Sistemas Concurrentes y Distribuidos

Portfolio Prácticas

Seminario 1

Ejercicio : Cálculo secuencial del número Π.
Queremos comprobar que la integral entre [0,1] de f(x) se podría aproximar como,

$$I \approx \frac{1}{m} * \sum_{i=0}^{m-1} f(x_i) \text{ donde } x_i = \frac{i+0.5}{m}$$

Para verificar la corrección del método, hemos usado una I con valor conocido.

$$f(x) = \frac{4}{1+x^2}$$

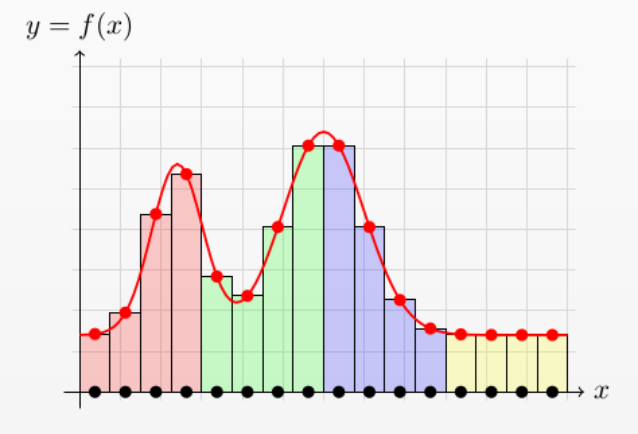
De forma que su integral entre [0,1] de corresponde con el número Π.
Hemos aplicado dos métodos diferentes para resolver el problema:
1.Método secuencial, mediante un único flujo de control realizamos todos los cálculos.

```
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 $f$
}
```

2. Método concurrente, creamos una serie de hebras (n) de forma que cada hilo hará m/n cálculos, agilizándose así el proceso de cálculo. Este método lo hemos dividido en dos funciones, la primera creará las hebras a las cuales les asignará la segunda función, que calcula las sumas parciales, y sumará el vector donde se almacenan las sumas parciales.

-La función calcular_integral_concurrente se encarga de crear las cuatro hebras, como todas van a hacer cálculos sobre una misma función le asignamos a cada una un identificador, los cuales almacenamos en un vector. La función esperara hasta que todas las hebras hayan calculado su suma parcial para realizar la suma total y devolverla.

-La funcion_hebra le asigna a cada hebra un rango de datos sobre los que hacer los cálculos, una vez termina almacena la suma parcial en un vector global, de este modo las distintas hebras depositarán, según su código de identificación, el resultado.



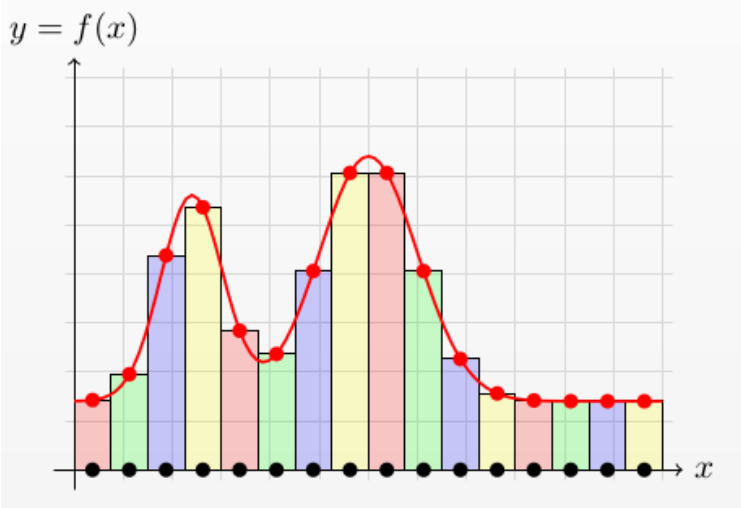

```
void * funcion_hebra( void * ih_void ) {
  unsigned long ih = (unsigned long) ih_void ;
// número o índice de esta hebra
  double sumap = 0.0;

  // calcular suma parcial en "sumap"
  for(unsigned long i = ih*(m/n);i < ((ih+1)*m/n) ; i++){
        sumap += f((i+0.5)/m);
  }

  resultado_parcial[ih] = sumap; // guardar suma parcial en vector.
  return NULL ;
}
// ---------------------------------------------------------------------
// cálculo concurrente

double calcular_integral_concurrente( )
{
  // crear y lanzar $n$ hebras, cada una ejecuta
"funcion\_concurrente"

  pthread_t id_hebra[n] ; // vector de identificadores de hebra

  for(unsigned long i = 0; i < n; i++){
        void * ih_void = (void *) i ; // convertir entero a puntero
      pthread_create( &(id_hebra[i]), NULL, funcion_hebra, ih_void );
  }

 // esperar (join) a que termine cada hebra, sumar su resultado
  for( unsigned i = 0 ; i < n ; i++ ){
        pthread_join( id_hebra[i], NULL);
  }

  double res = 0.0;
  // devolver resultado completo
  for(int i = 0; i < n; i++){
        res += resultado_parcial[i];
  }

  return res/m;
```

```
}
```

Si en vez de asignarle a cada hebra un rango de datos queremos que se entrelacen entonces el código debemos usar es el siguiente:

```
void * funcion_hebra( void * ih_void ) {
  unsigned long ih = (unsigned long) ih_void ; // número o índice de esta hebra
  double sumap = 0.0;

  //cálculo de la suma parcial con las hebras entrelazadas
  for (int i = 0; i <m; i++){
        if(ih == i%n)
              sumap += f((i+0.5)/m);
  }

  resultado_parcial[ih] = sumap; // guardar suma parcial en vector.
  return NULL ;
}
```

La representación gráfica de cómo están comportándose las hebras sería la siguiente



El resultado ha sido el mismo haciendo el cálculo con programación secuencial y concurrente, sin embargo usando programación concurrente se han agilizado los cálculos.
Para comprobar la mejoría en el tiempo de ejecución hemos usado la función clock_gettime, sirve para medir tiempo real transcurrido con precisión de nanosegundos. Los archivos que implementan esta función nos han sido proporcionados. La implementación en nuestro programa quedaría como sigue:

```
int main(){
  cout << endl << "Ejemplo 4 (cálculo de PI)" << endl ;
  double pi_sec = 0.0, pi_conc = 0.0 ;

  struct timespec inicio1 = ahora();
  pi_sec  = calcular_integral_secuencial() ;
  struct timespec fin1 = ahora();
```

```
  struct timespec inicio2 = ahora();
  pi_conc = calcular_integral_concurrente() ;
  struct timespec fin2 = ahora();

  cout << "valor de PI (calculado secuencialmente)  == " << pi_sec  <<
endl
       << "tiempo transcurrido usando programación secuencial " <<
duracion(&inicio1,&fin1) << " seg" << endl << endl
       << "valor de PI (calculado concurrentemente) == " << pi_conc <<
endl
        << "tiempo transcurrido usando programación concurrente " <<
duracion(&inicio2,&fin2) << " seg" << endl;
  cout << endl;
  return 0 ;
}
```

Hemos calculado el tiempo de ejecución de ambas funciones, y los resultados obtenidos son los que se esperaban, la función concurrente hace el cálculo más rápido que la función secuencial.

```
Ejemplo 4 (cálculo de PI)
valor de PI (calculado secuencialmente)  == 3.14159
tiempo transcurrido usando programación secuencial 0.0157523 seg

valor de PI (calculado concurrentemente) == 3.14159
tiempo transcurrido usando programación concurrente 0.0102408 seg
```

Practica 1

Problema del productor consumidor.

