SCD2020G3EXAMENP1P2.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 3]

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)

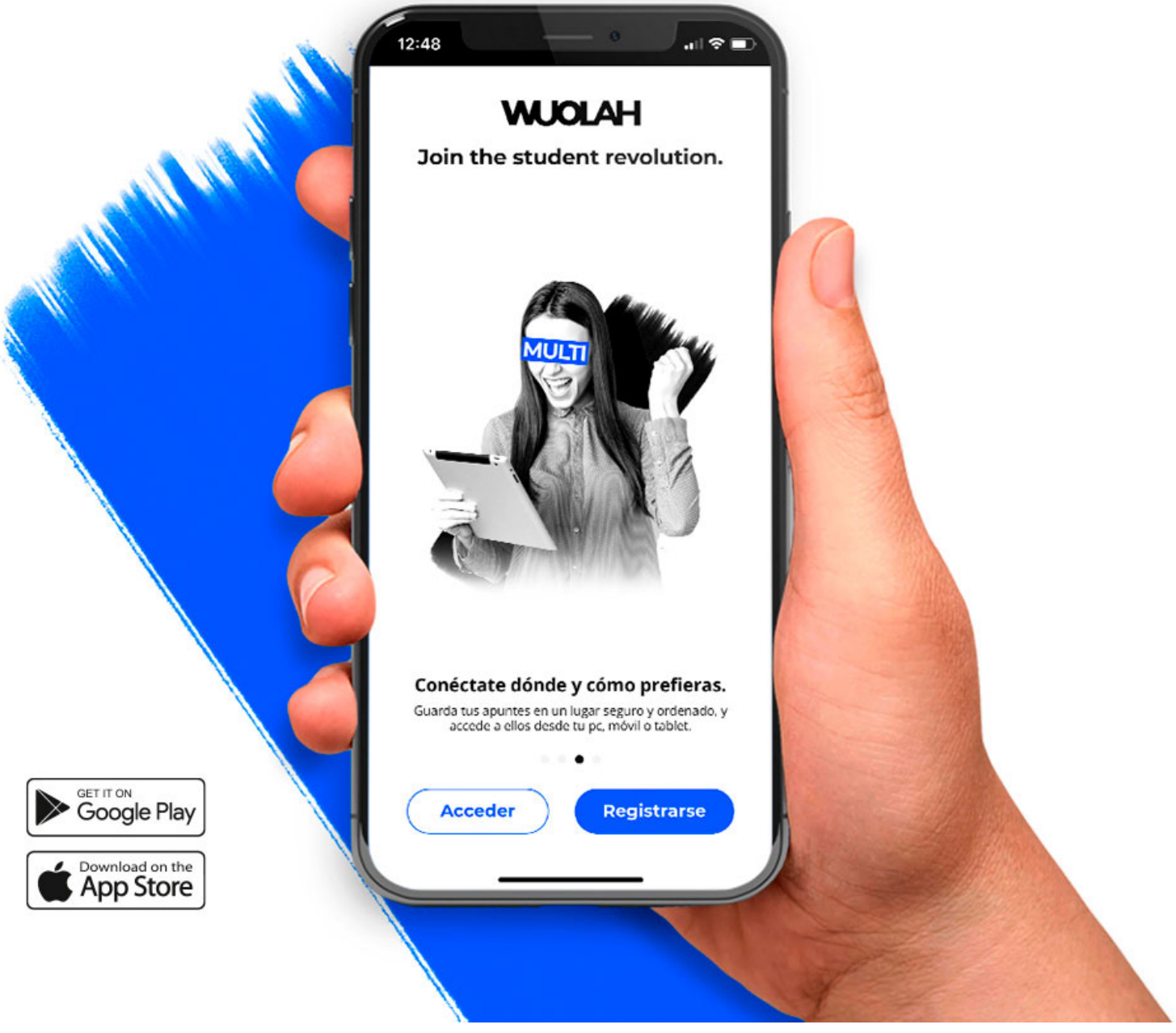
1. **Sube el archivo con nombre `p1e1_sem.cpp` con tu solución al problema de los fumadores usando semáforos. Asegúrate que el número de fumadores es una constante `num_fumadores`, de forma que el programa funcione para cualquier número de fumadores simplemente cambiando el valor de esa constante. Resuelve 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:**
   - **En lugar de haber un único estanquero, habrá un número arbitrario de ellos, definido por una constante con un valor no necesariamente igual a 1. En principio, cada estanquero funciona igual que el estanquero del problema original, todos comparten el mostrador. Cada estanquero se identifica por un número entero único, comenzando en 0. Cuando un estanquero imprime un mensaje, imprime también su número de estanquero.**
   - **En el mostrador habrá espacio para 4 raciones o items de ingredientes de cada tipo como mucho, en lugar de un único espacio para un único item de cualquier tipo de ingrediente como en el problema original (un item es lo que permite fumar a un fumador una vez). Por tanto, los estanqueros tienen que esperar antes de poner el ingrediente, si el mostrador ya tiene 4 items del ingrediente que quieren poner.**

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

```cpp
#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
//======================================================================

//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;
}
```

Exámenes, preguntas, apuntes.



GET IT ON Google Play

Download on the App Store

```
/**
 * @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 fumadores usando un monitor SU (basado en la clase HoareMonitor). Asegúrate que el número de fumadores es una constante `num_fumadores`, de forma que el programa funcion para cualquier número de fumadores simplemente cambiando el valor de esa constante.**
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 lugar de haber un único estanquero, habrá un número arbitrario de ellos, definido por una constante con un valor no necesariamente igual a 1. En principio, cada estanquero funciona igual que el estanquero del problema original, todos comparten el mostrador. Cada estanquero se identifica por un número entero único, comenzando en 0. Cuando un estanquero imprime un mensaje, también imprime su número de estanquero.**
    - **En el mostrador habrá espacio para 4 raciones o items de ingredientes de cada tipo como mucho, en lugar de un único espacio para un único item de cualquier tipo de ingrediente como en el problema original (un item es lo que permite fumar a un fumador una vez). Ahora es innecesario el método**

> **`esperarRecogidaIngrediente`, pero ten en cuenta que los estanqueros tienen que esperar antes de poner el ingrediente, si el mostrador ya tiene 4 items del ingrediente que quieren poner.**

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;
}
```