Postfolio-de-practicas-y-activid...

Anónimo

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

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

Practica 2 y actividades del seminario

Monitores

Contenido

KEEP CALM AND ESTUDIA UN POQUITO

WUOLAH + #QuédateEnCasa

Actividad 1: Productor/consumidor con semántica SC FIFO y LIFO

Para resolver el problema del productor/consumidor hemos usado monitores con semántica SC. Como explica en las transparencias, esta semántica cuando haces un signal en una cola para que siga un proceso que estaba esperando en esa cola el proceso que se desbloquea continua por donde estaba, pero después de haber esperado su turno en la cola de exclusión mutua del monitor. Esta es la principal diferencia con la otra semántica, la semántica SU, la cual como explicaremos más adelante se guarda en otra cola.

Por esta razón la condición por la esperaba se cumplió cuando se le hizo el signal pero mientras es pera en la cola de exclusión muta esta situación puede cambiar mientras esta en esta en esa cola. Para que esto no sea un problema hacemos las comprobaciones dentro de un bucle while en de un if (este último sí que se puede usar para la semántica SU porque no tiene esa problemática).

Esta problemática se presenta cuando hay varios consumidores y productores ejecutándose, y modificando esos if por bucles while se soluciona ya que cuando llega vuelve a comprobar si puede hacerlo y si no pues vuelve a esperar.

Para resolver el problema hemos añadido las siguientes variables a las existentes al sistema:

- LIFO:
  - *Ultima_ocupada:* Es un entero que sirve para saber dónde está la última posición del vector que está ocupada. Inicializada a 0.
  - Contador: Sirve para saber cuántas variables están ocupadas del vector por datos válidos. Inicializada a 0.
  - *Num_consumidoes:* Valor entero que dice cuántas hebras consumidoras hay.
  - *Num_productores*: Valor entero que dice cuántas hebras productoras hay.
- FIFO:
  - *Num_consumidoes*: Valor entero que dice cuántas hebras consumidoras hay.
  - *Num_productores*: Valor entero que dice cuántas hebras productoras hay.

Para resolver el problema con una cola LIFO lo único que hemos modificado es el if que compraba si podía consumir o producir por un bucle while dado la problemática anteriormente descrita.

Para resolver el problema con una cola FIFO hemos modificado carias cosas. La primera de ellas son las condiciones de los bucles while. Las cuales ahora son:

- Para los productores:

```
while (  contador == num_celdas_total)
```

Si el contador es igual número de celdas total significa que todas las posiciones del vector han sido ocupadas, por lo que hay que espera a que se quede libre alguna posición.

- Para los consumidores:

```
while ( contador == 0)
```

Si el contador es igual a 0 significa que no hay ningún valor que consumir por lo que hay que esperar a que se llene alguna casilla.

El siguiente cambio llega en cómo se escribe y se lee del buffer ya que ahora tenemos dos variables:

- Para los productores se cambia solo como como se va aumentado en la variable primera_libre, la cual se le suma uno y después se le hace el módulo para que no se pase del final del vector y vuelva al principio cuando llegue al final. Seguidamente se incremente el valor de contador.
- Para los consumidores cambiamos a ultima_ocupada para acceder al buffer. Y se va incrementando igual que primera_libre, sumándole 1 y haciéndole el módulo. Y por último se decrementando el contador.

Actividad 2: Productor/consumidor con semántica SU FIFO y LIFO

Ahora con el mismo problema que para la actividad anterior, productor/consumidor, usamos una semántica de monitores distinta, la semántica SU. Esta semántica se diferencia de la SC, en que cuando un proceso es sacado de la cola con un signal, este pasa a ejecución se termina de ejecutar haciendo asi que la condición por la que se desbloqueó no cambia y haya que comprobarla solo 1 vez. El proceso que hace el signal, pasa a una cola especial llamada “cola de urgentes”, la cual en cuanto termina de ejecutar el proceso que salió de la cola, se ejecuta, sin tener que esperar en la cola de exclusión mutua otra vez.

Los cambios más significativos con respecto a la solución anterior son en el monitor. Ya que este hereda de la clase HoareMonitor dada por los profesores y convirtiendo las variables de condición, en los tipos de datos ConVar, los cuales tienen que ser inicializados con la sentencia *“newCondVar()”* en el constructor del monitor. Aparte de ese solo está el cambio de los while que comprobaban la veracidad de que se podían o no escribir/leer del buffer por condicionales if en consecuencia de lo explicado anteriormente.

