dosexamenesscd.pdf

pr0gramming_312823

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

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

Examen SCD Temas 1 y 2

1. **Modificar** la solución al problema de los **fumadores** planteado en la **Práctica 1** tal como se muestra a continuación, adjuntando el archivo .cpp resultante con nombre ejercicio1.cpp:

Habrá **tres nuevas hebras que suministrarán continuamente ingredientes al estanquero**, denominadas **suministradora[0],..., suministradora[2]**. Estas hebras serán prácticamente idénticas e **irán suministrando ingredientes continuamente al estanquero mediante un vector buffer con capacidad para 3 ingredientes que seguirá una estrategia de inserción/extracción LIFO**. Las tres hebras suministradoras **estarán continuamente generando ingredientes** (0, 1,ó 2) y **escribiéndolos en el vector buffer**.

El **estanquero no producirá ingredientes por sí mismo, sino que los leerá del vector compartido con las suministradoras** (vector buffer). De hecho, **el estanquero no podrá poner un nuevo ingrediente en el mostrador hasta que lo lea del vector buffer**.

Las hebras suministradoras tendrán que esperar si el vector buffer está lleno al intentar escribir en el vector, y la hebra estanquera tendrá que esperar si el buffer está vacío al intentar leer un nuevo ingrediente del vector.

Se requiere que las **hebras suministradoras y fumadoras se describan usando una única función parametrizada en base a un índice entero para cada grupo**, y que **la solución use arrays de hebras y arrays de semáforos (cuando proceda).**

2. **Modifica** tu solución al problema de los **Lectores-Escritores** de la **práctica 2**, tal como se indica a continuación, adjuntando el archivo .cpp resultante con nombre ejercicio2.cpp:

Se **lanzarán 4 hebras lectoras y 3 hebras escritoras**.

Existirá una nueva hebra, denominada "**revisora**", que **accederá periódicamente a la misma estructura de datos, al igual que lectores y escritores, usando operaciones similares para el acceso a dicha estructura, llamadas "ini_revison" y "fin_revision".**

La **hebra revisora solo podrá acceder a la estructura cuando ya haya un escritor escribiendo en la estructura**. Por lo tanto, en esta versión, **un escritor no tiene acceso exclusivo a la estructura**, ya que **la revisora podría estar accediendo a la estructura concurrentemente con dicho escritor**.

La **hebra revisora no podrá salir de su proceso de revisión mientras un escritor permanezca dentro**. Por tanto, **si la hebra revisora quiere salir y un escritor aún no ha salido, la revisora deberá esperar a que esté saliendo o haya salido dicho escritor**.

Un **lector podrá acceder** a la **estructura**, **aunque la hebra revisora aún no haya salido de la estructura**.

```
 1: #include <iostream>
 2: #include <cassert>
 3: #include <thread>
 4: #include <mutex>
 5: #include <random> // dispositivos, generadores y distribuciones aleatorias
 6: #include <chrono> // duraciones (duration), unidades de tiempo
 7: #include "Semaphore.h"
 8:
 9: using namespace std ;
10: using namespace SEM ;
11:
12: const int num_fumadores = 3;
13: const int num_suministradores = 3;
14: const int capacidad_buffer = 10;
15: Semaphore mostrador=1; //1 si no se atiende a nadie, 0 si estÃ¡ ocupado con algÃºn fumador
16: std::vector<Semaphore> ingredientes; // 1 si ingrediente i estÃ¡ disponible, 0 si no. Inicializado a 0 para solo poder
17: vector<Semaphore> suministradores;
18:
19: int buffer[capacidad_buffer];
20: int num_items_buffer = 0;
21: bool mostrador_ocupado = false;
22:
23: mutex mtx;
24:
25: //**********************************************************************
26: // plantilla de funciÃ³n para generar un entero aleatorio uniformemente
27: // distribuido entre dos valores enteros, ambos incluidos
28: // (ambos tienen que ser dos constantes, conocidas en tiempo de compilaciÃ³n)
29: //----------------------------------------------------------------------
30:
31: template< int min, int max > int aleatorio()
32: {
33:   static default_random_engine generador( (random_device())() );
34:   static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
35:   return distribucion_uniforme( generador );
36: }
37:
38: //----------------------------------------------------------------------
39: // FunciÃ³n que simula la acciÃ³n de producir un ingrediente, como un retardo
40: // aleatorio de la hebra (devuelve nÃºmero de ingrediente producido)
41:
42: int producir_ingrediente(int num_suministrador)
43: {
44:    // calcular milisegundos aleatorios de duraciÃ³n de la acciÃ³n de fumar)
45:    chrono::milliseconds duracion_produ( aleatorio<10,100>() );
46:
47:    // informa de que comienza a producir
```

Cuando disfrutas de tu gente y de la cerveza,
con cabeza, disfrutas el doble.

