Instituto Politécnico Nacional

IPN

Escuela Superior de Cómputo

ESCOM

Academia de Redes Aplicaciones para comunicaciones en red

6CV2

Práctica 18

“Ciclo de vida del hilo”

Alumna:

Navarrete Becerril Sharon Anette

Fecha de entrega: “ 25 - Noviembre - 2024”

Profesor: Ojeda Santillan Rodrigo

OBJETIVO

Comprender y analizar el ciclo de vida de un hilo, desde su creación hasta su terminación, para entender cómo las aplicaciones pueden ejecutar tareas en paralelo, mejorando el rendimiento mediante la gestión eficiente de procesos concurrentes.

INTRODUCCIÓN

Un hilo, o thread, es una unidad fundamental de procesamiento en la ejecución de tareas dentro de un sistema operativo, permitiendo que las aplicaciones realicen múltiples tareas de forma concurrente y eficiente. Los hilos son una forma de ejecutar varios bloques de código en paralelo, especialmente útil en programas que requieren la ejecución simultánea de varias operaciones, como servidores web, aplicaciones de red y sistemas multitarea. Al pertenecer a un mismo proceso, todos los hilos de ese proceso comparten su espacio de memoria, lo que facilita el intercambio de datos y reduce la sobrecarga de recursos.

El uso de hilos permite que una aplicación mantenga una experiencia de usuario fluida, ya que se evita que una tarea extensa o intensiva en procesamiento bloquee el funcionamiento de otras. Por ejemplo, en un servidor web, un hilo puede manejar una solicitud de un cliente mientras otro hilo procesa una solicitud diferente, optimizando el tiempo de respuesta y el rendimiento general del servidor. Así, los hilos mejoran la eficiencia de programas complejos al distribuir la carga de trabajo entre múltiples hilos en lugar de depender de una sola secuencia de ejecución.

Cada hilo atraviesa un ciclo de vida definido que abarca desde su creación hasta su finalización, pasando por estados que regulan su ejecución, espera y terminación. Este ciclo de vida incluye varias fases, como el estado "Nuevo" (donde el hilo se ha creado pero aún no ha iniciado), el estado "Runnable" (cuando el hilo está listo para ejecutarse), el estado "Running" (cuando el hilo está siendo ejecutado por el procesador) y el estado "Blocked" o "Waiting" (cuando el hilo está en espera de un recurso o condición para continuar). Por último, el hilo llega al estado "Terminated" una vez que ha completado su tarea o ha encontrado una excepción que no puede manejar.

Orden de ejecución del ciclo de vida del hilo:

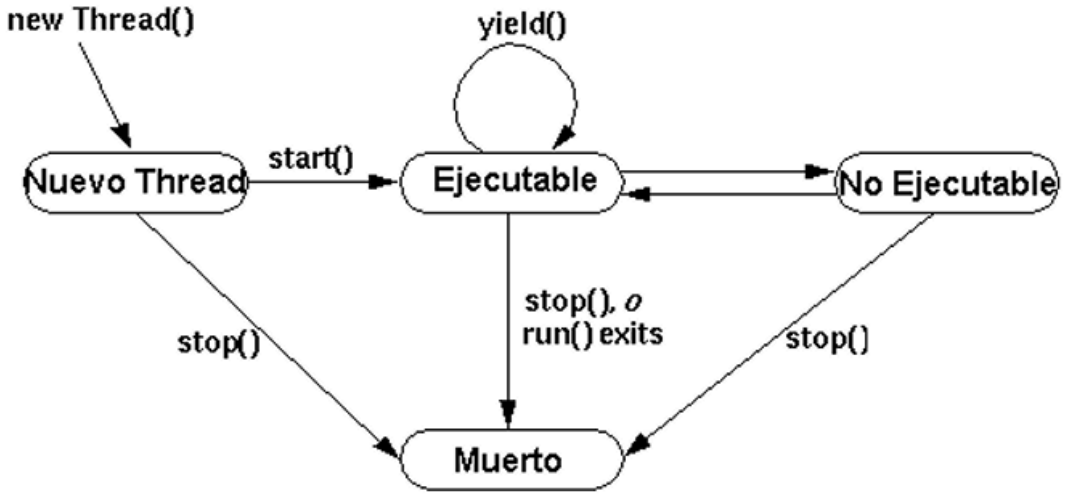
1. **New (Nuevo):**
   En esta etapa, el hilo ha sido creado, pero aún no se ha iniciado. Esto corresponde a la instancia de un objeto de hilo (como Thread en Java), que existe pero no se ha puesto en marcha.
2. **Runnable (Ejecutable):**
   Después de invocar el método start(), el hilo entra en el estado "Runnable". Aquí, el hilo está listo para ejecutarse y está en la cola de hilos esperando su turno en el procesador. Este estado se divide en dos subestados:
   - **Ready (Listo)**: El hilo está preparado para ejecutarse, pero aún espera su turno en el procesador.
   - **Running (En ejecución):** El hilo ha sido seleccionado por el procesador y está ejecutando su tarea.

