Alberto Rodríguez Santana 48950011-P Grupo D1

PRACTICA 1

Productor - Consumidor

Variables

Como variables compartidas están las siguientes:

- Num_items -> Variable de tipo entero, inicializada a 60 que representa el número de ítems que se producirán/leerán.
- Tam_vec -> Tamaño del buffer, con valor constante 10.
- Vec -> array compartido de valores de tipo (int), donde se introducirán y de donde se extraerán los datos siguiendo la estrategia LIFO.
- Primera_libre -> inicialmente 0. Representa el índice en el vector de la primera celda libre, esta variable será incrementada al escribir y decrementada al leer.
- Mtx1/mtx2 -> Objetos de la clase Mutex que son utilizados para que la salida en pantalla salga mezclada (Exclusión mutua de la SC comprendida en los cout de cada una de las funciones de las hebras, tanto consumidora como productora.

```
// variables compartidas

const int num_items = 60 ,   // número de items
          tam_vec   = 10 ;   // tamaño del buffer
unsigned  cont_prod[num_items] = {0}, // contadores de verificación: producidos
          cont_cons[num_items] = {0}; // contadores de verificación: consumidos

int vec[tam_vec];

int primera_libre=0;

//Mutex para exclusion mutua de los cout.
mutex mtx1, mtx2;
```

Semáforos

- Ocupadas -> Representa el número de entradas ocupadas en el vector. Inicialmente vale 0 vector.
- Libres -> Representa el numero de entradas libres. Se inicializa a tam_vec para que se cumpla la interfoliación de E/L/E/L.. Antes de la sentencia E (Producir) se hace sem_wait sobre el semáforo hasta que toma el valor 0, y una vez Insertado el valor producido en el buffer, se hace sem_signal sobre ocupadas incrementando el valor del semáforo en (1). Por tanto queda en espera bloqueada hasta que se decremente.

```
Semaphore ocupadas = 0;  //Semaforo para elementos ocupados (núm. entradas ocup. (#insertados - #extraidos))
Semaphore libres = tam_vec; //Semaforo para elementos libres (núm. entradas libres (tam_vec + #extraidos - #insertados))
```

Código Fuente

```
//**********************************************************************
// plantilla de función para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
//----------------------------------------------------------------------

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 );
}

//**********************************************************************
// funciones comunes a las dos soluciones (fifo y lifo)
//----------------------------------------------------------------------

int producir_dato()
{
   static int contador = 0 ;
   this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));

   cout << "producido: " << contador << endl << flush ;

   cont_prod[contador] ++ ;
   return contador++ ;
}
//----------------------------------------------------------------------

void consumir_dato( unsigned dato )
{
   assert( dato < num_items );
   cont_cons[dato] ++ ;
   this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));

   cout << "                  consumido: " << dato << endl ;

}
```

```
//----------------------------------------------------------------------

void test_contadores()
{
   bool ok = true ;
   cout << "comprobando contadores ...." ;
   for( unsigned i = 0 ; i < num_items ; i++ )
   {  if ( cont_prod[i] != 1 )
      {  cout << "error: valor " << i << " producido " << cont_prod[i] << " veces." << endl ;
         ok = false ;
      }
      if ( cont_cons[i] != 1 )
      {  cout << "error: valor " << i << " consumido " << cont_cons[i] << " veces" << endl ;
         ok = false ;
      }
   }
   if (ok)
      cout << endl << flush << "solución (aparentemente) correcta." << endl << flush ;
}

//----------------------------------------------------------------------

void  funcion_hebra_productora(  )
{
   for( unsigned i = 0 ; i < num_items ; i++ ){

      int dato = producir_dato() ;

      sem_wait(libres);

      vec[primera_libre]=dato;
      primera_libre++;

      sem_signal(ocupadas);

      mtx1.lock();
      cout<<"Dato Introducido: " << dato << endl;
      mtx1.unlock();

    }
}
```

```
void funcion_hebra_consumidora(  )
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato ;

      sem_wait(ocupadas);

      primera_libre--;
      dato = vec[primera_libre];

      sem_signal(libres);

      mtx2.lock();
      cout<<"Dato Consumido: " << dato <<  endl;
      mtx2.unlock();

      consumir_dato( dato );




    }
}
//----------------------------------------------------------------------

int main()
{
   cout << "------------------------------------------------------" << endl
        << "Problema de los productores-consumidores (solución LIFO)." << endl
        << "------------------------------------------------------" << endl
        << flush ;

   thread hebra_productora ( funcion_hebra_productora ),
          hebra_consumidora( funcion_hebra_consumidora );

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

   test_contadores();
}
```