UNIVERSIDAD
DE GRANADA

2ºA - Grupo A1

Grado en Ingeniería Informática

Sistemas Concurrentes y Distribuidos
Práctica 1 - Sincronización de hebras con semáforos

Autor:
Jose Luis Gallego Peña

11 de octubre de 2018

Índice

1. El problema del productor-consumidor

Se ha optado por implementar la solución con el vector LIFO.

1.1. Código fuente de la solución adoptada

Código 1: prodcons.cpp

```
1
2  #include <iostream>
3  #include <cassert>
4  #include <thread>
5  #include <mutex>
6  #include <random>
7  #include "Semaphore.h"
8
9  using namespace std;
10 using namespace SEM;
11
12 //**********************************************************************
13 // variables compartidas
14
15 const int num_items = 20,   // número de items
16           tam_vec   = 10;   // tamaño del buffer
17 unsigned  cont_prod[num_items] = {0}, // contadores de verificación: producidos
18           cont_cons[num_items] = {0}; // contadores de verificación: consumidos
19
20 int vec[tam_vec]; // Vector buffer intermedio
21
22 unsigned primera_libre = 0;  // Índice en el vector de la primera celda libre
23
24 // Semáforos
25 Semaphore ocupadas = 0;       // Número de entradas ocupadas (E - L)
26 Semaphore libres = tam_vec;   // Número de entradas libres (k + L - E)
27 Semaphore mut = 1;            // Semáforo para la exclusión mutua en el vector
28
29 //**********************************************************************
30 // plantilla de función para generar un entero aleatorio uniformemente
31 // distribuido entre dos valores enteros, ambos incluidos
32 // (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
33 //----------------------------------------------------------------------
34
35 template< int min, int max > int aleatorio(){
36    static default_random_engine generador( (random_device())() );
37    static uniform_int_distribution<int> distribucion_uniforme( min, max );
38    return distribucion_uniforme( generador );
39 }
40
41 //**********************************************************************
42 // funciones comunes a las dos soluciones (fifo y lifo)
43 //----------------------------------------------------------------------
```

```
44
45 int producir_dato(){
46
47    static int contador = 0;
48    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
49
50    cout << "producido: " << contador << endl << flush;
51
52    cont_prod[contador]++;
53    return contador++;
54
55 }
56
57 //----------------------------------------------------------------------
58
59 void consumir_dato( unsigned dato ){
60
61    assert(dato < num_items);
62    cont_cons[dato]++;
63    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
64
65    cout << "                    consumido: " << dato << endl;
66
67 }
68
69 //----------------------------------------------------------------------
70
71 void test_contadores(){
72
73    bool ok = true;
74    cout << "comprobando contadores ....";
75
76    for(unsigned i = 0 ; i < num_items ; i++){
77       if (cont_prod[i] != 1){
78          cout << "error: valor " << i << " producido " << cont_prod[i]
79               << " veces." << endl;
80          ok = false ;
81       }
82       if (cont_cons[i] != 1){
83          cout << "error: valor " << i << " consumido " << cont_cons[i]
84               << " veces" << endl;
85          ok = false;
86       }
87    }
88
89    if (ok)
90       cout << endl << flush << "Solución (aparentemente) correcta. Fin." << endl
91            << flush;
92 }
93
94 //----------------------------------------------------------------------
95
96 void funcion_hebra_productora(){
```

```
97
98      for(unsigned i = 0 ; i < num_items ; i++){
99         int dato = producir_dato();
100
101        sem_wait(libres);
102        sem_wait(mut);
103        // Sección crítica
104        vec[primera_libre] = dato; // Insertar el dato en el buffer (escribir)
105        primera_libre++;
106        cout << endl << "(Vector libres: " << tam_vec - primera_libre
107             << ", ocupadas: " << primera_libre << ")" << endl;
108        sem_signal(mut);
109        sem_signal(ocupadas);
110
111     }
112  }
113
114  //-------------------------------------------------------------------------
115
116  void funcion_hebra_consumidora(){
117
118     for(unsigned i = 0 ; i < num_items ; i++){
119        int dato;
120
121        sem_wait(ocupadas);
122        sem_wait(mut);
123        // Sección crítica
124        dato = vec[primera_libre - 1]; // Extraer el dato del buffer (leer)
125        primera_libre--;
126        cout << endl << "(Vector libres: " << tam_vec - primera_libre
127             << ", ocupadas: " << primera_libre << ")" << endl;
128        sem_signal(mut);
129        sem_signal(libres);
130
131        consumir_dato(dato);
132     }
133  }
134
135  //-------------------------------------------------------------------------
136
137  int main(){
138
139     cout << "-------------------------------------------------------------" << endl
140          << "Problema de los productores-consumidores (solución LIFO)." << endl
141          << "-------------------------------------------------------------" << endl
142          << flush;
143
144     thread hebra_productora(funcion_hebra_productora),
145            hebra_consumidora(funcion_hebra_consumidora);
146
147     hebra_productora.join();
148     hebra_consumidora.join();
149
```

```
150     test_contadores();
151 }
```

