	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	TISTA GARCÍA EDGAR
Asignatura:	PROGRAMACIÓN ORIENTADA A OBJETOS
Grupo:	3
No de Práctica(s):	PRÁCTICA 12 - HILOS
Integrante(s):	CARRANZA OCHOA JOSE DAVID CARRILLO CERVANTES IVETTE ALEJANDRA
No. de Equipo de cómputo empleado:	TRABAJO EN CASA
No. de Lista o Brigada:	07, 08
Semestre:	2022 - 1
Fecha de entrega:	11 DICIEMBRE 2021
Observaciones:	

CALIFICACIÓN: _____

PRÁCTICA #12 HILOS

Objetivo. Implementar el concepto de multiarea utilizando hilos en un lenguaje orientado a objetos.

ACTIVIDADES PARA EL DESARROLLO DE LA PRÁCTICA

Ejemplos de la guía

Java proporciona soporte para hilos através de una interfaz y un conjunto de clases, dichas interfaz y clases que proporcionan algunas funcionalidades sobre hilos son las siguientes:

- Thread
- Runnable (interfaz)
- ThreadDeath
- ThreadGroup
- Object

Clase Thread

Este programa tiene como función principal, mostrar la implementación de hilos heredando de la clase Thread, cuenta con un método constructor en el cual se manda a llamar al método constructor de la clase padre en el cual se inicializa el nombre del hilo; cuenta con el método sobrescrito run, el cual tiene como objetivo imprimir el número de iteraciones de cada hilo conforme se vaya llegan, así como indicar en qué momento se termina la ejecución de estos. Finalmente, se tiene el método main, el cual inicializa dos hilos con sus respectivos nombres y los inicia mediante el método “start()”, cabe recalcar que esta manda a llamar al método run y realiza las acciones correspondientes que tiene este. La salida es la siguiente:

```
run:
Termina el hilo principal
Iteración: 1 de Segundo hilo
Iteración: 1 de Primer hilo
Iteración: 2 de Segundo hilo
Iteración: 2 de Primer hilo
Iteración: 3 de Segundo hilo
Iteración: 3 de Primer hilo
Iteración: 4 de Segundo hilo
Iteración: 4 de Primer hilo
Iteración: 5 de Segundo hilo
Iteración: 5 de Primer hilo
Termina el Primer hilo
Termina el Segundo hilo
BUILD SUCCESSFUL (total time: 0 seconds)
```

La salida de este programa, más específicamente al realizar el bucle for, los hilos no llevan un orden en específico para ejecutarse debido a la condición carrera.

Interfaz Runnable

Aquí vemos la segunda forma de trabajar con hilos desde Java, esta es la interfaz “Runnable” quien nos permite implementar dicha interfaz para evitar heredar desde la clase “Thread” como anteriormente se explicó.

La ventaja de este método es justamente la implementación en lugar de utilizar la herencia; esto es porque en alguna parte de nuestro programa podríamos requerir la herencia desde otra clase y estaríamos limitados si empleamos “Thread” generando una nueva alternativa al momento de trabajar con hilos.

El código proporcionado implementa tal interfaz seguido del nombre de la clase, dentro de esta clase, se tendrá que llamar a la creación de un nuevo hilo para trabajar paralelamente.

Por medio de “run”, se establece la región que contendrá las acciones simultaneas dentro del programa, aquí, se tendrá la ejecución simultanea de 2 hilos previamente inicializados desde “main”.

