INSTITUTO POLITÉCTICO NACIONAL

Escuela Superior de Cómputo

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Practica 1:

Procesos e Hilos

Alumno:

Cardoso Osorio Atl Yosafat

Profesor:

Chadwick Carreto Arellano

Fecha:

24/02/2025

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Índice de contenido

Índice de figuras

Antecedentes

En la actualidad, los sistemas de reservas en línea deben ser capaces de manejar múltiples solicitudes simultáneas para garantizar una distribución equitativa y eficiente de los recursos disponibles. En el caso de la venta de boletos para eventos, la concurrencia en la compra puede ocasionar problemas si no se gestiona adecuadamente el acceso a los recursos compartidos.

Procesos

El concepto de procesos es considerado el más importante para todos los sistemas operativos. Los procesos son una de las abstracciones fundamentales, más antiguas e importantes que ofrecen los sistemas operativos, ya que permiten realizar operaciones de forma (pseudo) concurrente, incluso cuando solo contamos con una CPU disponible, convirtiéndola en varias CPU virtuales [1].

Algunas de las definiciones que se le ha dado a este concepto hacen referencia a la unidad de trabajo en los sistemas modernos de tiempo compartido o una entidad que comprende uno o más hilos y los recursos del sistema asociados (como la memoria, archivos abiertos y dispositivos) [2]. Además, una entidad que puede ser asignada a un procesador y ejecutada por él [3], un programa en ejecución, una instancia de programa ejecutándose en un computador, o en términos más amplios, una unidad de actividad caracterizada por un solo hilo secuencial de ejecución, un estado actual, y un conjunto de recursos del sistema asociado [4]. Así, podemos decir que un conjunto de procesos, que incluye tanto procesos del sistema operativo como procesos de usuario, constituyen a un sistema [2].

Cada proceso tiene asociado un espacio de direcciones en donde se encuentran el programa ejecutable, junto con los datos del programa (variables, espacio de trabajo, buffers, etc.) y su pila (parámetros de funciones, direcciones de retorno, variables locales, etc.). Así como una lista de ubicaciones de memoria y también un conjunto de recursos que generalmente incluye registros, una lista de archivos abiertos, alarmas pendientes, listas de procesos relacionados y toda la demás información necesaria para ejecutar el programa [1].

Cuando se ejecuta un proceso, normalmente se ejecuta solo durante un período pequeño de tiempo, antes de terminar o de que se necesite realizar una operación de E/S (acciones necesarias para la transferencia de un conjunto de datos). Así, a medida que se ejecuta un proceso, el proceso va cambiando de estado, el cual se define según la actividad actual de dicho proceso [2].

Cada proceso puede encontrarse en uno de los siguientes estados: nuevo, en ejecución, en espera, preparado o terminado, los cuales podemos observar en Figura 1. El estado de "nuevo" indica que el proceso está siendo creado, mientras que el estado de "en ejecución" indica que las instrucciones del proceso están siendo ejecutadas. Cuando un proceso se encuentra en el estado de "en espera", está esperando a que ocurra un suceso para poder

continuar su ejecución. Por otro lado, cuando se encuentra en el estado de "preparado", está a la espera de que se le asigne un procesador para ejecutarse. Y finalmente, cuando el proceso ha terminado su ejecución, se encuentra en el estado de "terminado". Es importante mencionar que, solamente puede haber un proceso ejecutándose en un procesador en cada instante de tiempo determinado [2].

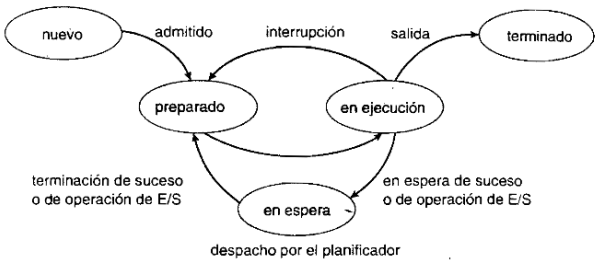

Figura 1. Estados de un proceso [2]

Un proceso en el sistema operativo es representado por un bloque de control de proceso (BCP) o también conocido como bloque de control de tarea. Este bloque contiene múltiples elementos de información que están relacionados con un proceso específico, como el identificador único del proceso (pid), el estado del proceso, la prioridad, el estado de hardware (contador de programa, códigos de condición, punteros de pila, etc.), la información para gestionar la memoria (punteros, tablas, registros), la información de estado del sistema de E/S, la información de contabilidad y planificación, entre otros [3].

