Tecnológico Nacional de México

Instituto Tecnológico Culiacán

Unidad III: Programación Concurrente

Síntesis de Multihilos

Asignatura: Tópicos Avanzados de Programación

Unidad III: Programación concurrente (Multihilos)

Docente: MC Jaime Arturo Félix Medina

Hora: 17:00 – 18:00

Alumno: Jaime Alonso Ruiz Lizarraga

No. De Ctrl: 19170736

Fecha: 12/12/2022

Índice

Introducción

Como tal, antes de indagar más en los conceptos de hilos, se requiere conocer más específicamente lo que es la programación concurrente. Esta teóricamente se podría conocer cómo la división de un problema en subproblemas que se solucionan de forma individual, para crear un programa o aplicación que no se vea afectado en tiempo real, por lo tanto, hace referencia a las técnicas de programación que son utilizadas para expresar la concurrencia entre tareas y solución de los problemas de comunicación y sincronización entre procesos. La programación concurrente es la ejecución simultanea de múltiples tareas interactivamente. Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa. Las tareas se pueden ejecutar en una sola CPU, en varios procesadores o en una red de computadoras distribuidas.

También debemos comprender el concepto de proceso. Un proceso no solamente es el código de un programa, este tiene un contador de programa el cual es un registro dentro de la computadora a que indica la dirección de la siguiente instrucción que será ejecutara por el procesoador

- **Pila:** la cual contiene datos temporales como: parámetros de funcionaes, direcciones de retorno, variables locales, etc. Etc.
- **Sección de datos:** contiene datos locales.
- **Heap:** es el cúmulo de memoria que se asigna dinámicamente al proceso.

1. Concepto de un hilo

Un hilo o también conocido como Thread por su traducción al inglés es un flujo de control dentro de un programa, el cual permite tener múltiples procesos corriendo de manera simultánea. Se puede definir como una unidad básica de ejecución del sistema operativo el uso del procesador. Es el que se encarga de realizar todos los cálculos para que el programa pueda ejecutarse correctamente. Es necesario ya que debe contar con al menos un hilo para que cualquier programa se pueda ejecutar. D entro de las características que se encuentran se tiene que cada hilo tiene información del código de máquina que se va a ejecutar en el procesador, sus respectivos datos, el acceso a los archivos, el registro y su respectivo stack en donde se guarda toda la información necesaria del hilo como son las variables locales, variables de retorno o algo a lo que se acceda en tiempo de ejecución. Dentro de las semejanzas de hilo y multihilo es que poseen un solo bloque de control de

proceso y un solo espacio de direcciones de proceso. Se dice que los hilos de ejecución que comparten recursos agregando estos recursos da como resultado el proceso.

Características:

- **Modelo apropiativo:** Los hilos deben operar bajo la suposición de que su ejecución puede verse interrumpida por el sistema operativo en cualquier momento.
- **Posibilidad de recursos compartidos:** Diferentes hilos en un proceso pueden acceder a la misma región de memoria, o descriptores de archivos comunes (entre otros).
- **Soporte directo al paralelismo:** Se puede asumir con confianza que usar hilos en un sistema multiprocesador lleva a una mayor utilización de esta característica.

Diseño**:**

- **Creación de un hilo:** La abstracción básica es la ejecución de una función como punto de entrada para un hilo individual, y la obtención de un recurso vinculado a dicha ejecución.
- **Un hilo puede comenzar con:**
  - Un puntero a una función
  - Una clausura
  - Un método en un objeto
- **Un hilo se puede monitorear con:**
  - Un identificador en una tabla
  - Un objeto nuevo
  - El estado del objeto que la crea
- **Comunicación:** Los hilos se pueden comunicar por medio de memoria compartida. Para asegurar que esta memoria es accedida de manera correcta, se provee al programador con primitivas de sincronización: semáforos, exclusiones mutuas, tipos atómicos.

Multiprogramación: Es una técnica de multiplexación que permite de múltiples procesos en un único procesador. Es importante resaltar que los procesos nunca corren en paralelo en el procesador, ya que en cada instante de tiempo solo se ejecuta un proceso en el procesador.

Multiproceso: Es una técnica en la cual se hace uso de dos o más procesadores en una computadora para ejecutar uno o varios procesos.

