

EL PROBLEMA DEL PRODUCTOR-CONSUMIDOR

EJERCICIOS:

Describe la variable o variables necesarias, y cómo se determina en qué posición se puede escribir y en qué posición se puede leer del buffer.

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

- `num_items`
 - Integer.
 - Número de items total.
- `tam_vec`
 - Integer.
 - Número de items que se rellena el buffer (tamaño).
 - El tamaño del buffer es el valor de esta variable.
- `cont_prod[num_items]` y `cont_cons[num_items]`
 - Unsigned.
 - Dos vectores que son contadores de verificación para los productores y consumidores.
- `puede_escribir = tam_vec` y `puede_leer = 0`
 - Semaphore.
 - Controla a la hebra productora y la hebra consumidora a la hora de producir y consumir del buffer para que el índice no se sobrescriba.

CÓDIGO FUENTE

```
#include <iostream>
#include <cassert>
#include <thread>
#include <mutex>
#include <random>
#include "Semaphore.h"

using namespace std ;
using namespace SEM ;

//*****
// variables compartidas

const int num_items = 40 , // 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
```

```
Semaphore puede_escribir = tam_vec,  
       puede_leer = 0;
```

```
mutex mtx;
```

```
int buffer [tam_vec];  
int indice = 0;
```

```
//*****
```

```
// 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>() ));  
  
    mtx.lock();  
    cout << "producido: " << contador << endl << flush ;  
    mtx.unlock();
```

```
    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>() ));  
  
    mtx.lock();  
    cout << "          consumido: " << dato << endl ;  
    mtx.unlock();  
}
```

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

```
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( puede_escribir );
        mtx.lock();
        indice = i % tam_vec;
        buffer[indice] = dato;
        mtx.unlock();

        sem_signal( puede_leer );
    }
}
```

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

```
void funcion_hebra_consumidora( )
{
    for( unsigned i = 0 ; i < num_items ; i++ )
    {
        int dato ;
        sem_wait( puede_leer );
        mtx.lock();
        indice = i % tam_vec;
        dato = buffer[indice];
        mtx.unlock();

        sem_signal( puede_escribir );
    }
}
```

```

        consumir_dato( dato ) ;
    }
}
//-----

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

    thread hebra_productora ( funcion_hebra_productora ),
        hebra_consumidora( funcion_hebra_consumidora );

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

    test_contadores();
}

```

EL PROBLEMA DE LOS FUMADORES

EJERCICIO

Nombres de los semáforos empleados para sincronización y, para cada uno de ellos:

- Utilidad
 - Valor inicial
 - Hebras que hacen wait y signal sobre dicho semáforo
- vector<Semaphore> ingr_disp
 - El semáforo controla si el ingrediente perteneciente a cada fumador esta disponible (1) o no (0).
 - El valor inicial del semáforo es 0.
 - Hace sem_signal(ingr_disp[ingrediente]) la hebra estanquero para indicar que el ingrediente está disponible.
 - Hace sem_wait(ingr_disp[ingrediente]) la hebra fumador cuando consume el ingrediente que estaba disponible.
 - Semaphore mostr_vacio
 - El semáforo controla si el mostrador esta vacío o no.
 - El valor inicial del semáforo es 1.
 - Hace sem_signal(mostr_vacio) la hebra fumador para indicar que el mostrador esta vacío y debe ser rellenado.
 - Hace sem_signal(mostr_vacio) la hebra estanquero para generar un ingrediente y que los fumadores puedan acceder.

CÓDIGO FUENTE

```
#include <iostream>
#include <cassert>
#include <thread>
#include <mutex>
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include "Semaphore.h"

using namespace std ;
using namespace SEM ;

// variables compartidas + Semáforos
const int NUM_FUMADORES = 5; // fumadores
Semaphore mostr_vacio = 1; // Semaforo para indicar si el mostrador esta vacío o no
vector<Semaphore> ingr_disp; // Semaforo para controlar si el ingrediente esta disponible(1) o no (0)
mutex mtx;

//*****
// 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 );
}

// funcion Producir() que produce un numero aleatorio '0,1,2' con retraso
// aleatorio

int producir(){
    // genera un ingrediente aleatorio
    int ingrediente_aleat(aleatorio<0,2>());
    // espera un tiempo aleatorio
    this_thread::sleep_for( chrono::milliseconds( aleatorio<50,200>() ));
    // devuelves el ingrediente generado
    return ingrediente_aleat;
}

// función que ejecuta la hebra del estanquero

void funcion_hebra_estanquero( )
{
    int ingrediente;
    while (true) {
        sem_wait(mostr_vacio); // pone mostr_vacio a 0

        // genera un ingrediente aleatorio y además hace un retraso aleatorio
        ingrediente = producir();

        mtx.lock();
        cout << "Estanquero: Ha producido ingrediente " << ingrediente << endl << flush;
        mtx.unlock();
        // el ingrediente pasa a estar disponible
        sem_signal(ingr_disp[ingrediente]);
    }
}

// Función que simula la acción de fumar, como un retardo aleatoria de la hebra

void fumar( int num_fumador )
{
    // calcular milisegundos aleatorios de duración de la acción de fumar)
    chrono::milliseconds duracion_fumar( aleatorio<50,200>() );

    // informa de que comienza a fumar
    mtx.lock();

```

```

    cout << "Fumador " << num_fumador << " : "
        << " empieza a fumar (" << duracion_fumar.count() << " milisegundos)" << endl <<
flush;
    mtx.unlock();
    // espera bloqueada un tiempo igual a "duracion_fumar" milisegundos
    this_thread::sleep_for( duracion_fumar );

    // informa de que ha terminado de fumar
    mtx.lock();
    cout << "Fumador " << num_fumador << " : termina de fumar, comienza espera de
ingrediente." << endl << flush;
    mtx.unlock();

}

// función que ejecuta la hebra del fumador

void funcion_hebra_fumador( int num_fumador )
{
    while( true ){
        // espera hasta que el ingrediente esta disponible
        sem_wait(ingr_disp[num_fumador]);
        mtx.lock();
        cout << "Fumador: retirado ingrediente: " << num_fumador << endl << flush;
        mtx.unlock();
        sem_signal(mostr_vacio);
        fumar(num_fumador);
    }
}

int main()
{
    for(int k=0; k<NUM_FUMADORES; k++){
        ingr_disp.push_back(0);
    }
    thread hebraEstanquero(funcion_hebra_estanquero);
    thread fumadores[NUM_FUMADORES];
    for(int i = 0; i<NUM_FUMADORES; i++){
        fumadores[i] = thread (funcion_hebra_fumador,i);
    }
    for(int j=0; j<NUM_FUMADORES; j++ ){
        fumadores[j].join();
    }
    hebraEstanquero.join();
}

```