Gracias a “start” se puede abrir un nuevo hilo para que opere, observando que un bucle “for” es quien recibe las iteraciones simultaneas de nuestra ejecución para ambos hilos.

```
run:
Termina el hilo principal
Iteración: 1 de Segundo hilo
Iteración: 1 de Primer hilo
Iteración: 2 de Segundo hilo
Iteración: 2 de Primer hilo
Iteración: 3 de Segundo hilo
Iteración: 3 de Primer hilo
Iteración: 4 de Segundo hilo
Iteración: 4 de Primer hilo
Iteración: 5 de Segundo hilo
Iteración: 5 de Primer hilo
Termina el Primer hilo
Termina el Segundo hilo
BUILD SUCCESSFUL (total time: 0 seconds)
```

Cabe mencionar que dentro de las iteraciones no se encuentra contenido alguna condición limitante para las iteraciones de cada hilo, por lo que se puede apreciar que las salidas no tendrán un orden “correcto” al momento de ser ejecutadas; a esto se le denomina condición de carrera.

Clase ThreadGroup

La clase ThreadGroup tiene como finalidad manejar un grupo de hilos de manera simplificada, es decir, en conjunto para controlar de modo eficiente la ejecución de dicha serie de hilos.

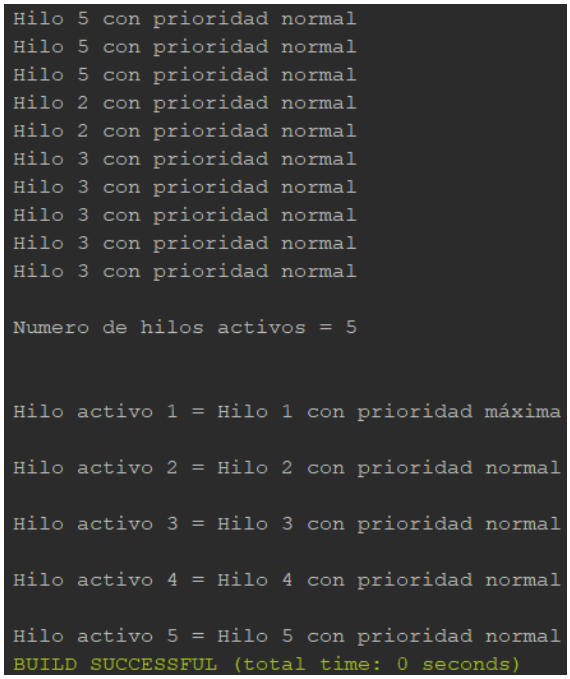
Para observar esta implementación se realizó un programa el cual tiene como función asignarle una prioridad a cada uno de los hilos creados, permitiendo tener una jerarquía anidada de hilos, en la clase principal; para ello, se creó un método constructor que a su vez llama al método padre pasando como parámetros el grupo de hilos en el que el hilo se quiere agregar y el nombre de dicho grupo; también se creó un método sobrescrito run, el cual imprime el nombre del hilo en el que se encuentra 5 veces.

En el método main se encuentra una instancia de la clase ThreadGroup, para generar un grupo de hilos en específico; después, se encuentran 5 instancias de hilos, indicando con el método constructor de la clase el grupo en el que se quiere agregar y el nombre que tendrá cada uno de los hilos.

Posteriormente, se le asigna una prioridad especifica al primer hilo creado, mediante el método setPriority() pasando como parámetro la máxima prioridad, mientras que a los demás que restan se les asigna una prioridad normal para luego imprimir la prioridad que tiene cada hilo e iniciar cada uno de ellos (cabe recalcar que al iniciarlos se manda a llamar a la función run, la cual en este caso imprime el nombre de cada hilo mediante un bucle for).

Finalmente, al terminar la ejecución de cada hilo, se manda a llamar al método listarHilos pasándole como parámetro el grupo de hilos, este método tiene como objetivo obtener el número de hilos que se encuentran activos en el grupo de este mediante el método “activeCount()” y después enumerar estos, para que después de esto se realice un ciclo for que se repetirá las veces correspondientes a los hilos activos del grupo, indicando en cada iteración el hilo en el que se encuentra.

```
run:
Prioridad del grupo = 5
Prioridad del Thread = 10
Prioridad del Thread = 5
Prioridad del Thread = 5
Prioridad del Thread = 5
Prioridad del Thread = 5
Hilo 1 con prioridad máxima
Hilo 2 con prioridad normal
Hilo 2 con prioridad normal
Hilo 2 con prioridad normal
Hilo 4 con prioridad normal
Hilo 4 con prioridad normal
Hilo 4 con prioridad normal
Hilo 4 con prioridad normal
Hilo 4 con prioridad normal
Hilo 1 con prioridad máxima
Hilo 1 con prioridad máxima
Hilo 1 con prioridad máxima
Hilo 1 con prioridad máxima
Hilo 5 con prioridad normal
Hilo 5 con prioridad normal
```