En la mayoría de los sistemas operativos, se guarda toda la información sobre cada proceso, además de su espacio de direcciones, en una tabla del sistema operativo. Esta tabla, conocida como tabla de procesos, es una lista o arreglo de estructuras, donde cada estructura corresponde a un proceso que está en ejecución en ese momento. Un proceso suspendido incluye dos elementos principales: su espacio de direcciones, también conocido como imagen de núcleo, y su entrada en la tabla de procesos. Esta entrada almacena diversos elementos necesarios para reiniciar el proceso en el futuro, como el contenido de sus registros y otros datos [1].

Los sistemas operativos también requieren de algún método para crear procesos. En sistemas sencillos o en aquellos diseñados para ejecutar una sola aplicación, es posible conocer todos los procesos que se necesitarán desde el inicio del sistema. Sin embargo, en sistemas de propósito general es necesario contar con una forma de crear y finalizar procesos según sea necesario durante la operación. Los cuatro eventos principales que pueden provocar la creación de procesos son el arranque del sistema, la ejecución de una llamada al sistema para creación de procesos desde otro proceso, una petición de usuario para crear un proceso y el inicio de un trabajo por lotes [1].

En cada uno de los casos, la creación de un proceso se realiza mediante la ejecución de una llamada al sistema de creación de proceso por parte de un proceso existente. El proceso

existente puede ser un proceso de usuario en ejecución, un proceso del sistema iniciado mediante el teclado o ratón, o un proceso del administrador de procesamiento por lotes. En esencia, lo que se hace en todos estos casos es solicitar al sistema operativo que cree un proceso mediante una llamada al sistema, que indica qué programa debe ejecutarse [1].

Como podemos observar en la Figura 3, cuando un proceso genera otros procesos, procesos hijos, y estos a su vez generan más procesos hijos, se forma una estructura de árbol de procesos. En este caso el proceso A crea dos procesos hijos, B y C, mientras que el proceso B crea tres procesos hijos, D, E y F [1].

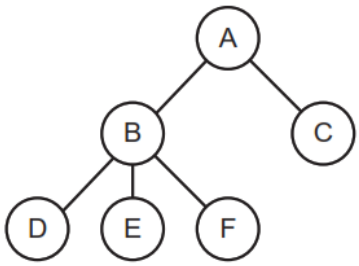

Figura 2. Árbol de procesos [1]

En ciertos sistemas, cuando un proceso da origen a otro, el proceso padre y el proceso hijo se mantienen conectados de cierta manera. El proceso hijo puede generar procesos adicionales por sí mismo, creando así una estructura jerárquica de procesos [1]. De este modo, la comunicación entre procesos se da en ocasiones cuando los procesos relacionados que trabajan para realizar una tarea necesitan comunicarse y sincronizar sus acciones.

Una vez que se crea un proceso, comienza su ejecución y realiza su tarea. Sin embargo, todos los procesos deben terminar en algún momento, esto se debe principalmente a una de estas condiciones: salida normal (voluntaria), salida por error (voluntaria), error fatal (involuntaria) o eliminación por otro proceso (involuntaria).

De esta manera, podemos decir que un proceso es una tarea o actividad que cuenta con su propio programa, una entrada, una salida y un estado. Con el uso de un algoritmo de planificación, múltiples procesos pueden compartir un procesador para determinar cuándo detener una tarea y atender otra.

Hilos

En los sistemas operativos convencionales, generalmente, cada proceso está compuesto por un espacio de direcciones y un único hilo de control. Sin embargo, hay ocasiones en las que es deseable tener varios hilos de control que compartan el mismo espacio de direcciones, y que se ejecuten casi en paralelo, pareciendo procesos independientes [5].