Event Loop

Características:

- **Modelo bloqueante deferido:** Se establece un mecanismo para registrar la ocurrencia de eventos y la ejecución de tareas en respuesta a diferentes tipos de eventos, al final de un ciclo de ejecución.
- **Concurrente, no paralelo por defecto:** Este modelo permite describir la ocurrencia de diferentes eventos intercaladamente en un sólo hilo, y responder a eventos que serían bloqueantes con el registro de un evento. De esta manera se consigue un uso óptimo de recursos, pero no se modela el acceso a capacidades paralelas.
- **Memoria local:** El alcance de los recursos es análogo al de un programa secuencial.

Diseño:

El registro de acciones a ejecutar en respuesta a eventos tiene una solución que se ajusta perfectamente: las clausuras.

Sin embargo, el uso de clausuras para el uso de los valores resultantes de computaciones deferidas tiene problemas de ergonomía (véase: pirámide de la perdición).

Alternativas:

- **Promesas/Futuros:** Objetos con métodos adecuados.
- **Async/Await:** Palabras claves para describir computaciones deferidas.

Corrutinas

Características:

- **Modelo cooperativo:** Las corrutinas tienen que ser programadas de manera que cedan la ejecución en momentos apropiados, y eviten acaparar recursos.
- **Contextos ligeros:** Las corrutinas requieren un uso de memoria considerablemente menor, gracias a que no definen el contexto completo: un solo hilo de S.O. puede aprovisionar múltiples corrutinas.
- **Soporte opcional de paralelismo:** Las corrutinas suelen modelar la concurrencia como un cambio de contexto dentro de un mismo hilo, sin embargo, es probable que un runtime moderno permita configurar para usar múltiples hilos.

Diseño:

- **Cooperación:** Deben existir constructos en el lenguaje (palabras clave, interfaces, u otros) que permitan señalizar el punto en el que una corrutina cede la ejecución. Algunas formas de sintaxis que pueden permitir esto:
    - Funciones que retornan handles, con métodos adecuados.
    - Objetos compartidos que monitorean el estado de cada subrutina.
    - Palabras claves que indican la oportunidad para cambiar de contexto, como yield o await.

Actores

Características:

- **Procesos ligeros:** Un actor es una unidad de procesamiento análoga al proceso, pero manejada por un proceso del S.O. real: la máquina virtual del lenguaje de programación. Gracias a esto, cada actor es extremadamente pequeño.
- **Modelo apropiativo:** La máquina virtual actúa como el árbitro de la ejecución de los actores, deteniendo su ejecución de acuerdo a medidas globales.
- **Paralelismo:** Según las capacidades de la máquina virtual, el paralelismo puede ser tan automático como al usar hilos.

Diseño:

El modelo de actores establece que estas unidades mínimas no comparten memoria, y su viabilidad es independiente de la de los otros. Esto implica:

- **Comunicación por mensajes:** los actores se informan de su estado y solicitan computaciones a partir de mensajes, y, por tanto, no comparten memoria mutable.
- **Pureza:** Los actores solo conocen su estado interno, y los mensajes que obtienen del exterior, que pueden pensarse como argumentos a una función pura. De ahí la capacidad de paralelismo.
- **Manejo de errores optimista:** los errores causados por condiciones inusuales y transitorias son manejados reiniciando un actor, manteniendo el programa global en curso.

Los procesos también cuentan con lo que llamamos estados, que es cuando está siendo ejecutado obviamente cambiará su estado en base a su uso, los cuales se clasifican de la siguiente forma:

- **Nuevo**: proceso en fase de creación.
- **Corriendo**: se están ejecutando sus instrucciones.
- **Espera**: nuestro proceso está a la espera de ‘tal’ evento.

- **Preparado**: listo a la espera de ser asignado al procesador.
- **Terminado:** el proceso ha terminado la ejecución.

