SINCRONIZACIÓN DE PROCESOS

Objetivo.

• El Alumno realizará implementaciones de la sincronización de procesos a través de la solución de los problemas clásicos de comunicación entre procesos.

Introducción

En ocasiones los procesos requieren compartir información para realizar alguna tarea

❑ Procesos disjuntos.- No se comunican y cada uno realiza su ejecución de manera independiente

❑ Procesos cooperativos.- Intercambian información para resolver un problema

Las formas de comunicación que se establecen para la comunicación son:

❑ Memoria compartida.- Se establece una región de la memoria para que sea compartida por los procesos cooperativos; pueden intercambiar información leyendo y escribiendo en las mismas localidades

❑ Paso de mensajes.- La comunicación tiene lugar mediante el intercambio directo de datos entre los procesos cooperativos

Cuando varios procesos deben manejar los mismos datos concurrentemente y el resultado de la ejecución depende del orden concreto en que se produzcan los accesos, se conoce como condición de carrera.

En tales situaciones se necesita garantizar que solo un proceso pueda acceder a esas variables o datos. Esto se realiza mediante la sincronización

La sección crítica de un proceso es un segmento de código en el que se van a encontrar variables que pueden ser accedidas por otros procesos.

Cuando un proceso se encuentra dentro de su sección crítica, ningún otro proceso puede ejecutar su respectiva región crítica.

Desarrollo.

Nota: Todos los ejercicios de esta práctica se deberán realizar en el sistema operativo Linux

PARTE 1.- Problemas a resolver

1- El alumno deberá realizar la implementación de los siguientes problemas de comunicación

• Cena de los filósofos

• Barbero dormilón

• Problema de los lectores y escritores

Se deberá implementar al menos una solución para cada uno de los problemas mencionados junto con la descripción detallada de los elementos del problema.

Se deberán utilizar las herramientas vistas en clase (Semáforos, Mutex, Monitores – Pthreads, hilos en java, Monitores en java)

Al menos una de las 3 en Pthreads y/o semáforos en C (<semaphore.h>)

El de los lectores y escritores (Courtois et al., 1971), es un problema que modela el acceso a una base de datos. Imaginemos, por ejemplo, un sistema de reservaciones de una línea aérea, con muchos procesos competidores que desean leerlo y escribir en él. Es aceptable tener múltiples procesos leyendo la base de datos al mismo tiempo, pero si un proceso está actualizando (escribiendo en) la base de datos, ningún otro podrá tener acceso a ella, ni siquiera los lectores

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_HILOS 10  //Cantidad de hilos que se van a usar

//Se declara un mutex y una variable de condicion, ambos se inicializan
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

//Se declaran dos contadores, uno para lectores y otro general, inicializando en 0
int contador = 0;
int lectores = 0;

//Esta es la función que se ejecutará en los hilos lectores
void *lector(void *arg){
    //Adquiere el mutex
    pthread_mutex_lock(&mutex);
    //Incrementa en 1 el contador de lectores
    lectores++;
    //Libera al mutex usado anteriormente
    pthread_mutex_unlock(&mutex);

    //Imprime el mensaje
    printf("Leyendo\n");
    //sleep(1);

    //Adquiere el mutex
    pthread_mutex_lock(&mutex);
    //Decrementa en 1 el contador de lectores
    lectores--;

    //Señala cualquier hilo en espera en la variable de condición
    pthread_cond_signal(&cond);
    //Libera el mutex
    pthread_mutex_unlock(&mutex);
```

Line 1, Column 19

```
}

//Esta es la función que se ejecutará en los hilos escirtores
void *escritor(void *arg){
    //Adquiere el mutex
    pthread_mutex_lock(&mutex);

    //El while se encargará de que mientras haya lectores, esta se espera en la variable de condición
    while (lectores > 0){
    pthread_cond_wait(&cond, &mutex);
    }
    printf("Escribiendo\n");

//  sleep(1);
    //Libera el mutex
    pthread_mutex_unlock(&mutex);
}

int main() {
    //Se declara una matriz de hilos
    pthread_t hilos[NUM_HILOS];

    //Crea y ejecuta cada hilo
    for (int i = 0; i < NUM_HILOS; i++){
        if (i % 2 == 0){
            //Si i es par se va a crear un hilo lector
            pthread_create(&hilos[i], NULL, lector, NULL);
        }else{
            //Si i es par se va a crear un hilo escritor
            pthread_create(&hilos[i], NULL, escritor, NULL);
        }
    }
    //Espera a que todos los hilos terminen
    for (int i = 0; i < NUM_HILOS; i++){
        //Funcion que espera a que un hilo en especifico termine
        pthread_join(hilos[i], NULL);
    }

