otroexamen.pdf

pr0gramming_312823

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

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

Simulacro examen - Ejercicio1

1.- Modificar el problema de los fumadores mediante **semáforos** de la siguiente manera (el archivo se debe llamar "**fuma_ex.cpp**"):

Se debe añadir un cuarto fumador, que dispone de papel y tabaco. Cuando el estanquero produce cerillas, hay dos fumadores que están esperando por dicho ingrediente. La primera vez que ocurra esta situación, el estanquero decide de forma aleatoria a cuál de los dos fumadores desbloquea. A partir de ahí, va alternando a ambos fumadores cada vez que produce cerillas.

DURACIÓN: 30 MINUTOS (9:35- 10:05)

2.- Modificar el problema de los productores-consumidores mediante **monitores** de la siguiente manera (el archivo se deberá llamar "**prodcons_ex.cpp**"):

Se utilizarán dos vectores para la comunicación entre productores y consumidores (v1 y v2) cuyos tamaños serán dos parámetros del programa). Los productores con número impar utilizarán sólo el v1 y los de número par sólo el v2. Los consumidores podrán obtener elementos de ambos vectores (en caso de que ninguno de ellos esté vacío es indiferente de cuál de los dos cogen un elemento)

DURACIÓN: 40 MINUTOS (10:05 - 10:45)

3.- Un programa concurrente está formado por hebras que representan coches y que son bucles infinitos (el número de hebras será una constante del programa). En algún momento los coches llegan a una zona de peaje con dos cabinas y se colocan en la cabina cuya cola tenga menos coches (si tienen el mismo número es indiferente la cola) y esperan a que les toque el turno de pagar (obviamente, el primer coche en llegar no espera). El "pago" del peaje y la parte de código después de pagar el peaje se simulan con retardos aleatorios. Implementar dicho programa (el archivo se deberá llamar "**ejercicio3.cpp**") con un monitor de acuerdo al esquema propuesto, en el que la función del monitor "llegada_peaje" devuelve la cabina en la que se ha colocado el coche. Incluir mensajes que permitan seguir la traza del programa.

```
Monitor Peaje
 var
    variables permanentes
    variables condición

Function llegada_peaje: integer;
var  X: integer;
begin
    · X:= cabina con  menos coches en su cola
    · actualizar estado (otro coche en cola cabina X)
    · si hay más coches en cabina X → esperar (bloqueo)
    · devolver X
end

Procedure pagado (var cab: integer)

begin
    · actualizar estado
    · liberar un coche
end
```

```
Hebra_coche;
 var cabina: integer;
  begin
     while (true) do
        begin
           cabina:= Peaje.llegada_peaje;
           {retardo aleatorio} (pago del peaje)
           Peaje.pagado(cabina);
           {retardo aleatorio}  (resto de código)
        end
  end
```

```
 1: #include <iostream>
 2: #include <iomanip>
 3: #include <cassert>
 4: #include <thread>
 5: #include <mutex>
 6: #include <condition_variable>
 7: #include <random>
 8: #include "HoareMonitor.h"
 9:
10: using namespace std ;
11: using namespace HM ;
12:
13: const int num_coches = 8;
14: int coches_en_1=0,
15:     coches_en_2=0;
16:
17: class Peaje : public HoareMonitor{
18:
19: private:
20:
21: CondVar c_cabina1, c_cabina2 ;
22:
23: public:
24:    Peaje();
25:    int llegada_peaje();
26:    void pagado(int cab);
27: };
28:
29: Peaje::Peaje(){
30:    c_cabina1 = newCondVar();
31:    c_cabina2 = newCondVar();
32: }
33:
34: int Peaje::llegada_peaje(){
35:
36:    int x;
37:
38:    if(coches_en_1<=coches_en_2){
39:       x = 1;
40:       coches_en_1++;
41:       cout << "Coche nuevo en cabina 1 " << endl;
42:       if(coches_en_1>1){
43:          cout << "Coche espera a que termine de pagar en cola 1 " << endl;
44:          c_cabina1.wait();
45:       }
46:    }
47:    else{
```

