Portafolio de seminarios y prácticas.

Índice.

1. Seminario 1: Calculo del número pi.
2. Práctica 1: Productor-consumidor (LIFO).
3. Práctica 1: Productor-consumidor (FIFO).
4. Práctica 1: Fumadores.
5. Práctica 2: Productor-consumidor con buffer limitado.
6. Práctica 2: El problema de los fumadores.
7. Práctica 2: El problema del barbero durmiente.
8. Práctica 3: Productor-consumidor.
9. Práctica 3: Cena de los filósofos.
10. Práctica 3: Cena de los filósofos con camarero.

1. Seminario 1: Cálculo del número pi.

Para implementar el cálculo del número pi mediante un programa concurrente usando hebras he optado por la implementación que permite que cada hebra calcule una franja determinada y continuada.
Cada hebra sabe cuales son sus entradas en función del identificador que tiene asignado. Para calcular las sumas parciales el bucle comienza en el valor de la primera hebra que es usada hasta el número de muestras indicado al ejecutar nuestro programa (la variable m en nuestro código).
Al ser un programa concurrente, a mayor número de procesadores el tiempo del cálculo del número pi es menor, ya que se pueden ejecutar un mayor número de hebras de forma concurrente.

El código fuente de dicho programa concurrente es el que sigue:

```
#include <iostream>
#include <pthread.h>
#include <stdlib.h>

using namespace std ;

// constante y variables globales (compartidas entre hebras)
unsigned long m; // número de muestras (un millón)
unsigned n;          // número de hebras
double resultado_parcial[100000] ; // tabla de sumas parciales (una por hebra)

double f( double x )
{
   return 4.0/(1+x*x) ;      // $~~~~f(x)\,=\,4/(1+x^2)$
}

// cálculo secuencial

double calcular_integral_secuencial( )
{
   double suma = 0.0 ;                       // inicializar suma
   for( unsigned long i = 0 ; i < m ; i++ ) // para cada $i$ entre $0$ y $m-1$
      suma += f( (i+0.5)/m );                // $~~~~~$ añadir $f(x_i)$ a la suma actual
   return suma/m ;                           // devolver valor promedio de $f$
}

// función que ejecuta cada hebra
void * funcion_hebra( void * ih_void )
{
   unsigned long ih = (unsigned long) ih_void ; // número o índice de esta hebra
   double sumap = 0.0 ;
   for(unsigned long i=ih; i<m; i=i+n){
      sumap=sumap+f((i+0.5)/m);
   }
   resultado_parcial[ih] = sumap/m ; // guardar suma parcial en vector.
   return NULL ;
}

// cálculo concurrente

double calcular_integral_concurrente()
{
   double resultado=0;
   // crear y lanzar n hebras, cada una ejecuta "funcion_hebra"
```

```
   pthread_t id_hebra[n];
       for(unsigned long i=0; i<n; i++){
              void * arg_ptr=(void *) i;
              pthread_create(&(id_hebra[i]), NULL, funcion_hebra, arg_ptr);
   }

   // esperar (join) a que termine cada hebra, sumar su resultado
   for(unsigned i=0; i<n; i++){
      pthread_join (id_hebra[i], NULL);
      //cout << "func(" << i << ") == " << resultado << endl;
      resultado+=resultado_parcial[i];
   }
   // devolver resultado completo
   return resultado ;
}

int main(int argc, char *argv[])
{
   m = atoi(argv[1]) ;
   n = atoi(argv[2]);
   cout << "Ejemplo 4 (cálculo de PI)" << endl ;
   double pi_sec = 0.0, pi_conc = 0.0 ;

   pi_sec  = calcular_integral_secuencial() ;
   pi_conc = calcular_integral_concurrente() ;

   cout << "valor de PI (calculado secuencialmente)  == " << pi_sec  << endl
        << "valor de PI (calculado concurrentemente) == " << pi_conc << endl ;

   return 0 ;
}
```

El resultado de dicho programa depende del número de hebras (variable n en el código fuente) y del número de muestras (variable m en el código fuente). Un ejemplo de la ejecución del programa con 100 hebras (n=100) y 100000 muestras (m=100000) sería:

```
Ejemplo 4 (cálculo de PI)
valor de PI (calculado secuencialmente)  == 3.14159
valor de PI (calculado concurrentemente) == 3.14159
```

2. Práctica 1: Productor-Consumidor (LIFO).

Para realizar la practica usando LIFO (pila acotada) para gestionar la ocupación necesitamos una única variable, primera_libre, la cual se incrementa al escribir y se decrementa al leer. De esta manera, siempre se va escribiendo hacia la derecha en el vector donde se almacenan los valores producidos y siempre se va leyendo de dicho vector hacia la izquierda.

Los semáforos usados son puede_leer y puede_escribir. El semáforo puede_leer determina cuando el proceso consumidor puede leer un dato nuevo del vector y el semáforo puede_escribir determina cuando el proceso productor puede escribir un dato nuevo en el vector. Al inicio de nuestro código, el vector está vacío, por lo tanto el número de veces que se puede escribir es el tamaño del vector y el número de veces que se puede leer es 0, ya que no hay ningún dato producido. Por lo tanto, la inicialización de los semáforos quedaría como sigue:

```
sem_init( &puede_escribir, 0, tam_vector);
sem_init( &puede_leer, 0, 0 );
```

También se ha utilizado un semáforo para exclusión mutua, mutex, para realizar la impresión por pantalla, su inicialización es la que sigue:

```
sem_init( &mutex, 0, 1 );
```

La función sem_wait para el semáforo puede_leer se debe usar en el proceso consumidor, que es el encargado de leer los datos del vector. Y se debe realizar justo antes de leer el dato correspondiente. La función sem_post para este mismo semáforo se debe usar en el proceso productor, y debe ser la última instrucción (a excepción de la instrucción return) ya que indica al proceso consumidor que puede leer el valor producido.
La función sem_wait para el semáforo puede_escribir se debe usar en el proceso producto, que es el encargado de producir los datos y almacenarlos en el vector. Se debe realizar justo antes de insertar el dato en el vector. La función sem_post para este mismo semáforo se debe usar en el proceso consumidor, y debe ser la última instrucción (a excepción de la instrucción return) ya que indica al proceso productor que puede producir un nuevo valor.

El código fuente es el que sigue:

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

using namespace std ;

// constantes
const unsigned
  num_items  = 50 ,
  tam_vector = 10 ;

int vector[tam_vector];
int primera_libre=0;
int primera_ocupada=0;

sem_t
   puede_escribir, // inicializado a tam_vector
   puede_leer,     // inicializado a 0
   mutex ;         // inicializado a 1
unsigned producir_dato()
```

```
{
  static int contador = 0 ;
  cout << "producido: " << contador << endl << flush ;
  return contador++ ;
}

void consumir_dato( int dato )
{}

void * productor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
   int dato = producir_dato() ;
   sem_wait( &puede_escribir ) ;
   vector[primera_libre]=dato;
   primera_libre++;
   sem_post( &puede_leer ) ;
  }
  return NULL ;
}

void * consumidor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
   int dato=0 ;
   sem_wait( &puede_leer ) ;
   dato = vector[primera_libre-1] ; // lee el valor generado
   primera_libre--;
   consumir_dato(dato);
   sem_wait( &mutex ) ;
   cout << "dato recibido: " << dato << endl;
   sem_post( &mutex ) ;
   sem_post( &puede_escribir ) ;
  }
  return NULL ;
}

int main()
{
   pthread_t hebra_escritora, hebra_lectora ;

   sem_init( &mutex,          0, 1 ); // semaforo para EM: inicializado a 1
  sem_init( &puede_escribir, 0, tam_vector ); // inicialmente se puede escribir
   sem_init( &puede_leer,     0, 0 ); // inicialmente no se puede leer

   pthread_create( &hebra_escritora, NULL, productor, NULL );
   pthread_create( &hebra_lectora,   NULL, consumidor, NULL );

   pthread_join( hebra_escritora, NULL ) ;
   pthread_join( hebra_lectora,   NULL ) ;

   sem_destroy( &puede_escribir );
   sem_destroy( &puede_leer );
   sem_destroy( &mutex );
   cout << endl << "fin" << endl;
```

