BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

PROGRAMACIÓN CONCURRENTE Y PARALELA

PRÁCTICA FINAL

EXCLUSIÓN MUTUA

RAYMUNDO DIAZ AGUILAR 201734010

NOVIEMBRE 2020

Planteamiento del problema: Exclusión Mutua

La exclusión mutua es la actividad que realiza el sistema operativo para evitar que dos o más procesos ingresen al mismo tiempo a un área de datos compartidos o accedan a un mismo recurso. En otras palabras, es la condición por la cual, de un conjunto de procesos, sólo uno puede acceder a un recurso dado o realizar una función dada en un instante de tiempo.

El problema que aborda la exclusión mutua es un problema de intercambio de recursos: ¿cómo puede un sistema de software controlar el acceso de múltiples procesos a un recurso compartido, cuando cada proceso necesita el control exclusivo de ese recurso mientras realiza su trabajo?
La solución de exclusión mutua para esto hace que el recurso compartido esté disponible solo mientras el proceso está en un segmento de código específico llamado sección crítica.

Algoritmos de la Práctica Final

Se implementaron los siguientes algoritmos, anteriormente realizados en clase:

- Condiciones de Competencia
- Desactivación de Interrupciones
- Variable de Cerradura
- Mutex
- Mutex de java.util.Concurrent.Lock
- Algoritmo de Dekker

Desarrollo de la Práctica

Se utilizó el programa condicionesCompetencias realizado en clase en el cual se muestra gráficamente el problema de los perritos.
La variable rc es el recurso compartido que será utilizado por los hilos t1,t2,t3 y t4 , la variable vcerr es la Cerradura y mutex la variable Mutex del recurso compartido y los enteros de cada hilo representan sus turnos para el algoritmo de Dekker. Después son asignados los nombres de los hilos y representan los perritos.

```
public condicionesCompetencias()
{
    setTitle("La cena de los perritos");
    initComponents();
    rc = new rCompartido();
    cerr = new Cerradura();
    mutex = new Mutex();
    t1 = new Hilo(area1,rc,cerr,1);
    t2 = new Hilo(area2,rc,cerr,2);
    t3 = new Hilo(area3,rc,cerr,3);
    t4 = new Hilo(area4,rc,cerr,4);

    t1.setName("Perrito 1");
    t2.setName("Perrito 2");
    t3.setName("Perrito 3");
    t4.setName("Perrito 4");

}
```

```
public class rCompartido {
    public Interrupcion i1 = new Interrupcion();
    public Interrupcion i2 = new Interrupcion();
    public Interrupcion i3 = new Interrupcion();
    private Cerradura cerr;
    private String rc;
    private Mutex mutex = new Mutex();
    boolean pDentro;
    boolean pDesea;
    int turno;

    public rCompartido(Cerradura cerr, String rc, int turno) {
        this.cerr = cerr;
        this.rc = rc;
        this.turno = turno;
    }
```

Esta es la clase del recurso compartido, cuenta con 3 variables de tipo Interrupción para realizar el algoritmo de desactivación de Interrupciones, con una variable de tipo Cerradura para el algoritmo de Cerradura, también cuenta con una variable de tipo Mutex para el algoritmo de mutex y con dos variables booleanas y una de tipo entero para realizar el algoritmo de Dekker.

```
    public String getRc() {
        return rc;
    }

    public void setRc(String dato){
        this.rc=dato;
    }

    public String getCerradura() {
        String aux = "En espera...";
        if(cerr.isCerr()){
            cerr.Cierra();
            aux=rc;
            cerr.Abre();
        }
        return aux;
    }

    public void setCerradura(String dato){
        if(!cerr.isCerr()){
            cerr.Cierra();
            this.rc=dato;
            this.rc = dato;
        }
    }
```

Estos son los getters y setters del recurso compartido, tiene dos aparte especialmente para el algoritmo de variable de cerradura, ya que los otros algoritmos deben funcionar sin una variable de cerradura que se encargue de brindar la exclusión mutua.