Los hilos, también conocidos como hebras o thread, se refiere a una entidad de trabajo que consta de un contexto de procesador, que contiene el contador de programa y el puntero de

pila, así como su propia área de datos para una pila que permite la ejecución de subrutinas. Los hilos se ejecutan en secuencia y pueden ser interrumpidos para permitir que otro hilo tome el control del procesador.

Los hilos fueron creados con el propósito de combinar la ejecución secuencial con el paralelismo y la capacidad de bloquear las llamadas al sistema [5]. La principal ventaja de utilizar hilos en las aplicaciones es la posibilidad de realizar múltiples tareas al mismo tiempo. Sin embargo, algunas de estas tareas pueden bloquearse en ciertos momentos. Al dividir la aplicación en varios hilos secuenciales que se ejecutan en cuasi-paralelo, se simplifica el modelo de programación y se puede lograr un mejor rendimiento en la ejecución de las tareas [1].

Otra razón para utilizar hilos en lugar de procesos es su rapidez y facilidad de creación y destrucción en comparación con los procesos. En algunos sistemas, la creación de un hilo puede ser de 10 a 100 veces más rápida que la de un proceso. Esto es especialmente útil cuando se necesita ajustar dinámicamente el número de hilos en función de las necesidades de la aplicación. Los hilos pueden mejorar el rendimiento en varias situaciones. Primero, cuando se realizan cálculos intensivos y operaciones de E/S, los hilos pueden superponer estas actividades para mejorar la velocidad de la aplicación. Además, en sistemas con múltiples CPUs, los hilos pueden permitir un verdadero paralelismo. También son más fáciles y rápidos de crear y destruir que los procesos, lo que es útil en situaciones donde el número de hilos necesarios cambia de manera dinámica y rápida [1].

En diversos aspectos, los hilos pueden considerarse como pequeñas unidades de procesamiento independientes. Cada hilo sigue una secuencia de ejecución rigurosa y posee su propio contador de programa y pila para mantener su posición actual. Del mismo modo que los procesos, los hilos comparten la CPU y se alternan en su ejecución mediante el tiempo compartido, a excepción de en sistemas multiprocesador, donde pueden realmente correr en paralelo. Los hilos tienen la capacidad de generar hilos descendientes y pueden ser bloqueados en espera de la finalización de las llamadas al sistema, tal como ocurre con los procesos regulares. Durante el periodo en que un hilo se encuentra bloqueado, otros procesos pueden ser ejecutados en la misma máquina [5].

Los hilos a menudo se denominan procesos ligeros, ya que comparten algunas propiedades con los procesos. El término "multihilamiento" se utiliza para describir la situación en la que se permiten varios hilos en el mismo proceso. Algunas CPUs tienen soporte directo en el hardware para el multihilamiento, lo que permite que las conmutaciones de hilos ocurran en una escala de tiempo muy rápida, incluso en nanosegundos. Cuando se ejecuta un proceso con varios hilos en un sistema que tiene una única CPU, los hilos se turnan para su ejecución.

Del lado izquierdo de la Figura 3, podemos observar tres procesos tradicionales, a cada proceso se le asigna su propio espacio de direcciones y se controla mediante un único hilo.

Mientras que, del lado derecho, vemos un solo proceso con tres hilos de control, los cuales comparten el mismo espacio de direcciones lo que significa que también comparten las mismas variables globales. Una situación adecuada para utilizar tres hilos juntos es cuando realmente forman parte de la misma tarea y están colaborando activa y estrechamente entre ellos [1].