También han desaparecido las guardas ya que no hace falta ya que la clase HoareMonitor ya lo hace sola. Y ahora en vez de “notify_one” lo cambiamos por el clásico signal y el wait con la guarda por un wait a secas.

Y la última modificación la sufre es la declaración del monitor en el main la cual se quedaría tal que asi:

```
MRef<ProdCons1SC> ref =Create<ProdCons1SC>();
```

Por lo demás es igual que la semántica SC explicada en la actividad anterior.

Para ambas actividades hemos cambiado la función producir por la siguiente que complace a lo dicho en las diapositivas de que cada productor tiene que producir x elementos asociados al número de hebra:

```
int producir_dato2(int i){
   int contador =  vector_producidos[i]+ i *(num_items/num_productores);
   int salida= vector_producidos[i]+ i *(num_items/num_productores);
   mtx.lock();
   cout << "Producido: " << contador<<"----
"<< salida<< endl;// << flush ;
   mtx.unlock();
   this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
   vector_producidos[i]++;
   cont_prod[contador] ++ ;
   return salida;//contador ;
}
```

También cambiamos las funciones de las respectivas hebras dividiéndolas por el número de hebras para que solo hagan iteraciones necesarias:

```
void funcion_hebra_productora( ProdCons1SC * monitor, int n ){
   for( unsigned i = 0 ; i < num_items/num_productores ; i++ )
   {
      int valor = producir_dato2(n) ;
      monitor->escribir( valor );
   }
}
// -----------------------------------------------------------------------
-------

void funcion_hebra_consumidora( ProdCons1SC * monitor, int n ){
   for( unsigned i = 0 ; i < num_items/num_consumdidores ; i++ )
   {
      int valor = monitor->leer();
      consumir_dato( valor ) ;
   }
}
```

Y la función main para que cree las un numero de hebras productoras y consumidoras (excepto la línea de declaración del monitor en semántica SU dicho anteriormente):

```
int main(){
   cout << "-----------------------------------------------------" << endl
        << "Problema de los productores-
consumidores (1 prod/cons, Monitor SC, buffer FIFO). " << endl
        << "-----------------------------------------------------" << endl
        << flush ;
```

```
   ProdCons1SC monitor ;
   for(int i=0; i <num_productores;i++){
      vector_producidos[i] =0;
   }

   thread hebra_productora[num_productores];
   for (int i=0; i <num_productores;i++){
      hebra_productora[i] = thread( funcion_hebra_productora, &monitor, i
 );
   }
   thread  hebra_consumidora[num_consumdidores];
   for (int i=0; i <num_consumdidores;i++){
      hebra_consumidora[i] = thread ( funcion_hebra_consumidora, &monitor
, i );
   }

   for (int i=0; i <num_consumdidores;i++){
      hebra_consumidora[i].join();
   }
   for (int i=0; i <num_productores;i++){
      hebra_productora[i].join();
   }
   // comprobar que cada item se ha producido y consumido exactamente una
 vez
   test_contadores() ;
}
```

Practica 1: Problema de los fumadores

El problema de los fumadores es el mismo que el de la primera práctica, pero esta vez resuelto con monitores de semántica SU. Para ello hemos usado el siguiente monitor:

```
class Estanco : public HoareMonitor{
 private:
 int ingrediente;
 //mutex
   //cerrojo_monitor ;        // cerrojo del monitor
 CondVar        // colas condicion:
   estanquero,                //  cola donde espera el consumidor (n>0)
   fumadores[3];

 public:                    // constructor y métodos públicos
   Estanco( ) ;           // constructor
   void obtenerIngrediente(int i);               // obtiene el ingrdient
e del mostrador
   void ponerIngrediente(int ingr); // insertar un valor (sentencia E) (p
roductor)
```

```
    void esperarRecogidaIngrediente();
} ;
```

Las colas que hemos usado son:

- Estanquero: Variable de tipo ConVar que sirve para controlar las esperas del estanquero
- Fumadores[i]: Array que contempla una cola por cada ingrediente que se desea. Fumadores[a] representa la cola de espera para los fumadores que necesitan el ingrediente a.