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

► En general, el productor calcula o produce una secuencia de items de datos (uno a uno), y el consumidor lee o consume dichos items (también uno a uno).
► El tiempo que se tarda en producir un item de datos puede ser variable y en general distinto al que se tarda en consumirlo.

La solución ha este problema la hemos diseñado de la siguiente forma:

- Hemos creado dos hebras, una es la productora y otra la consumidora.
- Hemos añadido dos semáforos, uno para controlar la lectura y otro para controlar la escritura, así aseguramos la exclusión mutua.
- Las funciones de tiempo de espera, nos las han dado ya.
- En cuanto a las variables y a los tipos de datos usados, en general;

buffer, es un vector que almacena los items que el productor crea.
num_items, es el número de items que puede llegar a producir el productor.
tam_vector, es el número de items que le caben al vector.
primera_libre, es la primera posición donde el productor puede introducir un dato.

Para la solución y el diseño general de este problema existen dos tipos de soluciones, una se basa en un diseño **LIFO**(last in first out), que constaría de todos los elementos mencionados anteriormente, y un diseño **FIFO**(first in first out), que tendría todos los elementos anteriores y además una variable extra, primera_ocupada, es la primera posición que el consumidor va a coger.

El diseño del algoritmo para un funcionamiento **LIFO** ha sido el siguiente:

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

   // insertar "dato" en el vector o buffer
   sem_wait(&puede_producir);
   buf[primera_libre] = dato;
   primera_libre++;
   sem_post(&puede_consumir);

   cout << "Productor :      dato insertado: " << dato << endl << flush ;
  }

  return NULL ;
}
// ----------------------------------------------------------------------
// función que ejecuta la hebra del consumidor

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

   // leer "dato" desde el vector intermedio
   sem_wait(&puede_consumir);
   dato = buf[primera_libre - 1];
   primera_libre--;
   sem_post(&puede_producir);

   cout << "Consumidor:    dato extraído : " << dato << endl << flush ;
   consumir_dato( dato ) ;
  }
   return NULL ;
}
```

Usando sem_wait y sem_post aseguramos la exclusión mutua, de forma que el productor no pueda crear datos si el vector está lleno y el consumidor no pueda consumir datos si éste está vacío. La declaración de los semáforos se hace de forma global. La creación de los semáforos se ha hecho, en el main, de la siguiente manera:

```
sem_init(&puede_producir, 0,tam_vector);
sem_init(&puede_consumir,0,0);
```

El diseño del algoritmo para un funcionamiento **FIFO** ha sido el siguiente:

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

    //insertar "dato" en el vector o buffer
    sem_wait(&puede_leer);
    buf[primera_libre] = dato;
    primera_libre++;
    //llego al final del vector para poder escribir otra vez al principio
tengo que comprobar primero que la casilla inicial esta vacía
    if(primera_libre >= tam_vector && primera_ocupada > 0)
      primera_libre = 0;
    sem_post(&puede_escribir);

    cout << "Productor :      dato insertado: " << dato << endl << flush ;
  }

  return NULL ;
}
// ----------------------------------------------------------------------
// función que ejecuta la hebra del consumidor

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

    // leer "dato" desde el vector intermedio
    sem_wait(&puede_escribir);
    dato = buf[primera_ocupada];
    primera_ocupada++;
    //para poder leer tengo que comprobar que al princpio hay datos
escritos
    if(primera_ocupada >= tam_vector && primera_libre > 0)
      primera_ocupada = 0;

    sem_post(&puede_leer);

    cout << "Consumidor:    dato extraído : " << dato << endl << flush ;
    consumir_dato( dato ) ;
  }
  return NULL ;
}
```

Cuestiones ha tener en cuenta en el diseño de funcion_productor:

- Exclusión mutua, mientras uno produce no se puede leer.
- Si primera_libre es igual a tam_vector tenemos que ponerla a 0, sino lo hacemos nos saldrá un core, y el productor no podrá seguir escribiendo, además tenemos que comprobar que la

primera_ocupada es mayor que 0, porque sino estaríamos sobrescribiendo un valor.

Cuestiones ha tener en cuenta en el diseño de función_consumidor:

- Exclusión mutua.
- Si primera_ocupada es igual a tam_vector entonces debemos ponerla a 0, para que pueda seguir consumiendo datos, sin embargo antes debemos comprobar que, la primera casilla del vector está ocupada, es decir, que primera_ocupada es mayor que 0.

Problema de los fumadores

Considerar un estanco en el que hay tres fumadores y un estanquero.

Cada fumador representa una hebra que realiza una actividad (fumar), invocando a una función fumar, en un bucle infinito.

- Cada fumador debe esperar antes de fumar a que se den ciertas condiciones (tener suministros para fumar), que dependen de la actividad del proceso que representa al estanquero.
- El estanquero produce suministros para que los fumadores puedan fumar, también en un bucle infinito.
- Debe permitir que varios fumadores fumen simultáneamente

A continuación se describen los requisitos para que los fumadores puedan fumar y el funcionamiento del proceso estanquero:

- Antes de fumar es necesario liar un cigarro, para ello el fumador necesita tres ingredientes: tabaco, papel y cerillas (0,1,2).
- Uno de los fumadores tiene papel y tabaco (0,1), otro tiene papel y cerillas (1,2) , y otro tabaco y cerillas (0,2) .
- El estanquero selecciona aleatoriamente un ingrediente de los tres que se necesitan para hacer un cigarro, lo pone en el mostrador, desbloquea al fumador que necesita dicho ingrediente y después se bloquea, esperando la retirada del ingrediente.
- El fumador desbloqueado toma el ingrediente del mostrador, desbloquea al estanquero para que pueda seguir sirviendo ingredientes y después fuma durante un tiempo aleatorio.
- El estanquero, cuando se desbloquea, vuelve a poner un ingrediente aleatorio en el mostrador, y se repite el ciclo.

