Facultad de
Ciencias Exactas
Físicas y Naturales

Universidad
Nacional
de Córdoba

Programación Concurrente

Informe Trabajo Práctico N° 1

Nombre del grupo: CONCURRENT LIFE

Integrantes:

Bernardi Mateo

Fassi Fernández Esteban

Ledesma Ignacio

Madrid Santiago

Robles Karen Yesica

Profesores:

Ing. Ventre Luis Orlando

Ing. Ludemann Mauricio

2024

ÍNDICE

1. Lineamientos

1.1. Enunciado

En un Sistema de reservas de vuelos, se necesita implementar una funcionalidad para gestionar las reservas de asientos de manera concurrente. Esta funcionalidad consta de varios procesos que deben ejecutarse de forma simultánea, teniendo en cuenta los siguientes lineamientos de diseño.

El sistema debe manejar la reserva de asientos de un avión, representado por una matriz de asientos donde cada asiento puede estar ocupado, libre o descartado. Además, se deben mantener cuatro registros distintos para las reservas:

- Reservas pendientes de pago.
- Reservas confirmadas.
- Reservas canceladas.
- Reservas verificadas.

El funcionamiento del sistema posee **cuatro etapas**:

Proceso de Reserva: Este proceso se encarga de recibir las solicitudes de reserva de los usuarios. Se tienen tres hilos que ejecutan este proceso. Cada hilo intenta reservar un asiento aleatorio en la matriz, verificando que esté disponible. Si el asiento no está libre, el hilo debe buscar otro asiento que sí lo esté. Una vez reservado el asiento, el mismo se marca como ocupado y se registra la reserva pendiente en el registro de reservas pendientes

Proceso de Pago: Este proceso es ejecutado por dos hilos, y se encarga de verificar el pago de las reservas pendientes. Cada hilo toma una reserva aleatoria del registro de reservas pendientes y realiza una verificación de pago. Se establece una probabilidad del 90% de que el pago sea aprobado y un 10% de que sea rechazado. Si el pago es aprobado, la reserva se elimina del registro de pendientes, y se agrega al registro de reservas confirmadas; de lo contrario, el asiento pasa a estado descartado mientras que la reserva se marca como cancelada, se elimina del registro de pendientes y se agrega al registro de reservas canceladas.

Proceso de Cancelación/Validación: Si un usuario decide cancelar su reserva, se pasa al proceso de cancelación. Tres hilos se encargan de cancelar las reservas. Cada hilo selecciona una reserva aleatoria de las reservas confirmadas y la cancela con una probabilidad del 10%. Si la reserva es cancelada, se elimina del registro de reservas confirmadas y se agrega al registro de **reservas canceladas** mientras que el asiento pasa a estado descartado. Si la reserva no es cancelada, la misma se marca como "checked".

Proceso de Verificación: Al finalizar la ejecución, se debe verificar el estado final

de la matriz de asientos y los registros de reservas para asegurar que las operaciones se hayan realizado correctamente. Este proceso selecciona de manera aleatoria una reserva del registro de reservas confirmadas. Para cada reserva marcada como “checked”, se debe eliminar del registro de reservas confirmadas y se debe insertar en el registro de **reservas verificadas**. Este proceso es ejecutado por dos hilos.

1.2. Consideraciones

- Cada proceso tiene una demora fija por proceso (cada iteración), pero distinta entre procesos, a ser definida por el equipo.
- Cada asiento debe ser accesible por un solo hilo a la vez para evitar conflictos de reserva simultanea
- Cada reserva debe ser revisada por un solo hilo a la vez (independientemente del proceso)
- Cada reserva puede ser aprobada, confirmada, cancelada o verificada solo una vez.
- Los procesos de reserva, pago, cancelación y verificación deben ejecutarse de forma recurrente para simular un entorno de reservas realista.
- Los tiempos de espera para realizar cada operación deben ser aleatorios y configurables por el grupo.
- Al iniciar el programa, todos los hilos deben ser lanzados para que comiencen su ejecución.
- El sistema debe contar con un LOG con fines estadísticos, el cual registre cada 200 milisegundos en un archivo:
  - Cantidad de reservas canceladas
  - Cantidad de reservas aprobadas (verificadas)

  Además, al finalizar, el log debe imprimir la ocupación final del vuelo y el tiempo total que demoró el programa

1.3. Ejercicios

1) Hacer un diagrama de clases que modele el sistema de datos con TODOS los actores y partes
2) Hacer un diagrama de secuencias que modele las interacciones del sistema
3) El vuelo a modelar tiene una capacidad de 186 pasajeros.
4) Se deben configurar los tiempos de los procesos de modo tal que el programa no demore más de 45 segundos.
5) Hacer un análisis analítico detallado de los tiempos que el programa demora. Luego contrastarlo con múltiples ejecuciones obteniendo las conclusiones pertinentes.
6) El grupo debe poder explicar los motivos de los resultados obtenidos, y los tiempos del sistema
7) Debe haber una clase Main que, al correrla, inicie los hilos.