```
   return 0 ;
}
```

El resultado de dicho programa depende del tamaño del vector y del número de valores que queremos calcular. Un ejemplo de la ejecución del programa con un vector de tamaño 10 y 35 elementos sería:

```
producido: 0
producido: 1
producido: 2
producido: 3
producido: 4
producido: 5
producido: 6
producido: 7
producido: 8
producido: 9
producido: 10
dato recibido: 9
dato recibido: 8
dato recibido: 10
dato recibido: 7
dato recibido: 6
dato recibido: 5
dato recibido: 4
dato recibido: 3
dato recibido: 2
dato recibido: 1
dato recibido: 0
producido: 11
producido: 12
producido: 13
producido: 14
producido: 15
producido: 16
producido: 17
producido: 18
producido: 19
producido: 20
producido: 21
dato recibido: 11
dato recibido: 20
dato recibido: 21
dato recibido: 19
dato recibido: 18
dato recibido: 17
dato recibido: 16
dato recibido: 15
dato recibido: 14
dato recibido: 13
dato recibido: 12
producido: 22
producido: 23
producido: 24
producido: 25
producido: 26
producido: 27
producido: 28
```

```
producido: 29
producido: 30
producido: 31
producido: 32
dato recibido: 22
dato recibido: 31
dato recibido: 32
dato recibido: 30
dato recibido: 29
dato recibido: 28
dato recibido: 27
dato recibido: 26
dato recibido: 25
dato recibido: 24
dato recibido: 23
producido: 33
producido: 34
dato recibido: 33
dato recibido: 34

fin
```

3. Práctica 1: Productor-Consumidor (FIFO).

Para realizar la practica usando FIFO (cola circular) para gestionar la ocupación necesitamos dos variables, primera_ocupada y primera_libre. La variable primera_ocupada indica el primer lugar que está ocupado y por lo tanto el primer lugar que se va a leer. Dicha variable se incrementa al leer (módulo tam_vector). La variable primera_libre indica el primer lugar donde se va a poder escribir y se incrementa al escribir (tam_vector). Por lo tanto se va a ir leyendo y escribiendo del vector de izquierda a derecha.

Los semáforos usados son puede_leer y puede_escribir. El semáforo puede_leer determina cuando el proceso consumidor puede leer un dato nuevo del vector y el semáforo puede_escribir determina cuando el proceso productor puede escribir un dato nuevo en el vector. Al inicio de nuestro código, el vector está vacío, por lo tanto el número de veces que se puede escribir es el tamaño del vector y el número de veces que se puede leer es 0, ya que no hay ningún dato producido. Por lo tanto, la inicialización de los semáforos quedaría como sigue:

```
sem_init( &puede_escribir, 0, tam_vector);
sem_init( &puede_leer, 0, 0 );
```

También se ha utilizado un semáforo para exclusión mutua, mutex, para realizar la impresión por pantalla, su inicialización es la que sigue:

```
sem_init( &mutex, 0, 1 );
```

La función sem_wait para el semáforo puede_leer se debe usar en el proceso consumidor, que es el encargado de leer los datos del vector. Y se debe realizar justo antes de leer el dato correspondiente. La función sem_post para este mismo semáforo se debe usar en el proceso productor, y debe ser la última instrucción (a excepción de la instrucción return) ya que indica al proceso consumidor que puede leer el valor producido.
La función sem_wait para el semáforo puede_escribir se debe usar en el proceso producto, que es el encargado de producir los datos y almacenarlos en el vector. Se debe realizar justo antes de insertar el dato en el vector. La función sem_post para este mismo semáforo se debe usar en el proceso consumidor, y debe ser la última instrucción (a excepción de la instrucción return) ya que indica al proceso productor que puede producir un nuevo valor.

El código fuente es el que sigue:

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

using namespace std ;

// constantes
const unsigned
  num_items  = 35 ,
  tam_vector = 10 ;

int vector[tam_vector];
int primera_libre=0;
int primera_ocupada=0;

sem_t
   puede_escribir, // inicializado a tam_vector
   puede_leer,     // inicializado a 0
```

```
   mutex ;         // inicializado a 1

unsigned producir_dato()
{
  static int contador = 0 ;
  cout << "producido: " << contador << endl << flush ;
  return contador++ ;
}

void consumir_dato( int dato )
{}

void * productor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
      int dato = producir_dato() ;
   sem_wait( &puede_escribir ) ;
      vector[primera_libre]=dato;
      primera_libre=(primera_libre+1)%tam_vector;
   sem_post( &puede_leer ) ;
  }
  return NULL ;
}

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

   sem_wait( &puede_leer ) ;
   dato = vector[primera_ocupada] ; // lee el valor generado
      primera_ocupada=(primera_ocupada+1)%tam_vector;
   consumir_dato(dato);
      sem_wait( &mutex ) ;
   cout << "dato recibido: " << dato << endl;
   sem_post( &mutex ) ;
   sem_post( &puede_escribir ) ;
  }
  return NULL ;
}

int main()
{
   pthread_t hebra_escritora, hebra_lectora ;

   sem_init( &mutex,          0, 1 ); // semaforo para EM: inicializado a 1
   sem_init( &puede_escribir, 0, tam_vector ); // inicialmente se puede escribir
   sem_init( &puede_leer,     0, 0 ); // inicialmente no se puede leer

   pthread_create( &hebra_escritora, NULL, productor, NULL );
   pthread_create( &hebra_lectora,   NULL, consumidor, NULL );

   pthread_join( hebra_escritora, NULL ) ;
   pthread_join( hebra_lectora,   NULL ) ;
   sem_destroy( &puede_escribir );
   sem_destroy( &puede_leer );
```

```
   sem_destroy( &mutex );

   cout << endl << "fin" << endl;

   return 0 ;
}
```

El resultado de dicho programa depende del tamaño del vector y del número de valores que queremos calcular. Un ejemplo de la ejecución del programa con un vector de tamaño 10 y 35 elementos sería:

```
producido: 0
producido: 1
dato recibido: producido: 2
producido: 3
producido: 4
producido: 5
producido: 6
0producido: 7
producido: 8
producido: 9
producido: 10

dato recibido: 1
dato recibido: 2
dato recibido: 3
dato recibido: 4
dato recibido: 5
dato recibido: 6
dato recibido: 7
dato recibido: 8
dato recibido: 9
dato recibido: 10
producido: 11
producido: 12
producido: 13
producido: 14
producido: 15
producido: 16
producido: 17
producido: 18
producido: 19
producido: 20
producido: 21
dato recibido: 11
dato recibido: 12
dato recibido: 13
dato recibido: 14
dato recibido: 15
dato recibido: 16
dato recibido: 17
dato recibido: 18
dato recibido: 19
dato recibido: 20
dato recibido: 21
producido: 22
producido: 23
producido: 24
```

```
producido: 25
producido: 26
producido: 27
producido: 28
producido: 29
producido: 30
producido: 31
producido: 32
dato recibido: 22
dato recibido: 23
dato recibido: 24
dato recibido: 25
dato recibido: 26
dato recibido: 27
dato recibido: 28
dato recibido: 29
dato recibido: 30
dato recibido: 31
dato recibido: 32
producido: 33
producido: 34
dato recibido: 33
dato recibido: 34

fin
```

