

PORTAFOLIO SCD:

Seminario 1:

Ejercicio integral concurrente: (ejemplo09-plantilla.cpp)

```
// -----  
// función que ejecuta cada hebra: recibe $i$ ==índice de la hebra, ($0\leq i<n$)  
double funcion_hebra( long i )  
{  
    double suma = 0.0;  
    for(int j=i*c; j < (i+1)*c; j++){          //Cada hebra calcula el área de c muestras donde c es m/n  
        const double xj = double(j+0.5)/m;  
        suma += f(xj);  
    }  
  
    return suma;                               //Se devuelve el área promedio de todas las muestras calculadas por una hebra  
}  
  
// -----  
// calculo de la integral de forma concurrente  
double calcular_integral_concurrente( )  
{  
    double suma = 0.0;  
    future<double> futuro[n];  
  
    for(int i=0; i < n; i++){  
        futuro[i] = async(launch::async, funcion_hebra, i);  
    }  
  
    for(int i=0; i < n; i++){  
        suma += futuro[i].get();  
    }  
  
    return suma/m;                             //Se devuelve el promedio de los promedios calculados por cada area  
}  
// -----
```

Se añade en la función hebra código para que cada hebra calcule la suma de c valores x_j donde c es el número de muestras entre el número de hebras. Finalmente en la función `calcular_integral_concurrente` lanzo todas las hebras y divido la suma de los resultados de todas las hebras entre el número total de muestras para obtener el valor promedio.

(ejemplo11.cpp) Se pide que se razone porque en el ejemplo 11.cpp

```
10 // -----  
11  
12 #include <iostream>  
13 #include <thread>  
14 #include <chrono>  
15 #include <atomic> // incluye la funcionalidad para tipos atomicos  
16 #include <mutex> // incluye funcionalidad para 'mutex'  
17 using namespace std ;  
18 using namespace std::chrono ;  
19  
20 const long num_iters = 1000000l ;  
21 int contador_no_atom ; // contador compartido (no atomico)  
22 atomic<int> contador_atom ; // contador compartido (atomico)  
23 int contador_mutex ;  
24  
25 mutex mtx ;  
26  
27 void funcion_hebra_no_atom( )  
28 { for( long i = 0 ; i < num_iters ; i++ )  
29     contador_no_atom ++ ; // incremento no atómico de la variable  
30 }  
31 void funcion_hebra_atom( )  
32 { for( long i = 0 ; i < num_iters ; i++ )  
33     contador_atom ++ ; // incremento atómico de la variable  
34 }  
35 void funcion_hebra_mutex( )  
36 { for( long i = 0 ; i < num_iters ; i++ )  
37 {  
38     mtx.lock();  
39     contador_mutex ++ ; // incremento atómico de la variable  
40     mtx.unlock();  
41 }  
42 }
```

Este ejercicio hace un contador una vez con dos hebras que hacen el incremento de forma atómica y la segunda con hebras que hacen el incremento de forma no atómica. Se observa que el tiempo de la hebra que hace el incremento con mutex y con atomic es mayor que el no atomico. Esto se debe a que la sincronización con mutex contempla la realización de operaciones más complejas que incrementos, decrementos e igualaciones atómicas, por lo tanto, se tiene que poder garantizar una sincronización que permita estas operaciones complejas lo cual conlleva tiempo.

```
daniel@daniel-XPS-15-9570:~/Desktop/DANIEL/SCD/Seminario_1/scd-s1-fuentes (1)$ ./ejemplo11
valor esperado      : 2000000
resultado (mutex)    : 2000000
resultado (atom.)    : 2000000
resultado (no atom.) : 1077429
tiempo mutex         : 160.582 milisegundos
tiempo atom.         : 111.645 milisegundos.
tiempo no atom.      : 15.8686 milisegundos.
```

Practica 1:

Prodcons-lifo:

Variables globales del programa

```
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

int vec[tam_vec]; //Buffer por el que se intercambian datos entre el productor y el consumidor.
int primera_libre=0; //Primera posición del buffer libre
Semaphore ocupadas=0, libres=tam_vec, exclusion=1; //El semáforo exclusion asegura que cuando se modifique primera libre y vec se haga en exclusión mutua
//Libres permite que la productora empiece a generar datos y que pueda seguir generandolos hasta al menos tam_vec
//Ocupadas obliga a la hebra consumidora a esperar hasta que se haya producido al menos un dato
```

Código de las funciones correspondientes a la ejecutada por la hebra consumidora y por la hebra productora. Como se puede apreciar ambas tienen que recibir permiso del semaforo exclusion para poder modificar vec y primera libre, lo que asegura que las modificaciones de estos dos ultimos se hace en exclusión mutua.

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

        sem_wait(libres);
        sem_wait(exclusion);
        vec[primera_libre] = dato;
        primera_libre++;

        sem_signal(exclusion);
        sem_signal(ocupadas);
    }
}

//-----

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

        sem_wait(ocupadas);
        sem_wait(exclusion);

        dato=vec[primera_libre-1];
        primera_libre--;

        sem_signal(exclusion);
        sem_signal(libres);

        consumir_dato( dato ) ;
    }
}
```

Prodcons-fifo:

```
11 //*****
12 // variables compartidas
13
14 const int num_items = 40 , // número de items
15       tam_vec = 10 ; // tamaño del buffer
16 unsigned cont_prod[num_items] = {0}, // contadores de verificación: producidos
17       cont_cons[num_items] = {0}; // contadores de verificación: consumidos
18
19 int vec[tam_vec]; //Buffer por el que se intercambian datos entre el productor y el consumidor.
20 int primera_libre=0; //Primera posición del buffer libre la aumenta la hebra productora
21 int primera_ocupada=0; //La aumenta la hebra consumidora
22 Semaphore ocupadas=0, libres=tam_vec, exclusion=1; //El semáforo exclusión asegura que cuando se modifique primera_libre y vec se haga en exclusión mutua
23 //Libres permite que la productora empiece a generar datos y que pueda seguir generandolos hasta al menos tam_vec
24 //Ocupadas obliga a la hebra consumidora a esperar hasta que se haya producido al menos un dato
25
```

Variables compartidas de prodcons-fifo (iguales a prodcons-lifo añadiendo una variable entera `primera_ocupada` para asegurar que se siga una estructura fifo).

```
89 void funcion_hebra_productora( )
90 {
91     for( unsigned i = 0 ; i < num_items ; i++ )
92     {
93         int dato = producir_dato() ;
94
95         sem_wait(libres);
96         sem_wait(exclusion);
97         vec[primera_libre] = dato;
98         primera_libre = (primera_libre+1)%tam_vec;
99
100        sem_signal(exclusion);
101        sem_signal(ocupadas);
102    }
103 }
104
105 //-----
106
107 void funcion_hebra_consumidora( )
108 {
109     for( unsigned i = 0 ; i < num_items ; i++ )
110     {
111         int dato;
112
113         sem_wait(ocupadas);
114         sem_wait(exclusion);
115
116         dato=vec[primera_ocupada];
117         primera_ocupada = (primera_ocupada+1)%tam_vec;
118
119         sem_signal(exclusion);
120         sem_signal(libres);
121
122         consumir_dato( dato ) ;
123     }
124 }
125
```

Código de las funciones que se ejecutan por las hebra consumidora y la productora. Al igual que en el caso lifo el semáforo `exclusion` asegura la exclusión mútua en el acceso al buffer y a las variables

primera libre y primera ocupada (Realmente no es necesario acceder en exclusión mutua a primera libre y primera ocupada, pues a estas acceden unicamente una hebra para cada).

Problema de los fumadores:

```
12 const int num_fumadores = 3; //Numero de fumadores que participan en el programa
13 Semaphore mostrador_vacio=1; //Semaforo que controla la ocupación del mostrador (1 ingrediente máximo)
14 Semaphore ingrediente_puesto[num_fumadores] = {0,0,0}; //Semaforos para controlar la espera de los fumadores hasta que su ingrediente esta dispuesto
15
```

Tenemos una variables num_fumadores, que es el número de fumadores del programa, mostrador_vacio que es un semaforo que asegura que al mostrador pueda acceder un máximo de una hebra en un instante. Ingrediente puesto es un vector de fumadores, uno por fumador, que se encargar de gestionar las esperar de los fumadores hasta que su ingrediente este puesto en el mostrador.

