Reto 4

Estudiante: Jose Javier Bailón Ortiz
Asignatura: Programacion de servicios y procesos
Convocatoria: 14/11/2023

Sumario

A – Estudio teórico de TIC,TAC,TOC

A.1 Planteamiento del problema

El objetivo es tener un programa que imprima en pantalla de manera indefinida y en este orden:

TIC

TAC

TOC

…

Debe hacerse habiendo dos hebras que intenten imprimir cada una de las palabras siendo agrupadas las hebras por la palabra que imprimen. Dos para TIC, dos para TAC y dos para TOC.

Las hebras deben competir para escribir su palabra no pudiendo haber turno estricto entre ellas.

A.2 Identificación de aspectos generales a controlar

El problema se puede abstraer a 2 aspectos generales que deben ser solucionados:

- Se debe conseguir que el orden de impresión sea el correcto: TIC, TAC, TOC de manera cíclica por tiempo indefinido. No TIC, TOC, TAC ni cualquier otra variante.
- Se debe evitar la impresión múltiple de la misma palabra de manera consecutiva dado que las hebras que imprimen la misma palabra compiten entre sí. No puede darse el caso de imprimir TIC, TIC por ejemplo.

A.3 Conseguir el orden correcto de palabras

Para conseguir el orden correcto no hay otra opción que crear un turno estricto entre hebras de diferentes grupos. Así en un momento dado le tocará solo a cualquier hebra del grupo TIC, luego del grupo TAC y luego del grupo TOC.

Para lograr este orden estricto se puede usar una variable común a todas las hebras que determine el grupo al que le toca:

Variable común a todas las hebras de todos los grupos		
int turnoGrupo = 0;		
Hebras Tic (grupo 0)	**Hebras Tac(grupo 1)**	**Hebras Toc(grupo 2)**

while (true){
while(turnoGrupo!=0);
print("Tic");
turnoGrupo=1;
}
	while (true){
	while(turnoGrupo!=1);
	print("Tic");
	turnoGrupo=2;
	}
		while (true){
		while(turnoGrupo!=2);
		print("Tic");
		turnoGrupo=0;
		}

O de manera genérica:

Variable común a todas las hebras de todos los grupos
int turnoGrupo = 0;
Hebras grupo G (G:indice de grupo, TG:total de grupos)

while (true){
//Espera ocupada
while(turnoGrupo!=G);
//Impresión
print(palabra de grupo G);
//cambio de grupo
turnoGrupo=((turnoGrupo+1)==TG) ? 0 : turnoGrupo+1;
//fin sección critica
}

Nota:La asignación de la última línea donde se incrementa turnoGrupo se puede extraer a una función común a todas las hebras.

Con esto se consigue un turno estricto entre las hebras de diferentes grupos.

A.4 Competición entre hebras del mismo grupo

Para preservar la competición entre las dos hebras del mismo grupo a la vez que se asegura una espera acotada se debe descartar cualquier sistema de exclusión mutua basada en un turno estricto entre hebras.

El algoritmo de Peterson quedaría descartado ya que supone una consecución de turnos estrictos.

El algoritmo de la panadería de Lamport supone también en la práctica una alternancia de turnos estrictos en este caso.

El algoritmo de Dekker pese a que también supone una alternancia de turnos estrictos entre hebras puede ser fácilmente modificado eliminando el control de turno para convertirlo en un algoritmo de competencia entre hebras a la vez que se conserva la espera acotada:

Variable común a todas las hebras del grupo

```
int[] banderas = new boolean[2];
```

Hebra 0

```
while (true){
        banderas[0]=true;
        while(banderas[1]){
                banderas[0]=false;
                esperarUnTiempo();
                banderas[0]=true;
        }
        print(“Tic”);
        banderas[0]=false
}
```

Hebra 1

```
while (true){
        banderas[1]=true;
        while(banderas[0]){
                banderas[1]=false;
                esperarUnTiempo();
                banderas[1]=true;
        }
        print(“Tic”);
        banderas[1]=false
}
```

O de manera genérica:

Variable común a todas las hebras del grupo

```
int[] banderas = new boolean[2];
```