1.2. Descripción de las variables

El programa consta de las siguientes variables compartidas (globales):

- **const int num_items**

  Número entero constante que indica el número de items que se van a producir y consumir. En el programa tiene de valor 90.

- **const int tam_vec**

  Número entero constante que indica el tamaño del vector buffer intermedio en el que se irán almacenando (escribiendo) los valores producidos que aún no han sido consumidos (leídos).

- **unsigned cont_prod[num_items]**

  Vector de datos de tipo unsigned (enteros sin signo) con tamaño num_items. Indica en la posición i del vector el número de veces que se ha producido el item número i. Esta variable es usada por la función test_contadores() para comprobar si la solución es correcta, es decir, comprueba si todos los items se han producido una sola vez.

- **unsigned cont_cons[num_items]**

  Vector de datos de tipo unsigned (enteros sin signo) con tamaño num_items. Indica en la posición i del vector el número de veces que se ha consumido el item número i. Esta variable es usada por la función test_contadores() para comprobar si la solución es correcta, es decir, comprueba si todos los items se han consumido una sola vez.

- **int vec[tam_vec]**

  Vector buffer intermedio en el que se irán almacenando (escribiendo) los valores producidos que aún no han sido consumidos (leídos). Es de tamaño tam_vec.

- **unsigned primera_libre**

  Variable de tipo unsigned (entero sin signo) que indica el índice del vector buffer en el cual está la primera celda libre para así poder insertar el item producido en una posición no ocupada. Puesto que esta variable siempre indica la primera celda libre, la celda de la cual se lee el dato del buffer será la primera_libre-1 (es decir, el último dato escrito en el buffer, ya que esta es la solución LIFO, el último en ser insertado es el primero en ser extraído).

1.3. Descripción de los semáforos

El programa consta de tres semáforos para controlar la sincronización de las hebras:

- **ocupadas**

  Semáforo que controla la ocupación del buffer (número de entradas ocupadas), que es #E - #L (número de items insertados en el buffer menos número de items extraídos del buffer). Está inicializado en el valor 0, puesto que al principio no se ha realizado ninguna operación.

  Se usa sem_wait sobre este semáforo en la función *funcion_hebra_consumidora* antes de realizar la operación de lectura. Controla que no se lea ningún dato del buffer si no hay ningún dato almacenado en el buffer, por eso antes de leer asegura que el semáforo *libres* tenga un valor mayor que 0. El valor se irá decrementando tantas veces como casillas ocupadas queden.

  El sem_signal se ejecuta una vez que se haya realizado la operación de escritura en la función *funcion_hebra_productora*, incrementando el semáforo y así indicando que el item ya se ha escrito y por tanto hay una posición ocupada más en el buffer.