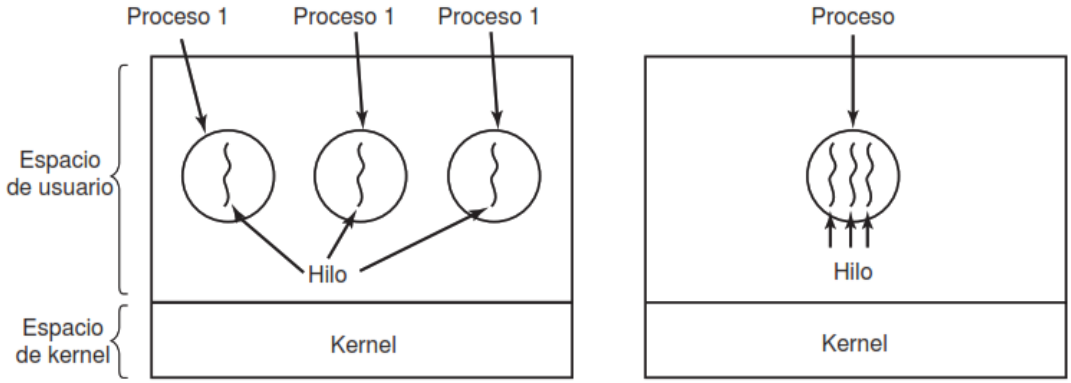

Figura 3. Proceso [1]

Los hilos de un proceso pueden acceder a cualquier dirección de memoria dentro del espacio de direcciones del proceso, lo que significa que un hilo puede leer, escribir o incluso sobrescribir la pila de otro hilo. No hay protección entre los hilos debido a que es imposible y no es necesario, ya que se supone que un proceso es propiedad de un solo usuario que ha creado los hilos para cooperar y no para pelear. Además de compartir el mismo espacio de direcciones, los hilos pueden compartir otros recursos como archivos abiertos, procesos hijos, alarmas y señales [1].

Planteamiento problema

En la actualidad, los sistemas de reservas en línea deben ser capaces de manejar múltiples solicitudes simultáneas para garantizar una distribución equitativa y eficiente de los recursos disponibles. En el caso de la venta de boletos para eventos, la concurrencia en la compra puede ocasionar problemas si no se gestiona adecuadamente el acceso a los recursos compartidos.

El manejo de procesos e hilos como se mencionó anteriormente es una técnica fundamental para administrar concurrencia y sincronización en procesos que requieren acceso a recursos limitados. El uso correcto de hilos permite evitar problemas como condiciones de carrera o inconsistencias en los datos, asegurando una ejecución confiable de las transacciones.

El objetivo de esta práctica es comprender a fondo el funcionamiento de procesos e hilos a través de una implementación practica qué nos permita simular un sistema de reservas de boletos de cine en un entorno multihilo, donde varios clientes intentan comprar boletos simultáneamente, pero con una cantidad limitada de boletos disponibles.

El problema radica en manejar la concurrencia de los hilos que representan a los clientes, para evitar problemas como la compra simultánea de boletos que exceda la cantidad disponible.

Propuesta de solución

Para la solución a este problema tendremos que emplear el uso de procesos e hilos en Java, donde cada cliente será representado por un hilo que intenta comprar boletos de manera concurrente. Para garantizar que el acceso a los boletos disponibles se maneje de manera segura, se utilizará la palabra clave "synchronized" para sincronizar el método de compra, asegurando que solo un cliente pueda modificar el número de boletos disponibles en un momento dado asegurando de esa manera que las transacciones se procedan de manera segura.

Materiales y métodos empleados

Para llevar a cabo la práctica y desarrollar la simulación de concurrencia con procesos e hilos se utilizaron las siguientes herramientas y métodos:

Materiales

1. *Lenguaje de programación:* ***Java***

Se eligió Java debido a su robusto manejo de hilos y sus herramientas integradas para la concurrencia, como la palabra clave synchronized.

2. *Entorno de desarrollo:* ***Visual Studio Code (VS Code)***

Se empleo este entorno de desarrollo debido a que es bastante cómodo trabajar en él, así como también es muy personalizable y bastante útil para desarrollar código.

3. ***JDK (Java Development Kit):***

Se usó el JDK para compilar y ejecutar el programa.

4. *Sistema Operativo:* ***Windows***

La práctica se desarrolló y ejecutó en un sistema operativo Windows, debido a que es el que se emplea mas a menudo y el más cómodo e intuitivo para trabajar.