3. **Blocked/Waiting (Bloqueado/Esperando):**
   Durante la ejecución, el hilo puede necesitar esperar por un recurso o por una condición específica, entrando en el estado "Blocked" o "Waiting":
   - **Blocked (Bloqueado):** El hilo espera que se libere un recurso, como un monitor o bloqueo.
   - **Waiting (Esperando):** El hilo espera indefinidamente a ser notificado por otra tarea para reanudar.
   - **Timed Waiting (Espera temporal):** El hilo espera un tiempo específico antes de reanudar (por ejemplo, mediante sleep() o wait(long millis)).
4. **Terminated (Terminado):**
   Cuando el método run() finaliza su ejecución o si ocurre una excepción que el hilo no puede manejar, el hilo pasa al estado "Terminated". En este estado, el hilo ha completado su ciclo de vida y no puede volver a ejecutarse.

Transiciones entre Estados:

- Runnable → Running: Cuando el sistema operativo selecciona el hilo para ejecución.
- Running → Blocked/Waiting: Si el hilo necesita un recurso o debe esperar una condición.
- Blocked/Waiting → Runnable: Cuando el recurso está disponible o se cumple la condición.
- Running → Terminated: Al completar el método run() o cuando ocurre una excepción no manejada.

Este ciclo de vida y su orden de ejecución permiten un control detallado sobre la asignación de recursos y la administración de tareas paralelas, optimizando la eficiencia y evitando bloqueos en la ejecución de tareas concurrentes en aplicaciones.



DESARROLLO

Se comienza por crear una clase en java con tu IDE, en este caso se va a realizar un cliente que se ejecute en Windows o en una máquina virtual y el servidor multihilo en Ubuntu. Se comienza estableciendo la lógica para el cliente:

```
import java.io.*;
import java.net.*;

public class Cliente {
    private static final String SERVER_ADDRESS = "127.0.0.0"; // Cambia por >
    private static final int SERVER_PORT = 8080;

    public static void main(String[] args) {
        try (Socket socket = new Socket(SERVER_ADDRESS, SERVER_PORT);
             BufferedReader input = new BufferedReader(new InputStreamReader>
             PrintWriter output = new PrintWriter(socket.getOutputStream(), >
             BufferedReader console = new BufferedReader(new InputStreamRead>

            System.out.println("Conectado al servidor. Escribe mensajes:");

            String mensaje;
            // Enviar mensajes al servidor
            while ((mensaje = console.readLine()) != null) {
                output.println(mensaje); // Enviar mensaje al servidor
                System.out.println("Servidor responde: " + input.readLine())>
            }
        } catch (IOException e) {
            e.printStackTrace();
        }    }   }
```

Este código en Java implementa un cliente que se conecta a un servidor mediante sockets, utilizando la dirección IP y un puerto específico del servidor. La funcionalidad se basa en las bibliotecas de Java para comunicación en red y manejo de entrada/salida (I/O).

El ciclo de vida del hilo principal en este programa sigue el siguiente orden:

- New (Nuevo): Al iniciar el programa, el hilo principal se encuentra en el estado "Nuevo", que representa la instancia del cliente antes de comenzar la ejecución.
- Runnable (Ejecutable): Cuando el programa comienza a ejecutarse, el hilo pasa al estado "Runnable" y continúa su ejecución en el método main().
- Running (En ejecución): El hilo principal entra en el estado "Running" cuando realiza las siguientes operaciones:
    - ★ Creación del socket: Se conecta al servidor utilizando la dirección IP (SERVER_ADDRESS) y el puerto (SERVER_PORT). En este punto, el cliente establece una conexión TCP con el servidor.
    - ★ Inicialización de flujos de I/O: Se crean flujos de entrada (BufferedReader input) y de salida (PrintWriter output) que permiten enviar y recibir mensajes entre el cliente y el servidor. Adicionalmente, se usa un tercer BufferedReader (console) para leer la entrada del usuario desde la consola.
- Blocked/Waiting (Bloqueado/Esperando): El hilo puede entrar en estado de espera cuando:
    - ★ Está esperando la entrada del usuario a través de console.readLine().
    - ★ Está esperando una respuesta del servidor mediante input.readLine().

