SISTEMAS CONCURRENTES Y DISTRIBUIDOS: PRÁCTICA 1

Sincronización de hebras con semáforos

Montserrat Rodríguez Zamorano

22 de noviembre de 2015

1. Objetivos

En esta práctica se realizarán dos implementaciones de dos problemas sencillos de sincronización usando librerías abiertas para programación multihebra y semáforos.

Los objetivos son:

1. Conocer el problema del productor-consumidor y sus aplicaciones.
2. Conocer un problema sencillo de sincronización de hebras (el problema de los fumadores).

2. El problema del productor-consumidor

El problema del productor-consumidor surge cuando se quiere diseñar un programa en el cual un proceso o hebra procuce items de datos en memoria que otro proceso o hebra consume.

2.1. Diseño de la solución

Se considerán dos hebras independientes, la hebra productora y la consumidora. La productora escribirá los números desde el 0 hasta el número de ítems establecido, mientras que la consumidorá los leerá. La implementación deberá tener en cuenta:

1. Se debe escribir y leer cada ítem una sola vez.
2. El productor no podrá escribir mientras no haya espacio en el buffer de escritura.
3. El orden de lectura no tiene por qué coincidir con el de lectura. Si fuese necesario, la única implementación posible sería usando una estructura $FIFO$.

2.1.1. Variables

1. Si se utiliza una pila, sólo será necesaria una variable, $primera\_libre$, que apuntará a la primera posición de la pila libre. Por tanto, a la hora de producir, se escribirá en $buffer[primera\_libre]$, y a la hora de leer, $buffer[primera\_libre - 1]$. Cuando se haya leído un valor se decrementará la variable, pues ya puede escribirse sobre ese valor, y cuando escriba se incrementará.

2. Si se utiliza una cola serán necesarias dos variables, *primera_libre*, que funcionará igual de la misma forma que el apartado anterior para la escritura, y *primera_ocupada*, que apuntará al primer elemento de los "no consumidos"que se produjo. Por tanto, a la hora de producir, se escribirá en $buffer[primera\_libre]$, y a la hora de leer, $buffer[primera\_ocupada]$. Cuando se haya leído un valor se incrementará la variable *primera_ocupada*. En caso de que una de las variables llegue al final del $buffer$, volverá a la posición 0.

3. Independientemente de si la implementación es $LIFO$ o $FIFO$, se declararán las siguientes variables globales: número de elementos a escribir (*num_items*, constante), buffer de escritura (array) y número de elementos de dicho array (*tam_vector*, constante).

2.1.2. Semáforos

Para garantizar la comunicación entre estas dos hebras, será necesario el uso de semáforos que faciliten la sincronización. Si no se usaran, la hebra consumidora podría leer antes de que la hebra productora escribiera, o la productora escribir sobre un dato que todavía no se ha consumido.

En cualquiera de las dos implementaciones se usarán tres semáforos:

1. *puede_leer*: Inicialmente no se puede leer, por lo que se inicializará a 0. Cuando la hebra productora comience a producir datos, se hará *sem_signal* para que la hebra consumidora pueda consumirlos.

```
void * productor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
    int dato = producir_dato() ;
    sem_wait(&puede_escribir) ;
      if(primera_libre < tam_vector) {
      (...)
      }
    sem_post(&puede_leer) //<--------
  }
  return NULL ;
}
```

   Se hará *sem_wait* en el inicio de la función *consumidor*, de modo que no comience a consumir datos mientras no se haya usado *sem_post*.

```
void * consumidor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
    sem_wait(&puede_leer) ; //<--------
    (...)
  }
  return NULL ;
}
```

2. *puede_escribir*: En un inicio la hebra productora tendrá permiso para escribir, por lo que se inicializará a 1. Se utilizará *sem_wait* al comienzo de la función *productor*, ya que si el proceso está bloqueado no puede escribir, y *sem_signal* al final de la función *consumidor*, ya que conforme se guardan y consumen los datos, hay más espacio en el buffer para que la hebra pueda seguir produciendo.

```
void * productor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
    int dato = producir_dato() ;
    sem_wait(&puede_escribir) ; //<--------
```

```
      if(primera_libre < tam_vector) {
      (...)
}
// -----------------------------------------------------------------------

void * consumidor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
    sem_wait(&puede_leer) ;
      int dato ;
      if(primera_ocupada < tam_vector) {
      (...)
      }
    sem_post(&puede_escribir) ; //<-------
    consumir_dato(dato) ;
  }
  return NULL ;
}
```

3. *mutex*: Se utiliza exclusivamente para garantizar la exclusión mutua a la hora de escribir por pantalla, de modo que los mensajes no se solapen y se pueda leer con claridad la sucesión de acciones. Se inicializa a 1.

```
  sem_wait (&mutex) ;
    cout << "dato recibido: " << dato << endl ;
  sem_post (&mutex) ;
```

2.2. Implementación

2.2.1. Código fuente (FIFO)

```
// *****************************************************************
// SCD. Practica 1.
//
// Ejercicio productor-consumidor (LIFO)
// Montserrat Rodriguez Zamorano
// *****************************************************************

#include <iostream>
#include <cassert>
#include <pthread.h>
#include <semaphore.h>

using namespace std ;

// -----------------------------------------------------------------------
// variables compartidas y constantes
const unsigned
  num_items  = 40 ,
  tam_vector = 10 ;

int buffer [tam_vector] ; //buffer de escritura
unsigned primera_libre = 0 ;
unsigned primera_ocupada = 0 ;

sem_t
  mutex ,
  puede_leer ,
  puede_escribir ;
// -----------------------------------------------------------------------

unsigned producir_dato()
{
  static int contador = 0 ;
```

```
  sem_wait (&mutex) ;
  cout << "producido: " << contador << endl << flush ;
  sem_post (&mutex) ;
  return contador++ ;
}
// ---------------------------------------------------------------------

void consumir_dato( int dato )
{
  sem_wait (&mutex) ;
    cout << "dato recibido: " << dato << endl ;
  sem_post (&mutex) ;
}
// ---------------------------------------------------------------------

void * productor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
    int dato = producir_dato() ;
    sem_wait(&puede_escribir) ;
      if(primera_libre < tam_vector) { // si se puede escribir en el buffer, producir
        buffer[primera_libre] = dato ;
        primera_libre++ ;

        if(primera_libre == tam_vector) {
          primera_libre = 0;
        }
      }
    sem_post(&puede_leer) ;
  }
  return NULL ;
}
// ---------------------------------------------------------------------

void * consumidor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
    sem_wait(&puede_leer) ;
      int dato ;
      if(primera_ocupada < tam_vector) {
        dato = buffer[primera_ocupada];
        primera_ocupada++ ;

        if(primera_ocupada == tam_vector){
          primera_ocupada = 0;
        }
      }
    sem_post(&puede_escribir) ;
    consumir_dato(dato) ;
  }
  return NULL ;
}
//----------------------------------------------------------------------

int main()
{
  pthread_t productora, consumidora ;
  // Inicializar los semaforos
  sem_init( &puede_leer , 0 , 0 ); // inicialmente no se puede leer
  sem_init( &puede_escribir , 0 , 1 ); // inicialmente se puede escribir
  sem_init( &mutex , 0, 1);

  // Crear las hebras
  pthread_create( &productora , NULL, productor, NULL) ;
  pthread_create( &consumidora, NULL, consumidor, NULL) ;

  //Esperar a que las hebras terminen
  pthread_join( productora, NULL ) ;
```

```
  pthread_join( consumidora, NULL ) ;

  //Escribir "fin" cuando hayan acabado las dos hebras
  cout << "fin" << endl ;

  //Destruir los semaforos
  sem_destroy( &puede_leer ) ;
  sem_destroy( &puede_escribir ) ;
  sem_destroy( &mutex ) ;


   return 0 ;
}
```