Métodos o bloques sincronizados

La sincronización aparece en Java por medio de la palabra reservada “synchronized”, al emplear dicha instrucción, se tiene por enterado que trabajaremos con el concepto de monitor, quien bloquea ciertas instrucciones para que los demás hilos no puedan sobrescribir sobre una variable o bien no puedan ejecutarse mientras un hilo esté en ejecución.

La sincronización entre hilos es de vital importancia al evitar que la información se pierda mientras se está ejecutando en paralelo.

Comenzando con nuestro método “main” se logra apreciar que trabajamos con 4 nuevos objetos del tipo “Cuenta”, Aquí se abren a su vez 4 hilos para que efectúen las operaciones contenidas en el método “run”.

Dentro del método anterior, existe una validación por si se tratara de un depósito simulado (comparando la cadena pasada como parámetro). Si el caso corresponde a un depósito, se ingresan cierta cantidad (100), caso contrario se extrae el dinero de la cuenta.

Dentro de la simulación, se efectúa primeramente la extracción de dinero, pero como el saldo es 0, se mantendrá a la espera a que se efectúa un depósito donde este hilo terminará su ejecución tras el cometido, por medio de la sincronización de los métodos “extraerDinero” y “depositarDinero” se determina que solo uno a la vez se podrá ejecutar respecto a su hilo actual.

Tras realizar un depósito, la variable “saldo” se incrementa y permite que solo un método acceda sobre esta al estar comunicados desde un monitor, por tanto, la salida que se espera dependerá de cuál hilo haya sido ingresado primero al recurso del saldo.

```
run:
Termina el hilo principal
Se depositaron 100 pesos
Se depositaron 100 pesos
Terminar el Deposito 1
Terminar el Deposito 2
Acceso 2 espera deposito
Saldo = 0
Acceso 1 espera deposito
Saldo = 0
Acceso 1 estrajo 50 pesos.
Saldo restante = 100
Terminar el Acceso 1
Acceso 2 estrajo 50 pesos.
Saldo restante = 150
Terminar el Acceso 2
BUILD SUCCESSFUL (total time: 5 seconds)
```

La ejecución resulta de 2 partes, la parte de los depósitos que se generan tras la espera de los accesos y finalmente, la extracción de dinero en la variable “saldo”.

Ejercicio 1. Creación de Hilos

Para este primer ejercicio se crearon dos clases: una clase principal con el método main y otra, “MiHilo2” para los métodos correspondientes para crear y ejecutar un hilo.

En la clase principal se encuentran 3 hilos creados por la función crearyComenzar de la clase MiHilo2; mientras que en esta última clase se almacenarán los métodos correspondientes para crear e iniciar un hilo, esta clase implementa a la interfaz Runnable y cuenta con los siguientes métodos: método constructor que instancia el hilo con su nombre correspondiente (se crea una instancia de Thread); se creó el método crearYcomenzar, el cual tiene la función de crear el hilo con el nombre que se le pasa como parámetro y posteriormente inicia el hilo mediante el método start(). Finalmente, se tiene el método sobrescrito run(), en el cual se indica el nombre del hilo el que se inicia y mediante un ciclo for, se manda a llamar al método sleep() el cual suspenderá el hilo mediante la cantidad de 1 segundo para posteriormente imprimir el número de iteración en la que se encuentra. Cabe destacar que los hilos creados no tendrán un orden en específico debido a que se entra en una condición de carrera, pero al momento de llegar al método sleep(), el hilo que se está ejecutando se debe de suspender por la cantidad de tiempo establecida, en este caso, con “sleep” de 1000 se establece que las iteraciones estarán ejecutándose cada segundo para cada integración de los hilos (en este caso son 3), quienes por cada recuento se tendrán que esperar la cantidad descrita.