Hebras grupo i (i:indice propio, j:indice de la otra hebra)

```
while (true){
        //subir bandera propia
        banderas[i]=true;
        //esperar mientras la otra hebra tiene la bandera subida y bajar y subir la propia a intervalos
        while(banderas[j]){
                banderas[i]=false;
                esperarUnTiempo();
                banderas[i]=true;
        }
        //seccion critica
        print(“Palabra del grupo”);
        //subir la bandera propia
        banderas[i]=false
}
```

A.5 Combinar la alternancia de grupos y la competición entre hebras del mismo grupo

Para conseguir combinar la alternancia entre grupos Tic,Tac,Toc y a la vez preservar la competición entre hebras del mismo grupo hay que combinar las estructuras de control de turnos de grupos y las de exclusión mutua de hebras del grupo:

Variable común a todas las hebras de todos los grupos

```
int turnoGrupo = 0;
```

Variable común a todas las hebras del mismo grupo

```
int[] banderas = new boolean[2];
```

Hebra i grupo G (i:indice propio, j:indice de la otra hebra del grupo , G:indice de grupo, TG:total de grupos)

```
while (true){
        //Esperar mientras no sea turno del grupo propio y el otro hilo del grupo propio tenga la bandera levantada
        //bajando y subiendo la bandera propia por intervalos durante la espera
        while(turnoGrupo!=G && banderas[j]){
                banderas[i]=false;
                esperarUnTiempo();
                banderas[i]=true;
        }
        //sección critica
        //Impresión
        print(palabra de grupo G);
        //cambio de grupo
        turnoGrupo=((turnoGrupo+1)==TG) ? 0 : turnoGrupo+1;
        //fin sección critica
        banderas[i]=false
}
```

A.6 Alternativa con semáforos en casos de más de 2 hebras

En un hipotético caso en el que cada grupo tuviese más de dos hebras se puede implementar una solución respetando una competencia total entre hebras usando un semáforo binario para cada grupo de hebras.

Variable común a todas las hebras de todos los grupos

```
int turnoGrupo = 0;
```

Variable común a todas las hebras del mismo grupo

```
Semáforo S = 1;
```

Hebras del grupo G (G:indice de grupo, TG:total de grupos)

```
while (true){
        //Esperar semaforo de grupo propio
        s.esperar();
        //Esperar a turno del grupo propio
        while(turnoGrupo!=G);
        //inicio sección critica
        //Impresión
        print(palabra de grupo G);
        //cambio de grupo
        turnoGrupo=((turnoGrupo+1)==TG) ? 0 : turnoGrupo+1;
        //fin sección critica
        //abrir el semaforo
        s.señalar();
}
```

B – Implementar en Java: Dekker, Peterson y Lamport

La implementación de estos algoritmos realizada en java se pueden encontrar en el código adjunto.

B.1 Dekker

Paquete: reto4b.dekker

Clase main: reto4b.dekker.MainDekker

Compilación: javac reto4b/dekker/MainDekker.java

Ejecución: java reto4b/dekker/MainDekker

Tras la ejecución se generan 2 hebras cada una sumando +1 o -1 a una variable compartida en cada ciclo. La exclusión mutua a la hora de acceder a la variable compartida se hace usando el algoritmo de Dekker el cual esta implementado en las hebras. Cada hebra da el mismo numero finito de ciclos con lo que el resultado final debe ser 0 si el algoritmo de Dekker ha funcionado.

En *reto4b.dekker.MainDekker.java,* en la sección de parámetros de creación, se puede configurar el número de ciclos que ejecutan las hebras.

Para ver más detalles sobre la implementación consultar la documentación del código.

B.2 Peterson

Paquete: reto4b.peterson

Clase main: reto4b.peterson.MainPeterson

Compilación: javac reto4b/peterson/MainPeterson.java

Ejecución: java reto4b/peterson/MainPeterson

Tras la ejecución se generan 2 hebras cada una sumando +1 o -1 a una variable compartida en cada ciclo. La exclusión mutua a la hora de acceder a la variable compartida se hace usando el algoritmo de Peterson el cual esta implementado en las hebras. Cada hebra da el mismo numero finito de ciclos con lo que el resultado final debe ser 0 si el algoritmo de Peterson ha funcionado.