    return 0;
}
```

En primer lugar, el código incluye las cabeceras necesarias y define una constante NUM_HILOS para el número de hilos que se crearán.

Luego, declara un mutex llamado mutex y una variable de condición cond. Estos dos son inicializados con sus respectivas macros de inicialización. También declara dos variables globales: contador, que no se utiliza en el código, y lectores, que se utiliza para llevar un registro del número de hilos lectores que actualmente están accediendo al recurso compartido.

A continuación, el código define una función lector que se utilizará como punto de entrada para los hilos lectores. Esta función toma un puntero void como argumento y devuelve un puntero void. Dentro de la función, se bloquea el mutex para asegurar que sólo un hilo pueda acceder al recurso compartido al mismo tiempo. Luego, se incrementa la variable lectores para indicar que hay un hilo lector más accediendo al recurso compartido. Se desbloquea entonces el mutex para permitir que otros hilos

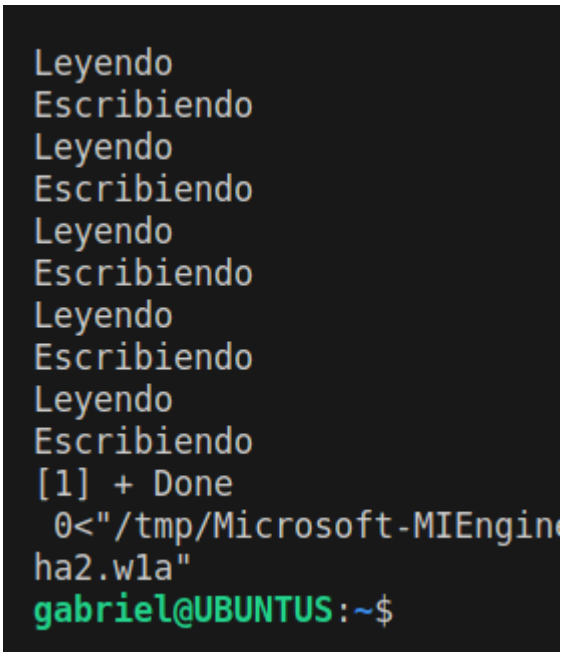
accedan al recurso compartido. El hilo lector luego imprime un mensaje en la consola y duerme durante 1 segundo. Esto simula el proceso de lectura del recurso compartido.

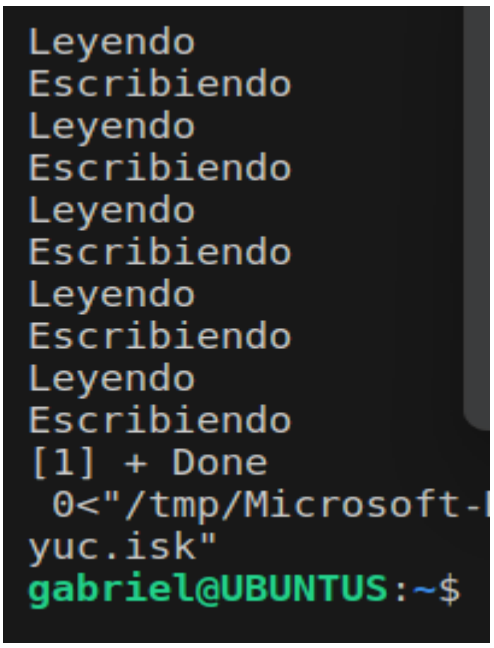
Después de leer, se vuelve a bloquear el mutex y se decrementa la variable lectores para indicar que el hilo lector ya no está accediendo al recurso compartido. Si ya no hay más lectores accediendo al recurso compartido, se envía una señal a la variable de condición para despertar a cualquier hilo escritor en espera. Finalmente, se desbloquea el mutex para permitir que otros hilos accedan al recurso compartido.

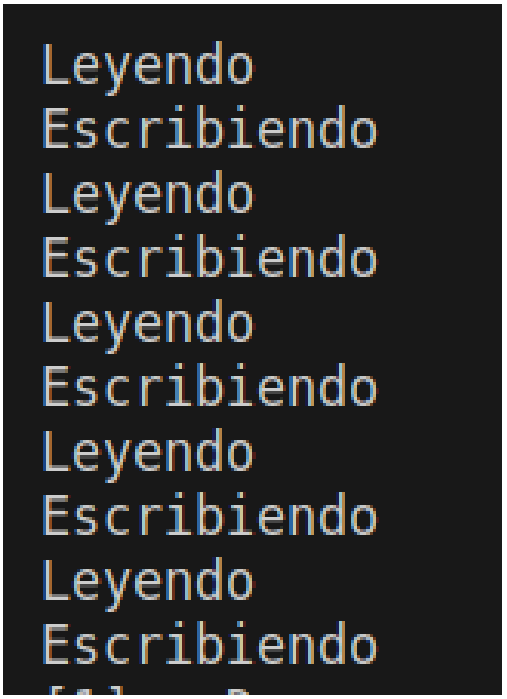
El código también define una función escritor que se utilizará como punto de entrada para los hilos escritores. Esta función es similar a la función lector, con la principal diferencia de que el hilo escritor espera en la variable de condición si hay hilos lectores accediendo al recurso compartido.

El hilo escritor espera en la variable de condición llamando a pthread_cond_wait(&cond, &mutex), lo que desbloqueará atómicamente el mutex y bloqueará al hilo. El hilo se desbloqueará y el mutex se volverá a bloquear cuando se envíe una señal a la variable de condición (ya sea por otro hilo que llame a pthread_cond_signal o pthread_cond_broadcast).

Finalmente, la función principal crea un array de hilos y crea un hilo lector para cada elemento con índice par y un hilo escritor para cada elemento con índice impar. Luego espera a que todos los hilos terminen utilizando pthread_join.




Leyendo
Escribiendo
Leyendo
Escribiendo
Leyendo
Escribiendo
Leyendo
Escribiendo
Leyendo
Escribiendo
[1] + Done
0<"/tmp/Microsoft-
yuc.isk"
gabriel@UBUNTUS:~$


Leyendo
Escribiendo
Leyendo
Escribiendo
Leyendo
Escribiendo
Leyendo
Escribiendo
Leyendo
Escribiendo

Al menos una de las 3 utilizando Semáforos en Java

Barbero dormilón

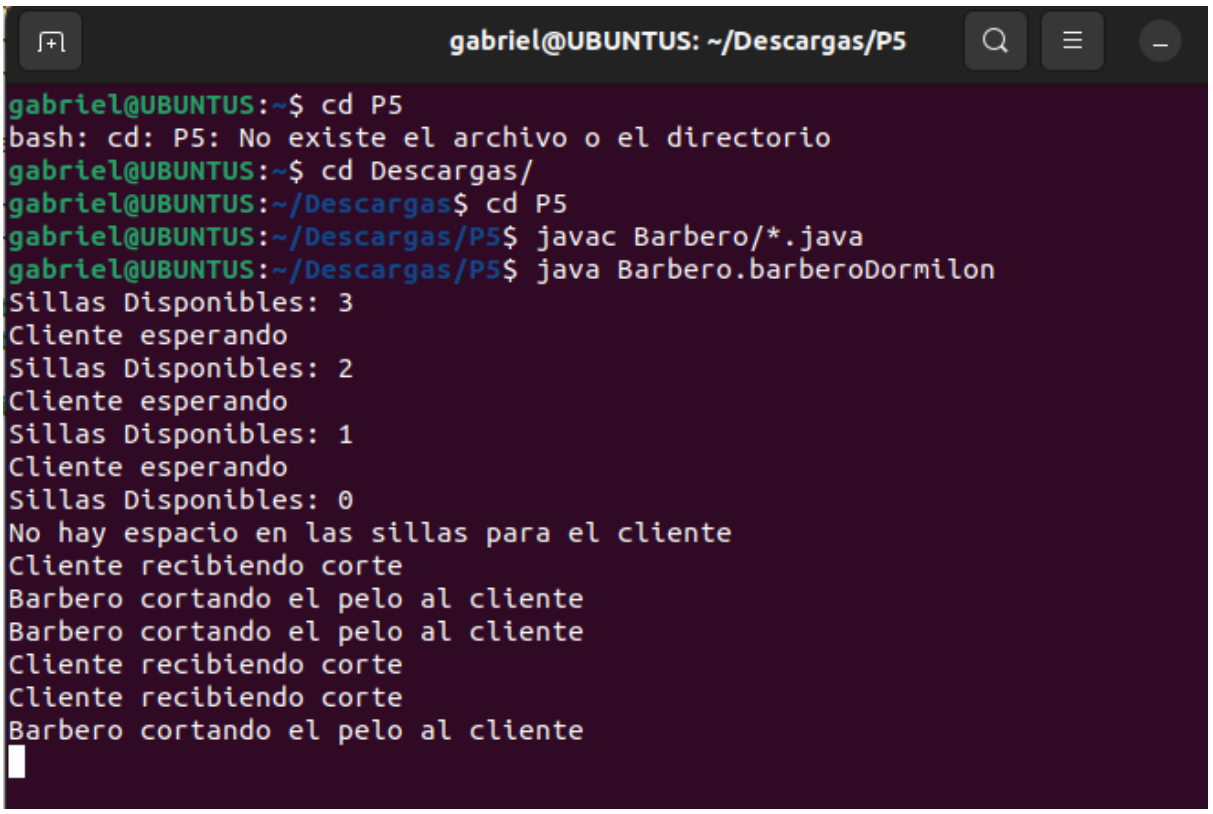
Barberos.java

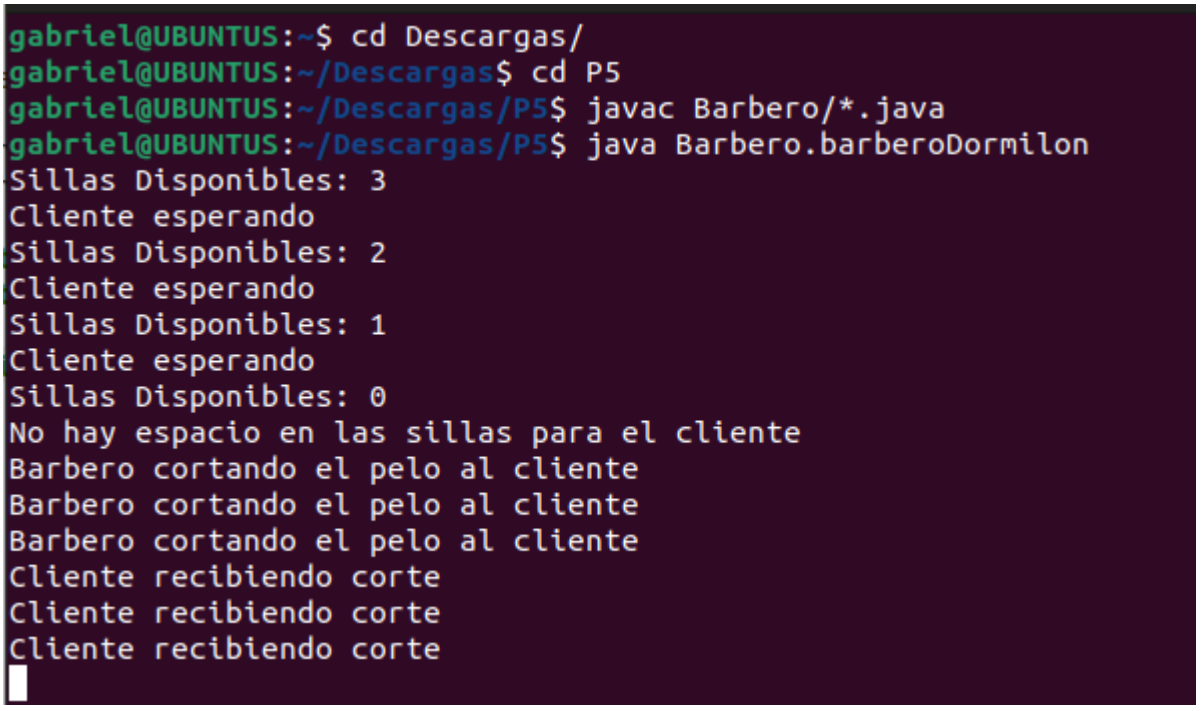
```java
package Barberos;
import java.util.concurrent.Semaphore; //Libreria para el uso de semaforos

