Práctica 1: Sincronización de hebras con semáforos

Sistemas Concurrentes y Distribuidos

Francisco Lara Marín

77447257-R

franciscolara@correo.ugr.es

Grado en Ingeniería Informática

ETS de Ingenierías Informática y de Telecomunicación

Universidad de Granada

Curso 2022-2023

Contenido

1

Introducción

La documentación contendrá la explicación y distintas anotaciones de dos problemas sencillos de sincronización.

- El problema del productor-consumidor y sus aplicaciones.
- El problema de los fumadores.

La práctica se apoya en varias plantillas ofrecidas por el profesorado que permitirán por ejemplo la generación de números aleatorios.

2

El problema del productor-consumidor

Se diseñará un programa en el cual un proceso o hebra produce items de datos en memoria que otro proceso consume. La implementación debe asegurar que:

- Cada ítem producido es leído
- Ningún ítem se lee más de una vez.

Antes de comenzar a implementar las distintas versiones es necesario declarar los semáforos mencionados en el guión ya que estos son comunes a todas las implementaciones. Para ello declaro dos variables de tipo "Semaphore":

```
Semaphore
   ocupadas(0),      // Inicialmente 0, el valor será: Insertados - extraidos
   libres(tam_vec);  // Inicialmente tam_vec, el valor será:
                     // tam_vector + extraidos - insertados
```

Figure 2.1: Declaración de los semáforos

Se declara a continuación el vector intermedio (buffer) y las variables que nos permitirán gestionar su ocupación.

```
int
   primera_libre = 0,
   primera_ocupada = 0,
   intermedio[tam_vec]; // Vector intermedio
```

Figure 2.2: Buffer

2.1 | Implementación versión LIFO

Se completará primero la función de la hebra productora. Para ello es necesario comprender las operaciones básicas que se pueden realizar sobre la variable de tipo semáforo.

```
void  funcion_hebra_productora(  )
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato = producir_dato() ;
      sem_wait(libres);
      intermedio[primera_libre] = dato;
      primera_libre++;
      sem_signal(ocupadas);
   }
}
```

Figure 2.3: Función hebra productora

1. Se produce un dato.
2. Se decrementa el valor de libres mediante sem_wait. Si el valor de libres es 0 se espera.
3. Se añade el dato en el primer hueco libre del buffer.
4. Se actualiza la primera posición libre.
5. Se incrementa ocupadas mediante sem_signal.

```
void funcion_hebra_consumidora(  )
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato ;
      // LIFO
      sem_wait(ocupadas);
      dato = intermedio[primera_libre-1];
      primera_libre--;
      sem_signal(libres);
      consumir_dato( dato ) ;
   }
}
```

Figure 2.4: Función hebra consumidora

1. Se decrementa el valor de ocupadas. Si el valor es 0 se espera.
2. El dato a consumir es el último añadido, es decir, el inmediatamente anterior a la primera posición libre.
3. Se decrementa la primera posición libre.
4. Se incrementa libres.
5. Finalmente se consume el dato.

2.2 | Implementacion con múltiples productores y consumidores

En este problema hay que tener en cuenta la exclusión mutua. Para ello se utilizarán en ambas versiones objetos mutex, estas variables permiten la exclusión mutua mediante espera bloqueada.

```
mutex
   mtx1,
   mtx2;
```

Figure 2.5: Objetos mutex

Un objeto mutex será utilizado en las funciones producir_dato y consumir_dato mientras que el segundo será utilizado en las funciones hebra. Las funciones producir y consumir reciben los siguientes cambios:

```
unsigned producir_dato()
{
   this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
   const unsigned dato_producido = siguiente_dato ;
   siguiente_dato++ ;
   cont_prod[dato_producido] ++ ;
   mtx1.lock();
   cout << "producido: " << dato_producido << endl << flush ;
   mtx1.unlock();
   return dato_producido ;
}
```

Figure 2.6: Función producir_dato modificada

```
void consumir_dato( unsigned dato )
{
   assert( dato < num_items );
   cont_cons[dato] ++ ;
   this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
   mtx1.lock();
   cout << "                  consumido: " << dato << endl ;
   mtx1.unlock();
}
```

Figure 2.7: Función consumir_dato modificada

En la operación lock, la hebra espera si hay una ejecutando (queda bloqueada). Unlock indica que ha terminado de ejecutar lo que permite a otras hebras comenzar su ejecución.