Clase Interrupción

```
public class Interrupcion{
    boolean inter;

    Interrupcion(){
        inter = true;
    }

    public boolean getInter(){
        return inter;
    }

    public void setInter(boolean inter){
        this.inter=inter;
    }

}
```

Esta es la clase Interrupción, cuenta con una sola variable que es booleana para indicar el estado de la interrupción y con sus métodos get y set.

Clase Cerradura

```
public class Cerradura {
    private boolean vcerr;

    Cerradura(){
        vcerr=false;
    }

    public boolean isCerr(){
        return vcerr;
    }

    public void setCerr(boolean vcerr){
        this.vcerr = vcerr;
    }

    public void Cierra(){
        vcerr=true;
    }

    public void Abre(){
        vcerr=false;
    }
```

La clase Cerradura cuenta con una sola variable booleana llamada vcerr, que va a determinar si la cerradura del recurso compartido por los procesos está abierta o cerrada, esto dependiendo de si un proceso está ocupando la sección compartida o no.
Tiene los métodos de get y set y cuenta con dos métodos que se encargan tanto de abrir la cerradura como de cerrarla.

Clase Mutex

```
public class Mutex {

    public Mutex(){}

    public void lock(){
        try{
            wait(100);
        }catch(Exception e){e.printStackTrace();}
    }

    public void unlock(){
        try{
            notify();
        }catch(Exception e){e.printStackTrace();}
    }
}
```

La clase Mutex cuenta dos métodos lock() y unlock() que se encargan de bloquear la sección crítica y también de desbloquearla, esto lo hacen con los métodos wait() y notify() respectivamente.

Algoritmos

A continuación se muestran los algoritmos usados en la práctica final, todos están implementados en el método run() de la clase Hilo y son ejecutados de acuerdo a qué algoritmo haya sido seleccionado a la hora de ejecutar el programa, esto en el menú de algoritmos.
Cuando se selecciona un algoritmo se vuelve verdadera una función booleana que lo identifica y desactiva todos los demás algoritmos. En la clase Hilo con condiciones if se determina que algoritmo ejecutar.

Condiciones de Competencia

```
    //condiciones competencia
    if(this.algCond == true){
        System.out.println("condiciones competencia");
    try{
       inicio=true;
       ejec=true;
        while(true){
            rc.setRc(this.getName());
            area.append(rc.getRc()+": Eats\n");
            Thread.sleep(500);
            if(ejec == false){
                while(pausar == true)
                {
                   Thread.sleep(1);
                }
                if(detener == true){
                    break;
                }
            }
        }
    }catch(Exception e ){ e.printStackTrace();}
}
```

El algoritmo Condiciones de Competencia que simplemente imprime en las áreas de texto, tiene dos sentencias if que determinan si el programa está en pausa o si tiene que detenerse por completo. Esto lo realiza con variables booleanas que cambian a verdadero cuando se oprime su botón en la interfaz gráfica.

Desactivación de Interrupciones

```
//interrupcion
if(this.algInt == true){
    System.out.println("algoritmo interrupciones");
try{
   inicio=true;
   while(true){
        if(rc.i1.inter == false && rc.i2.inter == false && rc.i3.inter == false
        {
            if(this.getName()=="Perrito 1"){
                while(true){
                    rc.setRc(this.getName());
                    area.append(rc.getRc()+": Eats\n");
                    Thread.sleep(500);
                }
            }
        sc=true;
        }
        if(sc == true){
        System.out.println("Recurso Compartido ocupado\n");
        exit();
        }
        rc.setRc(this.getName());
        area.append(rc.getRc()+": Eats\n");
        Thread.sleep(500);
    }
}catch(Exception e ){ e.printStackTrace();}
}
```

El algoritmo Desactivación de Interrupciones debe verificar que las interrupciones estén activadas en todo momento, si las interrupciones se desactivan con el botón, un proceso quedará dentro de la sección crítica y no se cambiará de proceso para imprimir en las áreas de texto, por lo que los demás procesos quedarán esperando de manera indefinida. En el algoritmo se opta por detener los procesos inactivos.