public class barberoDormilon{
    static Semaphore barbero = new Semaphore(0); //Declaracion del semaforo barbero
    static Semaphore cliente = new Semaphore(0); //Declaracion del semaforo cliente
    static Semaphore mutex = new Semaphore(1);   //Declaracion del semaforo mutex

    //Declaración de variables enteras
    static int sillaEspera = 3; //Sillas en espera
    static int sillasDis = 3;  //Sillas disponibles
    static int clientesEsperando = 0; //Clientes en espera

    //Se define la función main(), y aqui es donde se van a crear e iniciar los hilos de los barberos y clientes

    public static void main(String[] args) throws InterruptedException{

        Thread barberoThread1 = new Thread(new Barbero());
        //Thread barberoThread2 = new Thread(new Barbero()); → se pueden tener mpas barberos
        Thread clienteThread1 = new Thread(new Cliente());
        Thread clienteThread2 = new Thread(new Cliente());
        Thread clienteThread3 = new Thread(new Cliente());
        Thread clienteThread4 = new Thread(new Cliente());

        barberoThread1.start();

        clienteThread1.start();
        clienteThread2.start();
        clienteThread3.start();
        clienteThread4.start();
    }

    // Se define una clase Cliente para implementar la lógica de los cliente que llegan
    static class Cliente implements Runnable{
        @Override
    public void run(){
        try{
            /*acquire bloquea la ejecución del hilo en curso y queda a la espera de
            que otro hilo llame a release() en este caso se encuentran más abajode código esperando
            cliente y mutrx para que despues el barbero bloque nuevamente*/
```

Line 61, Column 114

Barberos.java