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:         x = 2;
49:         coches_en_2++;
50:         cout << "Coche nuevo en cabina 2 " << endl;
51:         if(coches_en_2>1){
52:            cout << "Coche espera a que termine de pagar en cola 2 " << endl;
53:            c_cabina2.wait();
54:         }
55:     }
56:
57:     return x;
58: }
59:
60: void Peaje::pagado(int cab){
61:
62:     if(cab==1){
63:         coches_en_1--;
64:         cout << "\tCoche en cabina 1 pagado, siguiente coche entra " << endl;
65:         c_cabina1.signal();
66:     }
67:     if(cab==2){
68:         coches_en_2--;
69:         cout << "\tCoche en cabina 2 pagado, siguiente coche entra " << endl;
70:         c_cabina2.signal();
71:     }
72: }
73:
74: template< int min, int max > int aleatorio()
75: {
76:   static default_random_engine generador( (random_device())() );
77:   static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
78:   return distribucion_uniforme( generador );
79: }
80:
81: void espera_pago_peaje(){
82:   chrono::milliseconds duracion_espera( aleatorio<500,2000>() );
83:   this_thread::sleep_for(duracion_espera);
84: }
85:
86: void espera_llegada_peaje(){
87:   chrono::milliseconds duracion_espera( aleatorio<500,1000>() );
88:   this_thread::sleep_for(duracion_espera);
89: }
90:
91:
92: void funcion_hebra_coche(MRef<Peaje> monitor, int i){
93:    int cabina;
94:    while(true){
```

```
 95:
 96:       espera_llegada_peaje();
 97:       cout << "Coche " << i << " ha llegado..." << endl;
 98:
 99:       cabina = monitor->llegada_peaje();
100:
101:       espera_pago_peaje();
102:
103:       monitor->pagado(cabina);
104:
105:       cout << "Coche " << i << " acaba de pagar y se retira... " << endl;
106:
107:
108:    }
109:
110: }
111:
112: int main(int argc, char const *argv[]) {
113:   MRef<Peaje> monitor = Create<Peaje>();
114:
115:   thread hebras_coches[num_coches];
116:
117:
118:   for(int i = 0; i < num_coches; i++)
119:     hebras_coches[i] = thread ( funcion_hebra_coche, monitor,i);
120:
121:
122:   for(int i = 0; i < num_coches; i++)
123:     hebras_coches[i].join();
124: }
125:
126:
```

```
 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[4] = {0, 0, 0, 0};  //el cuarto fumador necesita el mismo ingrediente que el tercero (2)
13: Semaphore mostr_vacio = 1;  //semaforo del estanquero
14: int ultima_vez;  //para saber cuÃ¡l es el fumador (tercero o cuarto) que fumÃ³ la Ãºltima vez
15: bool primera_vez = true;  //para decidir que fumador fuma la primera vez que se ponga el ingrediente 2
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: // funciÃ³n que ejecuta la hebra del estanquero
40:
41: void funcion_hebra_estanquero(  )
42: {
43:
44:   while (true){
45:      int ingrediente = Producir();
46:      sem_wait (mostr_vacio);
47:      sem_wait(mensajes);
```