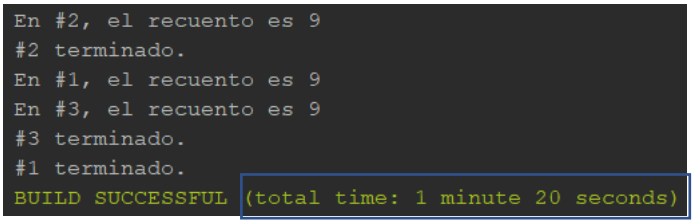
```
run:
Inicio del programa (hilo) principal
Hilo principal finalizado
#3 iniciando.
#1 iniciando.
#2 iniciando.
```

```
run:
Inicio del programa (hilo) principal
Hilo principal finalizado
#2 iniciando.
#3 iniciando.
#1 iniciando.
En #2, el recuento es 0
En #3, el recuento es 0
En #1, el recuento es 0
En #2, el recuento es 1
En #1, el recuento es 1
En #3, el recuento es 1
En #2, el recuento es 2
En #1, el recuento es 2
En #3, el recuento es 2
En #2, el recuento es 3
En #3, el recuento es 3
En #1, el recuento es 3
En #3, el recuento es 4
En #1, el recuento es 4
En #2, el recuento es 4
En #1, el recuento es 5
En #2, el recuento es 5
En #3, el recuento es 5
En #2, el recuento es 6
En #3, el recuento es 6
En #1, el recuento es 6
En #3, el recuento es 7
En #1, el recuento es 7
En #2, el recuento es 7
En #1, el recuento es 8
En #2, el recuento es 8
En #3, el recuento es 8
En #2, el recuento es 9
#2 terminado.
En #3, el recuento es 9
En #1, el recuento es 9
#3 terminado.
#1 terminado.
BUILD SUCCESSFUL (total time: 10 seconds)
```

Para un valor de 500, las llamadas al bucle serán mucho más rápida al tomar medio segundo para cada llamada a su siguiente instrucción, parece que el programa se ejecuta de corrido y se ha omitido el tiempo manual pero no es así.

```
En #2, el recuento es 8
En #1, el recuento es 8
En #3, el recuento es 9
#3 terminado.
En #2, el recuento es 9
#2 terminado.
En #1, el recuento es 9
#1 terminado.
BUILD SUCCESSFUL (total time: 5 seconds)
```

Por su parte, 8000 segundos equivalen a 8 segundos de espera por cada nueva llamada del bucle, las 10 iteraciones en total tendrán un total de 640 segundos para cada hilo (recordando que se ejecutan en simultaneo).



Al observar el constructor de Thread, notamos que maneja con 2 parámetros, el primero data de "this" quien hace referencia a la instancia actual, recordando que al trabajar sobre una interfaz es necesario tal constructor para trabajar paralelamente. El segundo parámetro ingresado constituye el nombre del hilo que se ejecuta, estos vienen dados para el ejemplo por #1, #2, #3.

Haciendo mención del constructor, contamos con otras formas de crear nuevas instancias de este mismo, tales son los casos de:

Thread().
Este constructor tiene como finalidad asignar un nuevo objeto tipo Thread(), es decir, un nuevo hilo.

Thread(Runnable tarea).
Dentro del constructor se crea un nuevo objeto a raíz de un objeto proveniente de la interfaz Runnable, este es usado para impelemtar la variante de la concurrencia en Java.

Thread(Runnable tarea, String name).
Este constructor crea un objeto proveniente de la interfaz Runnable, dándoles un nombre en específico.

