

qwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxmoricvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmoriqwertyumoriiopasdfghjklzxcvbnmoriqwertyumoriopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmoriqwertyumoriopasdfghjklzxcvbinmqwertyumoriopasdfghjklzxewtyucvbnmoriqwertyumoripsdfghjklztopxcmoriqwertyuopasdfghjklzixcvbnmoriertyumoriopasdhjiklzxcvbnmoriqwertyumorimoriopasdfghjklzxcvdasfgfhbnmtwreorirtyumoriopasdfghjklzxcvbnmoriqwertyumoriopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnm

|  |
| --- |
| Actividad 4: ejercicios hilos resueltos  2º Desarrollo de Aplicaciones Multiplataforma Programación de servicios y procesos  17/11/2021  Martínez Díez, Ángel Mori |

Contenido

[1. Saludo 3](#_Toc87339954)

[2. Check 6](#_Toc87339955)

[3 8](#_Toc87339956)

[4 8](#_Toc87339957)

# Orden y Saludo

import java.util.concurrent.Semaphore;

class Saludo extends Thread {

private Semaphore sem;

private int id;

Saludo(int orden, Semaphore s) {

this.id = orden;

this.sem = s;

}

public void run() {

if (id == 1) {

try {

sem.acquire();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println("Hola, soy el thread " + id);

if (id == 2) {

sem.release();

}

}

}

public class Orden {

public static void main(String[] args) {

Semaphore semaphore = new Semaphore(0);

Saludo t1 = new Saludo(1, semaphore);

Saludo t2 = new Saludo(2, semaphore);

t1.start();

t2.start();

try {

t1.join();

t2.join();

} catch (InterruptedException e) {

System.out.println("Hilo principal del proceso interrumpido.");

}

System.out.println("Proceso acabando.");

}

}

El programa comienza en la función *main* de la clase Orden. Su ejecución es bastante simple.

Primero se crea un semáforo con cero permisos que se usa para crear los dos siguientes objetos de tipo Saludo. A continuación, con los métodos *start*, se inician dos hilos independientes y en las siguientes líneas el programa principal decide esperar a estos hilos con los métodos *join*. Primero a un hijo y luego al otro.

El programa no continúa hasta que el hilo al que está esperando termina su ejecución o se lanza una excepción de tipo *InterruptedException*. Por último se imprime la frase “Proceso acabado”.

La miga del programa está en los métodos *run* de la clase Saludo. Estos son los que se ejecutan al llamar al método start de una clase que hereda de *Thread*.

Analizamos primero los atributos de la clase Saludo: un semáforo, que hemos visto que en nuestro código será el mismo; y un id, que será 1 o 2.

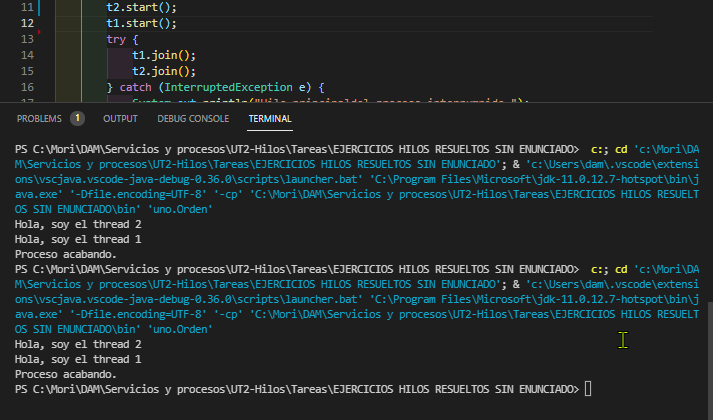
Veamos que hace cada hilo:

t1: como tiene de id 1, pide un permiso al semáforo con el método *acquire*. Los permisos disponibles del semáforo eran 0 al crearlos, por lo que en principio le toca quedarse esperando hasta que el semáforo le de permiso para continuar.

t2: como su id es 2, no pide permiso al semáforo. Lo que hace es imprimir “Hola, soy el thread 2” y libera un permiso del semáforo antes de acabar su ejecución. El semáforo tiene ahora un permiso disponible.

t1: recibe un permiso por parte del semáforo al liberarlo t2. El semáforo no tiene ahora ningún permiso disponible. Imprime la frase “Hola, soy el thread 1” y acaba su ejecución.

Dijimos que el programa principal esperaba a t1 y a t2 —en ese orden—, por lo que al acabar t1, pasa a esperar a t2, que ya ha acabado dado que para que acabe t1 tiene que haber finalizado t2 su ejecución.

Siempre se va a ejecutar t2 antes que t1. Independientemente de el orden en el que se hagan los *start*. 

