Práctica 2

Pablo Linari Pérez

Noviembre 2024

Contents

1 Introducción

En esta práctica se van a realizar dos implementaciones de tres problemas de sincronización usando librerías abiertas para programación multihebra y monitores, y así conocer el problema del *productor-consumidor* y gestionar la sincronización de varias hebras.

2 Productor consumidor FIFO

En esta sección veremos la implementación de el problema del productor consumidor FIFO , veamos la implementación de los métodos de la clase monitor

Listing 1: Función leer productor consumidor

```
int ProdConsSU1::leer(  )
{
   // esperar bloqueado hasta que 0 < primera_libre
   if ( primera_libre == primera_ocupada )
      ocupadas.wait();

   //cout << "leer: ocup == " << primera_libre << ", total
       == " << num_celdas_total << endl ;
   //assert( primera_libre == primera_ocupada);

   // hacer la operaci n de lectura, actualizando estado
       del monitor
   const int valor = buffer[primera_ocupada] ;
   primera_ocupada = (primera_ocupada+1 )%num_celdas_total;
    n--;

   // se alar al productor que hay un hueco libre, por si
       est  esperando
   libres.signal();

   // devolver valor
   return valor ;
}
```

Este método del monitor usado en este programa consiste en comprobar que el búfer de lectura tenga espacio. Si no es así, se debe esperar; de lo contrario, se continúa y se lee el valor del búfer en la posición de la primera unidad ocupada. Esta variable se incrementa en 1 unidad, teniendo en cuenta la longitud del búfer, y se decrementa la variable n, la cual nos indicará en el método de escritura si el búfer está lleno o no.

Listing 2: Función escribir productor consumidor

```
void ProdConsSU1::escribir( int valor )
{
```

```
3     // esperar bloqueado hasta que primera_libre <
          num_celdas_total
4     if ( n==num_celdas_total-1)
5        libres.wait();
6
7     //cout << "escribir: ocup == " << primera_libre << ",
          total == " << num_celdas_total << endl ;
8     //assert(primera_libre==-1);
9
10    // hacer la operaci n de inserci n, actualizando estado
          del monitor
11    buffer[primera_libre] = valor ;
12    primera_libre=(primera_libre+1) %num_celdas_total;
13     n++;
14
15    // se alar al consumidor que ya hay una celda ocupada (
          por si esta esperando)
16    ocupadas.signal();
17 }
```

Este método se encarga de escribir en el búfer. Para ello, primero comprueba que haya espacio en el búfer, haciendo uso de la variable `n`, que indica la cantidad de celdas ocupadas en el búfer. Si el búfer está lleno, la cola de libres deberá esperar. Cuando haya espacio, se escribe en el búfer y se incrementa la variable `primera_libre`, teniendo en cuenta el tamaño del búfer, y se envía una señal a "ocupadas" para que puedan ejecutarse. Podemos observar que la variable `n` se incrementa en una unidad; esta variable es la que hemos mencionado antes y que utilizaremos para saber cuántas celdas están ocupadas en cada momento.

3 Múltiples productores y consumidores

3.1 FIFO

Primero analizaremos la implementación FIFO, la cual es muy parecida a la del programa anterior, pero teniendo en cuenta que ahora hay múltiples consumidores y productores.

Los métodos de escritura y lectura funcionan de la misma manera; la única diferencia la encontramos en las funciones `funcion_hebra_productora` y `funcion_hebra_consumidora`, donde tenemos que considerar el número de hebras disponibles.

Listing 3: función hebra productora

```
1  void funcion_hebra_productora( MRef<ProdConsSU1> monitor ,
       int i)
2  {
3     for( unsigned j = p*i; j < p*i +p ; j++ )
4     {
5        int valor = producir_dato(i) ;
6        monitor->escribir( valor );
```

```
   }
}
```

En `funcion_hebra_productora` vemos que tenemos como argumentos el monitor y un entero `i` que hace referencia al identificador de la hebra que ejecuta la función. El código consiste en un bucle `for` que cada hebra ejecutará desde `p*i` hasta `p*i + p`, siendo `p = num_items / NUMPRODUCTORAS`. Esto hace que cada hebra productora produzca una única vez un conjunto de números enteros. Dentro de este `for`, las hebras producirán un dato y ejecutarán la función `escribir` con ese valor generado.

