Sistemas Concurrentes y Distribuidos. Práctica 1

Sincronización de hebras con Semáforos. Problema Productor-Consumidor

Ricardo Ruiz Fernández de Alba

01/10/2023

Índice

Descripción del problema

El problema del productor consumidor surge cuando se quiere diseñar un programa en el cual una hebra produce items de datos en memoria mientras que otra hebra los consume.

En general, la productora calcula o produce una secuencia de items de datos (uno a uno), y la consumidora lee o consume dichos items (tambien uno a uno).

El tiempo que se tarda en producir un item de datos puede ser variable y en general distinto al que se tarda en consumirlo (también variable).

En esta práctica se pide implementar una solución al problema del productor consumidor utilizando semáforos.

Esquema General de la Solución

```
{ variables compartidas y valores iniciales }
var tam_vec : integer := k ; { tamaño del vector }
num_items : integer := .... ; { número de items }
vec : array[0..tam_vec-1] of integer; { vector intermedio }

process HebraProductora ; var a : integer ;
  begin
    for i := 0 to num_items-1 do begin
      a := ProducirValor() ;
      { Sentencia E: } { (insertar valor 'a' en 'vec') }
    end
end

process HebraConsumidora var b : integer ;
begin
  for i := 0 to num_items-1 do begin
  { Sentencia L: } { (extraer valor 'b' de 'vec') }
  ConsumirValor(b) ;
end
```

Implementación función principal

```
void main() {
   [...]
   thread hebra_productora ( funcion_hebra_productora ),
          hebra_consumidora( funcion_hebra_consumidora );

   hebra_productora.join() ;
```

```
7     hebra_consumidora.join() ;
8     cout << "fin" << endl;
9     test_contadores() ;
10 }
```

Diseño general mediante semáforos

Describe los semáforos necesarios, la utilidad de los mismos, el valor inicial y en qué puntos del programa se debe usar sem_wait y sem_signal sobre ellos.

Utilizaremos tres semáforos para sincronizar las hebras: `libres`, `ocupadas` y `op_buffer`.

El semáforo `libres` nos indicará el número de posiciones libres del vector. Inicialmente, `libres = tam_vec`.

El semáforo `ocupadas` nos indicará el número de posiciones ocupadas del vector. Inicialmente, `ocupadas = 0`.

El semáforo `op_buffer` nos indicará si se está realizando una operación sobre el buffer. Inicialmente, `op_buffer = 1`. Esto permite que la hebra productora y la hebra consumidora no puedan realizar operaciones sobre el buffer a la vez. (Exclusión mutua entre inserción y extracción).

Sincronización entre los semáforos

La hebra productora deberá esperar a que haya al menos una posición libre en el vector para poder insertar un valor. Esto se consigue mediante la operación `libres.sem_wait()`.

La hebra consumidora deberá esperar a que haya al menos una posición ocupada en el vector para poder extraer un valor. Esto se consigue mediante la operación `ocupadas.sem_wait()`.

La hebra productora deberá esperar a que no se esté realizando ninguna operación sobre el buffer para poder insertar/extraer un valor. Esto se consigue mediante la operación `op_buffer.sem_wait()`.

Cuando una operación sobre el buffer ha finalizado, se debe liberar el semáforo `op_buffer` mediante la operación `op_buffer.sem_signal()`.

Cuando se ha insertado un valor en el vector, se debe incrementar el número de posiciones ocupadas mediante la operación `ocupadas.sem_signal()`.

Cuando se ha extraído un valor del vector, se debe incrementar el número de posiciones libres mediante la operación `libres.sem_signal()`.

Diseño mediante pila acotada (LIFO)

La solución LIFO (Last In First Out) consiste en que la hebra productora inserta los valores en la última posición libre del vector, y la hebra consumidora extrae los valores de la última posición ocupada del vector. Esto es equivalente a considerar el vector como una pila acotada.

Haremos uso de una variable entera `primera_libre` que nos indicará la primera posición libre del vector. Inicialmente, `primera_libre = 0`.

La hebra productora escribirá en la posición `vec[primera_libre]` y después incrementará el valor de `primera_libre`.

La hebra consumidora leerá de la posición `vec[primera_libre - 1]` y después decrementará el valor de `primera_libre`.

Implementación LIFO en C++11

Función hebra productora

```
1  Semaphore
2    libres(tam_vec),
3     ocupadas(0),
4     op_buffer(1);
5
6  unsigned int buffer[tam_vec], /
7  primera_libre = 0;
8  [...]
9  void  funcion_hebra_productora(  )
10 {
11    for( unsigned i = 0 ; i < num_items ; i++ )
12    {
13       int dato = producir_dato() ;
14
15       // Inicio SC
16       libres.sem_wait(); // Producir tantos datos como elementos libres
             haya en el buffer
17       op_buffer.sem_wait(); // Inserción en exclusión mutua con la
            extracción
18
19       // Inserción del dato
20       assert(0 <= primera_libre && primera_libre < tam_vec);
21       buffer[primera_libre++] = dato;
22       cout << "inserción en buffer: " << buffer[primera_libre] << endl;
23       mostrar_buffer();
24
25       op_buffer.sem_signal();
26       ocupadas.sem_signal(); //
```

```
27         // Fin SC
28      }
29   }
```

