UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

MODISMOS BASADOS EN TIPOS DE DATOS ATÓMICOS DE JAVA Y ANÁLISIS DE SU DESEMPEÑO.

TESIS

QUE PARA OPTAR POR EL GRADO DE MAESTRO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

PRESENTA:
JUAN CARLOS GONZÁLEZ TAMAYO

Director de Tesis:
Dr. Jorge Luis Ortega Arjona
Departamento de Matemáticas, Facultad de Ciencias.

Ciudad Universitaria, Ciudad de México, 2018
Octubre

Universidad Nacional
Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Dedicatoria

A mi madre, a mi padre, a mi familia.

Agradecimientos

A todos mis familiares, amigos, profesores, compañeros de estudio y demás personas que directa o indirectamente han contribuido a mi formación personal y profesional a lo largo de toda mi vida.
A Guillermo Olvera y a su familia por acogerme en su seno como parte de la misma.
A mi tutor, por su apoyo y guía en la realización de este trabajo.
A los sinodales por tomar de su tiempo para revisar la presente tesis.
Al Posgrado en Ciencia e Ingeniería de la Computación de la UNAM por la oportunidad de superarme.
Al Consejo Nacional de Ciencia y Tecnología (Conacyt) por el apoyo económico que me permitió realizar y culminar satisfactoriamente mis estudios de maestría.
A la Universidad Nacional Autónoma de México, a México y a los mexicanos.
A todos los muertos de mi felicidad.

"Soy feliz, soy un hombre feliz, y quiero que me perdonen por este día los muertos de mi felicidad."

Índice general

Dedicatoria I

Agradecimientos II

Índice de figuras VI

Índice de cuadros VIII

Resumen IX

1. Introducción 1
- 1.1. El contexto . . . 1
- 1.2. El problema . . . 2
- 1.3. La hipótesis . . . 2
- 1.4. La aproximación . . . 3
- 1.5. Contribuciones . . . 3
- 1.6. Estructura del documento . . . 3

2. Antecedentes 5
- 2.1. Sobre la programación concurrente y paralela . . . 5
  - 2.1.1. Procesos e hilos . . . 5
  - 2.1.2. Memoria compartida, variable compartida y memoria distribuida . . . 6
  - 2.1.3. Cambio de contexto y condición de carrera . . . 6
  - 2.1.4. Sección Crítica . . . 9
  - 2.1.5. No determinismo . . . 10
  - 2.1.6. Sincronización . . . 10
  - 2.1.7. *Spinlock* . . . 12

2.2. Programación no bloqueante 12
2.2.1. Instrucciones atómicas 13
2.2.2. Primitiva *Compare and Swap* (CAS) 13
2.2.3. El problema ABA 14
2.2.4. Biblioteca *Atomic* de Java 14
2.3. Corrección de programas paralelos 17
2.4. Presentando los patrones 19
2.4.1. Patrones de *software* 20
2.4.2. Patrones de diseño para la programación paralela 20
2.4.3. Forma POSA 21
2.5. Resumen 22

3. Trabajo Relacionado **23**
3.1. Implementaciones de estructuras de datos libres de bloqueos . . . 23
3.2. Planificadores libres de bloqueos 24
3.3. Resumen 26

4. Modismos basados en tipos de datos atómicos **27**
4.1. Modismo *Atomic Flag* 28
4.2. Modismo *Atomic Counter* 35
4.3. Modismo *Atomic Counters Array* 41
4.4. Modismo *Atomic Node* 48
4.5. Resumen 54

5. Experimentos y resultados **56**
5.1. Escenario de *hardware* y *software* 56
5.2. Experimentación 57
5.2.1. Experimentos con *Atomic Flag* 57
5.2.2. Experimentos con *Atomic Counter* 60
5.2.3. Experimentos con *Atomic Counters Array* 62
5.2.4. Experimentos con *Atomic Node* 67
5.3. Resumen 75

6. Conclusiones **76**
6.1. Evaluación de la hipótesis 76
6.1.1. Discusión 76
6.1.2. Análisis e interpretación de los resultados 77
6.2. Consideraciones 77

6.2.1. Ventajas 77
6.2.2. Desventajas 77
6.3. Comparación del trabajo relacionado 78
6.4. Resumen de las contribuciones 78
6.5. Trabajo futuro 79

Bibliografía **80**

Índice de figuras

2.1. Taxonomía de Flynn 6
2.2. Memoria compartida entre procesadores 7
2.3. Memoria compartida entre hilos 7
2.4. Memoria distribuida 7
2.5. Instrucción *Compare and Swap* 14
2.6. Un escenario donde ocurre el problema ABA 15

3.1. Comparación entre la cola propuesta y otras implementaciones . 24
3.2. Comparación del tiempo de ejecución entre el planificador cooperativo libre de bloqueos, denotado como *Native*, y los núcleos de los sistemas operativos Linux y Windows. 26

4.1. Diagrama de colaboración del modismo *Shared Variable Pipe* [24]. 28
4.2. Un diagrama representando un *AtomicBoolean* como un tipo de dato abstracto. 31
4.3. Diagrama de secuencia de dos componentes de *software* concurrentes o paralelos que acceden a su sección crítica protegidos por un *Atomic Flag*. 33
4.4. Diagrama de objetos de un contador de instancias creadas. . . . 35
4.5. Un diagrama representando un contador atómico como un tipo de dato abstracto. 38
4.6. Diagrama de secuencia de la interacción de varios componentes de *software* con una variable entera atómica. 39
4.7. Representación del *Counting Sort* donde *Array* es el arreglo a ordenar y *Counters Array* es el arreglo de contadores. 41
4.8. Representación de un arreglo de contadores enteros atómicos como un tipo de dato abstracto. 43

4.9. Diagrama de secuencia simplificado de dos componentes de *software* concurrentes y/o paralelos que acceden al arreglo de contadores atómicos, su sección crítica. 45
4.10. Árbol trie 48
4.11. Un diagrama representando un *Atomic Node* como un tipo de dato abstracto. 50
4.12. Diagrama de secuencia de la interacción de dos componentes de *software* con una referencia a un nodo atómico. 52

5.1. Gráfica comparativa entre los promedios del tiempo de ejecución del programa que usa protección con *Reentrant Lock* y el que emplea el modismo *Atomic Flag*. 60
5.2. Gráfica comparativa entre los promedios del tiempo de ejecución del programa que usa protección con *Atomic Flag* y el que emplea el modismo *Atomic Counter*. 62
5.3. Gráfica comparativa entre los promedios del tiempo de ejecución del programa que usa un arreglo de tipo *ArrayList* y el que emplea el modismo *Atomic Counters Array*. 66
5.4. Gráfica comparativa entre los promedios del tiempo de ejecución del programa que usa un árbol *Trie* sincronizado y el que emplea un árbol *Trie* basado en el modismo *Atomic Node*. 74

Índice de cuadros

2.1. Clases principales de la biblioteca *java.util.concurrent.atomic* [8] 18

5.1. Resultados experimentales del uso de *ReentrantLock* y del modismo *Atomic Flag* para proteger una variable entera. 59

5.2. Resultados experimentales del uso de *Atomic Flag* para proteger una variable entera y del modismo *Atomic Counter* como un contador entero modificado con operaciones atómicas. 61

5.3. Resultados experimentales en cuanto a tiempo de ejecución del uso de un arreglo de tipo *ArrayList* sincronizado con monitores de Java para contar ocurrencias de elementos en un arreglo a ordenar y del modismo *Atomic Counters Array* con la misma función pero sin sincronización a nivel de programación. 66

5.4. Resultados experimentales del tiempo de ejecución de un árbol *Trie* sincronizado con monitores de Java y de un árbol *Trie* basado en el modismo *Atomic Node* con la misma función pero sin sincronización a nivel de programación. 73

Resumen

La programación concurrente y/o paralela juega un papel importante en el desarrollo de programas informáticos que realizan gran procesamiento de datos y/o que necesitan realizar acciones simultáneamente. Introduce una serie de conceptos, como los mecanismos de sincronización entre hilos y/o procesos, para lograr la exclusividad en el acceso a datos compartidos. Entre los más conocidos están los semáforos, regiones críticas, monitores, etc. Todos ellos, aunque muy útiles por lo general, presentan el problema de ser lentos en tiempo de ejecución a la hora de adquirirlos y liberarlos cuando hay gran contención. Son poco intuitivos de usar en la implementación de una aplicación paralela con múltiples componentes concurrentes que comparten datos. Y presentan varios problemas inherentes a su posible mal uso como los *deadlocks*, *livelocks*, etc.

A partir de la implementación a nivel de *hardware* de instrucciones atómicas del tipo *read-write-update*, como *compare and swap*, se abre una nueva posibilidad de escribir algoritmos y estructuras de datos que pueden prescindir de los mecanismos de sincronización mencionados anteriormente.

Un patrón de *software* para la programación paralela "describe un problema que ocurre una y otra vez en nuestro entorno, y luego describe el núcleo de la solución a dicho problema, de tal forma que se puede usar esta solución muchas veces más, sin nunca hacerlo necesariamente de la misma forma dos veces" en el contexto del desarrollo de un programa paralelo [13].

En lenguajes de uso extendido, como Java o C++, se han introducido bibliotecas que permiten leer y modificar algunos tipos de datos mediante operaciones atómicas sobre ellos. El objetivo del presente trabajo es capturar soluciones basadas en el uso de estas facilidades para resolver ciertos problemas de forma concurrente y/o paralela, describirlas como patrones de bajo nivel o modismos, y medir su impacto en el desempeño en cuanto a tiempo de ejecución.

Capítulo 1

Introducción

1.1. El contexto

"La programación concurrente es la actividad de construir un programa que contiene múltiples procesos que se ejecutan en paralelo. Estos procesos compiten por el acceso a recursos críticos y cooperan en la realización de alguna tarea" [2]. El acceso controlado a los recursos críticos por parte de los diferentes procesos se resuelve mediante el empleo de mecanismos de sincronización. Estos comprenden a los semáforos, regiones críticas, monitores y otros, los cuales permiten el acceso seguro a la memoria compartida en tanto que la mayoría de las operaciones sobre la misma no son atómicas. Esto quiere decir que hay estados intermedios en que los datos son inconsistentes, siendo preciso prohibir el acceso a otros procesos hasta tanto no se complete la operación [31].

La sincronización entre procesos a nivel de programación, aunque útil en la mayoría de los casos, no es gratuita. Su uso introduce un retardo (*overhead*) que requiere ser medido y analizado. Además, sincronizar correctamente el acceso a recursos compartidos no es trivial, por tanto, es fuente de errores de programación e introduce problemas técnicos como la inversión de prioridades, *deadlocks*, *livelocks*, etc.

Es posible deshacerse en menor o mayor medida de la sincronización bloqueante mediante el uso de técnicas programación libre de bloqueos. Esta técnica consiste en el empleo de instrucciones atómicas para manipular la memoria compartida [21]. Lenguajes de programación como Java y C++ han introducido bibliotecas que proveen de clases que encapsulan datos y brindan una interfaz de operacio-

nes atómicas sobre dichos datos. Para esto, emplean a bajo nivel instrucciones de procesador como *compare and swap*.
Los patrones de diseño identifican un problema recurrente y propone una solución al mismo. De tal forma, esta solución puede ser utilizada cada vez que se presenta un problema similar [13]. Los patrones de diseño para la programación concurrente por su parte, capturan soluciones a problemas recurrentes de programación concurrente y paralela.

1.2. El problema

El empleo de patrones de *software* es práctica común en el desarrollo de programas informáticos por la rápida y elegante solución que los patrones dan a problemas conocidos. Aunque existen patrones de *software* para la programación paralela, estos suponen o hacen uso de primitivas de sincronización bloqueantes. El empleo de primitivas de sincronización bloqueantes para el acceso a los datos compartidos por los componentes que conforman el patrón introduce un retardo en ocasiones no despreciable. Además, "los algoritmos bloqueantes, sufren de una degradación significativa en el desempeño cuando un proceso es detenido o retrasado en un momento inoportuno. Algunas fuentes de retraso posibles son: el planificador del sistema operativo, fallos de página o faltas de la memoria caché" [21].
En lenguajes de programación como Java, ya existen bibliotecas para el trabajo con operaciones atómicas sobre datos. Aún así, no existen patrones de uso en forma de modismos para la programación concurrente basados en este tipo de mecanismo que ofrece en determinadas circunstancias mejor rendimiento en cuanto a tiempo de ejecución y mayor robustez del programa para hacer frente a los problemas mencionados para la programación bloqueante.

1.3. La hipótesis

Es posible extraer patrones de bajo nivel o modismos a partir del uso de los tipos de datos atómicos de Java, de tal manera que permitan el diseño e implementación de ciertos algoritmos y estructuras de datos de uso común, que sean más eficientes prescindiendo de las primitivas bloqueantes de sincronización.

1.4. La aproximación

Mediante el estudio de la programación libre de bloqueos y de patrones de programación paralela, el presente trabajo se propone presentar nuevos patrones de programación paralela basado en los tipos de datos atómicos de Java. Se diseñan y realizan los experimentos pertinentes para demostrar tanto su funcionalidad como su eficiencia en tiempo de ejecución. La experimentación consiste en implementar pruebas de *performance* a soluciones de problemas empleando técnicas de sincronización bloqueantes y los modismos no bloqueantes presentados en este trabajo, y realizar los análisis comparativos pertinentes.

1.5. Contribuciones

El presente trabajo se propone como contribuciones iniciales las siguientes:

- Demostración práctica de la viabilidad tanto funcional como de desempeño del uso de los tipos de datos atómicos de Java como modismos para la programación concurrente y/o paralela.
- Análisis comparativo entre el desempeño de soluciones basadas en primitivas de sincronización bloqueantes y de soluciones basadas en los modismos que se presentan en este trabajo.

1.6. Estructura del documento

La tesis está estructurada de la siguiente forma:

- **Capítulo 2 Antecedentes:**
  En este capítulo se presenta el marco teórico en el cual se sustenta el presente trabajo. Se explican los fundamentos de la programación concurrente y/o paralela. Esto incluye, los conceptos de procesos e hilos, memoria compartida, los diferentes mecanismos de sincronización, etc. Se introducen el concepto de programación libre de bloqueos, así como las operaciones atómicas sobre las cuales se basa esta técnica, y cómo estas se ven reflejadas en la biblioteca *java.util.concurrent.atomic* de Java. Se brinda además, una introducción al tema de los patrones de *software* en general y a los patrones para la programación paralela en específico.

- **Capítulo 3 Trabajo Relacionado:**
  Se comentan trabajos previos relacionados con el tema de la tesis y que brindan una base sobre la cual hacer nuevos adelantos y comparaciones. En específico, se comentan los trabajos: "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms" de Maged M. Michael y Michael L. Scott, y "On the Desing and Implementation of an Efficient Lock-Free Scheduler" de los autores Florian Negele, Felix Friedrich, Suwon Oh y Bernhard Egger.

- **Capítulo 4 Modismos basados en tipos de datos atómicos:**
  Como contribución del trabajo, se presentan cuatro modismos basados en los tipos de datos atómicos que ofrece Java. Para su descripción se emplea el formato propuesto y empleado en POSA [7]. Estos son *Atomic Flag*, *Atomic Counter*, *Atomic Counters Array* y *Atomic Node*.

- **Capítulo 5 Experimentos y resultados:**
  Se presenta un experimento por cada modismo presentando en al Capítulo 4, con el objetivo de comprobar la hipótesis de la tesis. Estos experimentos consisten en comparar el tiempo de ejecución de un programa basado en el modismo que se prueba y una versión equivalente empleando bloqueos. Por cada uno de ellos se hace un análisis de resultados y finalmente se realiza un análisis general de los resultados.

- **Capítulo 6 Conclusiones:**
  Luego de presentarse el trabajo se discuten y analizan los resultados generales y específicos de la tesis. Se recomiendan, además, posibles trabajos futuros que den continuidad al tema tratado.

Capítulo 2

Antecedentes

En el presente capítulo se plasman los conceptos fundamentales de la programación concurrente y paralela, haciendo énfasis en los diferentes mecanismos de sincronización bloqueantes existentes. Luego se introduce la teoría en torno a la programación libre de bloqueos, y finalmente se habla de los patrones de *software* para la programación paralela.

2.1. Sobre la programación concurrente y paralela

"Un sistema se dice que es concurrente si puede mantener dos o más tareas en progreso al mismo tiempo. En tanto un sistema se dice que es paralelo si puede mantener dos o más tareas ejecutándose simultáneamente. El concepto clave que diferencia estas definiciones es la frase en progreso" [6]. Existe el paralelismo de datos y de instrucciones, el cual permite clasificar los distintos sistemas según su grado de paralelismo en ambas dimensiones. Esta clasificación se conoce como la taxonomía de *Flynn* [29] y se ilustra en la figura 2.1.

2.1.1. Procesos e hilos

Los flujos de instrucciones toman forma en los conceptos de proceso e hilos, los cuales son ejecutados por una computadora: "Un proceso es una instancia de un programa en ejecución, incluyendo los valores actuales del contador de programa, los registros y las variables" [29]. Una definición más intuitiva es que

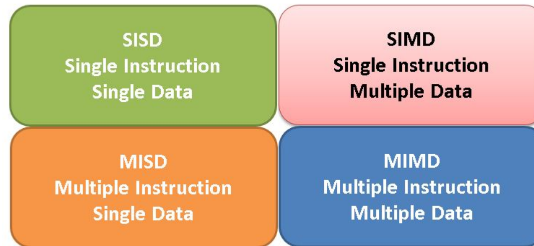

Figura 2.1: Taxonomía de Flynn

un proceso de computadora es el cambio controlado en el tiempo de la memoria con un fin determinado. La característica clave de los procesos y que la diferencia de los hilos es que cada proceso tiene su propio espacio de memoria.
Los hilos, en tanto, se definen como procesos secuenciales con la peculiaridad de que estos sí comparten el mismo espacio de memoria, siempre que estos se inicien bajo un mismo proceso [19]. También son conocidos como procesos ligeros.