2. Desarrollo

2.1. Modelado Inicial

Antes de adentrarnos en el desarrollo del código de nuestro programa, como equipo tomamos la decisión de empezar dibujando distintos modelos en papel y lápiz que describieran aproximadamente el funcionamiento que nosotros creíamos que debía seguir el programa, dándonos una mejor idea de qué es exactamente lo que había que hacer. Esta estrategia nos permitió explorar y anticipar una serie de desafíos potenciales, así como identificar conceptos clave que requerían un repaso detenido.

Si bien este fue el primer paso, también modelamos durante todo el desarrollo del programa. No solo modificamos los bocetos que ya habíamos hecho para agregar nuevas ideas antes de implementarlas, con el objetivo de tratar de predecir su funcionamiento, sino que modelar diferentes partes del código también nos permitió identificar mejor donde podía llegar a haber problemas de concurrencia o donde es que el código fallaba.

De esta forma, el modelado del programa fue una actividad frecuente y muy útil durante todo del desarrollo del proyecto, facilitando su desarrollo y permitiéndonos identificar y afrontar mejor los desafíos que fueron surgiendo. Modelar también permitió que pudiéramos comunicar mejor las ideas que cada uno tenía, algo muy importante a la hora de realizar trabajos en equipo.

2.2. Código

El segundo paso a seguir fue el desarrollo de los diferentes procesos que nuestro programa debía tener. Como fue nuestro primer acercamiento a la concurrencia, los primeros dos procesos fueron los más difíciles de hacer.

En esta etapa comprendimos realmente como se veía un problema de concurrencia y qué resultados negativos provocaba. No solo era importante asegurar la exclusión mutua en determinadas partes, sino también podía llegar a importar el orden en el que se ejecutaban los distintos métodos sincronizados.

Tuvimos que ir identificando y reordenando cada sección critica de interés, es decir, los lugares donde debíamos asegurar que hubiese exclusión mutua entre los recursos compartidos por cada hilo. Empezamos definiendo secciones criticas abarcativas, y a medida que fuimos distinguiendo los recursos que cada hilo compartía, fuimos optimizando el programa, reduciendo el tamaño de dichas secciones hasta dejarlas lo más pequeñas posibles.

Igualmente, determinar las secciones críticas no fue el único desafío. Muchas veces sucedía que las secciones criticas se ejecutaban en un orden que hacía que nuestro programa entrara en deadlock por ejemplo, o también sucedía que algunas variables mostraban un resultado que no era el esperado.

Aquí empezó a tomar vital importancia el modelado y tener un log disponible que reflejara el estado de cada hilo cada cierto periodo de tiempo. El log nos permitía identificar donde se daba el problema, mientras que el modelado nos permitía entender por qué y cómo sucedía el problema. Si en el log veíamos que un hilo

terminaba antes de lo planeado, o no terminaba, analizábamos los contenidos de los arreglos y eso nos permitía identificar donde podía llegar a estar el problema. La mayoría de las veces, la solución fue fácil y posible gracias a estas herramientas, pero otras veces tuvimos que recurrir al análisis de cada línea del código, el pensamiento crítico y la experiencia.

2.3. Ejecución y Optimización

Una vez que terminamos todas las clases necesarias y al obtener resultados satisfactorios tras ejecutar el programa, decidimos intentar optimizar el código lo máximo posible.

En el proceso, pudimos solucionar algunos inconvenientes, mejoramos los tiempos de ejecución e identificamos distintos aspectos del funcionamiento del programa que fueron útiles para otras instancias del proyecto, como determinar los tiempos óptimos para cada proceso.

2.4. Diagramas de Clases y Secuencia

Una vez que el código era de nuestro agrado y no se nos ocurria nada más por mejorar, decidimos empezar a hacer los Diagramas de Clases y Secuencias definitivos utilizando la herramienta Visual Paradigm. Ambos diagramas son adjuntados dentro de la carpeta “docs” de “TP1 CONCURRENT LIFE.rar”.

Como bien dijimos antes, el modelado fue algo frecuente durante el proyecto, así que solo tuvimos que combinar y cambiar algunas cosas de los distintos diagramas que fuimos haciendo en papel.