```
// función que ejecuta la hebra del estanquero

void funcion_hebra_estanquero( )
{
    int tabaco;

    while (true){
        tabaco = producir_ingrediente();

        sem_wait(mostrador_vacio);
        cout << "Puesto ingrediente " << tabaco << endl;
        sem_signal(ingrediente_puesto[tabaco]);
    }
}
```

Funcion que ejecuta el estanquero, tras producir un ingrediente se comprueba si el mostrador esta vacío, si lo está lo coloca y avisa al fumador correspondiente. Si no lo está espera hasta que lo esté.

```
// función que ejecuta la hebra del fumador
void funcion_hebra_fumador( int num_fumador )
{
    while( true )
    {
        sem_wait(ingrediente_puesto[num_fumador]);
        cout << "Retirado ingrediente: " << num_fumador << endl;
        sem_signal(mostrador_vacio);

        fumar(num_fumador);
    }
}
```

Función que ejecuta el fumador. Espera en cola hasta que el estanquero le comunica que su ingrediente está en el mostrador. Cuando se lo comunica lo retira y avisa al estanquero de que ha dejado el mostrador vacío para que coloque el siguiente ingrediente.

Seminario 2:

Prodcons1ifo_msc:

```
119 class ProdCons1SC
120 {
121     private:
122         static const int          // constantes:
123             num_celdas_total = 10; // núm. de entradas del buffer
124         int                      // variables permanentes
125             buffer[num_celdas_total], // buffer de tamaño fijo, con los datos
126             primera_libre ;           // índice de celda de la próxima inserción
127         mutex
128             cerrojo_monitor ;        // cerrojo del monitor
129         condition_variable         // colas condicion:
130             ocupadas,               // cola donde espera el consumidor (n>0)
131             libres ;                // cola donde espera el productor (n<num_celdas_total)
132     public:                          // constructor y métodos públicos
133         ProdCons1SC( ) ;            // constructor
134         int leer();                  // extraer un valor (sentencia L) (consumidor)
135         void escribir( int valor ); // insertar un valor (sentencia E) (productor)
136     } ;
137 // -----
138
139 ProdCons1SC::ProdCons1SC( )
140 {
141     primera_libre = 0 ;
142 }
143 }
```

```
147 int ProdCons1SC::leer( )
148 {
149     // ganar la exclusión mutua del monitor con una guarda
150     unique_lock<mutex> guarda( cerrojo_monitor );
151
152     // esperar bloqueado hasta que 0 < num_celdas_ocupadas
153     while ( primera_libre == 0 )
154         ocupadas.wait( guarda );
155
156     // hacer la operación de lectura, actualizando estado del monitor
157     assert( 0 < primera_libre );
158     primera_libre-- ;
159     const int valor = buffer[primera_libre] ;
160
161     // señalar al productor que hay un hueco libre, por si está esperando
162     libres.notify_one();
163
164     // devolver valor
165     return valor ;
166 }
167 // -----
168
169 void ProdCons1SC::escribir( int valor )
170 {
171     // ganar la exclusión mutua del monitor con una guarda
172     unique_lock<mutex> guarda( cerrojo_monitor );
173
174     // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total
175     while( primera_libre == num_celdas_total )
176         libres.wait( guarda );
177
178     //cout << "escribir: ocup == " << num_celdas_ocupadas << ", total == " << num_celdas_total << endl ;
179     assert( primera_libre < num_celdas_total );
180
181     // hacer la operación de inserción, actualizando estado del monitor
182     buffer[primera_libre] = valor ;
183     primera_libre++ ;
184
185     // señalar al consumidor que ya hay una celda ocupada (por si esta esperando)
186     ocupadas.notify_one();
187 }
188 }
```

