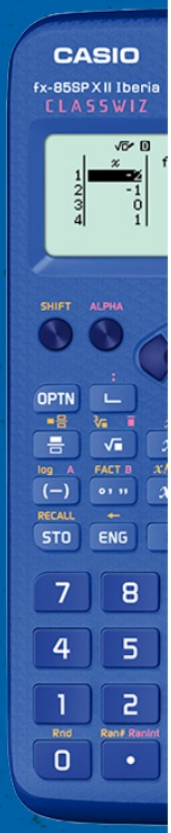
SCD2020G1EXAMENP1P2.pdf

danielsp10

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

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



SISTEMAS CONCURRENTES Y DISTRIBUIDOS

EXAMEN PRÁCTICAS 2020/21 - P1 Y P2 | [GRUPO 1]

Autor: DanielsP

En el siguiente documento se encuentran los enunciados del examen asociado a PRÁCTICA 1 y PRÁCTICA 2 realizado en el Doble Grado de Ingeniería Informática y Matemáticas (DGIIM) y Doble Grado de Ingeniería Informática y Administración y Dirección de Empresas (DGIIADE). Las soluciones se plantearán en otro documento diferente.

EJERCICIOS PRÁCTICA 1: (SEMÁFOROS)

1. **Sube el archivo con nombre `p1e1_sem.cpp` con tu solución al problema de los fumadores usando semáforos. Asegúrate que el número de fumadores es una constante `num_fumadores`, de forma que el programa funcione para cualquier número de fumadores simplemente cambiando el valor de esa constante. Dale a esa constante un valor mayor que 5. Resuelve la exclusión mutua en la salida de pantalla usando un semáforo en lugar de un objeto mútex.**
2. **Sube a la tarea un archivo de nombre exactamente `p1e2_sem.cpp` con una copia de `p1e1_sem.cpp` y extiéndelo para cumplir estos requerimientos adicionales:**
   - **Los fumadores, inmediatamente después de fumar, llaman a una función nueva que se llama `tirar_colilla` (simula la acción de tirar la colilla a una papelera).**
   - **Habrá una serie de nuevas hebras, llamadas hebras recolectoras. Define una constante con el número de dichas hebras, que debe ser divisor del número de fumadores. Cada hebra recolectora tiene asociada una papelera distinta. Las hebras recolectoras (y por tanto las papeleras) se numeran empezando en 0. Cada papelera tiene asociado un número de colillas en la misma (inicialmente cero).**
   - **Las hebras recolectoras ejecutan un bucle infinito, en cada iteración se bloquean esperando que su papelera esté llena, entonces son despertadas por un fumador, imprimen un mensaje, vacían la papelera (ponen su número de colillas a cero), y avisan al fumador que las ha despertado de que ya la han vaciado (ver siguiente punto).**
   - **El fumador número i usa la papelera cuyo número es el resto de la división entera de i entre el número de papeleras. En la función `tirar_colilla`, cada fumador debe tirar una colilla a su papelera, lo que significa: (a) incrementar el número de colillas de su papelera y después (b), si ese número ya ha llegado a 4, (la papelera está llena), debe desperatar a la hebra recolectora asociada a esa papelera y esperar a que la recolectora la vacíe.**

   Ten en cuenta que mientras un fumador está tirando una colilla a su papelera o esperando a que su papelera sea vaciada, ningún otro fumador puede tirar ninguna colilla a esa misma papelera (pero sí a otras).

EJERCICIOS PRÁCTICA 1: (SEMÁFOROS)

Nota: Esto es otra versión del ejercicio propuesto para el mismo grupo de prácticas (otro modelo, en el que sólo cambia algunas constantes y el nombre y significado de implementaciones a extender en el EJ2).