2.2.2. Código fuente (LIFO)

```
// *****************************************************************
// SCD. Practica 1.
//
// Ejercicio productor-consumidor (LIFO)
// Montserrat Rodriguez Zamorano
// *****************************************************************

#include <iostream>
#include <cassert>
#include <pthread.h>
#include <semaphore.h>

using namespace std ;

// ---------------------------------------------------------------------
// variables compartidas y constantes
const unsigned
  num_items  = 40 ,
  tam_vector = 10 ;

int buffer [tam_vector] ; //buffer de escritura
unsigned primera_libre = 0 ;

sem_t
  mutex ,
  puede_leer ,
  puede_escribir ;
// ---------------------------------------------------------------------

unsigned producir_dato()
{
  static int contador = 0 ;
  sem_wait (&mutex) ;
  cout << "producido: " << contador << endl << flush ;
  sem_post (&mutex) ;
  return contador++ ;
}
// ---------------------------------------------------------------------

void consumir_dato( int dato )
{
  sem_wait (&mutex) ;
    cout << "dato recibido: " << dato << endl ;
  sem_post (&mutex) ;
}
// ---------------------------------------------------------------------

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

```
    sem_wait(&puede_escribir) ;
      if(primera_libre < tam_vector) { // si se puede escribir en el buffer, producir
        buffer[primera_libre] = dato ;
        primera_libre++ ;
      }
    sem_post(&puede_leer) ;
  }
  return NULL ;
}
// ----------------------------------------------------------------------

void * consumidor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
    sem_wait(&puede_leer) ;
      int dato ;
      if( primera_libre >= 0) { // si hay algo en el buffer, consumir
        dato = buffer[primera_libre-1];
        primera_libre-- ;
      }
    sem_post(&puede_escribir) ;
    consumir_dato(dato) ;
  }
  return NULL ;
}
//----------------------------------------------------------------------

int main()
{
  pthread_t productora, consumidora ;
  // Inicializar los semaforos
  sem_init( &puede_leer , 0 , 0 ); // inicialmente no se puede leer
  sem_init( &puede_escribir , 0 , 1 ); // inicialmente se puede escribir
  sem_init( &mutex , 0, 1);

  // Crear las hebras
  pthread_create( &productora , NULL, productor, NULL) ;
  pthread_create( &consumidora, NULL, consumidor, NULL) ;

  //Esperar a que las hebras terminen
  pthread_join( productora, NULL ) ;
  pthread_join( consumidora, NULL ) ;

  //Escribir "fin" cuando hayan acabado las dos hebras
  cout << "fin" << endl ;

  //Destruir los semaforos
  sem_destroy( &puede_leer ) ;
  sem_destroy( &puede_escribir ) ;
  sem_destroy( &mutex ) ;


   return 0 ;
}
```

2.2.3. Capturas de pantalla

En las figuras (1) y (2)se puede ver la salida en terminal tras la ejecución del programa $prodcons - FIFO$ y $prodcons - LIFO$, respectivamente. La primera corresponde al principio de la ejecución, mientras que la segunda al final (se escribe la palabra fin cuando finalizan ambas hebras), ya que ambas ejecuciones son iguales en cuanto a la salida por terminal.

```
minim@minim:~/Desktop/practica1$ make pcFIFO
g++ -o prod-consFIFO -g -Wall  prod-consFIFO.cpp -lpthread
./prod-consFIFO
producido: 0
producido: 1
dato recibido: 0
producido: 2
producido: 3
dato recibido: 1
dato recibido: 2
dato recibido: 3
producido: 4
producido: 5
producido: 6
dato recibido: 4
dato recibido: 5
dato recibido: 6
```

Figura 1: Ejemplo de ejecución del programa $prodcons - FIFO$

3. El problema de los fumadores

Consideramos un estanco en el que hay tres fumadores y un estanquero. Antes de fumar, es necesario liar un cigarro. Para ello se necesitarán tres ingredientes: tabaco, papel y cerillas.

Cada uno de los fumadores tendrá dos de los tres ingredientes. El estanquero seleccionará aleatoriamente uno de los ingredientes y lo colocará sobre el mostrador. El fumador que necesita dicho ingrediente lo cogerá y podrá fumar. Hasta que no retire el ingrediente, el estanquero no podrá colocar otro sobre el mostrador.

3.1. Diseño de la solución

Consideraremos cada uno de los fumadores y el estanquero como hebras independienes. En un principio, las hebras correspondientes a los fumadores estarán bloqueadas, ya que no tienen los ingredientes necesarios. La hebra correspondiente al estanquero seleccionará un ingrediente, desbloqueará la hebra correspondiente al fumador que necesita dicho ingrediente, y se bloqueará hasta su retirada.

La hebra desbloqueada tomará el ingrediente, desbloqueará la hebra correspondiente al estanquero (volviendo a repetir el proceso anterior) y realizará la función $fumar$.

Esta actividad de suministrar-fumar se realizará en un bucle infinito, por lo que ambas funciones contienen un $while(true)$ que garantice que este proceso no se detendrá.

3.1.1. Variables y funciones

Se enumeran las variables usadas en el código fuente y su utilidad.

1. Variables constantes y globales: número de fumadores ($num\_fumadores$), array de parejas para los ingredientes (ingrediente-índice del fumador al que le falta), índice del último ingrediente que se colocó sobre el mostrador ($sobre\_el\_mostrador$).
2. Función $fumar$: simula la acción de fumar como un retardo aleatorio de la hebra.
3. Función $suministrar$: simula la acción de colocar sobre el mostrador generando un número aleatorio y devolviendo el índice del ingrediente.

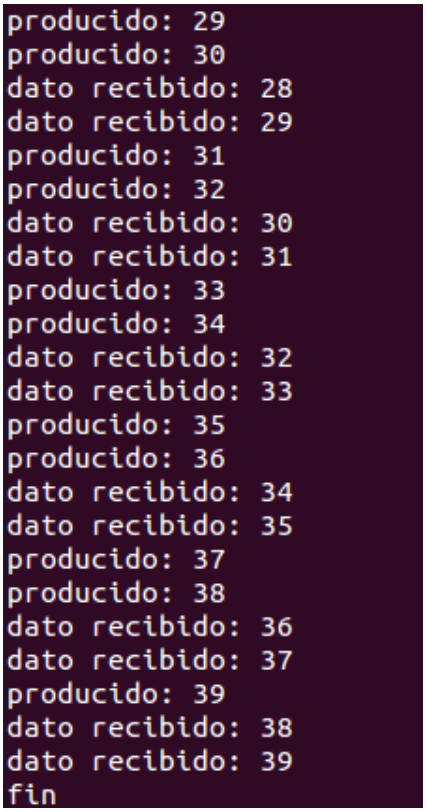

Figura 2: Ejemplo de ejecución del programa $prodcons - LIFO$

4. Función $fumador$: es la función que realiza cada hebra correspondiente a un fumador, llamando a la función $fumar$. Se le pasa como parámetro el índice del fumador (hebra) que ejecutará la función. En el problema del productor-consumidor, se correspondería con la función consumidor.

5. Función *estanquero* : Actualiza el último ingrediente sobre el mostrador haciendo uso de la función *suministrar*. En el problema del productor-consumidor, se correspondería con la función productor.

3.1.2. Semáforos

Para controlar el bloqueo y desbloqueo de las hebras es necesario el uso de semáforos que nos ayudaran en la sincronización para el correcto funcionamiento del programa. Se usarán tres semáforos.

1. $puede\_fumar[i]$: Se usará un array de semáforos, uno por cada uno de los fumadores. Estos fumadores no podrán empezar a fumar hasta que no se coloque algún ingrediente sobre el mostrador, por lo que se inicializará a 0.

   En la función $fumador$, el semáforo hará *sem_wait* sobre las hebras para que no comience la ejecución de la función antes de que se proporcione el ingrediente correspondiente a su índice.