```

192 void funcion_hebra_productora( ProdCons1SC * monitor, int num_productora )
193 {
194     for( unsigned i = (num_items/num_productoras)*num_productora; i < (num_items/num_productoras)*(num_productora+1) ; i++ )
195     {
196         int valor = producir_dato(num_productora) ;
197         monitor->escribir( valor );
198     }
199 }
200 // -----
201
202 void funcion_hebra_consumidora( ProdCons1SC * monitor, int num_consumidor)
203 {
204     for( unsigned i = (num_items/num_consumidoras)*num_consumidor ; i < (num_items/num_consumidoras)*(num_consumidor+1) ; i++ )
205     {
206         int valor = monitor->leer();
207         consumir_dato( valor, num_consumidor ) ;
208     }
209 }
210 // -----
211
212 int main()
213 {
214     cout << "-----" << endl
215         << "Problema de los productores-consumidores (1 prod/cons, Monitor SU, buffer LIFO). " << endl
216         << "-----" << endl
217         << flush ;
218
219     ProdCons1SC monitor;;
220
221     thread hebra_productora[num_productoras],
222           hebra_consumidora[num_consumidoras];
223
224     for(int i=0; i < num_productoras; i++){
225         hebra_productora[i]=thread(funcion_hebra_productora, &monitor, i);
226     }
227     for(int i=0; i < num_consumidoras; i++){
228         hebra_consumidora[i] = thread(funcion_hebra_consumidora, &monitor, i);
229     }
230
231     for(int i=0; i < num_productoras; i++){
232         hebra_productora[i].join();
233     }
234
235     for(int i=0; i < num_consumidoras; i++){
236         hebra_consumidora[i].join();
237     }

```

Prodconsfifo_msc:

```

119 class ProdConsISC
120 {
121     private:
122         static const int          // constantes:
123             num_celdas_total = 10; // núm. de entradas del buffer
124         int                      // variables permanentes
125             buffer[num_celdas_total], // buffer de tamaño fijo, con los datos
126             primera_libre,           // índice de celda de la próxima inserción
127             primera_ocupada, num_celdas_ocupadas;
128         mutex cerrojo_monitor ;      // cerrojo del monitor
129         condition_variable          // colas condicion:
130             ocupadas,              // cola donde espera el consumidor (n>0)
131             libres ;               // cola donde espera el productor (n<num_celdas_total)
132
133     public:                        // constructor y métodos públicos
134         ProdConsISC( ) ;          // constructor
135         int leer();                // extraer un valor (sentencia L) (consumidor)
136         void escribir( int valor ); // insertar un valor (sentencia E) (productor)
137     };
138 // -----
139 ProdConsISC::ProdConsISC( )
140 {
141     primera_libre = 0 ;
142     primera_ocupada = 0;
143     num_celdas_ocupadas=0;
144 }
145 // -----
146 // función llamada por el consumidor para extraer un dato
147 int ProdConsISC::leer( )
148 {
149     // ganar la exclusión mutua del monitor con una guarda
150     unique_lock<mutex> guarda( cerrojo_monitor );
151     // esperar bloqueado hasta que 0 < num_celdas_ocupadas
152     while( num_celdas_ocupadas == 0 )
153         ocupadas.wait( guarda );
154     // hacer la operación de lectura, actualizando estado del monitor
155     assert( 0 < num_celdas_ocupadas );
156     const int valor = buffer[primera_ocupada] ;
157     primera_ocupada = (primera_ocupada+1)%num_celdas_total;
158     num_celdas_ocupadas--;
159     // señalar al productor que hay un hueco libre, por si está esperando
160     libres.notify_one();
161     // devolver valor
162     return valor ;
163 }
164 // -----

```

