Práctica 1

Salvador Romero Cortés

Productores - Consumidores (1 productor-consumidor)

Para este problema solo necesitamos rellenar las funciones que faltaban dentro de la plantilla del archivo `prodcons-plantilla.cpp`, además de declarar ciertas variables y objetos globales.

Versión LIFO (*Last In First Out*)

- Variables globales

```
Semaphore puede_producir = tam_vec;
Semaphore puede_consumir = 0;
int posicion = 0;
int vec[tam_vec];
mutex mtx;
```

  Aquí tenemos dos semáforos. Uno para controlar a la hebra productora y otra para controlar la hebra consumidora.

  El de la hebra productora, `puede_producir` se inicializa al tamaño del vector usado como *buffer* para que pueda entrar inicialmente tantas veces como huecos haya en el *buffer*. El de la hebra consumidora, `puede_consumir` se inicializa a 0 para que antes de leer ningún dato espere a la productora.

  El entero `posicion` almacena la posición donde se lee y donde se escribe en el *buffer*.

  El mutex `mtx` sirve para garantizar la exclusión mutua a la hora de entrar en el vector.

- `funcion_hebra_productora`

```
void  funcion_hebra_productora(  )
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato = producir_dato() ;
      sem_wait(puede_producir);
      mtx.lock();
         vec[posicion] = dato;
         posicion++;
      mtx.unlock();
      sem_signal(puede_consumir);
   }
}
```

  Esta es la función que ejecuta la hebra productora y se encarga de producir los datos e introducirlos en el vector intermedio. Primero se produce un dato y se guarda en una variable local del *scope* de la función. A continuación, el semáforo encargado de controlar la producción se pone en espera(en el caso de la primera entrada, no espera puesto que el semáforo se inicializa a valores mayores a 0). A continuación activamos el cerrojo para asegurar que el acceso al vector intermedio se hace en exclusión mutua. Tras esta operación desbloqueamos el cerrojo y alertamos al semáforo de la hebra consumidora.

- `funcion_hebra_consumidora`

```
void funcion_hebra_consumidora(  )
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato ;
      sem_wait(puede_consumir);
      mtx.lock();
         posicion--;
         dato = vec[posicion];
      mtx.unlock();
      consumir_dato( dato ) ;
      sem_signal(puede_producir);

    }
}
```

Esta función sigue un esquema similar a la anterior. Con la diferencia de que en lugar de escribir datos en el *buffer* , leemos de él. Otra vez, se vuelve a realizar estas operaciones del vector en exclusión mutua con objetos mutex. Al terminar de consumir el dato se manda una señal al productor para indicar que puede producir el siguiente.

Versión FIFO (First In First Out)

Esta versión es bastante similar a la anteior.

- Variables globales

```
Semaphore puede_producir = tam_vec;
Semaphore puede_consumir = 0;
int posicion_prod = 0, posicion_cons = 0;
int vec[tam_vec];
```

Vemos que es casi igual que en la versión LIFO, solo que ahora prescindimos del mutex y añadimos otro contador para llevar la cuenta de la lectura y escritura del vector por separado. El resto de variables y su inicialización es igual que LIFO.

- `funcion_hebra_productora`

```
void  funcion_hebra_productora(  )
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato = producir_dato() ;
      sem_wait(puede_producir);
         vec[posicion_prod] = dato;
         posicion_prod = (posicion_prod+1)%tam_vec;
      sem_signal(puede_consumir);
   }
}
```

En ese caso producimos un dato y lo almacenamos en una variable local. A continuación el semáforo productor espera la señal para poder comenzar su trabajo (al igual que la versión LIFO, en la primera ocurrencia no espera). A continuación, accede al vector con el contador `posicion_prod`. Para incrementar el contador lo hacemos en módulo tam_vec. Tras realizar

la operación de escritura, manda señal al semáforo de la hebra consumidora para que comience a leer el *buffer*.

- `funcion_hebra_consumidora`

```
void funcion_hebra_consumidora(  )
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato ;
      sem_wait(puede_consumir);
         dato = vec[posicion_cons];
         posicion_cons = (posicion_cons+1) % tam_vec;
      consumir_dato( dato ) ;
      sem_signal(puede_producir);

    }
}
```

El funcionamiento de este procedimiento es análogo al anterior. Semáforo esperando señal de entrada, acceso al vector guardándose en variable local, se consume el dato y finalmente se manda señal a la hebra productora.

Estanquera - Fumadores

En este ejercicio disponemos de una estanquera que va generando materiales necesarios para que los fumadores fumen. Estos productos se disponen en un mostrador desde el cuál los fumadores obtienen los objetos. 1 estanquera produce 3 elementos distintos, tantos como fumadores.

Para la ejecución del programa se utilizan 4 semáforos:

```
const int num_fumadores = 3;
Semaphore materiales[num_fumadores] = {0,0,0};
Semaphore mostrador = 1;
```

Un vector de semáforos (uno para cada fumador) y otro para el mostrador.

El valor inicial de estos semáforos es de 0 para los fumadores y de 1 para el mostrador. Esto es así para que lo fumadores no puedan entrar al mostrador a recoger los ingredientes hasta que la estanquera avise de que haya disponibles. El de la estanquera/mostrador es 1 para que comience a producir sin esperar.

Las señales de los semáforos se dan en las funciones `funcion_hebra_fumador` y `funcion_hebra_estanquero`.

```
void funcion_hebra_estanquero(  )
{
   int mostrado = 0;
   while (true){
      mostrado = producir_ingrediente();
      sem_wait(mostrador);
      cout << "Puesto ingrediente numero: " << mostrado << endl << flush;
      sem_signal(materiales[mostrado]);
   }
}
```

Vemos que el funcionamiento de este procedimiento es muy similar al de la hebra productora del anterior ejercicio. Sin embargo, cambia la señal que envía. Cada vez que pone un ingrediente en el mostrador, el estanco manda una señal al semáforo del fumador con el mismo índice que el material que se ha producido. Esto es para que sólo este fumador acceda al mostrador. Mientras tanto puede producir más ingredientes.

```
void  funcion_hebra_fumador( int num_fumador )
{
   while( true )
   {
      sem_wait(materiales[num_fumador]);
      cout << "Recogido material " << num_fumador << endl << flush;
      sem_signal(mostrador);
      fumar(num_fumador);
   }
}
```

Funciona de manera análoga a las anteriores. Cada hebra fumadora espera la señal de su semáforo para entrar al mostrador. Una vez recoge el material, avisa a la estanquera y se pone a fumar.

Productor - Consumidor (varios)

Esto es una modificación del problema original de productores-consumidores pero adaptado a que hayan varios productores y varios consumidores. Fundamentalmente la diferencia está en que ahora disponemos de un array tanto de productores como consumidores, así como los pequeños cambios de formato para mostrar que hebra produce/consume. El código fuente tanto de la versión FIFO como de la LIFO están disponibles junto con el resto de ejercicios en esta misma carpeta.