```
void * fumador(void * ih_void) {
  while(true) {
    unsigned long ih = (unsigned long) ih_void ;
    sem_wait ( &puede_fumar[ih]) ; //<--------
    (...)
  }
}
```

En la función *suministrar*, una vez que el estanquero haya generado el índice, guardándola en la variable *sobre_el_mostrador*, desbloqueará la hebra correspondiente con *sem_signal*: $puede\_fumar[sobre\_el\_mostrador]$.

```
void * estanquero(void *) {
  while(true) {
    (...)
    sobre_el_mostrador = suministrar() ;
    sem_post( &puede_fumar[sobre_el_mostrador]) ;//<-------
  }
}
```

2. *puede_suministrar*: Se utiliza para bloquear la hebra correspondiente al estanquero mientras no se haya retirado el ingrediente y para desbloquearla cuando se haya recogido. Por tanto, hará *sem_wait* cuando suministre el ingrediente, en la función *suministrar*, y *sem_signal* cuando la hebra del fumador correspondiente la haya retirado, en la función $fumador$.

```
void * fumador(void * ih_void) {
  while(true) {
    unsigned long ih = (unsigned long) ih_void ;
    sem_wait ( &puede_fumar[ih]) ;
    (...)
    sem_post ( &puede_suministrar) ; //<-------
    fumar() ;
    (...)
  }
}

void * estanquero(void *) {
  while(true) {
    sem_wait( &puede_suministrar) ; //<-------
    sobre_el_mostrador = suministrar() ;
    (...)
  }
}
```

Se inicializa a 1, ya que es esta hebra la que tiene que comenzar generando el ingrediente.

3. *mutex*: Se utiliza exclusivamente para garantizar la exclusión mutua a la hora de escribir por pantalla, de modo que los mensajes no se solapen y se pueda leer con claridad la sucesión de acciones. Se inicializa a 1.

```
  sem_wait( &mutex) ;
    cout << "El estanquero coloca en el mostrador " << ingredientes[ing].second
    << "." << endl ;
  sem_post( &mutex) ;
```

3.2. Implementación

3.2.1. Código fuente

```
// *****************************************************************
// SCD. Practica 1.
//
// Problema de los fumadores.
// Montserrat Rodriguez Zamorano
// *****************************************************************

#include <iostream>
#include <cassert>
#include <pthread.h>
#include <semaphore.h>
```

```
#include <time.h>       // incluye "time(....)"
#include <unistd.h>     // incluye "usleep(...)"
#include <stdlib.h>     // incluye "rand(...)" y "srand"
#include <utility>

using namespace std ;

// ---------------------Variables constantes y globales-------------------------------
const unsigned num_fumadores = 3 ;
unsigned long sobre_el_mostrador = -1; //variable compartida que representa el ingrediente
//ingredientes y fumador al que le falta
pair <int, string> ingredientes[num_fumadores];
// ---------------------------------Semaforos-----------------------------------------
sem_t
  puede_suministrar ,
  puede_fumar[num_fumadores] ,
  mutex ;
// ---------------------------------Funciones-----------------------------------------
// funcion que simula la accion de fumar  como un retardo aleatorio de la hebra

void fumar()
{
   //  inicializa la semilla aleatoria  (solo la primera vez)
   static bool primera_vez = true ;
   if ( primera_vez )
   {   primera_vez = false ;
      srand( time(NULL) );
   }

   // calcular un numero aleatorio de milisegundos (entre 1/10 y 2 segundos)
   const unsigned miliseg = 100U + (rand() %1900U) ;

   // retraso bloqueado durante 'miliseg' milisegundos
   usleep( 1000U*miliseg );
}
// -----------------------------------------------------------------------------------
//funcion que simula la accion de suministrar ingredientes

int suministrar() {
  static bool primera_vez = true ;
  //  inicializa la semilla aleatoria  (solo la primera vez)
  if ( primera_vez )
  {   primera_vez = false ;
     srand( time(NULL) );
  }

  unsigned ing = rand() %3 ;

  sem_wait( &mutex) ;
    cout << "El estanquero coloca en el mostrador " << ingredientes[ing].second
    << "." << endl ;
  sem_post( &mutex) ;
  return ing ; //devuelve el indice del ingrediente correspondiente
}
// -----------------------------------------------------------------------------------
//funcion "consumidor"

void * fumador(void * ih_void) {
  while(true) {
    ////////////recoger/////////
    unsigned long ih = (unsigned long) ih_void ;
    sem_wait ( &puede_fumar[ih]) ;
    sem_wait( &mutex) ;
      cout << "El fumador " << ingredientes[ih].first
      << " recoge " << ingredientes[ih].second << "." << endl ;
    sem_post ( &mutex) ;
    /////////fumar////////////
    sem_wait( &mutex) ;
      cout << "El fumador " << ingredientes[ih].first
      << " empieza a fumar." << endl ;
```

```
    sem_post ( &mutex) ;
    sem_post ( &puede_suministrar) ; //desbloquea al estanquero
    fumar() ;
    sem_wait( &mutex) ;
      cout << "El fumador " << ingredientes[ih].first
      << " ha terminado de fumar." << endl ;
    sem_post ( &mutex) ;
  }
}
// -----------------------------------------------------------------------
//funcion "productor"

void * estanquero(void *) {
  while(true) {
    sem_wait( &puede_suministrar) ;
      sobre_el_mostrador = suministrar() ;
    sem_post( &puede_fumar[sobre_el_mostrador]) ;
  }
}
// -----------------------------------------------------------------------

int main()
{
  pair <int,string> par = make_pair(1,"cerillas");
  ingredientes[0] = par ;
  par = make_pair(2, "tabaco") ;
  ingredientes[1] = par ;
  par = make_pair (3, "papel") ;
  ingredientes[2] = par ;

  sem_init( &puede_suministrar, 0, 1) ;
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    sem_init( &puede_fumar[i], 0, 0) ;
  }
  sem_init( &mutex, 0, 1) ;