Thread(String nombre)
Inicializa un nuevo objeto Thread con el nombre pasado como cadena dentro del argumento

Thread(ThreadGroup grupo, Runnable tarea)
Tiene como proposito agregar un objeto proveniente de la interfaz Runnable, a un grupo de Hilos en específico.

Thread(ThreadGroup grupo ,Runnable tarea, String nombre)
Inicializa un nuevo objeto quien proviene de la interfaz Runnable, el objeto será ingresado a un grupo de hilos con quienes se compartirán acciones en común; la cadena como argumento definirá el nombre del mismo

Thread(ThreadGroup grupo, Runnable task, String name, long stackSize)
Instancia un objeto que proviene de la interfaz Runnable, el cual se le asignará a un grupo de hilos (subprocesos) y tenga un tamaño de pila especificado.

Thread(ThreadGroup grupo, String nombre)
Genera un nuevo hilo y lo asigna a un grupo de hilos; la cadena corresponde al nombre del hilo a crear.

Como se observa, existen diferentes constructores definidos en la API de Java, donde se nos presenta el cómo podemos inicializar los valores de un nuevo subproceso para posteriormente ejecutarlo.

Ejercicio 2. Prioridad en Hilos

Dentro de este ejercicio podemos observar un ejemplo cuando existe una serie de prioridades dentro de los hilos.

Primeramente, contamos con 5 implementaciones de hilos que son inicializados como hilos desde su constructor; tras esto, aparece la descripción explicita sobre la prioridad de 2 hilos en específico, tanto del primero como del segundo hilo, teniendo la prioridad más alta y la más baja respectivamente.

Tras esto, aparece nuestro método “join” quien define de forma jerárquica cuál será el hilo que tome por completo los recursos hasta que este finalice.

Cuando cada hilo termine, entonces se mostrarán la cantidad de operaciones que cada hilo realizó, tendiendo como premisa que el de mayor importancia será el que tenga un mayor valor de iteraciones realizadas.

Todo el proceso ocurre dentro de la clase “PrioridadHilos” donde se ve implementado la interfaz ya conocida para trabajar con la concurrencia de Java; es aquí donde aparece el método “run” quien define la región paralelizable en Java.

Tal método generará una serie de iteraciones que dependerán del nombre del hilo actual con su nombre del hilo en ejecución. Este ciclo tendrá 10,000,000 de iteraciones máximas posibles hasta que algún hilo finalice su ejecución y termine el proceso de los demás.

```
run:
Prioridad Normal #3 iniciando.
Prioridad Normal #1 iniciando.
Prioridad Alta iniciando.
Prioridad Normal #2 iniciando.
En Prioridad Normal #1
En Prioridad Normal #3
En Prioridad Alta
En Prioridad Normal #3
Prioridad Baja iniciando.
En Prioridad Normal #1
En Prioridad Normal #2
En Prioridad Normal #1
En Prioridad Normal #1
En Prioridad Baja
En Prioridad Normal #3
En Prioridad Alta
En Prioridad Normal #3
En Prioridad Baja
En Prioridad Normal #1
En Prioridad Normal #2
En Prioridad Normal #1
En Prioridad Baja
```

…

```
Prioridad Normal #2 terminado.

Prioridad Normal #3 terminado.

Prioridad Normal #1 terminado.

Hilo de alta prioridad contó hasta      100000000
Hilo de baja prioridad contó hasta      67668509
Hilo de normal prioridad #1 contó hasta 84992439
Hilo de normal prioridad #2 contó hasta 92814290
Hilo de normal prioridad #3 contó hasta 82928101
BUILD SUCCESSFUL (total time: 6 seconds)
```

Observando que justamente, nuestro hilo principal trabaja con una mayor prioridad para el hilo establecido a un principio, mientras que el hilo 2 que se definió por medio de “NORM_PRIORITY-4”.

Ahora nos puede surgir la interrogante para el caso donde no exista “join” y el programa se ejecute.

