2.5. Escribe una clase llamada *ModernSuperMarket* que implemente el funcionamiento de N cajas de supermercado. Los mismos M clientes del supermercado realizarán el mismo proceso que en el ejercicio anterior, situándose cuando han realizado la compra, en este caso, en una única cola. Cuando cualquier caja esté disponible, el primero de la cola será atendido en la caja correspondiente. Calcula el tiempo medio de espera por cliente y compáralo con el tiempo medio que se obtendría en el ejercicio anterior. ¿Cuál de las dos alternativas es más eficiente? ¿Cuál elegirías si tú tuvieras un supermercado? Razona la respuesta.

Solución

Este problema puede ser tratado o bien como un caso práctico o como un ejercicio para alumnos más avanzados, debido a la dificultad de gestionar una única cola con diferentes cajas.

Al igual que en el ejercicio anterior, para la resolución del problema se crean tantos *threads* como clientes haya comprando. Para gestionar el cobro de los clientes se implementa una única cola en la clase *Cola*. Añadir elementos y sacar elementos de la cola son operaciones sincronizadas mediante métodos sincronizados ya que la cola es única y hay que evitar condiciones de carrera en su acceso. El método *esperar()* detiene a los *threads* clientes en la cola mediante una operación *wait()* hasta que les llega el turno de ser atendidos (ser el primero en la cola y haya alguna caja libre). Sin embargo, cuando se atiende a un cliente, no se hace en la propia cola, sino en varias cajas que pueden estar funcionando al mismo tiempo. Para implementarlo, se crean diferentes objetos *Caja* que se pueden ejecutar en paralelo cada uno con su método *atender()*. Para dar paso al siguiente cliente, este proceso no se puede realizar en la propia caja, ya que los clientes están esperando en la *Cola*. Por ello es necesario un método *finalizar_compra()* donde se hace el *signal()* indicar el abandono de la caja a los clientes esperando en la cola.

Comprobando los resultados de ambas alternativas, después de varias ejecuciones, se puede observar cómo el *ModernSuperMarket* destaca frente a *SuperMarket*. Esto se debe a que si, por cualquier motivo, se forma un retraso en una de las colas, todos los clientes que están esperando en esa cola se ven afectados. En cambio, en la