Variable Cerradura

```
//cerradura
if(this.algCerr == true){
    System.out.println("algoritmo cerradura");
try{
    inicio=true;
    while(true){
        if(!cerr.isCerr()){
        cerr.Cierra();
        rc.setCerradura(this.getName());
        this.area.append(rc.getCerradura()+": Eats\n");
        cerr.Abre();
        }
        Thread.sleep(500);
        if(detener == true)
            break;
    }
}catch(Exception e ){e.printStackTrace();}
}
```

El algoritmo de Cerradura antes de imprimir en la GUI verifica que la cerradura está cerrada con el primer if, sino cierra la sección crítica y después de usar el recurso compartido abre la cerradura.

Variable Mutex

```
//Mutex
if(this.algMutex == true){
    System.out.println("algoritmo mutex");
try{
    while(true){
        mutex.lock();
        rc.setRc(this.getName());
        area.append(rc.getRc() + "\n");
        Thread.sleep(500);
        mutex.unlock();
        if(detener == true)
            break;
    }
    }catch(Exception e){System.out.println("err");}
}
```

El algoritmo Mutex es similar al de Cerradura, antes de imprimir en la interfaz gráfica se encarga de bloquear el mutex con el método lock() y una vez que haya terminado de usar el recurso compartido para imprimir, desbloquea el mutex con el método unlock(). Este mutex fue implementado en clase con ayuda del profesor.

Variable Mutex Lock

```
//Lock
if(this.algLock == true){
    System.out.println("algoritmo lock");

try{
    while(true){
        lock.lock();
        rc.setRc(this.getName());
        area.append(rc.getRc() + "\n");
        if(detener == true)
            break;
    }
    }catch(Exception e){e.printStackTrace();}
finally{lock.unlock();}
}
```

El algoritmo Lock funciona de la misma manera que el de Mutex, solo que en este caso, la variable mutex es llamada Lock y es usada directamente de la librería java.util.concurrent.locks.ReentrantLock;. Para poder usar el Lock, debemos crear una variable ReentrantLock nueva en este caso es lock. En el algoritmo, antes de usar el rc lo bloquea con lock.lock() y una vez que termina de usarlo la desbloquea en el bloque finally{lock.unlock()} del try catch. Esto asegura la exclusión mutua.

Dekker (Alternancia estricta)

```
//Dekker
if(this.algDekker == true){
    System.out.println("algoritmo dekker");
try
 {
    while(true){
        rc.pDesea = true;
        rc.pDentro = false;
        while(rc.pDesea){
            if(rc.turno==this.turno && rc.pDentro == false){
                rc.pDentro=true;
                rc.setRc(this.getName());
                area.append(rc.getRc() + "\n");
                Thread.sleep(1000);
                rc.turno = (int) Math.floor(Math.random()*(1-4+1)+1);
            }
            Thread.sleep(1000);
            rc.pDentro=false;
            rc.pDesea=true;
            turno= (int) Math.floor(Math.random()*(1-4+1)+1);
        }
        if(detener == true)
            break;
    }
 }catch(Exception e){e.printStackTrace();}
}
```