Métodos

Generación de hilos

Para simular el comportamiento de múltiples clientes intentando adquirir boletos al mismo tiempo, se crearon varios hilos (Thread). Cada cliente es representado por un hilo independiente, el cual ejecuta el método run(). Esta estrategia permite modelar un entorno

concurrencial en el que varias entidades intentan acceder al mismo recurso de manera simultánea.

Sincronización de procesos

Dado que múltiples hilos acceden a un recurso compartido (la cantidad de boletos disponibles), se implementó un bloque sincronizado (synchronized). Esto garantiza que solo un hilo pueda modificar la cantidad de boletos en un momento dado, evitando condiciones de carrera y posibles inconsistencias en la asignación.

Simulación de concurrencia y gestión de tiempos

Para hacer la simulación más realista y evitar que todos los clientes intenten comprar boletos exactamente al mismo tiempo, se utilizó Thread.sleep(). Esta función introduce retardos en la llegada de los clientes, emulando un entorno en el que los usuarios intentan adquirir boletos en diferentes momentos

Gestión del recurso compartido

Antes de permitir que un cliente realice una compra, se verifica la disponibilidad de boletos en el sistema. Este control evita que un cliente pueda adquirir más boletos de los que realmente están disponibles, asegurando que las asignaciones sean consistentes y reflejen la realidad del inventario.

Ejecución y pruebas

Para validar el correcto funcionamiento del sistema, se realizaron múltiples ejecuciones del programa bajo distintos escenarios.

Los resultados obtenidos confirmaron que el uso de hilos y la técnica de sincronización permitieron administrar adecuadamente los recursos compartidos, asegurando la integridad de los datos en un entorno de ejecución concurrente.

Desarrollo

Para poder resolver este problema se tuvo que realizar un sistema de reservas de boletos para un cine utilizando hilos en Java. La finalidad de esta práctica es demostrar la concurrencia en la ejecución de múltiples clientes que intentan comprar boletos simultáneamente, asegurando la correcta administración de los recursos compartidos mediante la sincronización.

Primeramente, realice la clase “Cine” la cual actúa como la entidad central que gestiona la disponibilidad de boletos en el sistema. Cuyo atributo guarda la cantidad de boletos disponibles en el cine, el cual se inicializa con el valor que se pasa al constructor cuando se crea un objeto de la clase “Cine”. Como se muestra en la siguiente figura:

```java
class Cine { // Clase Cine que simula la venta de boletos en un cine
    private int boletosDisponibles;
    // Constructor de la clase Cine
    public Cine(int boletosDisponibles) {
        this.boletosDisponibles = boletosDisponibles;
    }
```

Figura 4. Definición de la clase y el constructor.

Posteriormente el método “comprarBoleto” está implementado de manera sincronizada para prevenir problemas de acceso concurrente y garantizar la integridad de la variable “boletosDisponibles”. Al intentar realizar una compra, el sistema verifica si hay boletos suficientes para satisfacer la solicitud del cliente. Si es así, se actualiza la cantidad de boletos restantes y se muestra un mensaje confirmando la compra. Si no hay suficientes boletos, se informa al cliente que no es posible completar la transacción. Para simular el tiempo de procesamiento de la compra, se utiliza “Thread.sleep(10000)”, introduciendo una pausa de 10 segundos después de cada compra exitosa. Esto permite modelar el tiempo de espera asociado a la operación de compra.

Lo anteriormente mencionado lo podemos observar en el siguiente fragmento de código en la Figura 5.

```java
public void comprarBoleto(String cliente, int cantidad) { // Método para comprar boletos
    boolean successful = false;
    synchronized (this) { // Sincronizar el acceso a la sección crítica
        if (boletosDisponibles >= cantidad) {
            System.out.println(cliente + " está comprando " + cantidad + " boleto(s).");
            boletosDisponibles -= cantidad;
            System.out.println(cliente + " completó la compra. Boletos restantes: " + boletosDisponibles);
            successful = true;
        } else {
            System.out.println(cliente + " intentó comprar " + cantidad + " boleto(s), pero no hay suficientes disponibles.");
        }
    }
    if (successful) { // Simular tiempo de procesamiento de la compra
        try {
            Thread.sleep(millis:10000); // Simular tiempo de procesamiento de la compra
        } catch (InterruptedException e) {
            System.err.println("Thread interrupted: " + e.getMessage());
        }
    }
}
```

Figura 5. Método "comprarBoleto" implementado de manera sincronizada.