1. **Sube el archivo con nombre `p1e1_sem.cpp` con tu solución al problema de los fumadores usando semáforos. Asegúrate que el número de fumadores es una constante `num_fumadores`, de forma que el programa funcione para cualquier número de fumadores simplemente cambiando el valor de esa constante. Dale a esa constante un valor mayor que 10. Resuelve la exclusión mutua en la salida de pantalla usando un semáforo en lugar de un objeto mútex.**
2. **Sube a la tarea un archivo de nombre exactamente `p1e2_sem.cpp` con una copia de `p1e1_sem.cpp` y extiéndelo para cumplir estos requerimientos adicionales:**
   - **Los fumadores, inmediatamente después de fumar, llaman a una función nueva que se llama `actualizar_cenicero` (ver más adelante que hace).**
   - **Habrá una serie de nuevas hebras, llamadas hebras de limpieza. Define una constante con el número de dichas hebras, que debe ser divisor del número de fumadores. Cada hebra de limpieza tiene asociada un cenicero distinto. Las hebras de limpieza (y por tanto los ceniceros) se numeran empezando en 0. Para cada cenicero se lleva la cuenta de cuántas veces ha sido usado para fumar (inicialmente 0).**
   - **Las hebras de limpieza ejecutan un bucle infinito, en cada iteración se bloquean esperando que su cenicero esté muy sucio, entonces son despertadas por un fumador (que acaba de fumar usando ese cenicero), imprimen un mensaje, limpian el cenicero (ponen su número de usos a cero), y avisan al fumador que las ha despertado de que ya lo han limpiado (ver siguiente punto).**
   - **El fumador número i usa el cenicero cuyo número es el resto de la división entera de i entre el número de ceniceros. En la función `actualizar_cenicero`, cada fumador actualiza el cenicero que le corresponde, lo que significa: (a) incrementar el número de veces que el cenicero se ha usado y después (b), si ese número ya ha llegado a 7, (el cenicero está muy sucio), debe desperatar a la hebra de limpieza encargada de ese cenicero y (c) esperar a que lo limpie antes de continuar.**

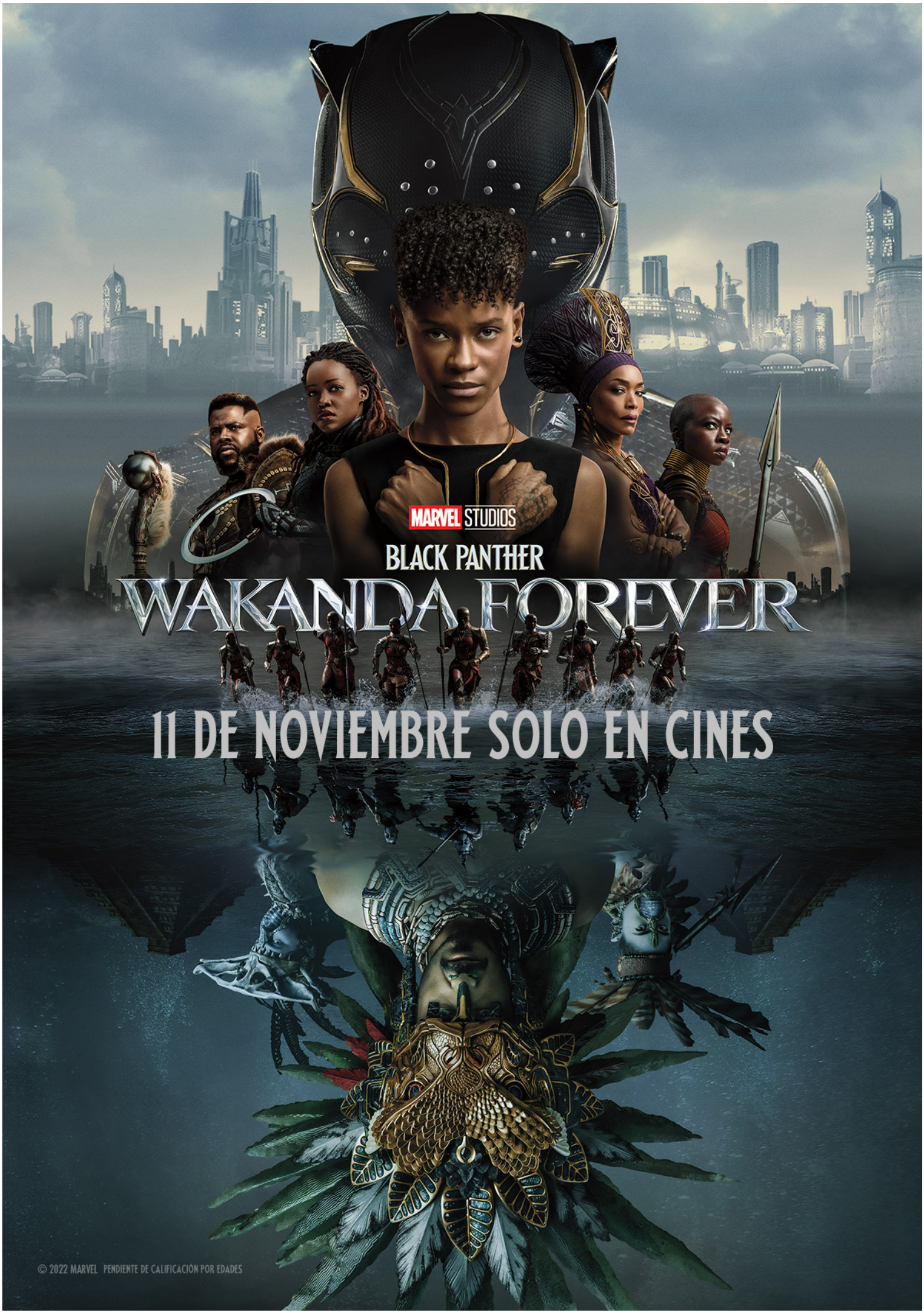
   Ten en cuenta que mientras un fumador está incrementando la cuenta de uso de su cenicero correspondiente, o esperando a que su cenicero sea limpiado, ningún otro fumador puede hacer lo mismo con ese cenicero (pero sí con otros).