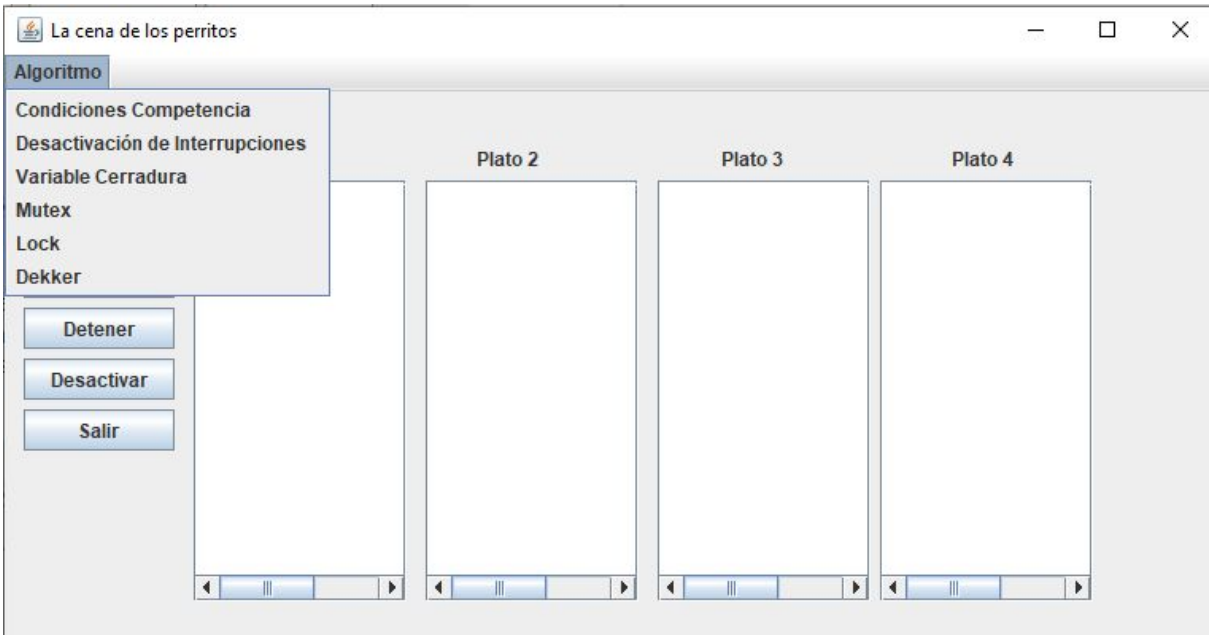
El algoritmo de Dekker es implementado con 2 variables booleanas: pDentro y pDesea que determinan si hay un proceso dentro de la sección crítica y si un proceso desea entrar a la sección crítica. También, cada proceso debe contar con un turno de uso del rc que es una variable de tipo entero y el rc de igual forma cuenta con una variable turno.

Para que un proceso pueda usar el recurso compartido la variable pDentro debe ser falsa y también necesita que el turno del rc sea igual a su turno de proceso, si ambas condiciones son verdaderas el proceso podrá entrar a la sección crítica y los demás procesos deberán esperar a que la sc esté desocupada y sea su turno de entrar.

El turno del proceso que va a entrar es determinado por el recurso compartido, esto lo hace mediante la función Math.floor(Math.random()*(1-4+1)+1) que genera un número aleatorio del 1 al 4 y lo determina cada vez que un proceso termina de usar el recurso compartido en la sección crítica.

