UNIVERSIDAD
DE GRANADA

2ºA - Grupo A1

Grado en Ingeniería Informática

Sistemas Concurrentes y Distribuidos Práctica 2 - Introducción a los monitores y casos prácticos en C++11

Autor:
Jose Luis Gallego Peña

11 de noviembre de 2018

Índice

1. Seminario: Introducción a los monitores en C++11

1.1. Encapsulamiento y exclusión mutua

En el resultado que da la compilación y ejecución del programa *monitor_em.cpp* tenemos los resultados de las funciones test_1, test_2 y test_3, en las que podemos observar que el valor obtenido es distinto del esperado en el caso del monitor sin exclusión mutua y que en los otros dos monitores (con EM) el valor obtenido coincide con el esperado.

```
→ ./monitor_em
Monitor contador (sin exclusión mutua):

  valor esperado == 20000
  valor obtenido == 13182

Monitor contador (EM con acceso directo a cerrojos):

  valor esperado == 20000
  valor obtenido == 20000

Monitor contador (EM con guardas de cerrojo):

  valor esperado == 20000
  valor obtenido == 20000
```

Figura 1: Ejecuciones monitor_em.cpp

Si quitamos el unlock de MContador2::incrementa, el programa se queda en estado de interbloqueo (ninguna hebra está ejecutando código pero ninguna puede entrar y todas esperan), ya que el cerrojo del monitor no ha sido desbloqueado, y nunca sale de la zona de exclusión mutua.

1.2. Monitores nativos tipo Señalar y Continuar (SC): Productor/Consumidor

1.2.1. Monitor Barrera1 (SC)

- La hebra que entra la última al método cita (la hebra señaladora) es siempre la primera en salir de dicho método porque es la que hace notify_one a las demás hebras, que quedan esperando en la cola, mientras que la hebra señaladora una vez que hace notify_one sigue con su ejecución y tiene el cerrojo hasta que sale del monitor (termina el método). Las hebras señaladas tienen que esperar en la cola hasta adquirir el cerrojo.
- El orden en el que las hebras señaladas logran entrar de nuevo al monitor no siempre coincide con el orden de salida de wait (se observa porque los números de orden de entrada no aparecen ordenados a la salida). Esto es debido a que notify_one despierta a cualquiera de

las hebras que estén esperando en una v.c., sin ninguna política concreta para seleccionar la hebra que reanuda la ejecución.

- El constructor de la clase no necesita ejecutarse en exclusión mutua ya que sólo se ejecuta una vez, antes de ejecutar ninguna hebra.
- Si usamos notify_all en lugar de notify_one no se observa ningún cambio significativo en la traza del programa, ya que hacer num_hebras veces el notify_one es lo mismo que hacer un solo notify_all.

1.2.2. Solución del Productor/Consumidor con monitores SC

Se ha modificado la implementación del Productor/Consumidor en el archivo prodcons1_sc.cpp para que el vector buffer sea FIFO. Esto supone que los primeros elementos que se introducen, son los primeros que se extraen, y por tanto se han necesitado crear dos nuevos atributos: **primera_ocupada**, que indica la posición donde se extrae del vector, y **num_celdas_ocupadas**, que indica el número total de celdas que se han ocupado. Por tanto, ahora los atributos que indican el índice de ocupadas y libres son independientes y solo realizan la función de índice, y la restante sirve para controlar que no se pase del número de celdas totales.

En la versión de múltiples productores y consumidores se deben de cambiar las sentencias if en extraer e insertar por bucles while (manteniendo la condición), ya que puede entrar otra hebra y liberar la guarda, haciendo por tanto que la hebra que esté esperando deje de esperar cuando no debe.

1.3. Monitores tipo Señalar y Espera Urgente (SU): Productor/Consumidor

1.3.1. Monitor Barrera2 (SU)

De la traza de la barrera parcial con semántica SU se deduce lo siguiente:

- Ahora el orden de salida de la cita coincide siempre con el orden de la entrada porque las hebras esperan en la cola de forma ordenada y sale siempre primero la que entró antes.
- Hasta que todas las hebras de un grupo han salido de la cita, ninguna otra hebra que no sea del grupo logra entrar porque estas hebras son las que hacen signal ya que no han entrado en el grupo, y por tanto se bloquean en cola de urgentes.

1.3.2. Solución del Productor/Consumidor con monitores SU

A diferencia de la versión con monitores SC, si usamos monitores SU no hará falta cambiar las sentencias if en extraer e insertar, usando estas mismas sentencias if el programa funciona ya que la guarda la controla el monitor hoare y por tanto la hebra señalada no tiene que competir con otras hebras para adquirir el cerrojo del monitor y reanudar la ejecución.

2. Práctica: Casos prácticos de monitores en C++11

2.1. El problema de los fumadores

Se implementa el mismo problema de los fumadores y el estanquero cuya solución con semáforos ya fue vista en la práctica 1, pero esta vez con monitores SU. El programa tiene las siguientes características:

- **Variables permanentes:**
  - mostr_vacio: De tipo bool. Es true cuando el mostrador está vacío, y false cuando en el mostrador hay un ingrediente. Sirve para indicar al estanquero que debe esperar hasta que se haya cogido el ingrediente y entonces pueda volver a poner otro.
  - ingrediente_mostrador: De tipo entero. Indica el ingrediente que está actualmente en el mostrador. Se usa para comprobar si el fumador debe esperar a su ingrediente, o el actual es el que le toca. Puede tomar valores desde 0 hasta NUM_FUMADORES - 1.
- **Colas condición:**
  - mostrador: De tipo CondVar. Cola donde espera el estanquero mientras el mostrador tiene un ingrediente (condición de espera: !mostr_vacio).
  - sin_ingrediente[NUM_FUMADORES]: Vector de variables de tipo CondVar. Cola donde espera cada uno de los fumadores en la posición que le toca cuando esperan a que se les de un ingrediente (condición de espera: ingrediente_mostrador == i, donde i es el fumador).
- **Pseudo-código de los tres procedimientos del monitor:**

```
1  procedure obtenerIngrediente(i : integer)
2  begin
3     if ingrediente_mostrador != i then   {si ingrediente no es el del fumador}
4        sin_ingrediente[i].wait();        {espera hasta tener su ingrediente}
5     mostr_vacio := true;                 {se coge el ingrediente}
6     {Mensaje que indica que el fumador ha retirado su ingrediente}
7     mostrador.signal();                  {nuevo ingrediente}
8  end
9
10 procedure ponerIngrediente(ingre : integer)
11 begin
12    ingrediente_mostrador := ingre;   {nuevo ingrediente en el mostrador}
13    mostr_vacio := false;             {el mostrador ya no está vacío}
14    {Mensaje que indica que el estanquero ha producido el ingrediente}
15    {el fumador ingre ya tiene ingrediente}
16    sin_ingrediente[ingrediente_mostrador].signal();
17 end
18
19 procedure esperarRecogidaIngrediente
20 begin
21    if (!mostr_vacio)       {si mostrador vacío}
```

```
22          mostrador.wait();    {El estanquero espera hasta que se consuma el ingrediente}
23  end
```

Un ejemplo de ejecución del programa:

```
➜ ./fumadores_su
------------------------------------------------------
            Problema de los fumadores.
------------------------------------------------------
Estanquero produce ingrediente 0
El fumador 0 retira su ingrediente
Fumador 0  : empieza a fumar (52 milisegundos)
Estanquero produce ingrediente 0
Fumador 0  : termina de fumar, comienza espera de ingrediente.
El fumador 0 retira su ingrediente
Fumador 0  : empieza a fumar (183 milisegundos)
Estanquero produce ingrediente 0
Fumador 0  : termina de fumar, comienza espera de ingrediente.
El fumador 0 retira su ingrediente
Fumador 0  : empieza a fumar (51 milisegundos)
Estanquero produce ingrediente 1
El fumador 1 retira su ingrediente
Fumador 1  : empieza a fumar (118 milisegundos)
Estanquero produce ingrediente 1
Fumador 0  : termina de fumar, comienza espera de ingrediente.
Fumador 1  : termina de fumar, comienza espera de ingrediente.
El fumador 1 retira su ingrediente
Fumador 1  : empieza a fumar (71 milisegundos)
Estanquero produce ingrediente 1
Fumador 1  : termina de fumar, comienza espera de ingrediente.
El fumador 1 retira su ingrediente
Fumador 1  : empieza a fumar (75 milisegundos)
Estanquero produce ingrediente 1
Fumador 1  : termina de fumar, comienza espera de ingrediente.
El fumador 1 retira su ingrediente
Fumador 1  : empieza a fumar (161 milisegundos)
Estanquero produce ingrediente 2
El fumador 2 retira su ingrediente
Fumador 2  : empieza a fumar (50 milisegundos)
Estanquero produce ingrediente 1
Fumador 2  : termina de fumar, comienza espera de ingrediente.
Fumador 1  : termina de fumar, comienza espera de ingrediente.
El fumador 1 retira su ingrediente
Fumador 1  : empieza a fumar (125 milisegundos)
Estanquero produce ingrediente 0
El fumador 0 retira su ingrediente
Fumador 0  : empieza a fumar (59 milisegundos)
Estanquero produce ingrediente 1
Fumador 0  : termina de fumar, comienza espera de ingrediente.
Fumador 1  : termina de fumar, comienza espera de ingrediente.
El fumador 1 retira su ingrediente
Fumador 1  : empieza a fumar (185 milisegundos)
Estanquero produce ingrediente 0
El fumador 0 retira su ingrediente
```

Figura 2: Parte de la traza de fumadores_su.cpp

2.2. El problema del barbero durmiente

Se implementa el problema del barbero durmiente cuya solución con monitores SU. El programa tiene las siguientes características:

- **Variables permanentes:**
  - cliente_actual: De tipo entero. Indica el cliente que está actualmente cortándose el pelo. Se usa para indicarlo por pantalla. Puede tomar valores desde 0 hasta NUM_CLIENTES - 1.
- **Colas condición:**
  - barbero: De tipo CondVar. Cola donde espera el barbero cuando no hay nadie en la sala de espera, es decir, cuando duerme. (condición de espera: sala_espera.empty()). Se le hace wait cuando la sala de espera esté vacía, y se le hace signal cuando hay un nuevo cliente listo para que le corten el pelo.
  - sala_espera: De tipo CondVar. Cola donde esperan todos los clientes mientras se le está cortando el pelo a otro. Al ser una cola, sale de ella para cortarse el pelo el primero que ha entrado (condición de espera: barbero.empty()). Se le hace wait cuando el barbero esté ocupado cortandole el pelo a otro cliente, y se le hace signal cuando el barbero se despierta y llama al siguiente cliente.
  - silla: De tipo CondVar. Cola donde espera el cliente mientras se le corta el pelo (condición de espera: !barbero.empty()). Se llama a wait cuando se le esté cortando el pelo a un cliente, y se llama a signal cuando se termina de pelar al cliente.
- **Pseudo-código de los tres procedimientos del monitor:**

```
1  procedure cortarPelo(i : integer)
2  begin
3     {Mensaje indicando que el cliente entra a la barbería}
4     if barbero.empty() then              {si ingrediente no es el del fumador}
5        {Mensaje que indica que el cliente está esperando}
6        sala_espera.wait();          {espera hasta tener su ingrediente}
7     else
8        {Mensaje indicando que el barbero se despierta}
9     cliente_actual := i;                 {Nuevo cliente se pela}
10    {Mensaje que indica que se le va a cortar el pelo al cliente}
11    barbero.signal();                    {Barbero se despierta si no lo está}
12    silla.signal();                      {Se sienta en la silla}
13    {Mensaje que indica que el cliente sale de la barberia y espera fuera}
14 end
15
16 procedure siguienteCliente
17 begin
18    if sala_espera.empty() then           {si la sala de espera está vacía}
19       {Mensaje indicando que el barbero duerme}
20       barbero.wait();                {Barbero duerme}
21    sala_espera.signal();             {Un nuevo cliente puede pelarse}
22 end
```

```
23
24 procedure finCliente
25 begin
26    {Mensaje que indica que el cliente_actual se termina de pelar}
27    silla.signal();                  {Se deja vacía la silla, nuevo cliente}
28 end
```

Un ejemplo de ejecución del programa:

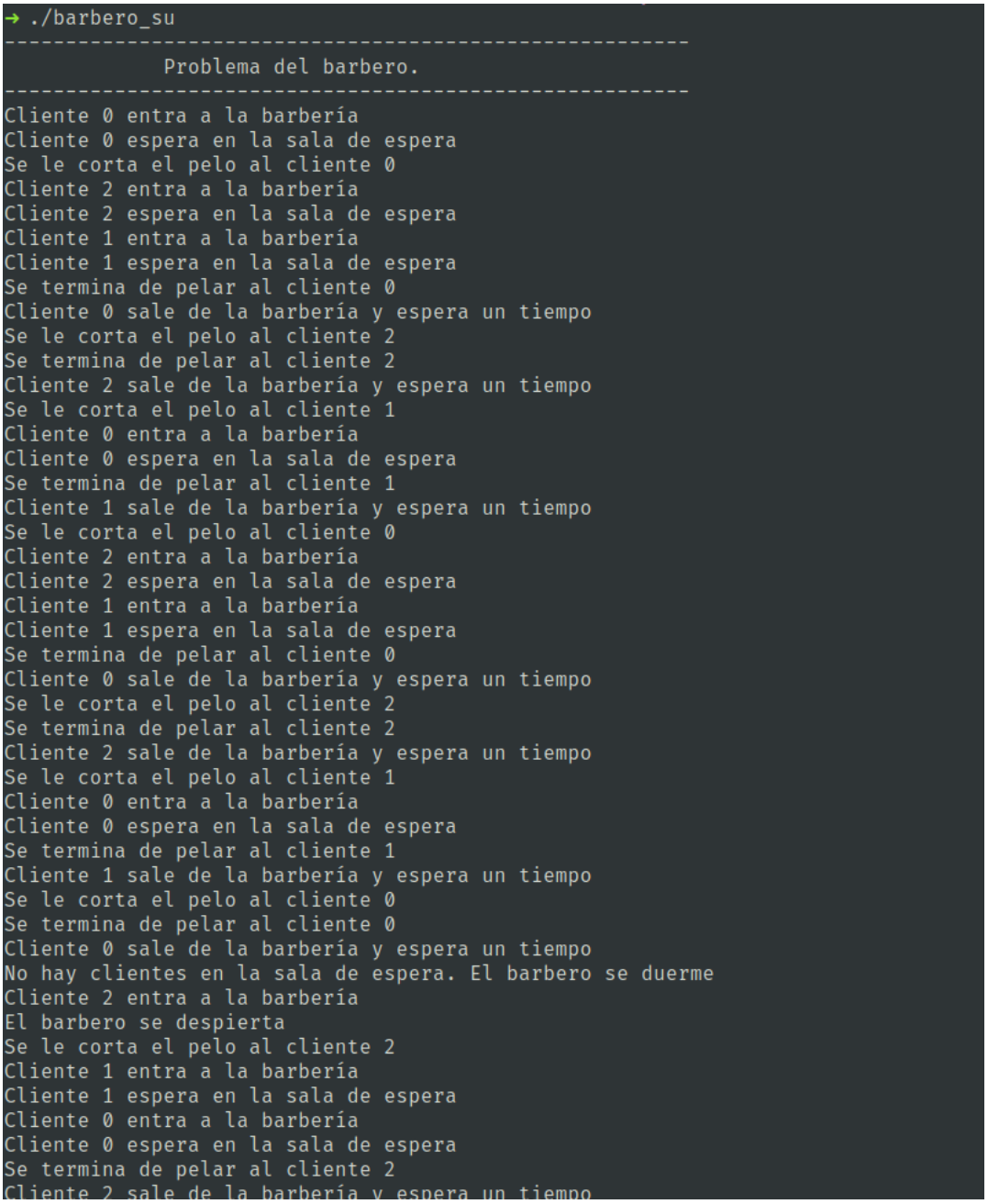

Figura 3: Parte de la traza de barbero_su.cpp