidor para insertar un dato
 */
void ProdConsMultSU::escribir( int valor )
{
    // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total
    if ( primera_libre == num_celdas_total )
        libres.wait();

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

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



WUOLAH

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

```

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

//=====
// FUNCIONES DE HEBRAS
//=====

/***
 * @brief Función de la hebra que produce datos
 * @param monitor : Puntero al monitor ProdCons
 */
void funcion_hebra_productora(MRef<ProdConsMultSU> monitor, int numHebra)
{
    for( unsigned i = 0 ; i < datos_producidos_hebra ; i++)
    {
        int valor = producir_dato(numHebra) ;
        monitor->escribir(valor);
    }
}

// -----
// **

/***
 * @brief Función de la hebra que consume datos
 * @param monitor : Puntero al monitor ProdCons
 */
void funcion_hebra_consumidora(MRef<ProdConsMultSU> monitor, int numHebra)
{
    for( unsigned i = 0 ; i < datos_consumidos_hebra ; i++)
    {
        int valor = monitor->leer();
        consumir_dato(valor, numHebra);
    }
}

//=====
// FUNCION MAIN
//=====

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

//DECLARACIÓN DEL MONITOR SU
MRef<ProdConsMultSU> monitor = Create<ProdConsMultSU>();

thread hebras_productoras[num_productores];
thread hebras_consumidoras[num_consumidores];

for(int i=0; i<num_productores; i++){
    hebras_productoras[i] = thread(funcion_hebra_productora, monitor, i);
}

```

```

    }

    for(int i=0; i<num_consumidores; i++){
        hebras_consumidoras[i] = thread(funcion_hebra_consumidora, monitor, i);
    }

    for(int i=0; i<num_productores; i++){
        hebras_productoras[i].join();
    }

    for(int i=0; i<num_consumidores; i++){
        hebras_consumidoras[i].join();
    }

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

```

La solución al problema de **ProdCons** con múltiples hebras productoras y consumidoras, utilizando un monitor *Señalar y Espera Urgente*, con política FIFO para el buffer, se basa en una combinación de los dos ejemplos anteriores, por lo que basta reemplazar en el fichero anterior el monitor con política FIFO para el caso de 1 hebra prod/cons.

4. PRÁCTICA2: MONITORES (SEÑALAR Y ESPERA URGENTE)

Tras lo visto en el *Seminario3*, en esta segunda práctica de la asignatura se hace énfasis en la implementación de soluciones a problemas que requieren uso de varias hebras utilizando como estrategia de control un Monitor Hoare (con semántica Señalar y Espera Urgente, explicada en el anterior seminario). Se vuelve a plantear el problema de los *Fumadores* y se introduce el problema de los *Lectores Y Escritores*, viéndose este último posiblemente en teoría aportando soluciones con semáforos.

NOTA: Para realizar los ejercicios correctamente, es necesario disponer de la implementación HoareMonitor realizada por los profesores de la asignatura.

4.1 EL PROBLEMA DE LOS FUMADORES

El planteamiento del problema de los fumadores sigue siendo exactamente el mismo que el de la versión de semáforos, se dispone de una serie de fumadores que necesitan un ingrediente específico para poder fumar, suministrado por un estanquero que los va produciendo y los coloca en una ventanilla de un estanco.

La solución a este problema utilizando un Monitor con semántica SU, mantiene la siguiente estructura: necesitamos un Monitor (Clase Hija de HoareMonitor), al que llamaremos *Estanco*, en el que competirán por entrar dos tipos de roles que son los que hemos mencionado anteriormente, *fumadores*, de los que habrá una cantidad fija *n*, y un único *estanquero*.

La Variable Permanente del monitor será el *mostrador*, un entero que cambiará su valor en función del ingrediente que el estanquero haya producido. El mostrador tomará valores de entre -1 y N, siendo N el número de fumadores que hay ejecutándose, y siendo -1 el valor que adquirirá el mostrador cuando éste se encuentre vacío. Está claro que con este criterio el fumador i-ésimo accederá al monitor de forma exclusiva cuando su ingrediente (valor i-ésimo) se encuentre disponible.

Habrá dos Colas Condición en este monitor, la primera se llamará *Mostrador Vacío*, en el que se quedará bloqueado esperando el estanquero si el mostrador NO se encuentra vacío, por lo que entonces no puede poner un nuevo ingrediente que ha producido. La segunda cola se llamará *ingredienteObtenido*, y será un vector de colas en la que se quedará esperando en la componente i-ésima, el fumador i-ésimo, esto sirve para que todos los fumadores se queden bloqueados esperando hasta que el ingrediente que necesitan esté disponible en el mostrador para que lo puedan retirar y proceder a fumar.

Para finalizar, a parte del constructor de la clase, tendremos tres métodos que serán los utilizados por las hebras. *obtenerIngrediente()* será utilizado por los fumadores y *ponerIngrediente()*, *esperarRecogidaIngrediente()* será utilizado por el estanquero, con uso similar a la proporcionada en la solución de los fumadores.

En conclusión, la solución para el problema de **Fumadores** con múltiples hebras fumadoras y un único estanquero, utilizando un monitor *Señalar y Espera Urgente* es la del fichero `fumadoresMon.cpp`:

```
/**  
 * @file fumadoresMonitor.cpp  
 * @author Daniel Pérez Ruiz  
 * @brief PRÁCTICA P2 - SISTEMAS CONCURRENTES Y DISTRIBUIDOS  
 */  
  
#include <iostream>  
#include <cassert>  
#include <thread>  
#include <mutex>  
#include <random> // dispositivos, generadores y distribuciones aleatorias  
#include <chrono> // duraciones (duration), unidades de tiempo  
#include "HoareMonitor.h"  
  
using namespace std;  
using namespace HM;  
  
//=====  
// VARIABLES COMPARTIDAS  
//=====  
  
const int num_fumadores = 3; //NUMERO DE FUMADORES TOTAL  
  
//=====  
  
/**  
 * @brief Genera un entero aleatorio uniformemente distribuido entre dos  
 * valores enteros, ambos incluidos  
 * @param<T,U> : Números enteros min y max, respectivamente  
 * @return un numero aleatorio generado entre min,max  
 */  
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);  
}  
  
//=====  
// MONITOR SU  
//=====
```



¿Quién te conoce mejor?

Tu madre

Tus amigos

Tu Wrapped

Si dudas, echa un ojo a tu Wrapped

#SPOTIFYWRAPPED



```
/*
 * @brief Esta clase representa un monitor con las siguientes características
 * -> Semática SU
 * -> Num Fumadores : Múltiples
 * -> Num Estanqueros : 1
 */
class Estanco : public HoareMonitor{
private:
    //VARIABLES PERMANENTES
    int mostrador; //Ventanilla sobre la que se obtendra el ingrediente

    //COLAS DE CONDICION
    CondVar mostradorVacio,           //Comprueba que podemos poner
    ingrediente
    ingredienteObtenido[num_fumadores]; //Comprueba qué ingrediente es el
    que está en mostrador

public:
    Estanco();                         //Constructor
    void obtenerIngrediente(int fumador); //Proceso de obtener el ingrediente
    por el fumador
    void ponerIngrediente(int ingr);    //Proceso de colocar el ingrediente
    por el estanquero
    void esperarRecogidaIngrediente(); //Proceso de espera del estanquero a
    mostrador Vacio
};

/*
 * @brief Constructor por defecto del monitor
 */
Estanco::Estanco(){
    mostrador = -1;
    mostradorVacio = newCondVar();

    for(int i=0; i<num_fumadores; i++)
        ingredienteObtenido[i] = newCondVar();
}

/*
 * @brief El fumador obtiene el ingrediente colocado por el estanquero
 * @param fumador : Número de fumador
 */
void Estanco::obtenerIngrediente(int fumador){
    //CONDICIÓN DE ESPERA: Hasta que el ingrediente no sea el que
    //necesita el fumador, se queda bloqueado
    if(mostrador != fumador)
        ingredienteObtenido[fumador].wait();

    //Comprobamos condición y actualizamos
    assert(0 <= mostrador && mostrador < num_fumadores);
    mostrador = -1; //Una vez se retira el ingrediente

    //Notificamos que el mostrador se encuentra vacío
    mostradorVacio.signal();
}

/*
 */
```

Quéééédate
y descubre
tu resumen
del año.

MÁS INFORMACIÓN



#SPOTIFYWRAPPED

Reservados todos los derechos. Queda permitida la impresión en su totalidad.
No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

WUOLAH

```

 * @brief El estanquero coloca el ingrediente en el mostrador
 * @param ingr : Número de ingrediente que coloca el estanquero
 */
void Estanco::ponerIngrediente(int ingr){
    //Comprobamos condición y actualizamos
    assert(0 <= ingr && ingr < num_fumadores);
    mostrador = ingr; //Se coloca ingrediente en el mostrador

    //Notificamos al fumador correspondiente que su ingrediente está disponible
    ingredienteObtenido[mostrador].signal();
}

/**
 * @brief El estanquero espera a que el mostrador esté vacío para
 * seguir colocando ingredientes
 */
void Estanco::esperarRecogidaIngrediente(){
    //CONDICIÓN DE ESPERA: Si el mostrador no está vacío se espera
    if(mostrador != -1)
        mostradorVacio.wait();

}

//-----
/***
 * @brief Simula la acción de producir un ingrediente, con un retardo aleatorio
 * de hebra
 * @return 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;
}

//=====
// FUNCIONES DE LAS HEBRAS
//=====

/***
 * @brief Función que simula la acción de fumar, con un retardo aleatorio de
 * hebra
 * @param num_fumador : Número de fumador que realiza la acción
 */

```

```

/*
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;
}

/**
 * @brief Función ejecutada por la hebra de estanquero
 */
void funcion_hebra_estanquero(MRef<Estanco> monitor){
    int i;

    while(true){
        i = producir_ingrediente();
        monitor->ponerIngrediente(i);
        monitor->esperarRecogidaIngrediente();
    }
}

/**
 * @brief Función ejecutada por la hebra del fumador
 */
void funcion_hebra_fumador(MRef<Estanco> monitor, int num_fumador){
    assert(0 <= num_fumador && num_fumador < num_fumadores);
    while(true){
        monitor->obtenerIngrediente(num_fumador);
        fumar(num_fumador);
    }
}

//-----
//=====
// FUNCION MAIN
//=====

int main(){
    //PARTE 0: DECLARACION DE HEBRAS
    thread estanquero;           //HEBRA DEL ESTANQUERO
    thread fumadores[num_fumadores]; //VECTOR DE HEBRAS DE LOS FUMADORES
    MRef<Estanco> monitor = Create<Estanco>(); //MONITOR SU

    string border =
"=====";

```

```

cout << border << endl << " PROBLEMA DE LOS FUMADORES - MONITOR SU " <<
endl << border << endl;

//PARTE 1: LANZAMIENTO DE LAS HEBRAS
estanquero = thread(funcion_hebra_estanquero, monitor);
for(int i=0; i<num_fumadores; i++){
    fumadores[i] = thread(funcion_hebra_fumador,monitor,i);
}

//PARTE 2: SINCRONIZACION ENTRE LAS HEBRAS
estanquero.join();
for(int i=0; i<num_fumadores;i++){
    fumadores[i].join();
}

return 0;
}

```

4.2 EL PROBLEMA DE LOS LECTORES Y ESCRITORES

A continuación se introduce otro problema típico de los Sistemas Concurrentes y Distribuidos, el problema de los lectores y escritores. El planteamiento atiende a la siguiente motivación: se quiere acceder a un recurso compartido, que puede ser por ejemplo, un fichero, y se desean leer y escribir datos en él de una manera mucho más rápida y eficiente de lo que se haría si se utilizara el procedimiento secuencial de carácter a carácter. En nuestro caso bastará con simular esa acción mediante acciones *leer* y *escribir* e introducir esperas aleatorias en unidades de milisegundos.

Por tanto, existirán dos roles dentro del sistema: los *lectores*, que son un conjunto de n hebras que se encargan de acceder al recurso y leer los datos, mostrando por pantalla junto con su identificador que efectivamente están leyendo; mientras que también están los *escritores*, que son un conjunto de m hebras que se encarga de acceder al recurso y escribir datos en él, siguiendo la misma rutina que los lectores, indicando naturalmente, que escriben.

Para que la solución se considere correcta y eficaz, las *Lecturas* deben poder realizarse Concurrentemente, mientras que una lectura y una escritura NO pueden. Las escrituras tampoco se pueden realizar concurrentemente, sólo una hebra debe ser capaz de efectuar dicha operación, las otras se tienen que quedar esperando.

Este monitor se puede diseñar de varias formas, la elegida por parte de las prácticas es aquella que da prioridad a las hebras lectoras (habrá más probabilidad de que una hebra lectora gane la exclusión mutua frente a una hebra escritora), aunque se puede diseñar dando prioridad a las escritoras, e incluso intentando equilibrar y no dar ventaja a ningún rol (una forma pj, sería implementar orden de llegada).

Por tanto, la estructura del monitor será la siguiente: Habrá dos *variables permanentes*, la primera será un contador de lectores, $nLec$, que contará el total de lectores que leerán el recurso, mientras que la segunda variable será un booleano, llamado *escrib* que advertirá si existe un escritor utilizando el recurso y por tanto, está escribiendo. Este es el motivo de por qué se le va a dar a esta solución prioridad a los lectores. El valor que puede tomar la variable $nLec$ será un número positivo, que se irá incrementando cada vez que una hebra lectora solicite leer, y una vez que ésta termine su tarea, se decrementará el contador. No se señalará a un escritor hasta que el número de lectores que refleje esta variable sea 0, mientras que, cuando llegue una nueva solicitud de lectura, se señalará a otra hebra LECTORA que se encontraba esperando en la cola condición para adquirir su acceso exclusivo, por lo que adquirirá mucha más prioridad para



¿Quién te conoce mejor?

Tu madre Tus amigos Tu Wrapped

#SPOTIFYWRAPPED

Si dudas, echa un ojo a tu Wrapped

¿Tu madre,
tus amigos
o tu
Wrapped?

Descubre
quién te conoce
mejor en tu
resumen del año.



MÁS INFORMACIÓN



entrar una hebra lectora frente a una escritora (para un mayor entendimiento, consultar implementación y explicación de los métodos).

Habrá dos *Colas Condición* dentro de este monitor. Las colas de *lectura* y *escritura*, donde quedarán esperando los lectores y escritores respectivamente.