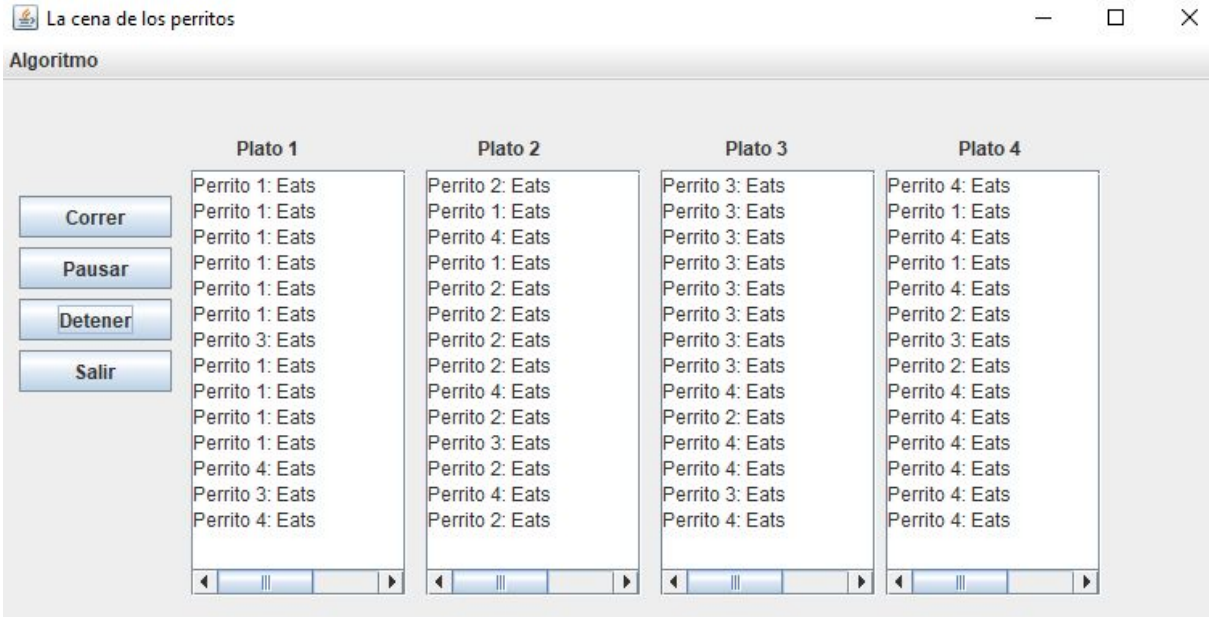
Cuando un proceso entra a la sección crítica, la variable pDentro se vuelve verdadera y una vez que sale de la sección crítica se vuelve falsa para volver a solicitar la entrada por parte de los procesos.

Pruebas y Resultados



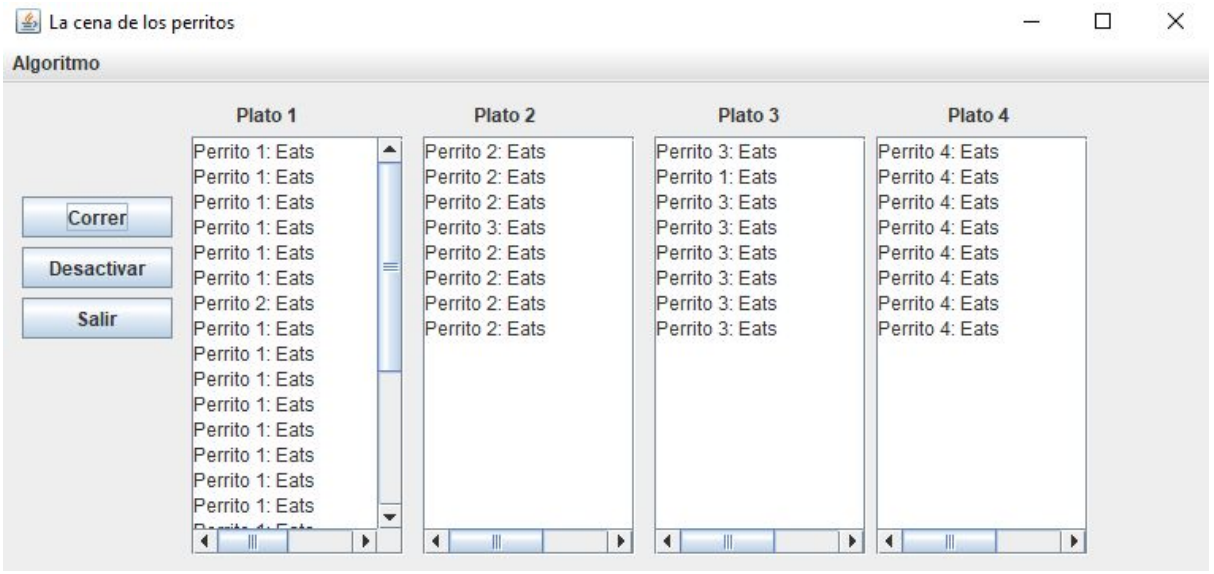
Esta es la interfaz gráfica antes de ser iniciado el programa. En el menú algoritmo se despliegan las opciones de algoritmos disponibles.
Una vez iniciado el programa se mostrará en cada área de texto como líneas que son escritas por los hilos utilizando el recurso compartido.
Se mostrará la salida de cada uno de los algoritmos y cómo funcionan de manera distinta.