2.1.2. Memoria compartida, variable compartida y memoria distribuida

La memoria se puede compartir tanto a nivel de *hardware* como de *software*. El espacio de memoria común accesible físicamente por todos los procesadores de una misma computadora es conocido como memoria compartida (figura 2.2). Por su parte, el concepto de variable compartida está dado a zonas de memoria accesibles solo a los hilos bajo un mismo proceso (figura 2.3). Ambas formas de compartir información representan mecanismos de comunicación entre procesos e hilos.
Otro modelo de memoria es la distribuida, donde varias computadoras se comunican en red para resolver un problema determinado. En este caso, cada computadora tiene su memoria local, la cual puede ser compartida, pero, el acceso a información en diferentes computadoras requiere el paso de mensajes (figura 2.4).

2.1.3. Cambio de contexto y condición de carrera

Un cambio de contexto (*context switch*) es el evento que ocurre cuando el procesador interrumpe la ejecución de un hilo y/o proceso para correr otro

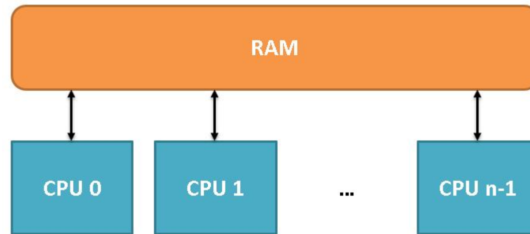

Figura 2.2: Memoria compartida entre procesadores

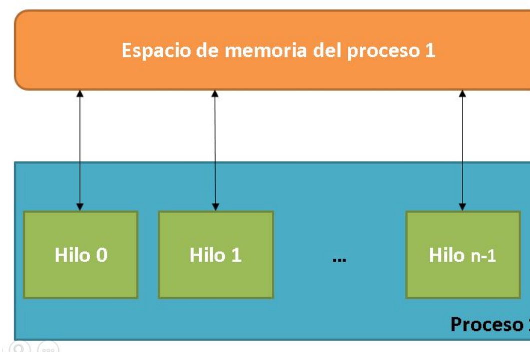

Figura 2.3: Memoria compartida entre hilos

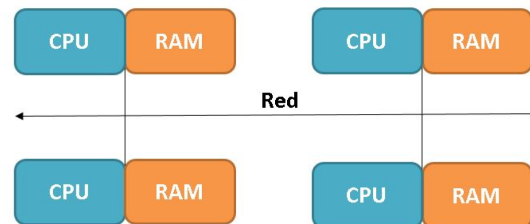

Figura 2.4: Memoria distribuida [4]

[17]. Para esto, se guarda el estado (contexto) del procesador al momento de la interrupción, de tal manera que se puede continuar la ejecución del hilo y/o proceso en otro momento a partir del estado en el que estaba, y se carga el estado o contexto del hilo que se va a continuar ejecutando.
Las condiciones de competencia (*race conditions*) ocurren cuando dos o más hilos comparten datos que están leyendo y escribiendo concurrentemente, y el resultado final, que puede ser erróneo, depende de la intercalación específica de las instrucciones en lenguaje de máquina ejecutados por los hilos y provocada por los cambios de contexto del procesador [16]. Esta condición de competencia trae consigo el problema de la "pérdida de actualizaciones" (*"lost update" problem*). Debido a que algunas operaciones sobre ciertos tipos de datos enteros a nivel de *hardware* toman más de un paso, si varios hilos o procesos intentan modificarla en el mismo instante de tiempo, sus respectivas secuencias de pasos se intercalan produciendo resultados impredecibles e inesperados. Por ejemplo, la operación de incremento:

```
int i = 0;
i++;
```

Esta expresión se descompone en los siguientes pasos:

1. Traer el valor de i desde la memoria a un registro interno del procesador.
2. Incrementar el contenido del registro.
3. Escribir el contenido del registro a memoria.

Si tenemos dos hilos, A y B, los cuales a partir de un estado inicial donde i = 0 modifican simultáneamente la variable i mediante la instrucción `i++`. Al terminar A y B la ejecución de dicha instrucción, la variable i debe tener un valor igual a 2. La siguiente secuencia de pasos demuestra que esto puede no ocurrir.

1. Hilo A: trae el valor de i desde la memoria y lo ubica en un registro Ra, Ra = 0.
2. Hilo B: trae el valor de i desde la memoria y lo ubica en un registro Rb, Rb = 0.
3. Hilo A: hace un incremento al registro Ra, quedando Ra = 1.

4. Hilo A: escribe en la dirección de memoria de i el nuevo valor incrementado en el registro Ra, i = Ra, i = 1.

5. Hilo B: hace un incremento al registro Rb, quedando Rb = 1.

6. Hilo B: escribe en la dirección de memoria de i el nuevo valor incrementado en el registro Rb, i = Rb, i = 1.

Como se aprecia, i termina con valor de 1 en tanto que ambos hilos leyeron 0 como valor de i antes de hacer el incremento. Esto es lo que se conoce como pérdida de actualizaciones.
Esta operación de incremento está compuesta de tres pasos: leer, actualizar y escribir. De tal modo, se considera una operación *read-update-write* no atómica. Por tanto, las operaciones de preincremento, postincremento, predecremento y postdecremento pueden generar condiciones de competencia si son usadas sin sincronización cuando se emplean en hilos de ejecución concurrentes y se realizan sobre la misma variable compartida.

2.1.4. Sección Crítica

Eliminar estas intercalaciones no deseadas es conocido como el problema de la exclusión mutua (*mutual exclusion problem*). Para evitar las condiciones de competencia y resultados erróneos, se tienen que identificar en cada hilo aquellas secciones críticas (*critical section*) del código [16].
"Las secciones críticas son aquellas partes del código de un programa en las que se":

1. "referencia una o más variables en forma *read-update-write* mientras cualquiera de esas variables está posiblemente siendo alterada por otro hilo, o"

2. "altera uno o más variables que posiblemente están siendo referenciadas en forma *read-update-write* por otro hilo, o"

3. "usa una estructura de datos, tal como una lista enlazada, mientras una parte de ella está posiblemente siendo alterada por otro hilo, o"

4. "altera cualquier parte de una estructura de datos, tal como un lista enlazada, mientras que está posiblemente siendo usada por otro hilo."

El acceso a esta memoria compartida representa una Sección Crítica en los programas, ya que para acceder a la misma tanto para lectura como modificación de las variables compartidas los hilos compiten [14]. Se llama Sección Crítica a la parte del programa que accede a un recurso compartido, y por tanto, debe ser protegido de tal manera que solo uno de los procesos o hilos pueda acceder a dicha sección a la vez [10]. Según Dijkstra, el criterio de corrección para los procesos o hilos cooperativos que se ejecutan concurrentemente y se sincronizan mediante variables compartidas y Secciones Críticas está dado por los siguientes elementos [10, 15]:

1. Exclusión Mutua: “En cualquier momento, a lo sumo uno de los procesos se encuentra dentro de su sección crítica”.

2. Justicia: “La decisión de cuál es el proceso que es el primero en entrar a su Sección Crítica no puede ser pospuesto eternamente”.

3. Velocidades independientes: “La detención de un proceso que se ejecuta fuera de su Sección Crítica no afecta a los demás procesos”.

2.1.5. No determinismo

La ejecución concurrente o paralela de procesos e hilos es no determinista, en tanto no es posible predecir mediante la revisión de sus especificaciones en qué orden se ejecutan las instrucciones de cada uno. El no determinismo se relaciona con la propiedad de velocidades independientes de los procesos cooperativos que se ejecutan concurrentemente. Dijkstra plantea que no se permiten suposiciones sobre las velocidades de los diferentes procesos cuando estos no se están comunicando explícitamente, y es por ello que los denomina procesos débilmente conectados [10].

2.1.6. Sincronización

Para garantizar la protección en el acceso a las variables compartidas con el objetivo de lograr la exclusión muta es necesaria la sincronización de los procesos. Según la Real Academia de la Lengua Española, sincronizar es “hacer que coincidan en el tiempo dos o más movimientos o fenómenos” [12]. En el contexto de la Computación, se generaliza a la relación entre eventos. Esta relación puede ser antes, durante o después. Por ejemplo, una restricción de sincronización es la exclusión mutua, donde dos eventos no pueden ocurrir a la misma vez [11]. En

el marco del presente trabajo, la exclusión mutua se refiere al acceso exclusivo y controlado a las variables compartidas.
Entre los mecanismos de alto nivel de sincronización de hilos que existen están los semáforos, los *locks* o candados y los monitores. Estos basan su funcionamiento en el uso de primitivas atómicas como *load/store* y *test-and-set* [14]. Una operación es atómica cuando garantiza que la lectura y modificación de la memoria se realizan sin la interferencia de ningún otro proceso [14]. O sea, es vista como una operación indivisible, hay un antes y un después de su ejecución, pero el estado intermedio es inmaterial.
Un semáforo es un entero no negativo con las siguientes características [11]:

1. "Cuando se crea un semáforo, se puede inicializar su valor a cualquier entero, pero luego las únicas operaciones permitidas son incremento y decremento en una unidad. No es posible leer el valor de un semáforo."

2. "Cuando un hilo decrementa el semáforo, si el resultado es negativo, el hilo se bloquea y no puede continuar hasta que otro hilo incrementa el semáforo."

3. "Cuando un hilo incrementa el semáforo, si hay otros hilos esperando, uno de los hilos en espera se desbloquea."

Un *lock*, por su parte, es una versión simplificada de un semáforo, donde su posible valor está limitado a 0 (no bloqueado) o 1 (bloqueado) [31].
Un monitor controla el acceso a un recurso específico, por ejemplo, la memoria, el disco, etc. Entonces, "un monitor consiste en una cierta cantidad de datos de control locales, junto con algunos procedimientos y funciones que son llamadas por programas que quieren adquirir y liberar el recurso controlado por el monitor" [18].
Todas estas primitivas de sincronización comparten la característica de ser bloqueantes. La sincronización mediante estas primitivas consta de los siguientes eventos [14]:

- Adquisición: Cuando un hilo intenta adquirir la primitiva de sincronización.

- Espera: Cuando el hilo "eficientemente" espera a que la primitiva de sincronización esté disponible.

- Liberación: Cuando un hilo ha terminado de operar en la Sección Crítica,

debe informar a los demás que la primitiva de sincronización está disponible.

2.1.7. *Spinlock*

Un *spinlock* es un *lock* que cuando no puede ser adquirido se mantiene intentando su adquisición mediante espera activa (*busy waiting*) hasta que el *lock* es liberado (*spinning*) [17]. Aunque la espera activa desperdicia ciclos del procesador, es útil en dos situaciones: si la Sección Crítica es pequeña, de tal forma que la espera sea menor que el costo de bloquear y reanudar el proceso, o si no hay otros trabajos disponibles. Como inconvenientes, la espera activa desperdicia ciclos del procesador, lo que ralentiza a otros procesadores por el uso del bus de la memoria, y además, sufre de las problemáticas de los métodos de sincronización presentados anteriormente [1].
En [1] y [17] se muestran varias implementaciones del concepto de *spinlock*. Dadas las instrucciones atómicas del tipo *read-modify-write*, es relativamente fácil desarrollar un *spinlock* correcto. Por ejemplo, cada procesador puede ejecutar una instrucción *test-and-set* para adquirir el *lock*; esta instrucción lee el valor anterior del *lock* y lo pone como ocupado. Si la lectura retorna que el *lock* está libre, el procesador obtiene el *lock*; si el *lock* está ocupado, el procesador tiene que intentar adquirirlo de nuevo. El *lock* es liberado (atómicamente), asignándole un valor que indique que está libre [1].
En Java el formato del protocolo de entrada a la Sección Crítica es el siguiente:

```
while (condition) /* do nothing */ ;
```

Se puede apreciar que el ciclo se ejecuta en tanto la condición sea cierta. En este caso, la condición es que hay otros hilos dentro de la Sección Crítica. En Java, un ";"por sí solo significa *noop* (no operation) o no hacer nada. Pero este modo no es la mejor forma de implementarlo en tanto que realiza un uso intensivo del procesador sin hacer trabajo útil. Es mejor al menos, ceder el procesador a otros hilos. En este caso, a los hilos que estén dentro de la Sección Crítica, de tal forma que progresen más rápido. Esto se logra de la siguiente forma [16]:

```
while (condition) Thread.yield();
```

2.2. Programación no bloqueante

El bloqueo es un cambio del estado del proceso en el sistema operativo, pasando a otra cola de procesos. Todo este procedimiento tiene un retardo asociado

(*overhead*) que según la granularidad de la Sección Crítica puede ser mayor o menor. ¿Es posible realizar sincronización sin bloqueos? ¿Es más eficiente? Existen técnicas para evitar el bloqueo. Al conjunto de algoritmos y estructuras de datos que las comprenden se les conoce como programación no bloqueante [17]. Los programas están conformados por algoritmos y estructuras de datos. O sea, algoritmos que operan sobre alguna estructura de datos con un fin determinado [32]. Estas estructuras de datos pueden ser compartidas o no, en tanto accedan a ella uno o varios hilos o procesos.
Existen tres clasificaciones para algoritmos no bloqueantes [17]:

- libre de espera (*wait-free*): plantea que todas las tareas concurrentes terminan en un número finito de pasos.
- libre de bloqueo (*lock-free*): establece que algunas de las tareas concurrentes terminan en un número finito de pasos.
- libre de obstrucción (*obstruction-free*): define que una sola tarea termina en un número finito de pasos.

Como se aprecia, estos criterios clasifican la condición de progreso del cómputo [17]. El presente trabajo se relaciona fundamentalmente con la segunda clasificación: programación libre de bloqueos. Al haber uno o más procesos activos intentando realizar operaciones sobre algún recurso compartido, al menos alguno de los procesos progresa en un número finito de pasos de tiempo [21].

2.2.1. Instrucciones atómicas

Cuando las operaciones sobre los datos compartidos son lo suficientemente simples, estas pueden ser encapsuladas en operaciones atómicas simples. La exclusión mutua es directamente garantizada por el *hardware*. Si varios procesadores simultáneamente intentan actualizar la misma localidad de memoria, cada uno espera su turno sin retornar el control al *software* [1].

2.2.2. Primitiva *Compare and Swap* (CAS)

Para lograr la sincronización libre de bloqueos, se emplea una operación atómica de sincronización denominada *Compare and Swap* (en la arquitectura Intel se conoce como CMPXCHG) [17].
"La primitiva atómica *Compare-And-Swap* (CAS) (figura 2.5) es una instrucción que permite a un procesador, evaluar y modificar una localidad de memoria de

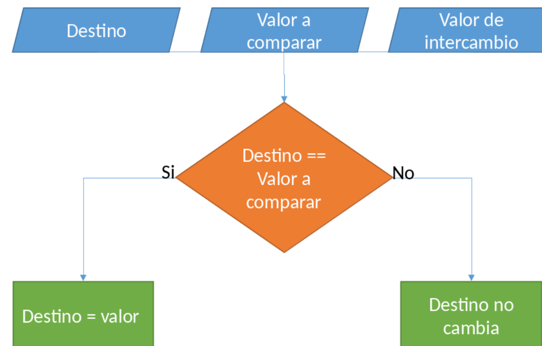

Figura 2.5: Instrucción *Compare and Swap*

tamaño palabra simple de forma atómica" [9]. Según Herlihy, esta operación es universal, en tanto puede implementar otras primitivas atómicas como *test-and-set* o *fetch-and-add* [22] y resuelve el problema del consenso para un número infinito de hilos [17]. Teniendo a disposición esta operación atómica se puede, según demuestran Donald Knuth y Edsger Dijkstra, garantizar sincronización y exclusión mutua [14].

2.2.3. El problema ABA

El empleo de la primitiva CAS trae aparejado el problema ABA. "El problema ABA es una ejecución falso positiva debido al uso de CAS sobre una localidad de memoria compartida" [9].
El escenario de la figura 2.6 muestra una ocurrencia del problema ABA en la implementación de una cola libre de bloqueos. "Un algoritmo no bloqueante es libre del problema ABA cuando su semántica no puede ser corrompida por la ocurrencia del problema ABA" [9].
En [21] este problema se resuelve asignando a cada nodo un contador, de tal forma que se pueda detectar la ocurrencia del problema ABA y actuar en consecuencia.

2.2.4. Biblioteca *Atomic* de Java

Desde la versión 5 del lenguaje de programación Java se añaden las siguientes bibliotecas para la programación concurrente [8]:

- *java.util.concurrent*: Clases utilizadas comúnmente en la programación concurrente.