He diseñado la solución de la siguiente manera:

1. Cinco semáforos, uno para el estanquero, tres diferentes para los fumadores y otro para garantizar exclusión mutua para mostrar mensajes por pantalla.
2. Cuatro hebras, una para el estanquero asociada a funcion_estanquero y tres para los fumadores asociadas a funcion_fumadores.

```
#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 ;
sem_t puede_producir;
//cada semáforo se corresponde con un fumador y el productor
sem_t puede_fumar[3];
sem_t mutex;
void retraso_aleatorio( const float smin, const float smax ){
  static bool primera = true ;
  if ( primera ){        // si es la primera vez:
    srand(time(NULL)); //   inicializar la semilla del generador
    primera = false ;  //   no repetir la inicialización
  }
  const float tsec = smin+(smax-smin)*((float)random()/(float)RAND_MAX);
  usleep( (useconds_t) (tsec*1000000.0)  );
}
void * fumar( unsigned long num_fumador){

  sem_wait(&mutex);
   cout << "Fumador número " << num_fumador << ": comienza a fumar." <<
endl << flush ;
  sem_post(&mutex);

  retraso_aleatorio( 0.2, 0.8 );
  sem_wait(&mutex);
   cout << "Fumador número " << num_fumador << ": termina de fumar." <<
endl << flush ;
  sem_post(&mutex);
}
```

```
unsigned int producir_dato(){
  unsigned int dato = (rand() % 3U);  //producir un numero aleatorio de 0 a 2
  retraso_aleatorio( 0.1, 0.5 );
  return dato;
}
void * funcion_estanquero(void *){
  while(true){
    sem_wait(&puede_producir);
    unsigned int dato = producir_dato();
    sem_wait(&mutex);
    cout << "Producido el ingrediente: " << dato << endl;
    sem_post(&mutex);
    sem_post(&puede_fumar[dato]);
}
  return NULL;
}

void * funcion_fumador(void * num_fumador){
  unsigned long num_fum = (unsigned long) num_fumador;
  while(true){
    sem_wait(&puede_fumar[num_fum]);
    sem_post(&puede_producir);
    fumar(num_fum);
  }
  return NULL;
}
// -----------------------------------------------------------------------

int main(){
   srand( time(NULL) ); // inicializa semilla aleatoria para selección aleatoria de fumador
```

```
  pthread_t fumador_1, fumador_2, fumador_3, estanquero; // vector de hebras, cada hebra es un fumador


  pthread_create( &fumador_1, NULL, funcion_fumador, (void *)0);
  pthread_create( &fumador_2, NULL, funcion_fumador, (void *)1);
  pthread_create( &fumador_3, NULL, funcion_fumador, (void *)2);
  pthread_create( &estanquero, NULL, funcion_estanquero, NULL);


  //inicializo los semáforos
  //primer semáforo, fumador0 puede_fumar0, no puede fumar porque le falta elemento 2
  sem_init(&puede_fumar[0],0,0);
   //segundo semaforo, fumador1 puede_fumar1, no puede fumar porque le falta elemento 0
  sem_init(&puede_fumar[1],0,0);
  //tercer semaforo, fumador2 puede_fumar2, no puede fumar porque le falta elemento 1
  sem_init(&puede_fumar[2],0,0);
  //cuarto semaforo, puede_producir, produce un elemento aleatorio
  sem_init(&puede_producir,0,1);
  sem_init(&mutex,0,1);


  pthread_join(fumador_1,NULL);
  pthread_join(fumador_2,NULL);
  pthread_join(fumador_3,NULL);
  pthread_join(estanquero,NULL);


  sem_destroy(&puede_producir);
  sem_destroy(&puede_fumar[0]);
  sem_destroy(&puede_fumar[1]);
  sem_destroy(&puede_fumar[2]);
```

```
  return 0 ;
}
```

PRÁCTICA 2

Problema de los fumadores con monitores Hoare

En este ejercicio consideraremos de nuevo el mismo problema de los fumadores y el estanquero que ya vimos en la práctica 1:

- Se mantienen exactamente igual todas las condiciones de sincronización entre las distintas hebras involucradas.
- Tenemos una clase hebra Estanquero y otra Fumador . De esta última habrá tres instancias, cada una almacenará el número de ingrediente que necesita (que es equivalente: el número de fumador).
- La interacción entre los fumadores y el estanquero será resuelta mediante un monitor Estanco basado en el paquete monitor.
- Tenemos además cuatro colas condición, puede_fumar y puede_producir:
  - puede_fumar, es un array de tres colas condición, en ellas irán los fumadores según el ingrediente que necesiten. Un fumador podrá fumar si su ingrediente está en el estanco.
  - puede_producir, indica cuando podrá el estanquero poner un ingrediente en el mostrador, esto es cuando esté vacío, el estado del mostrador vendrá dado por la variable ingrediente, que podrá almacenar tanto los ingredientes de los fumadores, 0,1,2 como -1 para indicar que el mostrador está vacío.

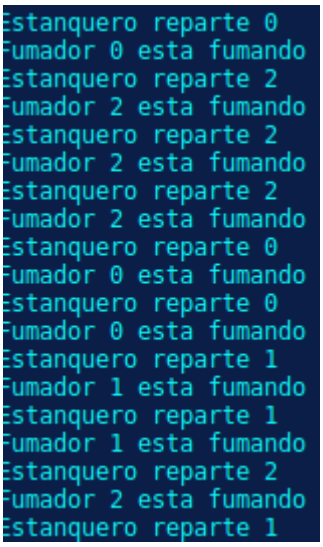
```
class Estanco extends AbstractMonitor{
        private int ingrediente;
        private Condition puede_producir = makeCondition();
        private Condition [] puede_fumar = new Condition[3];
        public Estanco(){
                for(int i = 0; i < 3;i++)
                        puede_fumar[i] = makeCondition();
                ingrediente = -1;
        }
        public void obtenerIngrediente( int miIngrediente ){
                enter();
```

```
                    if(ingrediente != miIngrediente)
                        puede_fumar[miIngrediente].await();
                    ingrediente = -1;
                    puede_producir.signal();
                leave();
            }
            public void ponerIngrediente( int ingrediente )  {
                enter();
                    this.ingrediente = ingrediente;
                    puede_fumar[this.ingrediente].signal();
                leave();
            }
            public void esperarRecogidaIngrediente(){
                enter();
                    if(ingrediente != -1)
                        puede_producir.await();
                leave();
            }
}
```

```
class Fumador implements Runnable{
	int miIngrediente;
	public Thread thr ;
	public Estanco estanco;
	public Fumador(Estanco miEstanco, int p_miIngrediente, String nombre){
		estanco = miEstanco;
		miIngrediente = p_miIngrediente;
		thr = new Thread(this,nombre);
	}
	public void run(){
		while ( true ){
			estanco.obtenerIngrediente(miIngrediente);
			aux.dormir_max(2000);
		}
	}
}

class Estanquero implements Runnable{
	public Thread thr ;
	public Estanco miEstanco;
	public Estanquero(Estanco estanco, String nombre){
		miEstanco = estanco;
		thr = new Thread(this, nombre);
	}
	public void run(){
		int ingrediente ;
		while (true){
			ingrediente = (int)(Math.random () * 3.0); // 0,1 o 2
			miEstanco.ponerIngrediente( ingrediente );
			miEstanco.esperarRecogidaIngrediente() ;
		}
	}
}

class MainFumadores {
```

```
  public static void main(String[] args) {
    // crear monitor
    Estanco estan = new Estanco();

    // crear los dos vectores de hebras:
    Fumador [] fumadores = new Fumador [3];
    Estanquero estanq = new Estanquero(estan,"estanquero");

    // crear hebras
    for( int i = 0; i < 3; i++)
      fumadores[i] = new Fumador(estan,i,"fumador"+(i));

    estanq.thr.start();
    // lanzar hebras
   for( int i = 0; i < 3; i++)
          fumadores[i].thr.start();
  }
}
```

Esta solución no usa un buffer para el estanco, los siguientes ejercicios serán fumadoresLIFO y fumadoresFIFO, en los que se sigue la metodología anterior pero usando un buffer.