Bloque de control de proceso

Cada proceso se representa en el sistema operativo mediante el PCB, entre los elementos de información que contiene:

- **Estado del proceso:** New, Ready, Running, Waiting, Halted...
- Contador del programa
- **Registros de la CPU:** Varían en cuanto a número y tipo dependiendo de la arquitectura del procesador.
- **Información de planificación de la CPU:** Parámetros de planificación como prioridad de procesos y punteros a las colas de planificación.
- **Información de gestión de memoria:** Tablas de páginas, tablas de segmentos, dependiendo los mecanismos de gestión del Sistema operativo.

2. Comparación de un programa de flujo único contra uno de flujo múltiple.

Con ayuda de los hilos podemos ejecutar dos o más procesos al mismo tiempo, sin tener que esperar a que finalice un proceso para poder ejecutar el siguiente. Pues las instrucciones de estos se van ejecutando confirme se van llamando, estas instrucciones pueden estar en diferentes métodos o clases, pero sólo se van ejecutando de manera única conforme son llamadas.

Un programa de flujo único o de tarea única, utiliza un único flujo de control para controlar su ejecución, lo cual no se ocupa especificar explícitamente. Para ejemplificarlo utilizaremos el famoso programa del “Hola mundo”:

```
public class HolaMundo{
        public static void main (string args[]){
                System.out.println( " Hola Mundo " );
        }
}
```

Aquí, Cuando se llama al main(), la aplicación imprime el mensaje y termina. Esto ocurre dentro de un único hilo. La sincronización entre las múltiples partes de un programa se lleva a cabo a medida de un bucle de sucesión único. Estos entornos son de tipo síncrono, gestionados por sucesos.

Por otra parte, contamos con el flujo múltiple, el cual permite que se estén realizando diferentes tareas en un programa al mismo tiempo; en decir, que esas tareas se ejecuten de forma paralela, mediante los elementos que conocemos como hilos.

Mientras que los programas de flujo único pueden realizar su tarea ejecutando las subtareas secuencialmente, un programa multitarea permite que cada tarea comience y termine tan pronto como sea posible. Un ejemplo sería el progrma de productor-consumidor realizado en esta unidad, en el que se muestran los diferentes estados en los que puede estar una celda del bufer(hilo), así como la comunicación que se puede generar entre diferentes hilos para lograr una aplicación más robusta.

A continuación, tenemos un ejemplo de hilos heredando la clase Thread:

```
class TestTh extends Thread {
        private String nombre;
        private int retardo;
        // Constructor para almacenar nuestro nombre y el retardo
        public TestTh( String s,int d ) {
                nombre = s;
                retardo = d;
        }
        /* El metodo run() es similar al main(), pero para threads. Cuando run() termina
        el hilo muere*/
        public void run() {
                // Retasamos la ejecución el tiempo especificado
                try {
                        sleep( retardo );
                } catch( InterruptedException e ) {
                        ;
                }
                // Ahora imprimimos el nombre
                System.out.println( "Hola Mundo! "+nombre+" "+retardo );
```

```
        }
        }
        public class MultiHola {
                public static void main( String args[] ) {
                        TestTh t1,t2,t3;
                        // Creamos los threads
                        t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
                        t2 = new TestTh( "Thread 2",(int)(Math.random()*2000) );
                        t3 = new TestTh( "Thread 3",(int)(Math.random()*2000) );
                        // Arrancamos los threads
                        t1.start();
                        t2.start();
                        t3.start();
                }
        }
}
```

Como se observó en el código, se crearon tres tareas individuales, que imprimen cada una de ellas, un mensaje distinto.

Mientras que los programas de flujo único pueden realizar su tarea ejecutando las subtareas secuencialmente, un programa multitarea permite que cada tarea comience y termine tan pronto como sea posible. Este comportamiento presenta una mejor respuesta a la entrada en tiempo real. Cada hilo realiza una tarea en específico, al tener varios hilos ejecutándose, se tendrán varias tareas corriendo en el mismo programa. Lo que permitirá que en el mismo programa se estén realizando diferentes actividades al mismo tiempo

3. Creación y control de hilos

En cuanto a la creación de hilos, en Java conocemos dos formas de hacer procesos concurrentes, la primera es hacer uso de la clase Thread creando una subclase, y la segunda es crear una clase que implemente la interfaz Runnable.

Uso de la clase Thread

Cuando se crea una subclase de Thread, la subclase debería definir su propio método run() para sobre montar el método run() de la clase Thread. La tarea concurrente es desarrollada en este método run().