```
48:       cout << "El estanquero ha puesto el ingrediente " << ingrediente << " en el mostrador" << endl;
49:       sem_signal(mensajes);
50:
51:       if (ingrediente == 2)
52:       {
53:          if (primera_vez)
54:             {
55:               ingrediente = aleatorio <2, 3>();
56:               primera_vez = false;
57:             }
58:          else{
59:             if (ultima_vez == 2)
60:                ingrediente = 3;
61:             else
62:                ingrediente = 2;
63:             }
64:          ultima_vez = ingrediente;
65:       }
66:
67:          sem_signal (disponibles[ingrediente]);
68:    }
69: }
70:
71: //-------------------------------------------------------------------------
72: // FunciÃ³n que simula la acciÃ³n de fumar, como un retardo aleatoria de la hebra
73:
74: void fumar( int num_fumador )
75: {
76:
77:    // calcular milisegundos aleatorios de duraciÃ³n de la acciÃ³n de fumar)
78:    chrono::milliseconds duracion_fumar( aleatorio<20,200>() );
79:
80:    // informa de que comienza a fumar
81:       sem_wait(mensajes);
82:       cout << "Fumador " << num_fumador << "  :"
83:            << " empieza a fumar (" << duracion_fumar.count() << " milisegundos)" << endl;
84:       sem_signal(mensajes);
85:
86:    // espera bloqueada un tiempo igual a ''duracion_fumar' milisegundos
87:    this_thread::sleep_for( duracion_fumar );
88:
89:    // informa de que ha terminado de fumar
90:       sem_wait(mensajes);
91:       cout << "Fumador " << num_fumador << "  : termina de fumar, comienza espera de ingrediente." << endl;
92:       sem_signal(mensajes);
93:
94: }
```

```
 95:
 96: //------------------------------------------------------------------------
 97: // funciÃ³n que ejecuta la hebra del fumador
 98: void  funcion_hebra_fumador( int num_fumador )
 99: {
100:    while( true )
101:    {
102:      sem_wait(mensajes);
103:      cout << "El fumador " << num_fumador << " espera el suministro " << num_fumador << endl;
104:      sem_signal(mensajes);
105:      sem_wait (disponibles[num_fumador]);
106:
107:      sem_wait(mensajes);
108:      cout << "El fumador " << num_fumador << " retira su ingrediente." << endl;
109:      sem_signal(mensajes);
110:      sem_signal (mostr_vacio);
111:      fumar (num_fumador);
112:
113:    }
114: }
115:
116: //------------------------------------------------------------------------
117:
118: int main()
119: {
120:    // declarar hebras y ponerlas en marcha
121:    // ......
122:
123:    cout << "\nLos fumadores 2 y 3 son los que necesitan cerillas para fumar.\n\n";
124:
125:    thread hebra_estanquero (funcion_hebra_estanquero);
126:    thread hebra_fumador[4];
127:    thread hebra_cuarto_fumador;
128:
129:    for (int i = 0; i < 4; i++){
130:       hebra_fumador[i] = thread (funcion_hebra_fumador, i);
131:    }
132:
133:    hebra_estanquero.join();
134:
135:    for (int i = 0; i < 4; i++){
136:       hebra_fumador[i].join();
137:    }
138: }
```

```
 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  = 40 ;      // 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=8; //numero de hebras productoras
22: const int numero_consumidores=5; //numero de hebras consumidoras
23:
24: int valores_por_hebra[numero_productores]={0};
25: int producir_por_hebra = num_items/numero_productores;  //numero de valores que produce cada hebra productora
26: int consumir_por_hebra = num_items/numero_consumidores;  //numero de valores que consume cada hebra consumidora
27: //**********************************************************************
28: // plantilla de funciÃ³n para generar un entero aleatorio uniformemente
29: // distribuido entre dos valores enteros, ambos incluidos
30: // (ambos tienen que ser dos constantes, conocidas en tiempo de compilaciÃ³n)
31: //----------------------------------------------------------------------
32:
33: template< int min, int max > int aleatorio()
34: {
35:   static default_random_engine generador( (random_device())() );
36:   static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
37:   return distribucion_uniforme( generador );
38: }
39:
40: //**********************************************************************
41: // funciones comunes a las dos soluciones (fifo y lifo)
42: //----------------------------------------------------------------------
43:
44: int producir_dato(int numero_hebra)
45: {
46:    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
47:    mtx.lock();
```