Después de esto tenemos a la clase “Cliente” extiende de “Thread” y simula a un cliente comprando boletos en el cine. En su constructor, se recibe un objeto de la clase “Cine”, el nombre del cliente y la cantidad de boletos que desea comprar. Dentro del método “run()”, se invoca el método “comprarBoleto()” de la clase Cine, el cual maneja la lógica de la compra

de boletos, permitiendo que el cliente realice su solicitud de manera concurrente a través del hilo correspondiente.

```java
class Cliente extends Thread { // Clase Cliente que simula a un cliente comprando boletos
    private final Cine cine;
    private final String nombre;
    private final int boletos;

    public Cliente(Cine cine, String nombre, int boletos) { // Constructor de la clase Cliente
        this.cine = cine;
        this.nombre = nombre;
        this.boletos = boletos;
    }

    @Override
    public void run() { // Método run que simula la compra de boletos por parte del cliente
        cine.comprarBoleto(nombre, boletos);
    }
}
```

Figura 6. Clase "Cliente" que simula a varios clientes comprando boletos.

Por ultimo en la clase SistemaReservasCine, dentro del método main(), se crea un objeto de la clase Cine con un total inicial de 20 boletos disponibles. A continuación, se crean seis hilos, cada uno representando a un cliente que solicita una cantidad específica de boletos. Los hilos se inician utilizando el método start(), lo que permite que cada cliente intente comprar boletos de manera concurrente. Para asegurar que el programa espere a que todas las compras se completen antes de mostrar el número final de boletos restantes, se utiliza el método join() en cada hilo, lo que bloquea la ejecución del hilo principal hasta que todos los hilos de los clientes hayan terminado su ejecución.

```java
public class SistemaReservasCine { // Clase principal que simula el sistema de reservas de un cine
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Cine cine = new Cine(boletosDisponibles:20); // Solo hay 20 boletos disponibles

        // Simulación de clientes intentando comprar boletos al mismo tiempo
        Thread cliente1 = new Cliente(cine, nombre:"Cliente 1", boletos:6);
        Thread cliente2 = new Cliente(cine, nombre:"Cliente 2", boletos:4);
        Thread cliente3 = new Cliente(cine, nombre:"Cliente 3", boletos:8);
        Thread cliente4 = new Cliente(cine, nombre:"Cliente 4", boletos:2);
        Thread cliente5 = new Cliente(cine, nombre:"Cliente 5", boletos:5);
        Thread cliente6 = new Cliente(cine, nombre:" Cliente 6", boletos:3);

        // Iniciar los hilos
        cliente1.start();
        cliente2.start();
        cliente3.start();
        cliente4.start();
        cliente5.start();
        cliente6.start();
```

```java
        try { // Esperar a que todos los clientes terminen de comprar boletos
            cliente1.join();
            cliente2.join();
            cliente3.join();
            cliente4.join();
            cliente5.join();
            cliente6.join();
        } catch (InterruptedException e) {
            System.err.println("Thread interrupted: " + e.getMessage());
        }

        System.out.println("Reservas finalizadas. Boletos restantes: " + cine.getBoletosDisponibles());
    }
}
```

Figura 7. Clase "SistemaReservasCine" es la que se encarga de simular el sistema de reservas.

Resultados

La simulación de un sistema de reservas de cine con múltiples clientes concurrentes puso en evidencia la necesidad de coordinar el acceso de los hilos a un recurso común, en este caso, los boletos. Los clientes se representaron como hilos que competían por la compra de boletos al mismo tiempo. La sincronización fue clave para garantizar que solo un hilo pudiera alterar la cantidad de boletos disponibles en cada momento, previniendo así los errores asociados a las condiciones de carrera y asegurando la consistencia de los datos.