Haré énfasis en la implementación de los métodos exportados de este monitor para que se entienda con la mayor claridad posible el motivo de la prioridad, *algo bastante importante en caso de que caiga en un examen*. Cada rol utilizará dos métodos, que harán de "cerrojo" para la simulación de lectura/escritura, garantizando así que sólo se hace una operación de las dos y no simultáneamente, permitiendo sólo que se puedan realizar simultáneamente operaciones lectura/lectura entre este tipo de hebras. Estos métodos serán *iniLectura*, *finLectura* y *iniEscritura*, *finEscritura*. Cuando una hebra lectora desee leer el recurso compartido, lo primero que hará será ejecutar el método exportado *iniLectura*. Si queremos que la restricción que se impuso en párrafos anteriores se cumpla, una hebra lectora que invoque a *iniLectura*, sólo se quedará esperando bloqueada en su respectiva cola condición SI EXISTE UNA HEBRA ESCRITORA escribiendo en el recurso, en caso contrario, incrementará el número de lectores que están leyendo *Concurrentemente*, y señalará a otra hebra lectora bloqueada dentro de la cola para que se active. Cuando se ha terminado de realizar el acto de *Leer()*, el contador de lectores se decrementará, y la hebra lectora saldrá del monitor. Si era el último lector en salir (pone el contador a 0), se despertará a un escritor que estuviese bloqueado en su cola condición (si lo hay).

El funcionamiento en los métodos exportados que utilizan las hebras escritoras es bastante similar. En este caso, cuando una hebra escritora invoca el método *iniEscritura*, se comprueba previamente que no hay lectores y tampoco que haya otro escritor realizando una operación de escritura, para así ceñirnos a las restricciones del ejercicio. Si esto no se cumple, la hebra se queda esperando en su respectiva cola condición, y en caso contrario, la variable booleana que cuenta si hay escritores en el sistema, se pondrá a true. Cuando se invoca *finEscritura*, se pone a false la variable booleana, y se despertará a un lector siempre que la cola de espera no esté vacía, y en caso contrario, se desbloquea a una hebra escritora.

En conclusión, la solución al problema de los **Lectores y Escritores** utilizando un monitor *Señalar* y *Espera Urgente* con prioridad a los lectores, es el del fichero *lectEscrMon.cpp*:

```
/*
 * @file lectEscr.cpp
 * @author Daniel Pérez Ruiz
 * @brief PRÁCTICA P2 - SISTEMAS CONCURRENTES Y DISTRIBUIDOS
 */

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

using namespace std;
using namespace HM;

//=====
// VARIABLES COMPARTIDAS
//=====

const int num_lectores = 3, //NUMERO DE LECTORES
```

```

        num_escritores = 3; //NUMERO DE ESCRITORES

mutex mtx;

//=====

/***
 * @brief Genera un entero aleatorio uniformemente distribuido entre dos
 * valores enteros, ambos incluidos
 * @param<T,U> : Números enteros min y max, respectivamente
 * @return un numero aleatorio generado entre min,max
 */
template<int min, int max> int aleatorio(){
    static default_random_engine generador( (random_device())() );
    static uniform_int_distribution<int> distribucion_uniforme(min, max);
    return distribucion_uniforme(generador);
}

//-----

//-----
// FUNCIONES SIMULADAS
//-----


/***
 * @brief Funcion de escribir simulada
 * @param numEscritor : Número de escritor que realiza la acción
 */
void escribir(int numEscritor){
    chrono::milliseconds duracion_escritura(aleatorio<20,200>());

    mtx.lock();
    cout << "[Escritor " << numEscritor << "]: Escribiendo datos... ("
        << duracion_escritura.count() << " milisegundos)" << endl;
    mtx.unlock();

    this_thread::sleep_for(duracion_escritura);
}

/***
 * @brief Funcion de leer simulada
 * @param numLector : Número de lector que realiza la acción
 */
void leer(int numLector){
    chrono::milliseconds duracion_lectura(aleatorio<20,200>());

    mtx.lock();
    cout << "[Lector " << numLector << "]: Leyendo datos... ("
        << duracion_lectura.count() << " milisegundos)" << endl;
    mtx.unlock();

    this_thread::sleep_for(duracion_lectura);
}

//-----

```

```

//=====================================================================
// MONITOR SU
//=====================================================================

/**
 * @brief Esta clase representa un monitor con las siguientes características
 * -> Semántica SU
 * -> Num Lectores : Múltiples
 * -> Num Escritores : Múltiples
 */
class Lec_Esc : public HoareMonitor{
private:
    //VARIABLES PERMANENTES
    int n_lec;      //Número de lectores
    bool escrib;    //Comprueba que hay un escritor activo

    //VARIABLES DE CONDICION
    CondVar lectura,    //Cola de lectores
              escritura; //Cola de escritores

public:
    Lec_Esc();           //Constructor por defecto
    void ini_lectura(); //Función de inicio de lectura
    void fin_lectura(); //Función de fin de lectura

    void ini_escritura(); //Función de inicio de escritura
    void fin_escritura(); //Función de fin de escritura
};

/**
 * @brief Constructor del monitor
 */
Lec_Esc::Lec_Esc(){
    n_lec = 0;
    escrib = false;

    lectura = newCondVar();
    escritura = newCondVar();
}

/**
 * @brief Función de inicio de lectura
 */
void Lec_Esc::ini_lectura(){
    //COMPROBACIÓN DE SI HAY ESCRITOR
    if(escrib)
        lectura.wait(); //ESPERAMOS A LECTURA

    //REGISTRAR UN LECTOR MÁS
    n_lec++;

    //DESBLOQUEO EN CADENA DE POSIBLES LECTORES
    lectura.signal();
}

/**
 * @brief Función de fin de lectura
 */

```

```

void Lec_Esc::fin_lectura(){
    //REGISTRAR UN LECTOR MENOS
    n_lec--;

    //SI ES EL ÚLTIMO LECTOR, DESBLOQUEAR
    //UN ESCRITOR
    if(n_lec == 0)
        escritura.signal();
}

/***
 * @brief Función de inicio de escritura
 */
void Lec_Esc::ini_escritura(){
    //
    if((n_lec > 0) or escrib)
        escritura.wait();

    escrib = true;
}

/***
 * @brief Funcion de fin de escritura
 */
void Lec_Esc::fin_escritura(){
    //REGISTRAR QUE YA NO HAY ESCRITOR
    escrib = false;

    //SI HAY LECTORES, DESPERTAR UNO
    if(!lectura.empty())
        lectura.signal();
    //SI NO HAY LECTORES, DESPERTAR UN ESCRITOR
    else
        escritura.signal();
}

//=====
// FUNCIONES DE LAS HEBRAS
//=====

/***
 * @brief Funcion de la hebra lectora
 * @param monitor : Puntero a monitor
 * @param numLector : Numero de lector
 */
void funcion_hebra_lector(MRef<Lec_Esc> monitor, int numLector){
    while(true){
        //RETARDO ALEATORIO
        chrono::milliseconds retardo(aleatorio<20,200>());
        this_thread::sleep_for(retardo);

        //PROCEDIMIENTO
        monitor->ini_lectura();
        leer(numLector);
        monitor->fin_lectura();
    }
}

```



Ábrete la Cuenta Online de BBVA y
llévate 1 año de Wuolah PRO

cómo??
→



1/6

Este número es
indicativo del
riesgo del
producto, siendo
1/6 indicativo de
menor riesgo y
6/6 de mayor
riesgo.

BBVA está
adherido al
Fondo de
Garantía de
Depósitos de
Entidades de
Crédito de
España.
La cantidad
máxima
garantizada es
de 100.000 euros
por la totalidad
de los depósitos
constituidos
en BBVA por
persona.

ventajas

PRO



Dí adiós a la publi
en los apuntes y
en la web



Participa gratis
en todos los
sorteos



Descarga
carpetas
completas

estudia sin publi
WUOLAH PRO

```
/**  
 * @brief Funcion de la hebra escritora  
 * @param monitor : Puntero a monitor  
 * @param numEscritor : Numero de escritor  
 */  
void funcion_hebra_escritor(MRef<Lec_Esc> monitor, int numEscritor){  
    while(true){  
        //RETARDO ALEATORIO  
        chrono::milliseconds retardo(aleatorio<20,200>());  
        this_thread::sleep_for(retardo);  
  
        //PROCEDIMIENTO  
        monitor->ini_escritura();  
        escribir(numEscritor);  
        monitor->fin_escritura();  
    }  
}  
  
//-----  
  
//=====//  
// FUNCION MAIN  
//=====//  
  
int main(){  
    //PARTE 0: DECLARACION DE HEBRAS  
    assert(0 < num_lecadores && 0 < num_escritores);  
    thread lectores[num_lecadores];  
    thread escritores[num_escritores];  
    MRef<Lec_Esc> monitor = Create<Lec_Esc>();  
  
    string border =  
"=====";  
    cout << border << endl << " LECTORES / ESCRITORES - MONITOR SU " << endl <<  
border << endl;  
  
    //PARTE 1: LANZAMIENTO DE LAS HEBRAS  
    for(int i=0; i<num_lecadores; i++){  
        lectores[i] = thread(funcion_hebra_lector, monitor, i);  
    }  
  
    for(int i=0; i<num_escritores; i++){  
        escritores[i] = thread(funcion_hebra_escritor, monitor, i);  
    }  
  
    //PARTE 2: SINCRONIZACION ENTRE LAS HEBRAS  
    for(int i=0; i<num_lecadores; i++){  
        lectores[i].join();  
    }  
  
    for(int i=0; i<num_escritores; i++){  
        escritores[i].join();  
    }  
  
    return 0;  
}
```

WUOLAH

4. SEMINARIO3: INTRODUCCIÓN A PASO DE MENSAJES CON MPI

En este seminario se introduce al estándar MPI que permite la programación paralela mediante paso de mensajes. El concepto de *mensaje*, cobra aquí unas nociones bastante abstractas pero muy precisas de lo que se va a realizar aquí. Se tendrán una serie de procesos que ejecutarán concurrentemente, con un espacios independientes y contextos diferentes entre cada procesos, que se comunicarán entre sí utilizando mensajes, que no serán ni más ni menos que bytes con información de interés. Tras tener montadas todas las instalaciones necesarias para utilizar esta herramienta, y habiendo estudiado previamente las diferentes funciones, atributos y variables que se pueden utilizar, explicaremos los ejercicios que se proponen a continuación.

4.1 EL PROBLEMA MÁS SENCILLO: HOLA MUNDO

Ya que todos los programadores, siempre empezamos programando el típico programa `Hello World` u `Hola Mundo`. Utilizaremos esto para crear un programa con varios procesos que impriman este mensaje, junto con el identificador de proceso.

Recordemos en lo que sigue que será necesario habilitar en la función `main()` de nuestros programas, una entrada con argumentos, recordemos que para ello basta poner como parámetros de `main` `int main(int argc, char* argv[])`, una para contar el número de argumentos que se le pasarán al programa y el segundo un vector en el que se almacenarán todos ellos.

Toda esta información se pasará por referencia al método `MPI_Init()` que será el que inicie todas las rutinas necesarias para utilizar el estándar MPI. A continuación, se creará el comunicador con la función `MPI_Comm_Size()`. En todos los programas siempre utilizaremos **MPI_Comm_World**, el comunicador universal, y pasaremos como segundo argumento una variable para almacenar el número de procesos. Para finalizar, se tiene que asignar un número identificador a cada proceso lanzado con la función `MPI_Comm_Rank()`, para ello declararemos una variable que pasaremos por referencia a dicha función, y también el comunicador universal, y se guardará el valor en dicha variable del número identificador que ha asignado MPI a dicho proceso.

Dicho esto, si queremos programar "Hola Mundo" utilizando el estándar MPI, viéndose por pantalla dicho mensaje y el número identificador de proceso al lado, basta seguir esta implementación, que es la que sugiere el esquema de las diapositivas en el fichero `holaMundo.cpp`:

```
#include <mpi.h>
#include <iostream>
using namespace std;

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

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

    cout << "Hola desde proceso " << id_propio << " de " << num_procesos_actual <<
        endl ;
}
```

```

    MPI_Finalize();
    return 0;
}

```

Es importante destacar que para terminar adecuadamente un programa que utilice el estándar MPI, debemos finalizar las líneas de código con `MPI_Finalize()`, ya que esta función se encarga de la terminación de los procesos y de limpiar la memoria utilizada.

4.2 ENVÍO Y RECEPCIÓN DE MENSAJES CON MPI

Todo lo que se ha introducido (o más bien repasado de lo que se expone en las diapositivas de clase) es lo básico para permitir lanzamientos de programas que utilicen procesos. Para completar esta introducción, es necesario establecer cómo se comunican los procesos entre sí dentro del comunicador que hemos definido. Para ello se utilizarán las familias de funciones de envío y recibo, con sus respectivos parámetros.

En el ejemplo que vamos a mostrar a continuación veremos una forma sencilla de comprender el funcionamiento de estas funciones. Este programa se lanzará con dos procesos, a los que le asignaremos un rol, que a su vez estará representado con un valor numérico que nosotros definiremos en tiempo de compilación.

Así pues, definiremos el proceso *Emisor*, que tendrá un id con valor 0, y otro proceso *Receptor*, que tendrá un id con valor 2. Hay que tener en cuenta que este programa necesita exactamente 2 procesos, por lo que si el usuario lanza el programa con un número mayor de procesos que el que se espera, éste puede no funcionar como se espera, por lo que sería conveniente añadir una medida de seguridad dentro de nuestro ejecutable para salir del programa si esto no se verifica. Es suficiente con añadir un condicional y una constante en tiempo de compilación justo después del inicio de MPI.

Recordemos del ejemplo anterior que para ver "qué proceso somos", lo almacenamos en una variable que llamamos *id_Propio*. Ahora nos será muy útil para la premisa que hemos dicho anteriormente, para así identificar los roles, usando un simple condicional, donde dentro del mismo escribiremos el código de dicho rol. Por tanto, si soy el emisor, enviaré el mensaje, que en este caso nos bastará con enviar un número entero para simplificar las cosas (véase los argumentos necesarios para la función `MPI_Send()`); mientras que si soy el receptor capto el mensaje y muestro lo que he recibido con la función `MPI_Recv()`.