4. Práctica 1: Fumadores.

Los semáforos utilizados para esta práctica son:
sem_t puede_fumar[NUMERO];
sem_t puede_producir;

El semáforo puede_producir se usa para indicar cuando el proceso productor (estanquero) puede producir un material aleatorio. Este semáforo esta inicializado a 1 ya que en el momento inicial el proceso productor puede producir un elemento. La inicialización sería:

```
sem_init( &puede_producir, 0, 1 );
```

La hebra que hace sem_wait sobre este semáforo es la hebra que ejecuta el proceso productor (estanquero), ya que debe esperar a que se pueda producir. Las hebras que hacen sem_signal sobre este semáforo son las hebras que ejecutan el proceso fumar, ya que deben indicar a la hebra que ejecuta el proceso productor (estanquero) que puede producir un nuevo ingrediente.

El semáforo puede_fumar es un vector de semáforos con 3 semáforos, uno para cada hebra. Se usan para indicar cuando el proceso consumidor (fumar) puede consumir un ingrediente producido por el estanquero. Este semáforo esta inicializado a 0 ya que en el momento inicial el proceso productor no ha producido ningún ingrediente y por lo tanto no se puede fumar. La inicialización sería:

```
sem_init( &puede_fumar[NUMERO[1]], 0, 0 );
```

[1] NUMERO es el numero de hebras fumadoras y por lo tanto de semáforos.

Las hebras que hacen sem_wait sobre estos semáforos son las hebras que ejecuta el proceso consumidor (fumar), ya que deben esperar a que la hebra que ejecuta el proceso productor produzca el ingrediente necesario para poder fumar. La hebra que hace sem_signal sobre estos semáforos es la hebra que ejecutan el proceso productor, ya que deben indicar a la hebra que ejecuta el proceso productor (estanquero) que puede fumar. Si el valor ingrediente producido es el 0, se hace sem_signal sobre el semáforo en la posición 0 de nuestro vector para que la hebra que fume sea la 0, al igual para los valores 1 y 2.

El código fuente sería:

```
#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"

using namespace std;

// ---------------------------------------------------------------------------
// función que simula la acción de fumar  como un retardo aleatorio de la hebra

const int NUMERO=3;

sem_t puede_fumar[NUMERO];
sem_t puede_producir;

void * fumador(void * s)
{
  while(true){
```

```
        cout << "Fumador " << (long int) s << " esperando." << endl;
      sem_wait(&puede_fumar[(unsigned int) s]);
      //  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) ;

      sem_post(&puede_producir);
      // retraso bloqueado durante 'miliseg' milisegundos
      usleep( 1000U*miliseg );
      cout << "Fumador " << (long int) s << " fumando." << endl;
   }
}

void * estanquero(void *){
   while(true){
      cout << "El estanquero esta esperando." << endl;
      sem_wait(&puede_producir);
      int aleatorio=rand()%NUMERO;
      cout << "Produzco: " << aleatorio << endl;
      sem_post(&puede_fumar[aleatorio]);
   }
}

int main()
{
   pthread_t hebra_estanquero;
       pthread_t hebra_fumador[NUMERO];

   sem_init( &puede_fumar[NUMERO], 0, 0 );
   sem_init( &puede_producir, 0, 1 );

   pthread_create(&hebra_estanquero, NULL, estanquero, NULL);

       for(unsigned long i=0; i<NUMERO; i++){
             void * arg_ptr=(void *) i;
             pthread_create(&hebra_fumador[i], NULL, fumador, arg_ptr);
   }

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

   pthread_join(hebra_estanquero, NULL);

   sem_destroy( &puede_fumar[NUMERO] );
   sem_destroy( &puede_producir );

   return 0 ;
}
```

5. Práctica 2: Productor-consumidor con buffer limitado.

Lo primero que he cambiado ha sido importar las clases de la carpeta monitor, añadiendo la línea de código:

```
import monitor.*;
```

Lo siguiente ha sido extender la clase para que use AbstractMonitor. A continuación he creado los dos objetos condición que usa mi programa, producir y consumir.

```
private Condition producir = makeCondition();
private Condition consumir = makeCondition();
```

El objeto condición producir es usado para indicar a las hebras si se puede producir. Se usa para avisar a las hebras de que se puede producir cuando el número de datos que hay producidos pero sin consumir (indicado por la variable cont) es menor que el número de datos que caben en el vector (indicado por la variable numSlots).
El objeto condición consumir es usado para indicar a las hebras si se puede consumir. Se usa para avisar a las hebras de que se puede consumir cuando se ha guardado en el buffer el dato. También se usa para avisar a las hebras de que deben esperar cuando no hay datos que leer (indicado por la variable cont cuando vale 0).

Por último, he cambiado los bucles while que había para realizar las esperas de las hebras por condicionales if.

Por último, en los métodos depositar y extraer, para asegurar la exclusión mutua he introducir una llamada a la función enter() antes de empezar ambos métodos. En el caso del método depositar, la última instrucción ejecutada es la orden leave(), pero en el método extraer() hago el leave() antes de devolver el valor consumido.

El código fuente es el que sigue:

```
import monitor.*;

// *****************************************************************************
class Buffer extends AbstractMonitor {
    private int numSlots = 0, cont = 0;
    private double[] buffer = null;
    // Para ver cuando puedo producir
    private Condition producir = makeCondition();
    // Para ver cuando puedo consumir
    private Condition consumir = makeCondition();

    public Buffer(int p_numSlots) {
        numSlots = p_numSlots;
        buffer = new double[numSlots];
    }

    public void depositar(double valor) throws InterruptedException {
        // Exclusión mutua a partir de enter()
        enter();
        // Si no hay espacio (cont == numSlots), espero
        if (cont == numSlots) {
            producir.await();
        }
        // Deposito el valor producido
```

```
        buffer[cont] = valor;
        // Aumento el valor de datos producido
        cont++;
        // Ya se puede consumir
        consumir.signal();
        // Libero la exclusión mutua
        leave();
    }

    public double extraer() throws InterruptedException {
        // Exclusión mutua a partir de enter()
        enter();
        double valor;
        // Si no hay valores, espero a que sean producidos
        if (cont == 0) {
            consumir.await();
        }
        cont--;
        // Guardo el valor ya que no podemos hacer return de un valor compartido
        valor = buffer[cont];
        producir.signal();
        // Libero la exclusión mutua
        leave();
        // Debo liberar la exclusion mutua antes de hacer return
        return valor;
    }
}

// ************************************************************************
class Productor implements Runnable {
    private Buffer bb;
    private int veces, numP;
    public Thread thr;

    public Productor(Buffer pbb, int pveces, int pnumP) {
        bb = pbb;
        veces = pveces;
        numP = pnumP;
        thr = new Thread(this, "productor " + numP);
    }

    public void run() {
        try {
            double item = 100 * numP;

            for (int i = 0; i < veces; i++) {
                System.out.println(thr.getName() + ", produciendo " + item);
                bb.depositar(item++);
            }
        } catch (Exception e) {
            System.err.println("Excepcion en main: " + e);
        }
    }
}

// ************************************************************************
class Consumidor implements Runnable {
    private Buffer bb;
```

```
    private int veces, numC;
    public Thread thr;

    public Consumidor(Buffer pbb, int pveces, int pnumC) {
        bb = pbb;
        veces = pveces;
        numC = pnumC;
        thr = new Thread(this, "consumidor " + numC);
    }

    public void run() {
        try {
            for (int i = 0; i < veces; i++) {
                double item = bb.extraer();
                System.out.println(thr.getName() + ", consumiendo " + item);
            }
        } catch (Exception e) {
            System.err.println("Excepcion en main: " + e);
        }
    }
}

// ****************************************************************************
class MainProductorConsumidor {
    public static void main(String[] args) {
        if (args.length != 5) {
            System.err.println("Uso: ncons nprod tambuf niterp niterc");
            return;
        }

        // leer parametros, crear vectores y buffer intermedio
        Consumidor[] cons = new Consumidor[Integer.parseInt(args[0])];
        Productor[] prod = new Productor[Integer.parseInt(args[1])];
        Buffer buffer = new Buffer(Integer.parseInt(args[2]));
        int iter_cons = Integer.parseInt(args[3]);
        int iter_prod = Integer.parseInt(args[4]);

        if (cons.length * iter_cons != prod.length * iter_prod) {
            System.err.println("no coinciden número de items a producir con a cosumir");
            return;
        }

        // crear hebras
        for (int i = 0; i < cons.length; i++) {
            cons[i] = new Consumidor(buffer, iter_cons, i);
        }
        for (int i = 0; i < prod.length; i++) {
            prod[i] = new Productor(buffer, iter_prod, i);
        }

        // poner en marcha las hebras
        for (int i = 0; i < prod.length; i++) {
            prod[i].thr.start();
        }
        for (int i = 0; i < cons.length; i++) {
            cons[i].thr.start();
        }
    }
```

```
}
```

