Simulación

Peluquerías Paquita

DAVANTE | MEDAC

Carlos Piernas Jiménez

Programación de servicios y procesos

José Ferrer Grau

30-11-2025

Índice

Introducción

Como proyecto final de trimestre se nos ha solicitado el desarrollo de una simulación de una peluquería donde tres peluqueras trabajan sobre quinientos clientes, los cuales van entrando de forma ordenada en las cuatro zonas que tiene esta peluquería: lavado, corte, tinte y peinado. Todo esto se ha hecho mediante hilos para una simulación concurrente sin espera activa, utilizando monitores para un funcionamiento adecuado.

Clase Zona

La clase Zona simula los puestos de servicio (Lavado, Corte, Tinte, Peinado) y es la región crítica donde la exclusión mutua es esencial.

Método synchronized ocuparPuestoCliente

Este método, ejecutado por el hilo Clientes, implementa la lógica de espera pasiva para el cliente:
El bucle while (puestoOcupado) fuerza al cliente a llamar a wait() si la zona está ocupada. Esto libera el monitor y coloca al cliente en un estado de espera pasiva hasta que la zona se desocupe.
Una vez que la zona está libre, el cliente la ocupa (puestoOcupado = true).
Inmediatamente después de ocupar el puesto, el cliente llama de nuevo a wait(). El cliente permanece bloqueado intencionadamente en este punto hasta que la Peluquera lo despierte, garantizando que no abandone el recurso antes de recibir el servicio.

```
// Método invocado por el Cliente para sentarse en la zona.
//El cliente espera si la zona está ocupada o hasta que el servicio finalice.
public synchronized void ocuparPuestoCliente(Clientes cli) {
    try {
        //El cliente espera hasta que la zona esté libre
        while (puestoOcupado) {
            wait();
        }

        //Ocupa la zona
        clienteActual = cli;
        puestoOcupado = true;

        System.out.println(cli.getNombre() + " se sienta en la Zona de " + nombreZona);
        System.out.println("**********************");

        // El cliente espera a que la peluquera lo atienda y finalice el servicio
        wait();

    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
```

Método synchronized intentarAtender(Peluqueras pelu)

Este método es ejecutado por el hilo Peluqueras para realizar el servicio, siendo ella la que impone los tiempos del trabajo.
La peluquera primero verifica que puestoOcupado sea true.
Se simula el servicio con un Thread.sleep() de tiempo aleatorio.
Al finalizar, la peluquera libera la zona (puestoOcupado = false) y llama a notifyAll(). Esta llamada es la señal de sincronización final que despierta al hilo Cliente bloqueado en su wait() interno.

```
public synchronized boolean intentarAtender(Peluqueras pelu) {
    // Solo se atiende si hay un cliente sentado
    if (!puestoOcupado) {
        return false;
    }
    try {
        // Inicio de la atención
        System.out.println(pelu.getNombre() + " empieza a atender a " + clienteActual.getNombre() +
        " en la Zona de " + nombreZona);
        System.out.println("**********************");
        // Simulación del tiempo de servicio
        Thread.sleep(ThreadLocalRandom.current().nextInt(500, 1500));

        System.out.println(pelu.getNombre() + " termina de atender a " + clienteActual.getNombre() + " en " + nombreZona);
        System.out.println("**********************");

        // El servicio termina y la zona queda libre
        puestoOcupado = false;
        clienteActual = null;

        // Notifica al cliente para que continúe su camino
        notifyAll();
        return true;

    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        // En caso de interrupción durante el servicio, liberamos la zona
        if (puestoOcupado) {
            puestoOcupado = false;
            clienteActual = null;
            notifyAll();
        }
        return true;
    }
}
```

Clase Cliente

La clase Clientes extiende de Thread y modela el ciclo de vida completo del usuario en el sistema.
El método run() define la ruta crítica que el cliente debe seguir obligatoriamente: Lavado, Corte, Tinte, y Peinado.
El cliente se comporta de manera pasiva durante la espera y la atención, activándose únicamente cuando la zona queda libre y el servicio ha finalizado, momento en el cual pasa a la siguiente zona.

```java
@Override
public void run() {
    // El cliente pasa secuencialmente por las 4 zonas en el orden establecido
    transitarYEntrarZona(zLavado);
    transitarYEntrarZona(zCorte);
    transitarYEntrarZona(zTinte);
    transitarYEntrarZona(zPeinado);

    // Una vez pasado por todas las zonas, el cliente se considera atendido y se marcha
    System.out.println(nombre + " ha completado su recorrido y se retira.");
    System.out.println("**********************");
}
//Simula el desplazamiento y la entrada a una zona de servicio.
public void transitarYEntrarZona(Zona zona) {
    // Simula un tiempo de desplazamiento aleatorio entre zonas
    try {
        Thread.sleep(ThreadLocalRandom.current().nextInt(20, 80));
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    // El cliente intenta sentarse y espera a ser atendido
    zona.ocuparPuestoCliente(this);
}
```

Clase Peluquera

La clase Peluqueras representa al actor activo del programa, cuyo diseño le permite reaccionar dinámicamente al estado de la peluquería.