Problema de los fumadoresLIFO

Contiene las mismas varibles usadas en el ejercicio anterior y además:

- primera_libre, como estamos usando el criterio first in last out usamos esta variable para sacar los elementos.
- Tam, que es el tamaño del mostrador del estanco
- mostrador, que es el buffer donde se almacenan los ingredientes
- estanco_lleno que es una variable boleana que hará esperar al estanquero para que no produzca.
- La condicion puede_producir depende de si el estanco está lleno o no.

El código usado es el siguiente:

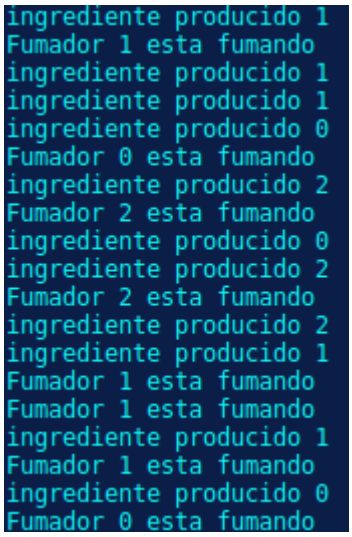
```
/*PROBLEMA DE LOS FUMADORES LIFO*/


import monitor.*;
/************************************************/

class Estanco extends AbstractMonitor{
  private int ingrediente;
  private int primera_libre;
  private int tam;
  private int [] mostrador;

  private Condition puede_producir = makeCondition();
  private Condition [] puede_fumar = new Condition[3];

  public Estanco(){
     for(int i = 0; i < 3; i++)
       puede_fumar[i] = makeCondition();

     ingrediente = -1;
     primera_libre = 0;
     tam = 10;
     mostrador = new int [tam];
  }

  public void obtenerIngrediente(int miIngrediente){
     enter();
       //si primera libre es 0 y el estanco esta lleno
       if(ingrediente != miIngrediente || primera_libre == 0)
         puede_fumar[miIngrediente].await();
```

```
        System.out.println("Fumador "+miIngrediente+" esta
fumando");
        primera_libre--;
        puede_producir.signal();
      leave();
    }

    public void ponerIngrediente(int ingrediente){
      enter();
        if(primera_libre == tam)
          puede_producir.await();

        mostrador[primera_libre] = ingrediente;
        primera_libre++;
        this.ingrediente = ingrediente;
        System.out.println("ingrediente producido
"+ingrediente);
        puede_fumar[this.ingrediente].signal();
      leave();
    }

    public void esperarRecogidaIngrediente(){
      enter();
        aux.dormir_max(2000);
      leave();
    }
}

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

    public Fumador(Estanco miEstanco, int p_miIngrediente,
String nombre){
      estanco = miEstanco;
      miIngrediente = p_miIngrediente;
      thr = new Thread(this,nombre);
    }

    public void run(){
      while(true){
        estanco.obtenerIngrediente(miIngrediente);
```

```
            aux.dormir_max(2000);
        }
    }
}

class Estanquero implements Runnable{
    public Thread thr;
    public Estanco miEstanco;

    public Estanquero(Estanco estanco, String nombre){
        miEstanco = estanco;
        thr = new Thread(this,nombre);
    }

    public void run(){
        int ingrediente;
        while(true){
            ingrediente = (int)(Math.random()*3.0);
            miEstanco.ponerIngrediente(ingrediente);
            miEstanco.esperarRecogidaIngrediente();
        }
    }
}



class MainFumadores {
 public static void main(String[] args) {
  // crear monitor
  Estanco estan = new Estanco();

  // crear los dos vectores de hebras:
  Fumador [] fumadores = new Fumador [3];
  Estanquero estanq = new Estanquero(estan,"estanquero");

  // crear hebras
  for( int i = 0; i < 3; i++)
   fumadores[i] = new Fumador(estan,i,"fumador"+(i));

  estanq.thr.start();
  // lanzar hebras
  for( int i = 0; i < 3; i++)
  fumadores[i].thr.start();
```

```
 }
}
```

La clase main es la misma que en el ejercicio anterior. La salida del programa es la siguiente,



Problema de los FumadoresFIFO

Este ejercicio es parecido al anterior, tenemos una hebra estanquero que pone los elementos en el buffer, mostrador, y tres hebras fumadoras, que necesitan un ingrediente determinado cada una. Las variables usadas en el monitor han sido:

- ingrediente, es el ingrediente que se va a producir de forma aleatoria, en el rango 0-2
- primera_libre, empieza siendo 0, primera casilla donde el estanquero dejará el tabaco.
- primera_ocupada, es la casilla desde donde los fumadores tomarán el ingrediente, siguiendo el criterio FIFO, la primera en entrar es la primera en salir.
- Tam, es el tamaño del mostrador
- mostrador, es el buffer donde se almacenan los ingredientes
- puede_producir, cola condición donde el estanquero esperará si el mostrador está lleno.
- puede_fumar, array de colas condición donde los fumadores esperarán si:
  - el mostrador está vacío