2.2.1 | LIFO

```
void  funcion_hebra_productora( int num_hebras )
{
   int items_hebra = num_items/num_hebras;

   for( unsigned i = 0 ; i < items_hebra ; i++ )
   {
      int dato = producir_dato() ;
      // LIFO
      sem_wait(libres);
      mtx2.lock();
      intermedio[primera_libre] = dato;
      primera_libre++;
      mtx2.unlock();
      sem_signal(ocupadas);
   }
```

El número de ítems a producir por hebra sera el número de ítems totales entre el número de hebras productoras, es decir, el número total de ítems debe ser múltiplo. De la misma manera modificamos también la función de la hebra consumidora:

```
void funcion_hebra_consumidora( int num_hebras )
{
   int items_hebra = num_items/num_hebras;

   for( unsigned i = 0 ; i < items_hebra ; i++ )
   {
      int dato ;
      // LIFO
      sem_wait(ocupadas);
      mtx2.lock();
      dato = intermedio[primera_libre-1];
      primera_libre--;
      mtx2.unlock();
      sem_signal(libres);
      consumir_dato( dato ) ;
    }
}
```

2.2.2 | FIFO

Para la segunda solución FIFO será necesario realizar unas modificaciones en las funciones hebra productora y consumidora. La manera de la que se gestiona el buffer cambia utilizando otra variable llamada primera_ocupada.

```
void  funcion_hebra_productora( int num_hebras )
{
   int items_hebra = num_items/num_hebras;

   for( unsigned i = 0 ; i < items_hebra ; i++ )
   {
      int dato = producir_dato() ;
      // FIFO
      sem_wait(libres);
      mtx2.lock();
      intermedio[primera_libre] = dato;
      primera_libre = (primera_libre+1)%tam_vec;
      mtx2.unlock();
      sem_signal(ocupadas);
   }
}
```

Figure 2.8: Función hebra productora FIFO

```
void funcion_hebra_consumidora( int num_hebras )
{
   int items_hebra = num_items/num_hebras;

   for( unsigned i = 0 ; i < items_hebra ; i++ )
   {
      int dato ;
      // FIFO
      sem_wait(ocupadas);
      mtx2.lock();
      dato = intermedio[primera_ocupada];
      primera_ocupada = (primera_ocupada+1)%tam_vec;
      mtx2.unlock();
      sem_signal(libres);
      consumir_dato( dato ) ;
   }
}
```

Figure 2.9: Función hebra consumidora FIFO

3

El problema de los fumadores

Como inicio se declaran dos variables de tipo semáforo:

```
Semaphore
    ingr_disp[num_fumadores] = {0,0,0},
    mostr_vacio(1);
```

Figure 3.1: Semáforos

- **ingr_disp:** Indica que ingrediente esta preparado y disponible en el mostrador.
- **mostr_vacio:** 1 si el mostrador está libre, 0 si esta ocupado por algún ingrediente.

```
void funcion_hebra_estanquero(  )
{
   int ingrediente;
   while(true){
      ingrediente = producir_ingrediente();
      sem_signal(ingr_disp[ingrediente]);
      sem_wait(mostr_vacio);
      cout << "Estanquero : coloca el ingrediente: " << ingrediente << endl;
   }
}
```

Figure 3.2: Función hebra estanquero

1. Se produce un ingrediente.
2. Se indica que el ingrediente esta listo.

3. Se espera a que el mostrador vuelva a estar vacío.

```
void  funcion_hebra_fumador( int num_fumador )
{
   while( true )
   {
      sem_wait(ingr_disp[num_fumador]);
      cout << "Fumador " << num_fumador << " retira: " << num_fumador << endl;
      sem_signal(mostr_vacio);
      fumar(num_fumador);
   }
}
```

Figure 3.3: Función hebra fumador

1. El fumador espera que su ingrediente esté preparado.
2. Se indica que el mostrador está vacío de nuevo.
3. El fumador fuma.