alternativa de una única cola, el resto de cajas palia este comportamiento. Si una caja se retrasa, al haber otra caja libre ese retraso no penaliza de forma significativa a los clientes.

```
import java.util.Random;
class Resultados{
public static int ganancias;
public static long tiempo_espera;
public static int clientes_atendidos;
}
class Caja {
private static final int MAX_TIME = 1000;
private boolean ocupada;

public Caja() {
this.ocupada = false;
}
public boolean ocupada(){
return ocupada;
}
synchronized public void atender(int pago) throws
InterruptedException {
ocupada = true;
int tiempo_atencion = new Random().nextInt(MAX_TIME);
Thread.sleep(tiempo_atencion);
Resultados.ganancias += pago;
Resultados.clientes_atendidos++;
ocupada = false;
}
}
class Cola{
class Nodo {
int cliente;
Nodo sig;
}
Nodo raiz,fondo;
Caja cajas[];
int N;
public Cola(int N) {
raiz=null;
fondo=null;
```

```
this.N = N;
cajas = new Caja[N];
for (int i= 0; i < N; i++){
cajas[i]= new Caja();
}
}
private boolean vacia (){
if (raiz == null)
return true;

else
return false;
}
private int cajaLibre(){
int i = 0;
while (i < N) {
if (!cajas[i].ocupada()){
break;
}
i++;
}
return i;
}
synchronized public int esperar (int id_cliente) throws
InterruptedException
{
int caja_id;
Nodo nuevo;
nuevo = new Nodo ();
nuevo.cliente = id_cliente;
nuevo.sig = null;
if (vacia ()) {
raiz = nuevo;
fondo = nuevo;
} else {
fondo.sig = nuevo;
fondo = nuevo;
}
// Esperar hasta el turno
while (((caja_id = cajaLibre()) == N) ||
(raiz.cliente != id_cliente) ){
// Me bloqueo hasta que sea mi turno
wait();
}
//Salgo de la cola
raiz = raiz.sig;
return caja_id;
}

public void atender(int id_caja, int pago) throws
InterruptedException{
cajas[id_caja].atender(pago);
}
synchronized public void finalizar_compra() throws
InterruptedException {
notify();
}
}
class Cliente extends Thread {
private static final int MAX_DELAY = 2000;
private static final int MAX_COST = 100;
private int id;
private Cola cola;
Cliente (int id, Cola cola) {
```

```
this.id = id;
this.cola = cola;
}
public void run() {
try {
int numero_caja;
System.out.println(“Cliente “ + id + “
realizando compra”);
Thread.sleep(new
Random().nextInt(MAX_DELAY));
long s = System.currentTimeMillis();
numero_caja = cola.esperar(id);
cola.atender(numero_caja, new
Random().nextInt(MAX_COST));
System.out.println(“Cliente “ + id + “ atendido
en caja “ + numero_caja);
cola.finalizar_compra();
System.out.println(“Cliente “ + id + “
finalizando”);
long espera = System.currentTimeMillis() - s;

Resultados.tiempo_espera += espera;
System.out.println(“Cliente “ + id + “ saliendo
despues de esperar “ + espera);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
public class ModernSuperMarket {
public static void main(String[] args) throws
InterruptedException {
int N = Integer.parseInt (args[0]);
Cola cola = new Cola(N);
int M = Integer.parseInt (args[1]);
Cliente clientes[] = new Cliente[M];
for (int i= 0; i < M; i++){
clientes[i]= new Cliente(i,cola);
clientes[i].start();
}
try {
for (int i= 0; i < M; i++){
clientes[i].join();
}
} catch (InterruptedException ex) {
System.out.println(“Hilo principal interrumpido.”);
}
System.out.println(“Supermercado cerrado.”);
System.out.println(“Ganancias: “ + Resultados.ganancias);
System.out.println(“Tiempo medio de espera: “ +
(Resultados.tiempo_espera / Resultados.clientes_atendidos));
}
}
```

2.6. Escribe una clase llamada *Parking* que reciba el número de plazas del *parking* y el número de coches existentes en el sistema. Se deben crear tantos *threads* como coches haya. El *parking* dispondrá de una única entrada y una única salida. En la entrada de vehículos habrá un dispositivo de control que permita o impida el acceso de los mismos al *parking*, dependiendo del estado actual del mismo (plazas de aparcamiento disponibles). Los tiempos de espera de los vehículos dentro del *parking* son aleatorios. En el momento en el que un vehículo sale del *parking*, notifica al dispositivo de control el número de la plaza que tenía asignada y se libera la plaza que estuviera ocupando, quedando así estas nuevamente disponibles. Un vehículo que ha salido del *parking* esperará un tiempo aleatorio para volver a entrar nuevamente en el mismo. Por tanto, los vehículos estarán entrando y saliendo indefinidamente del *parking*. Es importante que se diseñe el programa de tal forma que se asegure que, antes o después, un vehículo que permanece esperando a la entrada del *parking* entrará en el mismo (no se produzca inanición).