A lo largo de la simulación, cada cliente operó como un proceso autónomo que interactuaba con el sistema de reservas, ejecutándose en paralelo con los demás. Sin embargo, el manejo adecuado de la concurrencia permitió que las compras se procesaran sin interferencias ni bloqueos innecesarios.

A continuación, en la siguiente figura se presentan los resultados obtenidos tras la ejecución del programa:

```
PS C:\Users\Atl1God>  & 'C:\Program Files\Java\jdk-23\bin\java.exe' '--enable-
preview' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\Atl1God\App
Data\Local\Temp\vscodesws_58677\jdt_ws\jdt.ls-java-project\bin' 'SistemaReserv
asCine'
Cliente 2 está comprando 4 boleto(s).
Cliente 2 completó la compra. Boletos restantes: 16
 Cliente 6 está comprando 3 boleto(s).
 Cliente 6 completó la compra. Boletos restantes: 13
Cliente 5 está comprando 5 boleto(s).
Cliente 5 completó la compra. Boletos restantes: 8
Cliente 1 está comprando 6 boleto(s).
Cliente 1 completó la compra. Boletos restantes: 2
Cliente 4 está comprando 2 boleto(s).
Cliente 4 completó la compra. Boletos restantes: 0
Cliente 3 intentó comprar 8 boleto(s), pero no hay suficientes disponibles.
Reservas finalizadas. Boletos restantes: 0
PS C:\Users\Atl1God>
```

Figura 8. Resultados mostrados en consola.

Una vez concluida la simulación, el número de boletos restantes coincidió con el valor esperado, lo que validó la correcta sincronización de los hilos y el acceso controlado al

recurso. Además, los intentos de compra de boletos adicionales fueron gestionados de manera adecuada, enviando notificaciones a los clientes sin generar inconsistencias en el sistema.

Conclusión

En esta práctica de procesos e hilos, implementé un sistema que simula la compra de boletos en un cine mediante el uso de hilos en Java. Me ayudó mucho a poder aprender a gestionar el acceso concurrente a los recursos compartidos, en este caso, los boletos disponibles, utilizando la sincronización para evitar condiciones de carrera.

Durante la implementación, fui creando diversas clases como por ejemplo la clase Cine que gestiono la cantidad de boletos y el proceso de compra, asegurando que los clientes no puedan realizar compras que excedan la cantidad disponible. También pude comprender cómo es que funciona la palabra clave synchronized, la cual me sirvió para asegurar que solo un cliente pueda acceder a la sección crítica del código en un momento dado, lo que evita que varios hilos modifiquen el número de boletos simultáneamente.

Posteriormente al realizar la clase Cliente no tuve muchas complicaciones, ya que solo se trataba de simula la acción de un cliente comprando boletos, lo que se ejecuta en hilos independientes. Aquí es donde aprendí a que cada cliente tenía que comprar una cantidad de boletos, y la sincronización se aseguraba de que las compras sean procesadas de manera correcta y segura, sin que se produzcan errores debido a la modificación concurrente del número de boletos disponibles.

El resultado final me mostró cómo la sincronización entre hilos es esencial en sistemas distribuidos o de múltiples procesos, donde varios usuarios pueden acceder al mismo recurso.

Otro punto importante es que la ejecución del programa me mostró que el número total de boletos disponibles fue correctamente actualizado, y los clientes recibieron el mensaje adecuado según la disponibilidad de boletos en el momento de su solicitud.

Por último, cabe mencionar, esta práctica me permitió entender la importancia de la sincronización en la programación concurrente y cómo se pueden manejar los recursos compartidos en un entorno con múltiples procesos o hilos de manera segura y eficiente.

Referencias

[1] A. Tanenbaum, Sistemas Operativos Modernos, México: Pearson Educación, 2009.