Para esta situación, se realizó un análisis teórico y un análisis práctico. La parte teórica nos indica que todos los hilos al no contar con un limitador de la ejecución total de un hilo, cada quien tendrá una parte del tiempo del procesador donde la simultaneidad estará presente para todos los casos; y que al terminar algún hilo sus iteraciones que son realizadas en conjunto, el bucle finalizara cuando la iteración llegue a 100,000,000.

Esto se ve representado en una condición de carrera, donde debemos recordar que los accesos no son secuenciales, por lo que nuestra salida en pantalla corresponde justo al momento en que se ha finalizado con un subproceso.

```
En Prioridad Baja

Hilo de alta prioridad contó hasta      0
En Prioridad Baja
En Prioridad Normal #3
En Prioridad Alta
En Prioridad Baja
En Prioridad Normal #2
En Prioridad Normal #1
En Prioridad Normal #2
En Prioridad Baja
En Prioridad Alta
En Prioridad Normal #3
En Prioridad Alta
En Prioridad Baja
En Prioridad Normal #2
En Prioridad Normal #1
En Prioridad Normal #2
En Prioridad Baja
En Prioridad Normal #3
Hilo de baja prioridad contó hasta      184
En Prioridad Alta
```

```
En Prioridad Normal #3
Hilo de baja prioridad contó hasta      184
En Prioridad Alta
En Prioridad Normal #3
En Prioridad Baja
En Prioridad Normal #2
En Prioridad Normal #1
Hilo de normal prioridad #1 contó hasta 176
En Prioridad Normal #2
En Prioridad Baja
En Prioridad Normal #3
En Prioridad Alta
En Prioridad Normal #3
En Prioridad Baja
En Prioridad Normal #2
En Prioridad Normal #1
En Prioridad Normal #2
En Prioridad Baja
Hilo de normal prioridad #2 contó hasta 199
```

Añadiendo que “join” es fundamental para otorgar una decisión al hilo que es llamado desde este mecanismo para que actúe por su cuenta hasta que finalice todas sus operaciones, por lo que, sin su aparición para este caso, el programa pierde la secuencia que se esperaría, así como su finalización de cada hilo.

Al ejecutar la clase principal y al tener hilos con dos prioridades establecidas, en este caso el “Hilo de alta prioridad” con la prioridad máxima y el hilo “Hilo de baja prioridad” con la prioridad más baja, se observa que el hilo con de alta prioridad realizó todas las iteraciones establecidas desde un principio, mientras que el hilo con la prioridad más baja fue el que realizó el menor número de iteraciones de todos los hilos. Esto se debe a que el hilo con prioridad más alta, al realizar las iteraciones correspondientes (100,000,000) y al salir del bucle “while”, el valor inicializado booleano inicializado previamente como una variable de clase con false y condicionante del bucle, cambia su valor y convierte se convierte a true, indicando que el ciclo “while” de todos los hilos terminará.

La salida del programa (al momento de imprimir los contadores de cada hilo), es la siguiente, en donde podemos observar que el hilo de prioridad más alta realizó todas las operaciones correspondientes, mientras que las demás no.

```
Hilo de alta prioridad contó hasta      100000000
Hilo de baja prioridad contó hasta      54147734
Hilo de normal prioridad #1 contó hasta 82634006
Hilo de normal prioridad #2 contó hasta 63904773
Hilo de normal prioridad #3 contó hasta 96927608
BUILD SUCCESSFUL (total time: 6 seconds)
```

Se volvió a ejecutar el programa; sin embargo, esta vez se ejecutó sin asignar prioridades en los hilos, por lo cual cualquier hilo podría tener la prioridad máxima y cualquiera la prioridad mínima, en el caso de la siguiente salida podemos observar que la prioridad máxima la tiene el hilo de normal prioridad #1, mientras que el hilo que tiene una prioridad mínima es el hilo de normal prioridad #2.

```
Hilo de alta prioridad contó hasta      96146079
Hilo de baja prioridad contó hasta      76835575
Hilo de normal prioridad #1 contó hasta 100000000
Hilo de normal prioridad #2 contó hasta 48390407
Hilo de normal prioridad #3 contó hasta 50245429
BUILD SUCCESSFUL (total time: 6 seconds)
```

Ejercicio 3. Comunicación entre Hilos

Este último programa tiene como función realizar la suma de un arreglo en específico para dos hilos en partículas, para realizar este programa se realizaron 3 clases: clase principal, clase MiHilo y clase Suma.