Solución

Para resolver el problema se crean tantos *threads* como coches se quieran simular. Cada coche será un *thread* y compartirán la barrera del *parking*, la cual se utilizará como mecanismo de sincronización. Los métodos de acceso al *parking entrada*() y *salida*() deben estar sincronizados para provocar que los únicos accesos que modifican las plazas del *parking* se realicen en exclusión mutua. El método *entrada()* que ejecutan los coches en
la barrera para entrar, los detiene mediante una operación *wait()* hasta que les llega el turno de ser atendidos. Al salir del *parking*, el método *salida()* provoca que haya un hueco dentro del *parking*, por lo que se le da paso al siguiente coche mediante la operación *signal()*.

```
import java.util.Random;
class Barrera {
private int plazas[];
private int n_plazas;
private int libres;
Barrera (int N){
n_plazas = N;
plazas = new int[N];
for (int i=0; i<n_plazas; i++) {
plazas[i] = 0;
}
libres = n_plazas;
}
synchronized public int entrada(int coche) throws
InterruptedException {
int plaza=0;
imprimir();

while (libres == 0) {
System.out.println("Coche " + coche + "
esperando");
wait();
}
while (plazas[plaza] != 0) {
plaza++;
}
plazas[plaza] = coche;
libres--;
return plaza;
}
synchronized public void salida(int plaza) {
plazas[plaza] = 0;
libres++;
notify();
}
public void imprimir() {
System.out.print("Parking: ");
for (int i=0; i<n_plazas; i++) {
System.out.print("[" + plazas[i] + "] ");
}
System.out.println("");
}
```

```
}
class Coche extends Thread {
private static final int MAX_DELAY = 2000;
private int id;
private Barrera barrera;
Coche (int id, Barrera bar) {
this.id = id;
this.barrera = bar;
}
public void run() {
try {
Thread.sleep(new
Random().nextInt(MAX_DELAY));

System.out.println("Coche " + id + " intenta
entrar en parking");
int plaza = barrera.entrada(id);
System.out.println("Coche " + id + " aparcado
en " + plaza);
barrera.imprimir();
Thread.sleep(new
Random().nextInt(MAX_DELAY));
barrera.salida(plaza);
System.out.println("Coche " + id +
" saliendo");
barrera.imprimir();
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
public class Parking {
public static void main(String[] args) {
int N = Integer.parseInt (args[0]);
//Inicializar el parking
Barrera barrera = new Barrera(N);
int C = Integer.parseInt (args[1]);
Coche coches[] = new Coche[C];
for (int i= 0; i < C; i++){
coches[i]= new Coche(i+1, barrera);
coches[i].start();
}
try {
for (int i= 0; i < C; i++){
coches[i].join();
}
} catch (InterruptedException ex) {
System.out.println("Hilo principal interrumpido.");
}
```

2.7. Escribe una clase llamada *ParkingCamion* que reciba el número de plazas del *parking*, el número de coches y el número de camiones existentes en el sistema. Dicha clase debe realizar lo mismo que la clase *Parking* pero debe permitir a su vez aparcar camiones. Mientras un automóvil ocupa una plaza de aparcamiento dentro del *parking*, un camión ocupa dos plazas contiguas de aparcamiento. Hay que tener especial cuidado con la inanición de camiones, que puede producirse si están saliendo coches indefinidamente y asignando la nueva plaza a los coches que esperan en vez de esperar a que haya un hueco para el camión (un camión solo podrá acceder al *parking* si hay, al menos, dos plazas contiguas de aparcamiento libre).

Solución

Este problema puede ser tratado o bien como un caso práctico o como un ejercicio para alumnos más avanzados, debido al problema de inanición que se puede provocar al introducir camiones en el *parking*.

