Práctica 2 SCD

Fumadores y Lectores-Escritores

Salvador Romero Cortés

Problema de los fumadores

En esta ocasión vamos a resolver el ejercicio de los fumadores y el estanquero con monitores de semántica SU (espera urgente). Esta implementación la hacemos en `C++` como una clase derivada de la clase `HoareMonitor`.

Tendremos 3 hebras fumadoras y una hebra estanquera que produce los ingredientes necesarios para que las hebras puedan fumar.

- Monitor

  ```
  class Estanco : public HoareMonitor{
  private:
     int mostrador;
     CondVar mostrador_vacio, esta_mio[num_fumadores];
  public:
     Estanco();
     void ponerIngrediente(int ingre);
     void esperarRecogida();
     void obtenerIngrediente(int i);
  };
  ```

  En el monitor tenemos un entero que indica que ingrediente hay en el mostrador o si este está vacío. En esta implementación se ha establecido como valor para indicar que está vacío el `-1`.

  Además dispone de dos variables condición para el mostrador y para cada hebra fumadora.

  Como métodos tenemos el constructor que inicializa la clase y una serie de métodos para manejar los ingredientes.

  - `void ponerIngrediente(int ingre)`

    ```
    void Estanco::ponerIngrediente(int ingre){
       mostrador = ingre;
       esta_mio[ingre].signal();
    }
    ```

    Este método lo llama la hebra estanquera y se encarga de poner un ingrediente en el mostrador y avisar a la hebra fumadora que depende de ese ingrediente de que está listo.

  - `void esperarRecogida()`

    ```
    void Estanco::esperarRecogida(){
       if (mostrador != -1) //comprobamos que no este vacio
          mostrador_vacio.wait();
    }
    ```

Esté método consiste en una espera en la cola de condición de la variable mostrador. Es ejecutada por la hebra estanquera y sirve para esperar a que recojan el ingrediente del mostrador.

- `void obtenerIngrediente(int i)`

```
void Estanco::obtenerIngrediente(int i){
   if (mostrador != i)
      esta_mio[i].wait();
   cout << "                  Fumador " << i << "recoge ingrediente" <<
endl;
   mostrador = -1;      //marcamos el mostrador como vacia
   mostrador_vacio.signal();
}
```

Este método lo ejecutan las hebras fumadoras y se basa en esperar a recibir la señal de que el ingrediente necesario está disponible para recoger. Cuando se recoge se modifica el valor del mostrador (`mostrador = -1`) y se avisa a la cola condición del mostrador.

- Funciones hebras
  - `void funcion_hebra_fumadora(MRef<Estanco> monitor, int indice)`

```
void funcion_hebra_fumadora(MRef<Estanco> monitor, int indice){
   while (true){
      monitor->obtenerIngrediente(indice);
      fumar(indice);
   }
}
```

Se encarga de recoger ingredientes y fumar.

Función fumar:

```
void fumar(int indice)
{
   // mutex para que la salida no se mezcle
   // comentado para ver varios fumadores simultaneos
   //unique_lock<mutex> guarda(mtx);
   cout << "                  Fumador " << indice << " empieza a fumar"
<< endl;
   this_thread::sleep_for(chrono::milliseconds(aleatorio<500, 700>()));
   cout << "                  Fumador " << indice << " deja de fumar" <<
endl;

}
```

Se trata de una espera aleatoria para simular que fuma. Como se puede ver en los comentarios se comenta un mutex para que luego en salida se vean como varios fumadores fuman a la vez. Si se descomenta la salida se ve ordenada pero es más difícil ver la concurrencia de los fumadores.

  - `void funcion_hebra_estanquera(MRef<Estanco> monitor)`

```
void funcion_hebra_estanquera(MRef<Estanco> monitor){
   while (true){
      int ingrediente = producirIngrediente();
      monitor->ponerIngrediente(ingrediente);
      mtx.lock();
      cout << "Estanquero ha puesto ingrediente y espera que se recoja"
<< endl;
      mtx.unlock();
      monitor->esperarRecogida();
   }
}
```

Se encarga de producir ingredientes y colocarlos en el mostrador. Tras esto, espera a que se recojan los ingredientes.

Función para producir los ingredientes

```
int producirIngrediente(){
   //this_thread::sleep_for(chrono::milliseconds(aleatorio<20, 100>()));
   int ingrediente = aleatorio<0,num_fumadores-1>();
   unique_lock<mutex> guarda(mtx);
   cout << "Estanquera produce ingrediente: " << ingrediente << endl;
   return ingrediente;
}
```

Se trata de una espera aleatoria para simular la producción de ingredientes. El tiempo de espera está comentado para que en la salida se vea mejor la concurrencia de fumadores.