La solución es la que se propone en las diapositivas con el fichero `sendrecv1.cpp`:

```

#include <mpi.h>
#include <iostream>

using namespace std; // incluye declaraciones de funciones, tipos y ctes. MPI

const int id_emisor      = 0,   // identificador de emisor
         id_receptor     = 1,   // identificador de receptor
         num_procesos_esperado = 2; // numero de procesos esperados

int main( int argc, char *argv[] )
{
    int      id_propio,           // identificador de este proceso
            num_procesos_actual; // numero de procesos lanzados

    MPI_Init( &argc, &argv );                                // inicializa MPI

```

```

MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );           // averiguar mi ident
MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual ); // averiguar n.procs

if ( num_procesos Esperado == num_procesos_actual ) // si num. procs. ok
{
    // hacer envío o recepción (según id_propio)
    if ( id_propio == id_emisor ) // emisor: enviar
    {
        int valor_enviado = 100 ; // buffer del emisor (tiene 1 entero: MPI_INT)

        MPI_Send( &valor_enviado, 1, MPI_INT, id_receptor, 0, MPI_COMM_WORLD );
        cout << "Proceso " << id_propio << " ha enviado " << valor_enviado << endl
    ;
    }
    else // receptor: recibir
    {
        int valor_recibido ; // buffer del receptor (tiene 1 entero: MPI_INT)
        MPI_Status estado ; // estado de la recepción

        MPI_Recv( &valor_recibido, 1, MPI_INT, id_emisor, 0, MPI_COMM_WORLD,
&estado );
        cout << "Proceso " << id_propio << " ha recibido " << valor_recibido <<
endl ;
    }
}
else if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
    // escribir un mensaje de error
cerr
    << "el número de procesos esperados es: " << num_procesos Esperado <<
endl
    << "el número de procesos en ejecución es: " << num_procesos_actual <<
endl
    << "(programa abortado)" << endl ;

MPI_Finalize( ); // terminar MPI: debe llamarse siempre por cada proceso.
return 0;          // terminar proceso
}

```

Complicaremos este programa definiendo un esquema de envíos entre 2 procesos o más. La motivación es la siguiente. Se necesitará un mínimo de 2 procesos para poder lanzar adecuadamente este ejemplo. El primer proceso, que definiremos como P_0 , se encargará de pedir al usuario un dato, en el que optaremos como en el caso anterior por un número entero. Este proceso enviará al proceso P_1 el dato, que lo mostrará en pantalla, y enviará el dato al siguiente proceso (si existe) P_2 , así sucesivamente hasta llegar al último proceso P_{N-1} . Todo esto lo haremos en un bucle que terminará cuando se reciba un número negativo.

Esta es la implementación del programa que se encuentra en el fichero `sendrecv2.cpp`:

```

#include <iostream>
#include <mpi.h>

using namespace std;

const int num_min_procesos = 2 ;

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

```

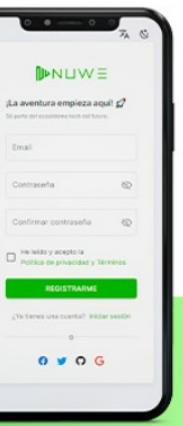
Encuentra el trabajo de tus sueños



Escanéame y obtén más info!!



```
{  
    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 );  
  
    //Comprobamos que el numero actual satisface el numero min de procs esperado  
    if ( num_min_procesos <= num_procesos_actual )  
    {  
        const int id_anterior = id_propio-1, // ident. proceso anterior  
                 id_siguiente = id_propio+1; // ident. proceso siguiente  
        int valor; // valor recibido o leído, y enviado  
        MPI_Status estado; // estado de la recepción  
  
        do  
        {  
            // recibir o leer valor  
            if ( id_anterior < 0 ) // si soy el primero (no hay anterior)  
                cin >> valor; // pedir valor por teclado (-1 para acabar)  
            else // si no soy el primer proceso: recibirllo  
                MPI_Recv( &valor, 1, MPI_INT, id_anterior, 0, MPI_COMM_WORLD, &estado );  
  
            // imprimir valor  
            cout << "Proc." << id_propio << ": recibido/leído: " << valor << endl ;  
  
            // si no soy el último (si hay siguiente): enviar valor,  
            if ( id_siguiente < num_procesos_actual )  
                MPI_Send( &valor, 1, MPI_INT, id_siguiente, 0, MPI_COMM_WORLD );  
        }  
        while( valor >= 0 ); // acaba cuando se teclea un valor negativo  
    }  
    else  
        cerr << "Se esperaban 2 procesos como mínimo, pero hay 1." << endl ;  
  
    MPI_Finalize();  
    return 0;  
}
```



4.3 ENVÍO Y RECEPCIÓN DE MENSAJES SEGUROS CON MPI

Las funciones que hemos utilizado en los ejemplos anteriores, `MPI_Send()` y `MPI_Recv()`, son envíos y recepciones en modo síncrono, es decir, si se produce un envío (con todos sus argumentos bien definidos), hay una llamada a un recibimiento al instante esperando a recibir el mensaje que contiene todos esos argumentos, lo que se conoce como una *cita*. Ahora, utilizaremos `MPI_Ssend()`, una función dentro de la familia de envíos que además de ser síncrono, es seguro, ya que el buffer de datos que se utiliza para transferir los bytes, vuela su contenido en otra dirección de memoria independiente, por lo que si se vuelve a escribir en el buffer los datos que se querían enviar se encuentran seguros libres de cualquier cambio.

El siguiente ejemplo que vamos a tratar (que puede ser bastante útil para las prácticas si se comprende adecuadamente), podemos ver la importancia de utilizar `MPI_Ssend()`. En esta ocasión, tendremos un número par de procesos como argumento válido para poder lanzar el programa, ya que lo que haremos será tratar la comunicación por pares de procesos, en los que



se enviarán y recibirán información mutuamente y sólo entre ellos. Esto puede dar a lugar a situaciones no deseables si no se trata con cuidado la política de envíos y recepciones, como el colapso del programa entero, más bien conocido como **interbloqueo**. Si suponemos que todos los procesos *envían* el flujo de bytes y después *reciben* el flujo del otro proceso, puede ser que ambos se queden esperando a una recepción que nunca llegará, ya que como se ha visto en teoría y en las diapositivas del seminario, en un envío síncrono de información, cuando se invoca un envío con `MPI_Ssend()` (el de nuestro ejercicio), el proceso (emisor) queda bloqueado y no continúa con la ejecución de su trozo de código correspondiente hasta que el proceso que debe recibir el mensaje (destinatario) efectivamente lo recoja, diciendo así que *está esperando a su cita*.

La solución para este ejercicio es tomar una estrategia diferente, suponiendo por ejemplo que los procesos con identificación par *envían* y *después reciben*, y los procesos con identificación impar *reciben* y *después envían*, asegurándonos así que todas las "citas" se concretan y no se produce ninguna situación de interbloqueo.

El diseño de este ejercicio en código es el siguiente, el del fichero `intercambioSincrono.cpp`:

```
#include <iostream>
#include <mpi.h>
using namespace std;

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_actual % 2 == 0 ) // si número de procesos correcto (par)
    {
        int      valor_enviado = id_propio*(id_propio+1), // dato a enviar
                 valor_recibido,
                 id_vecino ;
        MPI_Status estado ;

        if ( id_propio % 2 == 0 ) // si proceso par: enviar y recibir
        {
            id_vecino = id_propio+1 ; // el vecino es siguiente
            MPI_Ssend( &valor_enviado, 1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD );
            MPI_Recv ( &valor_recibido, 1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD,
                       &estado );
        }
        else // si proceso impar: recibir y enviar
        {
            id_vecino = id_propio-1 ; // el vecino es el anterior
            MPI_Recv ( &valor_recibido, 1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD,
                       &estado );
            MPI_Ssend( &valor_enviado, 1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD );
        }
        cout << "Proceso "<< id_propio << " recibe "<< valor_recibido
            << " de "<< id_vecino << endl ;
    }
    else if ( id_propio == 0 ) // si n.procs. impar, el primero da error
        cerr << "Se esperaba un número par de procesos, pero hay "
            << num_procesos_actual << endl ;
}
```

```

    MPI_Finalize();
    return 0;
}

```

4.4 SONDEO DE MENSAJES CON MPI

MPI dispone de dos métodos muy útiles que podemos invocar para comprobar si un determinado proceso (el que invoca alguno de los mismos), tiene pendiente recibir un mensaje, y además, obtener los datos asociados a dicho mensaje, como el destinatario, el tipo de dato y su tamaño y la etiqueta que tiene. Tenemos a `MPI_Probe()`, que hace que el proceso que lo invoca quede bloqueado hasta que haya al menos un mensaje para recibir, y `MPI_IProbe()`, que simplemente consulta en el momento si hay algún mensaje para recibir.

Tras ver convenientemente los parámetros que reciben estos métodos a través de la documentación entregada en prácticas por parte del profesor, aquí mostramos un ejemplo de uso del método `MPI_Probe()` en el fichero `ejemplo_probe.cpp`.

Vamos a explicar de qué trata este código. La motivación es la siguiente: queremos tener una serie de n procesos entre los cuales existirán dos roles bien definidos: el rol *Impresor*, que sólo lo tendrá un único proceso, y el rol *Emisor*, que lo tendrán el resto de procesos. La idea consiste en hacer que el proceso impresor imprima la vida útil de los procesos emisores, es decir, se encargará de imprimir cuándo el proceso se inicia, cuántos milisegundos se va a dormir el proceso, cuándo se despierta, y cuándo finaliza el proceso. Los procesos emisores dormirán (una duración aleatoria) una serie de veces definida por una constante en tiempo de ejecución, y envían estos mensajes al proceso *Impresor* a través de una llamada al método `MPI_Ssend()`, en el que hay que especificar previamente el número de caracteres que se envían para que el mensaje sea consistente y correcto, ya que por la parte del proceso que va a recibir dicho mensaje tiene que saber exactamente qué es lo que recibe, y CUÁNTO va a recibir, como hemos mencionado anteriormente.

El proceso impresor tiene varias funciones que realizar, la primera de ellas es ser consciente del total que mensajes que va a recibir, que será un compendio del número de emisores por el total que éstos emiten (los dos del inicio/fin del proceso mas los que se encuentren dentro del bucle). Aquí es donde entra en juego la función `MPI_Probe()`. Es natural que utilicemos esta función para sondear ya que el proceso impresor no tiene por qué estar despierto si no tiene nada que hacer, por lo que se quedará bloqueado esperando a la llegada del mensaje. Para cada mensaje "que se espere recibir", se invoca esta función, que almacenará los metadatos en la estructura de datos `MPI_Status`. A continuación, se almacena en un entero cuántos caracteres se esperan leer con el mensaje que se espera recibir. Se crea un buffer con memoria dinámica para almacenar exactamente la cadena y nada más, y por último, se recibe con `MPI_Recv()`. Es importante liberar la memoria dinámica ocupada por el buffer para poder reservar otra para otro mensaje diferente.

```

#include <iostream>
#include <random> // para 'aleatorio'
#include <thread> // this_thread::sleep_for
#include <chrono> // chrono::duration, chrono::milliseconds
#include <cstring> // strlen
#include <cstdio> // sprintf
#include <mpi.h>

using namespace std ;

```

```

// constantes definibles
const int
    id_impresor = 0,    // el proceso que recibe los mensajes es el 0
    num_iteraciones_por_emisor = 30 ;

// -----
// 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_impresor( int num_emisores )
{
    MPI_Status estado ;
    int num_chars_rec ;
    // total mensajes: 2 por emisor, + 2 por cada una de sus iteraciones
    const int num_total_msgs = num_emisores*(2*num_iteraciones_por_emisor+2) ;

    cout << "inicio del impresor" << endl ;
    for( int i = 0 ; i < num_total_msgs; i++ )
    {
        // espera un mensaje de cualquier emisor, sin recibirlo
        MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &estado );

        // leer el numero de chars del mensaje
        MPI_Get_count( &estado, MPI_CHAR, &num_chars_rec );

        // reservar memoria dinámica para los caracteres (incluyendo 0 al final)
        char * buffer = new char[num_chars_rec+1] ;

        // recibir el mensaje en el buffer y añadir un cero al final
        // IMPORTANTE: especificar exactamente mismo emisor detectado en el Probe
        MPI_Recv( buffer, num_chars_rec, MPI_CHAR, estado.MPI_SOURCE, MPI_ANY_TAG,
        MPI_COMM_WORLD, &estado );
        buffer[num_chars_rec] = 0 ;

        // imprimir la cadena recibida
        cout << buffer << endl ;

        // liberar memoria dinámica ocupada por el buffer
        delete [] buffer ;
    }
    cout << "fin del impresor" << endl ;
}

// -----


void enviar_cadena( const char * cadena )
{
    int num_chars = strlen( cadena );

```

Encuentra el trabajo de tus sueños

Participa en retos y competiciones de programación



Escanéame y
obtén más info!!

```
        MPI_Ssend( cadena, num_chars, MPI_CHAR, id_impresor, 0, MPI_COMM_WORLD );
    }

// ----

void funcion_emisores( int id_propio )
{
    const int maxlon = 1024 ; // máxima longitud de la cadena en 'str'
    char str[maxlon] ;

    snprintf(str,maxlon,"inicio del proceso número: %d", id_propio);
    enviar_cadena( str );

    for ( int i = 0 ; i < num_iteraciones_por_emisor ; i++ )
    {
        int dur_ms = aleatorio<20,200>();
        snprintf(str,maxlon,"proceso %d: voy a dormir %d
milisegundos",id_propio,dur_ms);
        enviar_cadena( str );
        this_thread::sleep_for( chrono::milliseconds( dur_ms ) );
        enviar_cadena("ya he dormido");
    }
    snprintf(str,maxlon,"fin del proceso número: %d", id_propio);
    enviar_cadena( str );
}

// ----

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_actual > 1 ) // si num. proc. ok
    {
        // ejecutar la función correspondiente al id. propio
        if ( id_propio == id_impresor )
            funcion_impresor( num_procesos_actual-1 );
        else
            funcion_emisores( id_propio );
    }
    else // si el numero de procesos es incorrecto
        cerr << "error: este programa se debe lanzar con al menos dos procesos" <<
endl ;

    MPI_Finalize();
    return 0;
}
```



WUOLAH

En el siguiente ejemplo veremos el uso de `MPI_IProbe()` para notar las diferencias entre el uso del método que bloquea al proceso que lo invoca hasta que éste reciba un mensaje. La motivación de este problema es muy similar al anterior, por lo que sólo comentaré qué cambios se han producido.

En este caso el proceso, que llamaremos *Receptor* en vez de *Impresor*, consultará id por id de procesos emisores si éstos han enviado un mensaje. Esto quiere decir que existirá un bucle en orden creciente de número de id, para el que se invocará `MPI_IProbe()` haciendo la pregunta. Si se responde de forma afirmativa (hay mensaje), entonces no se sigue comprobando más. En caso contrario, lo que hará el proceso receptor esta vez será permitir recibir un mensaje de cualquier proceso utilizando el comodín `MPI_ANY_TAG`. (Nota: Hay que tener cuidado con esto, ya que hay que saber siempre qué es exactamente lo que se espera [tipo de dato], y qué cantidad [size en bytes]).

