PROGRAMACIÓN CONCURRENTE

“Sistema de gestión de imágenes”

Año: 2023
Primer cuatrimestre
Profesor: Luis Orlando Ventre
Mauricio Ludemann

Grupo “Master of threads”		
Apellido y nombre	**Matricula**	**Mail**
Melia, Nicolás Osvaldo	43167517	nicolas.melia@mi.unc.edu.ar
Delamer, Ignacio	43359691	ignaciodelamer@mi.unc.edu.ar
Aldana, Pavet	43884931	aldana.pavet.garcia@mi.unc.edu.ar
Rivarola, Ignacio Agustin	42808907	ignacio.rivarola@mi.unc.edu.ar
Soria, Federico Isaia	40574892	federico.isaia.soria@mi.unc.edu.ar

Índice

Introducción

La programación concurrente es una técnica utilizada en sistemas críticos que permite el desarrollo de aplicaciones que puedan ejecutar varias tareas de forma simultánea, mejorando de este modo, la eficiencia y la capacidad de respuesta de un programa.

El objetivo en el trabajo expuesto a continuación será utilizar la programación concurrente para la creación de un gestor de imágenes, empleando un enfoque basado en 4 procesos donde cada uno lleva a cabo una tarea específica para lograr una mayor flexibilidad y eficiencia.

Estos procesos se presentan a continuación:

- Creación de un contenedor con 100 objetos imagen haciendo uso de 2 hilos.
- Modificar la iluminación de cada una de las imágenes presentes en el contenedor utilizando 3 hilos donde cada uno de ellos puede acceder una sola vez a cada imagen.
- Cada vez que una imagen finaliza el proceso 2, se pide ajustar el tamaño de cada imagen una única vez, siendo esta tarea realizada por 3 hilos.
- Y por último, se debe almacenar las 100 imágenes con todas sus modificaciones pertinentes en un contenedor final mediante el uso de 2 hilos.

Todos estos procesos se deben inicializar desde una clase main que permite la correcta implementación del programa.

Asimismo se detalla el proceso de diseño, la implementación del gestor y la argumentación para las decisiones tomadas en su desarrollo, como así también los resultados obtenidos, los problemas más frecuentes y las conclusiones obtenidas por parte del grupo.

Marco teórico

Es fundamental para la programación concurrente poder sincronizar los métodos que afecten a regiones críticas, es decir, llevar adelante las instrucciones de los hilos que ejecutan los procesos permitiendo el acceso solo a uno de ellos a la variable que conforma la región crítica.

Las regiones críticas son mecanismos de nivel medio de abstracción orientados a su implementación en el contexto de un lenguaje y que permiten la ejecución de un bloque de sentencias de forma segura.

Para controlar este acceso a recursos compartidos podemos encontrar en el semáforo, una herramienta muy valiosa. Creados por Dijkstra (1968), son elementos pasivos de bajo nivel de abstracción que se utilizan para arbitrar el acceso a un recurso por múltiples procesos e hilos. Estos son un tipo de datos y como cualquier tipo de datos, queda definido por un conjunto de valores y operaciones que se le pueden asignar.

También pueden clasificarse según la cantidad de permisos otorgados, llamándose semáforos binarios los que toman solo los valores 0 y 1, o bien pueden ser semáforos generales, aquellos que pueden tomar cualquier valor no negativo.

Otro método para aislar operaciones en una zona crítica es mediante la palabra “synchronized”, la cual es una característica de Java para sincronizar el acceso a recursos compartidos por varios hilos. Con este método, solo se permite el acceso a un único hilo a la región crítica del código englobada por dicha palabra, de este modo, se evitan las condiciones de carrera y asegura la coherencia de los datos.

Cuando un hilo invoca un método synchronized, trata de tomar el lock del objeto al que pertenezca. Si este se encuentra disponible, lo toma y se ejecuta. En cambio si está siendo ejecutado por otro hilo, suspende el que lo invoca hasta que el primero finalice y libere el lock.

Algunas consideraciones para este método es que el lock es tomado por el hilo por lo que mientras un hilo tiene tomado el lock de un objeto puede acceder a otro método synchronized del mismo objeto, lo que implica que el lock es por cada instancia del objeto.