```
48:    int producido = numero_hebra*producir_por_hebra+valores_por_hebra[numero_hebra];
49:    cout << "productor: " << numero_hebra <<" \tproducido: " << producido << endl << flush ;
50:    mtx.unlock();
51:    cont_prod[producido] ++ ;
52:    valores_por_hebra[numero_hebra]++; //aumentamos en uno los valores producidos por la hebra
53:    return producido ;
54: }
55: //----------------------------------------------------------------------
56:
57: void consumir_dato( int numero_hebra, unsigned dato )
58: {
59:    if ( num_items <= dato )
60:    {
61:       cout << " dato === " << dato << ", num_items == " << num_items << endl ;
62:       assert( dato < num_items );
63:    }
64:    cont_cons[dato] ++ ;
65:    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
66:    mtx.lock();
67:    cout << "consumidor: " << numero_hebra << "            \t\tconsumido: " << dato << endl ;
68:    mtx.unlock();
69: }
70: //----------------------------------------------------------------------
71:
72: void ini_contadores()
73: {
74:    for( unsigned i = 0 ; i < num_items ; i++ )
75:    {  cont_prod[i] = 0 ;
76:       cont_cons[i] = 0 ;
77:    }
78: }
79:
80: //----------------------------------------------------------------------
81:
82: void test_contadores()
83: {
84:    bool ok = true ;
85:    cout << "comprobando contadores ...." << flush ;
86:
87:    for( unsigned i = 0 ; i < num_items ; i++ )
88:    {
89:       if ( cont_prod[i] != 1 )
90:       {
91:          cout << "error: valor " << i << " producido " << cont_prod[i] << " veces." << endl ;
92:          ok = false ;
93:       }
94:       if ( cont_cons[i] != 1 )
```

```
 95:       {
 96:          cout << "error: valor " << i << " consumido " << cont_cons[i] << " veces" << endl ;
 97:          ok = false ;
 98:       }
 99:    }
100:    if (ok)
101:       cout << endl << flush << "soluciÃ³n (aparentemente) correcta." << endl << flush ;
102: }
103:
104: // *****************************************************************************
105: // clase para monitor buffer, version LIFO, semÃ¡ntica SC, un prod. y un cons.
106:
107: class ProdCons2SU : public HoareMonitor  //definimos la clase como derivada de tipo public
108: {
109:  private:
110:  static const int           // constantes:
111:    num_celdas_pares = 5,     //  nÃºm. de entradas del buffer v2
112:    num_celdas_impares = 4;   //  nÃºm. de entradas del buffer v1
113:  int                        // variables permanentes
114:    v2[num_celdas_pares],//  buffer de tamaÃ±o fijo donde introduciran las hebras pares
115:    v1[num_celdas_impares], //buffer de tamaÃ±o fijo donde introducirÃ¡n las hebras impares
116:    primera_libre_pares,            //indica la celda de la proxima inserciÃ³n
117:    primera_libre_impares;          //  indice de celda de la prÃ³xima inserciÃ³n
118:  CondVar         // colas condicion:
119:    ocupadas,                //  cola donde espera el consumidor (n>0)
120:    libres_pares,                   //cola donde espera el productor par
121:    libres_impares ;                //  cola donde espera el productor impar (n<num_celdas_total)
122:
123:  public:                    // constructor y mÃ©todos pÃºblicos
124:    ProdCons2SU(  ) ;           // constructor
125:    int  leer();                // extraer un valor (sentencia L) (consumidor)
126:    void escribir_pares( int valor ); // insertar un valor (sentencia E) (productor)
127:    void escribir_impares( int valor );
128: } ;
129: // -----------------------------------------------------------------------------
130:
131: ProdCons2SU::ProdCons2SU()
132: {
133:    primera_libre_pares = 0 ;
134:    primera_libre_impares = 0;
135:    libres_pares=newCondVar();
136:    ocupadas=newCondVar();
137:    libres_impares=newCondVar();
138: }
139: // -----------------------------------------------------------------------------
140: // funciÃ³n llamada por el consumidor para extraer un dato
141:
```