Cerveceros de España
100 años más

UNA GRAN CERVEZA.
UNA GRAN RESPONSABILIDAD.

```
48:    mtx.lock();
49:    cout << "Suministrador " << num_suministrador << " : empieza a producir ingrediente (" << duracion_produ.count() <<
50:    mtx.unlock();
51:
52:    // espera bloqueada un tiempo igual a ''duracion_produ' milisegundos
53:    this_thread::sleep_for( duracion_produ );
54:
55:    const int num_ingrediente = aleatorio<0,num_fumadores-1>() ;
56:
57:    // informa de que ha terminado de producir
58:    mtx.lock();
59:    cout << "Suministrador " << num_suministrador << " : termina de producir ingrediente " << num_ingrediente << endl;
60:    mtx.unlock();
61:
62:    return num_ingrediente ;
63: }
64:
65: void funcion_hebra_suministradora(int num_suministrador){
66:    int num_fumador;
67:    while(true){
68:       num_fumador = producir_ingrediente(num_suministrador);
69:       if(num_items_buffer > 9){
70:          mtx.lock();
71:          cout << "Suministrador " << num_suministrador << " se espera." << endl;
72:          mtx.unlock();
73:          sem_wait(suministradores[num_suministrador]);
74:       }
75:       if(num_items_buffer <= 9 && num_items_buffer >= 0){
76:          num_items_buffer++;
77:          buffer[num_items_buffer]=num_fumador;
78:          sem_signal(suministradores[num_suministrador]);
79:          if(!mostrador_ocupado){
80:             sem_signal(mostrador);
81:          }
82:       }
83:    }
84: }
85:
86: //----------------------------------------------------------------------
87: // funciÃ³n que ejecuta la hebra del estanquero
88:
89: void funcion_hebra_estanquero(  )
90: {
91:    int num_fumador;
92:    while(true){
93:       if(num_items_buffer == 0){
94:          sem_wait(mostrador);
```

```
 95:          sem_signal(suministradores[0]);
 96:          sem_signal(suministradores[1]);
 97:          sem_signal(suministradores[2]);
 98:       }
 99:       else if(num_items_buffer > 0 && !mostrador_ocupado){
100:          num_fumador = buffer[num_items_buffer];
101:          num_items_buffer--;
102:          mostrador_ocupado = true;
103:
104:          mtx.lock();
105:          cout << "Estanquero pone el ingrediente nÃºmero: " << num_fumador << endl;
106:          mtx.unlock();
107:
108:          sem_signal(ingredientes[num_fumador]);
109:
110:          if(num_items_buffer > 9){
111:             sem_signal(suministradores[0]);
112:             sem_signal(suministradores[1]);
113:             sem_signal(suministradores[2]);
114:          }
115:
116:       }
117:    }
118:
119: }
120:
121: //-------------------------------------------------------------------------
122: // FunciÃ³n que simula la acciÃ³n de fumar, como un retardo aleatoria de la hebra
123:
124: void fumar( int num_fumador )
125: {
126:
127:    // calcular milisegundos aleatorios de duraciÃ³n de la acciÃ³n de fumar)
128:    chrono::milliseconds duracion_fumar( aleatorio<20,200>() );
129:
130:    // informa de que comienza a fumar
131:    mtx.lock();
132:     cout << "Fumador " << num_fumador << "  :"
133:          << " empieza a fumar (" << duracion_fumar.count() << " milisegundos)" << endl;
134:    mtx.unlock();
135:    // espera bloqueada un tiempo igual a ''duracion_fumar' milisegundos
136:    this_thread::sleep_for( duracion_fumar );
137:
138:    // informa de que ha terminado de fumar
139:    mtx.lock();
140:     cout << "Fumador " << num_fumador << "  : termina de fumar, comienza espera de ingrediente." << endl;
141:    mtx.unlock();
```

```
142: }
143:
144: //---------------------------------------------------------------------
145: // funciÃ³n que ejecuta la hebra del fumador
146: void  funcion_hebra_fumador( int num_fumador )
147: {
148:    while( true )
149:    {
150:       sem_wait(ingredientes[num_fumador]);
151:
152:       mtx.lock();
153:       cout << "Se retira el ingrediente nÃºmero: " << num_fumador << endl;
154:       mtx.unlock();
155:
156:       mostrador_ocupado = false;
157:
158:       sem_signal(mostrador);
159:       fumar(num_fumador);
160:
161:    }
162: }
163:
164: //---------------------------------------------------------------------
165:
166: int main()
167: {
168:    // declarar hebras y ponerlas en marcha
169:    // ......
170:
171:    for(int i = 0; i < num_fumadores; i++)
172:       ingredientes.push_back(0);
173:    for(int i = 0; i < num_suministradores; i++)
174:       suministradores.push_back(0);
175:
176:    thread hebra_estanquero(funcion_hebra_estanquero);
177:    thread hebra_fumador[num_fumadores];
178:    thread hebra_suministradora[num_suministradores];
179:
180:    for(unsigned long i = 0; i < num_suministradores; i++)
181:       hebra_suministradora[i] = thread(funcion_hebra_suministradora, i);
182:
183:    for(unsigned long i = 0; i < num_fumadores; i++)
184:       hebra_fumador[i] = thread(funcion_hebra_fumador, i);
185:    hebra_estanquero.join();
186:
187:
188:    for(unsigned long i = 0; i < num_fumadores; i++)
```

```
189:       hebra_fumador[i].join();
190:    for(unsigned long i = 0; i < num_suministradores; i++)
191:       hebra_suministradora[i].join();
192:
193:
194: }
```

```
 1: #include <iostream>
 2: #include <iomanip>
 3: #include <cassert>
 4: #include <thread>
 5: #include <mutex>
 6: #include <condition_variable>
 7: #include <chrono> // duraciones (duration), unidades de tiempo
 8: #include <random> // dispositivos, generadores y distribuciones aleatorias
 9: #include "HoareMonitor.h"
10:
11: using namespace std;
12: using namespace HM;
13:
14: const int num_lectores = 4;
15: const int num_escritores = 3;
16:
17: mutex mtx;
18:
19: //**********************************************************************