Una instancia de la sublase es creada con new, luego llamamos al método start() del hilo para hacer que la máquina virtual de Java ejecute el método run(). Para iniciar la concurrencia invocamos al método start(), asó invocamos a run() de forma indirecta. Si invocamos a run() directamente, se comportará como el llamado a cualquier método centro de un mismo hilo (sin crear uno independiente).

Uso de la Interfaz Runnable

La interfaz Runnable requiere que solo un método sea implementado, el método run(). Primero creamos una instancia de esta clase con new, luego creamos una instancia de Thread con otra sentencia new y usamos el objeto recién creado en el constructor. Finalmente, llamamos al método start() de la instancia de Thread para iniciar la tarea definida en el método run().

Una instancia de una clase que defina el método run() (ya sea como subclase de Thread o implementando la interfaz Runnable) debe ser pasada como argumento en la creación de una instancia de Thread. Cuando el método start() de esta instancia es llamado, Java run time sabe qué método run() debe ejecutar.

Control de un hilo

Para el correcto control de un hilo contamos con varios métodos de la clase java.lang.Thread para controlar su ejecución:

- **void start():** este método es utilizado para iniciar el cuerpo de un hilo definido por el método run().
- **void sleep():** método utilizado para poner a dormir a un hilo por un tiempo mínimo previamente especificado.
- **void join():** método usado para esperar por el término de un hilo sobre el cual el método es invocado, por ejemplo, por término del método run().
- **void yield():** método utilizado para mover el hilo desde el estado de corriendo al final de la cola de procesos en espera por la CPU.

Java 2 dejó obsoleto varios métodos de control definidos en versiones previas para prevenir inconsistencia de datos y *deadlock*. Se recomienda evitar el uso de los siguientes métodos por los problemas comentados:

- **void stop():** método utilizado para detener la ejecución de un hilo sin importar consideración alguna.
- **void suspend():** método utilizado para detener temporalmente la ejecución de un hilo.
- **void resume():** método utilizado para reactivar reactiva un método previamente suspendido.

Deadlock

Al deadlock también se le conoce como abrazo mortal, ocurre cuando un proceso espera un evento que nunca va a pasar. Aunque se puede dar por comunicación entre procesos, es más frecuente que se dé por manejo de recursos. En este caso, deben cumplirse 4 condiciones para que se dé un "deadlock" :

- Los procesos deben reclamar un acceso exclusivo a los recursos.
- Los procesos deben retener los recursos mientras esperan otros.
- Los recursos pueden no ser removidos de los procesos que esperan.
- Existe una cadena circular de procesos donde cada proceso retiene uno o más recursos que el siguiente proceso de la cadena necesita.

Ciclo de vida de un hilo.

Cada hilo, después de su creación y antes de destrucción, estará en uno de los siguientes estados:

- **Recién creado (new thread):** entra aquí inmediatamente después de su creación. Es decir, luego del llamado a new. En este estado los datos locales son ubicados e iniciados. Luego de la invocación a start(), el hilo pasa al estado "corrible/Runnable".
- **Corrible (Runnable):** Aquí el contexto de ejecución existe y el hilo puede ocupar la CPU en cualquier momento. Este estado puede subdividirse en dos: Corriendo y encolado. La transición entre estos dos estados es manejada por le iterador de la máquina virtual. Un hilo que invoca al método yield() voluntariamente se mueve a sí misma al estado encolado desde el estado corriendo.
- **Bloqueado (not Runnable):** Se ingresa cuando se invoca suspend(), el hilo invoca el método wait() de algún objeto, el hilo invoca sleep(), el hilo espera por alguna operación de E/S, o el hilo invoca join() de otro hilo para espera por su término. El hilo vuelve al estado Corrible cuando el evento por que espera ocurre.

- **Muerto (Dead):** Se llega a este estado cuando el hilo termina su ejecución (concluye el método run) o es detenida por otro hilo llamando al su método stop(). Esta última acción no es recomendada.

4. Sincronización de hilos

Todos los hilos de un programa comparten el espacio de memoria, haciendo posible que dos hilos accedan la misma variable o corran el mismo método de un objeto al "mismo tiempo". Se crea así la necesidad de disponer de un mecanismo para bloquear el acceso de un hilo a un dato crítico si el dato está siendo usado por otro hilo.