```
142: int ProdCons2SU::leer(  )
143: {
144:
145:    // esperar bloqueado hasta que 0 < num_celdas_ocupadas
146:    if ( primera_libre_pares == 0 && primera_libre_impares==0)
147:       ocupadas.wait();
148:
149:    int valor=0;
150:
151:    if(primera_libre_pares>0 && primera_libre_impares>0) {
152:
153:       int de_donde_leer = aleatorio<1,2>();
154:
155:       if (de_donde_leer = 2){
156:           primera_libre_pares-- ;
157:           valor = v2[primera_libre_pares] ;
158:           libres_pares.signal();
159:       }
160:       else{
161:           primera_libre_impares--;
162:           valor = v1[primera_libre_impares];
163:           libres_impares.signal();
164:       }
165:    }
166:    else if(primera_libre_pares>0){
167:       primera_libre_pares-- ;
168:       valor = v2[primera_libre_pares] ;
169:       libres_pares.signal();
170:    }
171:    else {
172:       primera_libre_impares--;
173:       valor = v1[primera_libre_impares];
174:       libres_impares.signal();
175:    }
176:
177:
178:    // devolver valor
179:    return valor ;
180: }
181: // -----------------------------------------------------------------------------
182:
183: void ProdCons2SU::escribir_pares( int valor )
184: {
185:
186:    // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total
187:    if ( primera_libre_pares == num_celdas_pares )
188:       libres_pares.wait();
```

```
189:
190:    // hacer la operaciÃ³n de inserciÃ³n, actualizando estado del monitor
191:    v2[primera_libre_pares] = valor ;
192:    primera_libre_pares++ ;
193:
194:    // seÃ±alar al consumidor que ya hay una celda ocupada (por si esta esperando)
195:    ocupadas.signal();
196: }
197: // *****************************************************************************
198:
199: void ProdCons2SU::escribir_impares( int valor )
200: {
201:
202:    // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total
203:    if ( primera_libre_impares == num_celdas_impares )
204:       libres_impares.wait();
205:
206:    // hacer la operaciÃ³n de inserciÃ³n, actualizando estado del monitor
207:    v1[primera_libre_impares] = valor ;
208:    primera_libre_impares++ ;
209:
210:    // seÃ±alar al consumidor que ya hay una celda ocupada (por si esta esperando)
211:    ocupadas.signal();
212: }
213:
214: // *****************************************************************************
215: // funciones de hebras
216:
217: void funcion_hebra_productora( MRef<ProdCons2SU> monitor, int numero_hebra )
218: {
219:    for( unsigned i = 0 ; i < producir_por_hebra ; i++ )
220:    {
221:       int valor = producir_dato(numero_hebra) ;
222:
223:       if(numero_hebra%2==0)
224:          monitor->escribir_pares( valor );
225:       else
226:         monitor->escribir_impares(valor);
227:    }
228: }
229: // -----------------------------------------------------------------------------
230:
231: void funcion_hebra_consumidora( MRef<ProdCons2SU> monitor, int numero_hebra )
232: {
233:    for( unsigned i = 0 ; i < consumir_por_hebra ; i++ )
234:    {
235:       int valor = monitor->leer();
```

```
236:       consumir_dato( numero_hebra, valor ) ;
237:    }
238: }
239: // -----------------------------------------------------------------------------
240:
241: int main()
242: {
243:    cout << "--------------------------------------------------------------------------------" << endl
244:         << "Problema de los productores-consumidores (1 prod/cons, Monitor SC, buffer FIFO). " << endl
245:         << "--------------------------------------------------------------------------------" << endl
246:         << flush ;
247:
248:    MRef<ProdCons2SU> monitor = Create<ProdCons2SU>();
249:
250:    thread hebra_productora[numero_productores];
251:    thread hebra_consumidora[numero_consumidores];
252:
253:    for (int i=0;i<numero_productores;i++)
254:        hebra_productora[i] = thread(funcion_hebra_productora, monitor, i);
255:    for (int i =0;i<numero_consumidores;i++)
256:        hebra_consumidora[i] = thread(funcion_hebra_consumidora, monitor, i);
257:
258:    for (int i=0;i<numero_productores;i++)
259:        hebra_productora[i].join();
260:    for (int i =0;i<numero_consumidores;i++)
261:        hebra_consumidora[i].join();
262:
263:    // comprobar que cada item se ha producido y consumido exactamente una vez
264:    test_contadores() ;
265: }
```