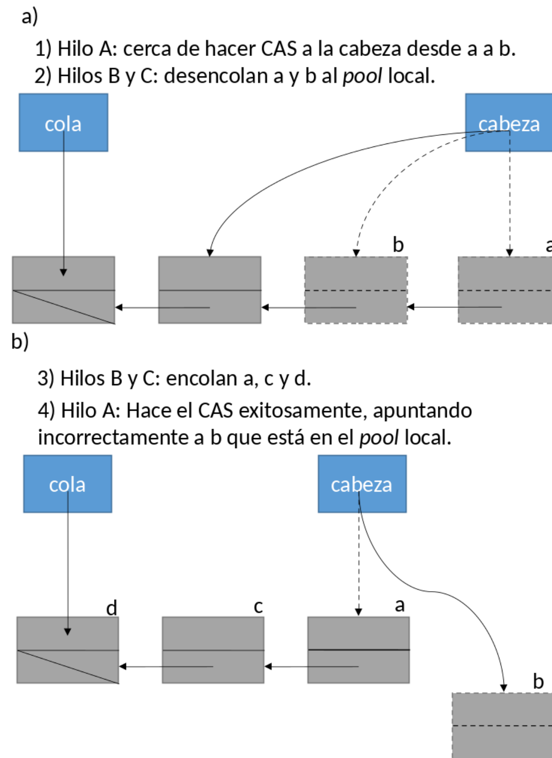

Figura 2.6: Un escenario donde ocurre el problema ABA: en la implementación de una cola (FIFO) libre de bloqueos, se asume que se usa un *pool* (banco) de memoria local de nodos reciclados. En la parte a), el hilo A que desencola, observa que el nodo centinela es a, y el siguiente nodo es b. Paso 1: el hilo A se prepara para actualizar la cabeza aplicando *Compare and Swap* con el viejo valor de a y el nuevo valor de b. Paso 2: antes de que el hilo A continúe, otros hilos desencolan a b y a su sucesor, ubicándolos en el *pool* de memoria libre. En el paso 3 de la parte b): el nodo a es reutilizado, y eventualmente reaparece como el nodo centinela de la cola. Paso 4: el hilo A despierta y llama *Compare and Swap*, siendo exitosa la operación estableciendo la cabeza a b en tanto que el viejo valor de la cabeza sigue siendo a. Ahora, la cabeza está incorrectamente ubicada en un nodo reciclado. (Tomado de [17]).

- *java.util.concurrent.atomic*: Un pequeño toolkit de clases que soporta la programación segura (*thread-safe*) libre de bloqueos (*lock-free*) sobre variables simples.
- *java.util.concurrent.locks*: Interfaces y clases que proveen un marco de trabajo para el bloqueo y la espera por condiciones. Estas no están rela-

cionadas con la sincronización y los monitores integrados al lenguaje.

De estas, es de particular interés e importancia para la presente tesis la biblioteca *java.util.concurrent.atomic* o *Atomic*: las instancias de las clases de *Atomic* mantienen valores que son accedidos y actualizados usando métodos que proclaman ser atómicos, o sea, métodos que pueden ser usados con seguridad de forma concurrente.
Las instancias de las clases *AtomicBoolean*, *AtomicInteger*, *AtomicLong*, y *AtomicReference* proveen acceso y actualización a una variable simple del tipo correspondiente. Cada clase provee métodos utilitarios apropiados para su tipo. Por ejemplo, las clases *AtomicLong* y *AtomicInteger* proveen métodos de incremento atómico. Una posible aplicación es generar una secuencia de números, como se propone en el siguiente código tomado de [8].

```
1  class Sequencer {
2          private final AtomicLong sequenceNumber = new AtomicLong(0);
3          public long next() {
4                  return sequenceNumber.getAndIncrement();
5          }
6  }
```

Las transformaciones arbitrarias del valor contenido en las instancias de estas clases son provistas por operaciones de bajo nivel del tipo *read-modify-write* como *compareAndSet* y por métodos de alto nivel como *getAndUpdate*.
El método compareAndSet() toma dos argumentos: un valor esperado y un valor nuevo para actualizar la variable. Si el valor actual de la variable es igual al valor esperado, entonces el valor de la variable es remplazada por el valor nuevo; si la variable no es igual al valor esperado, entonces el valor de la variable no se modifica. El método retorna un valor booleano indicando si se cambia el valor o no [17].
Estas clases no son reemplazos de propósito general para *java.lang.Integer* y otras clases relacionadas. Son una variante limitada de las mismas ya que, por ejemplo, no definen métodos tales como *equals*, *hashCode* y *compareTo*. Debido a que la intención de uso de las variables atómicas es que sean cambiadas, no deben ser usadas como llaves de tablas hash [8].
Las clases *AtomicIntegerArray*, *AtomicLongArray* y *AtomicReferenceArray* ex-

tienden el soporte de las operaciones atómicas para arreglos de los tipos que indican sus nombres. Las facilidades que ofrecen son el acceso y modificación atómica de los elementos del arreglo que gestionan [8].
En adición a las clases que representan valores y arreglos, este paquete contiene clases *Updater*, que pueden ser usadas para acceder a la operación *compareAndSet* y otras operaciones relacionadas en cualquier campo volátil de cualquier clase. *AtomicReferenceFieldUpdater*, *AtomicIntegerFieldUpdater* y *AtomicLongFieldUpdater* son utilidades basadas en la reflexión, que proveen acceso a los tipos de campos asociados. Estos son usados principalmente en estructuras de datos atómicas en las que varios campos volátiles del mismo nodo (por ejemplo, los enlaces a un nodo de un árbol) son independientemente sujetos a actualizaciones atómicas [8].
La clase *AtomicMarkableReference* asocia un booleano simple con una referencia. Por ejemplo, este bit puede ser usado en una estructura de datos para significar que el objeto que está siendo referenciado ha sido lógicamente eliminado. La clase *AtomicStampedReference* asocial un valor entero con una referencia. Esto puede ser usado, por ejemplo, para representar los números de versión correspondientes a una serie de actualizaciones [8].
Para usar variables con operaciones atómicas en Java se debe tener en cuenta lo especificado en la ayuda de la mencionada biblioteca de atómicos. En la tabla 2.1 se muestran las clases que forman parte de esta biblioteca. De esta lista de utilidades son utilizadas las de valores atómicos, las de arreglos de valores atómicos y las de referencias atómicas.

2.3. Corrección de programas paralelos

"Para que un programa concurrente sea correcto, tiene que ser escrito de tal forma que no sea dependiente de ninguna manera de cuándo ocurren los cambios de contexto, o de cuán largos son los *time slices*, o de la velocidad relativa de los procesadores en sistemas de multiprocesador. Para hacer que los programas concurrentes funcionen correctamente en un multiprocesador o en un único procesador en presencia de cambios de contexto arbitrarios y con cualquier tamaño de *time slice*, necesitamos coordinar o sincronizar la ejecución de los hilos de tal forma que no intenten hacer operaciones *read-update-write* simultáneamente sobre una variable compartida o sobre una estructura de datos. O sea, no puede haber intercalación de instrucciones de dos o más hilos que manipulan una va-

Clase	**Descripción**
AtomicBoolean	Un valor booleano que puede ser actualizado atómicamente.
AtomicInteger	Un valor entero que puede ser actualizado atómicamente.
AtomicIntegerArray	Un arreglo de enteros en el que sus elementos pueden ser actualizados atómicamente.
AtomicIntegerFieldUpdater<T>	Una utilidad que permite actualizar atómicamente atributos, de tipo entero, de una clase.
AtomicLong	Un valor entero largo que puede ser actualizado atómicamente.
AtomicLongArray	Un arreglo de enteros largos en el que sus elementos pueden ser actualizados atómicamente.
AtomicLongFieldUpdater<T>	Una utilidad que permite actualizar atómicamente atributos, de tipo entero largo, de una clase.
AtomicMarkableReference<V>	Una *AtomicMarkableReference* mantiene una referencia a un objeto junto con un bit de marca, que pueden ser actualizados atómicamente.
AtomicReference<V>	Una referencia a objeto que puede ser actualizada atómicamente.
AtomicReferenceArray<E>	Un arreglo de referencias a objetos en el que sus elementos pueden ser actualizados atómicamente.
AtomicReferenceFieldUpdater<T,V>	Una utilidad que permite actualizar atómicamente atributos, de tipo referencia a objetos, de una clase.
AtomicStampedReference<V>	Una *AtomicStampedReference* mantiene una referencia a un objeto junto con un entero “*stamp*”, que pueden ser actualizados atómicamente.
DoubleAccumulator	Una o más variables que juntas mantienen actualizado un valor acumulado de tipo punto flotante de precisión doble usando una función dada.
DoubleAdder	Una o más variables que juntas mantienen un suma de tipo punto flotante de precisión doble que inicialmente tiene valor cero.
LongAccumulator	Una o más variables que juntas mantienen actualizado un valor acumulado de tipo entero largo usando una función dada.
LongAdder	Una o más variables que juntas mantienen una suma de tipo entero largo que inicialmente tiene valor cero.

Cuadro 2.1: Clases principales de la biblioteca *java.util.concurrent.atomic* [8]

riable o una estructura de datos compartida. Estas instrucciones deben hacerse en una acción ininterrumpible, indivisible o atómica"[16].
Cuando se diseña e implementa un programa paralelo, es necesario validar su corrección. Esta corrección se basa en dos categorías de propiedades principales: seguridad (safety) y vitalidad (liveness) [17].
Las propiedades de seguridad plantean cuáles son los eventos negativos que no deberían ocurrir. Por otra parte, las propiedades de vitalidad especifican las cosas positivas que deberían ocurrir [17]. Por ejemplo, aquellos programas paralelos que hacen uso de Secciones Críticas, necesitan secuenciar el acceso a la misma, o sea, que solo un hilo puede entrar a la Sección Crítica a la vez. Esta propiedad se denomina exclusión mutua (mutual exclusion), que al ser una propiedad que de no ser garantizada, el programa provoca inconsistencias en los datos que procesa. Es por tanto, una propiedad de seguridad. Otra propiedad deseable es que el programa sea libre de puntos muertos (*deadlock freedom*). Quiere decir que al menos siempre haya un hilo progresando, de manera que es una propiedad de vitalidad. Y finalmente, está la propiedad de libre de inanición (starvation-freedom o lockout freedom) que plantea que todos los hilos eventualmente pueden progresar, o sea, que ningún hilo debe esperar eternamente para progresar. Se aprecia que esta propiedad engloba a la anterior de libre de puntos muertos. Es también una propiedad de vitalidad [17].

2.4. Presentando los patrones

Existen varias definiciones de lo que es un patrón [23]:

- "Es una regla de tres partes, que expresa una relación entre un cierto contexto, un problema y una solución."
- "Es una solución recurrente a un problema estándar."
- "Es una forma de capturar y sistematizar prácticas probadas en cualquier disciplina."

Para efectos del presente trabajo, se toma finalmente la siguiente definición:
"Un patrón es una relación función-forma que ocurre en un contexto, donde la función está descrita en términos del dominio del problema como un grupo de compromisos sin resolver o fuerzas, y la forma es una estructura descrita en

términos del dominio de solución que alcanza un buen y aceptable equilibrio entre esas fuerzas" [24].

2.4.1. Patrones de *software*

En el marco del desarrollo de *software*, los patrones de *software* identifican un problema recurrente en el contexto del desarrollo de un programa informático y propone una solución al mismo. De tal forma, esta solución puede ser utilizada cada vez que se presenta un problema similar [13]. Los patrones de *software* para la programación concurrente, por su parte, identifican problemas de programación concurrente y paralela, y propone soluciones a los mismos.

2.4.2. Patrones de diseño para la programación paralela

En la programación concurrente y paralela existen tres categorías de patrones asociados con el nivel de abstracción que ofrecen [7]:

- Patrones Arquitectónicos: "expresan un esquema de organización estructural fundamental para los sistemas de *software*. Proveen un conjunto de subsistemas predefinidos, especifican sus responsabilidades, e incluyen reglas y pautas para organizar la relación entre ellos."
- Patrones de Diseño: "proveen un esquema para refinar los subsistemas o componentes de un sistema de *software*, o la relación entre ellos. Describen una estructura comúnmente recurrente de componentes que se comunican que resuelve un problema de diseño general en un contexto particular."
- Modismos: "son patrones de bajo nivel que describen cómo implementar aspectos particulares de componentes de *software* o las relaciones entre estos con las herramientas de un lenguaje de programación dado."

Para los efectos del presente trabajo se toma la palabra modismo como la traducción al español del término en inglés *idiom*.
¿En qué niveles del funcionamiento de un programa paralelo se aplica cada categoría de patrón?

- Coordinación: Se ataca mediante el empleo de patrones arquitectónicos.
- Comunicación: Se ataca mediante la utilización de patrones de *software*.
- Sincronización: Se ataca mediante el uso de modismos de los lenguajes de programación.

El presente trabajo se enmarca en el nivel de modismos de sincronización. Particularmente evitando bloqueos mediante el uso de operaciones atómicas.

2.4.3. Forma POSA

Existen varias formas de describir los patrones de *software*. En el presente trabajo se usa la forma POSA (de *Pattern-Oriented Software Architecture*) que se propone en el libro [7]. La forma POSA tiene las siguientes secciones [23]:

- **Nombre**: El nombre y un resumen corto del patrón.
- **También conocido como**: Otros nombres para el patrón, si hay algún otro conocido.
- **Ejemplo**: Un ejemplo del mundo real en que se demuestre la existencia del problema y la necesidad del patrón.
- **Contexto**: La situación en la cual el patrón se aplica.
- **Problema**: El problema que el patrón resuelve, incluyendo una discusión de sus fuerzas asociadas.
- **Solución**: El principio fundamental de la solución que sirve como base al patrón.
- **Estructura**: Una especificación detallada de los aspectos estructurales del patrón, que incluye un diagrama de clases.
- **Dinámica**: Escenarios típicos que describen el comportamiento en tiempo de ejecución del patrón. Los escenarios se ilustran con diagramas de secuencia de mensajes entre objetos.
- **Implementación**: Guías para la implementación del patrón.
- **Ejemplo resuelto**: Discusión sobre cualquier aspecto importante para resolver el Ejemplo, que no se haya cubierto en las secciones Solución, Estructura, Dinámica e Implementación.
- **Variantes**: Una breve descripción de las variantes o especializaciones del patrón.
- **Usos conocidos**: Ejemplos de uso del patrón, tomados de sistemas existentes.

- **Consecuencias**: Los beneficios que provee el patrón, y cualquier potencial desventaja.
- **Véase también**: Referencias a patrones que resuelven problemas similares, y a patrones que ayudan a refinar el patrón que se describe

La forma POSA es una de las más utilizadas dentro de la descripción de patrones de *software*. Tiene su origen en la comunidad de programación orientada a objetos. Sin embargo, puede utilizarse para describir soluciones exitosas en aplicaciones no necesariamente orientadas a objetos [23].

2.5. Resumen

En el presente capítulo se habla sobre la programación concurrente y paralela, abarcando detalles sobre qué son los procesos y los hilos, variables compartidas, memorias compartida y distribuida, cambios de contexto y condiciones de carrera, secciones críticas, no determinismo, sincronización y *spinlocks*.
Se ofrecen detalles de la programación no bloqueante. Para esto se habla de las instrucciones atómicas, describiendo en particular la instrucción *Compare and Swap* y el problema ABA que surge al emplear dicha operación. Se introduce, además, la biblioteca *Atomic* de Java, la cual provee clases que permiten realizar operaciones atómicas sobre datos a nivel de programación.
Se finaliza con la exposición de los patrones de *software* para la programación paralela, y presentando la forma POSA como formato a utilizar en la descripción de los patrones que resultan del presente trabajo.

Capítulo 3

Trabajo Relacionado

El tema de modismos basados en instrucciones atómicas no fue encontrado en artículos o libros durante el desarrollo del presente trabajo. De hecho, esto fue el acicate fundamental para desarrollar esta tesis, ya que lo que se puede encontrar son usos de las instrucciones atómicas. No obstante, en el presente capítulo se comentan algunos de los trabajos relacionados con la tesis en cuanto a la propiedad de los algoritmos o estructuras de datos de ser no bloqueantes en un ambiente concurrente y/o paralelo para extraer de ellos patrones de uso de las operaciones atómicas presentadas en el Capítulo 2.

3.1. Implementaciones de estructuras de datos libres de bloqueos

Los autores Maged M. Michael y Michael L. Scott, en su trabajo "Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms" [21], proponen la implementación de una cola libre de bloqueos, esto, como alternativa a las implementaciones bloqueantes y a las no bloqueantes, pero deficientemente definidas por otros autores. Para esto, hacen un bosquejo del trabajo previo, el cual describen como no completo, en tanto ninguno ofrece de forma categórica y fácilmente reproducible una implementación de alguna estructura de datos libre de bloqueos. A partir de esto presentan una implementación de una cola libre de bloqueos. Esta implementación es conocida y citada por otros autores. De hecho, es la guía base en la implementación de la cola libre de bloqueos *boost::lockfree::queue* de la biblioteca para C/C++ *Boost*

LockFree y de *ConcurrentLinkedQueue* de Java. Esta cola, tal como lo plantea la teoría asociada, se basa en el uso de la primitiva de *hardware Compare and Swap*. Demuestran su corrección en base a tres aspectos principales:

- Seguridad (safety): Consistencia de los datos de la cola. Manejo del problema ABA.
- Linealización (linearizability): Atomicidad de las operaciones de la cola.
- Vitalidad (liveness): Al menos siempre hay una operación progresando.

Presentan, además, pruebas de rendimiento de esta cola, e implementaciones de diferentes autores, como se muestra en la figura 3.1.