La sincronización de hilos es una sección bastante importante en la programación concurrente, dando paso a lo que podría interpretarse como la administración de nuestros recursos, ya que nuestros subprocesos comparten objetos entre sí, es decir, que pueden ser modificados por uno o más de ellos, lo que nos puede llevar a errores/fallas o algún imprevisto en general durante el funcionamiento, lo cual se puede evitar o prevenir teniendo una correcta gestión de nuestros objetos.

¿Qué clase de errores se pueden derivar de la falta de sincronización?

Bueno en primer lugar, imaginemos que contamos con 2 procesos que intentan actualizar un recurso compartido de manera simultánea, es decir nuestro hilo 1 está en proceso de actualizarlo, pero a la vez hilo 2 desea hacerlo igual, no podemos saber cuál de las dos tomó nuestro programa. Por ende, el comportamiento del sistema puede ser impredecible, puede dar resultados tanto correctos como incorrectos, es como lanzar una moneda al aire (en cuanto a si el comportamiento será como se espera o no) y no podremos saber cómo resultará.

Pero por suerte nuestra problemática tiene una maravillosa solución llamara acceso exclusivo, que como su nombre lo indica, mantiene el acceso restringido a otros hilos cuando este objeto ya se encuentra en posesión de un hilo x, haciendo que cualquier otro que desee manipular y/o actualizar sus valores deba permanecer en espera. Y, una vez nuestro hilo x con acceso exclusivo libera nuestro maravilloso dato, este pasa otro de los tantos hilos que lo estaban esperando y continue con su paso.

Todo lo mencionado anteriormente se puede encapsular como la sincronización de hilos, que coordina los accesos los datos de nuestro programa en base a los hilos concurrentes. Al utilizar la sincronización podemos garantizar que los subprocesos privan de otros cuando acceden a recursos compartidos, para que no haya manipulación simultánea, los que se le conoce como exclusión mutua.

¿Cómo podemos implementar la sincronización de hilos en Java?

Afortunadamente el entorno de desarrollo de Java nos proporciona una valiosa herramienta llamada *monitores,* el cual se puede considerar como un módulo que encapsula servicios mediante métodos de acceso, así como sus variables locales y globales.

Java utiliza la idea de monitores para sincronizar el acceso a datos. Un monitor es un lugar bajo guardia donde todos los recursos tienen el mismo candado. Sólo una llave abre todos los candados dentro de un monitor, y un hilo debe obtener la llave para entrar al monitor y acceder a esos recursos protegidos. Cada objeto en Java posee un candado y una llave para manejo de zonas críticas. También existe un candado y llave por cada clase, éste se usa para los métodos estáticos.

Si varios hilos desean entrar al monitor simultáneamente, sólo uno obtiene la llave. Los otros son dejados fuera (bloqueados) hasta que el hilo con la llave termine de usar el recurso exclusivo y devuelva la llave a la Máquina Virtual Java. Puede ocurrir deadlock si dos hilos están esperando por el recurso cuya llave la tiene el otro. En un momento un hilo puede tener las llaves de varios monitores. En Java los recursos protegidos por un monitor son fragmento de programa en forma de métodos o bloques de sentencias encerradas con paréntesis {}.

Synchronized

La palabra reservada *synchronized* es usada para indicar que el siguiente método o bloque de sentencias es sincronizada por un monitor (aquél del objeto que tiene el método). Cuando queremos sincronizar un bloque, un objeto encerrado por ( ) sigue a la palabra synchronized, así la máquina virtual sabe qué monitor chequear.

Llave del monitor

Existe una llave única por cada objeto que contiene un método sincronizado (synchronized) asociado a su instancia (método no estático), o que es referenciado en un bloque synchroinized (es usado como argumento de esta sentencia). Por ello cada objeto y cada clase puede tener un monitor si hay cualquier método sincronizado o bloque de sentencias asociado con ellos. Más aún, la llave de un monitor de una clase es diferente de las llaves de los monitores de las instancias (esto porque el método puede ser llamado antes de existir ninguna instancia).