# Check, Lector y Escritor

package dos;

import java.nio.IntBuffer;

class Escritor extends Thread {

private int bloqueo;

private IntBuffer buffer;

private Object mutex;

private int contador;

Escritor(int opcion, Object mutex, IntBuffer buf) {

this.bloqueo = opcion;

this.buffer = buf;

this.mutex = mutex;

this.contador = 0;

}

private void escribir() {

int i;

for (i = 0; i < 10000; i++) {

buffer.put(i, contador);

}

contador++;

}

public void run() {

while (true) {

if (this.bloqueo == 1) {

synchronized (this.mutex) {

escribir();

}

} else {

escribir();

}

}

}

}

class Lector extends Thread {

private int bloqueo;

private IntBuffer buffer;

private Object mutex;

Lector(int opcion, Object mutex, IntBuffer buf) {

this.bloqueo = opcion;

this.buffer = buf;

this.mutex = mutex;

}

private void comprobar() {

int i;

int elementoDistinto = 0;

for (i = 1; i < 10000; i++) {

if (buffer.get(0) != buffer.get(i)) {

System.out.println("Trhread lector: Error.Elementos de buffer distintos");

elementoDistinto = 1;

break;

}

}

if (elementoDistinto == 0) {

System.out.println("Trhread lector:Elementos de buffer iguales");

}

}

public void run() {

while (true) {

if (this.bloqueo == 1) {

synchronized (this.mutex) {

comprobar();

}

} else {

comprobar();

}

}

}

}

public class Check {

public static void main(String[] args) {

IntBuffer buf = IntBuffer.allocate(10000);

Object mutex = new Object();

// Modificar primer parámetro entre:

// 0 = No usar mutex

// 1 = Usar mutex

Lector l = new Lector(1, mutex, buf);

Escritor e = new Escritor(1, mutex, buf);

l.start();

e.start();

try {

l.join();

e.join();

} catch (InterruptedException ex) {

System.out.println("Hilo principal interrumpido.");

}

System.out.println("Proceso acabando.");

}

}

El programa principal inicializa un búfer de enteros, un objeto genérico, otro de la clase Lector y otro Escritor. A estos dos últimos, los inicia con el método *start* y como en el ejercicio anterior, espera en un bloque *try-catch* a que acaben su ejecución. Si falla escribe un mensaje. Cuando termina, escribe otro.

El hilo Lector recorre eternamente una y otra vez el búfer de enteros comprobando que todos los enteros sean igual que el primero. Escribe un mensaje diciendo si es o no es así.

El hilo Escritor recorre eternamente una y otra vez el búfer, modificando los valores de los enteros. Primero iguala todos a cero, luego a uno, después a dos, etc.

Cuando el primer valor del constructor es 1, la ejecución de ambos hilos depende del acceso sincronizado a la variable mutex, que es el objeto genérico antes mencionado. Su función consiste en ser sólo accesible por un hilo a la vez, debiendo los demás esperar a que el hilo que la usa, deje de hacerlo. Esto es posible con el siguiente código:

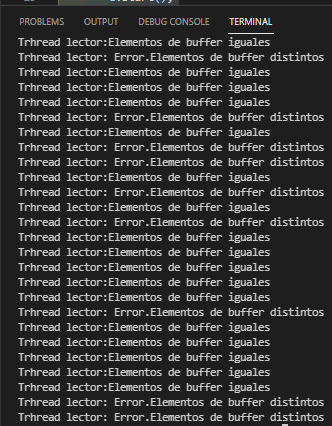
synchronized (this.mutex) { /\* Código a ejecutar \*/ }

De esta forma, si un hilo está ejecutando código dentro del *synchronized* de una misma variable (u objeto, se entiende), ninguno más podrá hacerlo antes de que este termine.

Es por esto que cuando el valor es 1, primero se ejecuta el lector y se va alternando la ejecución del lector y el escritor. Primero uno y luego el otro, tras finalizar el primero. Por esto, la ejecución del programa siempre da como resultado que los elementos del búfer son iguales:



Cuando el primer valor del constructor no es 1, no hay acceso sincronizado por ninguna variable. La alternancia de la ejecución de los hilos lector y escritor viene dada por el procesador. Es por ello que se alternen sin orden aparente y acceden al búfer sin esperar al otro hilo, obteniendo como resultado que el lector lea sin que el escritor hay acabado de modificar todos los valores del búfer:



# Relevos

class Testigo {

    private int siguiente;

    Testigo() {

        this.siguiente = 0;

    }

    synchronized public void next(int id) {

        this.siguiente = id;

        // Despierto a todos los threads

        // ya que no se sabe cuál de ellos

        // especificamente recibir el notify

        notifyAll();

    }

    synchronized public void check(int id) throws InterruptedException {

        while (siguiente != id) {

            // Me bloqueo hasta que sea mi turno

            wait();

        }

    }

}

class Corredor extends Thread {

    private static final int MAX\_DELAY = 1000;

    private int id;

    private Testigo testigo;

    Corredor(int id, Testigo t) {

        this.id = id;

        this.testigo = t;

    }

    public void run() {

        try {

            testigo.check(id);

            System.out.println("Soy el thread " + id + " corriendo . . .");

            Thread.sleep((int) Math.random() \* MAX\_DELAY);

            if (id != 4) {

                int receptor = id + 1;

                System.out.println("Terminé. Paso el testigo al hilo " + receptor);

                testigo.next(receptor);

            } else {

                System.out.println("Terminé!");

            }

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

}

public class Relevos {

    public static void main(String[] args) {

        Testigo testigo = new Testigo();