La clase principal tiene como objetivo inicializar el arreglo que se quiere sumar con números del 1 al 13 y posteriormente crear dos hilos llamando a la función creaEInicia de la clase MiHilo, la cual se describirá posteriormente, pasándole como parámetro el nombre de este y el arreglo que sumará este hilo. (en este caso, ambos hilos harán la suma de este arreglo), para después con el método join() poder indicar que el primer hilo que se ejecute debe de terminar todas sus tareas para que el otro pueda empeda empezar.

La clase MiHilo cuenta con 4 atributos, un hijo, un objeto tipo suma, un arreglo y un entero que contendrá la respuesta de la suma previamente mencionada. También, cuenta con un método constructor el cual construye un nuevo hilo; así como, el método mencionado anteriormente

“creaEInicia”, el cual como su nombre lo indica, tiene la función de crear e iniciar un hilo; mientras que, el último método que tiene esta clase es el método sobrescrito “run()”, el cual indica en un principio que hilo es el que se está iniciando, para posteriormente dentro de synchronized, llamar a sumArray de la clase Suma, con la finalidad de crear un canal de comunicación entre todos los hilos (que retornan la suma en el método sumArray) para después poder imprimir el resultado obtenido, que indica la suma que realizó cada hilo e indicar que la ejecución de este se terminó

Finalmente se creó la clase Suma, esta clase tiene como atributo un valor entero “sum” el cual se ocupará para guardar el resultado de las sumas que se vayan realizando; también cuenta con un solo método, mencionado anteriormente, el cual tiene como objetivo recorrer el arreglo con el que se trabajará (el cual se pasa como parámetro) e ir sumando cada valor para posteriormente retornar la suma. Cabe recalcar que, para cada cambio de tarea se ocupó el método sleep, para que el hilo se suspendiera por 10 milisegundos.