```
40              cliente y mutrx para que despues el barbero bloque nuevamente*/
41
42              mutex.acquire();
43              System.out.println("Sillas Disponibles: "+sillasDis);
44              sillasDis--;
45              if (clientesEsperando < sillaEspera) {
46                  clientesEsperando++;
47                  System.out.println("Cliente esperando"); //Mensaje que indica que un cliente esta esperando su turno
48                  cliente.release();
49                  mutex.release();
50                  barbero.acquire();
51                  cortarPelo();
52              }else{
53                  mutex.release();
54                  System.out.println("No hay espacio en las sillas para el cliente");
55              }
56          }catch (InterruptedException e) {
57              e.printStackTrace();
58          }
59      }
60      private void cortarPelo() throws InterruptedException{
61          System.out.println("Cliente recibiendo corte"); //Mensaje de que un cliente esta en ejecución de su corte
62          }
63      }
64
65      //Se define la clase Barbero que implementa la logica del barbero y que este se quede dormido hasta que llegue un nuevo cliente
66      static class Barbero implements Runnable{
67          @Override
68          public void run(){
69              while(true){
70                  try{
71                      cliente.acquire();
72                      mutex.acquire();
73                      clientesEsperando--;
74                      barbero.release();
75                      mutex.release();
76                      cortarPelo();
77                  } catch (InterruptedException e){
78                  }
79              }
80      }
81
82      private void cortarPelo() throws InterruptedException{
83          System.out.println("Barbero cortando el pelo al cliente");
84          }
85      }
86  }
```


gabriel@UBUNTUS: ~/Descargas/P5
gabriel@UBUNTUS:~$ cd P5
bash: cd: P5: No existe el archivo o el directorio
gabriel@UBUNTUS:~$ cd Descargas/
gabriel@UBUNTUS:~/Descargas$ cd P5
gabriel@UBUNTUS:~/Descargas/P5$ javac Barbero/*.java
gabriel@UBUNTUS:~/Descargas/P5$ java Barbero.barberoDormilon
Sillas Disponibles: 3
Cliente esperando
Sillas Disponibles: 2
Cliente esperando
Sillas Disponibles: 1
Cliente esperando
Sillas Disponibles: 0
No hay espacio en las sillas para el cliente
Cliente recibiendo corte
Barbero cortando el pelo al cliente
Barbero cortando el pelo al cliente
Cliente recibiendo corte
Cliente recibiendo corte
Barbero cortando el pelo al cliente


gabriel@UBUNTUS:~$ cd Descargas/
gabriel@UBUNTUS:~/Descargas$ cd P5
gabriel@UBUNTUS:~/Descargas/P5$ javac Barbero/*.java
gabriel@UBUNTUS:~/Descargas/P5$ java Barbero.barberoDormilon
Sillas Disponibles: 3
Cliente esperando
Sillas Disponibles: 2
Cliente esperando
Sillas Disponibles: 1
Cliente esperando
Sillas Disponibles: 0
No hay espacio en las sillas para el cliente
Barbero cortando el pelo al cliente
Barbero cortando el pelo al cliente
Barbero cortando el pelo al cliente
Cliente recibiendo corte
Cliente recibiendo corte
Cliente recibiendo corte

```
gabriel@UBUNTUS:~$ javac Barbero/*.java
error: file not found: Barbero/*.java
Usage: javac <options> <source files>
use --help for a list of possible options
gabriel@UBUNTUS:~$ cd Descargas/
gabriel@UBUNTUS:~/Descargas$ cd P5
gabriel@UBUNTUS:~/Descargas/P5$ javac Barbero/*.java
gabriel@UBUNTUS:~/Descargas/P5$ java Barbero.barberoDormilon
Sillas Disponibles: 3
Cliente esperando
Sillas Disponibles: 2
Cliente esperando
Sillas Disponibles: 1
Cliente esperando
Sillas Disponibles: 0
No hay espacio en las sillas para el cliente
Cliente recibiendo corte
Barbero cortando el pelo al cliente
Barbero cortando el pelo al cliente
Barbero cortando el pelo al cliente
Cliente recibiendo corte
Cliente recibiendo corte
```

El problema consiste en que esta peluquería tiene un peluquero, una silla de peluquero y “n” sillas donde pueden sentarse los clientes que esperan, si los hay. Si no hay clientes presentes, el peluquero se sienta en la silla del peluquero y se duerme. Cuando llega un cliente, tiene que despertar al peluquero dormido. Si llegan clientes adicionales mientras el peluquero está cortándole el pelo a un cliente, se sientan (si hay sillas vacías) o bien salen del establecimiento (si todas las sillas están ocupadas). El problema consiste en programar al peluquero y sus clientes sin entrar en condiciones de competencia.

La solución ocupada es con los objetos Semaphore en el programa se utilizan para sincronizar las acciones del barbero y los clientes. El semáforo barbero se utiliza para bloquear el hilo del barbero cuando no hay clientes esperando. El semáforo cliente se utiliza para bloquear los hilos de cliente cuando no hay sillas disponibles en el área de espera o cuando el barbero está ocupado. El semáforo mutex se utiliza para proteger el acceso a variables compartidas como sillasDis y clientesEsperando. La clase Prueba contiene el método principal, así como dos clases internas: Cliente y Barbero. La clase Cliente representa un hilo de cliente y la clase Barbero representa un hilo de barbero. El método main crea varios objetos Thread, cada uno ejecutando una instancia de la clase Cliente o Barbero. También crea tres objetos Semaphore: barbero, cliente y mutex. El semáforo barbero se inicializa con un valor de 0. Esto significa que el semáforo comienza en un estado "bloqueado", y cualquier hilo que intente adquirirlo se bloqueará hasta que el semáforo sea liberado. El semáforo cliente también se inicializa con un valor de 0. Esto significa que el semáforo también comienza en un estado "bloqueado", y cualquier hilo que intente adquirirlo se bloqueará hasta que el semáforo sea liberado. El semáforo mutex se inicializa con un valor de 1. Esto significa que el semáforo comienza en un estado "desbloqueado", y cualquier hilo que intente adquirirlo podrá hacerlo de inmediato. El método main también declara tres variables: sillasEspera, sillasDis y clientesEsperando. sillasEspera representa el número de sillas en el área de espera, sillasDis representa el número de sillas disponibles en el área de espera y clientesEsperando representa el número de clientes esperando su turno. Ahora veamos la clase Cliente. Cuando se inicia un hilo de cliente, se ejecuta el método run, que contiene el siguiente código:

```
 try {
mutex.acquire();
System.out.println("Sillas disponibles: " +sillasDis);
 sillasDis--;
if (clientesEsperando < sillasEspera) {
clientesEsperando++;
System.out.println("Cliente " +clientesEsperando+ " esperando");
Thread.sleep(2000);
cliente.release();
mutex.release();
```