El bucle principal (run()) se mantiene activo mediante la condición: while (gJ.verificarJornadaActiva() || hayClientesEnPuestos()). Esta condición garantiza la fase de limpieza, obligando a la peluquera a finalizar todos los servicios ya iniciados incluso si la jornada ha terminado lógicamente. La peluquera trabaja comprobando las zonas en un orden inverso al flujo del cliente (Peinado → Lavado), lo que evita cuellos de botella y prioriza la finalización de los servicios.

```java
public void run() {
    while (gJ.verificarJornadaActiva() || hayClientesEnPuestos()) {
        boolean atencionExitosa = false;
        for (Zona zona : puestosDeTrabajo) {

            if (zona.hayClienteOcupandoPuesto()) {
                if (zona.intentarAtender(this)) {
                    atencionExitosa = true;
                    if (zona.nombreZona.equals("Peinado")) {
                        gJ.registrarAtencion(nombrePelu);
                    }
                    break;
                }
            }
        }
        if (atencionExitosa) {
            clientesAtendidos++;

            if (clientesAtendidos >= nuevaSiesta) {
                iniciarSiesta();
            }
        } else {
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
    System.out.println(nombrePelu + " terminó su jornada.");
}
```

Sistema de Descanso Dinámico (Extra 10)

Se ha implementado un sistema de descanso dinámico que evita que el límite de siesta sea un número estático.

Al inicio de la jornada y al despertar de una siesta, la peluquera llama a nuevoLimiteSiesta(). Este método genera un valor aleatorio entre 8 y 15 clientes a atender antes del próximo descanso.

La variable interna clientesAtendidos rastrea el progreso. Una vez que clientesAtendidos es mayor o igual a nuevaSiesta, la peluquera entra en iniciarSiesta(). Al terminar el descanso, se llama nuevamente a nuevoLimiteSiesta() para establecer un nuevo objetivo aleatorio

```java
//Genera un nuevo límite aleatorio (8-15) usando Math.random(), resetea el contador y lo imprime.
private void nuevoLimiteSiesta() {
    nuevaSiesta = (int) (Math.random() * (15 - 8 + 1)) + 8;
    clientesAtendidos = 0;
    System.out.println(nombrePelu + " decide que va a atender " + nuevaSiesta + " antes de la próxima siesta");
    System.out.println("**********************");
}
```

Clase GestorJornada

Esta clase actúa como el gestor central para el control de la jornada. El método registrarAtencion() es crítico. Cuando se atiende al cliente final, este método llama a notifyAll() para despertar al hilo principal bloqueado en esperarCierreJornada(), lo que permite una finalización limpia.

```java
/**
 * Incrementa los contadores de clientes atendidos (total y por peluquera).
 */
public synchronized int registrarAtencion(String nombrePelu) {
    clientesAtendidosTotal++;

    atendidosPorPeluquera.put(nombrePelu, atendidosPorPeluquera.getOrDefault(nombrePelu, 0) + 1);

    if (clientesAtendidosTotal == totalClientes) {
        finalizarJornada();
        notifyAll();
    }

    return clientesAtendidosTotal;
}
```

```java
 * Comprueba si la peluquera ha alcanzado el umbral para tomar una siesta.
 */
public synchronized boolean verificarNecesidadSiesta(String nombrePelu) {
    int atendidos = atendidosPorPeluquera.getOrDefault(nombrePelu, 0);
    return atendidos > 0 && (atendidos % clientesSiesta == 0);
}

/**
 * Establece el estado de la jornada a inactiva (cerrada).
 */
public synchronized void finalizarJornada() {
    if (jornadaActiva) {
        jornadaActiva = false;
    }
}

/**
 * Comprueba el estado actual de la peluquería.
 */
public synchronized boolean verificarJornadaActiva() {
    return jornadaActiva;
}

/**
 * Devuelve el total de clientes atendidos hasta el momento.
 */
public synchronized int getClientesAtendidosTotal() {
    return clientesAtendidosTotal;
}

/**
 * Permite que el hilo principal espere a que todos los clientes sean
 * atendidos.
 */
public synchronized void esperarCierreJornada() throws InterruptedException {
    while (clientesAtendidosTotal < totalClientes) {
        wait();
    }
}
```

Clase Peluqueria

El método main actúa como el punto de entrada, configurando y preparando el inicio de la simulación.
El hilo principal invoca gJ.esperarCierreJornada(), lo que pone al hilo main en estado de espera pasiva hasta que el monitor GestorJornada le notifica que todos los servicios han sido completados.

Configuración Dinámica (Extra 5)

Mediante el uso de la clase Scanner, se proporciona al usuario la posibilidad de definir la carga de trabajo (número de clientes) y el número de peluqueras al inicio de la ejecución, adaptando la simulación sin necesidad de recompilar el código.