20: // plantilla de funciÃ³n para generar un entero aleatorio uniformemente
21: // distribuido entre dos valores enteros, ambos incluidos
22: // (ambos tienen que ser dos constantes, conocidas en tiempo de compilaciÃ³n)
23: //----------------------------------------------------------------------
24:
25: template< int min, int max > int aleatorio()
26: {
27:   static default_random_engine generador( (random_device())() );
28:   static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
29:   return distribucion_uniforme( generador );
30: }
31:
32: void escribir(int escritor){
33:     chrono::milliseconds dur_escritura(aleatorio<20,200>());
34:
35:     //  Se empieza a escribir
36:     mtx.lock();
37:     cout << "El escritor " << escritor << " comienza a escribir ( " << dur_escritura.count() << " milisegundos)" << end
38:     mtx.unlock();
39:
40:     //  Espera bloqueada de dur_escritura milisegundos
41:     this_thread::sleep_for(dur_escritura);
42:
43:     //  Informa que ha terminado de escribir
44:     mtx.lock();
45:     cout << "El escritor " << escritor << " ha terminado de escribir" << endl;
46:     mtx.unlock();
47:
```

```
48: }
49:
50: void Espera(){
51:     chrono::milliseconds tiempo(aleatorio<20,200>());
52:     this_thread::sleep_for(tiempo);
53: }
54:
55: void leer(int lector){
56:     chrono::milliseconds dur_lectura(aleatorio<20,200>());
57:
58:     //  Se empieza a leer
59:     mtx.lock();
60:     cout << "El lector " << lector << " comienza a leer ( " << dur_lectura.count() << " milisegundos)" << endl;
61:     mtx.unlock();
62:
63:     //  Espera bloqueada de dur_lectura milisegundos
64:     this_thread::sleep_for(dur_lectura);
65:
66:     //  Informa que ha terminado de leer
67:     mtx.lock();
68:     cout << "El lector " << lector << " ha terminado de leer" << endl;
69:     mtx.unlock();
70:
71: }
72:
73: void revisar(){
74:
75:     chrono::milliseconds dur_lectura(aleatorio<100,400>());
76:
77:     mtx.lock();
78:     cout << "La revision esta siendo realizada..." << endl;
79:     mtx.unlock();
80:
81:     // this_thread::sleep_for(dur_lectura);
82:
83: }
84:
85: class LectoresEscritores: public HoareMonitor{
86:     private:
87:         int num_lec, num_esc;
88:         bool escribiendo, primera_vez;
89:
90:         CondVar lectura, escritura, revisora;
91:
92:     public:
93:         LectoresEscritores(){
94:             num_lec = 0;
```

```
 95:             num_esc = 0;
 96:             escribiendo = false;
 97:             primera_vez = true;
 98:             lectura = newCondVar();
 99:             escritura = newCondVar();
100:             revisora = newCondVar();
101:         }
102:         void iniLec(){
103:             if(escribiendo || num_esc == 0)
104:                 lectura.wait();
105:             num_lec++;
106:             lectura.signal();
107:         }
108:         void finLec(){
109:             num_lec--;
110:             if(num_lec == 0)
111:                 escritura.signal();
112:         }
113:         void iniEsc(){
114:             if(num_lec > 0 || escribiendo)
115:                 escritura.wait();
116:             escribiendo = true;
117:             revisora.signal();
118:         }
119:         void finEsc(){
120:             num_esc++;
121:             escribiendo = false;
122:             primera_vez = false;
123:
124:             if(lectura.get_nwt() != 0)
125:                 lectura.signal();
126:             else
127:                 escritura.signal();
128:         }
129:
130:         void ini_revision(){
131:
132:         }
133:
134:         void fin_revision(){
135:             while(escribiendo){
136:                 revisora.wait();
137:             }
138:             num_esc--;
139:
140:             mtx.lock();
141:             cout << "Se terminÃ³ la revisiÃ³n" << endl;
```

```
142:             mtx.unlock();
143:
144:             // if(num_esc != 0)
145:             //     revisora.signal();
146:
147:         }
148: };
149:
150: void funcion_hebra_revisora(MRef<LectoresEscritores> monitor){
151:     while(true){
152:         Espera();
153:         monitor->ini_revision();
154:         revisar();
155:         monitor->fin_revision();
156:     }
157: }
158:
159: void funcion_hebra_lector(MRef<LectoresEscritores> monitor, int numLectores){
160:     while(true){
161:         Espera();
162:         monitor->iniLec();
163:         leer(numLectores);
164:         monitor->finLec();
165:     }
166: }
167:
168: void funcion_hebra_escritor(MRef<LectoresEscritores> monitor, int numEscritores){
169:     while(true){
170:         Espera();
171:         monitor->iniEsc();
172:         escribir(numEscritores);
173:         monitor->finEsc();
174:     }
175: }
176:
177:
178: int main(){
179:     cout << "-------------------------------------------------------" << endl <<
180:             "- Problema de los lectores y escritores. Monitor SU. --" << endl <<
181:             "-------------------------------------------------------" << endl << flush;
182:     MRef<LectoresEscritores> monitor = Create<LectoresEscritores>();
183:
184:     thread hebras_lectoras[num_lectores], hebras_escritoras[num_escritores];
185:     thread hebra_revisora(funcion_hebra_revisora, monitor);
186:
187:     for(int i = 0; i < num_lectores; i++)
188:         hebras_lectoras[i] = thread(funcion_hebra_lector, monitor, i);
```

```
189:
190:     for(int i = 0; i < num_escritores; i++)
191:         hebras_escritoras[i] = thread(funcion_hebra_escritor, monitor, i);
192:
193:     for(int i = 0; i < num_lectores; i++)
194:         hebras_lectoras[i].join();
195:
196:     for(int i = 0; i < num_escritores; i++)
197:         hebras_escritoras[i].join();
198:
199:     hebra_revisora.join();
200:
201:
202:
203: }
```