Los métodos usados para el monitor son los dados en las transparencias. Los cuales son:

- ***obtenerIngrediente(int i)***

```
void Estanco::obtenerIngrediente(int i){
   if(ingrediente != i){
      fumadores[i].wait();
   }
   ingrediente= -1;
   estanquero.signal();
}
```

Este método sirve para coger el ingrediente del mostrador. Para ello si el ingrediente del mostrador no es el mismo que quiero (el que necesito está contenido en i), espero. Si es el mío pues pongo el mostrador a -1 y doy paso al estanquero para que genere un ingrediente ya que el mostrador está vacío.

- ***ponerIngredieente(int ingr)***

```
void Estanco::ponerIngrediente(int ingr){
   ingrediente= ingr;
   fumadores[ingr].signal();
}
```

Esta función la ejecuta únicamente la hebra estanquero y sirve para poner un ingrediente en el mostrador. Como solo hay un estanquero y un mostrador no hace falta poner ninguna condición de comprobación. Cuando pone el ingrediente le da paso a la cola del fumador con el ingrediente puesto en el mostrador.

- ***esperarRecogidaIngrediente()***

```
void Estanco::esperarRecogidaIngrediente(){
   if(ingrediente != -1){
      estanquero.wait();
   }
}
```

Este método funciona solo para el estanquero y sirve para esperar hasta que se coja el ingrediente.

Las funciones de la hebra estanquero y fumador son iguales a las dadas por las transparencias.

La del fumador, simplemente en un bucle infinito llama a la función del monitor *"obtenerIngrediente"* y a la función fumar.

La hebra del estanquero es algo más compleja que la del fumador. Primero genero un ingrediente. Después llamo a la función del monitor *"ponerIngrediente"*. Saco por pantalla el ingrediente puesto en el mostrador y seguidamente espero a que lo recojan llamando a la función *"esperarRecogidaIngrediente".* Todo ello lo hace también en un bucle infinito.

La salida ejecutándolo con 3 fumadores y 1 estanquero es:

```
-----------------------------------------------------------------------
Problema de los fumadores (3 fumadores, 1 estaquero, Monitor SU).
-----------------------------------------------------------------------
Fumando: 2
Puesto el ingrediente: 2
Fumando: 1
Puesto el ingrediente: 1
Fumando: 0
Puesto el ingrediente: 0
Puesto el ingrediente: 0
Fumando: 0
Fumando: 2
Puesto el ingrediente: 2
Puesto el ingrediente: 0
Fumando: 0
Puesto el ingrediente: 0
Fumando: 0
Fumando: 1
Puesto el ingrediente: 1
Fumando: 2
Puesto el ingrediente: 2
Puesto el ingrediente: 2
Fumando: 2
Puesto el ingrediente: 0
Fumando: 0
Puesto el ingrediente: 2
Fumando: 2
Puesto el ingrediente: 0
Fumando: 0
Puesto el ingrediente: 2
Fumando: 2
Puesto el ingrediente: 2
Fumando: 2
Fumando: 0
Puesto el ingrediente: 0
```

```
Puesto el ingrediente: 0
Fumando: 0
Fumando: 1
Puesto el ingrediente: 1
Puesto el ingrediente: 1
Fumando: 1
Puesto el ingrediente: 0
Fumando: 0
Fumando: 2
Puesto el ingrediente: 2
Puesto el ingrediente: 2
Fumando: 2
Puesto el ingrediente: 2
Fumando: 2
Fumando: 0
Puesto el ingrediente: 0
Puesto el ingrediente: 0
Fumando: 0
Fumando: 1
Puesto el ingrediente: 1
Fumando: 2
Puesto el ingrediente: 2
Puesto el ingrediente: 2
Fumando: 2
Puesto el ingrediente: 2
Fumando: 2
```

Practica 2: Problema del barbero

El problema del barbero es parecido al de los fumadores. Hay un barbero que corta el pelo en una silla, cuando llega un cliente si la silla esta libre se sienta y avisa el barbero para que le corte el pelo, si esta ocupada pues se espera en la sala de espera. Cuando termine de cortase el pelo, el cliente sale de la barbería. El barbero cuando termina de cortar el pelo a un cliente llama a alguien de la sala de espera si está vacía se duerme.

Para resolver este problema usaremos un monitor con semítica SU, el cual es:

```
class Barberia : public HoareMonitor{
private:
  bool silla;
  CondVar         // colas condicion:
      salaEspera, //  cola donde espera el consumidor (n>0)
      pelando,
      barbero;

public:                   // constructor y métodos públicos
  Barberia();             // constructor
```

```
  void cortarPelo(int i);  // obtiene el ingrdiente del mostrador
  void siguienteCliente(); // insertar un valor (sentencia E) (productor)
  void finCliente();
};
```

Para poder controlar las premisas anteriores dichas vamos ha usar las siguientes colas:

- salaEsera: Variable de tipo CondVar, que se usara principalmente como sala de espera en las que las hebras de los clientes esperaran cuando la silla este ocupada.
- Pelando: Variable de tipo CondVar. Se usará para que el cliente sentado en la silla espere a que le corten el pelo cuando esta sentado en la silla.
- Barbero: Variable de tipo CondVar. Esta cola servirá para que el barbero, si no hay nadie en la sala de espera ni esta pelando a nadie, se duerma hasta que llegue un cliente.
- Silla: Variable de tipo bool que estará a false o true cuando este la silla ocupada o no respectivamente.

Para poder hacer sus funciones el monitor tiene 3 funciones:

- ***cortarPelo(int i)***

```
void Barberia::cortarPelo(int i){
    if(silla){
        cout << "La pelu esta llena me espero "<< i << endl;
        salaEspera.wait();
    }
    cout << "\tMe siento en la pelu "<<i<< endl;
    silla=true;
    if(!barbero.empty()){
        barbero.signal();
    }
    cout << "Sentado en la silla el cliente "<< i<< endl;
    pelando.wait();
    cout << "\tMe voy de la pelu "<<i<< endl;
    silla=false;
}
```

Conta función que ejecuta el cliente, esta el groso del de la solución del problema. Para ello lo primero es comprobar si la silla esta ocupada si no pues me voy a la sala de espera. Después si la silla esta libre pues me siento (silla=true) y si el barbero esta dormido lo despierto. Seguidamente me espero a que terminen de pelarme(*pelando.wait*). Y cuando termine pues me levanto de la silla.

- ***siguienteCliente()***

```
void Barberia::siguienteCliente(){
    if(salaEspera.empty() && !silla){
```

```
        barbero.wait();
    }else{
        if(!silla){
            cout<< "Sieguinete cliente"<<endl;
            salaEspera.signal();
        }
    }
}
```

Esta función se encarga de mandar a dormir al barbero si la sala de espera y la silla esta vacía. Si no pues compruebo que la silla esta vacía, si es asi llamo a un cliente de la sala de espera (*salaEspera.signal*).

- ***finCliente()***

```
void Barberia::finCliente(){
    cout << "Fin del pelado"<< endl;
    pelando.signal();
}
```

Esta función es la mas sencilla de las tres, la ejecuta el barbero y solo avisa al que estaba sentado en la silla pelándose de que el barbero ha terminado de pelarle.

Para usar el monitor hemos hecho dos funciones para los dos tipos de hebras, la del barbero y la del cliente:

- ***Función cliente***

```
void funcion_Cliente(MRef<Barberia> monitor, int i){
  while (true){
    monitor->cortarPelo(i);
    EsperarFueraBarberia(i);
  }
}
```

La función que ejecuta el cliente se basa en un bucle infinito en la cual solo llama a la función del monito *"cortarPelo"*, y después cuando termina, llama a la función *"EsperarfueraBarberia"* que dormirá a la hebra un periodo aleatorio de tiempo antes de volver a entrar en la barbería.

- ***Función Barbero***

```
void funcion_Barbero(MRef<Barberia> monitor){
    while(true){
        monitor->siguienteCliente();
        CortarPeloACliente();
        monitor->finCliente();
    }
}
```

Esta función la ejecuta la hebra del Barbero. Para ello hace uso de dos funciones del monitor. Primero llama a *"siguienteCliente"* la cual ya hemos explicado arriba. Después llama la función *"CortarPeloACliente"* que simula el corte de pelo durante un periodo de tiempo aleatorio. Por último, llama a la función del monito *"finCliente"* la cual avisa al que está pelando de que ha terminado de pelarlo como hemos dicho anteriormente. Como esta dentro de un bucle infinito estos pasos los hace infinitas veces.