El siguiente ejemplo es el del fichero `ejemplo_iprobe.cpp`.

```
#include <iostream>
#include <random> // para 'aleatorio'
#include <thread> // this_thread::sleep_for
#include <chrono> // chrono::duration, chrono::milliseconds
#include <cstring> // strlen
#include <mpi.h>

using namespace std ;
using namespace this_thread ; // sleep_for
using namespace chrono ; // milliseconds, seconds, etc..

// constantes definibles
const int
    id_receptor = 0 ; // el proceso que recibe los mensajes es el 0

const int
    num_mensajes_por_emisor = 100 ;

// -----
// plantilla de función para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
//-----

template< int min, int max > int aleatorio()
{
    static default_random_engine generador( (random_device())() );
    static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
    return distribucion_uniforme( generador );
}

// -----
// función que ejecuta el receptor: recibe 'n' mensajes de cada receptor

void funcion_receptor( int id_min, int id_max )
{
    int num_emisores = id_max - id_min +1 ; // total de emisores
    int total_mensajes = num_mensajes_por_emisor*num_emisores ; // total msgs
    int cuenta[num_emisores]; // número de mensajes recibidos por emisor

    // inicializar la cuenta:
```

```

for( unsigned i = 0 ; i < num_emisores ; i++ )
    cuenta[i] = 0 ;

cout << "inicio del receptor" << endl ;

for( int i = 0 ; i < total_mensajes ; i++ )
{
    MPI_Status estado ;      // estado de la recepción
    int      hay_mensaje, // ==0 si no hay mensajes, >0 si hay mensajes
            id_emisor , // identificador de emisor a recibir/recibido
            valor ;      // valor recibido

    // comprobar si hay mensajes, en orden creciente de los posibles emisores
    // (id_emisor toma valores entre id_min e id_max, ambos incluidos)
    for( id_emisor = id_min ; id_emisor <= id_max ; id_emisor++ )
    {
        // recibir mensaje del emisor
        MPI_Iprobe( id_emisor, MPI_ANY_TAG, MPI_COMM_WORLD, &hay_mensaje,
&estado ) ;
        if ( hay_mensaje ) break ;
    }

    // comprobar si hay mensajes pendientes o no
    if ( hay_mensaje )           // si no hay mensaje:
        cout << "Hay al menos un mensaje, del emisor: " << id_emisor << endl ;
    else
    {
        // si no hay mensaje
        id_emisor = MPI_ANY_SOURCE ; // aceptar de cualquiera
        cout << "No hay mensajes. Recibo de cualquiera." << endl ;
    }

    // recibir el mensaje e informar
    MPI_Recv( &valor, 1,MPI_INT, id_emisor, 0, MPI_COMM_WORLD, &estado );
    id_emisor = estado.MPI_SOURCE ;
    cout << "Receptor ha recibido de " << id_emisor << endl ;

    // incrementar cuenta e imprimir estadísticas
    cuenta[id_emisor-id_min] ++ ;
    cout << "Núm. de mensajes recibidos de cada emisor: " ;
    for( unsigned i = 0 ; i < num_emisores ; i++ )
        cout << cuenta[i] << ", " ;
    cout << endl << endl ;

    // dormir un poco
    sleep_for( milliseconds( aleatorio<8,12>() ) );
}

}

// -----
// función que ejecutan los emisores: emite 'n' mensajes a receptor

void funcion_emisores( int id_propio )
{
    for( unsigned i = 0 ; i < num_mensajes_por_emisor ; i++ )
    {
        int valor ;
        sleep_for( milliseconds( aleatorio<8,12>() ) );
        MPI_Ssend( &valor, 1,MPI_INT, id_receptor, 0, MPI_COMM_WORLD ) ;
    }
}

```

```

        cout << "acaba emisor : " << id_propio << 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 ( 1 < num_procesos_actual ) // si num. proc. ok
    {
        // ejecutar la función correspondiente al id. propio
        if ( id_propio == id_receptor )
            funcion_receptor( id_propio+1, num_procesos_actual-1 );
        else
            funcion_emisores( id_propio );
    }
    else // si el numero de procesos es incorrecto
        cerr << "error: este programa se debe lanzar con al menos dos procesos" <<
endl ;

    MPI_Finalize();
    return 0;
}

```

4.5 COMUNICACIÓN INSEGURA

Las operaciones de envío y recepción de datos entre procesos tienen una versión que denominaremos "insegura", ya que tras invocarlas, retornan el control de la ejecución al proceso, a diferencia de los otros métodos que hemos visto que son bloqueantes y esperan a una "cita". El control de la transacción, por tanto, se hace más difícil, por lo que necesitaremos otro recurso para comprobar si la operación se completó, a través del método `MPI_Wait()`.

En el ejemplo de aquí se utiliza un `MPI_Request` que pasaremos a la función `MPI_Wait()` para comprobar si la operación ha finalizado. Hay que notar que aquí las operaciones de envío y recepción pueden hacerse en cualquier orden.

El ejemplo es el del fichero `intercambio_nobloq.cpp`:

```

#include <mpi.h>
#include <iostream>
using namespace std;

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 );

```



¿Quién te conoce mejor?

Tu madre

Tus amigos

Tu Wrapped

Si dudas, echa un ojo a tu Wrapped

#SPOTIFYWRAPPED



```
if ( num_procesos_actual % 2 == 0 )
{
    int         valor_enviado = id_propio*(id_propio+1), // dato a enviar
                valor_recibido,
                id_vecino ;
    MPI_Status  estado ;
    MPI_Request ticket_envio,
                  ticket_recepcion;

    if ( id_propio % 2 == 0 )
        id_vecino = id_propio+1 ;
    else
        id_vecino = id_propio-1 ;

    // las siguientes dos llamadas pueden aparecer en cualquier orden
    MPI_Irecv( &valor_recibido, 1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD,
&ticket_recepcion );
    MPI_Isend( &valor_enviado, 1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD,
&ticket_envio );

    MPI_Wait( &ticket_envio, &estado );
    MPI_Wait( &ticket_recepcion, &estado );

    cout<< "Soy el proceso " << id_propio << " y he recibido el valor " <<
valor_recibido << " del proceso " << id_vecino << endl ;
}
else if ( id_propio == 0 ) // si n.procs. impar, el primero da error
    cerr << "Se esperaba un número par de procesos, pero hay "
        << num_procesos_actual << endl ;

MPI_Finalize();
return 0;
}
```

Quéééédate
y descubre
tu resumen
del año.

MÁS INFORMACIÓN



5. PRÁCTICA3: IMPLEMENTACIÓN DE ALGORITMOS DISTRIBUIDOS CON MPI

En esta última práctica obligatoria del curso de SCD, se propone dos ejercicios que serán los que se preguntarán en el examen, utilizando como es natural lo visto en el seminario anterior.

5.1 EL PROBLEMA DE LOS PRODUCTORES Y CONSUMIDORES

Hemos tratado este ejercicio en todas las prácticas. En primer lugar lo vimos con semáforos, después con monitores, y ahora veremos cómo implementarlo utilizando el estándar MPI con varios procesos. Para encontrar una solución al problema que sea lo suficientemente óptima como para decir que es viable, construiremos el programa poco a poco e iremos mejorándolo, explicando la motivación de cada cambio.

Reservados todos los derechos. Queda permitida la impresión en su totalidad.
No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

5.1.1 APROXIMACIÓN INICIAL

Definimos los siguientes roles entre los procesos que el programa tendrá que lanzar para su correcto funcionamiento: en primer lugar está claro que tendremos un total de n procesos de los cuales habrá un conjunto de procesos que serán los *Productores*, que se encargarán de producir los datos, y *Consumidores*, que serán los que se encargarán de consumirlos. Además, existirá un único proceso con un rol especial que será el *Buffer*, donde el productor se encargará de enviar los datos ahí, y donde recibirá peticiones de consumición por parte de los procesos consumidores, a las cuales el buffer tiene que responder.

Por tanto, para lanzar el programa correctamente será necesario al menos 3 procesos, dedicando siempre una etiqueta o identificación al proceso buffer sólo para un único proceso.

La solución a este ejercicio con la aproximación inicial es la que se encuentra en el fichero `prodcons.cpp`:

```
#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> // includes de MPI

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

// -----
// constantes que determinan la asignación de identificadores a roles:
const int
    id_productor      = 0 , // identificador del proceso productor
    id_buffer         = 1 , // identificador del proceso buffer
    id_consumidor     = 2 , // identificador del proceso consumidor
    num_procesos_esperado = 3 , // número total de procesos esperado
    num_items          = 20 ; // numero de items producidos o consumidos

//*****//
// plantilla de función para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
//-----

template< int min, int max > int aleatorio()
{
    static default_random_engine generador( random_device()() );
    static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
    return distribucion_uniforme( generador );
}
// -----
// produce los numeros en secuencia (1,2,3,...)

int producir()
{
    static int contador = 0 ;
    sleep_for( milliseconds( aleatorio<10,200>() ) );
    contador++ ;
    cout << "Productor ha producido valor " << contador << endl << flush;
    return contador ;
}
```

```

}

// ----

void funcion_productor()
{
    for ( unsigned int i= 0 ; i < num_items ; i++ )
    {
        // producir valor
        int valor_prod = producir();
        // enviar valor
        cout << "Productor va a enviar valor " << valor_prod << endl << flush;
        MPI_Ssend( &valor_prod, 1, MPI_INT, id_buffer, 0, MPI_COMM_WORLD );
    }
}
// ----

void consumir( int valor_cons )
{
    // espera bloqueada
    sleep_for( milliseconds( aleatorio<10,200>() ) );
    cout << "Consumidor ha consumido valor " << valor_cons << endl << flush ;
}
// ----

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

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

void funcion_buffer()
{
    int      valor ,
            peticion ;
    MPI_Status  estado ;

    for ( unsigned int i = 0 ; i < num_items ; i++ )
    {
        MPI_Recv( &valor, 1, MPI_INT, id_productor, 0, MPI_COMM_WORLD, &estado );
        cout << "Buffer ha recibido valor " << valor << endl ;

        MPI_Recv ( &peticion, 1, MPI_INT, id_consumidor, 0, MPI_COMM_WORLD, &estado );

        cout << "Buffer va a enviar valor " << valor << endl ;
        MPI_Ssend( &valor, 1, MPI_INT, id_consumidor, 0, MPI_COMM_WORLD );
    }
}

```

```

// ----

int main( int argc, char *argv[] )
{
    int id_propio, num_procesos_actual; // ident. propio, númer. 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 )
    {
        if ( id_propio == id_productor ) // si mi ident. es el del productor
            funcion_productor();           // ejecutar función del productor
        else if ( id_propio == id_buffer ) // si mi ident. es el del buffer
            funcion_buffer();             // ejecutar función buffer
        else                            // en otro caso, mi ident es consumidor
            funcion_consumidor();         // ejecutar función consumidor
    }
    else if ( id_propio == 0 ) // si hay error, el proceso 0 informa
        cerr << "error: número de procesos distinto del esperado." << endl ;

    MPI_Finalize( );
    return 0;
}
// -----

```

Como podemos observar no es una implementación muy difícil de realizar si se ha seguido de forma atenta las instrucciones del seminario. Sin embargo, no podemos dar como eficiente esta solución por diferentes motivos, siendo uno de los más fundamentales la sincronización. Como el buffer está siendo utilizado por el proceso productor y consumidor, le llegarán peticiones de ambos y como es lógico, no podrá atenderlos simultáneamente. El proceso productor, cuando envía un dato al buffer, no puede continuar produciendo más datos hasta que el proceso buffer reciba este mensaje, es decir, quedará esperando bloqueado. Por otra parte el proceso consumidor tiene que mandar peticiones al buffer para recibir un dato para consumir, por lo que también se quedará esperando bloqueado hasta que dicho dato que ha pedido, el buffer se lo envíe, por lo que no puede atender ningún mensaje más por parte del productor ni del consumidor, hasta que el consumidor no reciba el dato.

En conclusión, se pueden producir tiempos de espera bastante importantes desde el punto de vista de ambos roles, por lo que hay que intentar reducir eso obtando por otra estrategia.

5.1.2 SOLUCIÓN CON SELECCIÓN NO DETERMINISTA

En el planteamiento anterior el buffer sólo podía recibir un único dato por parte del proceso productor. Ahora, la idea será considerar un vector en el que el proceso productor producirá diversos valores y los irá enviando al proceso buffer, sin esperar al consumidor para que vacíe el hueco; y a su vez, el proceso consumidor podrá ir consumiendo los datos sin esperar a que el productor tenga que producir más.

Esta nueva aproximación, es mucho más parecida a la que hemos realizado tanto en semáforos como en monitores, y para llevarla a cabo se implementará la *espera selectiva*.



¿Quién te conoce mejor?

Tu madre

Tus amigos

Tu Wrapped

#SPOTIFYWRAPPED

Si dudas, echa un ojo a tu Wrapped

¿Tu madre,
tus amigos
o tu
Wrapped?

Descubre
quién te conoce
mejor en tu
resumen del año.



MÁS INFORMACIÓN



A partir de aquí es muy importante destacar el uso de las identificaciones del emisor para los mensajes. Si el vector está totalmente vacío, entonces los únicos mensajes que se esperan son los del proceso productor; si el vector está totalmente lleno, entonces los únicos mensajes que se esperan son los del proceso consumidor, y en otro caso, el proceso buffer esperará mensajes de cualquier emisor.

Una forma óptima de hacer esto, es considerar una única variable para el campo del emisor en la llamada a `MPI_Recv()` por parte del proceso buffer, a la que llamaremos `id_emisor_aceptable`. Este id tendrá un valor en función al criterio establecido anteriormente (dependiendo del estado en el que se encuentre el buffer).

