Práctica 1

Pablo Linari Pérez

Octubre 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 semáforos, y así conocer el problema del *productor-consumidor* y gestionar la sincronización de varias hebras.

2 Productor-consumidor

En este problema tenemos una hebra productora y otra consumidora. Debemos implementar una estructura de semáforos que haga que primero se produzca y después se lea el dato para evitar incoherencias en el programa. En este primer programa se ha elegido una estructura FIFO para el buffer de lectura y escritura.

Listing 1: Variables globales para el problema productor-consumidor

```
1  // Variables globales
2
3  const unsigned
4     num_items = 40 ,   // n mero de items
5      tam_vec   = 10 ;   // tama o del buffer
6  unsigned
7     cont_prod[num_items] = {0}, // contadores de
          verificaci n: para cada dato, n mero de veces que se
           ha producido.
8     cont_cons[num_items] = {0}, // contadores de
          verificaci n: para cada dato, n mero de veces que se
           ha consumido.
9     siguiente_dato       = 0 ; // siguiente dato a producir
          en 'producir_dato' (solo se usa ah )
10 unsigned buffer[tam_vec];
11 Semaphore puede_leer(0);
12 Semaphore puede_escribir(tam_vec);
13 int primera_ocupada =0,primera_libre = 0;
```

Al principio del programa vemos la declaración de varias variables globales, las cuales tienen los siguientes usos:

- **unsigned buffer[tam_vec]**: este vector es usado como buffer donde se escriben o se leen los datos producidos.
- **Semaphore puede_leer(0)**: semáforo que indica si la hebra consumidora puede leer. El dato se inicia a cero, ya que inicialmente no hay datos disponibles, por tanto no debe leer.
- **Semaphore puede_escribir(tam_vec)**: semáforo que indica si la hebra productora puede escribir. El dato se inicia a *tam_vec* ya que al principio puede producir tantos datos como huecos haya en el vector.

- **int primera_libre = 0**: entero que indica la posición donde la hebra productora puede escribir.
- **int primera_ocupada = 0**: entero que indica la posición de donde la hebra consumidora puede leer.

Listing 2: función hebra productora

```
1  void  funcion_hebra_productora(  )
2  {
3     for( unsigned i = 0 ; i < num_items ; i++ )
4     {
5        int dato = producir_dato() ;
6        sem_wait(puede_escribir);{
7              buffer[primera_libre] = dato;
8              primera_libre =(primera_libre+1) %tam_vec;
9          }
10         sem_signal(puede_leer);
11    }
12 }
```

Mirando la función *funcion_hebra_productora*, podemos ver una estructura en la cual se aplica un *sem_wait()* al semáforo *puede_escribir*. Esto reduce en una unidad la cantidad de veces que se podrá volver a pasar por ese semáforo estando abierto. Acto seguido, se ejecutan dos líneas de código: en la primera se inserta el dato en la primera posición libre del buffer y en la segunda se recalcula la primera posición libre (el incremento de la variable se hace módulo *tam_vec* ya que una vez el buffer esté lleno se comienza a reutilizar desde la posición 0). La última línea del código aplica *sem_signal()* al semáforo *puede_leer*; de esta manera, se incrementa en una unidad su valor, lo que indicará que puede pasar con el semáforo abierto una vez.

Con esta estructura de semáforos se pretende indicar a cada hebra cuándo y cuántas veces puede leer o escribir. Este proceso se repite el número de veces *num_items*.

Listing 3: función hebra consumidora

```
1
2  void funcion_hebra_consumidora( )
3  {
4     for( unsigned j = 0 ; j < num_items ; j++ )
5     {
6      sem_wait(puede_leer);{
7              consumir_dato(buffer[primera_ocupada]);
8              primera_ocupada = (primera_ocupada +1)%tam_vec;
9          }
10         sem_signal(puede_escribir);
11     }
12 }
```

La función *hebra_consumidora* es la encargada de hacer que la hebra lea del buffer. Para ello, se utiliza una estructura de semáforos similar a la de la función productora, pero esta vez *sem_wait()* se aplica sobre *puede_leer*, ya que cada vez que la hebra consumidora lee un dato del buffer, se debe disminuir en una unidad la posibilidad de volver a leer. Esto garantiza que la hebra solo lea tantas veces como datos producidos haya en ese momento, evitando que lea datos que aún no han sido producidos.