La salida del programa es:

```
productor 1, produciendo 100.0
productor 0, produciendo 0.0
consumidor 1, consumiendo 100.0
consumidor 0, consumiendo 0.0
productor 1, produciendo 101.0
productor 0, produciendo 1.0
productor 1, produciendo 102.0
consumidor 1, consumiendo 101.0
productor 1, produciendo 103.0
productor 0, produciendo 2.0
consumidor 0, consumiendo 1.0
productor 0, produciendo 3.0
productor 1, produciendo 104.0
consumidor 1, consumiendo 102.0
productor 0, produciendo 4.0
consumidor 0, consumiendo 2.0
consumidor 1, consumiendo 104.0
consumidor 0, consumiendo 4.0
consumidor 1, consumiendo 3.0
consumidor 0, consumiendo 103.0
```

6. Práctica 2: El problema de los fumadores.

En este programa he usado 4 objetos condición, un objetos condición para el estanquero y uno para cada fumador.
El objeto condición estanquero se usa para:
-Indicar cuando el estanquero puede producir el siguiente ingrediente. Esto ocurre en el método obtenerIngrediente, ejecutado por los fumadores, ya que cuando un fumador ha recogido su ingrediente debe avisar al estanquero de que produzca el siguiente ingrediente.
-Indicar cuando el estanquero debe esperar para producir el siguiente ingrediente. Esto ocurro en el método esperarRecogidaIngrediente, ejecutado por el estanquero, ya que si no se ha recogido el ingrediente correspondiente (hay algún booleano del vector de booleanos hay que vale true) debe esperar.
El objeto condición fumadores es un vector de objetos condición, uno para cada fumador. Este vector se usa para:
-Indicar cuando el fumador puede fumar. Esto ocurre en el método ponerIngrediente, ejecutado por el estanquero, ya que este debe avisar al fumador correspondiente indicándole que puede fumar.
-Indicar cuando el fumador correspondiente debe esperar. Esto ocurre en el método obtenerIngrediente, ya que si no esta el ingrediente que necesita para fumar, debe esperar a que el estanquero lo produzca.

El código fuente es el que sigue:

```
import monitor.*;

class Estanco extends AbstractMonitor{
    private boolean[] hay = new boolean[3];
    private Condition estanquero = makeCondition();
    private Condition[] fumadores = new Condition[3];

    public Estanco(){
        //Al principio no hay ningun ingrediente
        for (int i=0; i<3; i++)
            hay[i] = false;
        for (int i=0; i<3; i++)
            fumadores[i] = makeCondition();
    }

    //miIngrediente == ingrediente que quiere la hebra
    public void obtenerIngrediente( int miIngrediente ){
        enter();
        //Si no hay el que quiero, espero
        if (hay[miIngrediente] == false){
            fumadores[miIngrediente].await();
        }
        System.out.println("Fumador " + miIngrediente + " fumando.");
        //Ingrediente recogido
        hay[miIngrediente] = false;
        //Aviso al estanquero para que produzca el siguiente
        estanquero.signal();
        leave();
    }

    //ingrediente == ingredieente generado
    public void ponerIngrediente( int ingrediente ){
        enter();
        //Ingrediente no recogido y producido
        hay[ingrediente] = true;
```

```
        System.out.println("Estanquero produce: " + ingrediente + ".");
        //El fumador correspondiente puede fumar
        fumadores[ingrediente].signal();
        leave();
    }

    public void esperarRecogidaIngrediente(){
        enter();
        //Si no han recogido el ingrediente alguna de las hebras, espero
        if (hay[0] == true || hay[1] == true || hay[2] == true)
            estanquero.await();
        leave();
    }
}

class Fumador implements Runnable{
    int miIngrediente;
    public Thread thr;
    private Estanco estanco;

    public Fumador(Estanco e, int p_miIngrediente){
        estanco = e;
        miIngrediente = p_miIngrediente;
        thr = new Thread(this, "Fumador " + p_miIngrediente);
    }

    public void run(){
        while ( true ){
            estanco.obtenerIngrediente( miIngrediente );
            aux.dormir_max( 2000 );
            System.out.println("Fumador " + miIngrediente + " ha terminado de fumar.");
        }
    }
}

class Estanquero implements Runnable{
    public Thread thr ;
    private Estanco estanco;

    public Estanquero(Estanco e){
        estanco = e;
        thr = new Thread(this, "Estanquero");
    }

    public void run(){
        int ingrediente;
        while (true){
            ingrediente = (int) (Math.random () * 3.0); // 0,1 o 2
            estanco.ponerIngrediente( ingrediente );
            estanco.esperarRecogidaIngrediente() ;
        }
    }
}

class MainFumadores{
    public static void main(String[] args){
        Estanco estanco = new Estanco();
        Estanquero estanquero = new Estanquero(estanco);
```

```
        Fumador[] fumadores = new Fumador[3];

        fumadores[0]= new Fumador(estanco, 0);
        fumadores[1]= new Fumador(estanco, 1);
        fumadores[2]= new Fumador(estanco, 2);

        estanquero.thr.start();
        fumadores[0].thr.start();
        fumadores[1].thr.start();
        fumadores[2].thr.start();
    }
}
```