A continuación se proporciona una plantilla de código para la realización del examen:

```
#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;

//======================================================================
```


MARVEL STUDIOS
BLACK PANTHER
WAKANDA FOREVER
11 DE NOVIEMBRE SOLO EN CINES
© 2022 MARVEL PENDIENTE DE CALIFICACIÓN POR EDADES

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

//CONSTANTE PEDIDA EN EL EXAMEN
const int num_fumadores = 5;

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

//======================================================================

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


/**
 * @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;
}
```

EJERCICIOS PRÁCTICA 2: (MONITORES)

1. **Sube el archivo con nombre `p2e1_msu.cpp` con tu solución al problema de los lectores-escritores usando un monitor SU.**
2. **Crea un nuevo archivo llamado `p2e2_msu.cpp` con una copia de `p2e1_msu.cpp` Extiende o modifica el programa para cumplir estos requerimientos adicionales a los del problema original.**
   - **En este programa queremos comparar el orden de llegada con el orden de acceso al recurso. El orden de llegada es el orden en el que las hebras logran iniciar los métodos ini..., y el orden de acceso al recurso es el orden en el que las hebras terminan de ejecutar esos métodos (independientemente del rol). Para lograr esto, se usa una variable *contador* (lleva el orden de llegada), inicializada a 0, que se va incrementando en cada llegada (al inicio de ini...), y después se copia sobre otra variable local al método. La variable local se imprime al final del método (cuando la hebra ya tiene garantizado el acceso para leer o escribir en el recurso compartido).**
   - **En esta nueva versión queremos intentar que las hebras accedan al recurso en el orden de llegada, es decir, el orden en el que inician la ejecución de los métodos ini..., independientemente de su rol. Se usan dos variables condición (en lugar de las originales), llamadas *resto* y *puerta*. En la puerta únicamente esperará como**

mucho una hebra hasta que le sea posible acceder al recurso. El resto de hebras que estén esperando, esperarán en la cola *resto* hasta que la puerta quede libre (la cola *resto* es previa a la *puerta*).

- **Al inicio de los métodos ini... (después de incrementar el contador y conocer su orden de llegada), cada hebra comprueba si hay alguna otra hebra esperando en la puerta, en ese caso espera en la cola resto. Después espera bloqueada en la puerta mientras que no puedan acceder al recurso. Aunque la hebra en la puerta se desbloquee, debe volver a comprobar si puede acceder (en bucle), y si no puede debe volver a bloquearse en la puerta. Cuando finalmente pueden acceder (al final, antes del mensaje), dan paso a una hebra en la cola resto (si hay alguna) para que acceda a la puerta.**
- **Al final de los métodos fin... (cuando ya han terminado de usar el recurso), las hebras desbloquean a la hebra en la puerta (si hay alguna), para que dicha hebra en la puerta pueda comprobar si ahora ya puede efectivamente acceder al recurso.**

A continuación se proporciona una plantilla de código para la realización del examen:

```
#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
 * -> Semática SU
 * -> Num Lectores : Múltiples
 * -> Num Escritores : Múltiples
 */
class Lec_Esc : public HoareMonitor{
private:
    //VARIABLES PERMANENTES
    int n_lec;     //Numero 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();
    }
}

/**
 * @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_lectores && 0 < num_escritores);
    thread lectores[num_lectores];
    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_lectores; 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_lectores; i++){
        lectores[i].join();
    }

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

    return 0;
}
```