En *reto4b.dekker.MainPeterson.java,* en la sección de parámetros de creación, se puede configurar el número de ciclos que ejecutan las hebras.

Para ver más detalles sobre la implementación consultar la documentación del código.

B.3 Lamport

He abordado la implementación de este algoritmo en 2 fases. He implementado el algoritmo presuponiendo que no se llega al límite que es capaz de soportar el tipo de dato donde se almacena el numero de turno para entrada en la sección crítica. Luego he completado el algoritmo agregando un control de vuelta de número y limitando el número máximo que se asigna. Adjunto ambas versiones. En *MainLamportCompleto* está el algoritmo al completo teniendo en cuenta la gestión de vueltas de número. En la versión *MainLamportSimplificado* se presupone que las hebras nunca adquirirán números por encima del limite máximo del tipo de dato del numero con lo que no tiene implementada la gestión de vueltas a la hora de coger el número.

B.3.1 Versión completa

Paquete: reto4b.lamport_completo

Clase main: reto4b.lamport_completo.MainLamportCompleto

Compilación: javac reto4b/lamport_completo/MainLamportCompleto.java

Ejecución: java reto4b/lamport_completo/MainLamportCompleto

Tras la ejecución se generan múltiples hebras cada una sumando +1 o -1 a una variable compartida en cada ciclo. La exclusión mutua a la hora de acceder a la variable compartida se hace usando el algoritmo de Lamport el cual esta implementado en las hebras. Esta versión contempla un numero máximo finito y la gestión de vueltas para solventar ese limite. Cada hebra da el mismo numero finito de ciclos y en cada ciclo incrementa la variable compartida. Como hay tantas hebras +1 como -1 el resultado debe ser 0 si el algoritmo de Lamport ha funcionado.

Implementación de la gestión de vueltas de número

Para la implementación de la gestión de vueltas cada número que coge una hebra para entrar a la sección crítica está compuesto por un array[2] siendo el primer índice el número propiamente dicho y el segundo indice el número de vuelta. La vuelta puede ser vuelta 0 o vuelta 1. En todo momento se mantiene un seguimiento de cual es la vuelta mayor(mas nueva) y que por tanto tiene menos prioridad. Cuando el numero asignado excede el límite de numero determinado se le asigna el número 0 y la vuelta opuesta a la actual $0 \rightarrow 1$ o $1 \rightarrow 0$ y se actualiza en el seguimiento cuál es la vuelta mas nueva.

En *reto4b.lamport_completo.MainLamportCompleto.java,* en la sección de parámetros de creación, se puede configurar el número de ciclos que ejecutan las hebras así como la cantidad de hebras.

Para ver más detalles sobre la implementación consultar la documentación del código.

B.3.2 Versión simplificada

Paquete: reto4b.lamport_simplificado

Clase main: reto4b.lamport_simplificado.MainLamportSimplificado

Compilación: javac reto4b/lamport_simplificado/MainLamportSimplificado.java

Ejecución: java reto4b/lamport_simplificado/MainLamportSimplificado

Tras la ejecución se generan múltiples hebras cada una sumando +1 o -1 a una variable compartida en cada ciclo. La exclusión mutua a la hora de acceder a la variable compartida se hace usando el algoritmo de Lamport el cual esta implementado en las hebras. Esta versión presupone que nunca se llegará al limite máximo del tipo de dato donde se almacena el número y no tiene control de vuelta de número. Cada hebra da el mismo numero finito de ciclos y en cada ciclo incrementa la variable compartida. Como hay tantas hebras +1 como -1 el resultado debe ser 0 si el algoritmo de Lamport ha funcionado.

En *reto4b.lamport_simplificado.MainLamportSimplificado.java*, en la sección de parámetros de creación, se puede configurar el número de ciclos que ejecutan las hebras así como la cantidad de hebras.

Para ver más detalles sobre la implementación consultar la documentación del código.

C – Simular una base de datos

C.1 Planteamiento del problema

Simular una base de datos. Debe tener varias hebras que intenten leer de ella y varias hebras que intenten escribir en ella. Varias hebras pueden acceder de manera simultanea si es para leer. Cuando se esté escribiendo solo puede acceder la hebra que esta escribiendo. No podrá acceder ninguna otra para leer ni para escribir durante ese tiempo.

