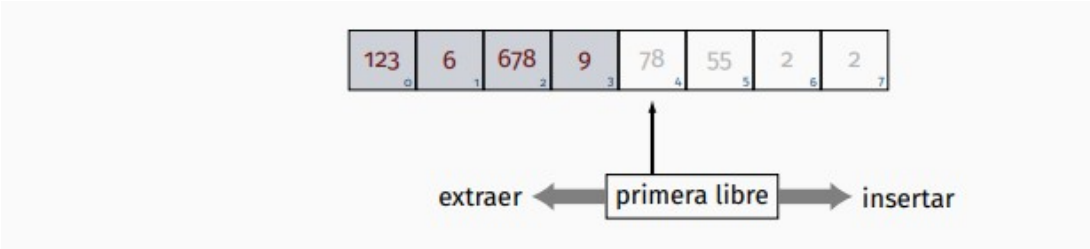
Entrega: Práctica 1 – SCD

Alumno: Fernando Cuesta Bueno – 2º Ing. Informática A1

Ejercicio 1: Productor – consumidor

El fichero del ejecicio es “prodcons.cpp”.
Hemos realizado el ejercicio con una implementación LIFO del vector (buffer):



En este tipo de representación el último valor introducido al buffer es el primero en ser consumido.

Para realizar el ejercicio hemos utilizado tres semáforos:

```
42  // Semáforos
43  Semaphore
44      libres      = tam_vec,
45      ocupadas    = 0      ,
46      acceso      = 1      ;
```

La utilidad de “libres” es manejar la cantidad de espacios libres que tenemos en el buffer intermedio. Es utilizado tanto en la hebra productora como en la consumidora. Empieza inicializado al tamaño del vector porque representa la cantidad de veces que se puede producir un valor en el buffer antes de consumir.

En la hebra productora lo utilizaremos en un wait antes de introducir el dato en el buffer. Esto es porque debemos asegurarnos que el valor de libres sea mayor que 0. No podemos producir un valor si el buffer no tiene espacio.

```
101 void  funcion_hebra_productora(  )
102 {
103    for( unsigned i = 0 ; i < num_items ; i++ )
104    {
105       int dato = producir_dato() ;
106          sem_wait(libres);
107          sem_wait(acceso);
108          buffer[primera_libre] = dato;
109          primera_libre++;
110          sem_signal(acceso);
111          sem_signal(ocupadas);
112    }
113 }
```

En la hebra consumidora haremos un signal del semáforo una vez hayamos consumido el dato. Esto nos indicará que se ha consumido un valor en el buffer, por lo tanto ese espacio queda libre.

```
117  void funcion_hebra_consumidora(  )
118  {
119     for( unsigned i = 0 ; i < num_items ; i++ )
120     {
121        int dato ;
122        sem_wait(ocupadas);
123        sem_wait(acceso);
124        dato = buffer[primera_libre-1];
125        primera_libre--;
126        sem_signal(acceso);
127        sem_signal(libres);
128        consumir_dato( dato ) ;
129     }
130  }
```

En cuanto al semáforo “ocupadas” es utilizado también en ambas funciones de las hebras. Empieza inicializado a 0 porque al inicio el buffer no tiene valores en su interior.

En la función de la hebra productora hacemos un signal del semáforo para indicar que hay una posición del buffer ocupada y que se puede consumir dicho valor.

```
101  void  funcion_hebra_productora(  )
102  {
103     for( unsigned i = 0 ; i < num_items ; i++ )
104     {
105        int dato = producir_dato() ;
106        sem_wait(libres);
107        sem_wait(acceso);
108        buffer[primera_libre] = dato;
109        primera_libre++;
110        sem_signal(acceso);
111        sem_signal(ocupadas);
112     }
113  }
```

En la función de la hebra consumidora utilizamos el semáforo con un wait, esto será porque si el valor de ocupadas es 0 deberemos esperar a que aumente. Esto ocurrirá cuando haya un valor en el buffer para ser consumido y la hebra podrá realizar su función.

```
117  void funcion_hebra_consumidora(  )
118  {
119     for( unsigned i = 0 ; i < num_items ; i++ )
120     {
121        int dato ;
122        sem_wait(ocupadas);
123        sem_wait(acceso);
124        dato = buffer[primera_libre-1];
125        primera_libre--;
126        sem_signal(acceso);
127        sem_signal(libres);
128        consumir_dato( dato ) ;
129     }
130  }
```

Para terminar, tenemos el semáforo “acceso”. Este semáforo nos ayudará a regular (como su propio nombre indica) el acceso al buffer, ya que debemos asegurarnos de que sea en exclusión mutua.