Estos objetos utilizados para el lock tienen métodos propios de la clase Object y solo pueden invocarse por el thread propietario del lock, se presentan a continuación:

- Wait(): Es una espera de tiempo indefinida hasta que recibe una notificación.
- Notify(): Notifica al objeto de un cambio de estado, esta notificación es transferida a un único hilo que se encuentran en espera, de forma aleatoria.
- NotifyAll(): Notifica a todos los hilos que se encuentran en espera del cambio en el estado del objeto.

Si bien la programación concurrente presenta otras herramientas para la sincronización, las mencionadas fueron las utilizadas para la resolución del trabajo práctico.

Desarrollo

El trabajo expuesto consta de 9 clases (incluyendo el main), y de 12 hilos.

El proceso 1 es el encargado de crear 100 imágenes en total y cargarlas en un contenedor para el cual se utilizó un elemento de la clase ArrayList, siendo esto realizado mediante la implementación de un bucle while.

En el proceso 2, se mejora la iluminación de las imágenes que se encuentran en el contenedor antes mencionado. Esta tarea crea una imagen auxiliar, a la cual se le mejora la iluminación (aumentando su valor en 1) y guardando el nombre del hilo que se encargó de dicha mejora en un registro, esto se implementa para evitar que un hilo mejore más de una vez la misma imagen.

Pasando así al proceso 3, proceso donde se recorta la imagen, ajustando al tamaño deseado, cuyo recorte queda indicado bajo el valor de un booleano. Aquí lo que se hizo fue tomar la imagen del contenedor, modificar su tamaño, estableciendo así su valor booleano en 1.

Y por último en el proceso 4, el cual se encarga de sacar las imágenes del contenedor original, y las carga en el contenedor final.

Cabe aclarar que todos los procesos se ejecutan de manera concurrente, comenzando todos al inicio del programa y terminan una vez que sus tareas hayan concluido.

En cuanto al diseño de las clases, para los procesos optamos por utilizar “while” para poder quedar en bucle realizando las tareas de cada proceso.

La clase Container, consta de de un contenedor (donde se guardan las imágenes base), un contenedor auxiliar (utilizado por algunos procesos), un contenedor final (donde están las imágenes procesadas y recortadas), su capacidad máxima (indicada por la variable capacidad) y un booleano que indica cuando un contenedor está lleno. Todos los contenedores antes mencionados, son contenedores (representados por Arraylist) de imágenes.

La clase imagen tiene un integer “iluminación”, el cual se aumenta cada vez que un hilo la mejora. Luego está el tamaño (booleano encargado de indicar si está recortada o no). “mejoras” que indica la cantidad de veces que fue mejorada, que a su vez, esto está acompañada de el booleano “iluminacion_lista” que se torna TRUE cuando la iluminación fue mejorada por cada hilo. Luego además de contener su nombre, también tiene un HashSet llamado registro, que guarda el nombre de cada hilo que se encargó de mejorar la iluminación. Esto fue realizado de esta manera, ya que como mencionamos antes, fue la manera en la cual nos encargamos de que un hilo no mejore más de una vez la misma imagen.