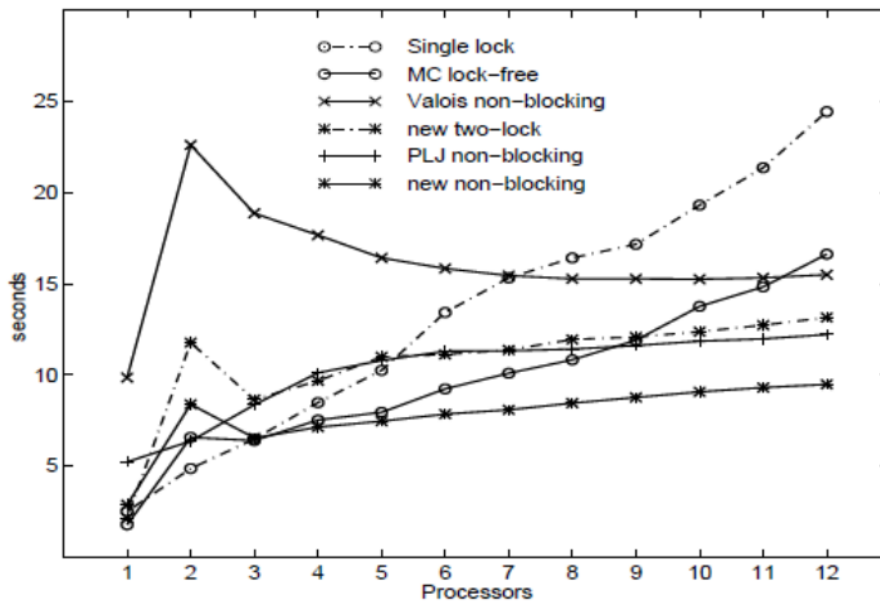

Figura 3.1: Comparación entre la cola propuesta y otras implementaciones

3.2. Planificadores libres de bloqueos

En el trabajo “On the Design and Implementation of an Efficient Lock-Free Scheduler” [22], de los autores Florian Negele, Felix Friedrich, Suwon Oh y Bernhard Egger, proponen un planificador de tareas que tiene la característica de

ser libre de bloqueos, en este caso comparables en eficiencia a otros planificadores modernos, como los de Windows y Linux. El planificador que presentan cumple dos propiedades fundamentales:

- El uso de primitivas no bloqueantes.
- El uso implícito de multitarea cooperativa.

La multitarea cooperativa es aquélla en la que el planificador no interrumpe a las tareas que ejecuta. En cambio, espera a que las tareas terminen para decidir cuál es la próxima que se va a enviar a ejecución. De ahí el nombre de cooperativa, ya que las tareas deben ceder voluntariamente el recurso de procesamiento. Según plantean los autores, se deciden por este esquema cooperativo en tanto uno de sus objetivos es que el planificador sea portable. La implementación de la contraparte apropiativa requiere el manejo de interrupciones en una amplia variedad de *hardware*, lo que hace compleja la implementación y la portabilidad del componente de *software* que se quiere lograr.
Está claro que los planificadores cooperativos son propensos a las fallas en cuanto a que una tarea que nunca devuelva el procesador al planificador compromete la estabilidad del todo el sistema. Para garantizar que esto no ocurra, los autores desarrollan un compilador de C/C++ modificado, de tal forma que, implícitamente, ubica en el código directivas de llamadas al planificador para que ninguna tarea se apropie indefinidamente de los recursos de procesamiento. Luego describen las propiedades no bloqueantes de las estructuras de datos que emplean para la gestión de las tareas. Finalmente realizan pruebas de desempeño del planificador propuesto y se compara con los planificadores de los sistemas Windows y Linux obteniendo resultados similares, y en algunos casos mejores, como se puede apreciar en la figura 3.2. Todo esto con una mayor simplicidad. Luego, los autores se atribuyen las siguientes contribuciones:

- El uso del espacio de almacenamiento local del procesador en lugar del almacenamiento local de los hilos.
- Implementación eficiente de una cola libre de bloqueos con reutilización de nodos.
- La creación del concepto de *Task Switch Finalizer* (Finalizador de cambio de tarea).
- El diseño e implementación de un planificador simple, ligero, confiable y portable.

Benchmark	Native	Linux	Windows
City ($N = 1000, K = 10, C = 100$)	61ms	63ms	138ms
Eratosthenes ($N = 10000$)	3'629ms	4'750ms	6'347ms
Mandelbrot ($N = 100, C = 2000, K = 5000$)	4'066ms	5'287ms	5'040ms
News ($N = 1000, M = 10, K = 10$)	1'260ms	374ms	280ms
Producer ($N = 1, C = 10, K = 10000$)	1'382ms	269ms	225ms
Producer ($N = 64, C = 10, K = 10000$)	54'495ms	105'086ms	31'032ms
Token Ring ($N = 1000, K = 1000$)	4'506ms	15'672ms	8'759ms

Figura 3.2: Comparación del tiempo de ejecución entre el planificador cooperativo libre de bloqueos, denotado como *Native*, y los núcleos de los sistemas operativos Linux y Windows.

También comentan las desventajas de la multitarea cooperativa, en tanto requiere la cooperación implícita mediante la adaptación de un compilador para lograrlo.

3.3. Resumen

En el presente capítulo se comentan un par de artículos representativos que se relacionan con la tesis en cuanto a estructuras de datos con la propiedad libre de bloqueos y planificadores. Estos permiten extraer patrones de uso en forma de modismos de las operaciones atómicas sobre datos, en particular sobre referencias a objetos.

Se analiza en primer lugar el artículo "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms" de los autores Maged M. Michael y Michael L. Scott. En este artículo se presenta una implementación de una cola libre de bloqueos basada en la instrucción atómica *Compare and Swap*. Esta implementación resulta tan o más rápida que otras implementaciones propuestas por otros autores.

Luego se analiza el artículo "On the Desing and Implementation of an Efficient Lock-Free Scheduler" de los autores Florian Negele, Felix Friedrich, Suwon Oh y Bernhard Egger. En este artículo los autores exponen el diseño e implementación de un planificador cooperativo basado en una cola libre de bloqueos que obtiene resultados similares en algunos casos y mejores en otros en comparación con los planificadores de sistemas operativos como Linux o Windows.

Capítulo 4

Modismos basados en tipos de datos atómicos

En el presente capítulo se presentan modismos basados en el uso de las variables atómicas de Java. El formato de presentación de patrones que se sigue es la forma POSA, tal como se presenta en la sección 2.4.3.
Primeramente, se describe el modismo *Atomic Flag*, el cual consiste en la definición y empleo conveniente de un objeto de tipo *AtomicBoolean* de Java como una bandera. La lectura y manipulación de la misma es *thread safe*.
Luego se describe el modismo *Atomic Counter*, como variable compartida que ejerce como contador. Esto se logra sin bloqueos a nivel de *software*, o sea, para realizar su cometido no es requerida la sincronización a nivel de programación. Por tanto, este contador no constituye una sección crítica en un programa concurrente.
Seguido se presenta el modismo *Atomic Counters Array*, que describe el empleo de un arreglo de enteros atómicos que puede ser usado, por ejemplo, en un algoritmo de ordenamiento por conteo (*Counting Sort*) concurrente.
Finalmente, se presenta el modismo *Atomic Node*, que muestra el concepto de nodo de una estructura de datos implementado con referencias atómicas y que permite su modificación y lectura de forma atómica.
Estos modismos hacen uso de variables con operaciones atómicas que provee el lenguaje de programación *Java* en su biblioteca estándar. Estas ideas se pueden portar al lenguaje de programación C++, que desde su especificación del 2011 incluye una biblioteca de atómicos similar a la que se emplea en el presente

trabajo.

4.1. Modismo *Atomic Flag*

El modismo *Atomic Flag* es una forma libre de bloqueos, y a la vez, *thread safe* para leer y modificar una variable booleana desde dos o más componentes de *software* concurrentes y/o paralelos, que se ejecutan en una plataforma de memoria compartida. Es una técnica conocida que, en este caso, se presenta como modismo empleando objetos con operaciones atómicas de Java.

Ejemplo: Considere un componente *pipe* basado en el patrón *Shared Variable Pipe* (Figura 4.1) para la comunicación de procesos concurrentes o paralelos [24]. Este componente *pipe* puede ser empleado como elemento de sincronización en el patrón arquitectónico de coordinación *Parallel Pipes and Filters*. Este patrón arquitectónico está basado en variables compartidas y comunicación asíncrona. En su presentación original, las *Shared Variable Pipe* se implementan usando monitores de Java para proteger un búfer compartido. El modismo *Atomic Flag* es empleando para implementar un *spinlock* en sustitución del monitor que protege el búfer. Por tanto, este spinlock es intercambiable con modismos de sincronización como los semáforos o regiones críticas y puede ser empleado para implementar monitores. ¿Cómo sincronizar el *Shared Variable Pipe* empleando *Atomic Flag*?

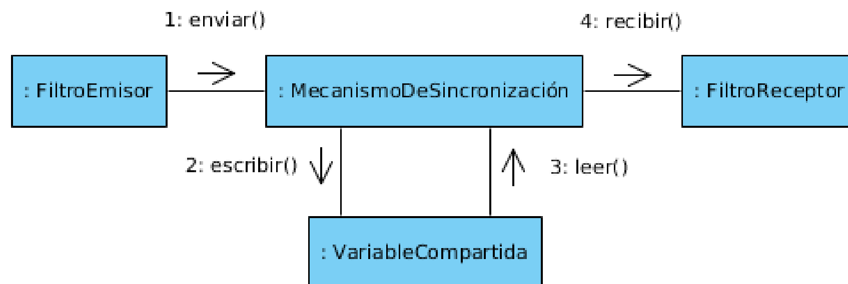

Figura 4.1: Diagrama de colaboración del modismo *Shared Variable Pipe* [24].

Contexto: Un programa concurrente y/o paralelo escrito en Java, que hace

uso de variables booleanas compartidas para indicar estados del sistema o cualquier otro estado específico. Estas variables son leídas y modificadas por dos o más hilos de ejecución que compiten por acceder a ellas.

Problema: Para preservar la integridad de las variables booleanas compartidas, así como el orden de las operaciones sobre ellas, es necesario dotar a los componentes de *software* que se ejecutan en diferentes hilos de procesamiento de acceso seguro a dichas variables compartidas, para un número arbitrario de operaciones de lecturas y escrituras [24].

Fuerzas: Para hacer un programa concurrente aplicando el presente modismo se deben tener en cuenta y balancear las siguientes fuerzas [10, 24]:

- El programa que se desarrolla requiere de estados binarios compartidos entre múltiples hilos de ejecución.
- Los hilos de ejecución del programa se ejecutan concurrentemente a diferentes velocidades relativas y de forma no determinista.
- Las operaciones de inspección y asignación para los propósitos de la sincronización son definidas como atómicas o indivisibles.
- La integridad de los valores de una variable compartida debe ser preservada siempre.
- El grado de concurrencia puede ser alto, o sea, el programa puede realizar un gran número de accesos al *Atomic Flag*.

Solución: Usar *Atomic Flag* para sincronizar de forma segura el acceso a las variables booleanas desde varios hilos de ejecución. Mediante su uso se espera como resultado hacer más veloz a aquellos programas paralelos escritos en lenguaje de programación Java que resuelven problemas enmarcados en el Contexto del patrón.
Para crear un *Atomic Flag*, se crea un objeto de la clase *AtomicBoolean*. En este caso con nombre *flag*. El objeto (indistintamente llamado variable, ya que es lo que encapsula) *flag* es creado e inicializado con valor *true* de la siguiente forma:

```
AtomicBoolean flag = new AtomicBoolean(true);
```

Sobre la variable *flag* se pueden realizar las siguientes operaciones atómicas, tanto para su lectura como para para su modificación, definidas en la especificación del lenguaje [8]:

- Para modificar la variable sin importar el valor que tiene se emplea el siguiente enunciado:

  ```
  flag.set(boolean newValue);
  ```

  En este enunciado se llama al método *set* sobre *flag* y le pasa como parámetro el nuevo valor de *flag*.

- Para leer la variable se emplea el enunciado:

  ```
  flag.get();
  ```

  En este enunciado se llama al método *get* sobre *flag* el cual devuelve el valor actual de *flag*.

- Para modificar la variable sin importar el valor que tiene y obtener el valor que tenía antes de la modificación, se emplea el siguiente enunciado:

  ```
  flag.getAndSet(boolean newValue);
  ```

  En este enunciado se llama al método *getAndSet* sobre *flag* y le pasa como parámetro el nuevo valor de *flag*. El método *getAndSet* devuelve el valor anterior a la modificación.

- Para modificar la variable en dependencia del valor que tiene actualmente se emplea el siguiente enunciado:

  ```
  flag.compareAndSet(boolean expectedValue, boolean newValue);
  ```

  En este enunciado se llama al método *compareAndSet* sobre *flag* y se le pasan dos parámetros: el primero es el valor esperado que debe tener *flag* para poder ser modificado con el nuevo valor dado como segundo parámetro. El método *compareAndSet* devuelve un booleano indicando si se lleva a cabo la modificación o no.

Estructura: La figura 4.2 muestra el concepto de variable booleana atómica como un tipo de dato abstracto con un valor *booleano* y las operaciones descritas.

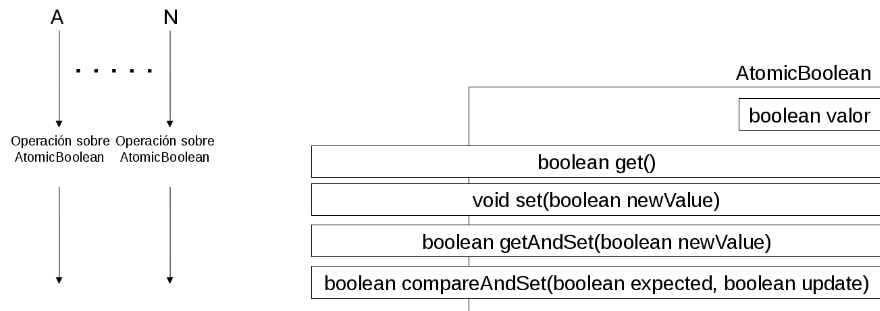

Figura 4.2: Un diagrama representando un *AtomicBoolean* como un tipo de dato abstracto.

En el lenguaje de programación Java algunos usos típicos de este modismo se muestran en el siguiente fragmento de código:

```
import java.util.concurrent.atomic.AtomicBoolean;

public class JavaAtomicFlagExample {
    static AtomicBoolean flag = new AtomicBoolean(true);
    .
    public static void main(String args[]) {
    }
    public static void concurrentMethod1(){
        if (flag.get()) // Hacer algo
        else // Hacer otra cosa
    }
    public static void concurrentMethod2(){
        boolean condition = ...;
        flag.set(condition);
    }
    public static void concurrentMethod3(){
        boolean condition = ...;
        if (flag.getAndSet(condition)) // Hacer algo
        else // Hacer otra cosa
    }
    public static void concurrentMethod3(){
        boolean expected = ...;
        boolean condition = ...;
```

```
        while (!flag.compareAndSet(expected, condition)) { // Hacer algo }
    }
}
```

Dinámica: El *Atomic Flag* puede ser empleado principalmente en el siguiente escenario, caracterizado por la existencia de pocos procesos y regiones críticas de granularidad fina.

- Exclusión Mutua: En la figura 4.3 se presenta un diagrama de secuencia UML que muestra dos componentes, A y B, que comparten datos que requieren acceso exclusivo. El acceso a los datos compartidos está protegido por una variable atómica *mutex* inicializada con valor `true`.
  El componente A adquiere el *mutex* asignándole valor `false`. Esto le da paso a la sección crítica. Luego el componente B intenta adquirir el *mutex* mediante una espera activa que termina cuando el componente A libera el acceso a la sección crítica asignando el valor *false*.

Ejemplo Resuelto: El siguiente código muestra la implementación del patrón de diseño para el componente de comunicación *Shared Variable Pipe* empleando *Atomic Flag*, basado en un objeto de tipo *AtomicBoolean* y con nombre *mutex*. Como se aprecia, los métodos no están sincronizados al estilo Java, de tal forma que pueden haber más de dos hilos a la vez intentando progresar dentro de los métodos. En el caso de que el búfer esté lleno al intentar enviar o vacío al intentar recibir, se lanza una excepción, siempre liberando antes al *lock mutex*.

```
import java.util.ArrayList;
import java.util.concurrent.atomic.AtomicBoolean;

public class SharedVariablePipe {
    static final int MAXSIZE = 100;
    private ArrayList<Double> buffer = new ArrayList<Double>();
    AtomicBoolean mutex = new AtomicBoolean(true);
    public void send(double data) {
        while (!mutex.compareAndSet(true, false));
        if (buffer.size() == MAXSIZE) {
            mutex.set(true);
            throw new FullStackException();
        }
        buffer.add(data);
        mutex.set(true);
    }
    public double receive() {
        while (!mutex.compareAndSet(true, false));
```

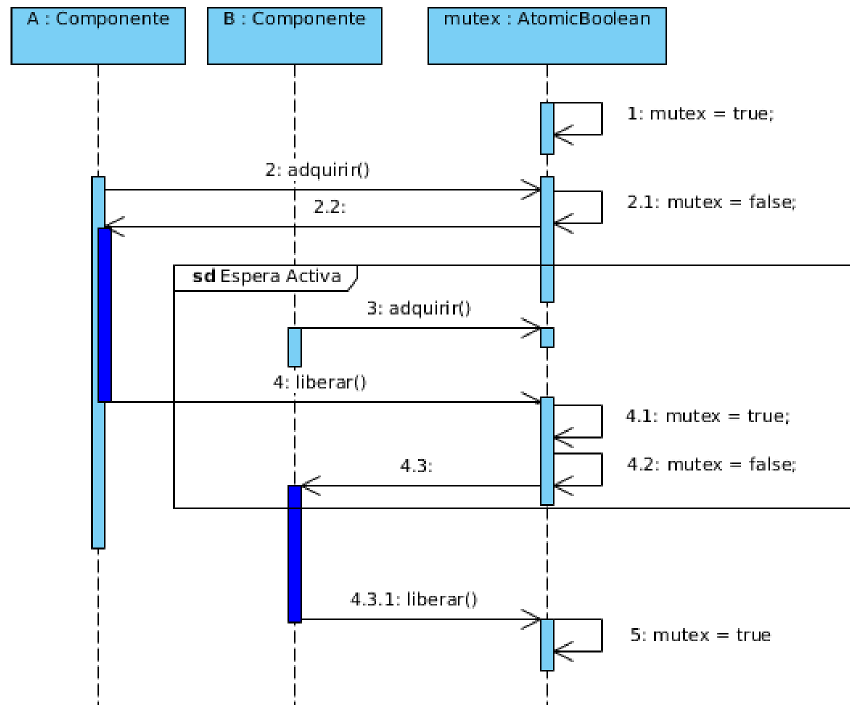

Figura 4.3: Diagrama de secuencia de dos componentes de *software* concurrentes o paralelos que acceden a su sección crítica protegidos por un *Atomic Flag*.

```
        if (buffer.size() == 0) {
            mutex.set(true);
            throw new EmptyStackException();
        }

        double data = ((Double)buffer.remove(0)).doubleValue();
        mutex.set(true);
        return data;
    }
}
```

Usos conocidos: Algunos programas que hacen uso de *Atomic Flag* en [17] son:

- *Test And Set Spinlock*: *Spinlock* implementado como en el ejemplo presentado [17].
- *Test And Test And Set Spinlock*: *Spinlock* implementado con una opera-

ción *get* sobre el *Atomic Flag*. Este programa presenta granularidad fina y se corre en memoria compartida [17].