Al igual que en el ejercicio anterior, para la resolución del problema se crean tantos *threads* como coches y camiones se quieran simular. Cada vehículo será un *thread* de una clase diferente, siendo la clase *Coche* la misma que en el ejercicio anterior. Ambos vehículos comparten la barrera del *parking*, la cual se utilizará como mecanismo de sincronización global. En la barrera aparecen dos métodos nuevos, *entrada_camion*() y *salida_camion*() para gestionar la entrada y salida de camiones. El método *entrada_camion()* debe comprobar que hay hueco para el camión antes de darle paso. Sin embargo, esto puede resultar en inanición de camiones, ya que si el *parking* esta lleno de coches y sale un coche, siempre habrá hueco para otro coche impidiendo que pueda entrar un camión. Existen diferentes alternativas para solucionarlo, incluyendo la utilización de un semáforo para contar el número de plazas disponibles. Otro método más sencillo, es forzar que los coches no llenen todo el *parking* dejando una plaza libre. De esta forma, si uno de los coches que están al lado de la plaza libre sale, un camión podrá entrar, ocupando esas dos plazas.

```
import java.util.Random;
class Barrera {
private int plazas[];
private int n_plazas;
private int libres;
Barrera (int N){
n_plazas = N;
plazas = new int[N];
for (int i=0; i<n_plazas; i++) {
plazas[i] = 0;
}
libres = n_plazas;
}
private int plazaCamion(){
int plaza = 0;
do {
if ((plazas[plaza] == 0) &&
(plazas[plaza+1] == 0)) {
return plaza;
} else {
plaza++;
}
} while (plaza < n_plazas-1);
return n_plazas;
}
synchronized public int entrada(int coche) throws
InterruptedException {
int plaza=0;
imprimir();
// Dejamos un hueco para asegurar que pueden
// entrar los camiones
while (libres <= 1) {
System.out.println("Coche " + coche + "
esperando");
wait();
}
while (plazas[plaza] != 0) {
plaza++;
}
plazas[plaza] = coche;
libres--;
return plaza;
```

```
}
synchronized public int entrada_camion(int camion)
throws InterruptedException {
int plaza=0;
imprimir();
// Mientras no haya plaza
while ((plaza = plazaCamion()) == n_plazas) {
System.out.println("Camion " + camion + "
esperando");
wait();
}
```

```
plazas[plaza] = camion;
plazas[plaza+1] = camion;
libres -= 2;
return plaza;
}
synchronized public void salida(int plaza) {
plazas[plaza] = 0;
libres++;
notify();
}
synchronized public void salida_camion(int plaza) {
plazas[plaza] = 0;
plazas[plaza+1] = 0;
libres += 2;
notify();
notify();
}
public void imprimir() {
System.out.print("Parking: ");
for (int i=0; i<n_plazas; i++) {
System.out.print("[" + plazas[i] + "] ");
}
System.out.println("");
}
}
class Camion extends Thread {
private static final int MAX_DELAY = 2000;
private int id;
private Barrera barrera;
Camion (int id, Barrera bar) {
this.id = id;
this.barrera = bar;
}
public void run() {
try {
Thread.sleep(new
Random().nextInt(MAX_DELAY));
System.out.println("Camion " + id + " intenta
entrar en parking");
int plaza = barrera.entrada_camion(id);
System.out.println("Camion " + id + "
aparcado en plazas " + plaza +
" y " + (plaza+1));
barrera.imprimir();
Thread.sleep(new
Random().nextInt(MAX_DELAY));
barrera.salida_camion(plaza);
System.out.println("Camion " + id + "
saliendo");
barrera.imprimir();
} catch (InterruptedException e) {
```

```
e.printStackTrace();
}
}
}
public class ParkingCamion {
public static void main(String[] args) {
int N = Integer.parseInt (args[0]);
//Inicializar el parking
Barrera barrera = new Barrera(N);
int C = Integer.parseInt (args[1]);
Coche coches[] = new Coche[C];
for (int i= 0; i < C; i++){
// Seleccionamos ya en qué caja se situará
coches[i]= new Coche(i+1, barrera);
coches[i].start();
}
int Ca = Integer.parseInt (args[2]);
Camion camiones[] = new Camion[Ca];
for (int i= 0; i < Ca; i++){
// Seleccionamos ya en que caja se situara
camiones[i]= new Camion(i+100, barrera);
```

```
camiones[i].start();
}
try {
for (int i= 0; i < C; i++){
coches[i].join();
}
for (int i= 0; i < Ca; i++){
camiones[i].join();
}
} catch (InterruptedException ex) {
System.out.println("Hilo principal interrumpido.");
}
}
}
```

Solución del caso práctico

Enunciado

En este caso práctico se va a desarrollar una solución multitarea al problema clásico de la cena de filósofos. En una mesa redonda hay N filósofos sentados. En total tiene N palillos para comer arroz, estando cada palillo compartido por dos filósofos, uno a la izquierda y otro a la derecha. Como buenos filósofos, se dedican a pensar, aunque de vez en cuando les entra hambre y quieren comer. Para poder comer, un filósofo necesita utilizar los dos palillos que hay a sus lados.