Condiciones de Competencia



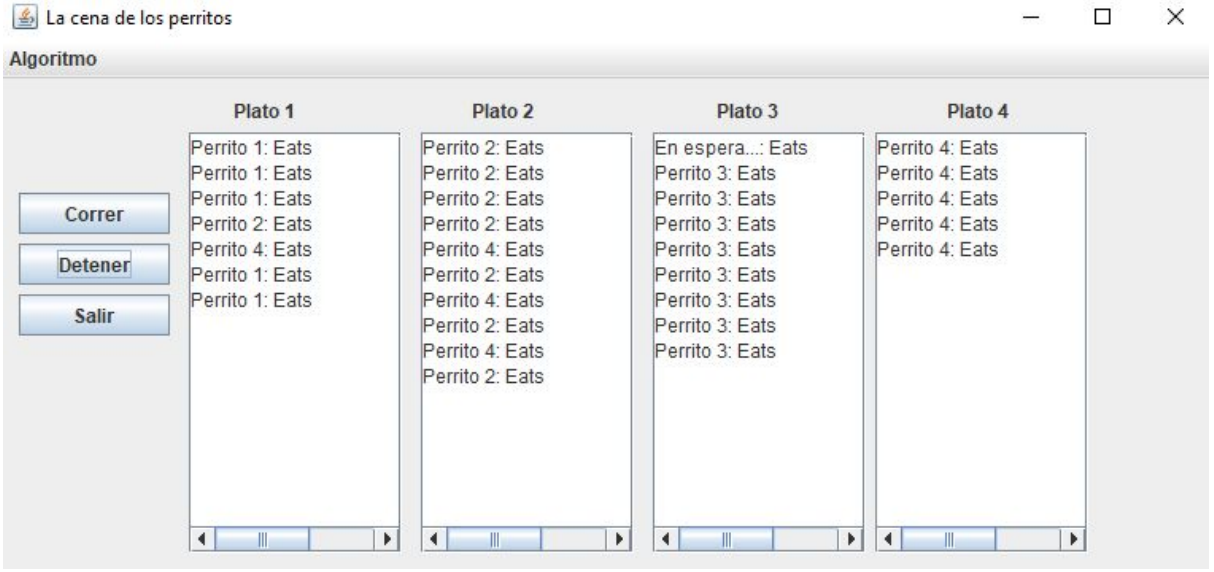
En este algoritmo se muestra la condición de competencia de los hilos, que no realizan de manera ordenada la impresión y buscan obtener el recurso compartido en todo momento.

Desactivación de Interrupciones



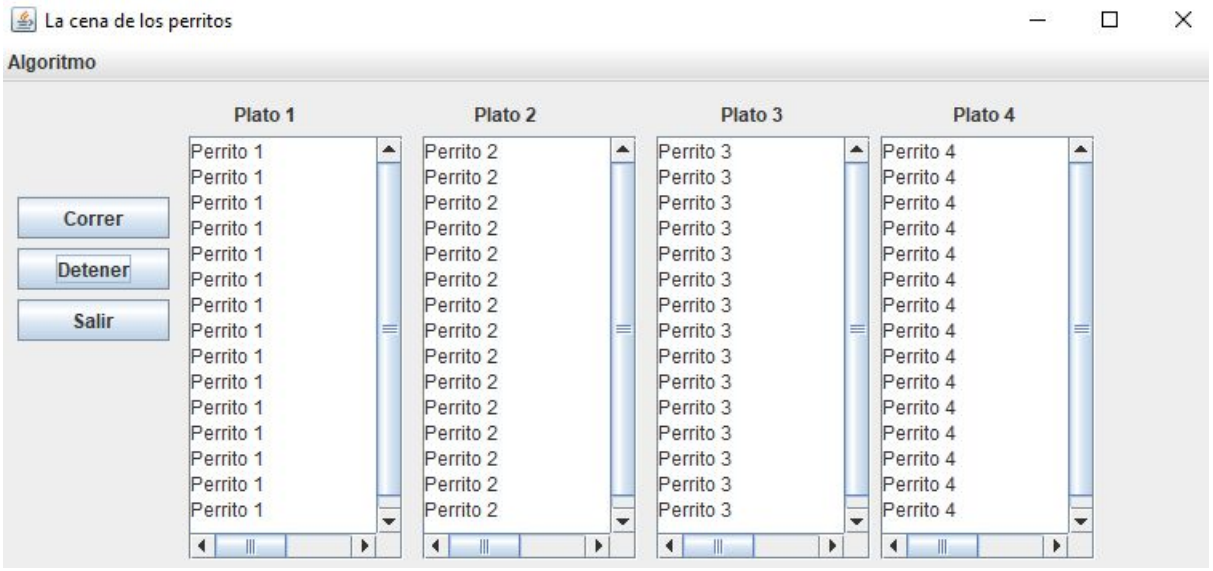
Con este algoritmo se puede ver como existe un mejor orden a la hora de la impresión pero si desactivamos las interrupciones, un proceso queda dentro de la sección crítica y los demás no pueden volver a entrar, ya que no hay interrupción que realice el cambio de proceso.