- Running (En ejecución): Luego de recibir los datos, el hilo retoma su ejecución para continuar enviando mensajes al servidor (output.println(mensaje)) y mostrando las respuestas del servidor en la consola (System.out.println("Servidor responde: " + input.readLine())).
- Terminated (Terminado): El hilo finaliza su ciclo de vida cuando el cliente decide cerrar la conexión (por ejemplo, enviando una señal de salida) o si ocurre una excepción que se maneja en el bloque catch. El bloque try-with-resources asegura que todos los recursos, como el socket y los flujos de datos, se cierren automáticamente al finalizar.

Ahora se debe de realizar la lógica para el servidor:

```
import java.io.*;
import java.net.*;

public class ServidorMultithread {
        private static final int PORT = 8080; // Puerto en el que el servido>
        public static void main(String[] args) {
                try (ServerSocket serverSocket = new ServerSocket(PORT)) {
                        System.out.println("Servidor en espera de conexiones>
                        while (true) {
                                // Aceptar la conexión del cliente
                                Socket clienteSocket = serverSocket.accept()>
                                System.out.println("Cliente conectado: " + c>
                                // Crear un nuevo hilo para manejar al clien>
                                ClientHandler handler = new ClientHandler(cl>
                          new Thread(handler).start();
                        }
                } catch (IOException e) {
                        e.printStackTrace();
                }     }    }
```

Este código implementa un servidor multihilo que permite gestionar múltiples clientes simultáneamente en una red. Utiliza la clase ServerSocket para escuchar en un puerto específico y aceptar conexiones entrantes de los clientes.
Cada vez que un cliente se conecta, el servidor crea un socket dedicado para esa conexión y lanza un nuevo hilo para manejar la comunicación con ese cliente. Esto permite que el servidor siga aceptando más conexiones sin bloquearse, ya que cada cliente es gestionado de forma independiente en su propio hilo. Gracias a esta arquitectura, el servidor puede atender varias conexiones a la vez, asegurando un manejo eficiente y simultáneo de múltiples clientes.

Debido a que ClientHandler.java no se encuentra en ninguna biblioteca estandar de java de forma nativa, fue necesario crearla con el siguiente código:

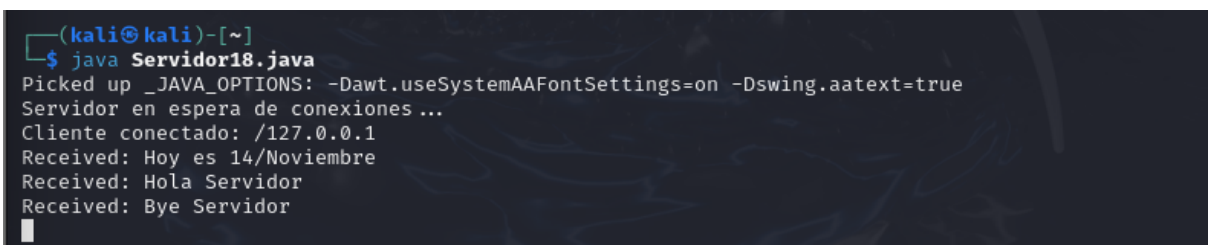
```
import java.io.*;
import java.net.*;

public class ClientHandler implements Runnable {
    private Socket clientSocket;

    public ClientHandler(Socket socket) { this.clientSocket = socket; }

    @Override
    public void run() {
```

```
        try (
            BufferedReader in = new BufferedReader(new
InputStreamReader(cli>
            PrintWriter out = new
PrintWriter(clientSocket.getOutputStream()>
        ) {
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                System.out.println("Received: " + inputLine);
                out.println("Echo: " + inputLine);
            }
        } catch (IOException e) {
            System.err.println("Error in ClientHandler: " +
e.getMessage());
        } finally {
            try {
                clientSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }    }    }
```

Capturas del funcionamiento:

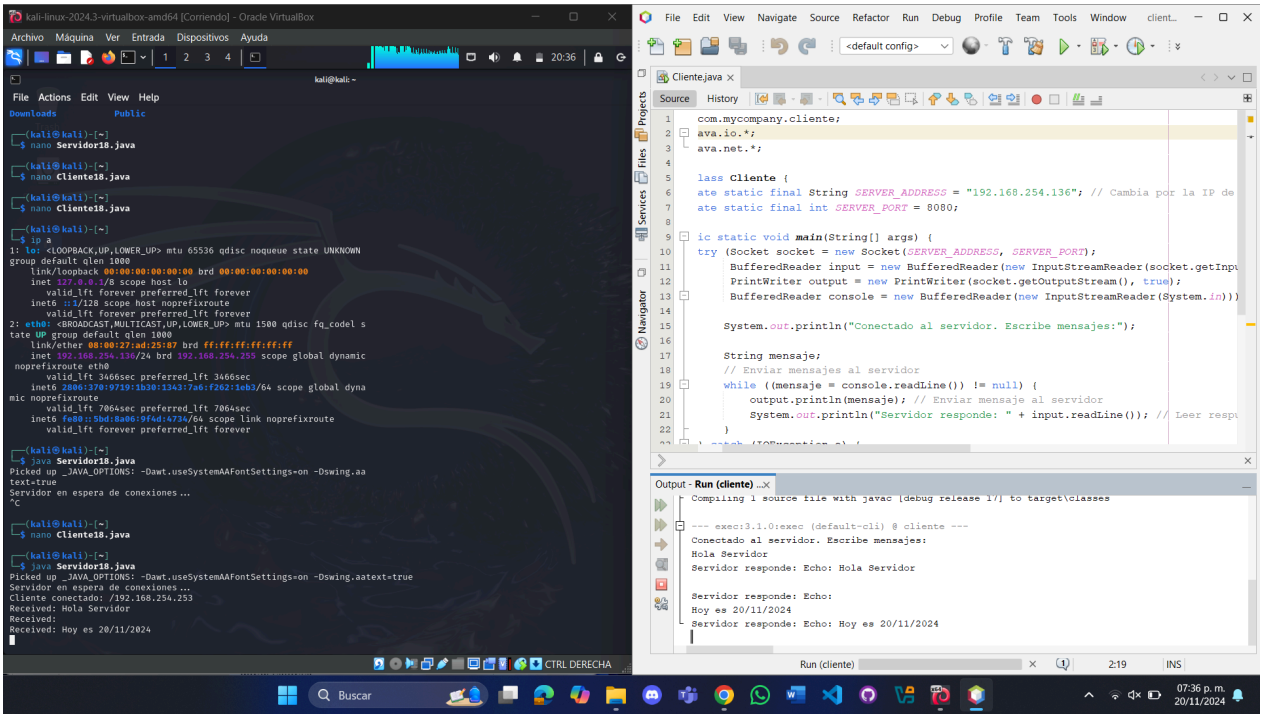
Cliente:

Servidor:



Al ejecutar las clases en un entorno Windows y el otro en java tenemos las siguientes salidas.

Primero usamos el comando ip a para poder conocer la ip que corresponde a la máquina virtual, ejecutamos el servidor en la MV de Kali la cual tiene un adaptador puente proporcionado por el mismo VirtualBox, posteriormente en Netbeans ejecutamos el servidor en mi maquina en Windows, modificando al cliente con la ip del servidor, teniendo como respuesta la siguiente:



```
┌──(kali㉿kali)-[~]
└─$ java Servidor18.java
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Servidor en espera de conexiones...
Cliente conectado: /192.168.254.253
Received: Hola Servidor
Received:
Received: Hoy es 20/11/2024
Received: Bye servidor
```

CONCLUSIONES

La independencia de cada hilo en el manejo de los clientes asegura que las tareas de comunicación sean ejecutadas en paralelo, permitiendo que los clientes interactúen con el servidor sin interferencias, lo que resulta en una experiencia de usuario fluida y en una administración de red efectiva.

BIBLIOGRAFÍA

Stevens, W. Richard. Programación en red con Unix: La API de Sockets. Addison-Wesley Professional, 2003.

Forouzan, Behrouz A. Comunicaciones de Datos y Redes. McGraw-Hill, 2012. Tanenbaum, Andrew S., y David J. Wetherall. Redes de Computadoras. Pearson, 2010.

Comer, Douglas E. Internetworking con TCP/IP Volumen Uno. Prentice Hall, 2006. Kurose, James F., y Keith W. Ross. Redes de Computadoras: Un Enfoque Descendente. Pearson, 2017.

Beazley, David, y Brian K. Jones. Python Cookbook: Recetas para Dominar Python 3. O'Reilly Media, 2013. Begg, A. Sockets TCP/IP en C: Guía Práctica para Programadores. Morgan Kaufmann, 2000