2.5. Decisiones de Diseño

Dentro del código de nuestro programa se explica por qué se tomaron diferentes decisiones en casos puntuales, las cuales suelen tener su justificación en intentar reducir el tiempo de ejecución al mínimo y evitar que el procesador de la computadora utilice recursos o realice tareas de forma innecesaria. Ejemplos de estas decisiones son algunos while’s e if’s que optimizan el programa. También hay otras decisiones que favorecen la versatilidad o la facilidad de ir cambiando o de acceder a diferentes variables de interés. Un ejemplo de esto es tener una clase específica para la matriz de objetos, donde el objeto que se cree a partir de esa clase va a ser el mismo para todos los threads que necesiten usar la matriz por algo.

Una decisión interesante fue la de hacer que el proceso que genera el archivo .log se ejecutara de forma secuencial en el main o como un proceso más, asignándole un hilo. Nosotros optamos por asignarle un hilo propio, y esto es algo muy positivo porque, además de ahorrarnos diferentes problemas que tuvimos cuando se planteó inicialmente de forma secuencial, nos permite iniciar al log antes que iniciar los threads, pudiendo visualizar el estado de los Threads mientras están siendo creados e iniciados.

La decisión más importante del programa fue la de hacer una clase seatRegisters que englobara a todos los registros que se utilizan en cada proceso. El objeto creado de esta clase va a ser pasado como parámetro a todos los threads y va a poseer a todos los recursos compartidos y secciones criticas relacionadas

con los problemas de concurrencia que tuvimos que afrontar. En primera instancia, hicimos esta clase porque teníamos problemas para pasar uno o más arrays a cada thread y que cada thread fuese capaz de modificar a los arrays que compartía con otros, por eso decidimos hacer una sola clase que contiene todos los arreglos. A su vez, esta clase va a ser la encargado de administrar los recursos compartidos, y los threads solo van a dar las ordenes de cómo administrarlos. Por estas razones, esta clase es la más larga, la más compleja e incluso la más importante de todas, porque es en ella donde se administra la concurrencia de los hilos.

2.6. Funcionamiento

2.6.1. Clase Main

Implementación de la **clase Main** donde:

- Se definen las siguientes variables:
  - ➢ amountReservators=3: Define el número de hilos (threads) para la reserva de asientos del avion.
  - ➢ amountPaymentators = 2: Define el número de hilos que se encarga de verificar el pago de las reservas.
  - ➢ amountValidators = 3: Define el número de hilos para cancelar una reserva que fue confirmada
  - ➢ amountVerificators = 2: Define el número de hilos para verificar el estado final de la matriz de asientos.
  - ➢ totalThreads =10: Define el número total de hilos
  - ➢ matrixRows : Define el número de filas de la matriz de asientos
  - ➢ matrixColumns : Define el número de columnas de la matriz de asientos.

- Se crea un objeto de la clase seatMatrix, para representar la disposición de los asientos del avión .
- Se crea un objeto de la clase seatRegisters para los registros de las reservas
- Se crea un objeto de la clase Reservator. Recibe como parámetros el objeto `seatMatrix` y `seatRegisters`.
- Se crea un objeto de la clase Paymentaror. Recibe como parámetros el objeto `seatMatrix` y `seatRegisters`.
- Se crea un objeto de la clase Validator. Recibe como parámetros el objeto `seatMatrix` y `seatRegisters`.

- Se crea un objeto de la clase Verificator. Recibe como parámetros el objeto `seatMatrix` y `seatRegisters`.
- Se lanzan 10 hilos:
  - ➢ 3 ejecutando un objeto de tipo Reservator
  - ➢ 2 ejecutando un objeto de tipo Paymentaror
  - ➢ 3 ejecutando un objeto de tipo Validator
  - ➢ 2 ejecutando un objeto de tipo Verificator

2.6.2. Clase seatMatrix

Esta clase proporciona una manera de representar una matriz de asientos de avion, inicializarla con números de asiento únicos y acceder a los asientos individuales mediante sus índices de fila y columna

- Variables de Instancia:

➢ rows y cols son enteros que representan el número de filas y columnas en la matriz de asientos.

➢ matrix es una matriz de objetos Seat (definidos por la clase Seat) que representa los asientos en la matriz.

- Constructor: El constructor seatMatrix toma dos parámetros, rows y cols, e inicializa la matriz de asientos con estas dimensiones. Llama al método setSeatNumbers() para inicializar los números de los asientos en la matriz.