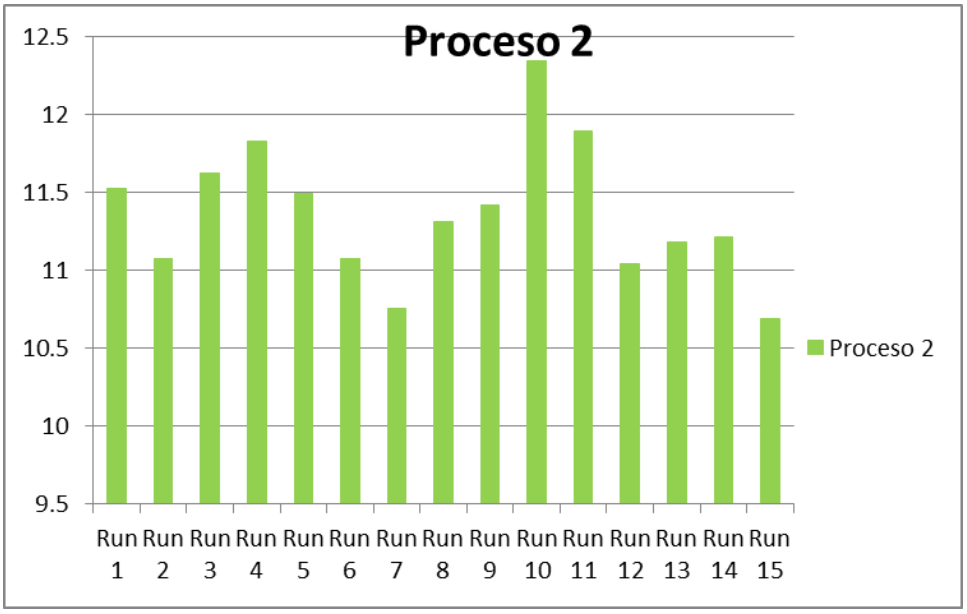
Resultados

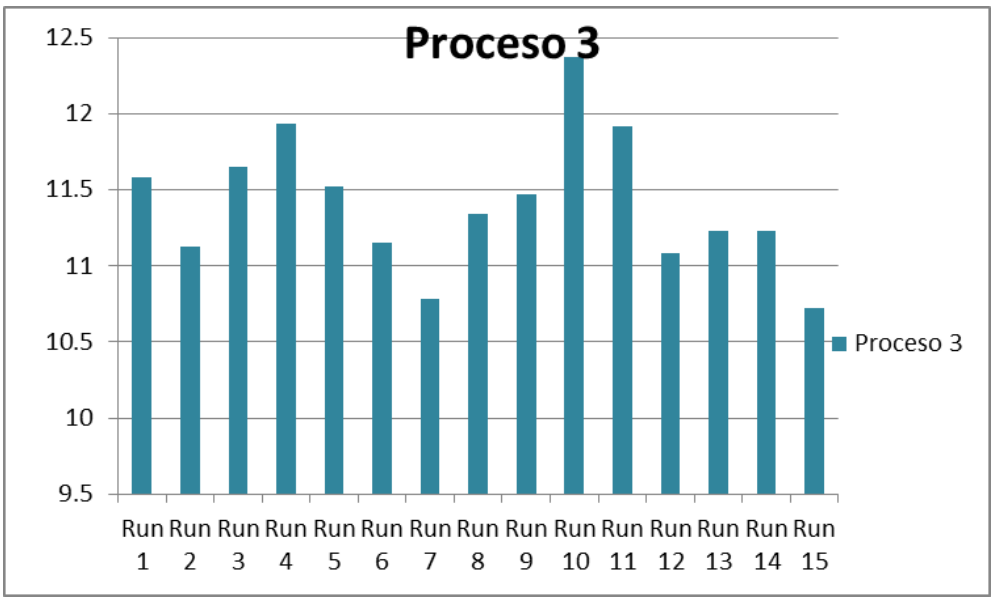
Enlace al diagrama de secuencias

[Diagrama de Secuencia: Lucidchart]