```

166 void ProdCons1SC::escribir( int valor )
167 {
168     // ganar la exclusión mutua del monitor con una guarda
169     unique_lock<mutex> guarda( cerrojo_monitor );
170
171     // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total
172     while ( num_celdas_ocupadas == num_celdas_total )
173         libres.wait( guarda );
174
175     //cout << "escribir: ocup == " << num_celdas_ocupadas << ", total == " << num_celdas_total << endl ;
176     assert( num_celdas_ocupadas < num_celdas_total );
177
178     // hacer la operación de inserción, actualizando estado del monitor
179     buffer[primera_libre] = valor ;
180     primera_libre = (primera_libre+1)%num_celdas_total ;
181     num_celdas_ocupadas++;
182
183     // señalar al consumidor que ya hay una celda ocupada (por si esta esperando)
184     ocupadas.notify_one();
185 }
186 // *****
187 // funciones de hebras
188
189 void funcion_hebra_productora( ProdCons1SC * monitor, int num_productora )
190 {
191     for( unsigned i = (num_items/num_productoras)*num_productora; i < (num_items/num_productoras)*(num_productora+1) ; i++ )
192     {
193         int valor = producir_dato(num_productora) ;
194         monitor->escribir( valor );
195     }
196 }
197 // -----
198
199 void funcion_hebra_consumidora( ProdCons1SC * monitor, int num_consumidor )
200 {
201     for( unsigned i = (num_items/num_consumidoras)*num_consumidor ; i < (num_items/num_consumidoras)*(num_consumidor+1) ; i++ )
202     {
203         int valor = monitor->leer();
204         consumir_dato( valor, num_consumidor ) ;
205     }
206 }
207 // -----

```

En estos dos programas se tienen múltiples hebras productoras y consumidoras. Además la sincronización se hace por monitores SC y no por semáforos. La principal diferencia es que la versión LIFO sólo necesita de la variable `primera_libre` para meter y sacar datos del buffer mientras que la versión FIFO necesita `primera_libre` y `primera_ocupada`. Además la forma de actualizar estas variables y las condiciones en los `while` son distintas. El `main` es común en ambas versiones

Prodconsfifo_msu:


```

119 class ProdConsSU : public HoareMonitor
120 {
121 private:
122 static const int          // constantes:
123     num_celdas_total = 10; // núm. de entradas del buffer
124 int                    // variables permanentes
125     buffer[num_celdas_total], // buffer de tamaño fijo, con los datos
126     primera_libre,           // índice de celda de la próxima inserción
127     primera_ocupada,
128     num_celdas_ocupadas;
129 CondVar                // colas condicion:
130     ocupadas,           // cola donde espera el consumidor (n>0)
131     libres ;            // cola donde espera el productor (n<num_celdas_total)
132
133 public:                  // constructor y métodos públicos
134     ProdConsSU( ) ;      // constructor
135     int leer();           // extraer un valor (sentencia L) (consumidor)
136     void escribir( int valor ); // insertar un valor (sentencia E) (productor)
137 } ;
138 // -----
139
140 ProdConsSU::ProdConsSU( )
141 {
142     primera_libre = 0 ;
143     primera_ocupada = 0;
144     num_celdas_ocupadas=0;
145     libres=newCondVar();
146     ocupadas=newCondVar();
147 }

```

```

151 int ProdConsSU::leer( )
152 {
153
154     // esperar bloqueado hasta que 0 < num_celdas_ocupadas
155     if ( num_celdas_ocupadas == 0 )
156         ocupadas.wait();
157
158     // hacer la operación de lectura, actualizando estado del monitor
159     assert( 0 < num_celdas_ocupadas );
160     const int valor = buffer[primera_ocupada] ;
161     primera_ocupada = (primera_ocupada+1)%num_celdas_total;
162     num_celdas_ocupadas--;
163
164
165     // señalar al productor que hay un hueco libre, por si está esperando
166     libres.signal();
167
168     // devolver valor
169     return valor ;
170 }
171 // -----
172
173 void ProdConsSU::escribir( int valor )
174 {
175
176     // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total
177     if ( num_celdas_ocupadas == num_celdas_total )
178         libres.wait();
179
180     //cout << "escribir: ocup == " << num_celdas_ocupadas << ", total == " << num_celdas_total << endl ;
181     assert( num_celdas_ocupadas < num_celdas_total );
182
183     // hacer la operación de inserción, actualizando estado del monitor
184     buffer[primera_libre] = valor ;
185     primera_libre = (primera_libre+1)%num_celdas_total ;
186     num_celdas_ocupadas++;
187
188     // señalar al consumidor que ya hay una celda ocupada (por si esta esperando)
189     ocupadas.signal();
190 }

```