Aquí se muestra la solución, que es la correspondiente al `prodcons2.cpp`:

```
#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;

//=====
// VARIABLES CONSTANTES
//=====

const int id_productor = 0,
          id_buffer = 1,
          id_consumidor = 2,
          num_procesos Esperado = 3,
          num_items = 20,
          tam_vector = 10;

//=====
// FUNCIONES COMUNES
//=====

/**
 * @brief Genera un entero aleatorio uniformemente distribuido entre dos
 * valores enteros, ambos incluidos
 * @param<T,U> : Números enteros min y max, respectivamente
 * @return un numero aleatorio generado entre min,max
 */
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);
}

/**
 * @brief Produce un valor (Ademas de incluir retardo de ejecucion)
 * @return valor producido
 */
int producir(){
    //CONTADOR DE VALORES PRODUCIDOS
    static int contador = 0;
```

```

//REALIZACION DE ESPERA BLOQUEADA
sleep_for(milliseconds(aleatorio<10,100>()));
contador++;
//INFORMA DE PRODUCCION DE VALOR
cout << "Productor ha producido valor " << contador << endl << flush;

return contador;
}

// ----

/***
 * @brief Consume el dato pasado como parametro
 * @param valor_cons : Valor a consumir
 */
void consumir(int valor_cons){
//REALIZACION DE ESPERA BLOQUEADA
sleep_for(milliseconds(aleatorio<110,200>()));

//MOSTRAR CONSUMICION
cout << "Consumidor ha consumido valor " << valor_cons << endl << flush;
}

//=====
// FUNCIONES DE PROCESOS
//=====

/***
 * @brief Funcion que ejecuta el proceso productor. Envia datos
 * producidos por él al buffer
 */
void funcion_productor(){
for(unsigned int i=0; i<num_items;i++){
//PRODUCCION DEL VALOR
int valor_prod = producir();

//ENVIO DEL VALOR AL BUFFER
cout << "Productor va a enviar valor " << valor_prod << endl << flush;

//MPI SSEND --> ENVIO SINCRONO
//-> void*      : valor_prod      = UBICACION DE BYTES EN MEMORIA
[ENVIO]
//-> int         : 1             = NUM DE ELEMENTOS ALMACENADOS
//-> MPI_Datatype : MPI_INT       = TIPO DE DATO ENVIADO
//-> int         : id_buffer     = IDENTIFICADOR DE PROCESO DESTINO
[BUFFER]
//-> int [tag]    : 0             = ETIQUETA DEL MENSAJE
//-> MPI_Comm     : MPI_COMM_WORLD = COMUNICADOR EN EL QUE SE REALIZA EL
ENVIO
MPI_Ssend(&valor_prod, 1, MPI_INT, id_buffer, 0, MPI_COMM_WORLD);
}

}

// ----

/***
 * @brief Funcion del proceso consmidor. Recibe los datos
 * enviados desde el proceso buffer y los consume

```

```

/*
void funcion_consumidor(){
    int peticion,          //BUFFER DE ENVIO DE DATOS [INT EXPECTED]
    valor_rec = 1;         //BUFFER DE RECEPCION DE DATOS [INT EXPECTED]

    MPI_Status estado;    //METADATO PARA OBTENER REMITENTE DEL MENSAJE

    for(unsigned int i=0; i<num_items; i++){
        //MPI SSEND --> ENVIO SINCRONO
        //--> void*      : peticion      = UBICACION DE BYTES EN MEMORIA
[ENVIO]
        //--> int         : 1           = NUM DE ELEMENTOS ALMACENADOS
        //--> MPI_Datatype : MPI_INT     = TIPO DE DATO ENVIADO
        //--> int         : id_buffer   = IDENTIFICADOR DE PROCESO DESTINO
[BUFFER]
        //--> int [tag]    : 0           = ETIQUETA DEL MENSAJE
        //--> MPI_Comm     : MPI_COMM_WORLD = COMUNICADOR EN EL QUE SE REALIZA EL
ENVIO
        MPI_Ssend(&peticion, 1, MPI_INT, id_buffer, 0, MPI_COMM_WORLD);

        //MPI RECV --> RECIBIMIENTO SINCRONO
        //--> void*      : valor_rec    = UBICACION DE MEMORIA DONDE SE
RECIBEN LOS DATOS
        //--> int         : 1           = NUM DE ELEMENTOS RECIBIDOS
        //--> MPI_Datatype : MPI_INT     = TIPO DE DATO ENVIADO
        //--> int         : id_buffer   = IDENTIFICADOR DE PROCESO RECEPTOR
        //--> int [tag]    : 0           = ETIQUETA DEL MENSAJE
        //--> MPI_Comm     : MPI_COMM_WORLD = COMUNICADOR EN EL QUE SE REALIZA LA
RECEPCION
        //--> MPI_Status   : estado       = METADATO PARA OBTENER INFORMACION
DEL EMISOR
        MPI_Recv (&valor_rec, 1, MPI_INT, id_buffer, 0, MPI_COMM_WORLD, &estado);

        //INFORME DE QUE SE HA RECIBIDO EL VALOR A CONSUMIR
        cout << "Consumidor ha recibido valor " << valor_rec << endl << flush;

        //CONSUMIR EL DATO RECIBIDO
        consumir(valor_rec);
    }
}

// -----
/***
 * @brief Funcion del proceso buffer. Recibe los datos producidos
 * por el proceso productor y envia los datos que almacena en el vector
 * al consumidor
 */
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
        id_emisor_aceptable; // identificador de emisor aceptable

    MPI_Status estado;          // metadatos del mensaje recibido
}

```

```

//BUCLE DE GESTION DE PETICIONES EN EL BUFFER
//-> TOTAL DE ITERACIONES: 2* NUM_ITEMS [ATIENDE A PETICIONES CONSUMIDOR +
PRODUCTOR]
for(unsigned int i=0; i<num_items*2; i++){
    //PASO 1 : DETERMINAR SI PUEDE ENVIAR [PROD], [CONS], [ANY_SOURCE]
    if (num_celdas_ocupadas == 0)           // si buffer vacío
        id_emisor_aceptable = id_productor; // $~~$ solo prod.

    else if (num_celdas_ocupadas == tam_vector) // si buffer lleno
        id_emisor_aceptable = id_consumidor; // $~~$ solo cons.

    else                                     // si no vacío ni lleno
        id_emisor_aceptable = MPI_ANY_SOURCE; // $~~$ cualquiera

    //PASO 2 : RECIBIR EL MENSAJE DEL EMISOR O EMISORES ACEPTABLES

    //MPI_RECV --> RECIBIMIENTO SINCRONO
    //-> void*      : valor                  = UBICACION DE MEMORIA DONDE
SE RECIBEN LOS DATOS
    //-> int         : 1                     = NUM DE ELEMENTOS RECIBIDOS
    //-> MPI_Datatype : MPI_INT              = TIPO DE DATO ENVIADO
    //-> int         : id_emisor_aceptable   = IDENTIFICADOR DE PROCESO
RECEPTOR
    //-> int [tag]   : 0                     = ETIQUETA DEL MENSAJE
    //-> MPI_Comm    : MPI_COMM_WORLD        = COMUNICADOR EN EL QUE SE
REALIZA LA RECEPCION
    //-> MPI_Status  : estado               = METADATO PARA OBTENER
INFORMACION DEL EMISOR
    MPI_Recv(&valor, 1, MPI_INT, id_emisor_aceptable, 0, MPI_COMM_WORLD,
&estado);

    //PASO 3 : PROCESAR EL MENSAJE RECIBIDO

    //COMPROBAMOS LOS METADATOS DEL EMISOR
    switch(estado.MPI_SOURCE) {
        //CASE ID_PRODUCTOR : ES EL PRODUCTOR QUIEN MANDO LA PETICION
        case id_productor:
            //INSERCIÓN EN VECTOR
            buffer[primera_libre] = valor;
            //MODIFICACION DE APUNTADOR
            primera_libre = (primera_libre+1) % tam_vector;
            //MODIFICACION DE CONTADOR DE OCUPADAS
            num_celdas_ocupadas++;
            //MOSTRAR MENSAJE DE RECEPCION
            cout << "Buffer ha recibido valor " << valor << endl;
            break;

        //CASE ID_CONSUMIDOR : ES EL CONSUMIDOR QUIEN REALIZO LA PETICION
        case id_consumidor: // si ha sido el consumidor: extraer y enviarle
            //EXTRACCION DEL DATO DEL VECTOR
            valor = buffer[primera_ocupada];
            //MODIFICACION DEL APUNTADOR
            primera_ocupada = (primera_ocupada+1) % tam_vector;
            //MODIFICACION DEL CONTADOR DE OCUPADAS
            num_celdas_ocupadas--;
            //MOSTRAR MENSAJE DE ENVIO
    }
}

```



Ábrete la Cuenta Online de BBVA y llévate 1 año de Wuolah PRO

cómo??



1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

ventajas

PRO



Dí adiós a la publicidad en los apuntes y en la web



Participa gratis en todos los sorteos



Descarga carpetas completas

estudia sin publicidad

WUOLAH PRO

```
cout << "Buffer va a enviar valor " << valor << endl;

//MPI SSEND --> ENVIO SINCRONO
//-> void*      : valor_prod      = UBICACION DE BYTES EN
MEMORIA [ENVIO]
    //-> int         : 1             = NUM DE ELEMENTOS ALMACENADOS
    //-> MPI_Datatype : MPI_INT       = TIPO DE DATO ENVIADO
    //-> int         : id_buffer     = IDENTIFICADOR DE PROCESO

DESTINO [BUFFER]
    //-> int [tag]   : 0             = ETIQUETA DEL MENSAJE
    //-> MPI_Comm    : MPI_COMM_WORLD = COMUNICADOR EN EL QUE SE

REALIZA EL ENVIO
    MPI_Ssend(&valor, 1, MPI_INT, id_consumidor, 0, MPI_COMM_WORLD);
    break;
}
}

//=====
// FUNCION PRINCIPAL
=====

/***
 * @brief Funcion principal
 * @param argc : Numero de argumentos
 * @param argv : Total de argumentos
 * @return 0 [EX COMPLETADA]
 */
int main(int argc, char *argv[]){
    //ID PROPIO : IDENTIFICADOR DE PROCESO
    //NUM_PROCESOS_ACTUAL : TOTAL DE PROCESOS EJECUTADOS
    int id_propio, num_procesos_actual;

    //PASO 1: INICIALIZAR MPI, LEER ID_PROPPIO Y NUM_PROCESOS_ACTUAL

    //MPI INIT --> INICIO DEL PROCEDIMIENTO MPI
    //-> int*        : argc = NUMERO DE ARGUMENTOS
    //-> char***     : argv = TOTAL DE ARGUMENTOS
    MPI_Init(&argc, &argv);

    //MPI_COMM_RANK -->
    //-> MPI_Comm    : MPI_COMM_WORLD = COMUNICADOR MPI
    //-> int*        : id_propio     = ALMACENA EL IDENTIFICADOR DEL PROCESO

[AUTO-INVOKE]
    MPI_Comm_rank(MPI_COMM_WORLD, &id_propio);

    //MPI_COMM_SIZE -->
    //-> MPI_Comm    : MPI_COMM_WORLD      = COMUNICADOR MPI
    //-> int*        : num_procesos_actual = ALMACENA EL TOTAL DE PROCESOS ACTUALES
    MPI_Comm_size(MPI_COMM_WORLD, &num_procesos_actual);

    //COMPROBACION DE QUE EL LANZAMIENTO DE PROCESOS ES EL ESPERADO
    if (num_procesos Esperado == num_procesos_actual){
        //ASIGNACION DE ROL: PRODUCTOR
        if (id_propio == id_productor)
            funcion_productor();
        //ASIGNACION DE ROL : BUFFER
        else if (id_propio == id_buffer)
```

```

        funcion_buffer();
//ASIGNACION DE ROL : CONSUMIDOR
else
    funcion_consumidor();
}
//NO SE CUMPLE LOS PARAMETROS ESPERADOS
else{
    //ERROR LANZADO DESDE EL PROCESO 0
    if (id_propio == 0){
        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;
    }
}

//MPI_FINALIZE --> Finaliza el procedimiento MPI
MPI_Finalize();

return 0;
}

```

5.1.3 VERSIÓN CON VARIOS PROCESOS PRODUCTORES Y CONSUMIDORES

La solución final al problema de los productores y consumidores es la siguiente, en la que se tiene como política del buffer FIFO (que ya hemos hablado de ella en otras secciones), y además, habrá varios productores y varios consumidores.

Hay que tener en cuenta que los items que se van a producir y que se van a consumir tienen que ser repartidos equitativamente por todos los procesos, y también, hay que definir adecuadamente los id asignados a cada proceso, a los productores, al buffer, y a los consumidores. Lo mejor para hacer esto último es considerar los siguientes rangos:

- IDs procesos productores: desde 0 hasta $numProductores - 1$
- ID proceso buffer: $numProductores$
- IDs procesos consumidores: desde $numProductores + 1$ hasta $totalProcesos - 1$.

(Nótese que $totalProcesos = numProductores + numConsumidores$)

Se puede seguir este criterio desde el lado del buffer para identificar, como hemos hecho en el anterior ejemplo, qué proceso es el que ha enviado el mensaje, para así procesar la información adecuadamente en función de si se trata de un proceso productor o consumidor. Otra forma de hacerlo (incluso más elegante que ésta y además muy útil de cara a los exámenes) es trabajar con las etiquetas de los mensajes. Hasta ahora, siempre hemos colocado en ese campo el valor 0 por defecto. Sin embargo, podemos poner el valor que nosotros consideremos para identificar un mensaje concreto. En esta ocasión, diremos que la etiqueta *etiq_productor* (con valor numérico 1) la tendrán los mensajes que procedan de un proceso productor, y *etiq_consumidor* (con valor numérico 2) la tendrán los mensajes que procedan de un proceso consumidor. Consultando la estructura de metadatos `MPI_Status`, se puede recuperar la etiqueta del mensaje, siendo el razonamiento exactamente el mismo que el que se utilizaría si consideramos la id.