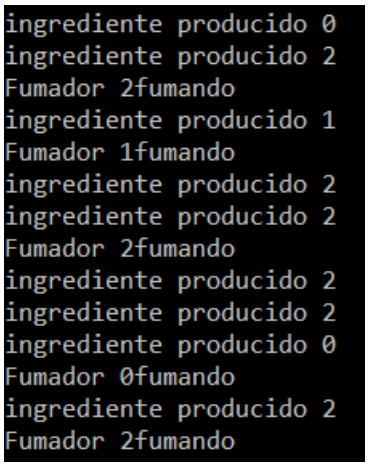
- no se ha producido el ingrediente que esperan

```
/*PROBLEMA DE LOS FUMADORESFIFO*/

import monitor.*;
/*****************************************/

class Estanco extends AbstractMonitor{
  private int ingrediente;
  private int primera_libre;
  private int primera_ocupada;
  private int tam;
  private int [] mostrador;

  private Condition puede_producir = makeCondition();
  private Condition [] puede_fumar = new Condition[3];

  public Estanco(){
     for(int i = 0; i < 3;i++)
       puede_fumar[i] = makeCondition();

     primera_libre = 0;
     primera_ocupada = 0;
     tam = 10;
     mostrador = new int [10];
  }

  public void obtenerIngrediente(int miIngrediente){
     enter();
//si la primera libre es 0 entonces aun no hay ningun
ingrediente en el mostrador
       if(ingrediente != miIngrediente || primera_libre > 0)
         puede_fumar[miIngrediente].await();
       System.out.println("Fumador "+miIngrediente+"fumando");
 //si coge el ingrediente de una posicion entonces el
siguiente ingrediente esta en la pos++
       primera_ocupada++;
//si la primera ocupada es al final del vector y la primera
libre es distinta de 0, entonces sabemos
       if(primera_ocupada == tam && primera_libre != 0)
         primera_ocupada = 0;
     leave();
  }

  public void ponerIngrediente(int ingrediente){
```

```
      enter();
//el mostrador esta lleno
        if(primera_libre == tam)
          puede_producir.await();

        mostrador[primera_libre] = ingrediente;
        primera_libre++;
        this.ingrediente = ingrediente;
        System.out.println("ingrediente producido
"+ingrediente);
//si la primera libre es al final del mostrador y la primera
ocupada no esta al principio
//entonces el principio está libre y el estanquero puede
dejar el ingrediente al principio
//no hace el wait
        if(primera_ocupada > 0 && primera_libre == tam)
          primera_libre = 0;
//le da permiso al fumador para fumar
        puede_fumar[this.ingrediente].signal();
      leave();
   }
   public void esperarRecogidaIngrediente(){
      enter();
        dormir.dormir_max(2000);
      leave();
   }
}

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

   public Fumador(Estanco miEstanco, int p_miIngrediente,
String nombre){
      estanco = miEstanco;
      miIngrediente = p_miIngrediente;
      thr = new Thread(this,nombre);
   }

   public void run(){
      while(true){
        estanco.obtenerIngrediente(miIngrediente);
```

```
            dormir.dormir_max(2000);
        }
    }
}

class Estanquero implements Runnable{
    public Thread thr;
    public Estanco miEstanco;

    public Estanquero(Estanco estanco, String nombre){
        miEstanco = estanco;
        thr = new Thread(this,nombre);
    }

    public void run(){
        int ingrediente;
        while(true){
            ingrediente = (int)(Math.random()*3.0);
            miEstanco.ponerIngrediente(ingrediente);
            miEstanco.esperarRecogidaIngrediente();
        }
    }
}

class MainFumadores {
 public static void main(String[] args) {
// crear monitor
  Estanco estan = new Estanco();

 // crear los dos vectores de hebras:
  Fumador [] fumadores = new Fumador [3];
  Estanquero estanq = new Estanquero(estan,"estanquero");

 // crear hebras
  for( int i = 0; i < 3; i++)
   fumadores[i] = new Fumador(estan,i,"fumador"+(i));

  estanq.thr.start();
 // lanzar hebras
  for( int i = 0; i < 3; i++)
  fumadores[i].thr.start();
 }
}
```

Ejemplo de ejecución:



Problema de la Barbería

Descripción

El problema del barbero durmiente es representativo de cierto tipo de problemas reales: ilustra perfectamente la relación de tipo cliente-servidor que a menudo aparece entre los procesos.

→ El problema trata sobre una barbería en la cual hay dos tipos de actores o hebras ejecutándose concurrentemente:

- Una única hebra llamada barbero
- Varias hebras llamadas clientes (un número fijo).

→ En la barbería hay:

- una única silla de cortar el pelo
- una sala de espera para los clientes, con una cantidad de sillas al

menos igual al número de clientes

→Las hebras no consumen CPU en estos casos:

- Barbero: cuando no hay clientes que atender (decimos que duerme),

cuando le está cortando el pelo a un cliente.

- Cliente: cuando está en la sala de espera, cuando el barbero le está

cortando el pelo, cuando está fuera de la barbería

Requerimientos de la sincronización

La sincronización entre hebras viene determinada por estas condiciones:

El barbero ejecuta un bucle infinito, en cada iteración:

- Si no hay clientes en la sala de espera, espera dormido a que llegue un cliente a la barbería, o bien, si hay clientes en dicha sala, llama a uno de ellos.
- Pela al cliente durante un intervalo de tiempo, cuya duración exacta la determina el barbero.
- Avisa al cliente de que ha terminado de pelarlo.

Cada cliente ejecuta un bucle infinito, en cada iteración:

- Entra a la barbería. Si el barbero está ocupado pelando, esperará en la sala de espera hasta que el barbero lo llame, o bien, si el barbero está dormido, el cliente despierta al barbero.
- Espera en la silla de corte a que el barbero lo pele, hasta que el barbero le avisa de que ha terminado. Sale de la barbería.
- Espera fuera de la barbería durante un intervalo de tiempo, cuya duración exacta la determina el cliente.

```
import java.util.Random;

import monitor.*;

class Barberia extends AbstractMonitor{
// sala_espera, un cliente pasará a la sala de espera sii la
silla está ocupada ó el barbero esta dormido
 private Condition sala_espera = makeCondition();
// barbero dormirá si la sala de espera está vacía
 private Condition barbero = makeCondition();
// silla estará ocupada si el barbero está pelando a alguien
 private Condition silla = makeCondition();

// invocado por los clientes para cortarse el pelo
 public void cortarPelo(){
  enter();
   // Si la silla está ocupada
   if (!silla.isEmpty()){
    //pasamos el cliente a la sala de espera
    System.out.println("Silla ocupada");
    sala_espera.await();
   }
 // Pelamos al cliente (despertamos al barbero y ponemos al
cliente en la silla)
   System.out.println("Cliente empieza a afeitarse ");
   barbero.signal();
   silla.await();
```

```
  leave();
 }

public void siguienteCliente(){
  enter();
   // Si la sala y la silla están vacias
   if (sala_espera.isEmpty() && silla.isEmpty()){
    System.out.println("Barbero se pone a dormir ");
    barbero.await();
   }
 // Sacamos al siguiente cliente de la sala de espera
   System.out.println("Barbero coge otro cliente ");
   sala_espera.signal();
   leave();
 }

 public void finCliente(){
  enter();
 // Sacamos al cliente de la silla
   System.out.println("Barbero termina de afeitar ");
   silla.signal();
  leave();
 }
}

class Cliente implements Runnable{
 private Barberia barberia;
 public Thread thr;
 public Cliente(Barberia bar,String nombre){
  barberia = bar;
  thr = new Thread(this,nombre);
 }

 public void run(){
  while (true){
   try{
    barberia.cortarPelo();
// el cliente espera (si procede) y se corta el pelo
    dormir.dormir_max( 2000 );
// el cliente esta fuera de la barberia un tiempo
   }
   catch(Exception e){
    System.err.println("Excepcion en main: " + e);
```

```
   }
  }
 }
}

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){
   try{
    barberia.siguienteCliente();
    dormir.dormir_max( 2500 );
// el barbero esta cortando el pelo
    barberia.finCliente();
   }
   catch(Exception e){
    System.err.println("Excepcion en main: " + e);
   }
  }
 }
}

class MainBArberia{
 public static void main(String[] args){

// leer parametros, crear vectores y buffer intermedio
  Barberia barberia = new Barberia();

 // crear hebras
  Barbero barbero = new Barbero(barberia);
  Cliente[] clientes = new Cliente[4];

  for (int i=0; i < 4; i++)
   clientes[i] = new Cliente(barberia,"cliente"+i);

 // poner en marcha las hebras
```

```
  barbero.thr.start();
  for (int i=0; i < 4; i++)
   clientes[i].thr.start();
 }
}
```