Examen SCD Practica 1 y 2

1- lectores escritores con la misma prioridad en vez de que uno de ellos tenga prioridad sobre el otro

Hacer el problema con monitores. Se mantienen las mismas condiciones de concurrencia del problema original para lectores y escritores. Pero en este caso se deberá modificar el funcionamiento, para **que se vayan igualando el número de escrituras con el de lecturas**. Tened en cuenta que la priorización **se ha de hacer sólo cuando haya procesos de los 2 tipos esperando** (esto es, **si una de las colas está vacía y la otra no, habrá que liberar de esta cola que sí tiene**). **En caso de igualdad de lecturas y escrituras en un momento dado, priorizar aleatoriamente**. Sacad mensajes por pantalla para ir haciendo el seguimiento de estado.

2- Activista en el estanco

Considerando el problema que ya conocemos de los fumadores, añadir la siguiente casuística:

En el estanco que ya conocemos se cuela un activista de Green War, acérrimo defensor de la ley antitabaco. Y en un momento de descuido se encadena al mostrador.

Este activista (ciclo infinito) lo que va a hacer es lanzar proclamas antitabaco, que simularemos con el mensaje en pantalla “Fumar mata!”,y una espera aleatoria.

Sin embargo, no considera suficiente las proclamas y decide que tiene que impedir fumar a algún fumador de vez en cuando.

Para ello acuerda con el estanquero (el cual no quiere que la cosa vaya a mayores) que, periódicamente, va a esperar a que éste(el estanquero)**coloque el siguiente ingrediente en el mostrador y lo va a mangar** (sea cual sea). Acuerda también con nuestro estanquero que cuando decida mangar el ingrediente, se va a tumbar en el suelo, cesar las proclamas y se quedará parado (bloqueado) hasta que el estanquero lo despierte.

Cada vez que ejecuta 8 veces su código de proclamas y retardo aleatorio, se tumba y espera. Una vez liberado por el estanquero, manga el ingrediente, lo tira a la basura, se jacta de ello (con un mensaje tipo: “JA, JA, JA! Destruido ingrediente nº”, i), libera al estanquero porque ha vaciado el mostrador y vuelve a su ciclo normal (proclamas y espera).

El estanquero, cada vez que pone un ingrediente en el mostrador, mira si el activista está tumbado o no. Si no está tumbado, hace su tarea habitual. Si está tumbado, libera al activista (en vez de a un fumador) y se bloquea hasta que el activista lo libere a él después de mangar el ingrediente (para así volver al principio de su ciclo y producir un nuevo ingrediente). Implementar la solución con Semáforos, modificando el problema original y añadiendo al estanquero y 3 fumadores, el comportamiento también del activista.

Examen SCD Practica 1 y 2

3- Bebedores profesionales.

Ante la amenaza de confinamiento inminente y hartos de malas noticias en la tele, 4 amigos deciden irse a beber y olvidar a un bar hasta el fin de sus días. Los 4 son de ideas fijas además de tener buen saque, y siempre piden el mismo cóctel cada uno: Uno gin-tonic, otro mojito, otro daiquiri y el otro vermú.

En su taberna favorita detrás de la barra hay 2 camareros que, como conocen bien a sus parroquianos, desde que los ven entrar por la puerta empiezan a preparar y servir sus bebidas ya sin interrupción todo el tiempo.

En el bar, por el nuevo aforo reducido, van a poder estar bebiendo solo estos 4 amigos; y tras sentarse con la suficiente distancia de seguridad entre ellos y respetando todas las normas, empiezan cada uno a beber sin fin según les van poniendo las bebidas en la barra; cada uno la suya.

