Universidad Nacional de Córdoba

PROGRAMACIÓN CONCURRENTE 2025

Facultad de Cs Exactas, Fís y Naturales

TRABAJO PRÁCTICO N°1

GRUPO: “ThreadStorm”

INTEGRANTES:

Benavides, María Candela - 45559700

Castellano, Juana - 45408569

Cechich. Federico - 43372534

Fariñas, Rafael - 52096795

Salinas, Joaquín Alejandro - 45211874

PROFESORES:

Ing. Luis Orlando Ventre

Ing. Mauricio Ludemann

Aspirante Adscripto: Ing. Agustin Carra

INTRODUCCIÓN

En este informe se busca diseñar, implementar y analizar un sistema concurrente para la gestión de entregas de una empresa de logística que realiza envíos de productos comprados en línea. Este sistema cuenta con cuatro etapas principales: preparación de pedido, despacho de pedido, entrega al cliente y verificación final.

Dentro de los desafíos para realizar el programa se encontraba el de modelar un entorno donde múltiples hilos pudieran ejecutarse de forma simultánea, evitando problemas de concurrencia, como deadlocks, inconsistencias en los recursos compartidos, etc.

En este trabajo no solo se buscó cumplir con el funcionamiento del programa si no también pudimos ver y entender la importancia del diseño concurrente en los tiempos de ejecución y el rendimiento. Por ello se realizan los registros y el análisis de resultados para entender el sistema bajo distintas condiciones.

DESARROLLO

Funcionamiento general

Este programa se basa en la colaboración entre clases para simular un sistema concurrente de almacenamiento y gestión de pedidos. La clase principal, Main, inicia el programa y crea instancias de otras clases fundamentales como SistemaAlmacenamiento y RegistrodePedidos, además de lanzar hilos que ejecutan tareas concurrentes.

En este sentido la clase SistemaAlmacenamiento, administra los casilleros y es utilizado por clases como Preparación y Despacho, que ocupan y liberan casilleros respectivamente. RegistrodePedidos mantiene listas compartidas de pedidos y es empleado por varias clases que los mueven entre distintos estados, desde la preparación hasta la verificación final.

Las clases Preparación, Despacho, Entrega y VerificacionFinal operan con hilos concurrentes y colaboran con el sistema de almacenamiento y el registro de pedidos para gestionar el flujo de los pedidos. Cada una de estas clases cumple un rol específico en el modelo productor-consumidor: Preparación genera pedidos y los asigna a casilleros, Despacho los mueve a tránsito y libera espacio, Entrega los marca como entregados, y VerificacionFinal los verifica.

De esta manera, el objeto Pedido se encarga de transitar por todas las etapas y es manipulado por cada una de estas clases, mientras que EstadoPedido define los distintos estados que puede tener un pedido.

En resumen la estructura general del programa sigue un esquema jerárquico claro: Main coordina todo el sistema, SistemaAlmacenamiento administra los casilleros, RegistrodePedidos organiza las listas de pedidos y los distintos hilos manejan el flujo de los pedidos de forma concurrente. En esencia, el programa

sigue un modelo productor-consumidor con múltiples etapas, donde las clases colaboran para gestionar pedidos de manera concurrente.

Tiempos de Proceso

El sistema debe procesar un total de 500 pedidos a lo largo de su ejecución, asegurando que el programa completo tenga una duración aproximada de entre 15 y 30 segundos.

Dado que los cuatro procesos principales -*Preparación, Despacho, Entrega y Verificación*- trabajan de manera paralela (ejecutándose al mismo tiempo en hilos independientes), es necesario ver cuánto tiempo debe tardar cada uno en procesar un solo pedido para que, en conjunto, se logre cumplir con el tiempo requerido.

En consecuencia, el tiempo de ejecución de cada proceso se define como la cantidad de pedidos totales divididos el tiempo de ejecución total del programa, el cual se encuentra entre 15 y 30 segundos; por lo tanto cada hilo tarda un tiempo de ejecución distinto para los diferentes procesos, por ejemplo los tres hilos que se encargan del proceso de preparación y los tres hilos que ocupan del proceso de entrega debe tardar entre 90 y 180 milisegundos para realizar cada proceso, mientras que los 2 hilos de despacho y los 2 hilos de verificación deben tardar entre 60 y 120 milisegundos para que realicen el proceso.

(Tiempo Total * n° de hilos) / (cant. de pedidos) = tiempo de iteración

Esto se realiza mediante (Thread.sleep()), este método se emplea en este programa para generar pausas controladas en la ejecución de los hilos. Esto cumple varios objetivos:

1. **Simulación de tiempos reales** En sistemas concurrentes, los procesos de preparación, despacho y entrega de pedidos no ocurren de inmediato en la

vida real. Thread.sleep().introduce retrasos artificiales para simular el tiempo requerido en cada etapa, como:

- · Preparar un pedido (Preparación).
- · Despachar un pedido (Despacho).
- · Entregar un pedido (Entrega).