La solución que se propone es la siguiente, encontrada en el fichero `prodcons2-muFIFO.cpp`:

```

/**
 * @file prodcons2-muFIFO.cpp

```

```

* @author Daniel Perez Ruiz
*/
// -----
// 
// 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
// (versión con un multiples productores y multipoles consumidores)
//
// Historial:
// Actualizado a C++11 en Septiembre de 2017
// -----
#include <iostream>
#include <thread> // this_thread::sleep_for
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include <cassert>
#include <cstdlib>
#include <mpi.h>

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

//=====
// VARIABLES CONSTANTES
//=====

//DEFINICION DEL NUMERO DE PRODUCTORES Y NUMERO DE CONSUMIDORES
static const int numProductores = 4,
                numConsumidores = 5;

//DEFINICION DEL NUMERO DE PROCESOS QUE SE ESPERAN LANZAR
const int num_procesos Esperado = numProductores + numConsumidores + 1;

//ASIGNACION DE ROLES : BUFFER DE DATOS
const int id_buffer = numProductores;

//DEFINICION DE NUMERO DE ITEMS Y DIMENSION BUFFER
const int multiplo = 1;
const int num_items = multiplo * numProductores * numConsumidores, // [DEBE SER
MULTIPLO DE NUMP Y NUMC]
tam_vector = 10,
//TOTAL DE ITEMS QUE PRODUCE UN PRODUCTOR
itemsProductor = num_items / numProductores,
//TOTAL DE ITEMS QUE CONSUME UN CONSUMIDOR
itemsConsumidor = num_items / numConsumidores;

//DEFINICION DE ETIQUETAS PARA EL PASO DE MENSAJES
const int etiq_productor = 1,
        etiq_consumidor = 2;

```

```

//=====================================================================
// FUNCIONES COMUNES
//=====================================================================

/**
 * @brief Comprueba que los parametros correctos
 * @return True si todo se cumple | False en otro caso
 */
int checkParameters(){
    //1: COMPROBAR NUM_ITEMS Y TAM_VECTOR POSITIVOS
    if((num_items < 1) || (tam_vector < 1)){
        return 1;
    }

    //2: COMPROBAR QUE NUM_ITEMS SEA MULTIPLO DE NUMP Y NUMC
    if((num_items % numProductores != 0) || (num_items % numConsumidores != 0)){
        return 2;
    }

    //3: COMPROBAR QUE ROL BUFFER SEA NUMP
    if(id_buffer != numProductores){
        return 3;
    }

    //4: COMPROBAR QUE NUM_PROCESOS_ESPERADO SEA IGUAL A LA SUMA DE PROD Y CONS
    if(num_procesos Esperado != (numProductores + numConsumidores +1)){
        return 4;
    }

    //5: COMPROBAR QUE LAS ETIQUETAS DEL PASO A MENSAJES SEAN NO NEGATIVAS
    if((etiq_productor < 0) || (etiq_consumidor < 0)){
        return 5;
    }

    return 0;
}

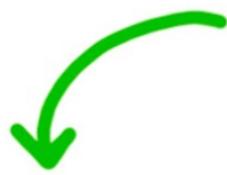
/**
 * @brief Muestra al usuario el tipo de error que se ha producido tras el
lanzamiento
 * del programa (error en variables constantes)
 */
void notifyError(int errorCode){
    switch(errorCode){
        case 1:
            cout << "ERROR: NUM_ITEMS o TAM_VECTOR ES NULO " << endl;
            break;

        case 2:
            cout << "ERROR: NUM_ITEMS NO ES MULTIPLO DE NUM_PROD o NUM_CONS" <<
endl;
            break;

        case 3:
            cout << "ERROR: EL IDENTIFICADO DEL BUFFER [" << id_buffer << "]"
DEBERIA SER [" << numProductores << "] " << endl;
            break;
    }
}

```

Encuentra el trabajo de tus sueños



Escanéame y obtén más info!!



```
case 4:  
    cout << "ERROR: EL NUMERO DE PROCESOS QUE SE ESPERA [" <<  
num_procesos Esperado << "] NO ES CORRECTO." << endl;  
    break;  
  
case 5:  
    cout << "ERROR: LA ETIQUETA DE PRODUCTOR O DE CONSUMIDOR ES NEGATIVA"  
<< endl;  
    break;  
  
}  
}  

```



```

//=====
// FUNCIONES DE PROCESOS
//=====

/***
 * @brief Funcion que ejecuta el proceso productor. Envia datos
 * producidos por él al buffer
 */
void funcion_productor(int numProductor){
    unsigned int i;
    for(i=0; i<itemsProductor;i++){
        //PRODUCCION DEL VALOR
        int valor_prod = producir(numProductor);

        //ENVIO DEL VALOR AL BUFFER
        cout << "">>>> [P- " << numProductor << "] Productor va a enviar valor " <<
        valor_prod << endl << flush;

        //MPI SSEND --> ENVIO SINCRONO
        //--> void*      : valor_prod      = UBICACION DE BYTES EN MEMORIA
[ENVIO]
        //--> int         : 1              = NUM DE ELEMENTOS ALMACENADOS
        //--> MPI_Datatype : MPI_INT       = TIPO DE DATO ENVIADO
        //--> int         : id_buffer     = IDENTIFICADOR DE PROCESO DESTINO
[BUFFER]
        //--> int [tag]    : etiq_productor = ETIQUETA DEL MENSAJE
        //--> MPI_Comm     : MPI_COMM_WORLD = COMUNICADOR EN EL QUE SE REALIZA EL
ENVIO
        MPI_Ssend(&valor_prod, 1, MPI_INT, id_buffer, etiq_productor,
MPI_COMM_WORLD);
    }
}

// -----
/***
 * @brief Funcion del proceso consumidor. Recibe los datos
 * enviados desde el proceso buffer y los consume
 */
void funcion_consumidor(int numConsumidor){
    int peticion,           //BUFFER DE ENVIO DE DATOS [INT EXPECTED]
        valor_rec = 1; //BUFFER DE RECEPCION DE DATOS [INT EXPECTED]

    MPI_Status estado; //METADATO PARA OBTENER REMITENTE DEL MENSAJE

    unsigned int i;
    for(i=0; i<itemsConsumidor; i++){
        //MPI SSEND --> ENVIO SINCRONO
        //--> void*      : peticion      = UBICACION DE BYTES EN MEMORIA
[ENVIO]
        //--> int         : 1              = NUM DE ELEMENTOS ALMACENADOS
        //--> MPI_Datatype : MPI_INT       = TIPO DE DATO ENVIADO
        //--> int         : id_buffer     = IDENTIFICADOR DE PROCESO DESTINO
[BUFFER]
        //--> int [tag]    : etiq_consumidor = ETIQUETA DEL MENSAJE
        //--> MPI_Comm     : MPI_COMM_WORLD = COMUNICADOR EN EL QUE SE REALIZA EL
ENVIO
}

```

```

        MPI_Ssend(&peticion, 1, MPI_INT, id_buffer, etiq_consumidor,
MPI_COMM_WORLD);

        //MPI_RECV --> RECIBIMIENTO SINCRONO
        //--> void*          : valor_rec      = UBICACION DE MEMORIA DONDE SE
RECIBEN LOS DATOS
        //--> int             : 1            = NUM DE ELEMENTOS RECIBIDOS
        //--> MPI_Datatype    : MPI_INT       = TIPO DE DATO ENVIADO
        //--> int             : id_buffer     = IDENTIFICADOR DE PROCESO RECEPTOR
        //--> int [tag]        : etiq_productor = ETIQUETA DEL MENSAJE
        //--> MPI_Comm         : MPI_COMM_WORLD = COMUNICADOR EN EL QUE SE REALIZA LA
RECEPCION
        //--> MPI_Status       : estado        = METADATO PARA OBTENER INFORMACION
DEL EMISOR
        MPI_Recv (&valor_rec, 1, MPI_INT, id_buffer, etiq_consumidor,
MPI_COMM_WORLD, &estado);

        //INFORME DE QUE SE HA RECIBIDO EL VALOR A CONSUMIR
cout << "<<< [C-" << numConsumidor << "] Consumidor ha recibido valor "
<< valor_rec << endl << flush;

        //CONSUMIR EL DATO RECIBIDO
consumir(numConsumidor, valor_rec);
}

}

// -----
/***
 * @brief Funcion del proceso buffer. Recibe los datos producidos
 * por el proceso productor y envia los datos que almacena en el vector
 * al consumidor
 */
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
        id_emisor_aceptable;        // identificador de emisor aceptable

    MPI_Status estado;               // metadatos del mensaje recibido
    int tag = MPI_ANY_TAG;

    //BUCLE DE GESTION DE PETICIONES EN EL BUFFER
    //--> TOTAL DE ITERACIONES: 2* NUM_ITEMS [ATIENDE A PETICIONES CONSUMIDOR +
PRODUCTOR]
    for(unsigned int i=0; i<num_items*2; i++){
        //PASO 1 : DETERMINAR SI PUEDE ENVIAR [PROD], [CONS], [ANY_SOURCE]
        if (num_celdas_ocupadas == 0){           // si buffer vacío
            tag = etiq_productor;                // $~~$ solo prod.
        }

        else if (num_celdas_ocupadas == tam_vector){ // si buffer lleno
            tag = etiq_consumidor;                // $~~$ solo cons.
        }

        else{                                     // si no vacío ni lleno

```

```

//PASO INTERMEDIO: BLOQUEAR HASTA QUE HAYA ALGUN MENSAJE ESPERANDO
tag = MPI_ANY_TAG;

//MPI PROBE --> ESPERA MENSAJE
//-> int          : MPI_ANY_SOURCE           = IDENTIFICADOR DE
EMISOR
//-> int [tag]    : tag                      = ETIQUETA DEL MENSAJE
//-> MPI_Comm     : MPI_COMM_WORLD           = COMUNICADOR EN EL QUE
SE REALIZA LA RECEPCION
//-> MPI_Status   : estado                  = METADATO PARA OBTENER
INFORMACION DEL EMISOR
MPI_Probe(MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &estado);

//OBTENER PROCEDENCIA DE LA PETICION DEL MENSAJE
tag = estado.MPI_TAG;
}

//PASO 2 : RECIBIR EL MENSAJE DEL EMISOR O EMISORES ACEPTABLES

//MPI RECV --> RECIBIMIENTO SINCRONO
//-> void*         : valor                 = UBICACION DE MEMORIA DONDE
SE RECIBEN LOS DATOS
//-> int           : 1                     = NUM DE ELEMENTOS RECIBIDOS
//-> MPI_Datatype : MPI_INT                = TIPO DE DATO ENVIADO
//-> int           : MPI_ANY_SOURCE        = IDENTIFICADOR DE PROCESO
RECEPTOR
//-> int [tag]    : tag                   = ETIQUETA DEL MENSAJE
//-> MPI_Comm     : MPI_COMM_WORLD        = COMUNICADOR EN EL QUE SE
REALIZA LA RECEPCION
//-> MPI_Status   : estado              = METADATO PARA OBTENER
INFORMACION DEL EMISOR
MPI_Recv(&valor, 1, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD,
&estado);

//PASO 3 : PROCESAR EL MENSAJE RECIBIDO

//SI EL RECIBO DE LA PETICION FUE DESDE EL PRODUCTOR
if(estado.MPI_TAG == etiq_productor){
    //INSECCION EN VECTOR
    buffer[primera_libre] = valor;
    //MODIFICACION DE APUNTADOR
    primera_libre = (primera_libre+1) % tam_vector;
    //MODIFICACION DE CONTADOR DE OCUPADAS
    num_celdas_ocupadas++;
    //MOSTRAR MENSAJE DE RECEPCION
    cout << "-----> Buffer ha recibido valor " << valor << endl;
}
//SI EL RECIBO DE LA PETICION FUE DEL CONSUMIDOR
else{
    //EXTRACCION DEL DATO DEL VECTOR
    valor = buffer[primera_ocupada];
    //MODIFICACION DEL APUNTADOR
    primera_ocupada = (primera_ocupada+1) % tam_vector;
    //MODIFICACION DEL CONTADOR DE OCUPADAS
    num_celdas_ocupadas--;
    //MOSTRAR MENSAJE DE ENVIO
    cout << "<----- Buffer va a enviar valor " << valor << endl;
}

```

Encuentra el trabajo de tus sueños

Participa en retos y competiciones de programación



Escanéame y
obtén más info!!

```
//MPI SSEND --> ENVIO SINCRONO
//-> void*      : valor_prod      = UBICACION DE BYTES EN MEMORIA
[ENVIO]
    //-> int       : 1             = NUM DE ELEMENTOS ALMACENADOS
    //-> MPI_Datatype : MPI_INT     = TIPO DE DATO ENVIADO
    //-> int       : id_buffer     = IDENTIFICADOR DE PROCESO DESTINO
[BUFFER]
    //-> int [tag]   : etiq_buffer   = ETIQUETA DEL MENSAJE
    //-> MPI_Comm   : MPI_COMM_WORLD = COMUNICADOR EN EL QUE SE REALIZA
EL ENVIO
    MPI_Ssend(&valor, 1, MPI_INT, estado.MPI_SOURCE, etiq_consumidor,
MPI_COMM_WORLD);
}
}

//=====
// FUNCION PRINCIPAL
//=====

/**
 * @brief Funcion principal
 * @param argc : Numero de argumentos
 * @param argv : Total de argumentos
 * @return 0 [EX COMPLETADA]
 */
int main(int argc, char *argv[]){
    //ID PROPIO : IDENTIFICADOR DE PROCESO
    //NUM_PROCESOS_ACTUAL : TOTAL DE PROCESOS EJECUTADOS
    int id_propio, num_procesos_actual;

    //PASO 1: INICIALIZAR MPI, LEER ID_PROPPIO Y NUM_PROCESOS_ACTUAL

    //MPI INIT --> INICIO DEL PROCEDIMIENTO MPI
    //-> int*       : argc = NUMERO DE ARGUMENTOS
    //-> char***     : argv = TOTAL DE ARGUMENTOS
    MPI_Init(&argc, &argv);

    //MPI_COMM_RANK --> OBTIENE LOS VALORES DE CADA PROCESO
    //-> MPI_Comm   : MPI_COMM_WORLD = COMUNICADOR MPI
    //-> int*       : id_propio     = ALMACENA EL IDENTIFICADOR DEL PROCESO
[AUTO-INVOKE]
    MPI_Comm_rank(MPI_COMM_WORLD, &id_propio);

    //MPI_COMM_SIZE --> OBTIENE EL TOTAL DE PROCESOS
    //-> MPI_Comm   : MPI_COMM_WORLD     = COMUNICADOR MPI
    //-> int*       : num_procesos_actual = ALMACENA EL TOTAL DE PROCESOS ACTUALES
    MPI_Comm_size(MPI_COMM_WORLD, &num_procesos_actual);

    //COMPROBACION DE QUE EL LANZAMIENTO DE PROCESOS ES EL ESPERADO
    if ((num_procesos Esperado == num_procesos_actual)){
        //COMPROBAMOS SI EL PROGRAMA ES CORRECTO
        int guarda = checkParameters();

        //SI LAS CONSTANTES SON CORRECTAS LANZAMOS APLICACION
        if(guarda == 0){
            //ASIGNACION DE ROL: PRODUCTOR
```



WUOLAH

Reservados todos los derechos. Queda permitida la impresión en su totalidad.
No se permite la explotación económica ni la transformación de esta obra.

```

    if (id_propio < id_buffer){
        funcion_productor(id_propio);
    }
    //ASIGNACION DE ROL : BUFFER
    else if (id_propio == id_buffer)
        funcion_buffer();
    //ASIGNACION DE ROL : CONSUMIDOR
    else
        funcion_consumidor(id_propio);
}
//LAS CONSTANTES NO TIENE PARAMETROS VALIDOS
else{
    if(id_propio == 0)
        notifyError(guarda);
}
}
//NO SE CUMPLE LOS PARAMETROS ESPERADOS
else{
    //ERROR LANZADO DESDE EL PROCESO 0
    if (id_propio == 0){
        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;
    }
}

//MPI_FINALIZE --> Finaliza el procedimiento MPI
MPI_Finalize();

return 0;
}

```

5.2 EL PROBLEMA DE LA CENA DE LOS FILÓSOFOS

Este es el último gran problema que se abordará en el curso. Aquí entra en juego todo lo visto con el estándar de paso de mensajes MPI. A continuación, se presenta la motivación del problema.