Esto ocurre porque solo puede acceder a él una hebra en cada instante de tiempo. Su valor inicial es 1 porque solo podrá llegar una hebra antes de que se bloquee.

El semáforo es utilizado tanto en la hebra productora como en la consumidora. El uso en ambas funciones es el mismo, comprobar que no haya ninguna otra hebra accediendo al vector. Con wait, si el valor es 1, accede al vector y pone el semáforo a 0 (lo bloquea). Cuando ha terminado de trabajar en el vector, aumenta el valor del semáforo mediante un signal (es decir, desbloquea el acceso al vector).

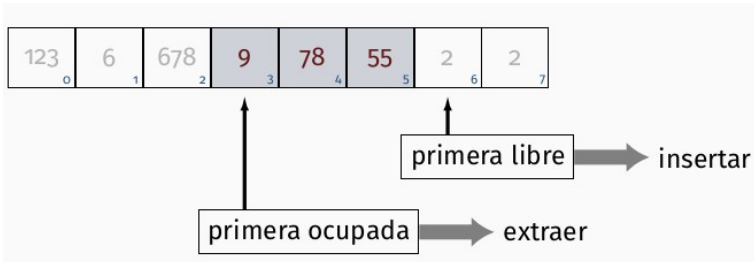
```
117  void funcion_hebra_consumidora(  )
118  {
119     for( unsigned i = 0 ; i < num_items ; i++ )
120     {
121        int dato ;
122        sem_wait(ocupadas);
123        sem_wait(acceso);
124        dato = buffer[primera_libre-1];
125        primera_libre--;
126        sem_signal(acceso);
127        sem_signal(libres);
128        consumir_dato( dato ) ;
129     }
130  }
```

```
101  void  funcion_hebra_productora(  )
102  {
103     for( unsigned i = 0 ; i < num_items ; i++ )
104     {
105        int dato = producir_dato() ;
106           sem_wait(libres);
107           sem_wait(acceso);
108           buffer[primera_libre] = dato;
109           primera_libre++;
110           sem_signal(acceso);
111           sem_signal(ocupadas);
112     }
113  }
```

Ejercicio 2: Múltiples productores y consumidores

De este ejercicio tenemos dos implementaciones, la primera con un buffer tipo LIFO y otra de tipo FIFO. Los ficheros son “prodcons-multi-lifo” y “prodcons-multi-fifo”, respectivamente.

Del vector tipo LIFO ya hemos mostrado su representación anteriormente. Esta es la del buffer tipo FIFO:



Mediante esta representación los primeros valores introducidos serán los primeros en ser consumidos.

La diferencia respecto al ejercicio anterior es que ahora en vez de haber una única hebra productora y una única hebra consumidora, podemos tener varias de ellas. Según la representación interna del buffer tendremos un número de semáforos u otro.

Mediante la implementación LIFO, como podemos observar, contamos con los mismos tres semáforos utilizados en el ejercicio anterior:

```
42  // Semáforos
43  Semaphore
44      libres      = tam_vec,
45      ocupadas    = 0      ,
46      acceso      = 1      ;
```

Los valores iniciales son los mismos, son utilizados en las mismas funciones y de la misma forma. Su uso es totalmente igual al descrito en el ejercicio anterior.