- **libres**

  Semáforo que controla la ocupación del buffer (número de entradas libres), que es k + #L - #E (tamaño del buffer más el número de items extraídos menos el número de items insertados). Está inicializado en el valor del tamaño del vector, puesto que al principio no se ha realizado ninguna operación y por tanto no se ha insertado ni extraído ningún valor del buffer.

  Se usa sem_wait sobre este semáforo en la función *funcion_hebra_productora* antes de realizar la operación de escritura. Controla que no se escriba ningún dato en el buffer si no hay espacio libre en este, por eso antes de escribir asegura que el semáforo *libres* tenga un valor mayor que 0. El valor se irá decrementando tantas veces como casillas libres queden.

  El sem_signal se ejecuta una vez que se haya realizado la operación de lectura en la función *funcion_hebra_consumidora*, incrementando el semáforo y así indicando que el item ya se ha leído y por tanto la posición que ocupaba vuelve a quedar libre.

- **mut**

  Semáforo que controla la exclusión mutua en la sección crítica que sucede cuando se escribe en el buffer y cuando se lee del buffer, ya que en la operación de inserción y extracción del vector puede haber indeterminación para algunas interfoliaciones debido a que se realizan varias instrucciones máquinas para una misma sentencia.

2. El problema de los fumadores

2.1. Código fuente de la solución adoptada

Código 2: fumadores.cpp

```
1
2 #include <iostream>
3 #include <cassert>
```

```
4  #include <thread>
5  #include <mutex>
6  #include <random>    // dispositivos, generadores y distribuciones aleatorias
7  #include <chrono>    // duraciones (duration), unidades de tiempo
8  #include "Semaphore.h"
9
10 using namespace std;
11 using namespace SEM;
12
13 //**************************************************************************
14 // variables compartidas
15 const int NUM_FUMADORES = 3;
16
17 // Semáforos
18 Semaphore ingr_disp[NUM_FUMADORES] = {0, 0, 0}; // Indica si el ingrediente está disponible
19 Semaphore mostr_vacio = 1;           // Indica si el mostrador está vacío (1) o no (0)
20 Semaphore mut = 1;                   // Semáforo para la exclusión mutua en los fumadores
21
22 //**************************************************************************
23 // plantilla de función para generar un entero aleatorio uniformemente
24 // distribuido entre dos valores enteros, ambos incluidos
25 // (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
26 //--------------------------------------------------------------------------
27 template< int min, int max > int aleatorio(){
28    static default_random_engine generador( (random_device())() );
29    static uniform_int_distribution<int> distribucion_uniforme( min, max );
30    return distribucion_uniforme( generador );
31 }
32
33 //--------------------------------------------------------------------------
34 // Función que simula la acción de fumar, como un retardo aleatoria de la hebra
35 void fumar(int num_fumador){
36
37    // calcular milisegundos aleatorios de duración de la acción de fumar
38    chrono::milliseconds duracion_fumar(aleatorio<20,200>());
39
40    // informa de que comienza a fumar
41    cout << "Fumador " << num_fumador << "  :"
42         << " empieza a fumar (" << duracion_fumar.count() << " milisegundos)"
43         << endl;
44
45    // espera bloqueada un tiempo igual a 'duracion_fumar' milisegundos
46    this_thread::sleep_for(duracion_fumar);
47
48    // informa de que ha terminado de fumar
49    cout << "Fumador " << num_fumador
50         << "  : termina de fumar, comienza espera de ingrediente." << endl;
51
52 }
53
54 //--------------------------------------------------------------------------
55 // función que ejecuta la hebra del estanquero
56 void funcion_hebra_estanquero(){
```

```
57      while(true){
58
59          // Retraso aleatorio y producción del ingrediente
60          this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
61          int i = int (aleatorio<0, 2>());
62
63          sem_wait(mostr_vacio);
64          cout << "Estanquero produce ingrediente " << i << endl;
65          sem_signal(ingr_disp[i]);
66
67      }
68  }
69
70  //----------------------------------------------------------------------
71  // función que ejecuta la hebra del fumador
72  void funcion_hebra_fumador(int num_fumador){
73      while(true){
74
75          sem_wait(mut);
76          // Sección crítica
77          cout << "Fumador " << num_fumador << " espera para fumar. "
78               << "Necesita el ingrediente " << num_fumador << endl;
79          sem_signal(mut);
80
81          sem_wait(ingr_disp[num_fumador]);
82          cout << "El fumador " << num_fumador << " retira su ingrediente" << endl;
83          sem_signal(mostr_vacio);
84
85          fumar(num_fumador);
86
87      }
88  }
89
90  //----------------------------------------------------------------------
91
92  int main(){
93
94      cout << "-------------------------------------------------------------" << endl
95           << "              Problema de los fumadores.                     " << endl
96           << "-------------------------------------------------------------" << endl
97           << flush;
98
99      // declarar hebras y ponerlas en marcha
100     thread hebra_estanquero(funcion_hebra_estanquero);
101
102     thread fumadores[NUM_FUMADORES];
103
104     for (int i = 0 ; i < NUM_FUMADORES ; i++)
105         fumadores[i] = thread(funcion_hebra_fumador, i);
106
107     hebra_estanquero.join();
108
109     for (int i = 0 ; i < NUM_FUMADORES ; i++)
```

```
110         fumadores[i].join();
111
112 }
```