- Métodos:
  - ➢ Método setSeatNumbers (Establecer Números de Asientos): Este método inicializa la matriz de asientos con instancias de la clase Seat, asignando un número de asiento único a cada asiento en la matriz. Los números de asiento comienzan desde 1 y aumentan secuencialmente a medida que se recorre la matriz.
  - ➢ Método getSeat (Obtener Asiento): Este método toma dos índices de fila (row) y columna (col) y devuelve el objeto Seat en esa ubicación de la matriz.
  - ➢ Método getSeatsAmount devuelve el número total de asientos en la matriz (filas * columnas).
  - ➢ getRows devuelve el número de filas en la matriz.
  - ➢ getCols devuelve el número de columnas en la matriz.

2.6.3. Clase Seat

Esta clase representa un asiento del avión en la matriz. Está diseñada para gestionar el estado de los asientos individuales en un lugar de la matriz, permitiendo realizar operaciones como reserva, verificación, cancelación y pago en cada asiento.

- Variables de Instancia:
  - ➢ free, taken, checked, canceled, verified y payed son variables booleanas que representan diferentes estados del asiento. Están marcadas como volatile, lo que significa que los cambios en estas variables son inmediatamente visibles para otros hilos.
  - ➢ number representa el número de asiento y es final, lo que significa que no se puede cambiar después de la inicialización.

- Constructor: Inicializa el asiento con un número de asiento dado y establece todas las variables de estado en sus valores iniciales (free se establece en true, y el resto se establecen en false).

- Métodos Getter y Setter: Cada variable de estado tiene un método getter (por ejemplo, isFree(), isTaken()) y un método setter (por ejemplo, setFree(boolean free), setTaken(boolean taken)) que definen el estado del asiento

- Método reserve (Reservar): El método reserve() está sincronizado y establece el estado free en false y el estado taken en true, marcando efectivamente el asiento como reservado.

2.6.4. Clase seatRegister

Esta clase mantiene los cuatro registros distintos para las reservas:

- Reservas pendientes de pago.
- Reservas confirmadas.
- Reservas canceladas.

- Reservas verificadas

Controlan el flujo de reserva, pago, cancelación, verificación y procesos relacionados con los asientos. Cada método está sincronizado para garantizar que las operaciones se realicen de manera segura en entornos de concurrencia.

- Variables de Instancia:
  - ➢ pending, confirmed, canceled y verified son listas de asientos pendientes de reserva, confirmados, cancelados y verificados respectivamente.
  - ➢ amountReserved, amountConfirmed, amountCanceled y amountVerified son variables que almacenan la cantidad total de asientos en cada lista
  - ➢ rand es una instancia de la clase Random utilizada para generar números aleatorios.

- Constructor: Inicializa las listas y variables de contadores con cero

- Métodos:

  - ➢ Métodos de Obtención de Cantidad: getAmountReserved(), getAmountConfirmed(), getAmountCanceled() y getAmountVerified() devuelvan el tamaño de la lista en ese momento cuando se consultan
  - ➢ Métodos para Procesos de Reserva y Pago:
    - reservationProcess(Seat seat): Recibe un asiento aleatorio de la matriz de asientos. Si está libre, reserva el asiento a través del método reserve() y lo agrega al arreglo pending.
    - confirmPaymentProcess(Seat seat): Recibe un asiento de pending, y chequea si está libre, no está cancelado, y tampoco pagado. Cuando estas condiciones se cumplen, marca el asiento como pagado con setPayed() y lo agrega al arreglo de asientos confirmados (confirmed).
    - cancelPaymentProcess(Seat seat): Recibe un asiento del arreglo pending. Define el estado del asiento como cancelado y lo mueve desde el arreglo pending a canceled.
    - cancelPayedProcess(Seat seat): Con el asiento que recibe del arreglo confirmed, confirma si está pagado, y no verificado. En este caso, marca el asiento como cancelado a través de

setCanceled() y lo agrega al arreglo de asientos cancelados (canceled).
- check(Seat seat):Recibe un asiento del arreglo confirmed. Si fue pagado y no fue validado lo marca como checked.
- verificationProcess(Seat seat): Recibe un asiento del arreglo confirmed, confirma que esté chequeado, y lo pone como verificado con setVerified(), agregandolo al arreglo de verificados (verified).

➢ Método para Obtener una Reserva Aleatoria:
- getRandomReservation(): Devuelve un asiento aleatorio de la lista de reservas pendientes

2.6.5. Clase Reservator

Esta clase representa el proceso de reserva de un asiento del avión. Se encarga de tomar asientos de forma aleatoria y reservarlos cuando sea posible hasta que todos los asientos se encuentren reservados.