Función hebra consumidora

```
1  void funcion_hebra_consumidora(  )
2  {
3     for( unsigned i = 0 ; i < num_items ; i++ )
4     {
5        int dato ;
6
7        // Inicio SC
8        ocupadas.sem_wait(); // Espera hasta que haya al menos un
            elemento en el buffer
9        op_buffer.sem_wait(); // La extraccion debe ocurrir en exclusion
            mutua con la inserción
10
11       // Extracción del dato
12       assert(0 <= primera_libre && primera_libre < tam_vec);
13       dato = buffer[--primera_libre];
14       buffer[primera_libre] = 0;
15       cout << "extraído de buffer: " << dato << endl;
16       mostrar_buffer();
17
18
19       op_buffer.sem_signal();
20       libres.sem_signal(); // Se ha extraido un elemento del buffer,
            queda uno mas libre
21       // Fin SC
22
23       consumir_dato( dato ) ;
24    }
25 }
```

Diseño mediante cola circular (FIFO)

La solución FIFO (First In First Out) consiste en que la hebra productora inserta los valores en la última posición libre del vector, y la hebra consumidora extrae los valores de la primera posición ocupada del vector. Esto es equivalente a considerar el vector como una cola circular.

En este caso, necesitaremos dos variables enteras `primera_libre` y `primera_ocupada` que nos indicarán la primera posición libre y la primera posición ocupada del vector, respectivamente. Inicialmente, `primera_libre = primera_ocupada = 0`.

La hebra productora escribirá en la posición `vec[primera_libre]` y después incrementará el valor de `primera_libre`.

La hebra consumidora leerá de la posición `vec[primera_ocupada]` y después incrementará el valor de `primera_ocupada`.

Cuando las variables `primera_libre` o `primera_ocupada` alcancen el valor `tam_vec`, se reiniciarán a 0. Esto se consigue mediante la operación módulo (%).

Implementación FIFO en C++11

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
7        // Inicio SC
8        libres.sem_wait(); // Producir tantos datos como elementos libres
9        op_buffer.sem_wait(); // La inserción y extracción del buffer
            deben ocurrir
10                            // en exclusión mutua.
11       // Inserción del dato
12       assert(0 <= primera_libre && primera_libre < tam_vec);
13       buffer[primera_libre] = dato;
14       cout << "inserción en buffer: " << buffer[primera_libre] << endl;
15       primera_libre = (primera_libre +1) % tam_vec;
16       mostrar_buffer();
17
18       op_buffer.sem_signal();
19       ocupadas.sem_signal();
20       // Fin SC
21    }
22 }
```

Función hebra consumidora

```
1  void funcion_hebra_consumidora(  )
2  {
3     for( unsigned i = 0 ; i < num_items ; i++ )
4     {
5        int dato ;
6
7        // Inicio SC
```

```
8         ocupadas.sem_wait(); // Consumir cuando haya al menos un elemento
              en el buffer
9         op_buffer.sem_wait(); // Extracción debe estar en exclusión mutua
              con la inserción.
10
11        // Extracción del dato
12        assert(0 <= primera_ocupada && primera_ocupada < tam_vec);
13        dato = buffer[primera_ocupada];
14        buffer[primera_ocupada] = 0;
15        primera_ocupada = (primera_ocupada +1) % tam_vec;
16        cout << "extraido de buffer: " << dato << endl;
17        mostrar_buffer();
18
19        op_buffer.sem_signal(); // Fin de la operacion
20        libres.sem_signal(); // Se ha liberado un elemento del buffer.
21        // Fin SC
22
23        consumir_dato(dato) ;
24      }
25  }
```

Corrección del programa

Verificaremos mediante la función `test_contadores` que el número de veces que se produce un número natural es igual al número de veces que se consume dicho número.

```
1  void test_contadores()
2  {
3     bool ok = true ;
4     cout << "comprobando contadores ...." ;
5     for( unsigned i = 0 ; i < num_items ; i++ )
6     {  if ( cont_prod[i] != 1 )
7        {  cout << "error: valor " << i << " producido " << cont_prod[i]
              << " veces." << endl ;
8           ok = false ;
9        }
10       if ( cont_cons[i] != 1 )
11       {  cout << "error: valor " << i << " consumido " << cont_cons[i]
              << " veces" << endl ;
12          ok = false ;
13       }
14    }
15    if (ok)
16       cout << endl << flush << "solución (aparentemente) correcta." <<
             endl << flush ;
17 }
```

Resultado de ejecución

En ambos casos, el test de contadores se ha pasado correctamente.

LIFO

```
[...]
fin
comprobando contadores ....
solución (aparentemente) correcta.
```

FIFO

```
[...]
fin
comprobando contadores ....
solución (aparentemente) correcta.
```

Código fuentes

Se adjuntan los códigos fuentes de los programas en C++11 en una carpeta ‘scd-p1-fuente“

Compilación y ejecución

Para compilar los programas, se ha creado un `Makefile` que permite compilar los programas mediante el comando `make`.

```
$ make prodcons-lifo_exe
$ ./prodcons-fifo_exe

$ make prodcons-fifo_exe
$ ./prodcons-fifo_exe
```