  //crear las hebras
  pthread_t fumadores[num_fumadores], estanco ;
  pthread_create( &estanco, NULL, estanquero, NULL) ;
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    void * arg_ptr = (void *) i ; //convertir entero a puntero
    pthread_create( &fumadores[i] , NULL, fumador, arg_ptr ) ;
  }

  //esperar a que las hebras se unan
  pthread_join( estanco, NULL ) ;
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    pthread_join( fumadores[i] , NULL ) ;
  }

  //destruir los semaforos
  sem_destroy( &puede_suministrar) ;
  sem_destroy( &mutex) ;
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    sem_destroy( &puede_fumar[i]) ;
  }

  return 0 ;
}
```

3.2.2. Capturas de pantalla

En la figura (3) se puede ver la salida en terminal tras la ejecución del programa.

Figura 3: Ejemplo de ejecución del programa $fumadores$

4. Ejercicios adicionales

Se presentan a continuación algunas variantes de los ejercicios que se piden en el cuaderno de prácticas.

4.1. El problema del productor - consumidor II : número de productores y consumidores múltiple

Se tratará el problema del productor-consumidor con varias hebras productoras y consumidoras.

4.1.1. Modificaciones en el código

El número de items a producir y posteriormente consumir debe ser múltiplo del número de hebras productoras y consumidoras. Esto es porque la variable contador es compartida entre las hebras productoras, y por tanto para que se produzcan y consuman el número de items deseados tendrá que cumplirse la siguiente igualdad:

$$n * numero\_hebras\_productoras = m * numero\_hebras\_consumidoras = numero\_items$$

donde n es el número de items que escribirá cada hebra productora y m el número de items que leerá cada hebra consumidora. Este ajuste se hará automáticamente mediante la función $ajustar$ a partir de num_items y el número de hebras productoras y consumidoras.

Además habrá que incluir un semáforo adicional en el caso de $LIFO$, que controle el acceso a la variable $primera\_libre$ en exclusión mutua. En el caso de $FIFO$, serán necesarios semáforos adicionales, uno para $primera\_libre$ y otro para $primera\_ocupada$.

4.1.2. Código fuente (LIFO)

```
// *****************************************************************
// SCD. Practica 1.
//
// Ejercicio productor-consumidor (LIFO)
// Montserrat Rodriguez Zamorano
// *****************************************************************

#include <iostream>
#include <cassert>
#include <pthread.h>
#include <semaphore.h>

using namespace std ;

// ---------------------------------------------------------------------
// variables compartidas y constantes
// condicion : num_items_p*num_productores = num_items_c*num_consumidores = num_items
// num_items debe ser multiplo de num_consumidores y num_productores
unsigned
  num_items_p = 0 ,
  num_items_c = 0 ;
const unsigned
  num_items = 30 ,
  num_consumidores = 3 ,
  num_productores = 3 ,
  tam_vector = 10 ;

int buffer [tam_vector] ; //buffer de escritura
unsigned primera_libre = 0 ;

sem_t
  mutex ,
  mutex1 ,
  puede_leer ,
  puede_escribir ;
// ---------------------------------------------------------------------
unsigned ajustar() {
  num_items_p = num_items/num_productores ;
  num_items_c = num_items/num_consumidores ;
}

unsigned producir_dato()
{
  static int contador = 0 ;
  sem_wait (&mutex) ;
  cout << "producido: " << contador << endl << flush ;
  sem_post (&mutex) ;
  return contador++ ;
}
// ---------------------------------------------------------------------

void consumir_dato( int dato )
{
  sem_wait (&mutex) ;
    cout << "dato recibido: " << dato << endl ;
  sem_post (&mutex) ;
}
// ---------------------------------------------------------------------

void * productor( void * )
{
  for( unsigned i = 0 ; i < num_items_p ; i++ )
  {
    int dato = producir_dato() ;
    sem_wait(&puede_escribir) ;
    sem_wait(&mutex1) ;
    buffer[primera_libre] = dato ;
```

```
    primera_libre++ ;
    sem_post(&mutex1) ;
    sem_post(&puede_leer) ;
  }
  return NULL ;
}
// ----------------------------------------------------------------------

void * consumidor( void * )
{
  for( unsigned i = 0 ; i < num_items_c ; i++ )
  {
    sem_wait(&puede_leer) ;
    int dato ;
    sem_wait(&mutex1) ;
    dato = buffer[primera_libre-1];
    primera_libre-- ;
    sem_post(&mutex1) ;
    sem_post(&puede_escribir) ;
    consumir_dato(dato) ;
  }
  return NULL ;
}
//----------------------------------------------------------------------

int main()
{
  ajustar() ;
  pthread_t productora[num_productores],consumidora[num_consumidores] ;
  // Inicializar los semaforos
  sem_init( &puede_leer , 0 , 0 ); // inicialmente no se puede leer
  sem_init( &puede_escribir , 0 , tam_vector );
  sem_init( &mutex , 0, 1);
  sem_init( &mutex1 , 0, 1);

  // Crear las hebras
  for ( unsigned i = 0 ; i < num_productores ; i++) {
    pthread_create( &productora[i] , NULL, productor, NULL) ;
  }

  for ( unsigned i = 0 ; i < num_consumidores ; i++) {
    pthread_create( &consumidora[i], NULL, consumidor, NULL) ;
  }

  //Esperar a que las hebras terminen
  for ( unsigned i = 0 ; i < num_productores ; i++) {
    pthread_join( productora[i], NULL ) ;
  }

  for ( unsigned i = 0 ; i < num_consumidores ; i++) {
    pthread_join( consumidora[i], NULL ) ;
  }

  //Escribir "fin" cuando hayan acabado las dos hebras
  cout << "fin" << endl ;

  //Destruir los semaforos
  sem_destroy( &puede_leer ) ;
  sem_destroy( &puede_escribir ) ;
  sem_destroy( &mutex ) ;
  sem_destroy( &mutex1) ;


   return 0 ;
}
```

4.1.3. Código fuente (FIFO)

```
// *****************************************************************
// SCD. Practica 1.
//
// Ejercicio productor-consumidor (LIFO)
// Montserrat Rodriguez Zamorano
// *****************************************************************

#include <iostream>
#include <cassert>
#include <pthread.h>
#include <semaphore.h>

using namespace std ;