        Corredor corredores[] = new Corredor[4];

        for (int i = 0; i < 4; i++) {

            corredores[i] = new Corredor(i + 1, testigo);

            corredores[i].start();

        }

        System.out.println("Todos los hilos creados.");

        testigo.next(1);

        System.out.println("Doy la salida!");

        try {

            for (int i = 0; i < 4; i++) {

                corredores[i].join();

            }

        } catch (InterruptedException ex) {

            System.out.println("Hilo principal interrumpido.");

        }

        System.out.println("Todos los hilos terminaron.");

    }

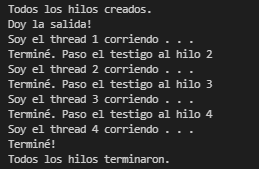
}

Este programa dispone de tres clases distintas: Relevos, Corredor y Testigo. La clase Relevos contiene el programa principal, que inicializa un objeto de clase Testigo y cuatro de Corredor. A estos últimos les da un id. Después ejecuta los hilos de los corredores, usa el método *next* del objeto Testigo y espera a que los cuatro hilos finalicen.

La clase Testigo es la más sencilla, pero la que encierra la clave del funcionamiento del programa. Su método *check* hace esperar al hijo indefinidamente si el id que le pasa como argumento no coincide con el atributo “siguiente” del objeto. Su método *notifyAll* despierta a todos los hilos que estaban esperando y cambia el valor de “siguiente” al id que se ha pasado como parámetro.

El método *run* de la clase corredor hace uso del *check* de testigo y si su id no coincide, espera, si no, duerme de 0 a 1 segundo y pasa el testigo al corredor siguiente al llamar al método *next* pasándole el id del siguiente corredor. A no ser que sea el cuarto, en cuyo caso termina simplemente.

De esta manera, al iniciar Relevos los corredores, todos duermen ya que el testigo guarda un 0 como atributo. No es hasta que usa el *next* donde los corredores despiertan por primera vez y comprueban si les toca a ellos. Entonces o duermen o corren y le pasan el testigo al siguiente despertando de nuevo a todos los hilos que no han acabado.



# SuperMarket

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;

    class Nodo {

        int cliente;

        Nodo sig;

    }

    Nodo raiz, fondo;

    public Caja() {

        raiz = null;

        fondo = null;

    }

    private boolean vacia() {

        if (raiz == null)

            return true;

        else

            return false;

    }

    synchronized public void esperar(int id\_cliente) throws InterruptedException {

        Nodo nuevo;

        nuevo = new Nodo();

        nuevo.cliente = id\_cliente;

        nuevo.sig = null;

        if (vacia()) {

            raiz = nuevo;

            fondo = nuevo;

        } else {

            fondo.sig = nuevo;

            fondo = nuevo;

        }

        while (raiz.cliente != id\_cliente) {

            // Me bloqueo hasta que sea mi turno

            wait();

        }

    }

    synchronized public void atender(int pago) throws InterruptedException {

        if (raiz == fondo) {

            raiz = null;

            fondo = null;

        } else {

            raiz = raiz.sig;

        }

        int tiempo\_atencion = new Random().nextInt(MAX\_TIME);

        Thread.sleep(tiempo\_atencion);

        Resultados.ganancias += pago;

        Resultados.clientes\_atendidos++;

        notify();

    }

    synchronized public void imprimir() {

        Nodo reco = raiz;

        while (reco != null) {

            System.out.print(reco.cliente + "-");

            reco = reco.sig;

        }

        System.out.println();

    }

}

class Cliente extends Thread {

    private static final int MAX\_DELAY = 2000;

    private static final int MAX\_COST = 100;

    private int id;

    private Caja caja;

    Cliente(int id, Caja caja) {

        this.id = id;

        this.caja = caja;

    }

    public void run() {

        try {

            System.out.println("Cliente " + id + " realizando compra");

            Thread.sleep(new Random().nextInt(MAX\_DELAY));

            long s = System.currentTimeMillis();

            caja.esperar(id);

            System.out.print("Cliente " + id + " en cola con ");

            caja.imprimir();

            caja.atender(new Random().nextInt(MAX\_COST));

            System.out.println("Cliente " + id + " atendido");

            long espera = System.currentTimeMillis() - s;

            Resultados.tiempo\_espera += espera;

            System.out.println("Cliente " + id + " saliendo después de esperar " + espera);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

}

public class SuperMarket {

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

        int N = Integer.parseInt(args[0]);

        Caja cajas[] = new Caja[N];

        for (int i = 0; i < N; i++) {

            cajas[i] = new Caja();

        }

        int M = Integer.parseInt(args[1]);

        Cliente clientes[] = new Cliente[M];

        for (int i = 0; i < M; i++) {

            // Seleccionamos ya en qué caja se situara

            j = new Random().nextInt(N);

            clientes[i] = new Cliente(i, cajas[j]);

            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));

    }

}

Lo primero que hay que aclarar es que hay un error en el código, ya que la variable j de método *main* de la clase SuperMarket no está declarada. Basta con poner la palabra reservada *int* antes del nombre de la variable para soluscionarlo.