Comprobar que se preserva la coherencia de datos.

C.2 Desarrollo del proceso de resolución

Para resolver el problema he decidido hacerlo sobre una base de datos simulada en archivo.

Se trata de un archivo en disco con un numero de tuplas consistentes cada una en un número y al inicio puestas a 0. Estas tuplas serán accedidas por varias hebras que vayan leyendo unas y escribiendo otras haciéndolo cada hebra de manera cíclica un número finito de veces. Las hebras que escriban lo harán una cantidad de veces previamente determinada y cada vez incrementarán en un valor conocido una tupla elegida aleatoriamente. Así al terminar las hebras se puede comprobar la coherencia de los datos guardados en la base de datos sumando los valores de todas las tuplas y comparándolo con el resultado esperado.

C.3 Técnicas y algoritmos usados

Para implementar el control de acceso de lectura y escritura, tal y como define el problema, opté por hacerlo usando el algoritmo lector/escritor de W. Stallins visto en clase.

Para hacerlo es necesario previamente implementar semáforos los cuales he realizado usando el algoritmo de la panadería de Lamport para lograr la exclusión mutua necesaria dentro de ellos. Los semáforos creados en primer momento fueron semáforos débiles.

Una vez tuve el sistema funcionando me enfrenté al problema de la prioridad de lectura que implementa el algoritmo de W. Stallins. Las hebras que leían bloqueaban permanentemente la escritura hasta que prácticamente todas las lecturas habían terminado. Eso era debido a la continuidad con las que estas intentaban la lectura y su prioridad de lectura. Optar por la prioridad de escritura no era un camino que pareciera que fuese a solucionar el problema así que decidí buscar una solución donde no hubiera prioridad de lectura o escritura preservando la lectura simultanea múltiple y la escritura de una sola hebra sin acceso del resto de hebras durante la escritura.

Buscando una solución para el problema de inanición encontré una pequeña modificación del algoritmo de W. Stallins que podía eliminar la prioridad de lectura. Se trata de incorporar un semáforo más al algoritmo cuya espera y señalado rodeen la espera de las escrituras y las lecturas como se ve a continuación.

Solución para eliminar la prioridad de lectura:

Semáforos y variables comunes

```
Semaforo escritores(1)
Semaforo lectores(1)
Semaforo cola(1)
numeroLectores=0
```

Escritor original

```
while (true){
        escritores.esperar();
                //seccion critica escribir
        escritores.senalar();
        }
```

Escritor modificado

```
while (true){
        cola.esperar();
        escritores.esperar();
        cola.senalar();
                //seccion critica escribir
        escritores.senalar();
}
```

Lector original

```
while (true){
                lectores.esperar();
                numeroLectores++;
                if (numeroLectores == 1)
                        escritores.esperar();
                lectores.senalar();
                //lectura
                lectores.esperar();
                numeroLectores--;
                if (numeroLectores == 0)
                        escritores.senalar();
                lectores.senalar();

        }
```

Lector modificado

```
while (true){
                cola.esperar();
                lectores.esperar();
                numeroLectores++;
                if (numeroLectores == 1)
                        escritores.esperar();
                cola.senalar();
                lectores.senalar();
                //lectura
                lectores.esperar();
                numeroLectores--;
                if (numeroLectores == 0)
                        escritores.senalar();
                lectores.senalar();
 }
```

Esta solución preserva la lectura simultánea y la exclusión del resto de hebras cuando una esté escribiendo y además añade una política FIFO que hace que si varias hebras de lectura llegan juntas podrán leer a la vez pero cuando llegue una de escritura entrará en su turno terminando así con la prioridad de lectura.