Listing 4: función producir dato para varias hebras

```
unsigned producir_dato(int i)
{
   this_thread::sleep_for( chrono::milliseconds( aleatorio
       <20,100>() ));
    int dato_producido = i*p + producidos[i];
   producidos[i]+=1;
   cont_prod[dato_producido]++;
   cout <<"hebra␣:"<<i<< "producido:" << dato_producido <<
       endl << flush ;
   return dato_producido ;
}
```

En la función *producir_dato* se añade el índice *i* para saber qué dato generar, ya que cada hebra productora *i* debe generar *i * p + p - 1* datos. Para llevar un registro de qué dato debe generar cada hebra en cada momento, cada hebra guarda en una posición del vector el número *n* que ya ha generado, de modo que, la próxima vez, se genere el número *n + 1*.

Listing 5: función hebra consumidora multiple

```

void funcion_hebra_consumidora( MRef<ProdConsSU1>  monitor )
{
   for( unsigned i = 0 ; i < c ; i++ )
   {
      int valor = monitor->leer();
      consumir_dato( valor ) ;
   }
}
```

La función *hebra_productora* consiste en un bucle `for` que se repite `c` veces, siendo `c` el valor de `num_items / NUMCONSUMIDORAS`. Dentro de este bucle se ejecuta la función `leer` del objeto monitor. Este `for` limita a cada hebra a un número `c` de consumiciones, de modo que todas tengan la misma carga.

3.2 LIFO

Veamos ahora la implementación LIFO. A diferencia de la primera práctica, las funciones que ejecutan las hebras tanto de consumo como de producción se mantienen igual, ya que lo único que cambia es la implementación en la clase del monitor de los métodos `escribir` y `leer`.

Listing 6: función leer monitor LIFO

```
1
2
3  int ProdConsSU1::leer(  )
4  {
5     // esperar bloqueado hasta que 0 < primera_libre
6     if ( primera_libre == 0 )
7        ocupadas.wait();
8
9     //cout << "leer: ocup == " << primera_libre << ", total
          == " << num_celdas_total << endl ;
10    assert( 0 < primera_libre  );
11
12    // hacer la operaci n de lectura, actualizando estado
          del monitor
13    primera_libre-- ;
14    const int valor = buffer[primera_libre] ;
15
16    // se alar al productor que hay un hueco libre, por si
          est  esperando
17    libres.signal();
18
19    // devolver valor
20    return valor ;
21 }
```

La implementación del método no varía demasiado respecto a su versión FIFO; lo único que se debe cambiar es la variable que indica en qué casilla del búfer leer. Para ello, primero comprobamos que esta variable no tenga valor 0, de manera que no se intente leer antes de que se haya escrito. Si el valor es 0, la hebra tendrá que esperar; si el valor es mayor que 0, decrementamos en una unidad `primera_libre` y leemos el búfer en la posición indicada. A continuación, enviamos una señal a `libres` y devolvemos el valor leído.

Listing 7: función escribir monitor LIFO

```
1
2
3  void ProdConsSU1::escribir( int valor )
4  {
5     // esperar bloqueado hasta que primera_libre <
          num_celdas_total
6     if ( primera_libre == num_celdas_total )
7        libres.wait();
8
9     //cout << "escribir: ocup == " << primera_libre << ",
          total == " << num_celdas_total << endl ;
10    assert( primera_libre < num_celdas_total );
11
12    // hacer la operaci n de inserci n, actualizando estado
          del monitor
13    buffer[primera_libre] = valor ;
14    primera_libre++ ;
15
16    // se alar al consumidor que ya hay una celda ocupada (
          por si esta esperando)
17    ocupadas.signal();
18 }
```

El método `escribir` sigue un esquema similar al de `leer`. Primero, comprueba si el búfer está libre; si lo está, inserta el valor pasado como argumento en el búfer, incrementa el valor de `primera_libre` y envía una señal a la variable de condición `ocupadas`. De lo contrario, si

Listing 8: función main multiproductor consumidor (FIFO y LIFO)

```
1
2  int main()
3  {
4      thread consumidoras[HEBRASCONSUMIDORAS];
5      thread productoras[HEBRASPRODUCTORAS];
6
7
8     for (int i= 0; i< HEBRASPRODUCTORAS; i++) {
9      productoras[i]=thread(funcion_hebra_productora,i);
10    }
11     for (int i= 0; i< HEBRASCONSUMIDORAS; i++) {
12         consumidoras[i]=thread(funcion_hebra_consumidora);
13    }
14    for (int i= 0; i< HEBRASPRODUCTORAS; i++) {
15     productoras[i].join();
16    }
17     for (int i= 0; i< HEBRASCONSUMIDORAS; i++) {
18         consumidoras[i].join();
19    }
```

```
20
21
22      test_contadores();
23  }
```

La función *main* es común a ambos programas (FIFO y LIFO). En ella encontramos dos vectores, uno de hebras productoras y otro de hebras consumidoras, los cuales se inician cada uno en su propio bucle *for*, ejecutando en cada uno su respectiva función.

