```
 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
```

```
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:  }
```