[2] A. Silberschatz, P. Galvin y G. Gagne, Fundamentos de Sistemas Operativos, Madrid: McGraw-Hill / Interamericana, 2006.

[3] J. Aranda, M. Canto, J. de la Cruz, S. Dormido y C. Mañoso, Sistemas Operativos: Teoriía y Problemas, Madrid: Sanz y Torres, S.L., 2002.

[4] W. Stallings, Sistemas operativos Aspectos internos y principios de diseño, Madrid: Pearson Educación, 2005.

[5] A. Tanenbaum, Sistemas Operativos Distribuidos, México: Prentice Hall , 1996.

Anexos

Código Fuente

```
/*
 * Nombre: Cardoso Osorio Atl Yosafat
 * Grupo: 7CM1
 * Materia: Sistemas Distribuidos
 * Practica 1: Procesos e Hilos
 * Fecha: 24 de Febrero del 2025
 */
class Cine { // Clase Cine que simula la venta de boletos en un
cine
    private int boletosDisponibles;
    // Constructor de la clase Cine
    public Cine(int boletosDisponibles) {
        this.boletosDisponibles = boletosDisponibles;
    }

    public void comprarBoleto(String cliente, int cantidad) { //
Método para comprar boletos
        boolean successful = false;
```

```
        synchronized (this) { // Sincronizar el acceso a la
sección crítica
            if (boletosDisponibles >= cantidad) {
                System.out.println(cliente + " está comprando " +
cantidad + " boleto(s).");
                boletosDisponibles -= cantidad;
                System.out.println(cliente + " completó la compra.
Boletos restantes: " + boletosDisponibles);
                successful = true;
            } else {
                System.out.println(cliente + " intentó comprar " +
cantidad + " boleto(s), pero no hay suficientes disponibles.");
            }
        }
        if (successful) { // Simular tiempo de procesamiento de la
compra
            try {
                Thread.sleep(10000); // Simular tiempo de
procesamiento de la compra
            } catch (InterruptedException e) {
                System.err.println("Thread interrupted: " +
e.getMessage());
            }
        }
    }

    public int getBoletosDisponibles() { // Método para obtener
los boletos disponibles
        return boletosDisponibles;
    }
}

class Cliente extends Thread { // Clase Cliente que simula a un
cliente comprando boletos
    private final Cine cine;
    private final String nombre;
    private final int boletos;
```

```
    public Cliente(Cine cine, String nombre, int boletos) { //
Constructor de la clase Cliente
        this.cine = cine;
        this.nombre = nombre;
        this.boletos = boletos;
    }

    @Override
    public void run() { // Método run que simula la compra de
boletos por parte del cliente
        cine.comprarBoleto(nombre, boletos);
    }
}

public class SistemaReservasCine { // Clase principal que simula
el sistema de reservas de un cine
    public static void main(String[] args) {
        Cine cine = new Cine(20); // Solo hay 20 boletos
disponibles

        // Simulación de clientes intentando comprar boletos al
mismo tiempo
        Thread cliente1 = new Cliente(cine, "Cliente 1", 6);
        Thread cliente2 = new Cliente(cine, "Cliente 2", 4);
        Thread cliente3 = new Cliente(cine, "Cliente 3", 8);
        Thread cliente4 = new Cliente(cine, "Cliente 4", 2);
        Thread cliente5 = new Cliente(cine, "Cliente 5", 5);
        Thread cliente6 = new Cliente(cine, " Cliente 6", 3);

        // Iniciar los hilos
        cliente1.start();
        cliente2.start();
        cliente3.start();
        cliente4.start();
        cliente5.start();
        cliente6.start();

        try { // Esperar a que todos los clientes terminen de
comprar boletos
```

```
            cliente1.join();
            cliente2.join();
            cliente3.join();
            cliente4.join();
            cliente5.join();
            cliente6.join();
        } catch (InterruptedException e) {
            System.err.println("Thread interrupted: " +
e.getMessage());
        }

        System.out.println("Reservas finalizadas. Boletos
restantes: " + cine.getBoletosDisponibles());
    }
}
```