Hay una solo caja registradora, y como los camareros no se fían del estado en el que terminará todo esto, deciden cobrar cada vez que se sirve una bebida.

De forma que **sólo puede haber una bebida a la vez en la barra**; y **un tabernero no podrá servir otra hasta que la anterior bebida haya sido recogida y pagada por el cliente aficionado a ella** (que asumiremos que esto –recoger y pagar-lo hacen en un solo paso). **Cada camarero elegirá aleatoriamente qué bebida de las 4 preparar**, pues saben seguro que, pongan la que pongan en la barra, alguno de sus clientes se la beberá. Por último, para evitar que alguno de los clientes se emborrache demasiado pronto, **si un camarero en el momento de decidir el cóctel a preparar, detecta que el inicialmente elegido coincide con el último servido en la barra, cambiará y elegirá preparar otro cóctel**. De forma que **no se servirá en la barra el mismo cóctel 2 veces seguidas**.

Se pide implementar las 4 hebras bebedores y 2 camareros(barmans)en bucle infinito, con el funcionamiento anteriormente descrito. Añadir esperas aleatorias en los procesos de preparar el cóctel y, sobre todo, en el de beber (para que los muchachos se puedan tomar un respiro).

Sacar por pantalla las bebidas servidas por cada camarero en el momento de ponerlas en la barra y recogidas y bebidas por cada bebedor cada vez. Implementarlo con Monitores, atendiendo a las exclusiones mutuas necesarias.

```
Monitor Bebercio
 var
     variables permanentes
     variables condición

Function elegir_cocktail ();
begin
    · c:= decidir cocktail
    · actualizar estado (guardar
      ultima copa preparada)
    · devolver ( c );
end

Procedure servir_cocktail (c: integer)
begin
   · si hay un cocktail en la barra
           esperar (bloqueo)
   · actualizar estado (hay un cocktail en la barra)
   · liberar bebedor
end

Procedure pedir_cocktail (i:integer)
begin
    · si no está mi cocktail en la barra
          esperar (bloqueo)
    · actualizar estado (no hay cocktail
      en la barra)
    · liberar barman
end
```

```
Hebra_barman [i:0..1]
var cocktail: integer;
  begin
     while (true) do
       begin
          coktail:= Bebercio.elegir_cocktail();
          {retardo aleatorio} (prepara_cocktail)
          Bebercio.servir_cocktail (cocktail);
       end
  end
```

```
Hebra_bebedor [i:0..3]

  begin
    while (true) do
      begin
         Bebercio.pedir_cocktail(i);
         {retardo aleatorio} (beber feliz)
      end
  end
```

```
 1: #include <iostream>
 2: #include <cassert>
 3: #include <thread>
 4: #include <mutex>
 5: #include <random> // dispositivos, generadores y distribuciones aleatorias
 6: #include <chrono> // duraciones (duration), unidades de tiempo
 7: #include "Semaphore.h"
 8:
 9: using namespace std ;
10: using namespace SEM ;
11:
12: Semaphore disponibles[3] = {0, 0, 0};  //el cuarto fumador necesita el mismo ingrediente que el tercero (2)
13: Semaphore mostr_vacio = 1;  //semaforo del estanquero
14: Semaphore activista = 1;
15: int ultimo_ingrediente;  //para saber cuÃ¡l es el fumador (tercero o cuarto) que fumÃ³ la Ãºltima vez
16: Semaphore mensajes = 1;  //semaforo de exclusion mutua para que no se solapen los mensajes en pantalla
17:
18:
19: //**********************************************************************
20: // plantilla de funciÃ³n para generar un entero aleatorio uniformemente
21: // distribuido entre dos valores enteros, ambos incluidos
22: // (ambos tienen que ser dos constantes, conocidas en tiempo de compilaciÃ³n)
23: //----------------------------------------------------------------------
24:
25: template< int min, int max > int aleatorio()
26: {
27:   static default_random_engine generador( (random_device())() );
28:   static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
29:   return distribucion_uniforme( generador );
30: }
31:
32: int Producir(){
33:    int producido = aleatorio <0, 2>();
34:
35:    return producido;
36: }
37:
38:
39: //----------------------------------------------------------------------
40: // funciÃ³n que ejecuta la hebra del estanquero
41:
42: void funcion_hebra_estanquero(  )
43: {
44:
45:   while (true){
46:      int ingrediente = Producir();
47:      sem_wait (mostr_vacio);
```

```
48:       sem_wait(mensajes);
49:       cout << "El estanquero ha puesto el ingrediente " << ingrediente << " en el mostrador" << endl;
50:           ultimo_ingrediente = ingrediente;
51:       sem_signal(mensajes);
52:
53:           sem_signal(activista);
54:       sem_signal (disponibles[ingrediente]);
55:     }
56: }
57:
58: void funcion_hebra_activista(){
59:         while(true){
60:                 chrono::milliseconds duracion_aleatoria( aleatorio<20,200>() );
61:                 for(int i = 0 ; i < 7; i++){
62:                         cout << "Fumar mata!" << endl;
63:                         this_thread::sleep_for(duracion_aleatoria);
64:                 }
65:                 sem_wait(activista);    //      Se tumba y espera
66:                 //      Manga un ingrediente