Entre *sem_signal()* y *sem_wait()*, la función se encarga de consumir un dato de la primera posición ocupada del buffer y de actualizar la posición de consumo de manera análoga a la función anterior, utilizando el operador módulo para poder reutilizar el buffer una vez llegado al final.

3 Múltiples productores y consumidores

3.1 FIFO

Primero analizaremos la implementacion fifo la cual es muy parecida a la del programa anterior pero teniendo en cuenta que ahora hay multiples consumidores y productores.

Respecto a la anterior declaracion de variables globales quedan todas igual y se aniaden las siguientes para poder gestionar las multiples hebras .

Listing 4: Variables globales para el problema multi productor consumidor

```
1 const int HEBRASCONSUMIDORAS = 2;
2 const int HEBRASPRODUCTORAS = 4;
3 Semaphore cambiaindice(1);
4 Semaphore cambiaindice1(1);
5 int producidos[HEBRASPRODUCTORAS]={0};
6 int p = num_items/HEBRASPRODUCTORAS;
```

- **HEBRAS_CONSUMIDORAS**: constante que indica el número de hebras consumidoras.
- **HEBRAS_PRODUCTORAS**: constante que indica el número de hebras productoras.
- **Semaphore cambia_indice(1)**: semáforo que se usa como cerrojo para indicar cuándo una hebra puede o no cambiar el índice del buffer FIFO.
- **Semaphore cambia_indice1(1)**: semáforo que se usa como cerrojo para indicar cuándo una hebra puede o no cambiar el índice del buffer FIFO.
- **int producidos[HEBRAS_PRODUCTORAS] = {0}**: vector de enteros que indica qué número ha producido ya la hebra i.
- **int p = num_items / HEBRAS_PRODUCTORAS;**: variable usada para no calcular siempre el cociente en la distribución de los cálculos.

Listing 5: 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 6: función hebra productora multiple (FIFO)

```

void  funcion_hebra_productora( int i )
{
for (unsigned j = i*p ; j<i*p +p; j++ )
   {
      int dato = producir_dato(i) ;
      sem_wait(puede_escribir);{
            sem_wait(cambiaindice);
            buffer[primera_libre] = dato;
            primera_libre =(primera_libre+1) %tam_vec;
            sem_signal(cambiaindice);
        }
        sem_signal(puede_leer);
   }
}
```

La función *hebra_productora* tiene como argumento el índice i para poder asignar a cada hebra que ejecuta la función un rango de valores a generar. La estructura que siguen los semáforos *puede_escribir* y *puede_leer* es la misma que en el programa anterior y sigue la misma lógica, solo que ahora encontramos un nuevo semáforo dentro de esta estructura. El semáforo *cambia_indice* se usa como cerrojo para no permitir que dos hebras realicen a la vez la operación de modificar el índice o asignar el dato. Por tanto, cada vez que una hebra llega a este semáforo, si está abierto, lo cierra, ejecuta su código, y al salir lo vuelve a abrir para modificar las variables sin error.

Listing 7: función hebra consumidora multiple (FIFO)

```
void funcion_hebra_consumidora(  )
```

```
{
   for( unsigned j = 0 ; j < num_items/HEBRASCONSUMIDORAS ;
       j++ )
   {
    sem_wait(puede_leer);{
             sem_wait(cambiaindice1);
             consumir_dato(buffer[primera_ocupada]);
             primera_ocupada = (primera_ocupada +1)%tam_vec;
             sem_signal(cambiaindice1);
         }
         sem_signal(puede_escribir);
    }
}
```

Por último, en la función hebra consumidora se pasa como argumento el índice de una hebra consumidora. El bucle se ejecuta $\frac{num\_items}{HEBRASCONSUMIDORAS}$ para que el trabajo se distribuya uniformemente entre cada hebra consumidora. Los semáforos *puede_leer* y *puede_escribir* funcionan de igual modo que en el primer programa y se utiliza el semáforo *cambiaindice1* como cerrojo para lograr el mismo objetivo que en la función anterior, es decir, mantener la atomicidad a la hora de cambiar los índices y los datos.