Para implementar este problema se debe crear un programa principal que cree N hilos ejecutando el mismo código. Cada hilo representa un filósofo. Una vez creado, se realiza un bucle infinito de espera. Cada uno de los

hilos tendrá que realizar los siguientes pasos:

1. Imprimir un mensaje por pantalla ("Filósofo *i* pensando"), siendo *i* el identificador del filósofo.
2. Pensar durante un cierto tiempo aleatorio.
3. Imprimir un mensaje por pantalla ("Filósofo *i* quiere comer").
4. Intentar coger los palillos que necesita para comer. El filósofo 0 necesitará los palillos 0 y 1, el filósofo 1, los palillos 1 y 2, y así sucesivamente.
5. Cuando tenga el control de los palillos, imprimirá un mensaje en pantalla "Filósofo *i* comiendo".
6. El filósofo estará comiendo durante un tiempo aleatorio.
7. Una vez que ha finalizado de comer, dejará los palillos en su sitio.
8. Volver al paso 1.

Sin embargo, se pueden producir interbloqueos si, por ejemplo, todos los filósofos quieren comer a la vez. Si todos consiguen coger el palillo de su izquierda, ninguno podrá coger el de su derecha. Para ello se plantean varias soluciones:

nn Permitir que como máximo haya N − 1 filósofos sentados a la mesa.
nn Permitir a cada filósofo coger sus palillos solamente si ambos palillos están libres.
nn Solución asimétrica: un filósofo impar coge primero el palillo de la izquierda y luego el de la derecha. Un filósofo par los coge en el orden inverso.

Hay que implementar una solución al problema de los filósofos, solución que no presente un problema de interbloqueo. Por sencillez, se recomienda utilizar el método propuesto de solución asimétrica.

Solución

La solución típica está basada en la utilización de métodos sincronizados. Es decir, se pueden utilizar para hacer que los filósofos cojan el palillo izquierdo y después el derecho. En caso de que cuando se desee coger un palillo esté ya cogido previamente, se bloqueará con *wait*() a ese hilo hasta que el filósofo que tiene dicho palillo

lo desbloquee con el correspondiente *notify*(). Nótese que el objeto Palillo sobre el que se produce el bloqueo y desbloqueo es el mismo.

```
class Palillo
{
private int numero;
private boolean cogido;
public Palillo (int id)
{
this.numero= id;
this.cogido=false;
}
public int getId(){
return this.numero;
}
synchronized public void coger()
{
while (cogido)
{
try
{
System.out.println ("Palillo " + numero + " bloqueado");
wait();
} catch (InterruptedException e) {
System.out.println (" -Espera por palillo " + numero + "
interrumpida");
System.exit(1);
}
}
```

```
cogido = true;
System.out.println (“Palillo “ + numero + “ ha sido cogido”);
}
synchronized public void soltar()
{
```

```
cogido= false;
System.out.println (“Palillo “ + numero + “ ha sido soltado”);
notify();
}
}
class Filosofo extends Thread
{
protected Palillo ind_izq, ind_der;
protected int identidad;
static final protected int MAX_DELAY=1000;
public Filosofo (int id)
{
this.identidad= id;
ind_izq= Filosofos.mesa.palillo_izquierdo(id);
ind_der= Filosofos.mesa.palillo_derecho(id);
}
protected void pensar()
{
try
{
System.out.println (“Filósofo “ + identidad + “ está pensando”);
Thread.sleep((int) Math.random()*MAX_DELAY);
} catch (InterruptedException e) {
System.out.println (“Filósofo “ + identidad + “ interrumpido”);
System.exit(1);
}
}
protected void comer()
{
try
{
System.out.println (“Filósofo “ + identidad + “ esta comiendo”);
Thread.sleep((int) Math.random()*MAX_DELAY);
} catch (InterruptedException e) {
System.out.println (“Filósofo “ + identidad + “ interrumpido”);
System.exit(1);
}
}
protected void queriendo_comer()
{
System.out.println (“Filósofo “ + identidad + “ quiere comer”);
ind_izq.coger();
ind_der.coger();
}
protected void dejando_de_comer()
{
ind_izq.soltar();
ind_der.soltar();
}
public void run()
{
while (true)
{
pensar();
queriendo_comer();
comer();
dejando_de_comer();
```

```
}
}
}
class MesaCircular
{
private Palillo palillos[];
private int filosofos;
public MesaCircular (int personas)
{
this.filosofos= personas;
palillos= new Palillo[personas];
for (int i= 0; i < personas; i++)
palillos[i]= new Palillo(i);
}
public Palillo palillo_derecho(int i)
{
return palillos[(i+1)%filosofos];
}
public Palillo palillo_izquierdo(int i)
{
return palillos[i];
}
}
public class CenaFilosofos {
public static MesaCircular mesa;
public static void main(String[] args) throws InterruptedException {
int filosofos = Integer.parseInt (args[0]);
mesa = new MesaCircular(filosofos);
System.out.println („Sentados „ + filosofos + „ filósofos“);
for (int i= 0; i < filosofos; i++)
{
Filosofo f = new Filosofo(i);
f.start();
}
}
}
```