67:                 sem_wait(disponibles[ultimo_ingrediente]);
68:                 cout << "JA, JA, JA! Destruido el ingrediente nÂº " << ultimo_ingrediente << endl;
69:                 sem_signal(mostr_vacio);
70:         }
71: }
72:
73: //-------------------------------------------------------------------------
74: // FunciÃ³n que simula la acciÃ³n de fumar, como un retardo aleatoria de la hebra
75:
76: void fumar( int num_fumador )
77: {
78:
79:    // calcular milisegundos aleatorios de duraciÃ³n de la acciÃ³n de fumar)
80:    chrono::milliseconds duracion_fumar( aleatorio<20,200>() );
81:
82:    // informa de que comienza a fumar
83:      sem_wait(mensajes);
84:      cout << "Fumador " << num_fumador << "  :"
85:           << " empieza a fumar (" << duracion_fumar.count() << " milisegundos)" << endl;
86:      sem_signal(mensajes);
87:
88:    // espera bloqueada un tiempo igual a ''duracion_fumar' milisegundos
89:    this_thread::sleep_for( duracion_fumar );
90:
91:    // informa de que ha terminado de fumar
92:      sem_wait(mensajes);
93:      cout << "Fumador " << num_fumador << "  : termina de fumar, comienza espera de ingrediente." << endl;
94:      sem_signal(mensajes);
```

```
 95:
 96: }
 97:
 98: //---------------------------------------------------------------------
 99: // funciÃ³n que ejecuta la hebra del fumador
100: void  funcion_hebra_fumador( int num_fumador )
101: {
102:    while( true )
103:    {
104:      sem_wait(mensajes);
105:      cout << "El fumador " << num_fumador << " espera el suministro " << num_fumador << endl;
106:      sem_signal(mensajes);
107:      sem_wait (disponibles[num_fumador]);
108:
109:      sem_wait(mensajes);
110:      cout << "El fumador " << num_fumador << " retira su ingrediente." << endl;
111:      sem_signal(mensajes);
112:      sem_signal (mostr_vacio);
113:      fumar (num_fumador);
114:
115:    }
116: }
117:
118: //---------------------------------------------------------------------
119:
120: int main()
121: {
122:    // declarar hebras y ponerlas en marcha
123:    // ......
124:
125:    cout << "\n Fumadores ex vicky.\n\n";
126:
127:    thread hebra_estanquero (funcion_hebra_estanquero);
128:    thread hebra_fumador[3];
129:    thread hebra_activista (funcion_hebra_activista);
130:
131:    for (int i = 0; i < 3; i++){
132:       hebra_fumador[i] = thread (funcion_hebra_fumador, i);
133:    }
134:
135:    hebra_estanquero.join();
136:    hebra_activista.join();
137:
138:    for (int i = 0; i < 4; i++){
139:       hebra_fumador[i].join();
140:    }
141: }
```

```
 1: #include <iostream>
 2: #include <iomanip>
 3: #include <cassert>
 4: #include <thread>
 5: #include <mutex>
 6: #include <condition_variable>
 7: #include <chrono> // duraciones (duration), unidades de tiempo
 8: #include <random> // dispositivos, generadores y distribuciones aleatorias
 9: #include "HoareMonitor.h"
10:
11: using namespace std;
12: using namespace HM;
13:
14: const int num_lectores = 3;
15: const int num_escritores = 3;
16:
17: mutex mtx;
18:
19: //**********************************************************************
20: // plantilla de funciÃ³n para generar un entero aleatorio uniformemente
21: // distribuido entre dos valores enteros, ambos incluidos
22: // (ambos tienen que ser dos constantes, conocidas en tiempo de compilaciÃ³n)
23: //----------------------------------------------------------------------
24:
25: template< int min, int max > int aleatorio()
26: {
27:   static default_random_engine generador( (random_device())() );
28:   static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
29:   return distribucion_uniforme( generador );
30: }
31:
32: void escribir(int escritor){
33:     chrono::milliseconds dur_escritura(aleatorio<20,200>());
34:
35:     //  Se empieza a escribir
36:     mtx.lock();
37:     cout << "El escritor " << escritor << " comienza a escribir ( " << dur_escritura.count() << " milisegundos)" << end
38:     mtx.unlock();
39:
40:     //  Espera bloqueada de dur_escritura milisegundos
41:     this_thread::sleep_for(dur_escritura);
42:
43:     //  Informa que ha terminado de escribir
44:     mtx.lock();
45:     cout << "El escritor " << escritor << " ha terminado de escribir" << endl;
46:     mtx.unlock();
47: }
```

```
48:
49: void Espera(){
50:     chrono::milliseconds tiempo(aleatorio<20,200>());
51:     this_thread::sleep_for(tiempo);
52: }
53:
54: void leer(int lector){
55:     chrono::milliseconds dur_lectura(aleatorio<20,200>());
56:
57:     //  Se empieza a leer
58:     mtx.lock();
59:     cout << "El lector " << lector << " comienza a leer ( " << dur_lectura.count() << " milisegundos)" << endl;
60:     mtx.unlock();
61:
62:     //  Espera bloqueada de dur_lectura milisegundos
63:     this_thread::sleep_for(dur_lectura);
64:
65:     mtx.lock();
66:     //  Informa que ha terminado de leer
67:     cout << "El lector " << lector << " ha terminado de leer" << endl;
68:     mtx.unlock();
69:
70: }
71:
72: class LectoresEscritores: public HoareMonitor{
73:     private:
74:         int num_lec, num_esc;