- *Semáforos binarios*: Los semáforos binarios se implementan con variables booleanas modificadas atómicamente. A diferencia de los *spinlocks*, se bloquea cuando la variable no tiene el valor esperado.
- *Variables que indican el estado de un recurso*: El típico uso de una bandera [30].

Consecuencias:
Ventajas:

- Mejora los tiempos de ejecución en casos de alta concurrencia.
- Es escalable en cuanto a que su funcionamiento es correcto, independientemente de la cantidad de hilos que lo emplean.
- No se utiliza ningún tipo de bloqueo a nivel de programación para leer y/o modificar la bandera. En consecuencia no se realiza ninguna llamada al sistema operativo.

Desventajas:

- En el caso de su uso como *spinlock*, su principal desventaja radica en el uso de la espera activa. Esta tiene un impacto mayor cuando el procesamiento en la sección crítica es de granularidad gruesa, debido a que la espera activa se vuelve más intensa. Es por ello que su uso debe ser evaluado. Para reducir el impacto de este problema, se debe evaluar el uso de la operación `yield()` en los hilos que esperan para que liberen el *CPU* para uso de los hilos que si están progresando. Se debe analizar la factibilidad de usar priorización dinámica de los hilos.

Patrones relacionados: El modismo *Atomic Flag* usado como *spinlock* está relacionado con los patrones de sincronización de hilos y/o procesos concurrentes y/o paralelos. Algunos patrones relacionados son el semáforo, la región crítica y el monitor [24]. También se relaciona con otros modismos presentados en esta tesis como el *Atomic Counter*.

4.2. Modismo *Atomic Counter*

El modismo *Atomic Counter* es una forma de manejar contadores enteros sin sincronización a nivel de programación. Esto permite que dos o más componentes de *software* concurrentes y/o paralelos puedan llevar a cabo incrementos y/o decrementos simultáneamente sin bloquearse.

Ejemplo: Considérese un contador de instancias creadas de una clase desde el inicio de un programa paralelo en Java (figura 4.4), que puede ser manipulado de forma concurrente. No es un contador de instancias activas en memoria debido a que el método *finalize* de la super clase de Java *Object* no es seguro [8], por tanto, no se decrementa en el caso de que los objetos sean eliminados por el recolector de basura de la Máquina Virtual de Java. En el caso de C++, esto si es posible en tanto el programador tiene control total sobre los constructores y el destructor de la clase.

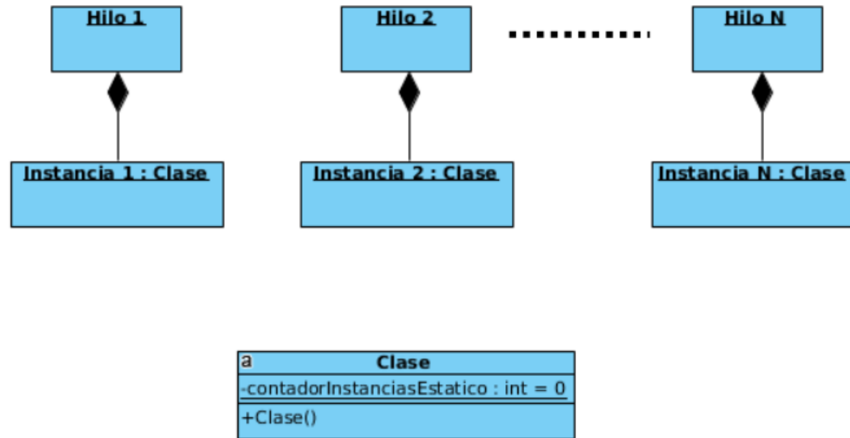

Figura 4.4: Diagrama de objetos de un contador de instancias creadas. En el caso que se muestra, en cada llamada al constructor de Clase el atributo estático `contadorInstanciasEstatico` es incrementado. De tal forma, el valor del contador es N.

Contexto: Un programa escrito en algún lenguaje de alto nivel con soporte para variables con operaciones atómicas sobre ellas, como C++ o Java, en el cual se requiere el empleo de variables contadoras con actualización concurrente

y libre de bloqueos.

Problema: Evitar condiciones de competencia que provoquen la pérdida de actualizaciones. O sea, preservar la integridad y consistencia de la variable contador.

Fuerzas: Para hacer un programa concurrente sincronizado mediante variables con operaciones atómicas se deben balancear las siguientes fuerzas:

- La variable contador puede ser leída y modificada por cualquier cantidad de hilos de ejecución.
- La variable contador solo puede ser incrementada, decrementada y leída.
- La solución debe ser intercambiable con contadores protegidos en las operaciones sobre él con modismos de sincronización como los semáforos, etc.

Solución: Usar variables con operaciones atómicas para lograr el efecto de un contador actualizado sin bloqueos a nivel de *software*. En el caso del lenguaje de programación Java tenemos las siguientes posibilidades [8]:

- Clase *AtomicInteger*: Clase que permite mantener un valor entero modificable con operaciones atómicas. O sea, no requieren sincronización a nivel de *software*.
- Clase *AtomicLong*: Clase que permite mantener un valor entero largo modificable con operaciones atómicas. O sea, no requieren sincronización a nivel de *software*.

En ambos casos contamos con las siguientes operaciones para modificar y leer la variable contador atómica [8].

- Método `int get()`: Devuelve el valor actual de la variable atómica.
- Método `int incrementAndGet()`: Incrementa de forma atómica en una unidad el valor de la variable y devuelve el nuevo valor incrementado.
- Método `int getAndIncrement()`: Incrementa de forma atómica en una unidad el valor de la variable y devuelve el valor anterior al incremento.
- Método `int decrementAndGet()`: Decrementa de forma atómica en una unidad el valor de la variable y devuelve el nuevo valor decrementado.

- Método `int getAndDecrement()`: Decrementa de forma atómica en una unidad el valor de la variable y devuelve el valor anterior al decrementado.

Las operaciones anteriores de incremento y decremento en una unidad son posibles de realizar de forma análoga con otros métodos que ofrece el *AtomicInteger*. Por ejemplo, la operación de incremento en uno puede lograrse además de las siguientes formas:

- Método `int accumulateAndGet(1, (x,y) ->x + y)`: Aplica de forma atómica la función lambda binaria `(x,y) ->x + y)` (en este caso suma los parámetros recibidos) tomando como parámetros el valor contenido en el objeto *Atomic Integer* y el primer parámetro del método `accumulateAndGet` (en este caso 1). Devuelve el valor luego de aplicar la operación.

- Método `int addAndGet(1)`: Suma de forma atómica el valor que se pasa por parámetro y devuelve el nuevo valor actualizado. Si se pasa un valor de uno se logra el efecto de incremento en una unidad.

- Método `int updateAndGet((x) ->x + 1)`: Aplica de forma atómica la función unaria (que recibe un único parámetro) que se pasa por parámetro al valor contenido en el entero atómico.

- Método `bool compareAndSet(int expectedValue, int newValue)`: De forma atómica compara el valor del entero atómico con el valor *expectedValue*, y de ser iguales lo actualiza con el valor del parámetro *newValue*. Para usar este método en un incremento se debe ser más cuidadoso que con las otras variantes ya que no es tan intuitivo. Sigue un ejemplo de cómo lograr un incremento empleando `compareAndSet`;

```
AtomicInteger ai = new AtomicInteger();
.
.
.
int currentValue;
int incrementedCurrentValue;
do
{
        currentValue = ai.get();
```

```
        incrementedCurrentValue = currentValue + 1;
}
while(!ai.compareAndSet(currentValue, incrementedCurrentValue));
```

Para el caso del decremento en una unidad, se puede derivar de las formas alternativas anteriores.

Estructura: La figura 4.5 muestra a una variable con operaciones atómicas en Java como un tipo de dato abstracto que tiene un valor y una interfaz con las operaciones de incrementar, decrementar y obtener valor.

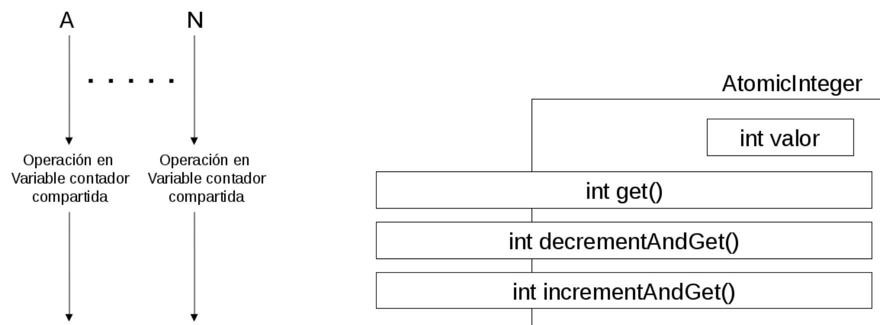

Figura 4.5: Un diagrama representando un contador atómico como un tipo de dato abstracto.

A nivel de código el uso de las mismas es el siguiente:

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicIntegerTester {
        static AtomicInteger ai = new AtomicInteger();
}

public void concurrentIncrement() {
        ai.incrementAndGet();
}

public void concurrentDecrement() {
        ai.decrementAndGet();
}
```

```
public int concurrentGet() {
        return ai.get();
}
```

Dinámica: El *Atomic Counter* puede ser empleado en cualquier escenario donde es necesario el empleo de un contador que puede ser actualizado de forma concurrente desde varios hilos. El diagrama de secuencia representado en la figura 4.6 muestra la interacción de dos componentes (A y B) con la variable entera atómica compartida av.

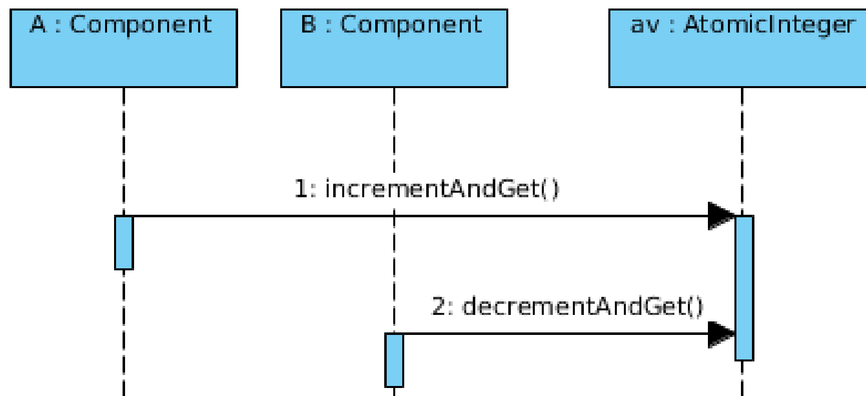

Figura 4.6: Diagrama de secuencia de la interacción de varios componentes de *software* con una variable entera atómica. Se aprecia que no hay sincronización a nivel de *software* en ningún momento. La cantidad de componentes puede ser n y las operaciones sobre la variable atómica pueden ser de cualquier tipo (incremento, decremento, obtención).

Ejemplo resuelto: A continuación, se muestra la implementación de una clase *MyObject* que hace uso de un contador atómico para contar las instancias creadas de la clase sin requerir sincronización desde la perspectiva del programador.

```
class MyObject
{
        static AtomicInteger instances = new AtomicInteger();
        MyObject()
        {
                instances.incrementAndGet();
        }
```

```
        public static int getInstances()
        {
                return instances.get();
        }
}
```

Usos conocidos: Algunos problemas que se pueden resolver empleando *Atomic Counter* son:

- Contador de referencias en un puntero de C++. O sea, que lleva la cuenta de cuántas referencias existen en el programa a la variable en cuestión. En tal caso de que este contador llegue a 0 se autodestruye la variable. Este tipo de variables con contador de referencias se conocen también como punteros inteligentes. Esto implica un comportamiento parecido a un recolector de basura.
- Implementación concurrente de un patrón *Singleton*.
- Cualquier tipo de estadísticas de ejecución en un programa.
- Implementación de un semáforo general.

Consecuencias:

Ventajas:

- Mejora los tiempos de ejecución en casos de alta concurrencia con independencia de la cantidad de hilos de ejecución.
- Es escalable en cuanto a que su funcionamiento es correcto independientemente de la cantidad de hilos que lo emplean.
- Se elimina el concepto de sección crítica para la manipulación del contador atómico. Es decir no se requiere sincronización a nivel de *software* para actualizar el *Atomic Counter*.
- Posibilidad de sobrecargar los operadores preincremento ($++var$), postincremento ($var++$), predecremento ($--var$), postdecremento ($var--$) en el lenguaje C++.

Desventajas:

- Imposibilidad de sobrecargar los operadores preincremento ($++var$), postincremento ($var{+}{+}$), predecremento ($-{-}var$), postdecremento ($var{-}{-}$) en Java, en tanto no lo permite el lenguaje.

Patrones relacionados: El modismo *Atomic Counter* está relacionado con el modismo *Atomic Flag*.

4.3. Modismo *Atomic Counters Array*

El modismo *Atomic Counters Array* es una forma libre de bloqueos de mantener un conjunto de contadores que pueden ser modificados de forma atómica. Esto implica que no es requerida ninguna sincronización a nivel de programación para acceder a cualquier posición del arreglo.

Ejemplo: Considere un algoritmo de ordenamiento por conteo (*Counting Sort*) paralelo. El ordenamiento por conteo tiene complejidad O(n). Su desventaja es que solo puede ser usado cuando los valores a ordenar pertenencen a un conjunto discreto acotado a un rango que permita la creación de un contador para cada posible valor en el arreglo a ordenar. Por ejemplo, números del 1 al 1000. Es posible paralelizar este algoritmo con m hilos de ejecución, cada uno contando un rango de aproximadamente n/m valores. En la figura 4.7 se presenta los elementos del ordenamiento por conteo.

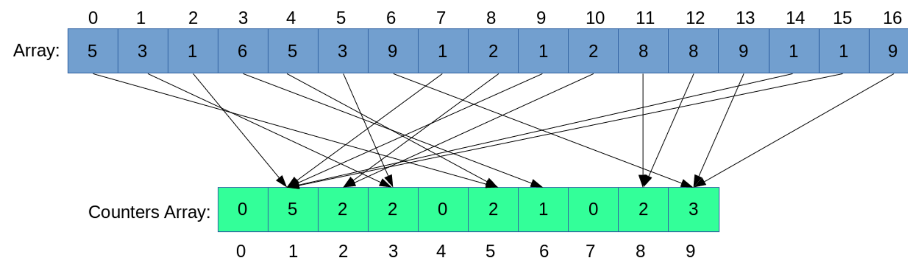

Figura 4.7: Representación del *Counting Sort* donde *Array* es el arreglo a ordenar y *Counters Array* es el arreglo de contadores.

Contexto: Un programa concurrente escrito en Java, donde se requiere realizar cualquier tipo de conteo de diversos elementos acotados en un rango limitado.

Problema: Para preservar la integridad de los valores de los contadores, es necesario dar a un conjunto de componentes de *software* acceso seguro a las

variables compartidas para un número arbitrario de operaciones de lecturas y escrituras [24] sin emplear sincronización a nivel de *software* en Java.

Fuerzas: Para hacer un programa concurrente aplicando el presente modismo se deben tener en cuenta y balancear las siguientes fuerzas:

- Los componentes de *software* se ejecutan concurrentemente a diferentes velocidades relativas, y de forma no determinista. Su sincronización debe ser tan independiente como sea posible de cualquier patrón de interacción o acción de cualquier otro componente de *software*.
- Las operaciones de inspección y asignación en los contadores son definidas como atómicas o indivisibles.
- Cada componente de *software* debe ser capaz de modificar cualquiera de las posiciones del arreglo compartido de contadores atómicos de forma segura.
- La integridad de los contadores atómicos en el arreglo debe ser preservada durante toda la ejecución del programa.
- La solución prescinde del empleo de modismos de sincronización bloqueantes como los semáforos, regiones críticas y monitores.
- La eficiencia en la modificación y lectura de las posiciones en el arreglo de contadores atómicos se degrada cuando hay gran contención. O sea, cuando muchos hilos intentan modificar la misma posición simultáneamente y a nivel de *hardware* se serializan los accesos. Sin embargo, la probabilidad de una gran contención es pequeña y de una gran contención durante un tiempo largo es aún menor.

Solución: Usar un arreglo de variables enteras atómicas en Java. La particularidad de esta opción radica en que los hilos de ejecución pueden manipular cualquier posición del arreglo sin bloquearse. Quiere decir que no hay sincronización a nivel de *software*. Mediante su uso, se espera como resultado hacer más veloz a aquellos programas paralelos escritos en lenguaje de programación Java, donde se requiere realizar cualquier tipo de conteo de valores discretos acotados a un rango manejable en memoria. Los arreglos pueden ser de los tipos *AtomicIntegerArray* y *AtomicLongArray*. Para los ejemplos de la presente tesis empleamos el primero. Un arreglo de enteros atómicos se declara y define de la siguiente manera:

```
private AtomicIntegerArray buffer = new AtomicIntegerArray(100);
```

Luego, para operar en cada una de las posiciones según requieran los hilos de ejecución del programa se emplean las siguientes operaciones:

- Método `int incrementAndGet(int i)`: De forma atómica incrementa el elemento en la posición i del arreglo.
- Método `int decrementAndGet(int i)`: De forma atómica decrementa el elemento en la posición i del arreglo.

Estructura: La figura 4.8 muestra el concepto de arreglo de variable enteras atómicas como un tipo de dato abstracto con un arreglo interno de *AtomicInteger* y con tres métodos fundamentales para la interacción con cada uno de los elementos tratándolos como contadores:

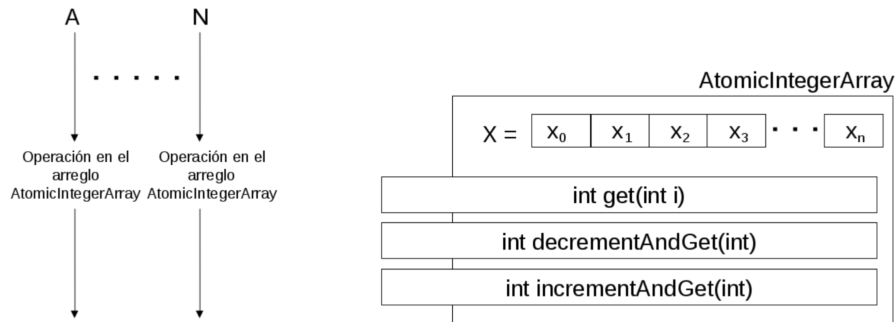

Figura 4.8: Representación de un arreglo de contadores enteros atómicos como un tipo de dato abstracto.

En el lenguaje de programación *Java*, el uso típico de este modismo se muestra en el siguiente fragmento de código. Asumimos el *buffer* creado como se muestra anteriormente. Se muestran tres posibles hilos que realizan las operaciones fundamentales sobre este arreglo. Pueden ser usados en cualquier combinación y cantidad según necesite el programador dado un problema específico.