```
barbero.acquire();
cortarPelo();
System.out.println("Corte hecho al cliente "+clientesEsperando);
} else {
mutex.release();
 System.out.println("No hay espacio en las sillas para el cliente");
} } catch (InterruptedException e) { e.printStackTrace();
}
```

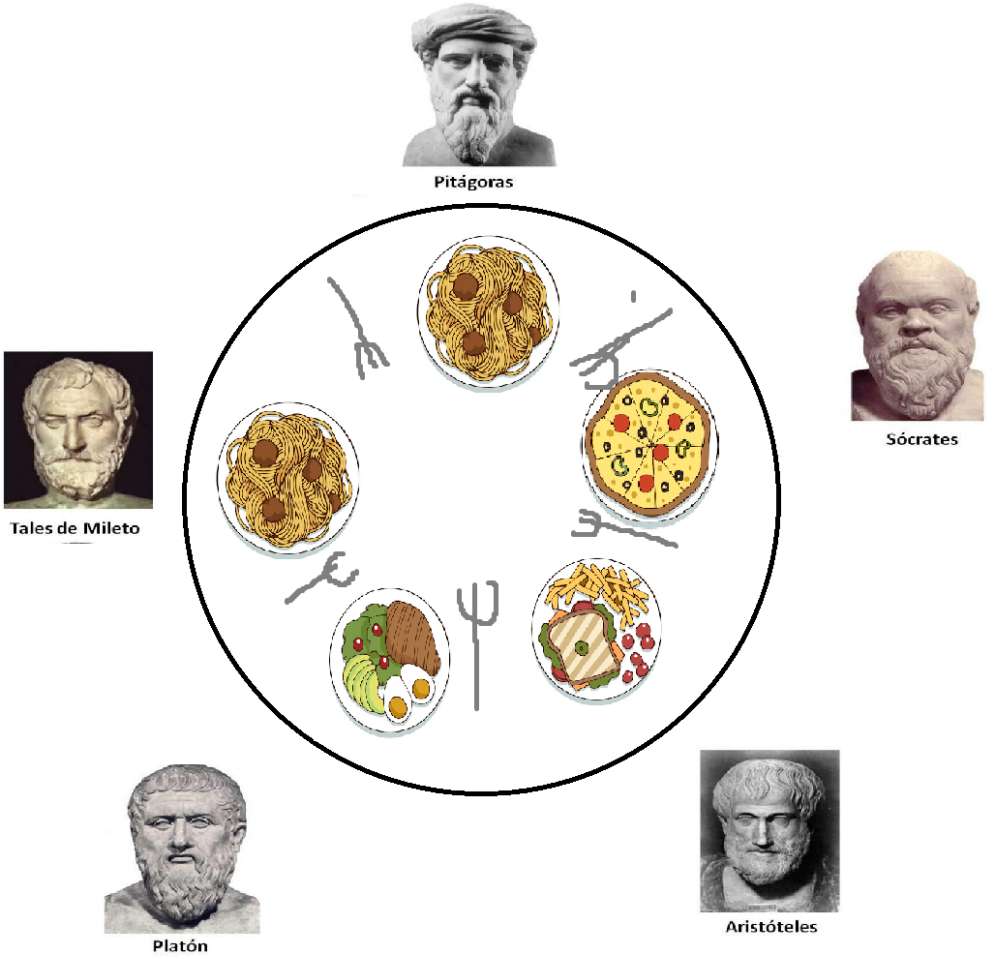
Lo primero que hace el cliente es intentar adquirir el semáforo mutex. Si el semáforo está disponible, significa que ningún otro cliente está accediendo actualmente a las variables compartidas, por lo que el cliente puede disminuir de forma segura el número de sillas disponibles y comprobar si hay sillas disponibles en la sala de espera. Si hay sillas disponibles, el cliente aumenta el número de clientes esperando y libera el semáforo cliente para señalar al barbero que hay un cliente esperando. Luego libera el semáforo mutex y espera a que el barbero termine de cortarle el cabello adquiriendo el semáforo barbero. Cuando el barbero libera el semáforo barbero, significa que el cabello del cliente ha sido cortado y el cliente puede adquirir el semáforo mutex de nuevo para actualizar las variables compartidas. El cliente disminuye el número de clientes esperando y aumenta el número de sillas disponibles. Finalmente, el cliente libera el semáforo mutex para permitir que otros clientes accedan a las variables compartidas

Al menos una de las 3 utilizando Monitores en Java (bloque synchronized)

El problema tiene un planteamiento muy sencillo. Cinco filósofos están sentados alrededor de una mesa circular. Cada filósofo tiene ante sí un plato de espagueti. El espagueti es tan resbaloso que un filósofo necesita dos tenedores para comerlo. Entre cada par de platos hay un tenedor.

La vida de un filósofo consiste en periodos alternantes de comer y pensar. (Esto es una abstracción, incluso en el caso de un filósofo, pero las demás actividades no son pertinentes aquí.) Cuando un filósofo siente hambre, trata de adquirir sus tenedores izquierdo y derecho, uno a la vez, en cualquier orden. Si logra adquirir

dos tenedores, comerá durante un rato, luego pondrá los tenedores en la mesa y seguirá pensando.



Clase Principal

```java
package FIlosofosPensadores;

public class Principal {

    public static void main(String[] args) {
        Monitor m = new Monitor(5); //creamos la mesa con sus 5 filosofos
        for (int i = 1; i <= 5; i++) {
            Filosofo f = new Filosofo(m, i);
            f.start();
        }
    }
}
```

- En esta clase solo se definirá el tamaño del monitor, es decir cuantos filósofos participarán.

Clase Monitor

Abrir | Monitor.java ~/Documentos/FIlosofosPensadores | Guardar

```java
package FIlosofosPensadores;

import java.util.logging.Level;
import java.util.logging.Logger;

public class Monitor { // Es el monitor del programa y sera el encargado de tomar o liberar los tenedores

    private boolean[] tenedores; //Creamos un arreglo donde guardaremos los tenedores en V o F

    public Monitor(int numTenedores){ // Le pasamos el numero de tenedores declarados en principal
        this.tenedores = new boolean[numTenedores];
    }

    public int tenedorIzquierda(int i){ // Si sera el filosofo, si toma el tenedor izquierdo sera igual a su indice
        return i;
    }

    public int tenedorDerecha(int i){ // Si toma derecha se decrementara
        if(i == 0){
            return this.tenedores.length - 1; //Esto es una cola circular el que primero filosofo tiene que tomar el tenedor de su izquierda osea el ultimo
```

- La clase monitor, contiene los tenedores que debemos manejar definido en un array de booleanos y esta con base a las funciones synchronized se encargará de tomar o liberar los tenedores que los filósofos requieran.