// ---------------------------------------------------------------------
// variables compartidas y constantes
// condicion : num_items_p*num_productores = num_items_c*num_consumidores = num_items
// num_items debe ser multiplo de num_consumidores y num_productores
unsigned
  num_items_p = 0 ,
  num_items_c = 0 ;
const unsigned
  num_items = 30 ,
  num_consumidores = 3 ,
  num_productores = 3 ,
  tam_vector = 10 ;

int buffer [tam_vector] ; //buffer de escritura
unsigned primera_libre = 0 ;
unsigned primera_ocupada = 0 ;

sem_t
  mutex ,
  mutex_productores,
  mutex_consumidores,
  puede_leer ,
  puede_escribir ;
// ---------------------------------------------------------------------

unsigned ajustar() {
  num_items_p = num_items/num_productores ;
  num_items_c = num_items/num_consumidores ;
}

unsigned producir_dato()
{
  static int contador = 0 ;
  sem_wait (&mutex) ;
  cout << "producido: " << contador << endl << flush ;
  sem_post (&mutex) ;
  return contador++ ;
}
// ---------------------------------------------------------------------

void consumir_dato( int dato )
{
  sem_wait (&mutex) ;
    cout << "dato recibido: " << dato << endl ;
  sem_post (&mutex) ;
}
// ---------------------------------------------------------------------

void * productor( void * )
{
  for( unsigned i = 0 ; i < num_items_p ; i++ )
  {
    int dato = producir_dato() ;
    sem_wait(&mutex_productores);
    sem_wait(&puede_escribir) ;
```

```
    buffer[primera_libre] = dato ;
    primera_libre++ ;

    if(primera_libre == tam_vector) {
      primera_libre = 0;
    }
    sem_post(&puede_leer) ;
    sem_post(&mutex_productores) ;
  }
  return NULL ;
}
// -----------------------------------------------------------------------

void * consumidor( void * )
{
  for( unsigned i = 0 ; i < num_items_c ; i++ )
  {
    sem_wait(&puede_leer) ;
    int dato ;
    sem_wait(&mutex_consumidores);
    dato = buffer[primera_ocupada];
    primera_ocupada++ ;

    if(primera_ocupada == tam_vector){
     primera_ocupada = 0;
    }

    sem_post(&puede_escribir) ;
    sem_post(&mutex_consumidores) ;
    consumir_dato(dato) ;
  }
  return NULL ;
}
//-----------------------------------------------------------------------

int main()
{
  ajustar() ;
  pthread_t productora[num_productores],consumidora[num_consumidores] ;
  // Inicializar los semaforos
  sem_init( &puede_leer , 0 , 0 ); // inicialmente no se puede leer
  sem_init( &puede_escribir , 0 , tam_vector ); // inicialmente se puede //escribir tantas
      veces como tamanio tenga el vector
  sem_init( &mutex , 0, 1);
  sem_init( &mutex_productores , 0, 1);
  sem_init( &mutex_consumidores , 0, 1);

  // Crear las hebras
  for ( unsigned i = 0 ; i < num_productores ; i++) {
    pthread_create( &productora[i] , NULL, productor, NULL) ;
  }

  for ( unsigned i = 0 ; i < num_consumidores ; i++) {
    pthread_create( &consumidora[i], NULL, consumidor, NULL) ;
  }

  //Esperar a que las hebras terminen
  for ( unsigned i = 0 ; i < num_productores ; i++) {
    pthread_join( productora[i], NULL ) ;
  }

  for ( unsigned i = 0 ; i < num_consumidores ; i++) {
    pthread_join( consumidora[i], NULL ) ;
  }

  //Escribir "fin" cuando hayan acabado las dos hebras
  cout << "fin" << endl ;

  //Destruir los semaforos
  sem_destroy( &puede_leer ) ;
```

```
  sem_destroy( &puede_escribir ) ;
  sem_destroy( &mutex ) ;
  sem_destroy( &mutex_productores ) ;
  sem_destroy( &mutex_consumidores ) ;


   return 0 ;
}
```

4.2. El problema de los fumadores II: número de cigarrillos limitado.

Este problema tendrá las mismas características que el anterior, con la excepción de que el bucle estanquero-fumador no será infinito, y habrá que modificar el código para que el programa acabe de manera elegante.

4.2.1. Modificaciones en el código

La única modificación consiste en incorporar una variable constante *limite_estanco* que será el número máximo de ingredientes que venderá el estanco en un día antes de cerrar. El número de ventas se controlará con un contador, y se cambiará la condición *while*(*true*) por *while*(*limite_estanco* < *contador_estanco*). No será necesario añadir más hebras ni semáforos.

4.2.2. Código fuente

```
// *****************************************************************
// SCD. Practica 1.
//
// Problema de los fumadores finito.
// Montserrat Rodriguez Zamorano
// *****************************************************************

#include <iostream>
#include <cassert>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>       // incluye "time(....)"
#include <unistd.h>     // incluye "usleep(...)"
#include <stdlib.h>     // incluye "rand(...)" y "srand"
#include <utility>

using namespace std ;

// ---------------------Variables constantes y globales-------------------------------
const unsigned num_fumadores = 3 ;
unsigned long sobre_el_mostrador = -1; //variable compartida que representa el ingrediente
//ingredientes y fumador al que le falta
pair <int, string> ingredientes[num_fumadores];
const unsigned limite_estanco = 10 ;
static unsigned contador_estanco = 0 ;
pthread_t fumadores[num_fumadores], estanco ; //se declaran aqui para poder hacer
    pthread_exit
// -----------------------------------Semaforos---------------------------------------
sem_t
  puede_suministrar ,
  puede_fumar[num_fumadores] ,
  mutex ;
// -----------------------------------Funciones---------------------------------------
// funcion que simula la accion de fumar  como un retardo aleatorio de la hebra

void fumar()
{
   // inicializa la semilla aleatoria  (solo la primera vez)
   static bool primera_vez = true ;
   if ( primera_vez )
   {   primera_vez = false ;
```

```
      srand( time(NULL) );
   }

   // calcular un numero aleatorio de milisegundos (entre 1/10 y 2 segundos)
   const unsigned miliseg = 100U + (rand() %1900U) ;

   // retraso bloqueado durante 'miliseg' milisegundos
   usleep( 1000U*miliseg );
}
// ---------------------------------------------------------------------
//funcion que simula la accion de suministrar ingredientes

int suministrar() {
  static bool primera_vez = true ;
  //  inicializa la semilla aleatoria  (solo la primera vez)
  if ( primera_vez )
  {   primera_vez = false ;
     srand( time(NULL) );
  }

  unsigned ing = rand() %3 ;

  sem_wait( &mutex) ;
    cout << "El estanquero coloca en el mostrador " << ingredientes[ing].second
    << "." << endl ;
  sem_post( &mutex) ;
  return ing ; //devuelve el indice del ingrediente correspondiente
}
// ---------------------------------------------------------------------
//funcion "consumidor"

void * fumador(void * ih_void) {
  unsigned long ih = (unsigned long) ih_void ;
  while(contador_estanco < limite_estanco) {
    ///////////recoger////////
    sem_wait ( &puede_fumar[ih]) ;
    sem_wait( &mutex) ;
      cout << "El fumador " << ingredientes[ih].first
      << " recoge " << ingredientes[ih].second << "." << endl ;
    sem_post ( &mutex) ;
    ////////fumar///////////
    sem_wait( &mutex) ;
      cout << "El fumador " << ingredientes[ih].first
      << " empieza a fumar." << endl ;
    sem_post ( &mutex) ;
    sem_post ( &puede_suministrar) ; //desbloquea al estanquero
    fumar() ;
    sem_wait( &mutex) ;
      cout << "El fumador " << ingredientes[ih].first
      << " ha terminado de fumar." << endl ;
    sem_post ( &mutex) ;
  }
  sem_wait( &mutex) ;
    cout << "El fumador " << ingredientes[ih].first
    << " recoge el ultimo ingrediente y se va a su casa." << endl ;
  sem_post ( &mutex) ;
  pthread_exit( &estanco) ;
        return NULL;
}
// ---------------------------------------------------------------------
//funcion "productor"

void * estanquero(void *) {
  while(contador_estanco < limite_estanco) {
    sem_wait( &puede_suministrar) ;
      sobre_el_mostrador = suministrar() ;
      contador_estanco++;
    sem_post( &puede_fumar[sobre_el_mostrador]) ;
  }
  sem_wait( &mutex) ;
```

```
    cout << "El estanquero coloca sobre el mostrador todos los ingredientes." << endl ;
  sem_post ( &mutex) ;
  for(unsigned i = 0; i < num_fumadores; i++) {
    sem_post( &puede_fumar[i]) ;
  }
  sem_wait( &mutex) ;
    cout << "El estanquero cierra el estanco." << endl ;
  sem_post ( &mutex) ;
  return NULL ;
}
// -----------------------------------------------------------------------------

int main()
{
  pair <int,string> par = make_pair(1,"cerillas");
  ingredientes[0] = par ;
  par = make_pair(2, "tabaco") ;
  ingredientes[1] = par ;
  par = make_pair (3, "papel") ;
  ingredientes[2] = par ;

  sem_init( &puede_suministrar, 0, 1) ;
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    sem_init( &puede_fumar[i], 0, 0) ;
  }
  sem_init( &mutex, 0, 1) ;

  //crear las hebras
  pthread_create( &estanco, NULL, estanquero, NULL) ;
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    void * arg_ptr = (void *) i ; //convertir entero a puntero
    pthread_create( &fumadores[i] , NULL, fumador, arg_ptr ) ;
  }

  //esperar a que las hebras se unan
  pthread_join( estanco, NULL ) ;
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    pthread_join( fumadores[i] , NULL ) ;
  }

  sem_wait( &mutex) ;
    cout << "Hasta manana." << endl ;
  sem_post ( &mutex) ;

  //destruir los semaforos
  sem_destroy( &puede_suministrar) ;
  sem_destroy( &mutex) ;
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    sem_destroy( &puede_fumar[i]) ;
  }

  return 0 ;
}
```

4.2.3. Capturas de pantalla

En la figura (4) se puede ver la salida en terminal en el final de la ejecución del programa. El resto del programa tiene la misma salida en terminal que en la versión anterior.

4.3. El problema de los fumadores III

En esta modificación, habrá varios estancos y los fumadores podrán ir a cualquiera de los dos que suministre el ingrediente necesario.

```
El estanquero coloca en el mostrador papel.
El estanquero coloca sobre el mostrador todos los ingredientes.
El estanquero cierra el estanco.
El fumador 1 recoge cerillas.
El fumador 1 empieza a fumar.
El fumador 3 recoge papel.
El fumador 3 empieza a fumar.
El fumador 1 ha terminado de fumar.
El fumador 1 recoge el último ingrediente y se va a su casa.
El fumador 2 ha terminado de fumar.
El fumador 2 recoge el último ingrediente y se va a su casa.
El fumador 3 ha terminado de fumar.
El fumador 3 recoge el último ingrediente y se va a su casa.
Hasta mañana.
minim@minim:~/Desktop/portafolios_scd/practica1/codigo_adicional$
```

Figura 4: Ejemplo de ejecución del programa *fumadores_finito*

4.3.1. Modificaciones en el código

Para evitar que algunos fumadores se queden sin fumar porque hay otros fumadores que necesitan el mismo ingrediente que ellos, se necesitará implementar un sistema de asignación de prioridades.

Como variables globales tenemos las mismas que en la versión anterior y además un vector de prioridades, un vector de string con los nombres de los ingredientes, ampliar la variable *sobre_el_mostrador* a un vector (cada estanco tiene su mostrador) y el array *desbloquea* que nos servirá para saber qué estanco ha desbloqueado a qué fumador.

El estanquero colocará un ingrediente sobre el mostrador y se lo ofrecerá a aquel fumador con máxima prioridad (función *buscarPrioridad*). La prioridad de los fumadores que necesitan el mismo ingrediente se aumentará en 1 y la del fumador escogido será 0, para que aquellos fumadores que lleven más tiempo sin fumar tengan prioridad sobre el resto.

El acceso al vector de prioridades tiene que hacerse en exclusión mutua, si no, los dos estanqueros podrían escoger al mismo fumador si venden el mismo ingrendiente.

4.3.2. Código fuente

```
// *****************************************************************
// SCD. Practica 1.
//
// Problema de los fumadores. Los fumadores pueden ir a varios estancos
// distintos
//
// Montserrat Rodriguez Zamorano
// *****************************************************************

#include <iostream>
#include <cassert>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>       // incluye "time(....)"
#include <unistd.h>     // incluye "usleep(...)"
#include <stdlib.h>     // incluye "rand(...)" y "srand"
#include <utility>
#include <vector>

using namespace std ;
```

```
// ---------------------Variables constantes y globales-------------------------------
const unsigned num_fumadores = 6 ;
const unsigned num_estanqueros = 3 ;
unsigned long sobre_el_mostrador[num_estanqueros] ; //variable compartida que representa el
    ingrediente
static bool primero = true ;
vector < pair <int,int> > prioridad_fumadores ; //vector para asignar prioridades
vector <string> nombres;
//ingredientes y fumador al que le falta
pair <int, string> ingredientes[num_fumadores];
//array numero fumadores y estanco que le desbloquea
int desbloquea[num_fumadores] ;
// ----------------------------------Semaforos---------------------------------------
sem_t
  puede_suministrar[num_estanqueros] ,
  puede_fumar[num_fumadores] ,
  mutex,
  mutex2 ;
// ----------------------------------Funciones---------------------------------------
void inicializarColaPrioridad(){
  pair < int, int > par ;
  for(int i = 0 ; i < num_fumadores ; i++) {
    par.first = i ;
    par.second = i ;
    prioridad_fumadores.push_back(par) ;
  }
}
//Devuelve el indice del fumador que tiene maxima prioridad para
//recibir el ingrediente
int buscarPrioridad(int i){
  int maximo = -1; //posicion de maxima prioridad
  int prioridad_actual ;
  for(int j = 0 ; j < prioridad_fumadores.size() ; j++){
    if(ingredientes[prioridad_fumadores.at(j).first].second == nombres.at(i))
    {
      prioridad_actual = prioridad_fumadores.at(j).second ;
      if(maximo == -1){ //solo la primera vez
        maximo = j;
      }
      if( prioridad_actual > prioridad_fumadores.at(maximo).second )
        maximo = j ;
    }
  }
  prioridad_fumadores.at(maximo).second = 0; //quitamos la prioridad
  for(int j = 0; j < prioridad_fumadores.size() ; j++) {
    if(ingredientes[prioridad_fumadores.at(j).first].second == nombres.at(i)
      && j != maximo)
    {
      prioridad_fumadores.at(j).second++ ; //incrementamos en uno la prioridad
    }
  }
  return maximo ;
}
// funcion que simula la accion de fumar  como un retardo aleatorio de la hebra

void fumar()
{
   //  inicializa la semilla aleatoria  (solo la primera vez)
   static bool primera_vez = true ;
   if ( primera_vez )
   {   primera_vez = false ;
      srand( time(NULL) );
   }

   // calcular un numero aleatorio de milisegundos (entre 1/10 y 2 segundos)
   const unsigned miliseg = 100U + (rand() %1900U) ;

   // retraso bloqueado durante 'miliseg' milisegundos
   usleep( 1000U*miliseg );
}
```

```
// -----------------------------------------------------------------------------
//funcion que simula la accion de suministrar ingredientes

int suministrar(int indice) {
  static bool primera_vez = true ;
  //  inicializa la semilla aleatoria  (solo la primera vez)
  if ( primera_vez )
  {   primera_vez = false ;
     srand( time(NULL) );
  }

  unsigned ing = rand() %3 ;

  sem_wait( &mutex) ;
    cout << "El estanquero " << indice
    << " coloca en el mostrador " << nombres[ing] << "." << endl ;
  sem_post( &mutex) ;
  return ing ; //devuelve el indice del ingrediente correspondiente
}
// -----------------------------------------------------------------------------
//funcion "consumidor"

void * fumador(void * ih_void) {
  unsigned long ih = (unsigned long) ih_void ;
  while(true) {
    ///////////recoger////////
    sem_wait ( &puede_fumar[ih]) ;
    sem_wait( &mutex) ;
      cout << "El fumador " << ingredientes[ih].first
      << " recoge " << ingredientes[ih].second << " del estanco "
      << desbloquea[ih] << "." << endl ;
    sem_post ( &mutex) ;
    /////////fumar///////////
    sem_wait( &mutex) ;
      cout << "El fumador " << ingredientes[ih].first
      << " empieza a fumar." << endl ;
    sem_post ( &mutex) ;
    sem_post ( &puede_suministrar[desbloquea[ih]]) ; //desbloquea al estanquero
    fumar() ;
    sem_wait( &mutex) ;
      cout << "El fumador " << ingredientes[ih].first
      << " ha terminado de fumar." << endl ;
    sem_post ( &mutex) ;
  }
        return NULL;
}
// -----------------------------------------------------------------------------
//funcion "productor"

void * estanquero(void * ih_void) {
  unsigned long ih = (unsigned long) ih_void ;
  int maxima_prioridad ;
  while(true) {
    sem_wait( &puede_suministrar[ih]) ;
      sobre_el_mostrador[ih] = suministrar(ih) ;
      sem_wait(&mutex2); //evitar que modifiquen al mismo tiempo el vector
      maxima_prioridad = buscarPrioridad(sobre_el_mostrador[ih]) ;
      desbloquea[maxima_prioridad] = ih ;
      sem_post(&mutex2);
    sem_post( &puede_fumar[maxima_prioridad]) ;
  }

  return NULL ;
}
// -----------------------------------------------------------------------------

int main()
{
  ////// fumadores y ingrediente que les falta ////////////////////////
  pair <int,string> par = make_pair(1,"cerillas");
```

```
  ingredientes[0] = par ;
  par = make_pair(4, "cerillas") ;
  ingredientes[3] = par ;
  nombres.push_back("cerillas") ;
  par = make_pair(2, "tabaco") ;
  ingredientes[1] = par ;
  par = make_pair(5, "tabaco") ;
  ingredientes[4] = par ;
  nombres.push_back("tabaco") ;
  par = make_pair (3, "papel") ;
  ingredientes[2] = par ;
  par = make_pair (6, "papel") ;
  ingredientes[5] = par ;
  nombres.push_back("papel") ;
  ////////////////////////////////////////////////////////////////////
  inicializarColaPrioridad();
  for(int i = 0; i < num_estanqueros; i++) {
    sobre_el_mostrador[i] = -1 ;
  } //inicializar el array
  for( unsigned i = 0 ; i < num_estanqueros ; i++ ) {
    sem_init( &puede_suministrar[i], 0, 1) ;
  }
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    sem_init( &puede_fumar[i], 0, 0) ;
  }
  sem_init( &mutex, 0, 1) ;
  sem_init( &mutex2, 0, 1) ;

  //crear las hebras
  pthread_t estanco[num_estanqueros], fumadores[num_fumadores] ;
  for( unsigned i = 0 ; i < num_estanqueros ; i++ ) {
    void * arg_ptr = (void *) i ; //convertir entero a puntero
    pthread_create( &estanco[i] , NULL, estanquero, arg_ptr ) ;
  }
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    void * arg_ptr = (void *) i ; //convertir entero a puntero
    pthread_create( &fumadores[i] , NULL, fumador, arg_ptr ) ;
  }

  //esperar a que las hebras se unan
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    pthread_join( fumadores[i] , NULL ) ;
  }
  for( unsigned i = 0 ; i < num_estanqueros ; i++ ) {
    pthread_join( estanco[i] , NULL ) ;
  }


  sem_wait( &mutex) ;
  sem_wait( &mutex2) ;
    cout << "Hasta maniana." << endl ;
  sem_post ( &mutex) ;

  //destruir los semaforos
  sem_destroy( &mutex) ;
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    sem_destroy( &puede_fumar[i]) ;
  }
  for( unsigned i = 0 ; i < num_estanqueros ; i++ ) {
    sem_destroy( &puede_suministrar[i]) ;
  }

  return 0 ;
}
```

4.4. El problema de los fumadores IV

En esta modificación, habrá varios estancos y los fumadores podrán ir sólo a uno de los dos estancos durante la ejecución del programa.

4.4.1. Modificaciones en el código

Se utiliza como base el mismo código que en el problema *III*, añadiendo una función *inicializaEstanco*, en la que determinaremos qué fumadores pueden ir a qué estanco, haciendo uso del array *desbloquea*. De esta forma, el array no servirá para ver qué estanco ha desbloqueado al fumador, sino para ver cual puede hacerlo. Esto se controlará en la función *buscarPrioridad*, en la que se añade la condición de que devuelve el fumador con máxima prioridad sólo si puede ir al estanco.

El resto del código no se altera, salvo que puede eliminarse el acceso en exclusión mutua, ya que esta vez cada estanco sólo accede a los fumadores que le corresponden.

4.4.2. Código fuente