La salida del programa es:

```
Estanquero produce: 0.
Fumador 0 fumando.
Estanquero produce: 0.
Fumador 0 ha terminado de fumar.
Fumador 0 fumando.
Estanquero produce: 2.
Fumador 2 fumando.
Estanquero produce: 2.
Fumador 0 ha terminado de fumar.
Fumador 2 ha terminado de fumar.
Fumador 2 fumando.
Estanquero produce: 1.
Fumador 1 fumando.
Estanquero produce: 1.
Fumador 1 ha terminado de fumar.
Fumador 1 fumando.
Estanquero produce: 1.
Fumador 2 ha terminado de fumar.
Fumador 1 ha terminado de fumar.
Fumador 1 fumando.
Estanquero produce: 1.
Fumador 1 ha terminado de fumar.
Fumador 1 fumando.
Estanquero produce: 0.
Fumador 0 fumando.
Estanquero produce: 0.
Fumador 0 ha terminado de fumar.
Fumador 0 fumando.
Estanquero produce: 0.
Fumador 1 ha terminado de fumar.
Fumador 0 ha terminado de fumar.
Fumador 0 fumando.
Estanquero produce: 0.
Fumador 0 ha terminado de fumar.
Fumador 0 fumando.
```

7. Práctica 2: El problema del barbero durmiente.

En este programa he usado 3 objetos condición, el objeto sala, el objeto barbero y el objeto silla.
El objeto condición sala se usa para crear una cola de hebras que simula la cola de espera real en la sala de la barbería. Cuando una hebra cliente ejecuta el método cortarPelo, si la silla está ocupada (hay alguien cortándose el pelo) hace un await sobre el objeto condición sala, quedando así en la cola esperando su turno. Cuando la hebra barbero termina de cortar el pelo y elige al siguiente cliente mediante el método siguienteCliente, hace un signal sobre el objeto condición sala para coger la primera hebra en la cola de dicho objeto, que es el primer cliente que debe ser atendido.
El objeto condición silla se usa para indicar si la silla donde se corta el pelo esta ocupada o no, es decir, si el barbero está cortando el pelo o no. Cuando una hebra cliente ejecuta el método cortarPelo se hace un await sobre la silla, indicando que esta está ocupada. Cuando la hebra barbero termina de cortar el pelo y ejecuta el método finCliente, hace un signal sobre el objeto condición silla, indicando así que la silla está vacía.
El objeto condición barbero se usa para:
-Indicar al barbero que debe cortarle el pelo a un cliente. Esto sucede cuando se ejecuta el método cortarPelo.
-Indicar al barbero que puede descansar. Esto sucede cuando se ejecuta el método siguienteCliente y no hay ningún cliente en las colas de espera de los objetos condición silla y sala (no hay clientes en la barbería).

El código fuente es el que sigue:

```
import monitor.*;

class Barberia extends AbstractMonitor {
    private Condition sala = makeCondition();
    private Condition barbero = makeCondition();
    private Condition silla = makeCondition();

    //Invocado por los clientes para cortarse el pelo
    public void cortarPelo() {
        enter();
        if (!silla.isEmpty()) {
            System.out.println("Silla ocupada.");
            sala.await();
        }
        //El barbero empieza a afeitar
        barbero.signal();
        System.out.println("Cliente pelandose.");
        //La silla pasa a estar ocupada
        silla.await();
        leave();
    }

    //Invocado por el barbero para esperar (si procede) a un nueve cliente
    //y sentarlo para el corte
    public void siguienteCliente() {
        enter();
        if (sala.isEmpty() && silla.isEmpty()) {
            System.out.println("Barbero durmiendo.");
            barbero.await();
        }
        System.out.println("Barbero trabajando.");
        sala.signal();
        leave();
    }
```

```
    //Invocado por el barbero para indicar que ha terminado de cortar el pelo
    public void finCliente() {
        enter();
        System.out.println("Barbero termina de cortar.");
        silla.signal();
        leave();
    }
}

class Cliente implements Runnable {
    private Barberia barberia;
    public Thread thr;

    public Cliente(Barberia b) {
        barberia = b;
        thr = new Thread(this, "cliente");
    }

    public void run() {
        while (true) {
            barberia.cortarPelo();  //el cliente espera (si procede) y se corta el pelo
            aux.dormir_max(2000); //el cliente esta fuera de la barberia un tiempo
        }
    }
}

class Barbero implements Runnable {
    private Barberia barberia;
    public Thread thr;

    public Barbero(Barberia mon) {
        barberia = mon;
        thr = new Thread(this, "barbero");
    }

    public void run() {
        while (true) {
            barberia.siguienteCliente();
            aux.dormir_max(2500); //el barbero esta cortando el pelo
            barberia.finCliente();
        }
    }
}

class MainBarbero {
    public static void main(String[] args) {
        Barberia barberia = new Barberia();

        Barbero barbero = new Barbero(barberia);
        Cliente[] clientes = new Cliente[5];
        for (int i = 0; i < 5; i++) {
            clientes[i] = new Cliente(barberia);
        }

        barbero.thr.start();
        for (int i = 0; i < 5; i++) {
            clientes[i].thr.start();
```

```
            }
        }
}
```

La salida del programa es:

```
Barbero durmiendo.
Barbero trabajando.
Cliente pelandose.
Silla ocupada.
Silla ocupada.
Silla ocupada.
Cliente pelandose.
Barbero termina de cortar.
Barbero trabajando.
Cliente pelandose.
Silla ocupada.
Barbero termina de cortar.
Barbero trabajando.
Cliente pelandose.
Silla ocupada.
Barbero termina de cortar.
Barbero trabajando.
Cliente pelandose.
Silla ocupada.
Barbero termina de cortar.
Barbero trabajando.
Cliente pelandose.
Silla ocupada.
Barbero termina de cortar.
Barbero trabajando.
Cliente pelandose.
Silla ocupada.
Barbero termina de cortar.
Barbero trabajando.
Cliente pelandose.
Silla ocupada.
Barbero termina de cortar.
Barbero trabajando.
Cliente pelandose.
```

8. Práctica 3: Productor-consumidor.

Dado que ahora hay 5 productores, el número de iteraciones debe ser 4, ya que 5*4=20. En el método productor mandamos el valor al buffer pero en vez de mandarlo con la etiqueta 0 lo mandamos con la etiqueta Productor, que vale 1 (#define Productor 1).
En el método consumidor cambiamos también el número de iteraciones del bucle por 5, por el mismo motivo que en el método anterior. Cuando realizamos el envío a Buffer lo hacemos con la etiqueta Consumidor.
En el caso del método Buffer cuando vamos a recibir de un productor debemos indicar que vamos a recibir de cualquier etiqueta y de Productor, no de 0. A la hora de recibir de los consumidores ocurre igual. También sucede lo mismo con el envío a Consumidor.
También se han realizado cambios en main para adaptarlo al nuevo modo de ejecución.