Tabla con tiempos de ejecución en segundos

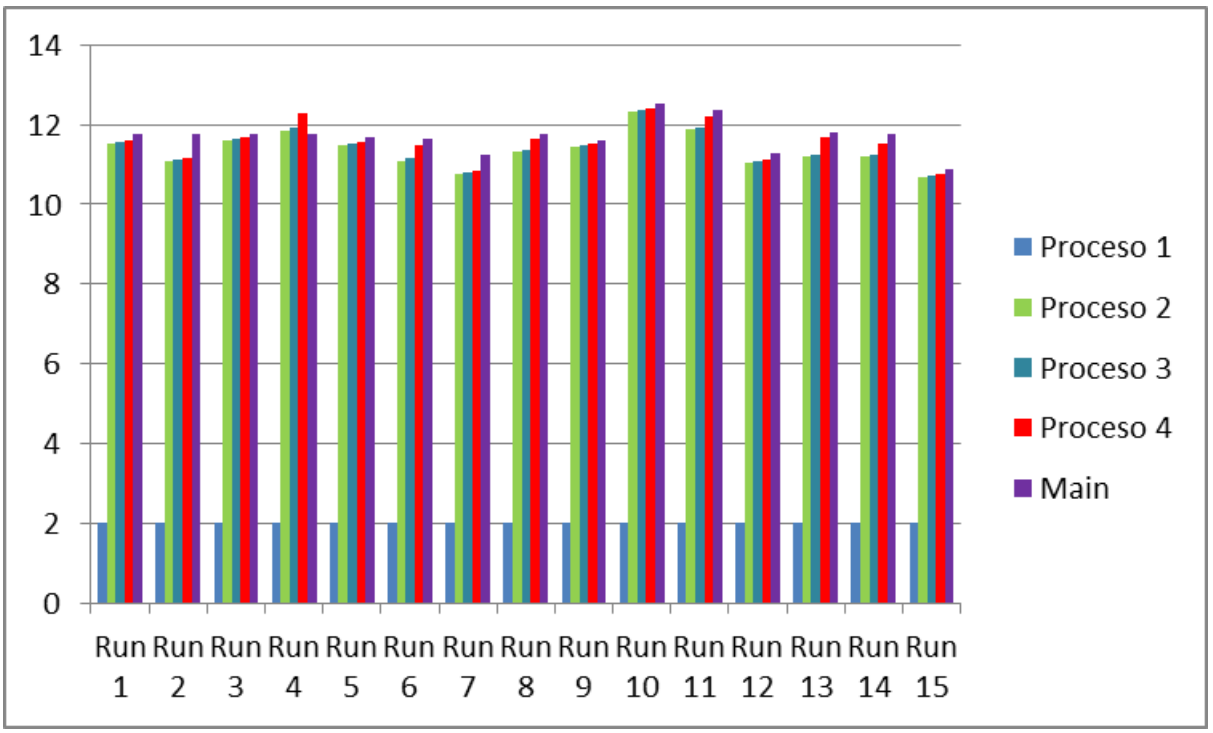
	Proceso 1	***Proceso 2***	***Proceso 3***	***Proceso 4***	***Main***
Run 1	2	11.52	11.58	11.61	11.76
Run 2	2	11.07	11.13	11.14	11.77
Run 3	2	11.62	11.65	11.68	11.76
Run 4	2	11.83	11.93	12.27	12.35
Run 5	2	11.49	11.52	11.56	11.68
Run 6	2	11.07	11.15	11.47	11.63
Run 7	2	10.75	10.78	10.82	11.25
Run 8	2	11.31	11.34	11.65	11.78
Run 9	2	11.42	11.47	11.5	11.62
Run 10	2	12.34	12.37	12.4	12.53
Run 11	2	11.89	11.92	12.22	12.35
Run 12	2	11.04	11.08	11.13	11.26
Run 13	2	11.18	11.23	11.68	11.82
Run 14	2	11.21	11.23	11.51	11.75
Run 15	2	10.69	10.72	10.75	10.88




Proceso 3
12.5
12
11.5
11
10.5
10
9.5
Run 1
Run 2
Run 3
Run 4
Run 5
Run 6
Run 7
Run 8
Run 9
Run 10
Run 11
Run 12
Run 13
Run 14
Run 15
Proceso 3