Capturas de pantalla de los resultados:

```
Estanquera produce ingrediente: 1
                  Fumador 2 empieza a fumar
                  Fumador 1recoge ingrediente
                  Fumador 1 empieza a fumar
Estanquero ha puesto ingrediente y espera que se recoja
Estanquera produce ingrediente: 1
Estanquero ha puesto ingrediente y espera que se recoja
                  Fumador 2 deja de fumar
                  Fumador 1 deja de fumar
                  Fumador 1recoge ingrediente
Estanquera produce ingrediente: 0                   Fumador
1 empieza a fumar
                  Fumador 0recoge ingrediente
                  Fumador 0 empieza a fumar
Estanquero ha puesto ingrediente y espera que se recoja
Estanquera produce ingrediente: 1
Estanquero ha puesto ingrediente y espera que se recoja
                  Fumador 1 deja de fumar
                  Fumador 1recoge ingrediente
Estanquera produce ingrediente: 0
Estanquero ha puesto ingrediente y espera que se recoja
                  Fumador 1 empieza a fumar
                  Fumador 0 deja de fumar
                  Fumador 0recoge ingrediente
Estanquera produce ingrediente: 1
Estanquero ha puesto ingrediente y espera que se recoja
                  Fumador 0 empieza a fumar
                  Fumador 1 deja de fumar
                  Fumador 1recoge ingrediente
Estanquera produce ingrediente: 2
                  Fumador 1 empieza a fumar
                  Fumador 2recoge ingrediente
                  Fumador 2 empieza a fumar
Estanquero ha puesto ingrediente y espera que se recoja
Estanquera produce ingrediente: 1
Estanquero ha puesto ingrediente y espera que se recoja
                  Fumador 0 deja de fumar
                  Fumador 1 deja de fumar
                  Fumador 1recoge ingrediente
Estanquera produce ingrediente: 0
                  Fumador 1 empieza a fumar
                  Fumador 0recoge ingrediente
                  Fumador 0 empieza a fumar
Estanquero ha puesto ingrediente y espera que se recoja
Estanquera produce ingrediente: 2
Estanquero ha puesto ingrediente y espera que se recoja
                  Fumador 2 deja de fumar
                  Fumador 2recoge ingrediente
Estanquera produce ingrediente: 1
```

Vemos como varios fumadores pueden fumar simultáneamente.

Problema de Lectores-Escritores

En este problema tenemos varias hebras lectoras y otras escritoras. Estas hebras acceden a una estructura de datos que solo puede ser modificada por una hebra a la vez. Sin embargo, sí que se puede leer de manera simultánea. Por ello pueden haber varias hebras lectoras accediendo a la estructura al mismo tiempo, pero solo puede haber una escritora si no hay ningún otra escritora ni lectora.

Resolvemos el problema con un monitor SU de `HoareMonitor`

En esta implementación le hemos dado prioridad a los lectores.

- Monitor

  ```
  class Lec_Esc : public HoareMonitor{
  private:
     bool escrib; //indica si un escritor esta escribiendo
     int n_lec;       //número de lectores simultaneos
     CondVar lectura,escritura;
  public:
     Lec_Esc();
     void ini_lectura();
     void fin_lectura();
     void ini_escritura();
     void fin_escritura();
  };
  ```

  Tenemos el entero `n_lec` que indica el número de lectores que hay actualmente en el monitor, la variable booleana `escrib` que indica si hay algún escritor en la estructura de datos. También disponemos de las variables condición `escritura` y `lectura` que se encargan de las colas de escritura/lectura.

  Como métodos tenemos el constructor, que inicializa el monitor, y un para de métodos para cada operación de lectura o escritura.

  - `void ini_lectura()`

    ```
    void Lec_Esc::ini_lectura(){
       if (escrib)
          lectura.wait();
       n_lec += 1;
       lectura.signal();
    }
    ```

    Este método lo ejecutan las hebras lectoras. Esperan si hay algún escritor en la estructura. Cuando salen de la cola condición aumentan el número de lectores simultáneos y avisa a otro posible lector de que puede entrar.

  - `void fin_lectura()`

```
void Lec_Esc::fin_lectura(){
   n_lec -= 1;
   if (n_lec == 0)
      escritura.signal();
}
```

Este método disminuye en 1 la variable contadora de lectores y en caso de que sea cero avisa a la variable condición de los escritores para que pueda entrar alguno.

- `void ini_escritura()`

```
void Lec_Esc::ini_escritura(){
   if (n_lec > 0 || escrib)
      escritura.wait();
   escrib = true;
}
```

Este método lo ejecutan las hebras escritoras. Esperan hasta que no haya lectores ni otros escritores en la estructura de datos. Cuando consiguen entrar, ponen a `true` la variable condición.

- `void fin_escritura()`

```
void Lec_Esc::fin_escritura(){
   escrib = false;
   if (lectura.empty())
      escritura.signal();
   else
      lectura.signal();
}
```

Este método lo ejecutan las hebras escritoras cuando acaban de escribir. Marcan como `false` la variable condición de los escritores y si no hay ningún lector avisa a otro posible escritor. En caso contrario avisa a los lectores.