Los últimos dos bucles *for* se encargan de hacer un *join* en todas las hebras de cada vector para que, una vez finalicen todas, se pase a ejecutar la función *test_contadores*.

4 Fumadores

Pasemos a ver el problema de los fumadores, donde una hebra hace de estanquero y tres hebras hacen de fumadores. Cada fumador necesita un ingrediente para poder fumar (cerillas, tabaco o papel). Identificando cada ingrediente con un índice i, podemos asignar, por ejemplo, el fumador 0 con el ingrediente 0, el fumador 1 con el ingrediente 1, y así sucesivamente.

Para sincronizar a los fumadores con el estanquero, debemos usar dos variables de condición en la clase del monitor. La variable `mostrador` se encargará de indicar cuándo el mostrador está libre, y la variable `fumador` será un array de variables de condición, el cual nos indicará qué fumador tiene que esperar y cuál puede fumar.

Listing 9: función poner$_i$*ngrediente*

```
1
2   void Estanco::poner_ingrediente(int i){
3       cout<<"puesto␣el␣ingrediente␣"<< i <<endl;
4       if(!fumador[i].empty()){
5           fumador[i].signal();
6       }
7   }
```

La función `poner_ingrediente` se encarga de colocar el ingrediente en el mostrador e indicar al fumador correspondiente que ya puede consumirlo. Para ello, primero comprobamos si el fumador está esperando, y en ese caso le damos la señal para que recoja el ingrediente.

Listing 10: función obtener$_i$*ngrediente*

```
1   void Estanco::obtener_ingrediente(int i){
2      if(ingrediente!=i){
3           fumador[i].wait();
4       }
5       ingrediente =-1;
```

```
6       cout << "fumador␣"<<i<<"␣coge␣el␣ingrediente␣" << i <<
            endl;
7       mostrador.signal();
8   }
```

En primer lugar, la función comprueba si el ingrediente que se encuentra en el mostrador se corresponde con el fumador que ejecuta la hebra. Si no es así, el fumador debe esperar; en caso contrario, el fumador puede coger el ingrediente, y se envía una señal al `mostrador` para notificar que ya se ha cogido el ingrediente y que ha quedado vacío.

Listing 11: función recogida$_i$*ngrediente*

```
1   void Estanco::esperar_recogida_ingrediente(){
2      if(ingrediente >=0){
3           mostrador.wait();
4       }
5   }
```

Por último, la función `esperar_recogida_ingrediente` consiste en esperar si el ingrediente sigue en el mostrador; es decir, si el ingrediente es mayor o igual a 0, ya que en la función anterior la variable `ingrediente` pasa a valer -1 cuando se retira el ingrediente.

Listing 12: función fumadores

```
1   void funcion_hebra_fumador(int num_fumador,MRef<Estanco>
        monitor){
2       while(true){
3           monitor->obtener_ingrediente(num_fumador);
4           fumar(num_fumador);
5       }
6   }
```

Las hebras fumadoras ejecutan una función que consiste en un bucle `while` infinito en el cual se ejecuta el método `obtener_ingrediente` y, acto seguido, la función `fumar`.

Listing 13: función estanquero

```
1   void funcion_hebra_estanquero(MRef<Estanco>monitor){
2       int ingrediente;
3       while (true) {
4           ingrediente = producir_ingrediente();
5           monitor->poner_ingrediente(ingrediente);
6           monitor->esperar_recogida_ingrediente();
7       }
8   }
```

Por parte de la función del estanquero, esta consiste en ejecutar un bucle infinito en el cual produce un ingrediente y ejecuta los métodos `poner_ingrediente` y `esperar_recogida_ingrediente`.

5 Lectores y escritores

En este problema, dos tipos de procesos acceden concurrentemente a datos compartidos: los escritores, que no deben ejecutarse concurrentemente con ningún otro escritor o lector, y los lectores, que no modifican la estructura de datos, pero que pueden ejecutarse concurrentemente con otros lectores de forma arbitraria.

A continuación, veremos cómo se han implementado los métodos de la clase del monitor que hemos utilizado para resolver el problema.

Para este monitor usaremos dos variables de condición que nos indicarán cuándo debemos leer o escribir, una variable booleana para indicar si se está escribiendo o no, y un entero que nos indica el número de lectores que están leyendo en un momento dado.

Listing 14: función inicio de lectura

```
void lectorescritor::ini_lectura(int i){
    if(escrib){
        lectura.wait();
    }
        cout<<"Hebra␣"<<i<<"␣empieza␣a␣leer"<<endl;
    n_lec++;
    lectura.signal();
}
```

En esta función, primero comprobamos si se está escribiendo. Si esto ocurre, hacemos `wait` a la variable `lectura`, ya que no podemos leer mientras se escribe. Si no se está escribiendo, la hebra puede comenzar a leer y se incrementa en una unidad el número de lectores. También se envía una señal a la variable `lectura` para indicar que, si hay otro proceso esperando, puede leer.