El código del programa es el que sigue:

```
#include <mpi.h>
#include <iostream>
#include <math.h>
#include <time.h>      // incluye "time"
#include <unistd.h>    // incluye "usleep"
#include <stdlib.h>    // incluye "rand" y "srand"


#define Productor  1
#define Consumidor 2
#define Buffer     5
#define ITERS      20
#define TAM        5

using namespace std;

void productor(int rank) {
    // 4 valores, por 5 productores serán 20
    for (unsigned int i = 0; i < 4; i++) {
        // Producimos un valor
        cout << "Productor " << rank << " produce valor " << i << endl << flush;

        // espera bloqueado durante un intervalo de tiempo aleatorio
        // (entre una décima de segundo y un segundo)
        usleep( 1000U * (100U+(rand()%900U)) );

        // La etiqueta del productor es 1
        MPI_Ssend(&i, 1, MPI_INT, Buffer, Productor, MPI_COMM_WORLD);
    }
}

void consumidor(int rank) {
    int value, peticion = 1;
    float raiz;
    MPI_Status status;

    // 5 valores, por 4 consumidores serán 20
    for (unsigned int i = 0; i < 5; i++) {

        // El consumidor envia peticion para recibir
        // La etiqueta de envio a consumidor es 2
        MPI_Ssend(&peticion, 1, MPI_INT, Buffer, Consumidor, MPI_COMM_WORLD);
```

```
        // Una vez aceptada la petición le pedimos un valor a Buffer
        // La etiqueta de envio de buffer a consumidor es 0
        MPI_Recv(&value, 1, MPI_INT, Buffer, 0, MPI_COMM_WORLD, &status);
        cout << "Consumidor " << rank << " recibe valor " << value << " de Buffer " <<
endl << flush;

        // espera bloqueado durante un intervalo de tiempo aleatorio
        // (entre una décima de segundo y un segundo)
        usleep( 1000U * (100U+(rand()%900U)) );
        raiz = sqrt(value);
    }
}

void buffer() {
    int        value[TAM] ,
              peticion ,
              pos  =  0,
              rama ;
    MPI_Status status ;

    for (unsigned int i = 0; i < 40; i++) {
        if ( pos==0 )      // el consumidor no puede consumir
         rama = 0 ;
        else if (pos==TAM) // el productor no puede producir
         rama = 1 ;
        else               // ambas guardas son ciertas
        {
            // leer 'status' del siguiente mensaje (esperando si no hay)
            MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status );

            // calcular la rama en función del origen del mensaje
            if ( status.MPI_SOURCE == Productor )
                rama = 0 ;
            else
                rama = 1 ;
        }

        switch (rama) {
            // Recibe del productor
            case 0:
            {
                // Recibe de cualquiera de los productores
                MPI_Recv(&value[pos], 1, MPI_INT, MPI_ANY_SOURCE, Productor,
MPI_COMM_WORLD, &status);
                cout << "Buffer recibe " << value[pos] << " de Prod. " <<
status.MPI_SOURCE << endl << flush;
                pos++;
                break;
            }
            // Envia al consumidor
            case 1:
            {
                // Recibe de cualquiera de los consumidores
                MPI_Recv(&peticion, 1, MPI_INT, MPI_ANY_SOURCE, Consumidor,
MPI_COMM_WORLD, &status);

                // Devuelve al consumidor que le envió la peticion con etiqueta 0
```

```
                MPI_Ssend(&value[pos - 1], 1, MPI_INT, status.MPI_SOURCE, 0,
MPI_COMM_WORLD);
                cout << "Buffer envia " << value[pos - 1] << " a Cons. " <<
status.MPI_SOURCE << endl << flush;
                pos--;
                break;
            }
        }
    }
}

int main(int argc, char *argv[]) {
    int rank, size;

    // inicializar MPI, leer identif. de proceso y número de procesos
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // inicializa la semilla aleatoria:
    srand ( time(NULL) );

    // comprobar el número de procesos con el que el programa
    // ha sido puesto en marcha (debe ser 10)
    if (size != 10) {
        if (rank == 0){
            cout << "Uso: mpirun -np 10 " << argv[0] << endl;
            return 0;
        }
    }

    // Llamamos a los metodos dependiendo de cual sea su numero "rank"
    if (rank < Buffer)
        productor(rank);
    else if (rank == Buffer)
        buffer();
    else
        consumidor(rank);


    // al terminar el proceso, finalizar MPI
    MPI_Finalize();
    return 0;
}
```

La salida del mismo es:

```
Productor 0 produce valor 0
Productor 1 produce valor 0
Productor 3 produce valor 0
Buffer recibe 0 de Prod. 0
Productor 4 produce valor 0
Productor 0 produce valor 1
Buffer recibe 0 de Prod. 1
Productor 1 produce valor 1
Buffer envia 0 a Cons. 8
Consumidor 8 recibe valor 0 de Buffer
```

```
Productor 2 produce valor 0
Buffer envia 0 a Cons. 7
Consumidor 7 recibe valor 0 de Buffer
Buffer recibe 1 de Prod. 0
Buffer recibe 1 de Prod. 1
Productor 0 produce valor 2
Productor 1 produce valor 2
Consumidor 6 recibe valor 1 de Buffer
Buffer envia 1 a Cons. 6
Consumidor 9 recibe valor 1 de Buffer
Buffer envia 1 a Cons. 9
Buffer recibe 2 de Prod. 0
Buffer recibe 2 de Prod. 1
Productor 1 produce valor 3
Productor 0 produce valor 3
Buffer envia 2 a Cons. 8
Consumidor 8 recibe valor 2 de Buffer
Buffer envia 2 a Cons. 7
Consumidor 7 recibe valor 2 de Buffer
Buffer recibe 3 de Prod. 0
Buffer recibe 3 de Prod. 1
Consumidor 6 recibe valor 3 de Buffer
Buffer envia 3 a Cons. 6
Buffer envia 3 a Cons. 9
Consumidor 9 recibe valor 3 de Buffer
Buffer recibe 0 de Prod. 2
Productor 2 produce valor 1
Consumidor 8 recibe valor 0 de Buffer
Buffer envia 0 a Cons. 8
Buffer recibe 1 de Prod. 2
Productor 2 produce valor 2
Buffer envia 1 a Cons. 7
Buffer recibe 2 de Prod. 2
Consumidor 7 recibe valor 1 de Buffer
Productor 2 produce valor 3
Consumidor 6 recibe valor 3 de Buffer
Buffer envia 2 a Cons. 9
Buffer recibe 3 de Prod. 2
Buffer envia 3 a Cons. 6
Buffer recibe 0 de Prod. 3
Consumidor 9 recibe valor 2 de Buffer
Productor 3 produce valor 1
Consumidor 8 recibe valor 0 de Buffer
Buffer envia 0 a Cons. 8
Buffer recibe 1 de Prod. 3
Productor 3 produce valor 2
Consumidor 7 recibe valor 1 de Buffer
Buffer envia 1 a Cons. 7
Buffer recibe 2 de Prod. 3
Productor 3 produce valor 3
Buffer envia 2 a Cons. 6
Buffer recibe 3 de Prod. 3
Buffer envia 3 a Cons. 9
Buffer recibe 0 de Prod. 4
Consumidor 6 recibe valor 2 de Buffer
Consumidor 9 recibe valor 3 de Buffer
Productor 4 produce valor 1
```

```
Buffer envia 0 a Cons. 8
Buffer recibe 1 de Prod. 4
Consumidor 8 recibe valor 0 de Buffer
Productor 4 produce valor 2
Buffer envia 1 a Cons. 7
Buffer recibe 2 de Prod. 4
Productor 4 produce valor 3
Consumidor 7 recibe valor 1 de Buffer
Consumidor 6 recibe valor 2 de Buffer
Buffer envia 2 a Cons. 6
Buffer recibe 3 de Prod. 4
Buffer envia 3 a Cons. 9
Consumidor 9 recibe valor 3 de Buffer
```

9. Práctica 3: Cena de los filósofos.

La situación que conduce a interbloqueo es la que se comentó en teoría. Sucede cuando todos los filósofos cogen el tenedor que tienen a su izquierda y todos se quedan esperando a poder adquirir el tenedor derecho correspondiente para poder comer. Para solucionar esto, cuando es el turno del filósofo 0, en vez de intentar adquirir primero el tenedor de su izquierda, intenta adquirir el tenedor de su derecha. Así nos aseguramos que al menos uno de los filósofos ha cogido el tenedor de su derecha y no causar la situación de interbloqueo.