```
// ---------------------Variables constantes y globales---------------------------
const unsigned num_fumadores = 6 ;
const unsigned num_estanqueros = 2 ;
unsigned long sobre_el_mostrador[num_estanqueros] ; //variable compartida que representa el
    ingrediente
static bool primero = true ;
vector < pair <int,int> > prioridad_fumadores ; //vector para asignar prioridades
vector <string> nombres;
//ingredientes y fumador al que le falta
pair <int, string> ingredientes[num_fumadores];
//array numero fumadores y estanco que le desbloquea
int desbloquea[num_fumadores] ;
// ------------------------------------Semaforos-----------------------------------
sem_t
  puede_suministrar[num_estanqueros] ,
  puede_fumar[num_fumadores] ,
  mutex ;
// ------------------------------------Funciones-----------------------------------
//en esta funcion tenemos que poner a que estanco puede ir cada fumador
void inicializarEstancos(){
  desbloquea[0] = 1 ;
  desbloquea[1] = 1 ;
  desbloquea[2] = 1 ;
  desbloquea[3] = 0 ;
  desbloquea[4] = 0 ;
  desbloquea[5] = 0 ;
}
void inicializarColaPrioridad(){
  pair < int, int > par ;
  for(int i = 0 ; i < num_fumadores ; i++) {
    par.first = i ;
    par.second = i ;
    prioridad_fumadores.push_back(par) ;
  }
}
//Devuelve el indice del fumador que tiene maxima prioridad para
//recibir el ingrediente
int buscarPrioridad(int i, int e){
  int maximo = -1; //posicion de maxima prioridad
  int prioridad_actual ;
  for(int j = 0 ; j < prioridad_fumadores.size() ; j++){
    if(desbloquea[j] == e
       && ingredientes[prioridad_fumadores.at(j).first].second == nombres.at(i))
    {
      prioridad_actual = prioridad_fumadores.at(j).second ;
      if(maximo == -1){ //solo la primera vez
        maximo = j;
      }
```

```
      if( prioridad_actual > prioridad_fumadores.at(maximo).second )
        maximo = j ;
    }
  }
  prioridad_fumadores.at(maximo).second = 0; //quitamos la prioridad
  for(int j = 0; j < prioridad_fumadores.size() ; j++) {
    if(ingredientes[prioridad_fumadores.at(j).first].second == nombres.at(i)
    && j != maximo)
    {
      prioridad_fumadores.at(j).second++ ; //incrementamos en uno la prioridad
    }
  }
  return maximo ;
}
// funcion que simula la accion de fumar  como un retardo aleatorio de la hebra

void fumar()
{
   //  inicializa la semilla aleatoria  (solo la primera vez)
   static bool primera_vez = true ;
   if ( primera_vez )
   {   primera_vez = false ;
      srand( time(NULL) );
   }

   // calcular un numero aleatorio de milisegundos (entre 1/10 y 2 segundos)
   const unsigned miliseg = 100U + (rand() %1900U) ;

   // retraso bloqueado durante 'miliseg' milisegundos
   usleep( 1000U*miliseg );
}
// -----------------------------------------------------------------------------
//funcion que simula la accion de suministrar ingredientes

int suministrar(int indice) {
  static bool primera_vez = true ;
  //  inicializa la semilla aleatoria  (solo la primera vez)
  if ( primera_vez )
  {   primera_vez = false ;
     srand( time(NULL) );
  }

  unsigned ing = rand() %3 ;

  sem_wait( &mutex) ;
    cout << "El estanquero " << indice
    << " coloca en el mostrador " << nombres[ing] << "." << endl ;
  sem_post( &mutex) ;
  return ing ; //devuelve el indice del ingrediente correspondiente
}
// -----------------------------------------------------------------------------
//funcion "consumidor"

void * fumador(void * ih_void) {
  unsigned long ih = (unsigned long) ih_void ;
  while(true) {
    ///////////recoger/////////
    sem_wait ( &puede_fumar[ih]) ;
    sem_wait( &mutex) ;
      cout << "El fumador " << ingredientes[ih].first
      << " recoge " << ingredientes[ih].second << " del estanco "
      << desbloquea[ih] << "." << endl ;
    sem_post ( &mutex) ;
    /////////fumar////////////
    sem_wait( &mutex) ;
      cout << "El fumador " << ingredientes[ih].first
      << " empieza a fumar." << endl ;
    sem_post ( &mutex) ;
    sem_post ( &puede_suministrar[desbloquea[ih]]) ; //desbloquea al estanquero
    fumar() ;
```

```
    sem_wait( &mutex) ;
      cout << "El fumador " << ingredientes[ih].first
      << " ha terminado de fumar." << endl ;
    sem_post ( &mutex) ;
  }
        return NULL;
}
// ---------------------------------------------------------------------
//funcion "productor"

void * estanquero(void * ih_void) {
  unsigned long ih = (unsigned long) ih_void ;
  int maxima_prioridad ;
  while(true) {
    sem_wait( &puede_suministrar[ih]) ;
      sobre_el_mostrador[ih] = suministrar(ih) ;
      maxima_prioridad = buscarPrioridad(sobre_el_mostrador[ih], ih) ;
    sem_post( &puede_fumar[maxima_prioridad]) ;
  }

  return NULL ;
}
// ---------------------------------------------------------------------

int main()
{
  ////// fumadores y ingrediente que les falta ////////////////////////
  pair <int,string> par = make_pair(1,"cerillas");
  ingredientes[0] = par ;
  par = make_pair(4, "cerillas") ;
  ingredientes[3] = par ;
  nombres.push_back("cerillas") ;
  par = make_pair(2, "tabaco") ;
  ingredientes[1] = par ;
  par = make_pair(5, "tabaco") ;
  ingredientes[4] = par ;
  nombres.push_back("tabaco") ;
  par = make_pair (3, "papel") ;
  ingredientes[2] = par ;
  par = make_pair (6, "papel") ;
  ingredientes[5] = par ;
  nombres.push_back("papel") ;
  ////////////////////////////////////////////////////////////////////
  inicializarEstancos();
  inicializarColaPrioridad();
  for(int i = 0; i < num_estanqueros; i++) {
    sobre_el_mostrador[i] = -1 ;
  } //inicializar el array
  for( unsigned i = 0 ; i < num_estanqueros ; i++ ) {
    sem_init( &puede_suministrar[i], 0, 1) ;
  }
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    sem_init( &puede_fumar[i], 0, 0) ;
  }
  sem_init( &mutex, 0, 1) ;

  //crear las hebras
  pthread_t estanco[num_estanqueros], fumadores[num_fumadores] ;
  for( unsigned i = 0 ; i < num_estanqueros ; i++ ) {
    void * arg_ptr = (void *) i ; //convertir entero a puntero
    pthread_create( &estanco[i] , NULL, estanquero, arg_ptr ) ;
  }
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    void * arg_ptr = (void *) i ; //convertir entero a puntero
    pthread_create( &fumadores[i] , NULL, fumador, arg_ptr ) ;
  }

  //esperar a que las hebras se unan
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    pthread_join( fumadores[i] , NULL ) ;
```

```
  }
  for( unsigned i = 0 ; i < num_estanqueros ; i++ ) {
    pthread_join( estanco[i] , NULL ) ;
  }


  sem_wait( &mutex) ;
    cout << "Hasta maniana." << endl ;
  sem_post ( &mutex) ;

  //destruir los semaforos
  sem_destroy( &mutex) ;
  for( unsigned i = 0 ; i < num_fumadores ; i++ ) {
    sem_destroy( &puede_fumar[i]) ;
  }
  for( unsigned i = 0 ; i < num_estanqueros ; i++ ) {
    sem_destroy( &puede_suministrar[i]) ;
  }

  return 0 ;
}
```