```java

    public synchronized void tomarTenedores(int nfilosofo){

        while(tenedores[tenedorIzquierda(nfilosofo)] || tenedores[tenedorDerecha(nfilosofo)]){
            try {
            System.out.println("Filosofo " + (nfilosofo+1) + " quiere comer pero no puede porque estan ocupando uno de sus tenedores");
                wait(); //Esto se encargara de comprobar si los tenemos <- o -> estan ocupados, si lo estan se bloquea con la funcion wait


            } catch (InterruptedException ex) {
                Logger.getLogger(Mesa.class.getName()).log(Level.SEVERE, null, ex);
            }
        }

        tenedores[tenedorIzquierda(nfilosofo)] = true; // si no estan ocupados los asignamos con un true diciendo que estan ocupados
        tenedores[tenedorDerecha(nfilosofo)] = true;
    }
```

- Cuando un filósofo quiera coger el tenedor de la izquierda y de la derecha, tendremos que decirle cual es la posición concretamente

```
42
43      public synchronized void dejarTenedores(int nfilosofo){ // cuando termine
    de usarlos los liberamos con un false indicando que estan disponibles
44          tenedores[tenedorIzquierda(nfilosofo)] = false;
45          tenedores[tenedorDerecha(nfilosofo)] = false;
46          notifyAll(); //Hace una llamada a los procesos que inicien la funcion
    de comprobacion
47      }
48
49  }
```

- A la hora de tomar los tenedores, se comprueba si el tenedor de la izquierda o derecha están ocupados y entrará en bloqueo hasta que se liberen, de lo contrario si no están ocupados, se le asigna True indicando que los ocupará.

- Al liberar los tenedores, el proceso asigna false a ambos tenedores que ocupa el filósofo y notificaremos a todos aquellos procesos que estuvieran parados en el anterior método para que vuelvan a comprobar los tenedores.

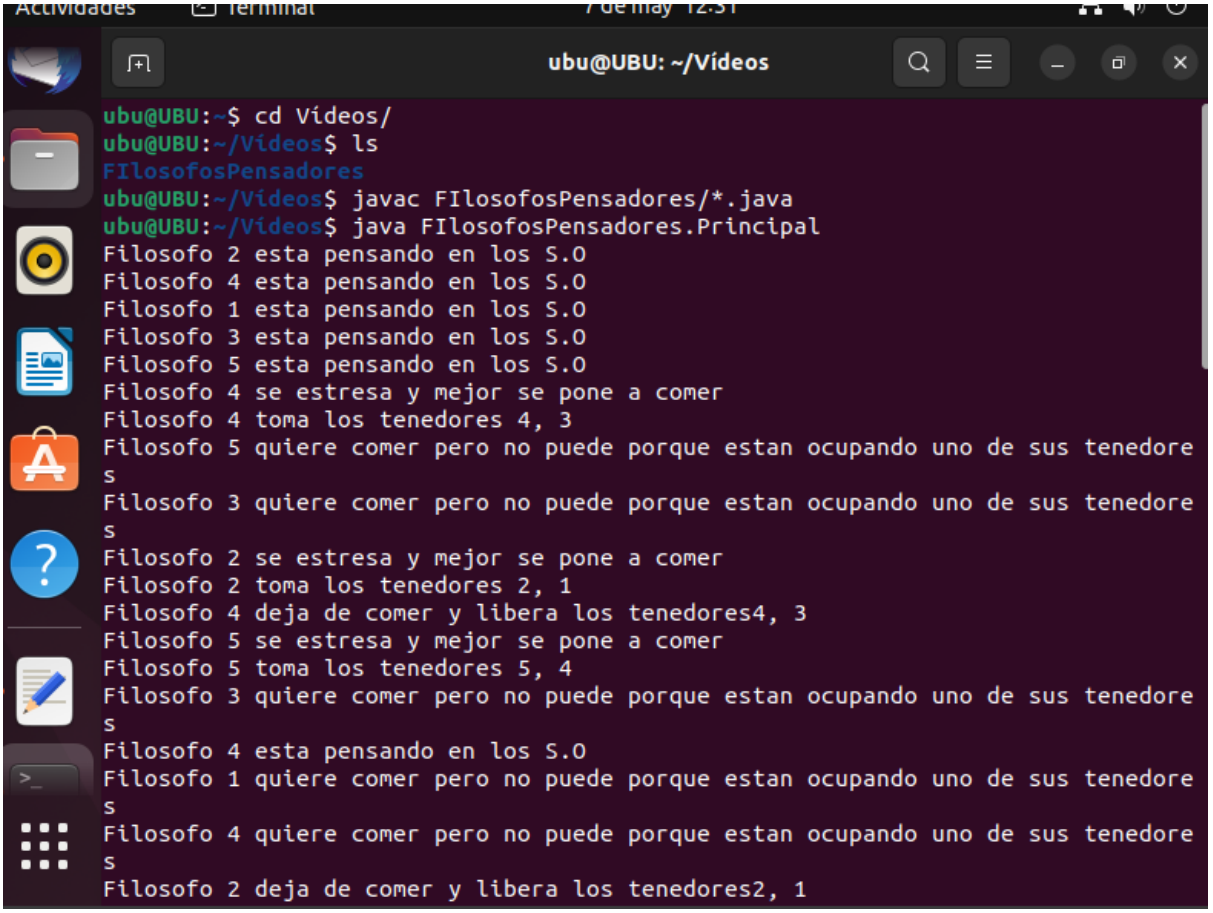
Clase Filósofo