El código sería el que sigue:

```
#include <iostream>
#include <time.h>      // incluye "time"
#include <unistd.h>    // incluye "usleep"
#include <stdlib.h>    // incluye "rand" y "srand"
#include <mpi.h>

using namespace std;

void Filosofo( int id, int nprocesos);
void Tenedor ( int id, int nprocesos);

// ---------------------------------------------------------------------

int main( int argc, char** argv )
{
   int rank, size;

   srand(time(0));
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );

   if( size!=10)
   {
      if( rank == 0)
         cout<<"El numero de procesos debe ser 10" << endl << flush ;
      MPI_Finalize( );
      return 0;
   }

   if ((rank%2) == 0)
      Filosofo(rank,size); // Los pares son Filosofos
   else
      Tenedor(rank,size);  // Los impares son Tenedores

   MPI_Finalize( );
   return 0;
}
// ---------------------------------------------------------------------

void Filosofo( int id, int nprocesos )
{
   int izq = (id+1) % nprocesos;
   int der = ((id+nprocesos)-1) % nprocesos;

   while(1)
```

```
   {
      // Piensa (espera bloqueada aleatorio del proceso)
      cout << "Filosofo " << id << " PENSANDO" << endl << flush;

      // espera bloqueado durante un intervalo de tiempo aleatorio
      // (entre una décima de segundo y un segundo)
      usleep( 1000U * (100U+(rand()%900U)) );

      if(id == 0){
        // Solicita tenedor derecho
        MPI_Ssend(&der, 1, MPI_INT, der, der, MPI_COMM_WORLD);
        cout << "Filosofo "<<id<< " adquiere tenedor der " << der << endl << flush;

        // Solicita tenedor izquierdo
        MPI_Ssend(&izq, 1, MPI_INT, izq, izq, MPI_COMM_WORLD);
        cout << "Filosofo "<<id<< " adquiere tenedor izq " << izq << endl << flush;
      }
      else{
        // Solicita tenedor izquierdo
        MPI_Ssend(&izq, 1, MPI_INT, izq, izq, MPI_COMM_WORLD);
        cout << "Filosofo "<<id<< " adquiere tenedor izq " << izq << endl << flush;

        // Solicita tenedor derecho
        MPI_Ssend(&der, 1, MPI_INT, der, der, MPI_COMM_WORLD);
        cout << "Filosofo "<<id<< " adquiere tenedor der " << der << endl << flush;
      }

      cout<<"Filosofo "<<id<< " COMIENDO"<<endl<<flush;
      sleep((rand() % 3)+1);  //comiendo

      // Suelta el tenedor izquierdo
      MPI_Ssend(&izq, 1, MPI_INT, izq, izq, MPI_COMM_WORLD);
      cout <<"Filosofo "<<id<< " ha soltado tenedor izq " << izq << endl << flush;

      // Suelta el tenedor derecho
      MPI_Ssend(&der, 1, MPI_INT, der, der, MPI_COMM_WORLD);
      cout <<"Filosofo "<<id<< " ha soltado tenedor der " << der << endl << flush;
 }
}
// ---------------------------------------------------------------------

void Tenedor(int id, int nprocesos)
{
  int buf;
  MPI_Status status;
  int Filo;

  while( true )
  {
    // Espera un peticion desde cualquier filosofo vecino ...
    MPI_Probe(MPI_ANY_SOURCE, id, MPI_COMM_WORLD, &status);

    // Recibe la peticion del filosofo ...
    MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE, id, MPI_COMM_WORLD, &status);

    Filo = status.MPI_SOURCE;

    cout << "Ten. " << id << " recibe petic. de " << Filo << endl << flush;
```

```
      // Espera un peticion desde cualquier filosofo vecino ...
    MPI_Probe(MPI_ANY_SOURCE, id, MPI_COMM_WORLD, &status);
    // Espera a que el filosofo suelte el tenedor...
    MPI_Recv(&buf, 1, MPI_INT, Filo, id, MPI_COMM_WORLD, &status);
    cout << "Ten. " << id << " recibe liberac. de " << Filo << endl << flush;
  }
}
// -----------------------------------------------------------------------
```

La salida obtenida es:

```
Filosofo 8 PENSANDO
Filosofo 2 PENSANDO
Filosofo 4 PENSANDO
Filosofo 0 PENSANDO
Filosofo 6 PENSANDO
Filosofo 2 adquiere tenedor izq 3
Ten. 3 recibe petic. de 2
Ten. 9 recibe petic. de 8
Filosofo 8 adquiere tenedor izq 9
Filosofo 6 adquiere tenedor izq 7
Ten. 7 recibe petic. de 6
Ten. 1 recibe petic. de 2
Filosofo 2 adquiere tenedor der 1
Filosofo 2 COMIENDO
Ten. 5 recibe petic. de 4
Filosofo 4 adquiere tenedor izq 5
Filosofo 2 ha soltado tenedor izq 3
Ten. 3 recibe liberac. de 2
Ten. 3 recibe petic. de 4
Ten. 1 recibe liberac. de 2
Filosofo 4 adquiere tenedor der 3
Filosofo 4 COMIENDO
Filosofo 2 ha soltado tenedor der 1
Filosofo 2 PENSANDO
Filosofo 4 ha soltado tenedor izq 5
Filosofo 4 ha soltado tenedor der 3
Filosofo 4 PENSANDO
Ten. 5 recibe liberac. de 4
Ten. 3 recibe liberac. de 4
Ten. 3 recibe petic. de 2
Filosofo 2 adquiere tenedor izq 3
Ten. 5 recibe petic. de 6
Filosofo 6 adquiere tenedor der 5
Filosofo 6 COMIENDO
Filosofo 2 adquiere tenedor der 1
Ten. 1 recibe petic. de 2
Filosofo 2 COMIENDO
```

10. Práctica 3: Cena de los filósofos con camarero.

Para evitar la situación de interbloqueo comentada en el ejercicio anterior se decide usar un camarero para hacer de intermediario y limitar el número de filósofos que pueden comer a la vez a 4. Para su correcto funcionamiento, cuando un filósofo decide sentarse a comer tras terminar de pensar, debe solicitar sentarse al camarero. El camarero comprobará si hay sitio en la mesa y de ser así, le mandará un mensaje indicándole que puede sentarse. El camarero esta esperando a que un filósofo quiera sentarse si el numero de filósofos comiendo es menor que 4 o que alguien quiera levantarse si ya hay 4 comiendo. Una vez recibe la petición, la procesa, si es posible.