En la implementación FIFO debemos introducir un nuevo semáforo debido a que ahora tenemos dos zonas de acceso (donde se consumen los datos y donde se producen) y el acceso a ambas lo debemos controlar de forma independiente.

```
42  // Semáforos
43  Semaphore
44      libres      = tam_vec,
45      ocupadas    = 0      ,
46      acceso_prod = 1      ,
47      acceso_cons = 1      ;
```

El uso de “acceso_prod” y “acceso_cons” es el mismo que uso de “acceso” en el ejercicio anterior, solo que ahora cada semáforo se ocupa de una zona distinta. Uno en la zona de producir y otro en la zona de consumir.

```
124 void  funcion_hebra_productora(unsigned ih) // Cor
125 {
126    for( unsigned i = 0; i < p ; i++ )
127    {
128       int dato = producir_dato(ih) ;
129       sem_wait(libres);
130       sem_wait(acceso_prod);
131       buffer[primera_ocupada % tam_vec] = dato;
132       primera_ocupada++;
133       sem_signal(acceso_prod);
134       sem_signal(ocupadas);
135    }
136 }
```

```
140 void funcion_hebra_consumidora(unsigned ih) //
141 {
142    for( unsigned i = 0 ; i < c ; i++ )
143    {
144       int dato;
145       sem_wait(ocupadas);
146       sem_wait(acceso_cons);
147       dato = buffer[primera_libre % tam_vec];
148       primera_libre++;
149       sem_signal(libres);
150       sem_signal(acceso_cons);
151       consumir_dato( dato, ih ) ;
152    }
153 }
```

En la implementación LIFO nos basta con un único semáforo de acceso porque solo hay una zona de acceso al buffer.

Ejercicio 3: El problema de los fumadores

Este último ejercicio se corresponde con el archivo “fumadores.cpp”. En este ejercicio contamos con un número de semáforos variable, esto dependerá del número de fumadores que definamos para el problema.

```
34  // Semáforos
35  Semaphore mostr_vacio = 1;
36  Semaphore ingr_disp[num_fumadores] = {0, 0, 0};
```

En nuestro caso trabajaremos con una implementación con 3 fumadores. Por lo tanto contamos con el semáforo “mostr_vacio” y el array de semáforos “ingr_disp”.

“mostr_vacio” representa que el mostrador está vacío cuando vale 1 y que está lleno cuando vale 0. Empieza valiendo 1 porque el estanquero no ha producido todavía ningún ingrediente.

El array “ingr_disp” representa si está disponible el ingrediente que necesita cada fumador para poder fumar. Empieza inicializado en su totalidad a 0 porque inicialmente no hay ningún ingrediente producido.

```
67  void funcion_hebra_estanquero(  )
68  {
69     while (true){
70        // Obtenemos el indice del ingrediente producido
71        int ingrediente = producir_ingrediente();
72
73        // Comprobamos que el mostrador este vacio
74        sem_wait(mostr_vacio);
75
76        // Indicamos el ingrediente que hemos puesto en el mostrador
77        cout << "Puesto ingrediente " << ingrediente << endl;
78
79        // Aumentamos la disponibilidad de dicho ingrediente
80        sem_signal(ingr_disp[ingrediente]);
```

En la función del estanquero usamos “most_vacio” para comprobar si el mostrador está vacío, en ese caso podemos colocar el ingrediente generado.

Una vez hemos generado el ingrediente, lo indicamos en el array “ingr_disp” haciendo un signal de dicho ingrediente.

```
108 // función que ejecuta la hebra del fumador
109 void  funcion_hebra_fumador( int num_fumador )
110 {
111    while( true )
112    {
113       // Comprobar que haya disponibilidad del ingrediente
114       sem_wait(ingr_disp[num_fumador]);
115
116       // Informar del ingrediente retirado
117       cout << "Retirado ingrediente " << num_fumador << endl;
118
119       // Indicar que el mostrador está vacío
120       sem_signal(mostr_vacio);
121
122       // Fumar
123       fumar(num_fumador);
124    }
125 }
```

En cuanto a los semáforos en la hebra del fumador, primero usamos la orden wait para comprobar si está disponible el ingrediente que necesita el fumador. Una vez que hemos recogido el ingrediente liberamos el mostrador haciendo un signal de dicho semáforo. Esto permitirá al estanquero (hasta entonces bloqueado) producir un nuevo ingrediente.