Variable Cerradura



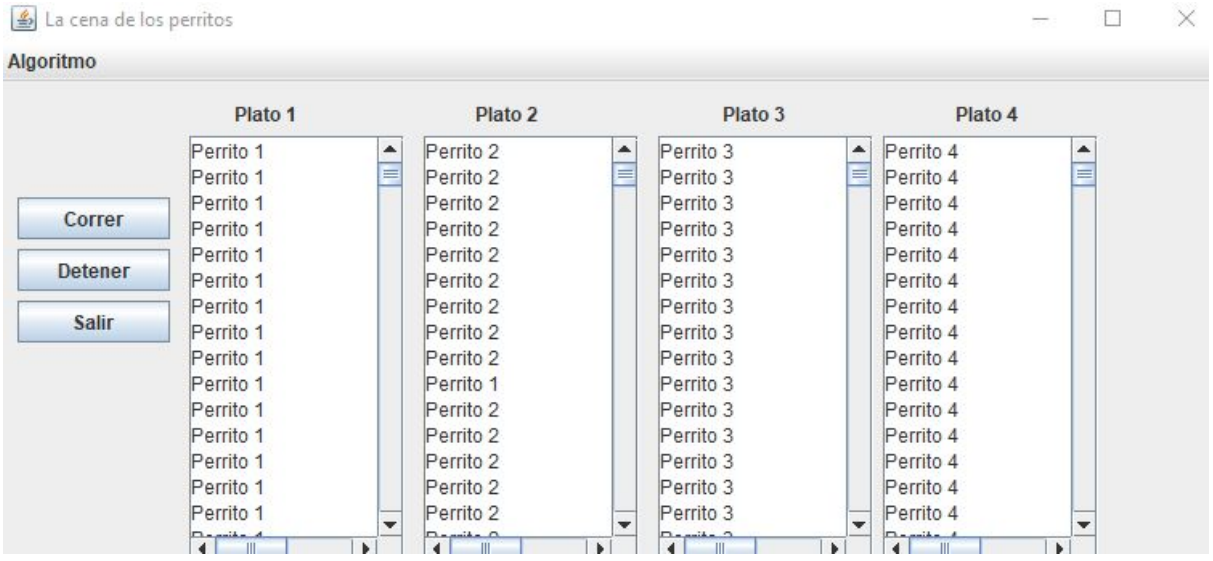
Con la variable de Cerradura los procesos deben esperar a que la sección crítica esté abierta, si no, deben esperar a que un proceso salga y se vuelva a abrir. Se puede ver que ciertos procesos entran a la sección crítica antes que otros y no importa si ya ha entrado y otro proceso aún no lo hace. Se puede ver que este algoritmo no es eficiente del todo.