La salida del programa al ejecutarlo desde el método main, es la siguiente:

```
run:
#2 iniciando.
#1 iniciando.
Total acumulado de #2 es 1
Total acumulado de #2 es 3
Total acumulado de #2 es 6
Total acumulado de #2 es 10
Total acumulado de #2 es 15
Total acumulado de #2 es 21
Total acumulado de #2 es 28
Total acumulado de #2 es 36
Total acumulado de #2 es 45
Total acumulado de #2 es 55
Total acumulado de #2 es 66
Total acumulado de #2 es 78
Total acumulado de #2 es 91
Total acumulado de #1 es 1
Suma para #2 es 91
#2 terminado.
Total acumulado de #1 es 3
Total acumulado de #1 es 6
Total acumulado de #1 es 10
Total acumulado de #1 es 15
Total acumulado de #1 es 21
Total acumulado de #1 es 28
Total acumulado de #1 es 36
Total acumulado de #1 es 45
Total acumulado de #1 es 55
Total acumulado de #1 es 66
Total acumulado de #1 es 78
Total acumulado de #1 es 91
Suma para #1 es 91
#1 terminado.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Tras este resultado podemos observar un aspecto fundamental dentro del código, es la sincronización como medio de comunicación entre los hilos presentes.

El bloqueo que aparece desde el monitor presente representa una forma en la que no se pueda acceder a un recurso que es compartido por los subprocesos presentes; si los hilos trabajan sin la sincronización, la sobreescritura de valores y la pérdida de información estarían presentes en todo momento.

Se puede observar, por ejemplo, que el recurso compartido “sumarray” está siendo sincronizado por los 2 hilos presentes, por lo que, través de un canal de comunicación con este método, el envío de información está presente para conocer si es que ya está siendo empleado el recurso.

No obstante, este no es el único ejemplo de sincronización, como bien se llegó a mencionar, las entradas a los cines es un claro ejemplo de la misma aplicada en el día a día.

Otro claro ejemplo en la vida cotidiana es en el uso de las redes sociales, pues varios usuarios acceden a un mismo programa (en este caso una app) al mismo tiempo y pueden compartir, reaccionar a publicaciones, así como mandar mensajes en una versión paralela. Así mismo, la natación sincronizada es otro ejemplo, ya que como su nombre lo indica, se basa en que dos o más personas (hilos), ejecuten una acción al mismo tiempo.

Igualmente, se tiene la sincronización entre los procesadores de texto al estar trabajando dentro de un mismo proyecto, evitando de esta forma que algún otro editor modifique en tiempo real lo que un primer editor ha realizado (mientras este esté activo)

Conclusiones

Carranza Ochoa José David

Para el desarrollo de esta práctica fue fundamental el conocimiento de la parte teórica para poder comprender los temas que se debía manejar dentro del paradigma orientado a objetos.

El trabajar desde un ambiente paralelo en conjunto a un ambiente estructurado como se observó en una materia hermana, contribuyó a un mejor desempeño de la misma al relacionar los conceptos de relaciones paralelas y algunos mecanismos para poder convertir algoritmos secuenciales en paralelo.

Así mismo, al igual que en prácticas previas, el trabajo en equipo ha sido pieza clave para trabajar justamente con este tipo d4e temas que son considerados de gran complejidad, ya que el poder abstraer la información dentro de una simple salida en pantalla resulta ser muy complicado teniendo en cuenta nuestra forma de pensar basada en la secuencialidad.

Es por ello que los conocimientos entre mi pareja y yo se conjuntaron para obtener un óptimo progreso de la práctica en cuestión, obteniendo resultados muy favorables que incentivaron a investigar un poco más sobre regiones paralelas dentro de la cotidianidad.

Sin embargo, a pesar de trabajar de forma correcta, algunos primeros análisis de los códigos resultaron en respuestas vagas que tuvieron que ser modificadas por completo para poder obtener un avance exitoso de la práctica, por lo que se determina que antes de efectuar la práctica, los conocimientos teóricos se encuentren bien cimentados para cumplir con los objetivos.

Tales objetivos para esta ocasión se cumplieron en su totalidad (100%), retornando el avance tenido dentro del paradigma orientado a objetos como una buena forma de aumentar de nivel del mismo.

Carrillo Cervantes Ivette Alejandra

Se cumplieron los objetivos de esta práctica, ya que utilizamos el concepto de multiarea utilizando hilos en un lenguaje orientado a objetos, en este caso Java. Pudimos ver esto mediante los ejercicios impartidos por el laboratorio, tanto como con los ejercicios impartidos por el profesor.

Durante el desarrollo de esta práctica, fue de gran ayuda conocer los conceptos básicos para trabajar con hilos, ya que antes de ejecutar el programa ya teníamos una idea de lo que podría salir como resultado; sin embargo, no fue en todos los casos, ya que en algunos casos no se tenía la salida como la imaginábamos, no obstante, considero que gracias a estas pequeñas inquietudes fue mejor nuestro aprendizaje, ya que ambos nos explicábamos el concepto si uno no lo entendía.

Considero que realizar esta práctica en equipo, fue de gran apoyo para comprender de una mejor forma el tema de hilos, puesto que como lo mencione anteriormente, nos íbamos explicando en caso de no entender algún concepto, o bien alguna parte del código.

Finalmente, puedo decir que los ejercicios de esta práctica estuvieron bien planteados para observar y analizar tanto la creación de hilos, sus prioridades, así como la forma en que estos se pueden comunicar. Personalmente, reforcé varios conceptos con ayuda de esta práctica. (: