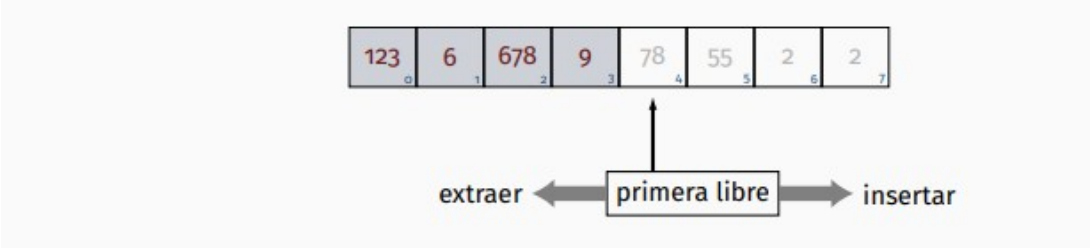
Entrega: Práctica 2 – SCD

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

Todos los ejercicios son realizados con semántica SU, es decir, Señalar y Espera Urgente.

Ejercicio 1: Múltiples productores y consumidores – LIFO

El fichero del ejecicio es “prodcons_mu_lifo.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.

```
110  // *****************************************************************************
111  // clase para monitor buffer, version FIFO, semántica SC, multiples prod/cons
112
113  class ProdConsSU1 : public HoareMonitor
114  {
115   private:
116   static const int           // constantes ('static' ya que no dependen de la instancia)
117     num_celdas_total = 10;   //   núm. de entradas del buffer
118   int                        // variables permanentes
119     buffer[num_celdas_total],//   buffer de tamaño fijo, con los datos
120     primera_libre ;          //   indice de celda de la próxima inserción ( == número de
121
122   CondVar                    // colas condicion:
123     ocupadas,                //  cola donde espera el consumidor (n>0)
124     libres ;                 //  cola donde espera el productor  (n<num_celdas_total)
125
126   public:                    // constructor y métodos públicos
127     ProdConsSU1() ;             // constructor
128     int  leer();                // extraer un valor (sentencia L) (consumidor)
129     void escribir( int valor ); // insertar un valor (sentencia E) (productor)
130  } ;
```

En la clase para el monitor destacamos:
- **int num_celdas_total:** será el tamaño del buffer donde escribirán y leerán los productores y los consumidores, respectivamente.
- **int buffer[num_celdas_total]:** el propio buffer.
- **CondVar ocupadas, libres:** colas de condición donde esperarán los consumidores y los productores, respectivamente, cuando se bloqueen.
- **métodos públicos:** los describimos a continuación.

```
133  ProdConsSU1::ProdConsSU1(  )
134  {
135     primera_libre = 0 ;
136     ocupadas      = newCondVar();
137     libres        = newCondVar();
138  }
```

Inicializamos los campos del monitor. La primera posición donde escribir será la 0 del buffer, e inicializamos las colas de condición.

```
139  // -----------------------------------------------------------------------------
140  // función llamada por el consumidor para extraer un dato
141
142  int ProdConsSU1::leer(  )
143  {
144     // esperar bloqueado hasta que 0 < primera_libre
145     if ( primera_libre == 0 )
146        ocupadas.wait();
147
148     //cout << "leer: ocup == " << primera_libre << ", total == " << num_celdas_total << endl ;
149     assert( 0 < primera_libre  );
150
151     // hacer la operación de lectura, actualizando estado del monitor
152     primera_libre-- ;
153     const int valor = buffer[primera_libre] ;
154
155     // señalar al productor que hay un hueco libre, por si está esperando
156     libres.signal();
157
158     // devolver valor
159     return valor ;
160  }
```

La función “leer” será usada por el consumidor para extraer un valor del buffer.
Primero comprobamos que la primera libre no sea 0, ya que esto implicaría que no hay elementos para leer y tendría que bloquearse la hebra consumidora.

Una vez hemos leído el valor, desbloqueamos una hebra productora con “libres.signal()”, en el caso de que haya alguna bloqueada.

```
163  void ProdConsSU1::escribir( int valor )
164  {
165     // esperar bloqueado hasta que primera_libre < num_celdas_total
166     if ( primera_libre == num_celdas_total )
167        libres.wait();
168
169     //cout << "escribir: ocup == " << primera_libre << ", total == " << num_celdas_total << end
170     assert( primera_libre < num_celdas_total );
171
172     // hacer la operación de inserción, actualizando estado del monitor
173     buffer[primera_libre] = valor ;
174     primera_libre++ ;
175
176     // señalar al consumidor que ya hay una celda ocupada (por si esta esperando)
177     ocupadas.signal();
178  }
```

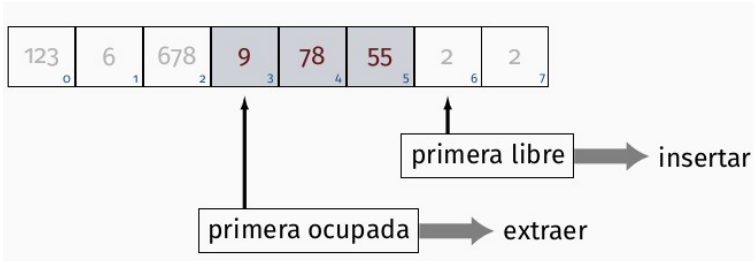
Esta función “escribir” es llamada por las hebras productoras para añadir un nuevo valor al buffer. Tras comprobar que haya espacios libres en el buffer para poder escribir, inserta un nuevo valor y desbloquea a una hebra consumidora (si es que hubiera alguna bloqueada).

```
182  void funcion_hebra_productora( MRef<ProdConsSU1> monitor, int ih) // C
183  {
184     for( unsigned i = 0 ; i < p ; i++ )
185     {
186        int valor = producir_dato( ih ) ;
187        monitor->escribir( valor );
188     }
189  }
190  // ----------------------------------------------------------------------
191
192  void funcion_hebra_consumidora( MRef<ProdConsSU1>  monitor, int ih) //
193  {
194     for( unsigned i = 0 ; i < c ; i++ )
195     {
196        int valor = monitor->leer();
197        consumir_dato( valor, ih ) ;
198     }
199  }
```

En cuanto a las funciones de las hebras: la productora producirá el dato que le corresponda y lo insertará en el buffer; la consumidora extraerá el próximo valor a ser consumido y lo mostrará por pantalla.

Ejercicio 2: Múltiples productores y consumidores – FIFO

El fichero del ejecicio es “prodcons_mu_fifo.cpp”.
La implementación del problema con el modelo FIFO es muy semejante al LIFO, por lo que solo comentaremos las diferencias entre ambos. La primera es el tipo de buffer, esta es la representación del buffer tipo FIFO:



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

```
120  int                         // variables permanentes
121    buffer[num_celdas_total],//   buffer de tamaño fijo, con los datos
122    primera_libre ,          //   indice de celda de la próxima inserción ( =
123    primera_ocupada;         //   indice de celda de la próxima extracción
```

En cuanto al monitor, podemos observar que hay un nuevo campo privado “primera_ocupada”, esto se debe a que ahora debemos tener dos índices, uno que nos indique dónde extraer y otro dónde insertar en el buffer.

```
55  // Numero de valores con los que cuenta el vector en cada instante
56  int num_valores = 0;
```

Aparece una variable globar nueva “num_valores” que representará la cantidad de valores que hay en cada instante en el buffer. Nos ayudará a sincronizar las hebras productoras y consumidoras.

```
155     // hacer la operación de lectura, actualizando estado del monitor
156     const int valor = buffer[primera_ocupada % num_celdas_total] ;
157     primera_ocupada++;
158     num_valores--;
```

```
177     // hacer la operación de inserción, actualizando estado del monitor
178     buffer[primera_libre % num_celdas_total] = valor ;
179     primera_libre++ ;
180     num_valores++;
```

Ahora los índices de insercción y extracción solo incrementan (al contrario que antes, que el único índice que había, podía también decrementar. Además, tanto en la insercción como en la extracción trabajamos en módulo para no salirnos de las casillas disponibles del buffer.

```
148     // esperar bloqueado hasta que 0 < primera_libre
149     if ( num_valores == 0 )
150        ocupadas.wait();
```

```
170     // esperar bloqueado hasta que primera_libre < num_celdas_total
171     if ( num_valores == num_celdas_total )
172        libres.wait();
```

En cuanto a “num_valores”, si estamos en la función de consumir y es igual a 0, bloqueamos la hebra en la cola de condición ya que no hay valores en el buffer. Si estamos en la función de producción y es igual al tamaño del buffer, bloqueamos la hebra productora ya que no hay espacio libre donde añadir el valor en el buffer.

Ejercicio 3: problema de los fumadores

```
80  // *****************************************************************************
81  // clase para monitor buffer, version FIFO, semántica SC, único prod/cons
82
83  class Estanco : public HoareMonitor
84  {
85   private:
86      int mostrador;          // el mostrador almacena el indice del ingrediente que contiene
87                              // si vale -1 implica que está vacío
88
89   CondVar                    // colas condicion:
90     estanquero,              //  cola donde espera el estanquero (mostrador lleno)
91     fumadores[NUM_FUMADORES] ;  //  array de colas donde esperan los fumadores (mostrador vacío)
92
93   public:                                // constructor y métodos públicos
94     Estanco() ;                          // constructor
95     void obtenerIngrediente ( int num_ingrediente ) ;
96     void ponerIngrediente ( int num_ingrediente ) ;
97     void esperarRecogidaIngrediente ( ) ;
98  } ;
```

El fichero del ejecicio es “fumadores_su.cpp”.
En cuanto al monitor del ejercicio:
- int mostrador: campo privado donde se almacena el índice del ingrediente que hay en el mostrador en cada instante. Es -1 si el mostrador está vacío.
- CondVar estanquero, fumadores[NUM_FUMADORES]: cola de condición para la hebra estanquero y un array de colas de condición, una para cada fumador.
- métodos públicos: los comentamos a continuación.

```
103   Estanco::Estanco(  )
104   {
105       mostrador = -1;                 // mostrador vacío
106       estanquero = newCondVar();
107
108       for(int i = 0; i < NUM_FUMADORES; i++)
109           fumadores[i]  = newCondVar();
110   }
```

Constructor para inicializar el monitor. Inicialmente el mostrador está vacío (vale -1) e inicializamos las colas de condición.

```
112   void Estanco::obtenerIngrediente( int num_ingrediente )
113   {
114       // si no está su ingrediente, se bloquea
115       if ( mostrador != num_ingrediente )
116           fumadores[num_ingrediente].wait();
117
118       // retirar ingrediente, mostrador vacío
119       mostrador = -1;
120
121       // Informar del ingrediente retirado
122       cout << "Retirado ingrediente " << num_ingrediente << endl;
123
124       // desbloquear al estanquero
125       estanquero.signal() ;
126   }
```

La función “obtenerIngrediente” será llamada por los fumadores para retirar el ingrediente que les falta para fumar. Primero se comprueba que el ingrediente del mostrador sea el que necesitan, si no, se bloquean.

Retiran el ingrediente, indican que el mostrador está vacío y desbloquean al estanquero para que produzca el siguiente ingrediente.

```
128   void Estanco::ponerIngrediente( int num_ingrediente )
129   {
130       // informa de que ha puesto un ingrediente
131       cout << "Estanquero : pone ingrediente " << num_ingrediente << " en el mostrador." << endl;
132
133       // poner ingrediente
134       mostrador = num_ingrediente;
135
136       // desbloquear al fumador
137       fumadores[num_ingrediente].signal();
138   }
```

La función “ponerIngrediente” es llamada por el estanquero para colocar un ingrediente en el mostrador. Una vez que lo ha colocado desbloquea al fumador que necesita dicho ingrediente.

```
140  void Estanco::esperarRecogidaIngrediente (  )
141  {
142      // bloquear al estanquero si el mostrador está lleno
143      if( mostrador != -1 ){
144          // informa de que va a bloquearse
145          cout << "Estanquero : espera recogida del ingrediente." << endl;
146          estanquero.wait() ;
147      }
148  }
```

El método “esperarRecogidaIngrediente” también es llamado por el estanquero. En él, si el mostrador está lleno, se bloquea el estanquero a la espera de que se recoja dicho ingrediente.

```
154  void funcion_hebra_fumador( MRef<Estanco> monitor, int num_fumador )
155  {
156      while( true )
157      {
158          // comprobar que el mostrador este lleno
159          monitor->obtenerIngrediente( num_fumador );
160
161          // fumar
162          fumar( num_fumador );
163      }
164  }
```

```
169  void funcion_hebra_estanquero( MRef<Estanco> monitor )
170  {
171      while ( true )
172      {
173          // obtener el indice del ingrediente
174          int ingrediente = ProducirIngrediente(  ) ;
175
176          monitor->ponerIngrediente( ingrediente ) ;
177          monitor->esperarRecogidaIngrediente(  ) ;
178      }
179  }
```

En cuanto a las hebras del estanquero y los fumadores observamos que los fumadores obtienen el ingrediente y una vez obtenido fuman (retraso aleatorio). Mientras que el estanquero crea un ingrediente aleatorio, lo cola en el mostrador y espera a que sea recogido.

Ejercicio 4: problema de los lectores-escritores

```
// *****************************************************************************
// clase para monitor buffer, version FIFO, semántica SU, multiples escritores / lectores

class Lec_Esc : public HoareMonitor
{
 private:
    int n_lec;
    bool escrib;

 CondVar                        // colas condicion:
   lectura,                      //  cola donde esperan los lectores
   escritura ;                   //  cola donde esperan los escritores

 public:                                    // constructor y métodos públicos
    Lec_Esc( ) ;

    void ini_lectura( int num_hebra ) ;
    void fin_lectura( int num_hebra ) ;

    void ini_escritura( int num_hebra ) ;
    void fin_escritura( int num_hebra ) ;
} ;
```

En cuanto al monitor del problema de los lectores y los escritores:
- **int n_lec:** número de lectores que hay leyendo en cada instante.
- **bool escrib:** variable lógica que vale true si algún escritor está escribiendo, false si ocurre lo contrario.
- **ConfVar lectura, escritura:** colas donde esperarán bloqueados los lectores y los escritores, respectivamente.
- **métodos públicos:** comentados a continuación.

```
56  Lec_Esc::Lec_Esc( )
57  {
58      n_lec = 0;
59      escrib = false;
60
61      // inicializar variables de condicion
62      lectura = newCondVar();
63      escritura = newCondVar();
64  }
```

Constructor que inicializa los valores del monitor. Inicialmente no hay lectores leyendo ni escritores escribiendo. Se inicializan las colas de condición.

```
69
70  void Lec_Esc::ini_lectura ( int num_hebra )
71  {
72      if ( escrib )
73          lectura.wait();
74
75      n_lec += 1;
76
77      cout << "Lector "<< num_hebra << " : Entra un nuevo lector. Ahora hay " << n_lec << endl;
78
79      lectura.signal( );
80  }
```

La función “ini_lectura” es usada por los lectores para iniciar la lectura. Primero comprueba si hay algún escritor escribiendo, si es así se bloquea; si no, aumenta en uno el número de lectores.

```
81
82  void Lec_Esc::fin_lectura ( int num_hebra )
83  {
84      n_lec -= 1;
85
86      cout << "Lector " << num_hebra << " : Termina un lector. Ahora hay " << n_lec << endl;
87
88      if ( n_lec == 0 )
89          escritura.signal( );
90  }
```

En cuanto a “fin_lectura”, también empleada por los lectores, se decrementa el número de lectores y si este es 0 se desbloquea una hebra de escritores.

```
92  void Lec_Esc::ini_escritura ( int num_hebra )
93  {
94      if ( n_lec > 0 || escrib )
95          escritura.wait();
96
97      cout << "Escritor " << num_hebra << " : Se empieza a escribir." << endl;
98
99      escrib = true;
100 }
```

La función “ini_escritura” la emplean los escritores para empezar a escribir. Inicialmente comprueba si hay algún lector o algún otro escritor activo, ya que la escritura debe ejecutarse de forma aislada, sin ningún otro proceso activo en dicho momento.

```
102  void Lec_Esc::fin_escritura ( int num_hebra )
103  {
104      escrib = false;
105
106      cout << "Escritor " << num_hebra << " : Se termina de escribir." << endl;
107
108      if ( !lectura.empty() )
109          lectura.signal();
110
111      else
112          escritura.signal();
113  }
```

En cuanto a “fin_escritura”, es empleada por los escritores cuando finalizan la escritura. Primero se indica que ya no hay ningún proceso escribiendo y luego se comprueba si hay algún proceso de lectura esperando, si es así, lo desbloquea. Si no ocurre lo anterior, desbloquea a un nuevo proceso de escritura.

En cuanto a las funciones de las hebras:

```
115  //----------------------------------------------------------------------
116  // función que ejecuta la hebra del lector
117
118  void funcion_hebra_lector( MRef<Lec_Esc> monitor, int num_hebra)
119  {
120      while( true )
121      {
122          // iniciar lectura
123          monitor->ini_lectura( num_hebra );
124
125          // calcular milisegundos aleatorios de duración de la acción de "leer" y bloquear
126          chrono::milliseconds duracion_leer( aleatorio<20,200>() );
127
128          // espera bloqueada un tiempo igual a 'duracion_leer' milisegundos
129          this_thread::sleep_for( duracion_leer );
130
131          // terminar lectura
132          monitor->fin_lectura( num_hebra );
133
134          // calcular milisegundos aleatorios de duración de la acción de "resto de codigo"
135          chrono::milliseconds duracion_resto_codigo( aleatorio<20,200>() );
136
137          // espera bloqueada un tiempo igual a 'duracion_escribir' milisegundos
138          this_thread::sleep_for( duracion_resto_codigo );
139      }
140  }
```

En las hebras de los lectores se inicia la lectura, se crea un retraso aleatorio simulando dicha lectura, luego se finaliza la lectura y finalmente se genera un nuevo retraso aleatorio simulando más partes de código.

```cpp
//----------------------------------------------------------------------
// función que ejecuta la hebra del escritor

void funcion_hebra_escritor( MRef<Lec_Esc> monitor, int num_hebra )
{
    while( true )
    {
        // iniciar escritura
        monitor->ini_escritura( num_hebra );

        // calcular milisegundos aleatorios de duración de la acción de "escribir"
        chrono::milliseconds duracion_escribir( aleatorio<20,200>() );

        // espera bloqueada un tiempo igual a 'duracion_escribir' milisegundos
        this_thread::sleep_for( duracion_escribir );

        // terminar lectura
        monitor->fin_escritura( num_hebra );

        // calcular milisegundos aleatorios de duración de la acción de "resto de codigo"
        chrono::milliseconds duracion_resto_codigo( aleatorio<20,200>() );

        // espera bloqueada un tiempo igual a 'duracion_escribir' milisegundos
        this_thread::sleep_for( duracion_resto_codigo );
    }
}
```

Las hebras de los escritores funcionan de forma semejante a la de lectores, salvo que inician y finalizan la escritura en vez de la lectura, como no podía ser de otra forma.

Como detalle adición a la implementación de la práctica, se pasan los índices de las hebras a las funciones que ejecutan. De esta forma podemos indicar por pantalla qué número de hebra está realizando cada acción.