```

194 void funcion_hebra_productora( MRef<ProdConsSU> monitor, int num_productora )
195 {
196     for( unsigned i = (num_items/num_productoras)*num_productora; i < (num_items/num_productoras)*(num_productora+1) ; i++ )
197     {
198         int valor = producir_dato(num_productora) ;
199         monitor->escribir( valor );
200     }
201 }
202 // -----
203
204 void funcion_hebra_consumidora( MRef<ProdConsSU> monitor, int num_consumidor)
205 {
206     for( unsigned i = (num_items/num_consumidoras)*num_consumidor ; i < (num_items/num_consumidoras)*(num_consumidor+1) ; i++ )
207     {
208         int valor = monitor->leer();
209         consumir_dato( valor, num_consumidor ) ;
210     }
211 }
212 // -----
213
214 int main()
215 {
216     cout << "-----" << endl
217         << "Problema de los productores-consumidores (1 prod/cons, Monitor SU, buffer LIFO). " << endl
218         << "-----" << endl
219         << flush ;
220
221     MRef<ProdConsSU> monitor = Create<ProdConsSU>();
222
223     thread hebra_productora[num_productoras],
224           hebra_consumidora[num_consumidoras];
225
226     for(int i=0; i < num_productoras; i++){
227         hebra_productora[i]=thread(funcion_hebra_productora, monitor, i);
228     }
229     for(int i=0; i < num_consumidoras; i++){
230         hebra_consumidora[i] = thread(funcion_hebra_consumidora, monitor, i);
231     }
232
233     for(int i=0; i < num_productoras; i++){
234         hebra_productora[i].join();
235     }
236
237     for(int i=0; i < num_consumidoras; i++){
238         hebra_consumidora[i].join();
239     }
240

```

Prodconslifo_msu:

```

119 class ProdConsSU : public HoareMonitor
120 {
121     private:
122         static const int          // constantes:
123             num_celdas_total = 10; // núm. de entradas del buffer
124         int                    // variables permanentes
125             buffer[num_celdas_total], // buffer de tamaño fijo, con los datos
126             primera_libre ;           // índice de celda de la próxima inserción
127         CondVar                // colas condicion:
128             ocupadas,           // cola donde espera el consumidor (n>0)
129             libres ;            // cola donde espera el productor (n<num_celdas_total)
130
131     public:                    // constructor y métodos públicos
132         ProdConsSU( ) ;        // constructor
133         int leer();             // extraer un valor (sentencia L) (consumidor)
134         void escribir( int valor ); // insertar un valor (sentencia E) (productor)
135 } ;
136 // -----
137
138 ProdConsSU::ProdConsSU( )
139 {
140     primera_libre = 0 ;
141     ocupadas=newCondVar();
142     libres=newCondVar();
143 }
144 // -----
145 // función llamada por el consumidor para extraer un dato
146
147 int ProdConsSU::leer( )
148 {
149
150     // esperar bloqueado hasta que 0 < num_celdas_ocupadas
151     if ( primera_libre == 0 )
152         ocupadas.wait();
153
154     // hacer la operación de lectura, actualizando estado del monitor
155     assert( 0 < primera_libre );
156     primera_libre-- ;
157     const int valor = buffer[primera_libre] ;
158
159
160     // señalar al productor que hay un hueco libre, por si está esperando
161     libres.signal();
162
163     // devolver valor
164     return valor ;
165 }

```