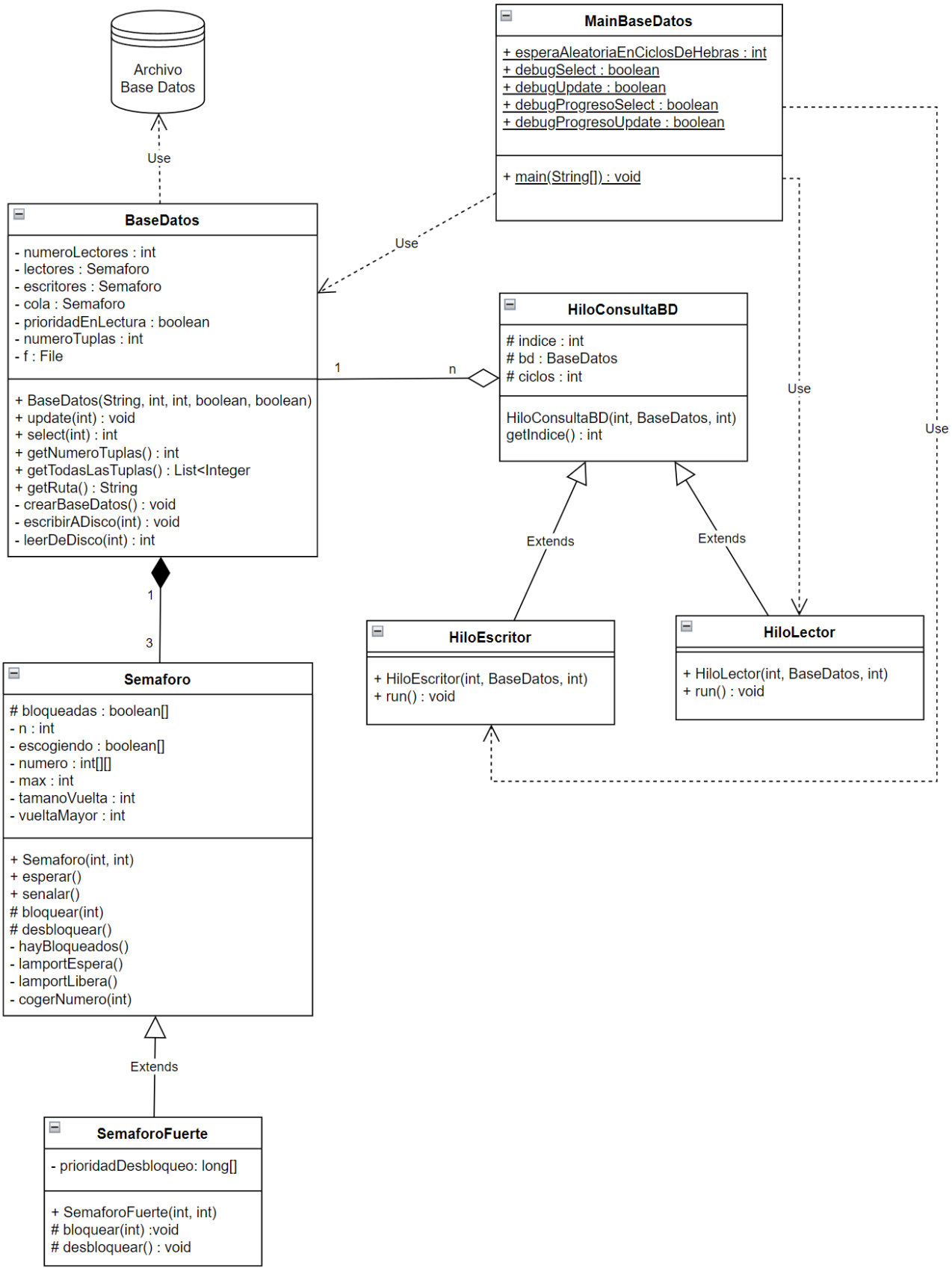
Para que sea una política FIFO los semáforos deben ser fuertes pero en este punto yo los tenia implementados como débiles así que implementé una versión del semáforo fuerte modificando la selección de qué hebra se liberaba tras señalarse el semáforo.

Intenté hacerlo con una cola agregando al final al esperar y sacando el primer elemento al señalar, pero no funcionaba y la coherencia del semáforo se perdía. La solución ha sido crear un registro de en qué tiempo había entrado cada hebra que esperaba el semáforo. A la hora de liberar una hebra se libera la que tenga un tiempo de registro de entrada menor.

No obstante en la configuración de la ejecución se puede elegir el tipo de semáforo a usar y se puede activar la prioridad de lectura para probar los comportamientos descritos.

C.3 Implementación final

C.3.1 Diagrama de clases



C.3.2 Notas sobre la implementación

La implementación final queda de la siguiente manera.