○ Variables de Instancia:
- ➢ matrix es una matriz de asientos de la clase seatMatrix.
- ➢ registers es un elemento de la clase seatRegisters.
- ➢ maxOpTime es un entero que representa el máximo tiempo de operación del proceso.
- ➢ randomRow y randomCol son enteros que nos permitirán guardar el par aleatorio de la matriz a reservar.
- ➢ seatToReserve es un elemento de la clase Seat que representa el asiento a intentar reservar.

○ Constructor: Toma como parámetros la matriz de asientos de la clase seatMatrix y un elemento registers de la clase seatRegisters. Genera un par aleatorio de la matriz y se lo asigna a la variable seatToReserve.

○ Métodos:
- run(): si el asiento seatToReserve está libre, llama al proceso de reserva en registers y reserva el asiento. En caso de no estar libre, llama a randomizeSeat() y vuelve a intentar reservar. Repite mientras la cantidad de asientos reservados sea menor al total de asientos en el avión. “FINISHED RESERVATION . . .”

- randomizeSeat(): Informa por el terminal que el asiento seleccionado ya está reservado y que necesita otro. Devuelve otro par aleatorio de la matriz de asientos.

2.6.6. Clase Paymentator

Esta clase representa el proceso de pago de los asientos ya reservados. Se encarga de tomar un asiento aleatorio ya reservado que no sea nulo y procesar el pago con una probabilidad del 10% de ser rechazado. Si el pago es aprobado se confirma la reserva y el pago. Si es cancelado se rechaza el pago y la reserva pasa a ser cancelada. Termina cuando ya procesó el pago de todos los asientos en el arreglo pending.

- Variables de Instancia:
  - ➢ matrix es una matriz de asientos de la clase seatMatrix.
  - ➢ registers es un elemento de la clase seatRegisters.
  - ➢ maxOpTime es un entero que representa el máximo tiempo de operación del proceso.
  - ➢ random es un elemento de la clase Random que permite tomar un asiento reservado aleatoriamente.
  - ➢ chanceToFail representa la probabilidad de que la reserva sea cancelada.
  - ➢ chance es una variable int que guarda números aleatorios que determinan si el pago es cancelado o aprobado.

- Constructor: Toma como parámetros la matriz de asientos de la clase seatMatrix y un elemento registers de la clase seatRegisters. Asigna a chanceToFail el valor de probabilidad de cancelar el pago (10 en este caso) y a chance un número aleatorio entre 1 y 100.

- Métodos:
  - run():el método espera si el arreglo pending está vacío. Cuando pending no está vacío, asigna a una variable Seat llamada seatToPay un asiento reservado aleatorio y actualiza la variable chance mediante checkPayment(). Si seatToPay no es nulo, está ocupado y no fue pagado, pasa al proceso de confirmación/cancelación de pago. Compara la variable chance con chanceToFail. Si chance es mayor a chanceToFail, aplica el proceso de confirmar pago al asiento. Si en

cambio chance es menor o igual, se le aplica el proceso de cancelación de pago. Se ejecuta mientras la cantidad de asientos cancelados sumado a la cantidad de asientos confirmados sea menor que la cantidad total de asientos. Cuando finaliza el método imprime por pantalla “FINISHED PAYCHECK . . .”.

- checkPayment(): devuelve otro número aleatorio entre 1 y 100.

2.6.7. Clase Validator

Esta clase se encarga de validar los asientos que aprobaron el pago guardados en el arreglo confirmed, y los cancela con una probabilidad del 10%. En caso de validar el asiento lo registra en su estado. En caso de cancelar la reserva, lo marca como cancelado y lo guarda con el resto de reservas canceladas. Termina cuando ya no quedan asientos sin validar o cancelar en el arreglo confirmed.

- Variables de Instancia:
  - ➢ matrix es una matriz de asientos de la clase seatMatrix.
  - ➢ registers es un elemento de la clase seatRegisters.
  - ➢ maxOpTime es un entero que representa el máximo tiempo de operación del proceso.
  - ➢ random es un elemento de la clase Random que guarda un asiento reservado aleatoriamente.
  - ➢ chanceToFail representa la probabilidad de que la reserva sea cancelada.
  - ➢ chance es una variable int que guarda números aleatorios que determinan si el pago es cancelado o aprobado.

- Constructor: Toma como parámetros la matriz de asientos de la clase seatMatrix y un elemento registers de la clase seatRegisters. Asigna a chanceToFail el valor de probabilidad de cancelar el pago (10 en este caso) y a chance un número aleatorio entre 1 y 100.