- Funciones hebras:
  - `void funcion_hebra_lectora(MRef<Lec_Esc> monitor, int indice)`

```
void funcion_hebra_lectora(MRef<Lec_Esc> monitor, int indice)
{
   while (true){
      monitor->ini_lectura();
      leer(indice);
      monitor->fin_lectura();
      this_thread::sleep_for(chrono::milliseconds(aleatorio<20, 200>
()));
   }
}
```

Esta función se basa en utilizar los métodos proporcionados para la lectura por el monitor antes y después de leer. Cuando acaba de leer se produce una espera aleatoria para poder dar oportunidad a los escritores.

  - `void funcion_hebra_escritora(MRef<Lec_Esc> monitor, int indice)`

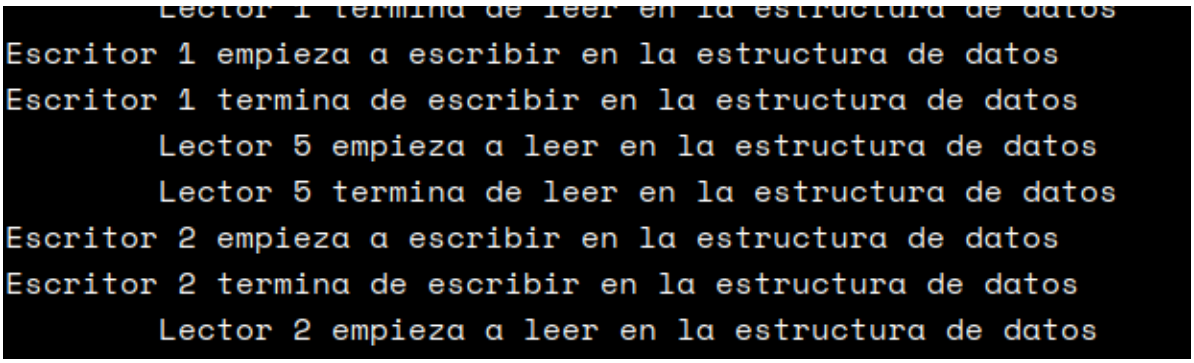
```
void funcion_hebra_escritora(MRef<Lec_Esc> monitor, int indice)
{
   while (true){
      monitor->ini_escritura();
      escribir(indice);
      monitor->fin_escritura();
      this_thread::sleep_for(chrono::milliseconds(aleatorio<20, 200>
()));
   }
}
```

Esta función es completamente análoga a la de la hebra lectora.

Las funciones de leer y escribir son simples esperas aleatorias.

Captura de ejecución:

```
Escritor 0 empieza a escribir en la estructura de datos
Escritor 0 termina de escribir en la estructura de datos
        Lector 5 empieza a leer en la estructura de datos
        Lector 0 empieza a leer en la estructura de datos
        Lector 2 empieza a leer en la estructura de datos
        Lector 4 empieza a leer en la estructura de datos
        Lector 3 empieza a leer en la estructura de datos
        Lector 1 empieza a leer en la estructura de datos
        Lector 2 termina de leer en la estructura de datos
        Lector 0 termina de leer en la estructura de datos
        Lector 1 termina de leer en la estructura de datos
        Lector 4 termina de leer en la estructura de datos
        Lector 5 termina de leer en la estructura de datos
        Lector 0 empieza a leer en la estructura de datos
        Lector 2 empieza a leer en la estructura de datos
        Lector 3 termina de leer en la estructura de datos
        Lector 2 termina de leer en la estructura de datos
        Lector 5 empieza a leer en la estructura de datos
        Lector 1 empieza a leer en la estructura de datos
        Lector 4 empieza a leer en la estructura de datos
        Lector 2 empieza a leer en la estructura de datos
        Lector 0 termina de leer en la estructura de datos
        Lector 5 termina de leer en la estructura de datos
        Lector 1 termina de leer en la estructura de datos
        Lector 3 empieza a leer en la estructura de datos
        Lector 5 empieza a leer en la estructura de datos
        Lector 3 termina de leer en la estructura de datos
        Lector 0 empieza a leer en la estructura de datos
        Lector 2 termina de leer en la estructura de datos
        Lector 1 empieza a leer en la estructura de datos
        Lector 0 termina de leer en la estructura de datos
        Lector 4 termina de leer en la estructura de datos
        Lector 1 termina de leer en la estructura de datos
        Lector 2 empieza a leer en la estructura de datos
        Lector 3 empieza a leer en la estructura de datos
        Lector 5 termina de leer en la estructura de datos
        Lector 1 empieza a leer en la estructura de datos
        Lector 2 termina de leer en la estructura de datos
        Lector 0 empieza a leer en la estructura de datos
        Lector 4 empieza a leer en la estructura de datos
        Lector 3 termina de leer en la estructura de datos
        Lector 4 termina de leer en la estructura de datos
        Lector 2 empieza a leer en la estructura de datos
        Lector 0 termina de leer en la estructura de datos
        Lector 2 termina de leer en la estructura de datos
        Lector 4 termina de leer en la estructura de datos
```



Vemos que la ejecución es correcta.