```

168 void ProdConsSU::escribir( int valor )
169 {
170
171     // esperar bloqueado hasta que num_celdas_ocupadas < num_celdas_total
172     if ( primera_libre == num_celdas_total )
173         libres.wait();
174
175     //cout << "escribir: ocup == " << num_celdas_ocupadas << ", total == " << num_celdas_total << endl ;
176     assert( primera_libre < num_celdas_total );
177
178     // hacer la operación de inserción, actualizando estado del monitor
179     buffer[primera_libre] = valor ;
180     primera_libre++ ;
181
182     // señalar al consumidor que ya hay una celda ocupada (por si esta esperando)
183     ocupadas.signal();
184 }
185 // *****
186 // funciones de hebras
187
188 void funcion_hebra_productora( MRef<ProdConsSU> monitor, int num_productora )
189 {
190     for( unsigned i = (num_items/num_productoras)*num_productora; i < (num_items/num_productoras)*(num_productora+1) ; i++ )
191     {
192         int valor = producir_dato(num_productora) ;
193         monitor->escribir( valor ) ;
194     }
195 }
196 // -----
197
198 void funcion_hebra_consumidora( MRef<ProdConsSU> monitor, int num_consumidor )
199 {
200     for( unsigned i = (num_items/num_consumidoras)*num_consumidor ; i < (num_items/num_consumidoras)*(num_consumidor+1) ; i++ )
201     {
202         int valor = monitor->leer();
203         consumir_dato( valor, num_consumidor ) ;
204     }
205 }
206 // -----

```

Al igual que para el caso con monitores SC, la única diferencia se da en que la versión FIFO requiere de una variable primera_ocupada y que se actualizan de forma distinta a las de la versión LIFO.

Problema_fumadores_msu:


```

28 class Estanco : public HoareMonitor
29 {
30     private:
31         int mostrador;
32         CondVar estanquero;           //Cola en la que espera el estanquero
33         CondVar fumadores[num_fumadores]; //Colas en la que esperan los fumadores, cada uno la suya
34
35     public:
36         Estanco(){
37             mostrador = -1;
38             estanquero = newCondVar();
39
40             for(int i=0; i < num_fumadores;i++){
41                 fumadores[i]=newCondVar();
42             }
43         }
44
45         void ponerIngrediente(int ingre){
46             cout << "Estanquero pone ingrediente " << ingre << " en mostrador." << endl; //El estanquero pone un ingrediente y actualiza el mostrador
47             mostrador = ingre;
48
49             fumadores[ingre].signal(); //Avisa al fumador correspondiente para que recoja su tabaco
50         }
51
52         void esperarRecogidaIngrediente(){
53             if(mostrador != -1){ //Si el mostrador no está vacío espera a que se recoja el ingrediente, sino continua
54                 estanquero.wait();
55             }
56         }
57
58         void obtenerIngrediente(int i){
59             if(mostrador != i){ //Si su ingrediente no esta en mostrador el fumador espera
60                 fumadores[i].wait();
61             }
62
63             cout << "Fumador recoge ingrediente " << i << " de mostrador." << endl;
64
65             mostrador=-1; //Si lo está lo recoge y pone el mostrador vacío
66
67             if(!estanquero.empty()) //SI el estanquero está esperando es su cola lo despierta para que produzca más tabaco sino continua
68                 estanquero.signal();
69         }
70 };

```

```

98 void funcion_hebra_estanquero( MRef<Estanco> monitor)
99 {
100
101     while (true){
102         int ingre = producir_ingrediente();
103
104         monitor->ponerIngrediente(ingre);
105         monitor->esperarRecogidaIngrediente();
106     }
107 }

```

```

132 //-----
133 // función que ejecuta la hebra del fumador
134 void funcion_hebra_fumador( int num_fumador, MRef<Estanco> monitor )
135 {
136     while( true )
137     {
138         monitor->obtenerIngrediente(num_fumador);
139         fumar(num_fumador);
140     }
141 }

```

Como variable permanente tenemos mostrados, que almacena el número del ingrediente en el mostrador o -1 si este está vacío.

Lectores escritores:

```
23 class Lec_Esc : public HoareMonitor
24 {
25     private:
26         int n_lec;        //número de lectores leyendo
27         bool escrib;      //true si hay algún escritor escribiendo
28         CondVar lectura;
29         CondVar escritura;
30
31     public:
32         Lec_Esc(){
33             n_lec=0;
34             escrib=false;
35             lectura=newCondVar();
36             escritura=newCondVar();
37         }
38
39         //Funciones ejecutadas por los lectores
40
41         void ini_lectura(){
42             if(escrib)          //Si hay alguien escribiendo se espera
43                 lectura.wait();
44
45             n_lec++;            //Una vez se deje de escribir se aumenta el número de lectores en el momento y
46                                 //se avisa a otras hebras para que lean concurrentemente
47             lectura.signal();
48         }
49
50         void fin_lectura(){
51             n_lec--;            //Una vez termina de leer se disminuye el número de lectores en el momento
52
53             if(n_lec == 0){     //Si no hay nadie que siga leyendo entonces se avisa al escritor para que siga escribiendo
54                 escritura.signal();
55             }
56         }
57
58         //Funciones ejecutadas por los escritores
59         void ini_escritura(){
60             if (n_lec > 0 || escrib)    //Si hay alguien leyendo o escribiendo se espera a que termine
61                 escritura.wait();
62
63             escrib=true;                //Una vez no haya nadie leyendo o escribiendo se empieza a escribir
64         }
65
66         void fin_escritura(){
67             escrib=false;                //Se indica a todas las hebras que ya no se está escribiendo
68
69             if(!lectura.empty())        //Si hay algún lector esperando se le indica que puede leer
70                 lectura.signal();
71
72             else                        //Si no lo hay se indica al siguiente escritor que puede escribir (preferencia lectores)
73                 escritura.signal();
74         }
75     };
76
77     void lee(int num_lector){
78         // calcular milisegundos aleatorios de duración de la acción de fumar
79         chrono::milliseconds duracion_leer( aleatorio<10,100>() );
80
81         cout << "Lector " << num_lector << " empieza a leer." << endl << flush;
82
83         this_thread::sleep_for( duracion_leer );
84     }
85 }
86
87     void escribir(int num_escritor){
88         // calcular milisegundos aleatorios de duración de la acción de fumar
89         chrono::milliseconds duracion_escribir( aleatorio<20,200>() );
90
91         cout << "Escribir " << num_escritor << " empieza a escribir." << endl << flush;
92
93         this_thread::sleep_for( duracion_escribir );
94     }
95
96     void funcion_hebra_escritor(int num_escritor, MRef<Lec_Esc> lec_esc){
97         while(true){
98             lec_esc->ini_escritura();
99             escribir(num_escritor);
100             lec_esc->fin_escritura();
101         }
102     }
```

```

106 void funcion_hebra_escritor(int num_escritor, MRef<Lec_Esc> lec_esc){
107     while(true){
108         lec_esc->ini_escritura();
109         escribir(num_escritor);
110         lec_esc->fin_escritura();
111     }
112 }
113
114 void funcion_hebra_lectora(int num_lector, MRef<Lec_Esc> lec_esc){
115     while(true){
116         lec_esc->ini_lectura();
117         lee(num_lector);
118         lec_esc->fin_lectura();
119     }
120 }
121
122 int main(int argc, char * argv[])
123 {
124     MRef<Lec_Esc> lec_esc=Create<Lec_Esc>();
125     thread lectores[num_lectores], escritores[num_escritores];
126
127     for(int i=0; i < num_escritores; i++){
128         escritores[i] = thread(funcion_hebra_escritor, i, lec_esc);
129     }
130
131     for(int i=0; i < num_lectores; i++){
132         lectores[i] = thread(funcion_hebra_lectora, i, lec_esc);
133     }
134
135     for(int i=0; i < num_escritores; i++){
136         escritores[i].join();
137     }
138
139     for(int i=0; i < num_lectores; i++){
140         lectores[i].join();
141     }
142
143     return 0;
144 }

```