Mutex



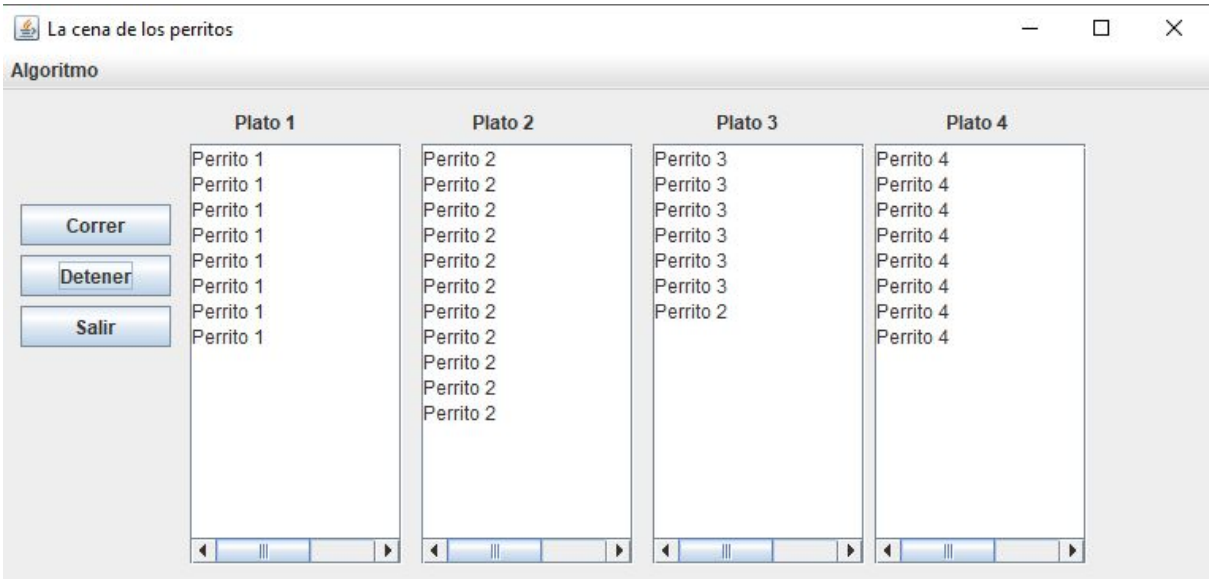
Este algoritmo es eficiente a la hora de garantizar la exclusión mutua. Los procesos esperan a que la sección crítica esté desbloqueada para usar el recurso compartido e imprimir en las áreas de texto.

Lock



Este algoritmo fue implementado con la librería concurrent.Lock de java y funciona de la misma manera que Mutex, es eficiente, garantiza la exclusión mutua y los procesos deben esperar a que la sección crítica esté libre para poder usar el recurso compartido.

Dekker (Alternancia estricta)



Este algoritmo es eficiente pero llega a tener fallos, la impresión de los hilos es por turnos, por lo que cada hilo escribirá solo uno a la vez, por lo que hay un orden y se garantiza la exclusión mutua pero para eliminar los fallos que puede tener se debe implementar la versión final del algoritmo de Dekker (este algoritmo es el de alternancia estricta, que es la primera versión del algoritmo de Dekker).

Conclusión

La exclusión mutua es una condición que se encarga de que dos o más procesos solucionen la condición de competencia, permitiendo que una aplicación multiprogramada con un solo procesador funcione de manera óptima evitando problemas que lleven a graves fallos durante la ejecución del programa.

Existen métodos tanto por software como por hardware que se encargan de asegurar la exclusión mutua, aunque la efectividad dependerá del método o del algoritmo implementado ya que algunos tienen un mejor rendimiento a la hora de sincronizar los procesos.

Referencias

-Mariano Larios Gómez. Programación Concurrente: Un enfoque práctico. Facultad de Ciencias de la Computación, BUAP.

-http://lsi.vc.ehu.es/pablogn/docencia/manuales/SO/TemasSOuJaen/CONCURRENCIA/2UtilizandoMemoriaCompartidaI.html