2. **Reducción de la carga del procesador** Sin pausas, los hilos podrían ejecutarse en un bucle constante, consumiendo innecesariamente recursos del procesador. Thread.sleep().suspende temporalmente el hilo actual, permitiendo que otros hilos se ejecuten y mejorando la eficiencia del sistema concurrente.

3. **Coordinación entre hilos** Las pausas evitan que un solo hilo monopolice la ejecución, permitiendo que otros realicen sus tareas. Por ejemplo:

   - · En **Preparación**, el hilo se detiene brevemente tras preparar un pedido, facilitando que otros hilos procesen los pedidos pendientes en la clase **Despacho** .
   - · En **Despacho** y **Entrega**, los retrasos ayudan a mover los pedidos entre las listas de manera ordenada.

4. **Simulación de eventos aleatorios** En **Despacho**, Thread.sleep().también simula un retraso antes de agregar un "pedido poison" (pedido especial que indica la finalización del procesamiento). Esto garantiza que todos los pedidos normales sean gestionados antes de que el sistema reciba la señal de cierre Thread.sleep().

Consideraciones de concurrencia

- Se hizo uso de *AtomicInteger*, el cual nos permite realizar operaciones seguras entre múltiples hilos sobre un enteros.

- Implementamos el mecanismo de sincronización *synchronized*, permitiendo asegurar que sólo un hilo acceda a un bloque de código o método a la vez, protegiendo así las secciones críticas.
- Se hizo uso del mecanismo de sincronización *ReentrantsLock*. Este mecanismo es una alternativa más flexible que el mecanismo synchronized; permite un mayor control sobre el acceso a los recursos compartidos.
- Se utilizaron los métodos wait() y notify() para coordinar la ejecución entre hilos, permitiendo que un hilo espere hasta que otro lo despierte al cumplirse cierta condición.

Algunas consideraciones generales que se tuvieron en cuenta durante el trabajo fueron:

- Evitar condiciones de carrera, de esta manera se asegura el acceso a los recursos compartidos mediante mecanismos de sincronización adecuados.
- Prevenir deadlocks mediante el uso de los bloqueos.
- Facilitar la comunicación entre hilos utilizando estructuras seguras, como por ejemplo variables atómicas.

Análisis de tiempos de ejecución

Para la configuración de los tiempos de ejecución, se calculó un tiempo de procesamiento ideal por pedido, con el objetivo de realizar pruebas cuya duración total oscile entre 15 y 30 segundos. A continuación se presenta la tabla con los valores calculados y los resultados obtenidos:

Tiempo programa ideal [s]	Pedidos	Tiempo ideal por pedido [ms]	Tiempo real por pedido [ms]	Overhead por pedido	Tiempo total Programa [ms]	Overhead
30	500	60	62,132	1,73%	31066	3,45%
15	500	30	32,084	3,56%	16042	7,13%
22	500	44	47,206	2,44%	23603	4,88%

Dado que cada pedido se procesa casi en el tiempo predeterminado, podemos afirmar que los hilos trabajan en paralelo de forma eficiente y que no existen cuellos de botella graves en el acceso a los recursos críticos. La sobrecarga de sincronización es mínima y no impacta significativamente el rendimiento global.

Si el programa llegará a comportarse de forma más secuencial por secciones críticas demasiado largas, el tiempo real por pedido aumentaría y, para mantener el tiempo total de ejecución dentro de la consigna, habría que reducir el tiempo de procesamiento por pedido (ajustar los sleepTime).

En síntesis, el sistema logra prácticamente el “paralelismo puro” esperado, con una sobrecarga de sincronización máxima al 7 %. Por tanto, el acceso a recursos críticos es gestionado con éxito.

CONCLUSIÓN

A lo largo del desarrollo del primer trabajo práctico se logró diseñar e implementar un sistema concurrente para la gestión de entregas de una empresa logística. Dado que las distintas etapas del sistema -*Preparación, Despacho, Entrega y Verificación Final*- se ejecutan de manera simultánea mediante múltiples hilos, fue necesario aplicar mecanismo de control para la coherencia de los datos compartidos entre procesos.

Pudimos observar y manejar problemas de concurrencia, dado que las etapas del proceso se ejecutan en forma paralela con distintos hilos funcionando al mismo tiempo y para evitar estos errores, se utilizaron herramientas como AtomicINteger, synchronized, ReentrantsLock.

Además, se empleó la función Thread.sleep() para simular los tiempos de procesamiento de cada etapa. Esto permitió que la duración de cada operación

sea de forma controlada, asegurando que el tiempo total de ejecución del sistema se mantuviera dentro del rango pedido.

El programa pone en práctica conceptos teóricos fundamentales de la programación concurrente como la sincronización, la coordinación entre hilos y la importancia del diseño de los tiempos de espera, así como también reforzar el valor de una arquitectura organizada para gestionar procesos paralelos de forma eficiente y segura.