Al inicio del programa la clase *MainBaseDatos* genera un objeto de la clase *BaseDatos* pasándole los datos de configuración inicial adecuados. Al inicio, *BaseDatos* genera un nuevo archivo según lo especificado en la configuración. Por defecto *"basedatos.dat"* en la carpeta de ejecución. Lo rellena de varios enteros 0 y **si el archivo existe lo elimina y recrea**. Tras eso *MainBaseDatos* crea varias hebras de las clases *HiloEscritor* e *HiloLector* pasándoles una referencia del objeto *BaseDatos* y las pone a correr.

Estas hebras contienen en su *run()* un bucle con el que mandan a la *BaseDatos* mensajes requiriendo leer o escribir. En cada escritura una de las tuplas del archivo es seleccionada aleatoriamente, leída, su valor incrementado en 1 y escrita nuevamente.

BaseDatos se encarga de gestionar el comportamiento de acceso al archivo según la implementación descrita en secciones anteriores haciendo uso de semáforos implementados en las clases *Semaforo* y *SemaforoFuerte*. *Semaforo* es uno de tipo débil y SemaforoFuerte es de tipo fuerte. Están implementados usando el algoritmo de la panadería de Lamport.

Cuando *MainBaseDatos* ha generado todo lo anterior, espera a que las hebras terminen. Tras eso presenta un informe sobre el resultado y las condiciones de ejecución.

C.3.3 Ejecutar el código

Paquete: reto4c

Clase main: reto4c.MainBaseDatos

Compilación: javac reto4c/MainBaseDatos.java

Ejecución: java reto4c/MainBaseDatos

Al ejecutarlo se verán las acciones que se van realizando sobre la base de datos según se tenga definido en la configuración de ejecución.

Al terminar presenta un informe mostrando:

- Ruta del archivo de base de datos
- Listado de tuplas de la base de datos
- Suma de valores de tuplas esperado
- Suma real de los valores de las tuplas
- Tiempo consumido en realizar la operación

- Cantidad de hebras de lectura
- Cantidad de hebras de escritura
- Número de consultas realizadas por cada hebra

C.3.1 Configurar la ejecución

En la clase *reto4c.MainBaseDatos* se pueden configurar los siguientes parámetros de ejecución:

- **esperaAleatoriaEnCiclosDeHebras:** Milisegundos máximos que cada hebra espera entre cada ciclo. Se puede dejar a 0 o aumentar si se quiere aleatoriedad en la frecuencia en la que cada hilo intenta acceder a la base de datos.
- **debugSelect:** Mostrar los mensajes de lectura de base de datos.
- **debugUpdate:** Mostrar los mensajes de escritura de base de datos.
- **debugProgresoSelect:** Mostrar mensajes de progreso de ciclos de los hilos de lectura.
- **debugProgresoUpdate:** Mostrar mensajes de progreso de ciclos de los hilos de escritura.
- **rutaArchivo:** Ruta al archivo de la base de datos. **Este archivo se borrará al inicio de cada ejecución.**
- **prioridadEnLectura:** True establece que hay prioridad en la lectura. False que no la hay.
- **semaforosFuertes:** True para usar semáforos fuertes, False para usar semáforos débiles.
- **hebrasLectura:** Cantidad de hebras que leerán.
- **hebrasEscritura:** Cantidad de hebras que escribirán.
- **ciclos:** Cantidad de lecturas o escrituras que realizara cada hebra.
- **numeroTuplas:** Cantidad de tuplas en la base de datos
- **startAleatorioDeHebras:** True iniciar las hebras en un orden aleatorio. False iniciarlas en el orden de creación(primero se crean las hebras lectoras y luego las escritoras).

Valores por defecto:

- **esperaAleatoriaEnCiclosDeHebras:** 0
- **debugSelect:** True
- **debugUpdate:** True
- **debugProgresoSelect:** True
- **debugProgresoUpdate:** True
- **rutaArchivo:** basedatos.dat
- **prioridadEnLectura:** False

- **semaforosFuertes:** True
- **hebrasLectura:** 30
- **hebrasEscritura:** 30
- **ciclos:** 50
- **numeroTuplas:** 30
- **startAleatorioDeHebras:** True