- Métodos:
    - run(): el método chequea si el arreglo de asientos confirmados no está vacio, en caso de ser así, toma un elemento aleatorio y lo verifica con el método check() de la clase seatRegisters, el cual pone el estado checked() como verdadero. Esto lo realiza con una probabilidad del 90%. En el caso del 10% restante, el asiento se cancela, se lo agrega al arreglo canceled, y se marca el estado cancelado como verdadero. Todo esto es realizado mientras la cantidad de asientos confirmados más la cantidad de asientos cancelados sea menor que la cantidad total de asientos. Cuando el proceso termina se imprime el mensaje “FINISHED VALIDATION…”.
    - checkValidation(): devuelve otro número aleatorio entre 1 y 100.

2.6.8. Clase Verificator

Esta clase representa el proceso de verificación de los asientos del avión. Se encarga de quitar los asientos ya válidos del arreglo confirmed y guardarlos en un arreglo con el resto de asientos confirmados y validados.

- Variables de Instancia:
    - ➢ matrix es una matriz de asientos de la clase seatMatrix.
    - ➢ registers es un elemento de la clase seatRegisters.
    - ➢ maxOpTime es un entero que representa el máximo tiempo de operación del proceso.
    - ➢ random es un elemento de la clase Random que guarda un asiento reservado y verificado aleatorio.

- Constructor: Toma como parámetros la matriz de asientos de la clase seatMatrix y un elemento registers de la clase seatRegisters.

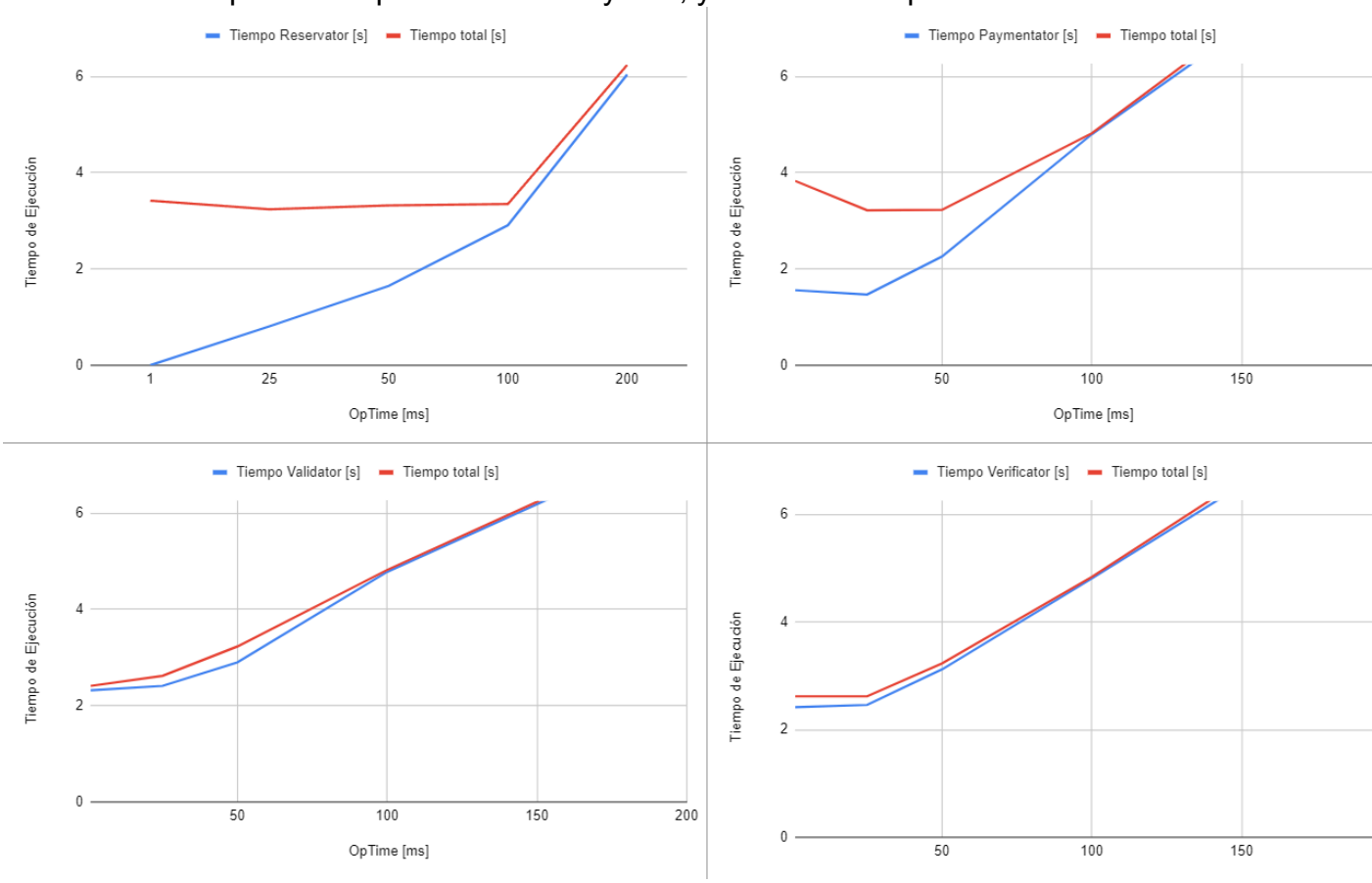
- Métodos:
    - run(): este método verifica que el arreglo confirmed no esté vacío y espera hasta que deje de estarlo. Cuando hay elementos en el arreglo confirmed toma uno de forma aleatoria y chequea que no este vacio y que esté validado por el validator. Si es así verifica el asiento tomado y lo mueve del arreglo confirmed al arreglo verified. Se ejecuta mientras la suma de asientos cancelados y verificados sea menor que el total de asientos. Cuando termina el proceso imprime por pantalla “FINISHED VERIFICATIONS . . .”.