```
private class IncrementerThread implements Runnable
{
        public void run() {
            /* Hace algo */
            while (true)
            {
                /* Hace algo */
                buffer.incrementAndGet(position);
                /* Hace algo */
            }
            /* Hace algo */
        }
}
private class DecrementerThread implements Runnable
{
        public void run() {
            /* Hace algo */
            while (true)
            {
                /* Hace algo */
                buffer.decrementAndGet(position);
                /* Hace algo */
            }
            /* Hace algo */
        }
}
private class ConsumerThread implements Runnable
{
        public void run() {
            /* Hace algo */
            while (true)
            {
                /* Hace algo */
                buffer.get(position);
                /* Hace algo */
            }
            /* Hace algo */
        }
}
```

Dinámica: La dinámica que ocurre en el uso de un *Atomic Counters Array* se puede apreciar en el diagrama de secuencia de la figura 4.9, donde dos componentes de *software*, que en la práctica son hilos de ejecución, interactúan con

el arreglo atómico. Este arreglo atómico en el estilo de programación basada en bloqueos para lograr exclusión mutua representa la sección crítica. Pero, como se aprecia en el diagrama, nunca ocurren bloqueos a nivel de *software*. De hecho, se aprecia además que las llamadas 3 y 4 son concurrentes y aún así no hay bloqueo para el programador.

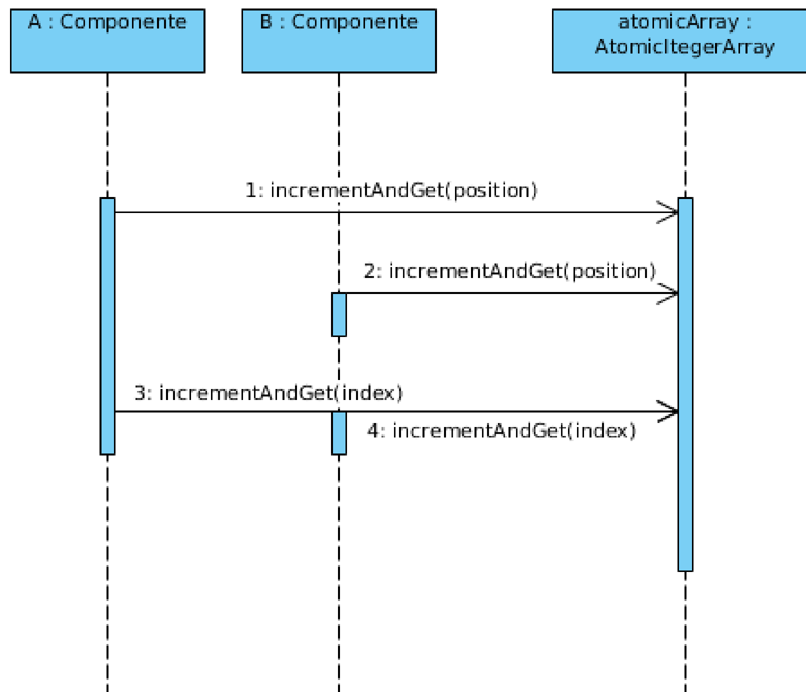

Figura 4.9: Diagrama de secuencia simplificado de dos componentes de *software* concurrentes y/o paralelos que acceden al arreglo de contadores atómicos, su sección crítica.

Ejemplo Resuelto: El siguiente código muestra la implementación de una clase para ordenar de forma paralela empleando el algoritmo *Counting Sort*. Como se aprecia, el acceso de los hilos de ejecución a los métodos no está sincronizado al estilo Java, de tal forma que pueden haber más de dos hilos a la vez intentando progresar.

```
import java.util.List;
import java.util.concurrent.atomic.AtomicIntegerArray;
import java.util.concurrent.ThreadLocalRandom;
public class ConcurrentCountingSort {
    private AtomicIntegerArray atomicCountersArray = null;
    public final int NUMOFSORTERS =
     ↪  Runtime.getRuntime().availableProcessors();
    Thread[] sorters = new Thread[NUMOFSORTERS];
    private List<Integer> array;
    private int minValue;
    private int maxValue;
    ConcurrentCountingSort(int min, int max)
    {
        atomicCountersArray = new AtomicIntegerArray(max - min + 1);
        minValue = min;
        maxValue = max;
    }
    private class SubArraySorter implements Runnable
    {
        private int begin;
        private int end;
        SubArraySorter(int b, int e)
        {
            begin = b;
            end = e;
        }
        public void run() {
            for (int i = begin; i <= end; ++i) {
                atomicCountersArray.incrementAndGet(1 +
                 ↪  ThreadLocalRandom.current().nextInt(maxValue) -
                 ↪  minValue);
            }
        }
    }
    // Continúa en el siguiente fragmento de código
}
```

```
public void sort(/*List<Integer> input*/) throws InterruptedException
{
    int elementsPerSorter = maxValue / NUMOFSORTERS;
    int begin = 0;
    int end = elementsPerSorter - 1;
    for (int i = 0; i < NUMOFSORTERS; ++i) {
        sorters[i] = new Thread(new SubArraySorter(begin, end));
        sorters[i].start();
        begin = end + 1;
        end += elementsPerSorter;
    }
    for (int i = 0; i < NUMOFSORTERS; ++i) {
        sorters[i].join();
    }
}
```

Usos conocidos: Algunos programas que hacen uso de *Atomic Counters Array* son:

- *Estadísticas*: Estadísticas de tiempo de ejecución en un programa concurrente.

Consecuencias:
Ventajas:

- Mejora los tiempos de ejecución de los algoritmos que ameritan su uso.
- Es escalable en cuanto a que su funcionamiento es correcto independientemente de la cantidad de hilos que lo emplean.
- Aunque hay una sección crítica esta no es gestionada desde el nivel de *software*.

Desventajas:

- Aplicación muy específica.
- No es una idea intuitiva de aplicar a problemas generales. Requiere mucho análisis.

Patrones relacionados: El modismo *Atomic Counters Array* está relacionado con los patrones de sincronización de hilos y/o procesos concurrentes y/o paralelos. Algunos patrones relacionados son el semáforo, la región crítica y el monitor.

4.4. Modismo *Atomic Node*

El modismo *Atomic Node* es una forma de manejar referencias a objetos sin sincronización a nivel de programación. Esto permite que dos o más componentes de *software* concurrentes y/o paralelos puedan llevar a cabo cambios de forma atómica en la referencias. Esto brinda la posibilidad de implementar ciertas estructuras de datos basadas en referencias a objetos en Java. En este caso, un *Node* es el elemento básico del que se componen las estructuras de datos [5].

Ejemplo: Considere un sistema distribuido en el cual diferentes nodos (estaciones) del sistema hacen uso de un diccionario de palabras que se encuentra ubicado en alguna de las estaciones. Este diccionario es una estructura de datos *trie*. Según [5] un *trie* es “un árbol para almacenar cadenas de caracteres en el que hay un nodo para cada prefijo común. Las cadenas son almacenadas en los nodos hojas”. En la figura 4.10 se puede apreciar esta estructura de datos.

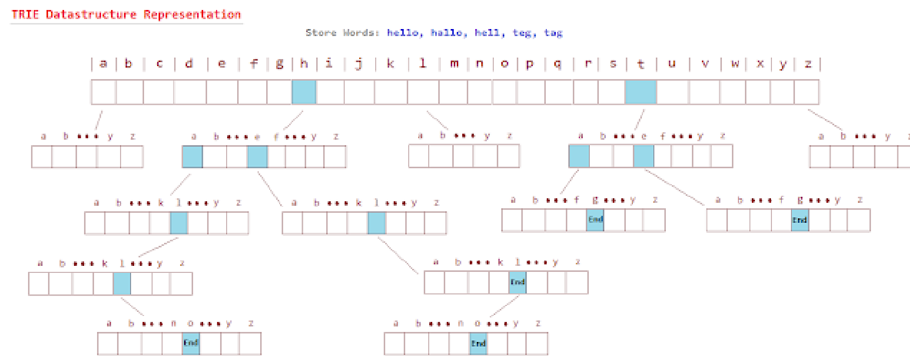

Figura 4.10: Árbol trie. Tomado de [25].

Contexto: Un programa paralelo o concurrente escrito en algún lenguaje de alto nivel con soporte para variables de tipo referencia o puntero con operaciones atómicas sobre ellas como C++ o Java en el cual se requieren implementar estructuras de datos basadas en referencias o punteros.

Problema: Preservar la integridad de las referencias a nodos de tal forma que la estructura de datos de la que forman parte no pierda la integridad en ningún momento aunque sea empleada concurrentemente por varios hilos de

ejecución.

Fuerzas: Para implementar una estructura de datos *lock-free* concurrente empleando *Atomic Node* en Java se deben tener en cuenta y balancear las siguientes fuerzas:

- Los nodos atómicos pueden ser creados e insertados concurrentemente por cualquier números de hilos de ejecución en paralelo.
- La estructura de datos resultante debe ser correcta, incluyendo la propiedad de *thread-safe*. Esto quiere decir que en un ambiente concurrente y/o paralelo, la solución debe comportarse como si estuviera protegida en sus operaciones con modismos de sincronización como semáforos, regiones críticas, monitores, etc.

Solución: Usar objetos *Atomic Node* basado en las referencias atómicas de Java para lograr el efecto de una referencia actualizada sin bloqueos a nivel de *software*. En el caso del lenguaje de programación Java tenemos las siguientes posibilidades [8]:

- Clase *AtomicReference<Node>*: Permite mantener una referencia a un objeto autoreferenciado *Node*. Este objeto permite realizar la operación *Compare and Swap*, la cual brinda la posibilidad de modificar una referencia compartida a un nodo de forma segura y libre de bloqueos.
- Clase *AtomicReferenceArray<Node>*: Permite mantener un arreglo de referencias atómicas a nodos, las cuales pueden ser modificadas de forma atómica mediante la operación de *hardware Compare and Swap*.
- Clase *AtomicMarkableReference<Node>*: Permite mantener una referencia a un objeto con un bit adicional en el que tanto la referencia como el bit se pueden modificar de forma atómica en una única operación a nivel de programación.
- Clase *AtomicStampedReference<Node>*: Permite mantener una referencia a un objeto con un campo adicional que representa una marca de tiempo. Tanto la referencia como la marca de tiempo se pueden modificar de forma atómica en una única operación a nivel de programación. Es empleada para evitar el problema ABA.

- Clase *AtomicReferenceFieldUpdater<T, Node>*: Permite actualizar un campo de tipo referencia a objeto nodo de un objeto T de forma atómica.

En todos los casos contamos con alguna variación de las siguientes operaciones para modificar y leer la variable contador atómica:

- Método `boolean compareAndSet(V expectedValue, V newValue)`: Compara la referencia atómica de tipo V con el argumento *expectedValue* y si son iguales se devuelve el valor actual de la variable atómica.

Estructura: La figura 4.11 muestra un nodo definido como *AtomicReference<Node>*. Se presenta como un tipo de dato abstracto que encapsula una referencia a otro nodo y una interfaz con las operaciones de *compareAndSet* y obtener el valor.

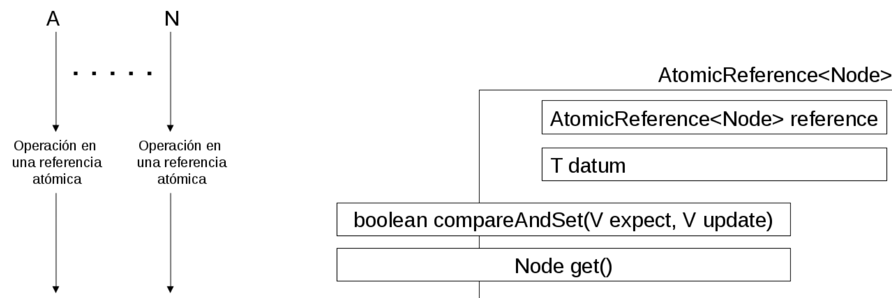

Figura 4.11: Un diagrama representando un *Atomic Node* como un tipo de dato abstracto.

A nivel de código el uso de las mismas es el siguiente:

```
import java.util.concurrent.atomic.AtomicInteger;

public class Node
{
    T datum;
    AtomicReference<Node> next;
}

public class AtomicNodeTester {
        static AtomicReference<Node> first = new AtomicReference<Node>();
}
```

```
public void concurrentUse() {
    Node current = first.get();
    /* Hacer algo */
    T something = current.datum;
    /* Hace algo */
}

public void concurrentInsertion(T datum) {
    AtomicReference<Node> current = first.get();
    /* Hacer algo*/
    Node n = new Node(datum);
    current.next.compareAndSwap(null, n);
    /* Hace algo */
}

public int concurrentDeletion(T datum) {
    AtomicReference<Node> current = first.get();
    /* Hacer algo*/
    Node temp = current.get().next;
    if (temp != null &&  temp.datum == datum)
        temp.next.compareAndSwap(temp, null);
    /* Hace algo */
}
```

Dinámica: El *Atomic Node* puede ser empleado en cualquier escenario donde se necesita crear estructuras de datos basadas en nodos. Por ejemplo, listas, colas, pilas, árboles, etc. La característica principal para que estas estructuras de datos puedan implementarse con *Atomic Node* es que las modificaciones sobre las mismas solo modifiquen un nodo de la estructura. O sea, que una modificación en la estructura de datos no cambie las referencias a los demás nodos. El diagrama de secuencia representado en la figura 4.12 muestra la interacción de dos componentes (A y B) con una referencia a un nodo atómico.

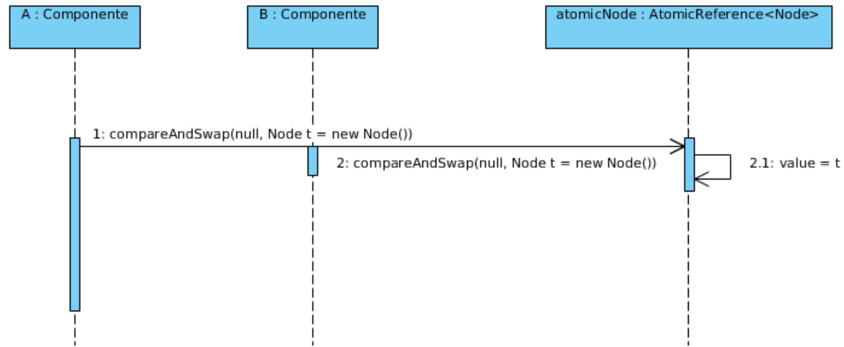

Figura 4.12: Diagrama de secuencia de la interacción de dos componentes de *software* con una referencia a un nodo atómico. Se aprecia que no hay sincronización a nivel de *software* en ningún momento. Ambos componentes A y B intentan modificar a atomicNode si es que este es null, solo una de las dos operaciones será efectiva. Esto se refleja en el resultado booleano que retorna la operación *compareAndSwap*

Ejemplo resuelto: A continuación se muestra la implementación de una clase *Trie* que hace uso de un *Atomic Node* el cual tiene dentro de si un arreglo de referencias atómicas a nodos de su mismo tipo y una variable booleana que indica si el nodo representa una palabra completa. O sea, es una palabra válida en el diccionario. Esta implementación está basada en [26].