3.2 LIFO

Listing 8: función hebra productora multiple (LIFO)

```

void  funcion_hebra_productora( int i )
{
for (unsigned j = i*p ; j<i*p +p; j++ )
   {
      int dato = producir_dato(i) ;
      sem_wait(puede_escribir);{
             sem_wait(cambiaindice1);
             buffer[primera_libre] = dato;
             primera_libre++;
             sem_signal(cambiaindice1);
        }
        sem_signal(puede_leer);
   }
}
```

Listing 9: función hebra consumidora multiple (LIFO)

```
1
2
3  void funcion_hebra_consumidora(  )
4  {
5     for( unsigned j = 0 ; j < num_items/HEBRASCONSUMIDORAS ;
          j++)
6     {
7      sem_wait(puede_leer);{
8               sem_wait(cambiaindice1);
9               primera_libre--;
10              consumir_dato(buffer[primera_libre],i);
11              sem_signal(cambiaindice1);
12          }
13          sem_signal(puede_escribir);
14     }
15 }
```

Como se puede observar, la diferencia entre el FIFO y el LIFO se encuentra únicamente en el modo de uso del *buffer*, ya que en el caso del LIFO este adopta una estructura de pila, pasando así de necesitar dos índices a usar solamente uno, el cual indicará dónde escribe y de dónde lee cada hebra.

Dado esto, podemos usar un único cerrojo, ya que solo empleamos una variable de índice. (El funcionamiento del cerrojo es igual que en el FIFO, pero esta vez con una sola variable).

Listing 10: 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 los fumadores con el estanquero, debemos usar un semáforo para indicar cuándo el mostrador está vacío, y un array de semáforos para indicar cuándo está disponible cada ingrediente para ser consumido.

Listing 11: función hebra estanquero

```
1
2
3  void funcion_hebra_estanquero(  )
4  {
5      int i ;
6      while (true) {
7          i = producir_ingrediente();
8          sem_wait(mostrador_vacio);{
9              cout<<"puesto ingrediente "<<i<<endl;
10         }sem_signal(ingredientes[i]);
11
12     }
13
14 }
```

La función *hebra_estanquero* tiene una estructura de semáforos que le permite saber si el mostrador está vacío.
Primero, se le aplica *sem_wait* al semáforo *mostrador_vacio*, el cual está inicializado en 1, y por tanto, comienza vacío. Cuando se le aplica el primer *sem_wait*, su valor pasa a 0, indicando que el mostrador está lleno. Si el mostrador está vacío, se coloca el ingrediente y se aplica *sem_signal* al semáforo *ingredientes[i]* (estos se inicializan en 0, ya que si el mostrador está vacío los fumadores no pueden consumir), para indicar que el ingrediente del fumador i puede ser recogido. Tras esto, el valor del semáforo *ingredientes[i]* se incrementa en una unidad, indicando así que se podrá ejecutar el código del fumador i.

Listing 12: función hebra fumador

```
void  funcion_hebra_fumador( int num_fumador )
{
   while( true )
   {
        sem_wait(ingredientes[num_fumador]);{
            cout<<"Retirado el ingrediente "<<num_fumador<<
                endl;
        }sem_signal(mostrador_vacio);
        fumar(num_fumador);

   }
}
```

Esta función recibe como parámetro el número del fumador que va a ejecutar dicha función y, dependiendo de este índice, se utilizará un semáforo u otro. En la estructura de semáforos, primero se aplica un *sem_wait* al semáforo en la posición *num_fumador* del array. Si este tiene un valor superior a cero, el fumador *num_fumador* puede comenzar a fumar. A continuación, se aplica *sem_signal* al semáforo *mostrador_vacio*, lo que disminuye en una unidad el valor del semáforo e indica a la hebra del estanquero que puede depositar un nuevo ingrediente en el mostrador.

Listing 13: función main

```
int main()
{
   thread vendedor( funcion_hebra_estanquero ),fumador1(
       funcion_hebra_fumador,0),fumador2(
       funcion_hebra_fumador,1),fumador3(
       funcion_hebra_fumador,2);
    vendedor.join();
    fumador1.join();
    fumador2.join();
    fumador3.join();
    return 0;
}
```

Por último, la función *main* de este programa se encarga de activar las cuatro hebras que participan: la del estanquero y las tres de fumadores, las cuales se inicializan con los índices 0, 1 y 2 como argumento a las funciones que ejecutan. Finalmente, se realiza un *join* de todas las hebras para que todas esperen a las demás antes de terminar. Sin embargo, dado que no se ejecuta ninguna función o código después, y la ejecución de todas las funciones es infinita, este *join* no tendrá efecto.