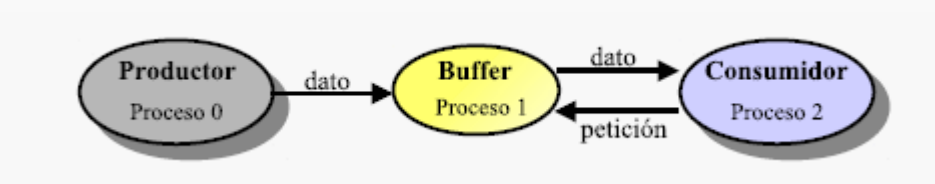
Ejemplo de ejecución:

```
Barbero se pone a dormir
Cliente empieza a afeitarse
Barbero coge otro cliente
Silla ocupada
Silla ocupada
Silla ocupada
Barbero termina de afeitar
Barbero coge otro cliente
Cliente empieza a afeitarse
Silla ocupada
Barbero termina de afeitar
Barbero coge otro cliente
Cliente empieza a afeitarse
Barbero termina de afeitar
```

PRÁCTICA 3

Problema del productor-consumidor selección no determinista

- Se debe permitir que el productor pueda enviar TAM datos sin tener que interrumpirse, y que el consumidor no se retrase cuando haya datos almacenados en el proceso buffer.
- Una forma de corregir dicho código consiste en usar una sentencia de selección no determinista de órdenes con guarda en el proceso Buffer que permita cierta asincronía entre productor y consumidor en función del tamaño del buffer temporal (TAM).
- En MPI, no hay ninguna sentencia de selección no determinista de órdenes con guarda, pero es fácil emularla con las funciones de sondeo MPI_Probe y/o MPI_Iprobe
- El proceso buffer da servicio a 5 productores y 4 consumidores. Para ello, se lanzarán 10 procesos y asumiremos que los procesos 0 . . . 4 son productores, el proceso Buffer es el proceso 5 y el resto de procesos en el comunicador universal (6 . . . 9) son consumidores.
- Esquema del funcionamiento:



El proceso Productor se encarga de ir generando enteros comenzando por el 0 y enviárselos al proceso Buffer. El proceso Consumidor envía peticiones al proceso Buffer, recibe los enteros de Buffer, los imprime por pantalla y calcula su raíz cuadrada.

El proceso Buffer debería atender las peticiones de ambos procesos (Productor y Consumidor).

```
/*Productor consumidor con selección no determinista*/

#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 TAM              5
#define PRODUCTORES      4
#define CONSUMIDORES     5

using namespace std;

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

void productor(int tag){
 for ( unsigned int i= 0 ; i < 80 ; i++ ){
  cout << "Productor" << tag << " 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)) );

  // enviar valor
  MPI_Ssend( &i, 1, MPI_INT, TAM, 0, MPI_COMM_WORLD );
 }
}
//
------------------------------------------------------------------------
-

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

 for (unsigned int i = 0 ; i < 80 ; i++ ){
        if(pos == 0)
                rama = 0;       //el consumidor no consume
        else{
                if (pos == TAM)
                {
                        rama = 1;       //el productor no puede
producir
```

```
                }
                else{
                        //Se puede consumir y producir

MPI_Probe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD, &status);
                        if(status.MPI_SOURCE <= 4)
                                rama = 0;
                        else
                                rama = 1;
                }
        }
        switch (rama){
                case 0:
                        MPI_Recv(&value[pos], 1,
MPI_INT,MPI_ANY_SOURCE, 0 , MPI_COMM_WORLD,&status);
                        cout << "Buffer recibe " << value[pos] << "
del Productor " << status.MPI_TAG << endl << flush;
                        pos++;
                        break;
                case 1:

MPI_Recv(&peticion,1,MPI_INT,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&status
);
                        MPI_Ssend(&value[pos-
1],1,MPI_INT,status.MPI_SOURCE,0,MPI_COMM_WORLD);
                        cout << "Buffer envía " << value[pos - 1] <<
" al Consumidor " << status.MPI_TAG << endl << flush;
      pos--;
      break;
        }
 }
}
//
-----------------------------------------------------------------------
-

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

 for (unsigned int i=0;i < 80;i++){
  MPI_Ssend(&peticion, 1, MPI_INT, TAM, 1, MPI_COMM_WORLD);
  MPI_Recv(&value, 1,     MPI_INT, TAM, 0, MPI_COMM_WORLD,&status );

  cout << "Consumidor" << tag <<" 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)) );
```

```
  // calcular raíz del valor recibido
  raiz = sqrt( value);
  cout << "Consumidor -> raiz cuadrada de " << value << " = " <<
raiz << endl << flush;
 }
}
//
--------------------------------------------------------------------
-

int main( int argc, char *argv[] ) {
 int rank ,  // identificador de proceso (comienza en 0)
     size ;  // numero de procesos

 // inicializar MPI y leer identificador 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) );
 if (size != 10) {
  cout << "El numero de procesos debe ser 10 " << endl;
  return 0;
 }
 // verificar el identificador de proceso (rank), y ejecutar la
 // operación apropiada a dicho identificador

 switch(rank){
        case 0:
                productor(0);
                break;
        case 1:
                productor(1);
                break;
        case 2:
                productor(2);
                break;
        case 3:
                productor(3);
                break;
        case 4:
                productor(4);
                break;
        case 5:
                buffer();
                break;
        case 6:
                consumidor(6);
                break;
```

```
        case 7:
                consumidor(7);
                break;
        case 8:
                consumidor(8);
                break;
        case 9:
                consumidor(9);
                break;
 }

 // al terminar el proceso, finalizar MPI
 MPI_Finalize( );

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