```
1  package FIlosofosPensadores;
2
3  import java.util.logging.Level;
4  import java.util.logging.Logger;
5
6  public class Filosofo extends Thread {
7
8      private Monitor monitor; //Gestiona el uso de tenedores
9      private int nfilosofo; //numero de filosofo
10     private int indiceTenedor;  //numero de tenedor
11
12     public Filosofo(Monitor m, int nfilosofo){
13         this.monitor = m;
14         this.nfilosofo = nfilosofo;
15        this.indiceTenedor = nfilosofo-1;
16
17     }
```

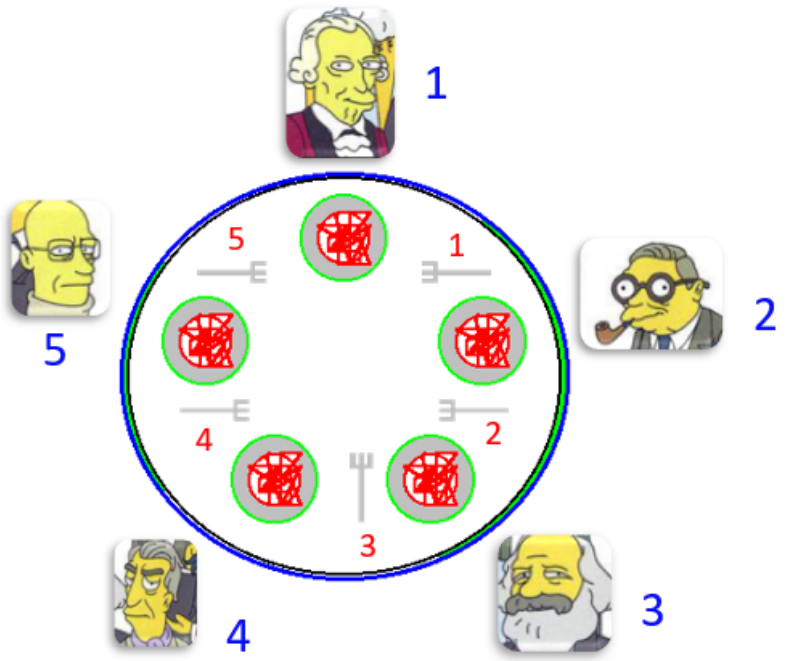
```
19      public void run(){ //Esta funcion se encargara de recibir los datos de
    herencia
20          while(true){
21              this.pensando();
22              this.monitor.tomarTenedores(this.indiceTenedor); //Una vez que
    puede comer, toma los tenedores
23              this.comiendo();
24              System.out.println("Filosofo " + nfilosofo +  " deja de comer y
    libera los tenedores" + (this.monitor.tenedorIzquierda(this.indiceTenedor) +
    1) + ", " + (this.monitor.tenedorDerecha(this.indiceTenedor) + 1) );
25              this.monitor.dejarTenedores(this.indiceTenedor);
26          }
27
28      }
29      public void pensando(){
30
31          System.out.println("Filosofo " + nfilosofo + " esta pensando en los
    S.O");
32          try {
33              sleep((long) (Math.random() * 4000)); //Funciona que determinara
    el tiempo que se la pasa pensando
34          } catch (InterruptedException ex) { }
35
36      }
37
38      public void comiendo(){
39          System.out.println("Filosofo " + nfilosofo + " se estresa y mejor se
    pone a comer");//Funciona que determinara el tiempo que se la pasa comiendo
40
41     System.out.println("Filosofo " + nfilosofo +  " toma los tenedores " +
    (this.monitor.tenedorIzquierda(this.indiceTenedor) + 1) + ", " +
    (this.monitor.tenedorDerecha(this.indiceTenedor) + 1) );
42
43          try {
44              sleep((long) (Math.random() * 4000));
45          } catch (InterruptedException ex) { }
46      }
47
48  }
```

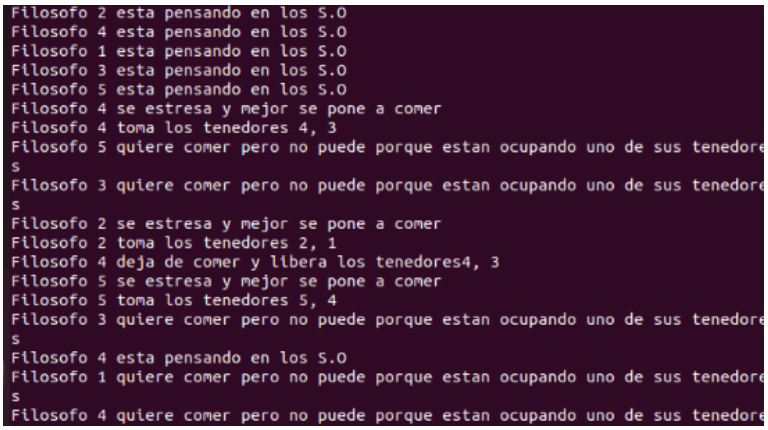
- En esta clase trabajaremos con los datos declarados en las otras clases, así mismo se definen las funciones que va a realizar el filósofo:tomar tenedores,liberar tenedores, bloqueo o liberación de los tenedores.Asimismo, asignaremos el tiempo que se tardara el filósofo en pensar y comer.

Ejecución Final



Para la ejecución a simple vista podemos observar que cumple con el principal propósito de los monitores: el bloqueo o modo de espera hasta que se liberen los recursos necesarios para la ejecución. Pero para una mejor demostración se realizará un prueba de escritorio para conocer mejor el funcionamiento.





	1	2	3	4	5
F4->3,4= S I			•	•	
F5->4,5=N O			•	•	
F3->2,3=N O			•	•	
F2->1,2=S I	•	•			
F4-----4,3	•	•			
F5->4,5=S I	•	•		•	•
F3->2,3=N O	•	•		•	•
F1->5,1=N O	•	•		•	•
F4->3,4=N O	•	•		•	•
F2-----1,2				•	•

- Para concluir, podemos observar que cumple con el propósito del problema y nos ejemplifica de manera clara su funcionamiento.Los únicos problemas que surgieron en su programación fue en el conteo donde los tenedores deben coincidir con cada filósofo pero después de varias pruebas se implementó una solución sencilla pero efectiva.

PARTE 2.- REPORTE Y VIDEO

En el reporte se deberá realizar un análisis de los elementos teórico-prácticos utilizados para resolver el problema explicando los siguientes puntos

• Descripción del problema y la forma en la que se expresará la solución

• Principales dificultades al realizar/comprender el programa

• Capturas de ejecuciones de las 3 soluciones para respaldar los análisis realizados.

Además del reporte, se deberá entregar el código fuente de los programas realizados, el cual deberá estar comentado en las secciones más relevantes del código. Adicional a los puntos anteriores, el alumno deberá elaborar un video (duración máxima de 25 min) donde se deberá explicar brevemente la solución de

cada problema y se deberá mostrar la ejecución del programa. En caso de realizar la práctica en equipo, los integrantes deben participar en la explicación de los 3 programas

Nota: Se permite que parte de la solución sea obtenida de fuentes confiables siempre y cuando estas se incluyan de manera correcta como Bibliografía de la práctica.

En caso de obtener código de internet, si no es una fuente confiable, no se indica de dónde se obtuvo, o no se explica adecuadamente, la práctica será anulada.

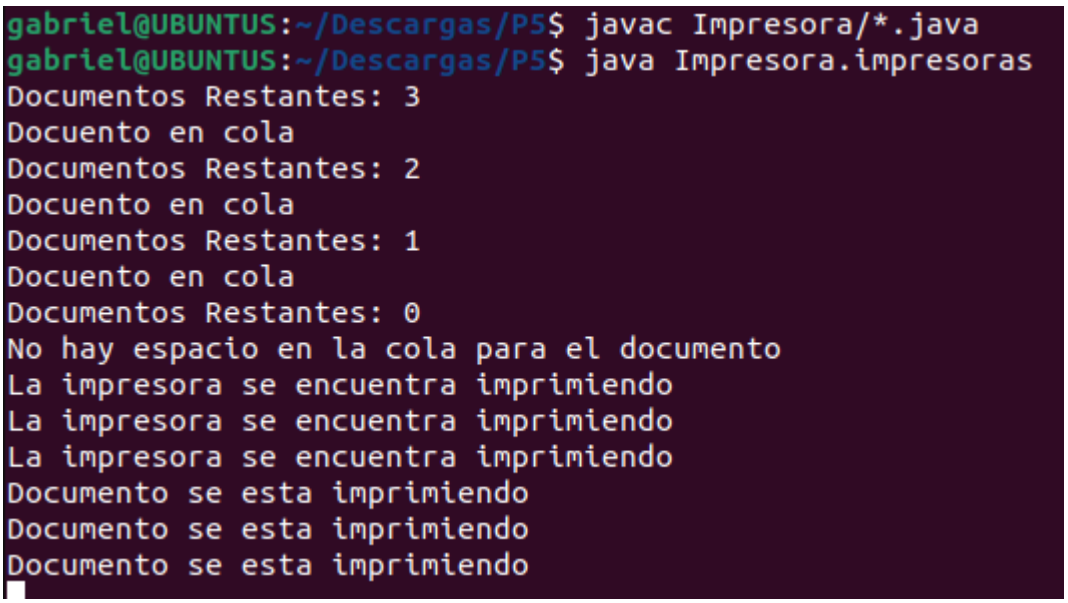
PARTE 3.- APLICACIÓN REAL

Analizando todas las soluciones llegamos a la conclusión de que una aplicación en la vida real sería el uso de impresoras comparado con el programa del barbero dormilón. Esto porque la impresora serviría como el barbero, ambos se ponen a chambear mientras tienen una “cola” de procesos, que serían los documentos para la impresora y los clientes para el barbero. Si no tienen nada en su “cola de espera” se ponen a dormir hasta que se les requiera para realizar su actividad, de igual forma los dos tienen un cupo “n” para la cantidad de procesos que pueden almacenar y de esta forma no saturarse y controlar la cantidad de solicitudes que se pueden llegar hacer. Ligado a esto, si un proceso se encuentra en espera para que se le corte el pelo y el barbero ya terminó con el anterior, este proceso ya puede pasar a cortarse el pelo, con las impresoras es igual, se imprime primero el primer documento conforme va llegando, termina la impresión y comienza a imprimirse el siguiente documento.

OPCIONAL PARA PUNTOS EXTRA

Codifica la solución indicada, agrega en el reporte las capturas de pantalla de la ejecución del programa y las dificultades que tuvo el equipo al realizarla. Agrega una carpeta con la documentación de este programa.