2.2. Descripción de los semáforos

El programa consta de tres semáforos para controlar la sincronización de las hebras:

- **ingr_disp[NUM_FUMADORES]**

  (const int NUM_FUMADORES es una variable constante de números enteros para indicar cuántos fumadores habrá. En el caso de esta implementación hay tres fumadores).

  Vector de semáforos cuyo tamaño depende del número de fumadores (en concreto es un semáforo para cada fumador). Indica si el ingrediente i está disponible (1) para el fumador i o no (0). Está inicializado al valor 0 puesto que al principio no hay ningún ingrediente disponible (no se ha puesto ingrediente en el mostrador).

  Se usa sem_wait sobre este semáforo en la función *funcion_hebra_fumador* antes de mostrar el mensaje que indica que el fumador ha retirado su ingrediente. Controla que el número de veces que el fumador retire su ingrediente no pueda ser mayor que el número de veces que se ha producido el ingrediente. Una vez que está disponible el ingrediente, el valor se decrementará a 0, indicando que ya no está disponible (porque ya se ha cogido y puede empezar a fumar).

  El sem_signal se ejecuta una vez que se haya realizado el retraso aleatorio y la producción del ingrediente en la función *funcion_hebra_estanquero*, incrementando el semáforo y así indicando que el fumador i ya puede coger el ingrediente i (es decir, hace que el sem_wait de este semáforo deje de esperar).

- **mostr_vacio**

  Semáforo que controla si el mostrador está vacío (1) o no (0). Es decir, si el estanquero ha producido un ingrediente para que un fumador lo consuma. Está inicializado a 1 puesto que al principio del programa el estanquero (productor) aún no ha producido ningún item.

  Se usa sem_wait sobre este semáforo en la función *funcion_hebra_estanquero* antes de informar de que se ha producido el ingrediente i. Controla que no haya más de un ingrediente en el mostrador, cuando su valor es 1 (mostrador vacío) permite indicar que se ha producido el ingrediente y que pueda actuar el sem_signal de ingr_disp[i]. El valor se decrementará a 0 una vez que el mostrador deje de estar ocupado, indicando que vuelve a estar vacío.

  El sem_signal se ejecuta una vez que se haya consumido el ingrediente i del fumador i en la *funcion_hebra_fumador*, incrementando el semáforo a 1 y así indicando que el mostrador vuelve a estar vacío ya que se ha consumido el item.

- **mut**

Semáforo que controla la exclusión mutua en la sección crítica que sucede cuando se muestra por pantalla el mensaje del fumador i que está esperando i ingrediente, ya que al ejecutarse más de un fumador a la vez (en este caso tres), los mensajes por pantalla se solapan y no se pueden leer bien.