Otra forma para mejorar el tiempo de ejecución sería utilizar una sentencia sincronizada utilizando los propios palillos como objetos monitores. Como se deben coger dos, se puede poner una sentencia sincronizada dentro de otra, como se ve en el ejemplo.

```
class Palillo
{
private int numero;
public Palillo (int id)
{
this.numero= id;
}
public int getId(){
return this.numero;
}
}
class Filosofo extends Thread
{
protected Palillo ind_izq, ind_der;
protected int identidad;
static final protected int MAX_DELAY=1000;
public Filosofo (int id)
{
this.identidad= id;
ind_izq= CenaFilosofos.mesa.palillo_izquierdo(identidad);
ind_der= CenaFilosofos.mesa.palillo_derecho(identidad);
}
protected void pensar()
{
try
```

```
{
System.out.println (“Filósofo “ + identidad + “ esta pensando”);
Thread.sleep((int) Math.random()*MAX_DELAY);
} catch (InterruptedException e) {
System.out.println (“Filósofo “ + identidad + “ interrumpido”);
System.exit(1);
}
}
protected void comer()
{
try
{
System.out.println (“Filósofo “ + identidad + “ comiendo usando “ +
CenaFilosofos.mesa.palillo_izquierdo(identidad).getId() + “ y “ +
CenaFilosofos.mesa.palillo_derecho(identidad).getId());
Thread.sleep((int) Math.random()*MAX_DELAY);
} catch (InterruptedException e) {
System.out.println (“Filósofo “ + identidad + “ interrumpido”);
System.exit(1);
}
}
protected void queriendo_comer()
{
System.out.println (“Filosofo “ + identidad + “ quiere comer”);
synchronized(CenaFilosofos.mesa.palillo_izquierdo(identidad)) {
synchronized(CenaFilosofos.mesa.palillo_derecho(identidad)){
comer();
}
}
}
public void run()
{
while (true)
{
pensar();
queriendo_comer();
}
}
}
```

Para solucionar el interbloqueo se puede utilizar una solución asimétrica. Para ello solamente se debe modificar la forma en que se cogen los palillos, identificando si el filósofo que los coge es par o impar. Por ejemplo,
para la primera solución mostrada valdría con:

```
protected void queriendo_comer()
{
System.out.println (“Filósofo “ + identidad + “ quiere comer”);
if (identidad %2 ==0) {
ind_izq.coger();
ind_der.coger();
} else {
ind_der.coger();
ind_izq.coger();
}
}
```

Para la segunda solución el cambio a realizar sería:

```
protected void queriendo_comer()
{
System.out.println („Filósofo „ + identidad + „ quiere comer“);
if (identidad % 2 == 0) {
synchronized(CenaFilosofos.mesa.palillo_izquierdo(identidad)) {
synchronized(CenaFilosofos.mesa.palillo_derecho(identidad)) {
comer();
}
}
```

```
} else {
synchronized(CenaFilosofos.mesa.palillo_derecho(identidad)) {
synchronized(CenaFilosofos.mesa.palillo_izquierdo(identidad)) {
comer();
}
}
}
}
```