2.7. Tiempos de los Procesos

Por último, tuvimos que decidir cuanto tiempo iba a durar cada proceso. Para ello, fuimos haciendo varias ejecuciones con distintos tiempos de cada proceso y comparamos los resultados. Buscamos identificar qué tan marcada era la dependencia entre procesos a medida que aumenta el tiempo de cada uno (es decir, ver que tanto afecta en el tiempo de ejecución total que un proceso en particular dure una cierta cantidad de tiempo en comparación al resto).

Para identificar estas cuestiones, hicimos 3 secuencias de ejecuciones, donde se analiza el tiempo de ejecución de cada proceso y el tiempo total de ejecución del programa:

1- Se fijan todos los tiempos en 50 milisegundos salvo los de un proceso, y se hace variar el tiempo de ese proceso entre 1 y 200, y así con cada proceso.



Estos gráficos nos permiten observar que la velocidad de ejecución del sistema va a la par de la velocidad de Validator y Verificator, pero, en el caso de Paymentator y más aún en Reservator, esto no es así: hay un límite en el que, aunque se reduzca el tiempo que hacen estos procesos, el sistema no reduce su tiempo de ejecución. Esto nos puede indicar que lo conveniente para minimizar el tiempo de ejecución del sistema es reducir el tiempo de Verificator y Validator.

2- Se fija a todos los procesos en 200 salvo a uno en 1, y se va rotando con cada proceso, lo que se mide en este caso es el tiempo de ejecución total.

Este grafico confirma nuevamente que el tiempo de ejecución de nuestro programa realmente es reducido cuando los tiempos de Validator y Verificator son menores.

3- Se hace lo mismo que en la segunda secuencia, pero todos con 0 y un solo proceso con 200 milisegundos de duración.

En este caso, lo que podemos observar es que las operaciones que mayor cuello de botella producen en nuestro programa son Verificator y Paymentator, ya que una duración larga de estos induce un retardo en el tiempo de ejecución del sistema, así que debemos evitar que estos procesos duren mucho tiempo.

La posible **explicación de los tiempos** observados es la siguiente:

- *Cuando el tiempo de Reserva es demasiado largo, el tiempo de ejecución del programa aumenta*: cuando este caso se da, todos los hilos terminan al mismo tiempo, lo que nos indica que es mayor la cantidad de asientos que “salen del sistema”, es decir, que son cancelados o verificados, que los que “entran”, es

decir, los que son reservados. En este caso, el problema lo trae tener un proceso de reserva muy largo.

- *Cuando el tiempo de Reserva es demasiado corto, el tiempo de ejecución del programa no se reduce necesariamente*: cuando este caso se da, el proceso de reserva termina mucho antes que el resto, lo que nos indica que es mayor la cantidad de asientos que “entran al sistema” que los que “salen”.
- *Cuando los tiempos de Validacion y Verificacion son cortos, el tiempo de ejecución del programa tiende a disminuir*: esto se da porque aseguro una “salida” rápida de datos, de tal forma que si el proceso de pago termina, me aseguro que validación y verificación termina en ese momento también practicamente.
- Se puede concluir que la velocidad del proceso de Reserva determina la rapidez de la entrada de datos, y la velocidad de los procesos de Validacion y Verificacion determina la rapidez de la salida de datos.
- De esta forma, los procesos de Validacion y Verificacion deberían ser rápidos, el de reserva debería ser equivalente al tiempo de validación y verificación, y el proceso de pago o confirmación determina el flujo de datos, por lo que debería ser un balance entre ambos tiempos de procesamiento. Si el proceso de pago es muy rápido pero la entrada o la salida de datos es lenta, entonces no se va a ver afectado el tiempo de ejecución del sistema, en cambio si la entrada y la salida son rápidas pero el proceso de pago no, ahí si va a tener una influencia directa sobre el tiempo de ejecución del sistema (por eso en ambos gráficos de barras el tiempo de ejecución de paymentator se mantiene como uno de los más altos)