```
public class TriesNode{
        public boolean isEnd = false;
        public AtomicReferenceArray<TriesNode> array = new
         ↪  AtomicReferenceArray<TriesNode>(26);
    }

    public class Trie
    {
        public TriesNode root;
        public Trie()
        {
            root = new TriesNode();
        }
        public static void insert(String word){
        TrieNode currentNode = root;
```

```
        for (int i = 0; i < word.length(); i++) {
            if (currentNode.series.get(word.charAt(i) - 97) == null)
                currentNode.series.compareAndSet(word.charAt(i)-97, null,
                 ↪  new TrieNode());

            if(word.length() - 1 == i)
            {
                currentNode.series.get(word.charAt(i) - 97).isAWord = true;
                break;
            }

            currentNode = currentNode.series.get(word.charAt(i)-97);
        }
    }

    private static boolean contains(String word){
        TrieNode currentNode = root;
        boolean isFound = true;
        for (int i = 0; i < word.length(); i++) {
            char character = word.charAt(i);

            if(start.series.get(character-97) != null) {
                if(word.length()-1 != i){
                    start = start.series.get(character-97);
                }
                else{
                    if(!start.series.get(character-97).isAWord) {
                        isFound = false;
                    }
                }
            }
            else {
                isFound = false;
                break;
            }
        }
        return isFound;
    }
}
```

Usos conocidos: Algunos problemas resueltos empleando *Atomic Node* son:

- Estructuras de datos secuenciales como colas, pilas, listas [21].
- Estructuras de datos arbóreas como árboles de búsqueda, árboles binarios de búsquedas, árboles hash, tries, etc [28].

Consecuencias:

Ventajas:

- Mejora los tiempos de ejecución en casos de alta concurrencia con independencia de la cantidad de hilos de ejecución.
- Es escalable en cuanto a que su funcionamiento es correcto independientemente de la cantidad de hilos que lo emplean.
- Se elimina el concepto de sección crítica para la manipulación del contador atómico. Es decir no se requiere sincronización a nivel de *software* para modificar un *Atomic Node*.

Desventajas:

- Solo resuelve cuando la modificación es puntual, o sea, no funciona en estructuras de datos que requieran reasignaciones de referencias.

Patrones relacionados: El modismo *Atomic Node* está relacionado con los modismos *Atomic Counter* y *Atomic Counters Array*.

4.5. Resumen

En el presente capítulo se presentan cuatro modismos basados en tipos de datos con operaciones atómicas de la biblioteca *Atomic* de Java. Los modismos se presentan usando la forma POSA de [7].
En primer lugar se presenta el modismo *Atomic Flag*, el cual, basado en un tipo de dato atómico, permite la modificación concurrente y segura de una variable booleana. Este se puede usar como base en la implementación de un *spinlock*. También se emplea para representar estados del programa en el que se use.
Luego se introduce el modismo *Atomic Counter*, que se basa en un valor entero modificado mediante operaciones de incremento o decremento atómicas. Este modismo puede ser usado en estadísticas de tiempo de ejecución de un programa, en la implementación de semáforos generales, etc.
A continuación se desglosa el modismo *Atomic Counters Array*, el cual generaliza la idea de un *Atomic Counter* a un arreglo de *Atomic Counter*, permitiendo realizar operaciones atómicas sobre las posiciones del arreglo. Este modismo puede ser empleado por ejemplo, en la implementación de un algoritmo de ordenamiento *Counting Sort* concurrente.
Finalmente se presenta el modismo *Atomic Node* el cual describe el concepto

de nodo de una estructura de datos. Pero en este caso puede ser modificada de forma atómica, permitiendo con esto la implementación de varias estructuras de datos libres de bloqueos.

Capítulo 5

Experimentos y resultados

En el presente capítulo se muestran los diferentes experimentos diseñados y desarrollados, así como sus resultados, para comprobar el correcto funcionamiento y desempeño de las ideas planteadas en el Capítulo 4. Se describe de forma específica el escenario real, tanto de *hardware* como de *software* donde son llevados a cabo. El capítulo está desglosado en experimentos para cada uno de los modismos, cada uno con su análisis de resultados propio.

5.1. Escenario de *hardware* y *software*

Los experimentos descritos en el presente capítulo fueron realizados en un ambiente de *hardware* conformado por los siguientes recursos:

- **Procesador:** AMD A6-3400M APU with Radeon(tm) HD Graphics × 4 1.4 GHz.
- **RAM:** 7.3 GiB.

Por su parte el entorno de *software* está conformado por:

- **Sistema operativo:** Debian GNU/Linux 9.4 (stretch).
- **Compilador y Máquina Virtual de Java:** Java(TM) SE Runtime Environment build (1.8.0_172-b11) de Oracle.
- **IDE:** Eclipse IDE for Java Developers Version: Oxygen.3a Release (4.7.3a) Build id: 20180405-1200

- **time**: Comando utilitario del interprete de comandos de los sitemas operativos GNU/Linux para medir el tiempo de ejecución de un programa.

5.2. Experimentación

Los experimentos están diseñados para comparar en cuanto a tiempo de ejecución (*wall-clock time*) un ejemplo de uso del modismo con un equivalente bloqueante. Solo se realiza un experimento por cada modismo, de tal forma que sea representativo del uso general de los mismos.
No es objetivo de los experimentos hacer mediciones comparativas con versiones no concurrentes de los programas que se prueban y por ello no se mide *speedup*. Los programas de prueba se mantienen lo más simple posible en tanto que el objetivo es hacer énfasis en la contención en el acceso concurrente a los objetos involucrados en los modismos. Teniendo en cuenta la idea anterior, para probar la contención, o sea, la competencia por modificar los datos compartidos, la variabilidad está únicamente dada por la cantidad de hilos. Por tanto, el nivel de contención en estos experimentos es proporcional a la cantidad de hilos.
Estos experimentos buscan responder dos preguntas fundamentales:

- ¿Se obtienen mejores tiempos de ejecución mediante el empleo de los modismos presentados?
- ¿Qué impacto sufre el desempeño en el uso de los modismos a medida que se aumenta la cantidad de hilos?

Se toman los tiempos de ejecución de diez ejecuciones por cada programa de prueba en la experimentación. Se calcula un promedio de los tiempos de ejecución de cada uno de los programas y se calcula un intervalo de confianza de *T-Student*.
A partir de estos datos se realizan las comparaciones pertinentes.

5.2.1. Experimentos con *Atomic Flag*

La prueba consiste en incrementar un entero volátil desde varios hilos de ejecución. Cada hilo incrementa el contador en 10000000 unidades. Se comparan dos programas. Uno que usa el modismo *Atomic Flag* y otro que es una versión del mismo programa pero que emplea mecanismos de sincronización bloqueantes como *ReentrantLock* de Java.
A continuación se muestra el código de la versión bloqueante.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
class Incrementer implements Runnable {
    public void run() {
        for (int i = 0; i < BlockingExperiment.numOfItems; ++i) {
            BlockingExperiment.rlock.lock();
            ++BlockingExperiment.variable;
            BlockingExperiment.rlock.unlock();
        }
    }
}
public class BlockingExperiment {
    static Lock rlock = new ReentrantLock();
    static volatile int variable = 0;
    static int numOfItems = 10000000;
    static int numOfThreads = 4; // Parámetro variable entre las pruebas
    static Thread[] incrementers = new Thread[numOfThreads];

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < numOfThreads; ++i) {
            incrementers[i] = new Thread(new Incrementer());
        }
        for (int i = 0; i < numOfThreads; ++i) {
            incrementers[i].start();
        }
        for (int i = 0; i < numOfThreads; ++i) {
            incrementers[i].join();
        }
        System.out.println("Valor final de la variable = " + variable);
        System.exit(0);
    }
}
```

A continuación el código fuente de la versión empleando el modismo *Atomic Flag*:

```
import java.util.concurrent.atomic.AtomicBoolean;
class Incrementer implements Runnable {
    public void run() {
        for (int i = 0; i < BlockingExperiment.numOfItems; ++i) {
            while (!BlockingExperiment.mutex.compareAndSet(false, true))
            try {
                Thread.sleep(0, 1);
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
            }
            ++BlockingExperiment.variable;
            BlockingExperiment.mutex.set(false);
        }
    }
}
public class BlockingExperiment {
    static AtomicBoolean mutex = new AtomicBoolean();
    static volatile int variable = 0;
    static int numOfItems = 10000000;
    static int numOfThreads = 2;
    static Thread[] incrementers = new Thread[numOfThreads];
    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < numOfThreads; ++i) {
            incrementers[i] = new Thread(new Incrementer());
        }
        for (int i = 0; i < numOfThreads; ++i) {
            incrementers[i].start();
        }
        for (int i = 0; i < numOfThreads; ++i) {
            incrementers[i].join();
        }
        System.out.println("Valor final de la variable = " + variable);
        System.exit(0);
    }
}
```

Los resultados obtenidos se muestran en el cuadro 5.1:

Tiempo (s)		
Cantidad de hilos	*ReentrantLock*	*Atomic Flag*
2	$2{,}93 \pm 0{,}15$	$1{,}25 \pm 0{,}03$
4	$4{,}67 \pm 0{,}17$	$2{,}68 \pm 0{,}02$
8	$9{,}01 \pm 0{,}10$	$5{,}25 \pm 0{,}02$
16	$17{,}87 \pm 0{,}26$	$10{,}30 \pm 0{,}04$

Cuadro 5.1: Resultados experimentales del uso de *ReentrantLock* y del modismo *Atomic Flag* para proteger una variable entera.

Análisis de resultados

En la figura 5.1 se muestra la gráfica comparativa entre ambos programas:

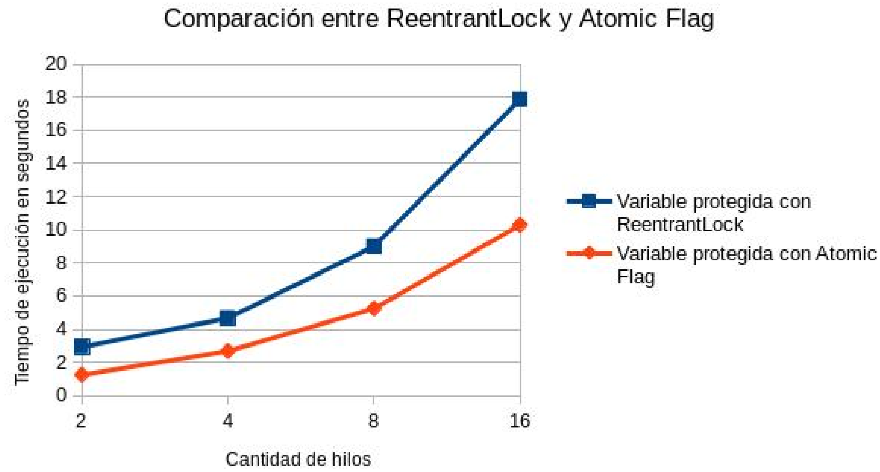

Figura 5.1: Gráfica comparativa entre los promedios del tiempo de ejecución del programa que usa protección con *Reentrant Lock* y el que emplea el modismo *Atomic Flag*.

Como se puede apreciar, tanto en la tabla 5.1 como en la gráfica 5.1, la versión con *Atomic Flag* es casi un 50 % más rápido en todos los casos. Ambos programas como es esperado degradan su rendimiento a medida que hay mayor cantidad de hilos y por tanto de contención.

5.2.2. Experimentos con *Atomic Counter*

La prueba consiste en incrementar un entero volátil protegido con el *Atomic Flag* desde varios hilos de ejecución, en el caso del *Atomic Counter* este se incrementa igualmente desde varios hilos de ejecución pero sin sincronización a nivel de programación. Cada hilo incrementa el contador en 10000000 unidades. Se comparan dos programas. Uno que usa el modismo *Atomic Counter* y el programa de la sección anterior con mejor rendimiento. Este es el *spinlock* implementado con *Atomic Flag* y por tanto el código fuente es el mismo de la sección anterior.
A continuación se muestra el código de la versión con *Atomic Counter*.

```java
import java.util.concurrent.atomic.AtomicInteger;
class Incrementer implements Runnable {
    public void run() {
        for (int i = 0; i < BlockingExperiment.numOfItems; ++i) {
            BlockingExperiment.variable.incrementAndGet();
        }
    }
}
public class BlockingExperiment {
    static AtomicInteger variable = new AtomicInteger();
    static int numOfItems = 10000000;
    static int numOfThreads = 2;
    static Thread[] incrementers = new Thread[numOfThreads];
    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < numOfThreads; ++i) {
            incrementers[i] = new Thread(new Incrementer());
        }
        for (int i = 0; i < numOfThreads; ++i) {
            incrementers[i].start();
        }
        for (int i = 0; i < numOfThreads; ++i) {
            incrementers[i].join();
        }
        System.out.println("Valor final de la variable = " + variable);
        System.exit(0);
    }
}
```

Los resultados obtenidos se muestran en el cuadro 5.2:

Tiempo (s)		
Cantidad de hilos	*Atomic Flag*	*Atomic Counter*
2	$1{,}25 \pm 0{,}03$	$0{,}63 \pm 0{,}02$
4	$2{,}68 \pm 0{,}02$	$1{,}40 \pm 0{,}04$
8	$5{,}25 \pm 0{,}02$	$2{,}76 \pm 0{,}09$
16	$10{,}30 \pm 0{,}04$	$5{,}25 \pm 0{,}11$

Cuadro 5.2: Resultados experimentales del uso de *Atomic Flag* para proteger una variable entera y del modismo *Atomic Counter* como un contador entero modificado con operaciones atómicas.

Análisis de resultados

En la figura 5.2 se muestra la gráfica comparativa entre ambos programas:

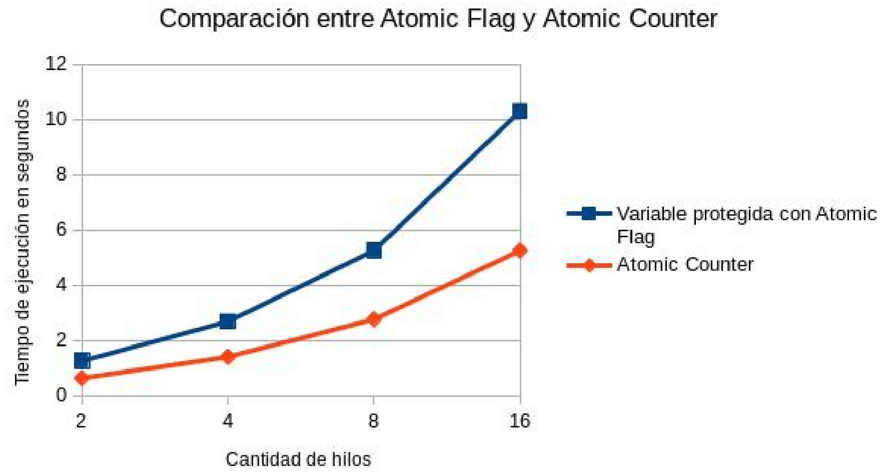

Figura 5.2: Gráfica comparativa entre los promedios del tiempo de ejecución del programa que usa protección con *Atomic Flag* y el que emplea el modismo *Atomic Counter*.

Como se puede apreciar, tanto en la tabla 5.2 como en la gráfica 5.2, la versión sin bloqueos implementada con *Atomic Counter* es casi un 50 % más rápido en todos los casos. Ambos programas como es esperado degradan su rendimiento a medida que hay mayor cantidad de hilos y por tanto de contención.

5.2.3. Experimentos con *Atomic Counters Array*

El experimento consiste en probar el ejemplo que se da para este modismo en la sección 4.3. Este consiste en simular el conteo en un algoritmo de ordenamiento *Counting Sort*. El arreglo a ordenar tiene tamaño de 100000000. Se comparan dos programas. Uno que usa el modismo *Atomic Counters Array* y otro que es una versión del mismo programa pero implementada con un *ArrayList* de Java sincronizado con monitores.
La versión bloqueante es implementada con un objeto arreglo de tipo *ArrayList* que se sincroniza en un bloque *synchronized* usando el monitor del propio objeto arreglo. La versión empleando el modismo *Atomic Counters Array* usa un arreglo de enteros atómicos que se modifica desde varios hilos de ejecución pero sin sincronización a nivel de programación.

