Práctica 1: Sistemas concurrentes y distribuidos.

Pablo Manresa Nebot.
11/10/2018
Grupo: A2

Índice:

Problema Productor-Consumidor.

Consta de una función llamada **productor** que produce un resultado como consecuencia de alguna operación y dicho **consumidor**, accede a dicho resultado y lo almacena(entre otras tantas cosas que puede realizar).

Se resolverá este problema de manera concurrente.

Se usarán lo siguiente:

```
thread hebra_productora ( funcion_hebra_productora ),
       hebra_consumidora( funcion_hebra_consumidora );

hebra_productora.join() ;
hebra_consumidora.join() ;
```

Para este caso, en concreto, la función **productor** produce un número aleatorio mediante el siguiente trozo de código, usando las características de C++11.

```
template< int min, int max > int aleatorio()
{
  static default_random_engine generador( (random_device())() );
  static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
  return distribucion_uniforme( generador );
}
```

1. Productor-Consumidor. FIFO.

1.1 Variables.

A continuación, se explicará de manera breve cada variable global:

```
const int num_items = 60 ,   // número de items
          tam_vec   = 10 ;   // tamaño del buffer

// contadores de verificación: producidos
unsigned  cont_prod[num_items] = {0},

 // contadores de verificación: consumidos
         cont_cons[num_items] = {0};

int buffer_intermedio[tam_vec]; //buffer de datos


// variables para asegurar el Fifo siendo primera_libre la primera celda libre de buffer_intermedio y primera_ocupada la primera ocupada del mismo buffer.
int primera_libre = 0,
   primera_ocupada = 0;

// Semáforos para controlar la sincronización de hebras estando inicializadas libres al tamaño del vector(10) y ocupadas a 0.
Semaphore libres{tam_vec}, ocupadas{0};
```

-Se ha utilizado la estrategia **FIFO** de modo que para garantizarla, se han usado las variables primera_libre y primera_ocupada.

primera_libre se incrementa tras cada escritura en el buffer y primera_ocupada tras cada lectura de buffer. Para garantizar que primera_librey primera_ocupada sean menores que tam_vec y de este modo no se produzca un **segmentation fault** se usa la operación módulo.

Ejemplo:

```
primera_libre%=tam_vec;
```

que es equivalente a:

```
if(primera_libre == tam_vec-1)
    primera_libre = 0;
else
    primera_libre++;
```

Para acceder al vector e incrementar las variables, se deben usar semáforos para garantizar la exclusión mútua. De este modo, se evitará que se modifique de manera primera_libre por la hebra consumidora haciendo que se consuman valores repetidos.

1.2 Código:

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

```
void funcion_hebra_consumidora(  )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
    int dato ;
    sem_wait(ocupadas);
    dato = buffer_intermedio[primera_ocupada];
    primera_ocupada++;
    consumir_dato( dato ) ;
    sem_signal(libres);
    primera_ocupada%=tam_vec;
   }
}
```

2. Productor-Consumidor. LIFO.

2.1 Variables.

Para resolver dicho problema se han usado las siguientes variables globales:

```
const int num_items = 60 ,   // número de items
          tam_vec   = 10 ;   // tamaño del buffer

// contadores de verificación: producidos
unsigned  cont_prod[num_items] = {0},

// contadores de verificación: consumidos
         cont_cons[num_items] = {0};

int buffer_intermedio[tam_vec]; //buffer de datos


// variable para asegurar el LIFO siendo primera_libre la primera celda libre de buffer_intermedio.
int primera_libre = 0;


// Semáforos para controlar la sincronización de hebras estando inicializadas libres al tamaño del vector(10) y ocupadas a 0.

Semaphore libres{tam_vec}, ocupadas{0};
```

-Todas son iguales al método **FIFO** excepto primera_ocupada, que no se utiliza.

En cuanto a los semáforos, libres se inicializa a tam_vec y ocupadas a 0.

2.2 Código:

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


void funcion_hebra_consumidora(  )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
    sem_wait(ocupadas);
                        int dato ;
                        primera_libre--;
                        dato = buffer_intermedio[primera_libre];


    consumir_dato( dato ) ;
                        sem_signal(libres);
   }
}
```

3. SEMÁFOROS.

La función funcion_hebra_productora invoca a sem_wait(libres) y sem_signal(ocupadas) aumentando ocupadas en 1.
A continuación, funcion_hebra_consumidora invoca a sem_wait(ocupadas) y sem_signal(libres) para que, de este modo, se garantice la lectura después de la escritura, dejando ocupadas a 0.
Además, ocupadas no incrementará a 1 hasta que no se vuelva a llamar a funcion_hebra_productora.

4. Problema de los fumadores.

El siguiente problema consiste en lo siguiente:

-Hay 3 fumadores y 1 estanquero, el estanquero produce un ingrediente y uno de los fumadores ha de recogerlo para poder fumar.

-El periodo por el cual “fuman” consiste en una espera de tiempo aleatorio, pudiendo varios fumadores fumar a la vez.

Para ello, cada fumador es representado mediante una hebra y el estanquero mediante otra. El siguiente código muestra dicha representación:

```
std::thread fumadores[3];
  std::thread estanquero(funcion_hebra_estanquero);

for(int i = 0; i < 3; i++) fumadores[i] = thread(funcion_hebra_fumador, i);
```

Para sincronizarlas, se ha usado el siguiente trozo de código:

```
for(int i = 0; i < 3; i++) fumadores[i].join();
  estanquero.join();
```

4.1 Variables:

```
// para indicar si el mostrador se encuentra vacío o no
Semaphore mostr_vacio = 1;

// para indicar los ingredientes disponibles siendo:
// el 0 cerillas, el 1 tabaco y el 2 papel.
Semaphore ingr_disp[3] = {0,0,0};

// vector cuyo uso es para mostrar el ingrediente producido
std::vector<std::string> ingredientes{"cerillas", "tabaco", "papel"};
```

4.2 Código:

```
void funcion_hebra_estanquero(  ){
  int dato;
  while(true){
    dato = aleatorio<0,2>();
    sem_wait(mostr_vacio);
    std::cout << "Ingrediente " << ingredientes[dato] << std::endl;
    sem_signal(ingr_disp[dato]);
  }
}

void funcion_hebra_fumador( int num_fumador )
{
//  std::cout << "Fumador " << num_fumador << " ingrediente " <<
ingredientes[num_fumador] << std::endl;
   while( true ){
     sem_wait(ingr_disp[num_fumador]);
     std::cout << "retirado por el fumador " << num_fumador<< " ingrediente " <<
ingredientes[num_fumador] << std::endl;
     sem_signal(mostr_vacio);
     fumar(num_fumador);
   }
}
```

4.3 Semáforos:

La función funcion_hebra_estanquero tiene los siguientes semáforos:

-sem_wait(mostr_vacio).
-sem_signal(ingr_disp[dato]);

Para poder producir un ingrediente, dejando a 0 una vez producido y poniendo a 1 el ingrediente producido.

Sin embargo, la función funcion_hebra_fumador, tiene los siguientes semáforos:

-sem_wait(ingr_disp[num_fumador]);
-sem_signal(mostr_vacio);

Dejando a 0 el ingrediente producido para poder ser consumido y poniendo a 1 el mostrador para poder producir un ingrediente.