Con todas estas conclusiones en mente, se determina que los tiempos de operación de cada proceso van a ser los siguientes:

$$Tiempo\ de\ Operacion\ de\ Reserva\ =\ 250ms$$

$$Tiempo\ de\ Operacion\ de\ Pago\ =\ 200ms$$

$$Tiempo\ de\ Operacion\ de\ Validacion\ o\ Cancelacion\ =\ 300ms$$

$$Tiempo\ de\ Operacion\ de\ Verificacion\ =\ 100ms$$

El equipo ha optado por explorar dos escenarios para la asignación de tiempos de ejecución de los procesos:

- Secuencialización completa de tareas (con una cota de tiempo superior).
- Paralelización total de tareas (con una cota de tiempo inferior).

A continuación, se procederá con el análisis temporal.

Secuencialización completa de tareas;

Se tiene la siguiente suma:

250ms (Reservator)+ 200ms (Paymentator)+ 300ms(Validator)+ 100ms(Verificator)=850[ms]

Multiplicando lo anterior por las 186 reservas a procesar dan como resultado 158100[ms]

- 250ms Reservator = 1 reserva cada 250 ms = 46.6[s]
- 200ms Paymentator = 1 pago/cancelación cada 200ms = 37.2 [s]
- 300ms Validator = 1 validación/cancelación cada 300ms = 55.8 [s]
- 100ms Verificator = 1 verificación cada 100ms = 18.6[s]

Como el trabajo es secuencial, se estima que en el mejor de los casos el proceso termine en promedio 158.2[s]

Paralelización total de tareas

- 250ms 3 Reservator = 3 reservas cada 250 ms = 15.5[s]
- 200ms 2 Paymentator = 2 pago/cancelación cada 200ms = 18.6 [s]
- 300ms 3 Validator = 3 validación/cancelación cada 300ms = 18.6 [s]
- 100ms 2 Verificator = 2 verificación cada 100ms = 9.3 [s]

Con el trabajo no secuencial, se estima que en el mejor de los casos el proceso terminará en promedio 18.6 [s]

3. Conclusiones

Al finalizar con este trabajo práctico, se derivan las siguientes conclusiones:

- Es de fundamental importancia gestionar adecuadamente la sincronización de los hilos. Si no se administra correctamente el acceso a las zonas de memoria compartida (también conocidas como secciones críticas), se pone en riesgo la integridad de los datos, lo que puede resultar en consecuencias inesperadas.
- Hay una complejidad inherente al manejo de variables en un entorno concurrente que se hace visible durante el proceso de desarrollo de programas de este tipo. A medida que un programa concurrente crece, evitar errores se vuelve más desafiante.
- Si bien en un programa completamente secuencial, el tiempo de finalización del sistema aumenta considerablemente, la programación tiende a ser más simple. En contraste, en la programación concurrente, el tiempo de finalización es menor, pero la complejidad de la programación se incrementa notablemente debido a la dificultad de lograr una independencia total entre los recursos utilizados por cada hilo.
- Pudimos implementar y aprender a utilizar la palabra reservada “synchronized” para solucionar distintos problemas relacionados con la concurrencia. Esta herramienta nos permitió establecer secciones criticas
- Mediante el uso de secciones críticas, se aprovecha tanto los momentos de paralelismo para acelerar tareas como los momentos de secuencialidad para evitar conflictos en el código. Esto permite beneficiarse de tiempos de ejecución acotados sin comprometer la estabilidad del programa.
- En conclusión, la identificación de secciones críticas y el uso adecuado de la palabra reservada synchronized son elementos fundamentales en el desarrollo de programas concurrentes. Con synchronized podemos establecer mecanismos de bloque que aseguran que solo un hilo pueda acceder a una sección critica al mismo tiempo. Estas prácticas son esenciales para evitar problemas de concurrencia y garantizar la exclusión mutua, asegurando así la integridad y consistencia de los datos compartidos entre múltiples hilos.