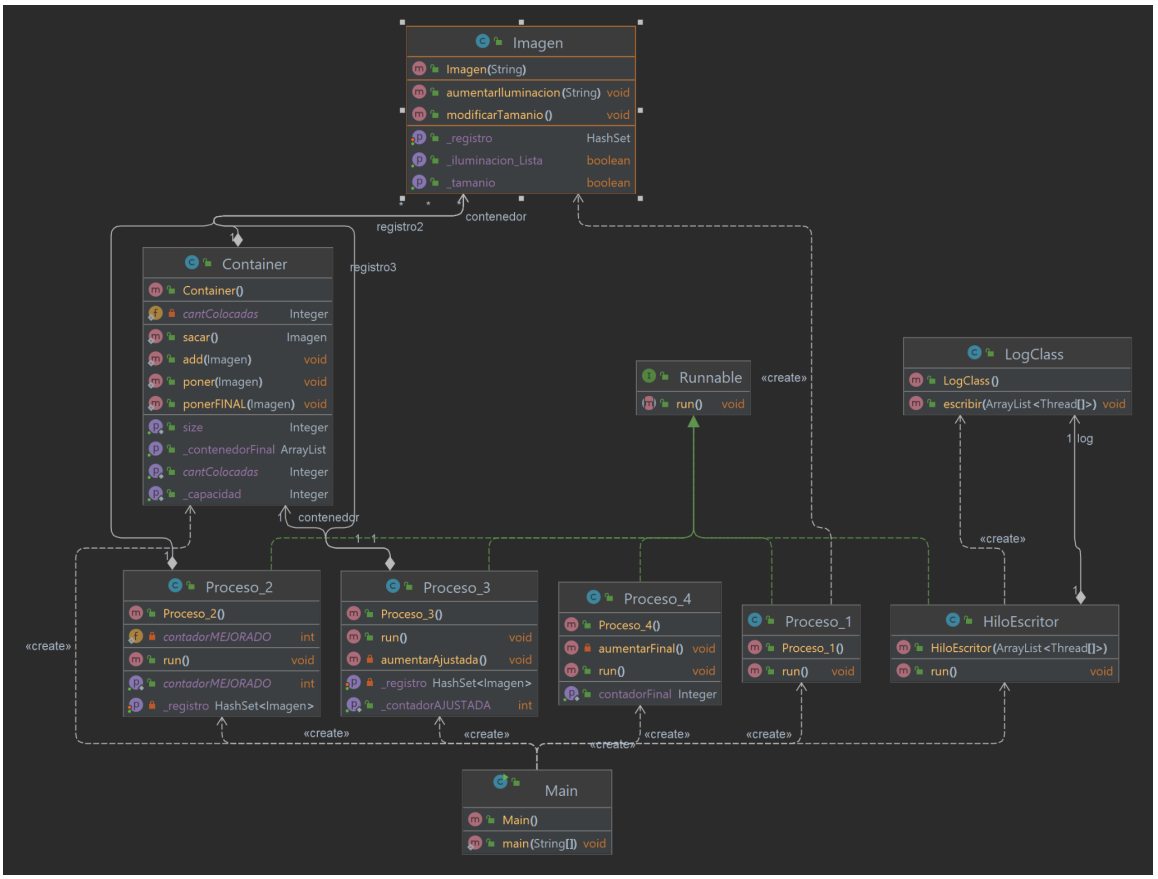

Proceso 4
13
12.5
12
11.5
11
10.5
10
9.5
Run 1
Run 2
Run 3
Run 4
Run 5
Run 6
Run 7
Run 8
Run 9
Run 10
Run 11
Run 12
Run 13
Run 14
Run 15
Proceso 4


Main
13
12.5
12
11.5
11
10.5
10
Run 1
Run 2
Run 3
Run 4
Run 5
Run 6
Run 7
Run 8
Run 9
Run 10
Run 11
Run 12
Run 13
Run 14
Run 15
Main



Como podemos apreciar en la tabla, vemos que el proceso 3 y el proceso 2, tienen un tiempo de ejecución similar, aunque el proceso 3 debería ser mucho más rápido que el proceso 2. Esto se debe a que el proceso 3 recorta las imágenes, si y sólo si, dicha imagen a modificar, cuenta con su iluminación al máximo (recordar que para que esto suceda, cada hilo del proceso 2, debe modificar la iluminación de UNA sola imagen. Es decir, un hilo no puede mejorar la iluminación dos veces de una misma imagen). Motivo por el cual, hasta que la última imagen no haya sido iluminada en el proceso 2, no podrá ser recortada en el proceso 3, produciendo así los resultados obtenidos. Cabe aclarar que tomamos el proceso terminado, en el momento que se modificaron las 100 imágenes del contenedor.

Diagrama de clases



Conclusiones

Los principales inconvenientes encontrados en este trabajo fue programar involuntariamente de forma secuencial, característica propia de los programas que hemos ido diseñando en los primeros años de carrera. Era un problema recurrente ejecutar una serie de hilos, de los cuales solo uno ejecutaba todo el método run() de la clase asociada, aunque fue un problema que pudimos sortear.

Un inconveniente fue no liberar el lock antes de ejecutar la instrucción sleep de un hilo, como consecuencia el hilo "dormía" con la llave lo que impedía el acceso por otros hilos en espera. Este problema nos ayudó a entender el traspaso de permisos entre hilos para el correcto desarrollo de las instrucciones asignadas.

Otro aprendizaje valioso fue entender que para una correcta sincronización debíamos establecer como síncronos los métodos asociados a las regiones de exclusión y no los métodos "run" implementados por los hilos en tiempo de ejecución. Por esto, sincronizamos solamente donde se escribe o lee información de dichas variables.

Por último podemos decir que la gran conclusión obtenida al realizar este trabajo práctico fue que la concurrencia no es para nada trivial, a veces algo que parece resuelto no lo está del todo. Dada esta complejidad entendimos por qué el saber estos temas nos da herramientas valiosas, lo que se traduce en una ventaja en el mercado laboral. Es por eso que debemos prestar atención y tener los conocimientos necesarios para poder abordar y resolver con claridad y eficiencia los problemas que surgen a la hora de diseñar sistemas complejos.