Nos encontramos en un restaurante en el que van a comer una serie de filósofos en una mesa redonda. Dicha mesa dispone de una serie de tenedores, una cantidad exactamente igual a la del número de filósofos, que se encuentran repartidos por toda la mesa, por lo que cada filósofo verá la posibilidad de coger exactamente dos tenedores, el de su izquierda y el de su derecha. Si un filósofo se dispone a comer, necesita retirar los dos tenedores que se han mencionado anteriormente, una vez finalice, se pondrá a pensar, dejándolos libres.

Es por tanto por lo que el programa que intentaremos simular tendrá los siguientes roles definidos: los procesos *filósofos* serán la simulación de los filósofos que se sientan a la mesa, comen, y por último piensan; mientras que también estarán los procesos *tenedores*, que serán los utensilios con los que los filósofos comerán. En esta ocasión se dispondrá de 5 filósofos y 5 tenedores, empezando la numeración dentro del programa en 0.

Hay una serie de restricciones que el programa debe cumplir, como bien especifica el documento de prácticas de la asignatura. Se incluyen aquí para complementar la información:

1. Para comer, el filósofo tiene que coger el tenedor de su izquierda y después el de su derecha.
2. Cuando lo primero se cumpla, el filósofo come.
3. Cuando termine, el filósofo soltará los tenedores (da igual cuál primero).
4. Para finalizar, el filósofo se pondrá a pensar (ponemos a dormir el proceso con duración aleatoria).

Además también se especifica que el envío de mensajes tiene que ser síncrono seguro, por lo que obligatoriamente esas peticiones tienen que realizarse utilizando `MPI_Ssend()`.

Desde el punto de vista de los procesos tenedores también hay una serie de restricciones que tienen que ser cumplimentadas:

1. Los tenedores tienen que esperar bloqueados hasta que un filósofo envié una petición para cogerlo.
2. Una vez el tenedor sea cogido por el filósofo que le corresponde, tendrá que esperar a que éste envíe un nuevo mensaje solicitando que lo va a liberar.

Ambos procesos ejecutarán todos sus envíos y recepciones de mensajes en un bucle infinito. También hay que considerar que los filósofos sólo pueden hacer estas solicitudes a los tenedores que vean en sus manos, es decir, no pueden coger otro tenedor que no sea el que se encuentra a su izquierda y a su derecha.

Una aproximación de solución a esta descripción es la que se encuentra en el fichero `filosofos_interbloq.cpp`:

```
/*
 * @file filosofos-interb.cpp
 * @author Daniel Perez Ruiz
 */

// -----
// 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.
//
// Historial:
// Actualizado a C++11 en Septiembre de 2017
// -----


#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;
```

```

//=====
// VARIABLES CONSTANTES
//=====

const int num_filosofos = 5,
          num_procesos = 2 * num_filosofos;

const int etiq_reservarTenedor = 1,
          etiq_liberarTenedor = 2;

//=====
// FUNCIONES COMUNES
//=====

/***
 * @brief Genera un entero aleatorio uniformemente distribuido entre dos
 * valores enteros, ambos incluidos
 * @param<T,U> : Números enteros min y max, respectivamente
 * @return un numero aleatorio generado entre min,max
 */
template< int min, int max > int aleatorio(){
    static default_random_engine generador((random_device())());
    static uniform_int_distribution<int> distribucion_uniforme(min, max);
    return distribucion_uniforme(generador);
}

// -----
//=====
// FUNCIONES DE PROCESOS
//=====

/***
 * @brief
 * @param id
 */
void funcion_filosofos(int id){
    int id_ten_izq = (id+1) % num_procesos, //id. tenedor izq.
        id_ten_der = (id+num_procesos-1) % num_procesos; //id. tenedor der.

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

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

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

```



¿Quién te conoce mejor?

Tu madre

Tus amigos

Tu Wrapped

Si dudas, echa un ojo a tu Wrapped



Quéééédate
y descubre
tu resumen
del año.

MÁS INFORMACIÓN



#SPOTIFYWRAPPED

```

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

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

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

// ----

/** 
 * @brief
 * @param id
 */
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)
        // ..... guardar en 'id_filosofo' el id. del emisor (completar)
        MPI_Recv(&valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_reservarTenedor,
MPI_COMM_WORLD, &estado);
        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_liberarTenedor,
MPI_COMM_WORLD, &estado);

        cout << "Ten. " << id << " ha sido liberado por filo. " << id_filosofo
<< endl;
    }
}
// =====
// FUNCION PRINCIPAL
// =====

/** 
 * @brief Funcion principal
 * @param argc : Numero de argumentos
 * @param argv : Total de argumentos
 * @return 0 [EX COMPLETADA]
 */
int main(int argc, char** argv){

```

```

//ID PROPIO : IDENTIFICADOR DE PROCESO
//NUM_PROCESOS_ACTUAL : TOTAL DE PROCESOS EJECUTADOS
int id_propio, num_procesos_actual;

//PASO 1: INICIALIZAR MPI, LEER ID_PROPPIO Y NUM_PROCESOS_ACTUAL

//MPI INIT --> INICIO DEL PROCEDIMIENTO MPI
//-> int*      : argc = NUMERO DE ARGUMENTOS
//-> char***    : argv = TOTAL DE ARGUMENTOS
MPI_Init(&argc, &argv);

//MPI_COMM_RANK --> OBTIENE LOS VALORES DE CADA PROCESO
//-> MPI_Comm   : MPI_COMM_WORLD = COMUNICADOR MPI
//-> int*       : id_propio     = ALMACENA EL IDENTIFICADOR DEL PROCESO
[AUTO-INVOC]
MPI_Comm_rank(MPI_COMM_WORLD, &id_propio);

//MPI_COMM_SIZE --> OBTIENE EL TOTAL DE PROCESOS
//-> MPI_Comm   : MPI_COMM_WORLD      = COMUNICADOR MPI
//-> int*       : num_procesos_actual = ALMACENA EL TOTAL DE PROCESOS ACTUALES
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{
    // solo el primero escribe error, indep. del rol
    if(id_propio == 0){
        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;
}

// -----

```

Esta no es una buena solución. Aunque efectivamente hace todo lo que se nos pedía, el programa a la larga puede padecer una situación de **interbloqueo**, concepto que ya hemos mencionado en otros capítulos de este documento. La explicación es muy sencilla: hemos dicho que el criterio que seguiremos será coger el tenedor de la izquierda y después el de la derecha, por parte de todos los filósofos. Puede suceder que se de el caso de que todos los filósofos consigan adquirir el tenedor de su izquierda, reservándolo para ellos solos. El problema pues, llega cuando intentan coger el de su derecha, que coincidirá con ser el tenedor de la izquierda del filósofo que tienen a su lado, por lo que cada filósofo se quedará esperando hasta que el tenedor que les falta se

quede libre para poder comer. Sin embargo, para que un filósofo pueda comer necesita exclusivamente dos tenedores, por lo que ningún filósofo podrá realizar esa acción y liberar el tenedor que ocupan, produciéndose esa situación de interbloqueo en la que todos se quedan esperando a algo que nunca ocurrirá.

Intentaremos resolver esta situación adoptando un criterio diferente en la siguiente solución.

Vamos ahora a intentar arreglar el problema del interbloqueo, de una forma muy sencilla. Puesto que esta situación se produce porque todos los filósofos empiezan a coger el mismo tenedor de su izquierda, podemos hacer que uno de los filósofos empiecen con el tenedor derecho en vez del izquierdo. Esto sería más que suficiente para que jamás se produzca más bloqueos y por tanto, el colapso del programa.

Sin embargo, una forma más elegante de hacerlo, podría ser especificar que los filósofos con identificación par tienen un criterio para coger los tenedores diferente a los filósofos con identificación impar. La solución es la que se encuentra en el fichero [filosofos.cpp](#):

```
/**  
 * @file filosofos.cpp  
 * @author Daniel Perez Ruiz  
 */  
  
// -----  
//  
// 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.  
//  
// Historial:  
// Actualizado a C++11 en Septiembre de 2017  
// -----  
  
  
#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;  
  
//=====  
// VARIABLES CONSTANTES  
//=====  
  
const int num_filosofos = 5,  
        num_procesos = 2 * num_filosofos;  
  
const int etiq_reservarTenedor = 1,
```

```

etiq_liberarTenedor = 2;

//=====
// FUNCIONES COMUNES
//=====

/***
 * @brief Genera un entero aleatorio uniformemente distribuido entre dos
 * valores enteros, ambos incluidos
 * @param<T,U> : Números enteros min y max, respectivamente
 * @return un numero aleatorio generado entre min,max
 */
template< int min, int max > int aleatorio(){
    static default_random_engine generador((random_device())());
    static uniform_int_distribution<int> distribucion_uniforme(min, max);
    return distribucion_uniforme(generador);
}

// -----
//=====

// FUNCIONES DE PROCESOS
//=====

/***
 * @brief
 * @param id
 */
void funcion_filosofos(int id){
    int id_ten_izq = (id+1) % num_procesos, //id. tenedor izq.
        id_ten_der = (id+num_procesos-1) % num_procesos; //id. tenedor der.

    while(true){
        //SI NUM DE FILOSOFO ES IMPAR ACCEDE PRIMERO AL TENEDOR DE SU IZQUIERDA
        if(((id % 4 == 0) && (id != 0)) || (id == 2)){
            cout << "Filósofo " << id << " solicita ten. izq: " << id_ten_izq <<
endl;
            // ... solicitar tenedor izquierdo (completar)
            MPI_Ssend(&id, 1, MPI_INT, id_ten_izq, etiq_reservarTenedor,
MPI_COMM_WORLD);

            cout << "Filósofo " << id << " solicita ten. der: " << id_ten_der <<
endl;
            // ... solicitar tenedor derecho (completar)
            MPI_Ssend(&id, 1, MPI_INT, id_ten_der, etiq_reservarTenedor,
MPI_COMM_WORLD);
        }
        //SI NUM DE FILOSOFO ES PAR ACCEDE PRIMERO AL TENEDOR DE SU DERECHA
        else{
            cout << "Filósofo " << id << " solicita ten. der: " << id_ten_der <<
endl;
            // ... solicitar tenedor derecho (completar)
            MPI_Ssend(&id, 1, MPI_INT, id_ten_der, etiq_reservarTenedor,
MPI_COMM_WORLD);

            cout << "Filósofo " << id << " solicita ten. izq: " << id_ten_izq <<
endl;

```



¿Quién te conoce mejor?

#SPOTIFYWRAPPED

 Tu madre Tus amigos Tu Wrapped**Si dudas, echa un ojo a tu Wrapped**

¿Tu madre,
tus amigos
o tu
Wrapped?

Descubre
quién te conoce
mejor en tu
resumen del año.



MÁS INFORMACIÓN



```
// ... solicitar tenedor izquierdo (completar)
MPI_Ssend(&id, 1, MPI_INT, id_ten_izq, etiq_reservarTenedor,
MPI_COMM_WORLD);
}

//LOS FILOSOFOS EMPIEZAN A COMER
cout << "Filósofo " << id << " comienza a comer" << endl;
sleep_for(milliseconds(aleatorio<10,100>()));

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

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

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

// -----
/** 
 * @brief
 * @param id
 */
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)
        // ..... guardar en 'id_filosofo' el id. del emisor (completar)
        MPI_Recv(&valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_reservarTenedor,
MPI_COMM_WORLD, &estado);
        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_liberarTenedor,
MPI_COMM_WORLD, &estado);

        cout << "Ten. " << id << " ha sido liberado por filo. " << id_filosofo
<< endl;
    }
}
// -----
//=====
// FUNCION PRINCIPAL
```

Reservados todos los derechos. Queda permitida la impresión en su totalidad.
No se permite la explotación económica ni la transformación de esta obra.

WUOLAH

```

//=====

/**
 * @brief Funcion principal
 * @param argc : Numero de argumentos
 * @param argv : Total de argumentos
 * @return 0 [EX COMPLETADA]
 */
int main(int argc, char** argv){
    //ID PROPIO : IDENTIFICADOR DE PROCESO
    //NUM_PROCESOS_ACTUAL : TOTAL DE PROCESOS EJECUTADOS
    int id_propio, num_procesos_actual;

    //PASO 1: INICIALIZAR MPI, LEER ID_PROPPIO Y NUM_PROCESOS_ACTUAL

    //MPI INIT --> INICIO DEL PROCEDIMIENTO MPI
    //-> int*      : argc = NUMERO DE ARGUMENTOS
    //-> char***    : argv = TOTAL DE ARGUMENTOS
    MPI_Init(&argc, &argv);

    //MPI_COMM_RANK --> OBTIENE LOS VALORES DE CADA PROCESO
    //-> MPI_Comm   : MPI_COMM_WORLD = COMUNICADOR MPI
    //-> int*       : id_propio      = ALMACENA EL IDENTIFICADOR DEL PROCESO
[AUTO-INVOCO]
    MPI_Comm_rank(MPI_COMM_WORLD, &id_propio);

    //MPI_COMM_SIZE --> OBTIENE EL TOTAL DE PROCESOS
    //-> MPI_Comm   : MPI_COMM_WORLD      = COMUNICADOR MPI
    //-> int*       : num_procesos_actual = ALMACENA EL TOTAL DE PROCESOS ACTUALES
    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{
        // solo el primero escribe error, indep. del rol
        if(id_propio == 0){
            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;
}

// -----

```

SOLUCIÓN CON PROCESO CAMARERO Y ESPERA SELECTIVA

Existe otra forma de solucionar el problema del interbloqueo utilizando un proceso más que se lanzará por parte del programa, al cual llamaremos *Camarero*, que se encargará de gestionar la situación del restaurante y concretamente, la mesa donde tienen que comer los filósofos. Su objetivo principal es controlar que no haya 5 filósofos sentados a la vez en la mesa, que es lo que haría que se produjera con altas probabilidades una situación de bloqueo.