75:         bool escribiendo, escritores_esperando, lectores_esperando;
76:
77:         CondVar lectura, escritura;
78:
79:     public:
80:         LectoresEscritores(){
81:             num_lec = 0;
82:             num_esc = 0;
83:             escribiendo = false;
84:             escritores_esperando = false;
85:             lectores_esperando = false;
86:             lectura = newCondVar();
87:             escritura = newCondVar();
88:         }
89:         void iniLec(){
90:             // if(escribiendo)
91:             //     lectura.wait();
92:             // num_lec++;
93:             // lectura.signal();
94:
```

```
95:             if(escribiendo){     //  Si hay un escritor dentro
96:                 if(num_lec > num_esc){  //  Si se han hecho mÃ¡s lecturas que escrituras
97:                     if(!(!lectura.empty() && escritura.empty())){   //  Si no ocurre que la cola de lecturas estÃ¡ vacÃ
98:                         lectura.wait();
99:                     }
100:                }
101:                else if(num_lec == num_esc){     // Si se han hecho mismo numero de lecturas q de escrituras
102:                    if(!escritores_esperando){   //  Si el escritor no esta esperando al lector
103:                        int espero = aleatorio<1,2>();
104:                        if(espero == 1){
105:                            lectura.wait();
106:                            lectores_esperando = true;
107:                        }
108:                    }
109:                }
110:            }
111:
112:            num_lec++;
113:            lectura.signal();
114:
115:        }
116:        void finLec(){
117:            num_lec--;
118:            if(lectores_esperando){
119:                lectores_esperando = false;
120:            }
121:            if(num_lec == 0)
122:                escritura.signal();
123:        }
124:        void iniEsc(){
125:            if(escribiendo || num_lec > 0) {
126:                if(num_esc > num_lec){
127:                    if(!(!escritura.empty() && lectura.empty()))
128:                        escritura.wait();
129:                }
130:                else if (num_lec == num_esc){
131:                    if(!lectores_esperando){
132:                        int espero = aleatorio<0,1>();
133:
134:                        if(espero == 1){
135:                            escritura.wait();
136:                            escritores_esperando = true;
137:                        }
138:                    }
139:                }
140:            }
141:            escribiendo = true;
```

```
142:         }
143:         void finEsc(){
144:             escribiendo = false;
145:
146:             if(escritores_esperando)
147:                 escritores_esperando=false;
148:
149:             if(!lectura.empty())
150:                 lectura.signal();
151:             else
152:                 escritura.signal();
153:
154:
155:         }
156: };
157:
158: void funcion_hebra_lector(MRef<LectoresEscritores> monitor, int numLectores){
159:     while(true){
160:         Espera();
161:         monitor->iniLec();
162:         leer(numLectores);
163:         monitor->finLec();
164:     }
165: }
166:
167: void funcion_hebra_escritor(MRef<LectoresEscritores> monitor, int numEscritores){
168:     while(true){
169:         Espera();
170:         monitor->iniEsc();
171:         escribir(numEscritores);
172:         monitor->finEsc();
173:     }
174: }
175:
176:
177: int main(){
178:     cout << "-------------------------------------------------------" << endl <<
179:             "- Problema de los lectores y escritores. Monitor SU. --" << endl <<
180:             "-------------------------------------------------------" << endl << flush;
181:     MRef<LectoresEscritores> monitor = Create<LectoresEscritores>();
182:
183:     thread hebras_lectoras[num_lectores], hebras_escritoras[num_escritores];
184:
185:     for(int i = 0; i < num_lectores; i++)
186:         hebras_lectoras[i] = thread(funcion_hebra_lector, monitor, i);
187:
188:     for(int i = 0; i < num_escritores; i++)
```

```
189:          hebras_escritoras[i] = thread(funcion_hebra_escritor, monitor, i);
190:
191:      for(int i = 0; i < num_lectores; i++)
192:          hebras_lectoras[i].join();
193:
194:      for(int i = 0; i < num_escritores; i++)
195:          hebras_escritoras[i].join();
196:
197:
198:  }
```

```
 1: // CorrecciÃ³n del examen
 2: #include <iostream>
 3: #include <iomanip>
 4: #include <cassert>
 5: #include <thread>
 6: #include <mutex>
 7: #include <condition_variable>
 8: #include <random>
 9: #include "HoareMonitor.h"
10: using namespace std;
11: using namespace HM;
12:
13: constexpr int
14:    num_items  = 4 ;      // nÃºmero de items a producir/consumir
15:
16: mutex
17:    mtx ;                 // mutex de escritura en pantalla
18: unsigned
19:    cont_prod[num_items], // contadores de verificaciÃ³n: producidos
20:    cont_cons[num_items]; // contadores de verificaciÃ³n: consumidos
21: const int numero_productores=2; //numero de hebras productoras