```java
package Impresora;
import java.util.concurrent.Semaphore; //Libreria para el uso de semaforos

public class impresoras{
        static Semaphore impresora = new Semaphore(0); //Declaracion del semaforo impresora
        static Semaphore documento = new Semaphore(0); //Declaracion del semaforo documento
        static Semaphore mutex = new Semaphore(1);   //Declaracion del semaforo mutex

        //Declaración de variables enteras
        static int documentosEnCola = 3; //docuentos en la cola
        static int documentosRestantes = 3;  //docuentos disponibles
        static int documentoEsperando = 0; //documentos en espera

        //Se define la función main(), y aqui es donde se van a crear e iniciar los hilos de los barberos y clientes

        public static void main(String[] args) throws InterruptedException{

                Thread ImpresoraThread1 = new Thread(new Impresora());
                Thread DocumentoThread1 = new Thread(new Documento());
                Thread DocumentoThread2 = new Thread(new Documento());
                Thread DocumentoThread3 = new Thread(new Documento());
                Thread DocumentoThread4 = new Thread(new Documento());

                ImpresoraThread1.start();
                DocumentoThread1.start();
                DocumentoThread2.start();
                DocumentoThread3.start();
                DocumentoThread4.start();
        }

        // Se define una clase Cliente para implementar la lógica de los cliente que llegan
        static class Documento implements Runnable{
                @Override
        public void run(){
                try{
                        /*acquire bloquea la ejecución del hilo en curso y queda a la espera de
                        que otro hilo llame a release() en este caso se encuentran más abajode código esperando
                documento y mutrx para que despues el impresora bloque nuevamente*/

                mutex.acquire();
                System.out.println("Documentos Restantes: "+documentosRestantes);
                documentosRestantes--;
                if (documentoEsperando < documentosEnCola) {
                        documentoEsperando++;
                        System.out.println("Docuento en cola"); //Mensaje que indica que un documento esta esperando su turno
                        documento.release();
                        mutex.release();
                        impresora.acquire();
                        imprimir();
                }else{
                        mutex.release();
                        System.out.println("No hay espacio en la cola para el documento");
                }
              }catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
        private void imprimir() throws InterruptedException{
                System.out.println("Documento se esta imprimiendo"); //Mensaje de que un documento esta en ejecución de su corte
                }
        }

        //Se define la clase Impresora que implementa la logica del barbero y que este se quede dormido hasta que llegue un nuevo documento
        static class Impresora implements Runnable{
                @Override
                public void run(){
                        while(true){
                                try{
                                        documento.acquire();
                                        mutex.acquire();
                                        documentoEsperando--;
                                        impresora.release();
                                        mutex.release();
                                        imprimir();
                                } catch (InterruptedException e){
                                }
                        }
                }

                private void imprimir() throws InterruptedException{
                        System.out.println("La impresora se encuentra imprimiendo");
                        }
                }
}
```



BIBLIOGRAFÍA:

[1] Silberschatz, A., Galvin, P. B. & Gagne, G. (2006). Fundamentos de sistemas operativos. McGraw-Hill Education.

[2] Walton, A. (2022, 17 febrero). Sincronización de Hilos en Java. Java desde Cero. https://javadesdecero.es/avanzado/sincronizacion-de-hilos/

[3] Sincronización de hilos. (s. f.). https://www.chuidiang.org/java/hilos/sincronizar_hilos_java.php

[4] Irfan, M. (2022, 15 enero). Qué es Monitor en Java. Delft Stack. https://www.delftstack.com/es/howto/java/monitor-in-java/

[5] Java - Monitores en java. (s. f.). https://www.lawebdelprogramador.com/foros/Java/1538729-Monitores-en-java.html

[6] Semáforos en C para Unix/Linux. (s. f.). https://www.chuidiang.org/clinux/ipcs/semaforo.php

[7] Semáforos en Java. (2017, 30 marzo). Stack Overflow en español. https://es.stackoverflow.com/questions/59304/sem%C3%A1foros-en-java

[8] Khintibidze, L. (2021, 30 marzo). Usar un semáforo en C. Delft Stack. https://www.delftstack.com/es/howto/c/semaphore-example-in-c/

[9] ANENBAUM, ANDREW S. (2009). SISTEMAS OPERATIVOS MODERNOS. PEARSON EDUCACIÓN

[10] ANENBAUM, ANDREW S. (1998). SISTEMAS OPERATIVOS: Diseño e implementación. PRENTICE HELL