Como es natural ahora el planteamiento del problema ha cambiado un poco. Los filósofos no empiezan sentados en la mesa. En primer lugar, los filósofos pedirán permiso al camarero para sentarse a la mesa, y una vez esta solicitud se vea cumplimentada, todo lo que se ha dicho hasta ahora seguirá sucediendo exactamente igual, es decir, el filósofo se sentará a la mesa y solicitará dos tenedores para disponerse a comer y después pensar. Ahora la diferencia reside en que, cuando terminen de pensar, se levantan de la mesa para ceder el asiento a otro filósofo. Si el filósofo no pudiera sentarse, tiene que esperar bloqueado hasta que el camarero le conceda el permiso.

Esta versión del programa es la más importante de todas, y la que seguramente se tenga en cuenta para las modificaciones que se pedirán a la hora de realizar el examen acerca de esta práctica. Hay que destacar otra vez la importancia de las etiquetas para identificar correctamente la motivación del mensaje que se envía a través de los procesos, para así saber cuándo nos estamos refiriendo a una petición de reserva/liberación de tenedor, o petición de levantarse/sentarse a la mesa, por parte del camarero. La función del camarero implementada en código es procesar los mensaje que les llega por parte de los filósofos, por lo que tiene que calcular el tipo de petición que tiene que atender, y si además puede atenderla.

En conclusión, la solución al problema con el proceso *camarero* y espera selectiva es la que se encuentra en el fichero `filosofos-camarero.cpp`:

```
/*
 * @file filosofos-camarero.cpp
 * @author Daniel Perez Ruiz
 */

// -----
// 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.
//
// Historial:
// Actualizado a C++11 en Septiembre de 2017
// -----


#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;

//=====
// VARIABLES CONSTANTES
//=====

const int num_filosofos = 5,
          num_procesos = 2 * num_filosofos +1;

const int id_camarero = num_procesos -1;

const int etiq_reservarTenedor = 1,
        etiq_liberarTenedor = 2,
        etiq_sentarseMesa = 3,
        etiq_levantarseMesa = 4;

const int sentadosPermitidos = 3;

//=====
// FUNCIONES COMUNES
//=====

/***
 * @brief Genera un entero aleatorio uniformemente distribuido entre dos
 * valores enteros, ambos incluidos
 * @param<T,U> : Números enteros min y max, respectivamente
 * @return un numero aleatorio generado entre min,max
 */
template< int min, int max > int aleatorio(){
    static default_random_engine generador((random_device())());
    static uniform_int_distribution<int> distribucion_uniforme(min, max);
    return distribucion_uniforme(generador);
}

// -----
// FUNCIONES DE PROCESOS
//=====

/***
 * @brief Funcion que simula un filosofo
 * @param id : id de proceso
 */
void funcion_filosofos(int id){
    int id_ten_izq = (id+1) % (num_procesos-1), //id. tenedor izq.
        id_ten_der = (id+num_procesos-2) % (num_procesos-1); //id. tenedor der.

    //ITERACIONES INFINITAS
    while(true){
        //LOS FILOSOFOS SOLICITAN SENTARSE EN LA MESA
        cout << ">>> Filósofo " << id << " solicita al camarero sentarse en la
mesa." << endl;
        MPI_Ssend(&id, 1, MPI_INT, id_camarero, etiq_sentarseMesa,
MPI_COMM_WORLD);
}

```



Ábrete la Cuenta Online de BBVA y
llévate 1 año de Wuolah PRO

cómo??



1/6

Este número es
indicativo del
riesgo del
producto, siendo
1/6 indicativo de
menor riesgo y
6/6 de mayor
riesgo.

BBVA está
adherido al
Fondo de
Garantía de
Depósitos de
Entidades de
Crédito de
España.
La cantidad
máxima
garantizada es
de 100.000 euros
por la totalidad
de los depósitos
constituidos
en BBVA por
persona.

ventajas

PRO



Dí adiós a la publi
en los apuntes y
en la web



Participa gratis
en todos los
sorteos



Descarga
carpetas
completas

estudia sin publi
WUOLAH PRO

```
//LOS FILOSOFOS SOLICITAN TENEDORES
cout << "">>>> Filósofo " << id << " solicita ten. izq: " << id_ten_izq <
endl;
// ... solicitar tenedor izquierdo
MPI_Ssend(&id, 1, MPI_INT, id_ten_izq, etiq_reservarTenedor,
MPI_COMM_WORLD);

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

//LOS FILOSOFOS EMPIEZAN A COMER
cout << ""><--> Filósofo " << id << " comienza a comer" << endl;
sleep_for(milliseconds(aleatorio<10,100>()));

//LIBERAR TENEDORES
cout << ""><<< Filósofo " << id << " suelta ten. izq: " << id_ten_izq
<< endl;
// ... soltar el tenedor izquierdo
MPI_Ssend(&id, 1, MPI_INT, id_ten_izq, etiq_liberarTenedor,
MPI_COMM_WORLD);

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

//LOS FILOSOFOS SE LEVANTAN DE LA MESA
cout << "">>>> Filósofo " << id << " avisa al camarero que se levanta de la
mesa." << endl;
MPI_Ssend(&id, 1, MPI_INT, id_camarero, etiq_levantarseMesa,
MPI_COMM_WORLD);

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

// -----
/***
 * @brief Función que simula un tenedor
 * @param id : id de proceso
 */
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)
        // ..... guardar en 'id_filosofo' el id. del emisor (completar)
        MPI_Recv(&valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_reservarTenedor,
MPI_COMM_WORLD, &estado);
        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_liberarTenedor,
MPI_COMM_WORLD, &estado);

        cout << "Ten. " << id << " ha sido liberado por filo. " << id_filosofo
<< endl;
    }

}

/***
 * @brief
 */
void funcion_camarero(){
    int valor, etiq_aceptable,
        sentados = 0;

    MPI_Status estado;

    while(true){
        //PASO 1: ATENDER PETICIONES DE SENTARSE A LA MESA

        //COMPROBAMOS SI SE PUEDEN SENTAR A LA MESA
        if(sentados == 0)
            etiq_aceptable = etiq_sentarseMesa;
        else if (sentados == sentadosPermitidos)
            etiq_aceptable = etiq_levantarseMesa;
        else
            etiq_aceptable = MPI_ANY_TAG;

        //RECIBIMOS EL MENSAJE

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

        //MOSTRAMOS POR PANTALLA LA PETICION INDICADA
        if(estado.MPI_TAG == etiq_sentarseMesa){
            sentados++;
            cout << "-----> Camarero acepta la peticion de sentarse a filosofo:
" << estado.MPI_SOURCE << endl;
        }
        if(estado.MPI_TAG == etiq_levantarseMesa){
            sentados--;
            cout << "<----- Camarero acepta la peticion de levantarse a
filosofo: " << estado.MPI_SOURCE << endl;
        }
    }

}

// -----
//=====
// FUNCION PRINCIPAL
//=====

/***

```

```

* @brief Funcion principal
* @param argc : Numero de argumentos
* @param argv : Total de argumentos
* @return 0 [EX COMPLETADA]
*/
int main(int argc, char** argv){
    //ID PROPIO : IDENTIFICADOR DE PROCESO
    //NUM_PROCESOS_ACTUAL : TOTAL DE PROCESOS EJECUTADOS
    int id_propio, num_procesos_actual;

    //PASO 1: INICIALIZAR MPI, LEER ID_PROPPIO Y NUM_PROCESOS_ACTUAL

    //MPI INIT --> INICIO DEL PROCEDIMIENTO MPI
    //-> int*      : argc = NUMERO DE ARGUMENTOS
    //-> char***    : argv = TOTAL DE ARGUMENTOS
    MPI_Init(&argc, &argv);

    //MPI_COMM_RANK --> OBTIENE LOS VALORES DE CADA PROCESO
    //-> MPI_Comm   : MPI_COMM_WORLD = COMUNICADOR MPI
    //-> int*       : id_propio      = ALMACENA EL IDENTIFICADOR DEL PROCESO
[AUTO-INVOK]
    MPI_Comm_rank(MPI_COMM_WORLD, &id_propio);

    //MPI_COMM_SIZE --> OBTIENE EL TOTAL DE PROCESOS
    //-> MPI_Comm   : MPI_COMM_WORLD      = COMUNICADOR MPI
    //-> int*       : num_procesos_actual = ALMACENA EL TOTAL DE PROCESOS ACTUALES
    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 == (id_camarero))
            funcion_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{
        // solo el primero escribe error, indep. del rol
        if(id_propio == 0){
            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;
}

// -----

```

6. PRÁCTICA4: PLANIFICACIÓN DE PROCESOS

En función del tiempo que se disponga por parte del profesor de prácticas, esta sección puede que no tenga examen de evaluación y se disponga a la entrega simplemente de los códigos que se pide implementar por parte del documento de profesor.

Aquí pondré únicamente las soluciones a los problemas.

6.1 PRIMERA PLANIFICACIÓN

```
//DANIEL PEREZ RUIZ

// -----
// Sistemas concurrentes y Distribuidos.
// Práctica 4. Implementación de sistemas de tiempo real.
//
// Archivo: ejecutivo1.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 |
// *-----*-----*-----*-----*
//
//
// Historial:
// Creado en Diciembre de 2017
// -----



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

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;
```

Encuentra el trabajo de tus sueños



Escanéame y obtén más info!!



```
// -----  
// tarea genérica: duerme durante un intervalo de tiempo (de determinada  
duración)  
  
/**  
 * @brief Tarea generica : duerme durante un intervalo de tiempo  
 * @param nombre : Nombre de tarea  
 * @param tcomputo : Intervalo de tiempo  
 */  
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  
    const milliseconds Ts(250);  
  
    // ini_sec = instante de inicio de la iteración actual del ciclo secundario  
    time_point<steady_clock> ini_sec = steady_clock::now();  
  
    //CICLO PRINCIPAL  
    while(true){  
        cout << endl  
            << "-----" << endl  
            << "Comienza iteración del ciclo principal." << endl;  
  
        //CICLO SECUNDARIO (4 ITERACIONES)  
        for(int i = 1; i <= 4; i++){  
            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;  
    }
```

Reservados todos los derechos.
No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

```

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

        //MODIFICACION 1: CALCULAMOS EL INSTANTE FINAL ACTUAL
        //--> TIPO DE DATOS USADO: TIME POINT
        //--> RELOJ UTILIZADO: STEADY CLOCK
        time_point<steady_clock> fin_sec = steady_clock::now();

        //MODIFICACION 2: CALCULO EL RETRASO DEL INSTANTE FINAL ACTUAL RESP.
ESPERADO
        //TIPO DE DATOS USADO: DURATION
        steady_clock::duration duracion = fin_sec - ini_sec;

        //SI NO HA HABIDO RETARDO. NOTIFICAR
        if(milliseconds_f(duracion).count() <= 0)
            cout << " NO HUBO RETRASO EN ESTE CICLO." << endl;
        //EN CASO CONTRARIO, MOSTRAR
        else
            cout << " RETRASO --> " << milliseconds_f(duracion).count() << "
milisegundos."<<endl;

        //MODIFICACION 3: EN CASO DE QUE EL RETARDO SEA MAYOR A 20ms. ABORTAR
        if (milliseconds_f(duracion).count() > 20.0){
            cerr << " <> Programa abortado, error. <>" << endl;
            exit(EXIT_FAILURE);
        }
    }
}

```

6.2 SEGUNDA PLANIFICACIÓN

```

//DANIEL PEREZ RUIZ

// -----
// Sistemas concurrentes y Distribuidos.
// Práctica 4. Implementación de sistemas de tiempo real.
//
// Archivo: ejecutivo2.cpp
// Implementación del primer ejercicio de ejecutivo cíclico:
//
// Datos de las tareas:
// -----
// Ta. T C
// -----
// A 500 100
// B 500 150
// C 1000 200
// D 2000 240
// -----
//
// Planificación (con Ts == 500 ms)
// *-----*-----*-----*-----*
// | A B C | A B D | A B C | A B | 

```

```

// *-----*-----*-----*
//
//
// Historial:
// Creado en Diciembre de 2017
// -----


/***
 * PREGUNTA 1:
 *
 * ¿Cuál es el mínimo tiempo de espera que queda al final de las
 * iteraciones del ciclo secundario con tu solución ?
 * -> El ciclo [A B D] (2ºCiclo) es el que más consume de CPU, consumiendo
 * un tiempo total de 490ms (100ms+150ms+240ms),
 * por lo que el mínimo tiempo de espera es de 500ms -490ms = 10ms.
 *
 *
 * PREGUNTA 2:
 * ¿sería planificable si la tarea D tuviese un tiempo cómputo de
 * 250 ms ?
 * -> Sí, sería planificable. Sin embargo sería un poco justo porque no habría
 * tiempo de espera en el segundo ciclo (secundario). Cualquier
 * otra espera adicional provocaría un pequeño retraso respecto al
 * siguiente ciclo.
*/

```

```

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

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)

/***
 * @brief Tarea generica : duerme durante un intervalo de tiempo
 * @param nombre : Nombre de tarea
 * @param tcomputo : Intervalo de tiempo
 */
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
    const milliseconds Ts(500);

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

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

        //CICLO SECUNDARIO (4 ITERACIONES)
        for(int i = 1; i <= 4; i++){

            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;

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

            //MODIFICACION 1: CALCULAMOS EL INSTANTE FINAL ACTUAL
            //--> TIPO DE DATOS USADO: TIME POINT
            //--> RELOJ UTILIZADO: STEADY CLOCK
            time_point<steady_clock> fin_sec = steady_clock::now();

            //MODIFICACION 2: CALCULO EL RETRASO DEL INSTANTE FINAL ACTUAL RESP.
            ESPERADO
            //TIPO DE DATOS USADO: DURATION
            steady_clock::duration duracion = fin_sec - ini_sec;

            //SI NO HA HABIDO RETARDO. NOTIFICAR
            if(milliseconds_f(duracion).count() <= 0)
                cout << " NO HUBO RETRASO EN ESTE CICLO." << endl;
            //EN CASO CONTRARIO, MOSTRAR
        }
    }
}

```

Encuentra el trabajo de tus sueños

Participa en retos y competiciones de programación



Escanéame y
obtén más info!!

```
        else
            cout << " RETRASO --> " << milliseconds_f(duracion).count() << "
milisegundos."<<endl;

//MODIFICACION 3: EN CASO DE QUE EL RETARDO SEA MAYOR A 20ms. ABORTAR
if (milliseconds_f(duracion).count() > 20.0){
    cerr << " >> Programa abortado, error. >>" << endl;
    exit(0);
}
}
```

7. FINAL

Si has llegado hasta aquí es porque has visto todo lo relacionado con la asignatura de Sistemas Concurrentes y Distribuidos. Espero que hayas tenido un buen paso por esta bonita asignatura, y que este documento te haya servido para el aprendizaje de la misma. Muchas gracias por leer este documento. Atentamente.

DanielsP



WUOLAH