```
Productor0 produce valor 0
Productor1 produce valor 0
Productor2 produce valor 0
Productor3 produce valor 0
Productor4 produce valor 0
Productor1 produce valor 1
Buffer recibe 0 del Productor 0
Buffer envía 0 al Consumidor 1
Buffer recibe 0 del Productor 0
Consumidor6 recibe valor 0 de Buffer
Productor0 produce valor 1
Productor2 produce valor 1
Productor3 produce valor 1
Buffer envía 0 al Consumidor 1
Buffer recibe 0 del Productor 0
Buffer recibe 0 del Productor 0
Buffer envía 0 al Consumidor 1
Buffer envía 0 al Consumidor 1
```

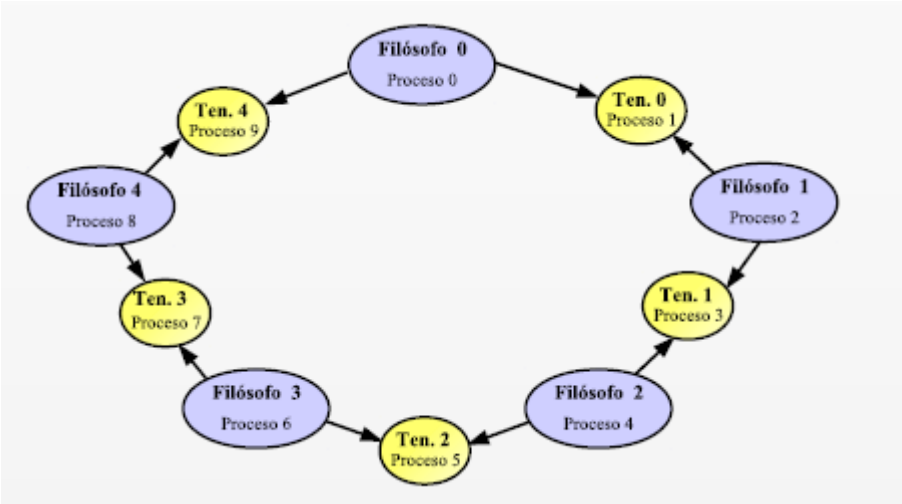
Modificaciones realizadas con respecto del programa de partida:

- En el main he añadido un switch para poder asignar productores, consumidor y buffer.
- He quitado #define BUFFER 5 y lo he sustituido por TAM 5. Este cambio lo he hecho para que se entienda mejor el código.
- Paso de parámetros a las funciones productor y consumidor para poder identificarlos.

Problema de los filósofos

Se pretende realizar una implementación del problema de la cena de los filósofos en MPI utilizando el siguiente esquema:

- Tenemos 5 procesos filósofos y 5 procesos tenedor (10 procesos en total). Supondremos que los procs. filósofos se identifican con número pares y los tenedores con números impares. El filósofo i (i = 0, ..., 4) será el proc. 2i y el tenedor i será el 2i + 1.



En principio, cada filósofo realiza repetidamente la siguiente secuencia de acciones:

- Pensar (sleep aleatorio).
- Tomar los tenedores (primero el tenedor izquierdo y después el derecho).
- Comer (sleep aleatorio).
- Soltar tenedores (en el mismo orden).

Las acciones pensar y comer pueden implementarse mediante un mensaje por pantalla seguido de un retardo durante un tiempo aleatorio. Las acciones de tomar tenedores y soltar tenedores pueden implementarse enviando mensajes de petición y de liberación a los procesos tenedor situados a ambos lados de cada filósofo.

Un tenedor solamente podrá ser asignado a uno de los dos filósofos que realicen la petición. Hasta que el tenedor no sea liberado no podrá ser asignado de nuevo. Cada proceso tenedor tendrá que ejecutar repetidamente la siguiente secuencia de acciones:

- Esperar mensajes de petición de tenedor y recibir uno.
- Esperar mensaje de liberación.

```
/*Filósofos*/

#include "mpi.h"
#include <iostream>
#include <time.h>
#include <stdlib.h>

using namespace std;

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

        while(1){
                if(id == 0){
                        // Solicitar tenedor derecho
            MPI_Ssend(&id, 1, MPI_INT, der, id, MPI_COMM_WORLD);
            cout << "El filosofo " << id << " coge el tenedor de su
derecha (" << der << ")." << endl << flush;

            // Solicitar tenedor izquierda
            MPI_Ssend(&id, 1, MPI_INT, izq, id, MPI_COMM_WORLD);
            cout << "El filosofo " << id << " coge el tenedor de su
izquierda (" << izq << ")." << endl << flush;
                }
                else {
      // Solicitar tenedor izquierda
      MPI_Ssend(&id, 1, MPI_INT, izq, id, MPI_COMM_WORLD);
      cout << "El filosofo " << id << " coge el tenedor de su
izquierda (" << izq << ")." << endl << flush;

      // Solicitar tenedor derecho
      MPI_Ssend(&id, 1, MPI_INT, der, id, MPI_COMM_WORLD);
      cout << "El filosofo " << id << " coge el tenedor de su
derecha (" << der << ")." << endl << flush;
        }

        cout << "El filosofo " << id << " esta COMIENDO..." << endl
<< flush;
        sleep((rand() % 3) + 1);

                // Suelta el tenedor izquierda
    MPI_Ssend(&id, 1, MPI_INT, izq, id, MPI_COMM_WORLD);
    cout << "El filosofo " << id << " suelta el tenedor de su
izquierda " << izq << "." << endl << flush;

    // Suelta el tenedor derecha
    MPI_Ssend(&id, 1, MPI_INT, der, id, MPI_COMM_WORLD);
    cout << "El filosofo " << id << " suelta el tenedor de su
derecha " << der << "." << endl << flush;
```

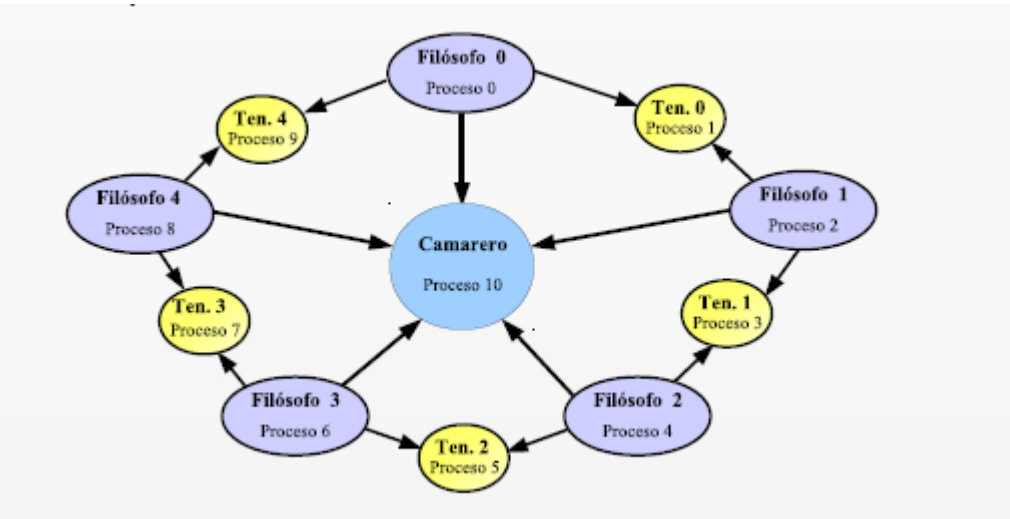
```
    cout << "El filosofo " << id << " esta PENSANDO..." << endl <<
flush;
    sleep((rand() % 3) + 1);
        }
}

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

  while (1) {
    // Espera una peticion desde cualquier filosofo vecino...
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag,
&status);

    // Recibe la peticion de filosofo...
    MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
    Filo = status.MPI_SOURCE;
    cout << "El tenedor " << id << " recibe una peticion del
filosofo " << Filo << "." << endl << flush;

    // Espera a que el filosofo suelte el tenedor...
    MPI_Recv(&buf, 1, MPI_INT, Filo, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
    cout << "El tenedor " << id << " recibe la liberacion del
filosofo " << Filo << "." << endl << flush;
  }
}