22: const int numero_consumidores=4; //numero de hebras consumidoras
23:
24: int ultimo_servido = -1;
25: int valores_por_hebra[numero_productores]={0};
26: int producir_por_hebra = num_items/numero_productores;  //numero de valores que produce cada hebra productora
27: int consumir_por_hebra = num_items/numero_consumidores;  //numero de valores que consume cada hebra consumidora
28:
29:
30:
31: //**********************************************************************
32: // plantilla de funciÃ³n para generar un entero aleatorio uniformemente
33: // distribuido entre dos valores enteros, ambos incluidos
34: // (ambos tienen que ser dos constantes, conocidas en tiempo de compilaciÃ³n)
35: //----------------------------------------------------------------------
36:
37: template< int min, int max > int aleatorio()
38: {
39:   static default_random_engine generador( (random_device())() );
40:   static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
41:   return distribucion_uniforme( generador );
42: }
43:
44: //**********************************************************************
45: // funciones comunes a las dos soluciones (fifo y lifo)
46: //----------------------------------------------------------------------
47:
```

```
48: int elegir_cocktail(int numero_hebra)
49: {
50:    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
51:
52:    int producido;
53:
54:    do{
55:       producido = aleatorio<0,3>();
56:    }while(ultimo_servido == producido);   // no se servirÃ¡ en la barra el mismo cÃ³ctel 2 veces seguidas
57:
58:    return producido ;
59: }
60:
61: // *****************************************************************************
62: // clase para monitor buffer, version LIFO, semÃ¡ntica SC, un prod. y un cons.
63:
64: class Bebercio : public HoareMonitor
65: {
66: private:
67:    CondVar buffer[4], productores;
68:    int producido;
69:
70: public:
71:    Bebercio(){
72:       producido = -1;
73:       productores = newCondVar();
74:       for(int i = 0; i < numero_consumidores; i++)
75:          buffer[i]= newCondVar();
76:    }
77:
78:    void servir_cocktail(int c){
79:
80:       if(producido != -1){
81:          productores.wait();
82:       }
83:
84:       producido = c;
85:       buffer[c].signal();
86:
87:    }
88:
89:    void pedir_cocktail(int i){
90:
91:       if(i == producido){
92:          producido=-1;
93:          productores.signal();
94:       }
```

```
95:       else{
96:          buffer[i].wait();
97:       }
98:    }
99: } ;
100:
101: // *****************************************************************************
102: // funciones de hebras
103:
104: void funcion_hebra_barman( MRef<Bebercio> monitor, int num_hebra )
105: {
106:    // for( unsigned i = 0 ; i < producir_por_hebra*3 ; i++ ){
107:    while(true){
108:                int cocktail = elegir_cocktail(num_hebra) ;
109:       this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() )); // Preparando la bebida
110:                mtx.lock();
111:       cout << "Barman " << num_hebra << " ha servido el cocktail " << cocktail << endl;
112:       mtx.unlock();
113:       monitor->servir_cocktail( cocktail );
114:         }
115: }
116: //----------------------------------------------------------------------
117: void funcion_hebra_consumidora( MRef<Bebercio> monitor, int num_hebra )
118: {
119:    // for( unsigned i = 0 ; i < consumir_por_hebra*3 ; i++ ){
120:    while(true){
121:       mtx.lock();
122:       cout << "Consumidor " << num_hebra << " pide el cocktail " << num_hebra << endl;
123:       mtx.unlock();
124:       monitor->pedir_cocktail(num_hebra);
125:       this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() )); // Preparando la bebida
126:       mtx.lock();
127:       cout << "Consumidor " << num_hebra << " estÃ¡ bebiendo feliz." << endl;
128:       mtx.unlock();
129:    }
130: }
131: // ----------------------------------------------------------------------
132:
133: int main()
134: {
135:    cout << "------------------------------------------------------------------------------" << endl
136:         << " Productores consumidores con  (" << numero_productores
137:                  << " productores y " << numero_consumidores << " consumidores, versiÃ³n SU y LIFO). " << endl
138:         << "------------------------------------------------------------------------------" << endl
139:         << flush ;
140:
141:    MRef<Bebercio> monitor = Create<Bebercio>();
```

```
142:
143:    thread hebra_productora[numero_productores];
144:    thread hebra_consumidora[numero_consumidores];
145:
146:    for (int i=0;i<numero_productores;i++)
147:          hebra_productora[i] = thread(funcion_hebra_barman, monitor, i);
148:    for (int i =0;i<numero_consumidores;i++)
149:       hebra_consumidora[i] = thread(funcion_hebra_consumidora, monitor, i);
150:
151:    for (int i=0;i<numero_productores;i++)
152:       hebra_productora[i].join();
153:    for (int i =0;i<numero_consumidores;i++)
154:       hebra_consumidora[i].join();
155: }
```