El código del programa es el que sigue:

```
#include <iostream>
#include <time.h>      // incluye "time"
#include <unistd.h>    // incluye "usleep"
#include <stdlib.h>    // incluye "rand" y "srand"
#include <mpi.h>

#define CAMARERO 10

using namespace std;

void Filosofo( int id, int nprocesos);
void Tenedor ( int id, int nprocesos);
void Camarero ();

int sentarse = 111;
int levantarse = 222;

// ---------------------------------------------------------------------

int main( int argc, char** argv )
{
   int rank, size;

   srand(time(0));
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );

   if( size!=11)
   {
      if( rank == 0)
         cout<<"El numero de procesos debe ser 11" << endl << flush ;
      MPI_Finalize( );
      return 0;
   }

       if (rank == 10)
         Camarero(); // El proceso 10 es el camarero
    else if (rank%2 == 0)
        Filosofo(rank,size); // Los pares son Filosofos
    else
        Tenedor(rank,size); // Los impares son Tenedor

   MPI_Finalize( );
```

```
   return 0;
}
// ----------------------------------------------------------------------

void Camarero(){
  int filosofo, contador=0;
    MPI_Status status;
    while (true){
        // El maximo de filosofos comiendo es 4
        if (contador < 4)
                        // Puede sentarse o levantarse
            MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        else
                        // Solo puede levantarse
            MPI_Probe(MPI_ANY_SOURCE, levantarse, MPI_COMM_WORLD, &status);

        if (status.MPI_TAG == sentarse){
                        // Sentarse
            filosofo=status.MPI_SOURCE;
            MPI_Recv(NULL, 0, MPI_INT, filosofo, sentarse, MPI_COMM_WORLD, &status);
            contador++;
            MPI_Send(NULL, 0, MPI_INT, filosofo, sentarse, MPI_COMM_WORLD);
        }

        if (status.MPI_TAG == levantarse){
                        // Levantarse
            filosofo=status.MPI_SOURCE;
            MPI_Recv( NULL, 0, MPI_INT, filosofo, levantarse, MPI_COMM_WORLD, &status);
            contador--;
        }
    }
}

void Filosofo( int id, int nprocesos )
{
   // Ahora hay un proceso mas
   int izq = (id+1) % (nprocesos-1);
   int der = (id+nprocesos-2) % (nprocesos-1);
   MPI_Status status;

   while(1)
   {
      // Piensa (espera bloqueada aleatorio del proceso)
      cout << "Filosofo " << id << " PENSANDO" << endl << flush;

      // espera bloqueado durante un intervalo de tiempo aleatorio
      // (entre una décima de segundo y un segundo)
      usleep( 1000U * (100U+(rand()%900U)) );

      // El filosofo pide sentarse
      cout << "Filosofo " << id << " solicita sentarse " << endl << flush;
      MPI_Send(NULL, 0, MPI_INT, CAMARERO, sentarse, MPI_COMM_WORLD);

      // El filosofo espera a que le digan que puede sentarse
      MPI_Recv(NULL, 0, MPI_INT, CAMARERO, sentarse, MPI_COMM_WORLD, &status);
      cout << "Filosofo " << id << " se sienta " << endl << flush;
      // El filosofo se sienta
```

```
      // Solicita tenedor izquierdo
      MPI_Ssend(&izq, 1, MPI_INT, izq, izq, MPI_COMM_WORLD);
      cout << "Filosofo "<< id << " adquiere tenedor izq " << izq << endl << flush;

      // Solicita tenedor derecho
      MPI_Ssend(&der, 1, MPI_INT, der, der, MPI_COMM_WORLD);
      cout << "Filosofo "<< id << " adquiere tenedor der " << der << endl << flush;

      cout<<"Filosofo "<< id << " COMIENDO"<<endl<<flush;
      sleep((rand() % 3)+1);  //comiendo

      // Suelta el tenedor izquierdo
      MPI_Ssend(&izq, 1, MPI_INT, izq, izq, MPI_COMM_WORLD);
      cout <<"Filosofo "<< id << " ha soltado tenedor izq " << izq << endl << flush;

      // Suelta el tenedor derecho
      MPI_Ssend(&der, 1, MPI_INT, der, der, MPI_COMM_WORLD);
      cout <<"Filosofo "<< id << " ha soltado tenedor der " << der << endl << flush;

      // El filosofo se levanta
      cout << "Filosofo " << id << " se levanta " << endl << flush;
      MPI_Ssend(NULL, 0, MPI_INT, CAMARERO, levantarse, MPI_COMM_WORLD);

 }
}
// ---------------------------------------------------------------------

void Tenedor(int id, int nprocesos)
{
  int buf;
  MPI_Status status;
  int Filo;

  while( true ){
    // Espera un peticion desde cualquier filosofo vecino ...
    MPI_Probe(MPI_ANY_SOURCE, id, MPI_COMM_WORLD, &status);

    // Recibe la peticion del filosofo ...
    MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE, id, MPI_COMM_WORLD, &status);

    Filo = status.MPI_SOURCE;

    cout << "Ten. " << id << " recibe petic. de " << Filo << endl << flush;

        // Espera un peticion desde cualquier filosofo vecino ...
    MPI_Probe(MPI_ANY_SOURCE, id, MPI_COMM_WORLD, &status);
    // Espera a que el filosofo suelte el tenedor...
    MPI_Recv(&buf, 1, MPI_INT, Filo, id, MPI_COMM_WORLD, &status);
    cout << "Ten. " << id << " recibe liberac. de " << Filo << endl << flush;
  }
}
// ---------------------------------------------------------------------
```

La salida producida es:

```
Filosofo 2 PENSANDO
Filosofo 6 PENSANDO
Filosofo 8 PENSANDO
Filosofo 0 PENSANDO
Filosofo 4 PENSANDO
Filosofo 8 solicita sentarse
Filosofo 0 solicita sentarse
Filosofo 6 solicita sentarse
Filosofo 8 se sienta
Filosofo 6 se sienta
Filosofo 0 se sienta
Filosofo 8 adquiere tenedor izq 9
Ten. 9 recibe petic. de 8
Filosofo 4 solicita sentarse
Filosofo 2 solicita sentarse
Filosofo 6 adquiere tenedor izq 7
Ten. 7 recibe petic. de 6
Filosofo 4 se sienta
Filosofo 6 adquiere tenedor der 5
Ten. 5 recibe petic. de 6
Filosofo 6 COMIENDO
Ten. 1 recibe petic. de 0
Filosofo 0 adquiere tenedor izq 1
Ten. 7 recibe liberac. de 6
Filosofo 6 ha soltado tenedor izq 7
Filosofo 6 ha soltado tenedor der 5
Filosofo 6 se levanta
Ten. 5 recibe liberac. de 6
Ten. 5 recibe petic. de 4
Ten. 7 recibe petic. de 8
Filosofo 4 adquiere tenedor izq 5
Filosofo 8 adquiere tenedor der 7
Filosofo 8 COMIENDO
Filosofo 6 PENSANDO
Filosofo 2 se sienta
Filosofo 4 adquiere tenedor der 3
Filosofo 4 COMIENDO
Ten. 3 recibe petic. de 4
Filosofo 6 solicita sentarse
Ten. 5 recibe liberac. de 4
Ten. 3 recibe liberac. de 4
Filosofo 4 ha soltado tenedor izq 5
Filosofo 4 ha soltado tenedor der 3
Filosofo 4 se levanta
Ten. 9 recibe liberac. de 8
Filosofo 8 ha soltado tenedor izq 9
Ten. 3 recibe petic. de 2
Ten. 7 recibe liberac. de 8
Filosofo 8 ha soltado tenedor der 7
Filosofo 8 se levanta
Ten. 9 recibe petic. de 0
Filosofo 2 adquiere tenedor izq 3
Filosofo 0 adquiere tenedor der 9
Filosofo 0 COMIENDO
Filosofo 4 PENSANDO
Filosofo 6 se sienta
```

```
Filosofo 8 PENSANDO
Ten. 7 recibe petic. de 6
Filosofo 6 adquiere tenedor izq 7
Ten. 5 recibe petic. de 6
Filosofo 6 adquiere tenedor der 5
Filosofo 6 COMIENDO
Filosofo 4 solicita sentarse
Filosofo 8 solicita sentarse
Filosofo 8 se sienta
Ten. 1 recibe liberac. de 0
Ten. 1 recibe petic. de 2
Filosofo 0 ha soltado tenedor izq 1
Ten. 9 recibe liberac. de 0
Ten. 9 recibe petic. de 8
```