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;
}
```

```
El filosofo 0 coge el tenedor de su derecha (1).
El tenedor 1 recibe una peticion del filosofo 0.
El tenedor 3 recibe una peticion del filosofo 4.
El tenedor 7 recibe una peticion del filosofo 6.
El filosofo 0 coge el tenedor de su izquierda (9).
El filosofo 0 esta COMIENDO...
El filosofo 4 coge el tenedor de su izquierda (3).
El tenedor 5 recibe una peticion del filosofo 6.
El filosofo 6 coge el tenedor de su izquierda (5).
El filosofo 6 coge el tenedor de su derecha (7).
El filosofo 6 esta COMIENDO...
El tenedor 9 recibe una peticion del filosofo 0.
El tenedor 5 recibe la liberacion del filosofo 6.
El tenedor 5 recibe una peticion del filosofo 4.
El filosofo 4 coge el tenedor de su derecha (5).
El filosofo 4 esta COMIENDO...
```

Problema de los filósofos con camarero

Una forma de evitar la posibilidad de interbloqueo consiste en impedir que todos los filósofos intenten ejecutar la acción de "tomar tenedor" al mismo tiempo. Para ello podemos usar un proceso camarero central que permita sentarse a la mesa como máximo a 4 filósofos. Podemos suponer que tenemos 11 procesos y que el camarero es el proc. 10.



Ahora, cada filósofo ejecutará repetidamente la siguiente secuencia de acciones:

- Pensar
- Sentarse
- Tomar tenedores
- Comer
- Soltar tenedores
- Levantarse

Cada filósofo pedirá permiso para sentarse o levantarse enviando un mensaje al camarero, el cual llevará una cuenta del número de filósofos que hay sentados a la mesa en cada momento.

El camarero acepta las peticiones de levantarse sin problema, sin embargo solo acepta peticiones de sentarse si hay sitio en la mesa, para hacer esto hay que tener en cuenta quien realiza la petición, uso de etiquetas para implementar esperas selectivas.

```
/*Filósofos*/

#include "mpi.h"
#include <iostream>
#include <time.h>
#include <stdlib.h>
```

```
#include <unistd.h>

using namespace std;

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

        while(1){
                cout << "El filosofo " << id << " esta PENSANDO..."
<< endl << flush;
    sleep((rand() % 3) + 1);
    // Solicitar poder sentarse a la mesa
    MPI_Ssend(&id, 1, MPI_INT, 10, id, MPI_COMM_WORLD);
    cout << "El filosofo " << id << " se sienta a la mesa." << endl
<< flush;

    if(id == 0){

      // Solicitar tenedor derecho
            MPI_Ssend(&id, 1, MPI_INT, der, id, MPI_COMM_WORLD);
            cout << "El filosofo " << id << " coge el tenedor de su
derecha (" << der << ")." << endl << flush;

            // Solicitar tenedor izquierda
            MPI_Ssend(&id, 1, MPI_INT, izq, id, MPI_COMM_WORLD);
            cout << "El filosofo " << id << " coge el tenedor de su
izquierda (" << izq << ")." << endl << flush;
                }
                else {
      // Solicitar tenedor izquierda
      MPI_Ssend(&id, 1, MPI_INT, izq, id, MPI_COMM_WORLD);
      cout << "El filosofo " << id << " coge el tenedor de su
izquierda (" << izq << ")." << endl << flush;

      // Solicitar tenedor derecho
      MPI_Ssend(&id, 1, MPI_INT, der, id, MPI_COMM_WORLD);
      cout << "El filosofo " << id << " coge el tenedor de su
derecha (" << der << ")." << endl << flush;
        }

        cout << "El filosofo " << id << " esta COMIENDO..." << endl
<< flush;
        sleep((rand() % 3) + 1);

                // Suelta el tenedor izquierda
    MPI_Ssend(&id, 1, MPI_INT, izq, id, MPI_COMM_WORLD);
    cout << "El filosofo " << id << " suelta el tenedor de su
izquierda " << izq << "." << endl << flush;

    // Suelta el tenedor derecha
    MPI_Ssend(&id, 1, MPI_INT, der, id, MPI_COMM_WORLD);
```

```
    cout << "El filosofo " << id << " suelta el tenedor de su
derecha " << der << "." << endl << flush;

    // Solicitar poder levantarse de la mesa
    cout << "El filosofo " << id << " se levanta de la mesa." <<
endl << flush;
    MPI_Ssend(&id, 1, MPI_INT, 10, id, MPI_COMM_WORLD);
        }
}

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

  while (1) {
    // Espera una peticion desde cualquier filosofo vecino...
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag,
&status);

    // Recibe la peticion de filosofo...
    MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
    Filo = status.MPI_SOURCE;
    cout << "El tenedor " << id << " recibe una peticion del
filosofo " << Filo << "." << endl << flush;

    // Espera a que el filosofo suelte el tenedor...
    MPI_Recv(&buf, 1, MPI_INT, Filo, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
    cout << "El tenedor " << id << " recibe la liberacion del
filosofo " << Filo << "." << endl << flush;
  }
}

void Camarero(int id, int nprocesos){
  MPI_Status status;
  int max = 4, num = 0, buf, Filo, flag;

  while (1) {
    // Espera una peticion desde cualquier filosofo para sentarse o
levantarse...
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag,
&status);

    if (num < max) {
      // Recibe la peticion de filosofo...
      MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
      Filo = status.MPI_SOURCE;
```

```
      cout << "El camarero recibe una peticion del filosofo " << Filo << "." << endl << flush;
      num++;

      // Espera a que el filosofo se levante...
      MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      cout << "El camarero le da permiso al filosofo " << Filo << " para levantarse de la mesa." << endl << flush;
      num--;
    }
    else {
      Filo = status.MPI_SOURCE;
      cout << "El filosofo " << Filo << " no puede sentarse a la mesa porque esta llena." << endl << flush;
    }
  }
}

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 % 2) == 0)
    if(rank == 10)
      Camarero(rank,size);  //El proceso camarero es el 10

    Filosofo(rank, size); // Los pares son Filosofos
  else
    Tenedor(rank, size); // Los impares son Tenedores

  MPI_Finalize();

  return 0;
}
```

```
El filosofo 0 esta PENSANDO...
El filosofo 2 esta PENSANDO...
El filosofo 4 esta PENSANDO...
El filosofo 6 esta PENSANDO...
El filosofo 8 esta PENSANDO...
El filosofo 0 se sienta a la mesa.
El filosofo 0 coge el tenedor de su derecha (1).
El filosofo 0 coge el tenedor de su izquierda (10).
El filosofo 0 esta COMIENDO...
El tenedor 1 recibe una peticion del filosofo 0.
El camarero recibe una peticion del filosofo 0.
El camarero le da permiso al filosofo 0 para levantarse de la mesa.
El camarero recibe una peticion del filosofo 2.
El filosofo 2 se sienta a la mesa.
El camarero le da permiso al filosofo 2 para levantarse de la mesa.
El tenedor 3 recibe una peticion del filosofo 4.
El tenedor 5 recibe una peticion del filosofo 4.
El filosofo 4 se sienta a la mesa.
El filosofo 4 coge el tenedor de su izquierda (3).
El filosofo 4 coge el tenedor de su derecha (5).
El filosofo 4 esta COMIENDO...
El camarero recibe una peticion del filosofo 6.
El camarero le da permiso al filosofo 6 para levantarse de la mesa.
El tenedor 7 recibe una peticion del filosofo 8.
El filosofo 6 se sienta a la mesa.
El tenedor 9 recibe una peticion del filosofo 8.
El filosofo 8 se sienta a la mesa.
El filosofo 8 coge el tenedor de su izquierda (7).
El filosofo 8 coge el tenedor de su derecha (9).
El filosofo 8 esta COMIENDO...
```