Listing 15: función fin de lectura

```
void lectorescritor::fin_lectura(int i){
    cout<<"Hebra␣"<<i<<"␣␣termina␣de␣leer"<<endl;
    n_lec--;
    if(n_lec ==0){
        escritura.signal();
    }
}
```

La función `fin_de_lectura` se encarga de que las hebras terminen de leer. Decrementa en una unidad la cantidad de lectores, y cuando el número de lectores sea 0, envía una señal a la variable `escritura` para indicar que no hay nadie leyendo y, por tanto, se puede escribir.

Listing 16: función inicio de escritura

```

void lectorescritor::ini_escritura(int i){
    if(n_lec>0 || escrib){
```

```
        escritura.wait();
    }
    cout<<"Hebra␣"<<i<<"␣empieza␣a␣escribir"<<endl;
    escrib=true;
}
```

Esta función se encarga de iniciar la escritura de la hebra. Primero, comprueba si hay lectores o si la variable `escrib` es verdadera en caso de que ya haya un escritor escribiendo. En este caso, el proceso escritor deberá esperar; de lo contrario, podrá escribir, por lo tanto, comienza a escribir y asigna el valor `true` a la variable `escrib`.

Listing 17: función fin de escritura

```
void lectorescritor::fin_escritura(int i){
    cout<<"Hebra␣"<<i<<"␣termina␣de␣escribir"<<endl;
    escrib=false;
    if(!lectura.empty()){
        lectura.signal();
    }else{
        escritura.signal();
    }
}
```

Esta función se encarga de finalizar la escritura. Para ello, asigna a la variable `escrib` el valor `false` y comprueba si hay lectores. En caso de haberlos, envía una señal a los lectores mediante la variable de condición `lectura`; en caso contrario, envía una señal a la variable de condición `escritura` para indicar que se puede volver a escribir.

Listing 18: función lector

```
void lector(MRef<lectorescritor>monitor,int i){
    while(true){
        monitor->ini_lectura(i);
        this_thread::sleep_for( chrono::milliseconds(
            aleatorio<min_ms,max_ms>() ));
        monitor->fin_lectura(i);
   this_thread::sleep_for( chrono::milliseconds( aleatorio<
      min_ms,max_ms>() ));
    }

}
```

Listing 19: función escritor

```
void  escritor(MRef<lectorescritor>monitor,int i){
    while(true){
        monitor->ini_escritura(i);
   this_thread::sleep_for( chrono::milliseconds( aleatorio<
      min_ms,max_ms>() ));
```

```
5
6           monitor->fin_escritura(i);
7       this_thread::sleep_for( chrono::milliseconds( aleatorio<
            min_ms,max_ms>() ));
8       }
9
10  }
```

Como podemos observar, las funciones de lector y escritor son muy parecidas; ambas tienen la misma estructura: un bucle `while` infinito que ejecuta los métodos del monitor correspondientes en cada caso. El lector ejecuta `ini_lectura` y `fin_lectura`, separados por tiempos aleatorios, y en el caso del escritor se procede ig

6 Main

En esta ultima seccion se hara una breve explicacion de los metodos main usados en estos programas , he decidido usar como ejemplo el de los lectores pero los otros main se han escrito con el mismo metodo . En este caso se declaran dos arrays de hebras uno de lectoras y otro de escritoras y tambien se declara un objeto monitor de la clase que se ha definido previamente . Ambos arrays de hebras se inicializan en un bucle for que recorre el array al completo iniciando cada hebra con su funcion y argumentos correspondientes . Los dos siguientes bucles for se usan para aplicar el metodo join sobre todas las hebras de los arrays para que unas esperen a otras a la hora de finalizar , en el caso de los lectores y los fumadores nunca terminan ya que son programas infinitos .

Listing 20: función main lectores

```
1   int main (int argc, char *argv[]) {
2       int numlec=4,numesc=4;
3       thread lectoras[numlec],escritoras[numesc];
4
5      MRef<lectorescritor> monitor = Create<lectorescritor>() ;
6       for(int i = 0; i< numesc; i++){
7           escritoras[i]=thread(escritor,monitor,i);
8       }
9
10      for(int i = 0; i< numlec; i++){
11          lectoras[i]=thread(lector,monitor,i);
12      }
13      for(int i = 0; i< numlec; i++){
14          lectoras[i].join();
15      }
16      for(int i = 0; i< numesc; i++){
17          escritoras[i].join();
18      }
19
```

```
20      return 0;
21  }
```