El código de la clase principal desde donde se instancia la clase que implementa el algoritmo de ordenamiento:

```
public class CountingSortTester {
    static public void main(String []args) throws InterruptedException
    {
        int MAX = 100000000;
        ConcurrentCountingSort ccs = new ConcurrentCountingSort(1, MAX, 16);
        ccs.sort();
        System.exit(0);
    }
}
```

A continuación el código de la clase *ConcurrentCountingSort* implementada con bloqueos:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.concurrent.ThreadLocalRandom;
public class ConcurrentCountingSort {
    private ArrayList<Integer> countersArray = null;
    public int NUMOFSORTERS = Runtime.getRuntime().availableProcessors();
    Thread[] sorters;
    private int minValue;
    private int maxValue;
    ConcurrentCountingSort(int min, int max, int threads)
    {
        countersArray = new ArrayList<Integer>(Collections.nCopies(max - min
         ↪   + 1, 0));
        minValue = min;
        maxValue = max;
        NUMOFSORTERS = threads;
        sorters = new Thread[NUMOFSORTERS];
    }
    private class SubArraySorter implements Runnable
    {
        private int begin;
        private int end;
        SubArraySorter(int b, int e)
        {
            begin = b;
            end = e;
        }
        public void run() {
            for (int i = begin; i <= end; ++i) {
```

```
                synchronized (countersArray) {
                    int pos = 1 +
                     ↪  ThreadLocalRandom.current().nextInt(maxValue) -
                     ↪  minValue;
                    countersArray.set(pos, countersArray.get(pos) + 1);
                }
            }
        }
    }
    public void sort() throws InterruptedException
    {
        int elementsPerSorter = maxValue / NUMOFSORTERS;
        int begin = 0;
        int end = elementsPerSorter - 1;
        for (int i = 0; i < NUMOFSORTERS; ++i) {
            sorters[i] = new Thread(new SubArraySorter(begin, end));
            sorters[i].start();
            begin = end + 1;
            end += elementsPerSorter;
        }
        for (int i = 0; i < NUMOFSORTERS; ++i) {
            sorters[i].join();
        }
    }
}
```

A continuación el código de la clase *ConcurrentCountingSort* implementada usando el modismo *Atomic Counters Array*:

```
import java.util.concurrent.atomic.AtomicIntegerArray;
import java.util.concurrent.ThreadLocalRandom;
public class ConcurrentCountingSort {
    private AtomicIntegerArray atomicCountersArray = null;
    public int NUMOFSORTERS = Runtime.getRuntime().availableProcessors();
    Thread[] sorters;
    private int minValue;
    private int maxValue;
    ConcurrentCountingSort(int min, int max, int threads)
    {
        atomicCountersArray = new AtomicIntegerArray(max - min + 1);
        minValue = min;
        maxValue = max;
        NUMOFSORTERS = threads;
        sorters = new Thread[NUMOFSORTERS];
    }
```

```
    private class SubArraySorter implements Runnable
    {
        private int begin;
        private int end;
        SubArraySorter(int b, int e)
        {
            begin = b;
            end = e;
        }
        public void run() {
            for (int i = begin; i <= end; ++i) {
                atomicCountersArray.incrementAndGet(1 +
                 ↪  ThreadLocalRandom.current().nextInt(maxValue) -
                 ↪  minValue);
            }
        }
    }
    public void sort() throws InterruptedException
    {
        int elementsPerSorter = maxValue / NUMOFSORTERS;
        int begin = 0;
        int end = elementsPerSorter - 1;
        for (int i = 0; i < NUMOFSORTERS; ++i) {
            sorters[i] = new Thread(new SubArraySorter(begin, end));
            sorters[i].start();
            begin = end + 1;
            end += elementsPerSorter;
        }
        for (int i = 0; i < NUMOFSORTERS; ++i) {
            sorters[i].join();
        }
    }
}
```

Los resultados obtenidos se muestran en el cuadro 5.3:

Tiempo (s)		
Cantidad de hilos	*Synchronized ArrayList*	*Atomic Counters Array*
2	$73{,}16 \pm 0{,}74$	$11{,}42 \pm 0{,}09$
4	$84{,}67 \pm 0{,}54$	$6{,}94 \pm 0{,}13$
8	$87{,}09 \pm 0{,}69$	$6{,}92 \pm 0{,}07$
16	$86{,}73 \pm 0{,}67$	$6{,}92 \pm 0{,}04$

Cuadro 5.3: Resultados experimentales en cuanto a tiempo de ejecución del uso de un arreglo de tipo *ArrayList* sincronizado con monitores de Java para contar ocurrencias de elementos en un arreglo a ordenar y del modismo *Atomic Counters Array* con la misma función pero sin sincronización a nivel de programación.

Análisis de resultados

En la figura 5.3 se muestra la gráfica comparativa entre ambos programas:

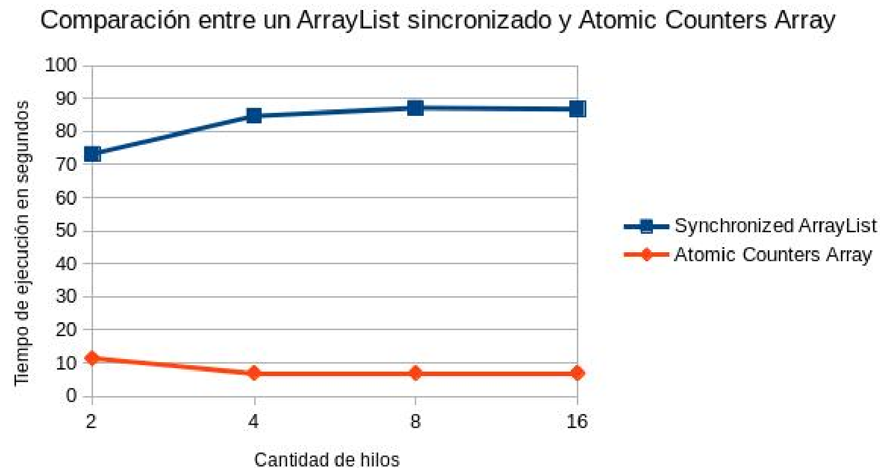

Figura 5.3: Gráfica comparativa entre los promedios del tiempo de ejecución del programa que usa un arreglo de tipo *ArrayList* y el que emplea el modismo *Atomic Counters Array*.

Como se puede apreciar, tanto en la tabla 5.3 como en la gráfica 5.3, la

versión con *Atomic Counters Array* es en más de un 90 % más rápido en todos los casos. En este experimento ninguno de los dos programas bajo prueba degrada su rendimiento a medida que hay mayor cantidad de hilos. Esto ocurre debido a la aleatoriedad de los valores a ordenar, que en el caso de *Atomic Counters Array*, a diferencia de experimentos anteriores con otros modismos, la contención disminuye al ser repartida entre todas las posiciones del arreglo atómico. Es de esperar que para mayores cantidades de hilos de ejecución, o para arreglos con valores menos uniformemente distribuidos de forma aleatoria, en el ambiente de *hardware* que se prueba, se observe una degradación del rendimiento.

5.2.4. Experimentos con *Atomic Node*

El experimento consiste en probar el ejemplo que se da para este modismo en la sección 4.4 en el que se implementa un árbol de diccionario *trie*. Cada hilo inserta en el *trie* unas 23132 palabras. Cuando el experimento se realiza con 16 hilos de ejecución se insertan en total 370099 palabras entre todos los hilos. Se comparan dos programas. Uno que usa el modismo *Atomic Node* y otro que es una versión del mismo programa pero implementada con arreglos planos de Java y sincronizado con monitores.
La versión bloqueante es implementada con un arreglo plano de referencias que se sincroniza en el método insertar usando el monitor de la clase. La versión empleando el modismo *Atomic Node* por su parte emplea un arreglo de referencias atómicas (*Atomic Reference*) a *AtomicNodes* los cuales se modifican desde varios hilos de ejecución pero sin sincronización a nivel de programación.
A continuación el código de la versión bloqueante:

```
//File TrieNode.java
class TrieNode{
    public boolean isAWord;
    public TrieNode[] series = new TrieNode[26];
}
//File Trie.java
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
class TrieInserterThread implements Runnable
{
```

```
    private int id;
    BufferedReader reader;
    TrieNode start;
    TrieInserterThread(int _id)
    {
        id = _id;
        Path path = FileSystems.getDefault().getPath("wordlist" + id +
         ↪  ".txt");
        try {
            reader = Files.newBufferedReader(path);
        } catch (IOException x) {
            System.err.format("IOException: %s%n", x);
        }
    }
    public void run() {
        String line = null;
        try {
            while ((line = reader.readLine()) != null) {
                Trie.insert(Trie.start, line.toLowerCase());
            }
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
public class Trie {
    static TrieNode start = new TrieNode();
    final static int NUMOFTHREADS = 16;
    private static int wordsCounter;
    static Thread []trieInserters = new Thread[NUMOFTHREADS];
    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < NUMOFTHREADS; ++i) {
            trieInserters[i] = new Thread(new TrieInserterThread(i + 1));
        }
        for (int i = 0; i < NUMOFTHREADS; ++i) {
            trieInserters[i].start();
        }
        for (int i = 0; i < NUMOFTHREADS; ++i) {
            trieInserters[i].join();
        }

        //printDictionary(start,"");
        System.out.println(size(start));
    }
```

```
public static synchronized void insert(TrieNode currentNode, String
 ↪  word){
    for (int i = 0; i < word.length(); i++) {
        if (currentNode.series[word.charAt(i) - 97] != null) {
            if(word.length()-1 == i){
                currentNode.series[word.charAt(i) - 97].isAWord = true;
            }
            currentNode = currentNode.series[word.charAt(i)-97];
        }
        else{
            TrieNode trie = new TrieNode();
            trie.isAWord = (word.length() - 1 == i ? true : false);
            currentNode.series[word.charAt(i)-97] = trie;
            currentNode = currentNode.series[word.charAt(i)-97];
        }
    }
}
private static boolean contains(TrieNode start, String word){
    boolean isFound = true;
    for (int i = 0; i < word.length(); i++) {
        char character = word.charAt(i);

        if(start.series[character-97]!=null) {
            if(word.length()-1 != i){
                start = start.series[character-97];
            }
            else{
                if(!start.series[character-97].isAWord) {
                    isFound = false;
                }
            }
        }
        else {
            isFound = false;
            break;
        }
    }
    return isFound;
}
private static void printDictionary(TrieNode start, String toPrint) {
    if(start.isAWord) {
        System.out.println(toPrint);
    }
    for (int i = 0; i < start.series.length; i++) {
        if (start.series[i] != null)
```

```
                printDictionary(start.series[i], toPrint + (char)(97+i));
        }
    }
    private static int size(TrieNode start) {
        wordsCounter = 0;
        countWords(start);
        return wordsCounter;
    }
    private static void countWords(TrieNode start) {
        if(start.isAWord) {
            ++wordsCounter;
        }
        for (int i = 0; i < start.series.length; i++) {
            if (start.series[i] != null)
                countWords(start.series[i]);
        }
    }
}
```

A continuación el código de la versión empleando el modismo *Atomic Node*:

```
// File AtomicNode.java
class AtomicNode{
    public boolean isAWord;
    public AtomicReferenceArray<AtomicNode> series = new
     ↪  AtomicReferenceArray<AtomicNode>(26);
}
// File Trie.java
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.concurrent.atomic.AtomicInteger;
class TrieInserterThread implements Runnable
{
    private int id;
    BufferedReader reader;

    TrieInserterThread(int _id)
    {
        id = _id;
        Path path = FileSystems.getDefault().getPath("wordlist" + id +
         ↪  ".txt");
        try {
```

```
            reader = Files.newBufferedReader(path);
        } catch (IOException x) {
            System.err.format("IOException: %s%n", x);
        }
    }
    public void run() {
        String line = null;
        try {
            while ((line = reader.readLine()) != null) {
                Trie.insert(Trie.start, line.toLowerCase());
            }
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
public class Trie {
    static AtomicNode start;
    static AtomicInteger nodesIndex = new AtomicInteger();
    final static int NUMOFTHREADS = 16;
    private static int wordsCounter;
    static Thread []trieInserters = new Thread[NUMOFTHREADS];
    public static void main(String[] args) throws InterruptedException {
        start = new AtomicNode();
        for (int i = 0; i < NUMOFTHREADS; ++i) {
            trieInserters[i] = new Thread(new TrieInserterThread(i + 1));
        }
        for (int i = 0; i < NUMOFTHREADS; ++i) {
            trieInserters[i].start();
        }
        for (int i = 0; i < NUMOFTHREADS; ++i) {
            trieInserters[i].join();
        }

        System.out.println(size(start));
    }
    public static void insert(AtomicNode currentNode, String word){
        for (int i = 0; i < word.length(); i++) {
            if (currentNode.series.get(word.charAt(i) - 97) == null)
                currentNode.series.compareAndSet(word.charAt(i)-97, null,
                 ↪   new AtomicNode());

            if(word.length() - 1 == i)
            {
```

```
                currentNode.series.get(word.charAt(i) - 97).isAWord = true;
                break;
            }

            currentNode = currentNode.series.get(word.charAt(i)-97);
        }
    }
    private static boolean contains(AtomicNode start, String word){
        boolean isFound = true;
        for (int i = 0; i < word.length(); i++) {
            char character = word.charAt(i);

            if(start.series.get(character-97) != null) {
                if(word.length()-1 != i){
                    start = start.series.get(character-97);
                }
                else{
                    if(!start.series.get(character-97).isAWord) {
                        isFound = false;
                    }
                }
            }
            else {
                isFound = false;
                break;
            }
        }
        return isFound;
    }
    private static void printDictionary(AtomicNode start, String toPrint) {
        if(start.isAWord) {
            System.out.println(toPrint);
        }
        for (int i = 0; i < start.series.length(); i++) {
            if (start.series.get(i) != null)
                printDictionary(start.series.get(i), toPrint +
                 ↪  (char)(97+i));
        }
    }
    private static int size(AtomicNode start) {
        wordsCounter = 0;
        countWords(start);
        return wordsCounter;
    }
    private static void countWords(AtomicNode start) {
```

```
        if(start.isAWord) {
            ++wordsCounter;
        }
        for (int i = 0; i < start.series.length(); i++) {
            if (start.series.get(i) != null)
                countWords(start.series.get(i));
        }
    }
}
```

Los resultados obtenidos se muestran en el cuadro 5.4:

Tiempo (s)		
Cantidad de hilos	*Trie sincronizado*	*Atomic Node Trie*
2	$0{,}36 \pm 0{,}01$	$0{,}36 \pm 0{,}01$
4	$0{,}43 \pm 0{,}01$	$0{,}48 \pm 0{,}02$
8	$0{,}67 \pm 0{,}03$	$0{,}71 \pm 0{,}04$
16	$1{,}67 \pm 0{,}17$	$2{,}84 \pm 0{,}14$

Cuadro 5.4: Resultados experimentales del tiempo de ejecución de un árbol *Trie* sincronizado con monitores de Java y de un árbol *Trie* basado en el modismo *Atomic Node* con la misma función pero sin sincronización a nivel de programación.

Análisis de resultados

En la figura 5.4 se muestra la gráfica comparativa entre ambos programas:

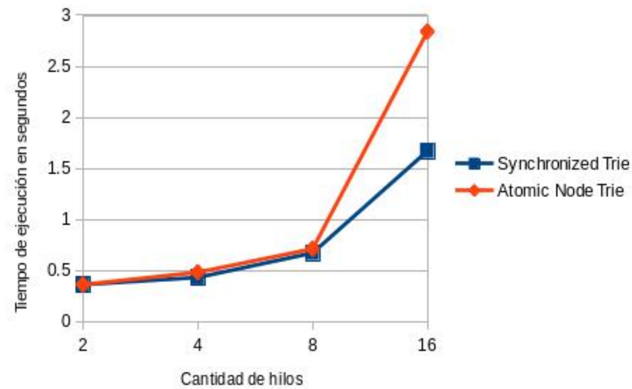

Figura 5.4: Gráfica comparativa entre los promedios del tiempo de ejecución del programa que usa un árbol *Trie* sincronizado y el que emplea un árbol *Trie* basado en el modismo *Atomic Node*.

Como se puede apreciar, en este caso la versión del programa que hace uso del modismo *Atomic Node* para implementar el árbol *Trie* no es mejor que la versión sincronizada. Esto se debe a la creación de los objetos *Node*, ya que crear un *AtomicReferenceArray* es mucho más costoso en tiempo que crear un arreglo plano de objetos *TrieNode*.
Resulta curioso que solo para la versión basada en *Atomic Node* con 16 hilos de ejecución obtenemos una mejora significativa si eliminamos la condición `if (currentNode.series[word.charAt(i) - 97] != null)`, quedando el método de inserción de la siguiente manera:

```
public static void insert(AtomicNode currentNode, String word){
    for (int i = 0; i < word.length(); i++) {
         currentNode.series.compareAndSet(word.charAt(i) - 97, null, new
          ↪   AtomicNode());

        if(word.length() - 1 == i)
            currentNode.series.get(word.charAt(i) - 97).isAWord = true;
        else
            currentNode = currentNode.series.get(word.charAt(i) - 97);
    }
}
```

Con este cambio logra un tiempo de 2.02 ± 0.08 segundos que es casi un 30 %

más rápido que la versión que prueba la condición. Esto como se menciona solo ocurre con el experimento de 16 hilos.

5.3. Resumen

En este capítulo se presentan los experimentos diseñados y llevados a cabo para comprobar la hipótesis de la tesis. Para cada modismo del Capítulo 4 se realiza un experimento que consiste en un programa empleando el modismo y un programa equivalente pero que emplea mecanismos de sincronización bloqueantes como los *locks*. Ambos se ejecutan en diez ocasiones, se les calcula el promedio con intervalo de confianza de *T-Student* y luego estos resultados se analizan.

Capítulo 6

Conclusiones

En el presente capítulo se retoma la hipótesis del Capítulo 1 para ser evaluada de tal manera que permita llegar a las conclusiones de la tesis. Para esto se toman en cuenta los experimentos comparativos realizados en el Capítulo 5.

6.1. Evaluación de la hipótesis

Considérese la hipótesis presentada en el capítulo introductorio:
Es posible extraer patrones de bajo nivel o modismos a partir del uso de los tipos de datos atómicos de Java, de tal manera que permitan el diseño e implementación de ciertos algoritmos y estructuras de datos de uso común, que sean más eficientes prescindiendo de las primitivas bloqueantes de sincronización.

6.1.1. Discusión

Con la presentación de los modismos *Atomic Flag*, *Atomic Counter*, *Atomic Counters Array* y *Atomic Node* se obtienen métodos no bloqueantes para modificar de forma concurrente variables simples. A partir de las pruebas comparativas realizadas se obtiene que, en efecto, el empleo de los tipos atómicos de Java, en forma de modismos, trae ventajas de rendimiento en casos similares a los experimentados en el Capítulo 5.
Se desprende de esto que los lenguajes de programación que también incorporan estos conceptos pueden beneficiarse también de la aplicación de estos patrones. Por ejemplo, el lenguaje de programación C++ desde su especificación del 2011

incluye una biblioteca de atómicos con funcionalidades similares a las presentadas en este trabajo.

6.1.2. Análisis e interpretación de los resultados

El análisis de los resultados de la experimentación realizada en el Capítulo 5 muestra que en efecto la hipótesis es válida. Sí es posible identificar patrones de bajo nivel o modismos a partir del uso de la biblioteca de atómicos de Java y mediante su empleo obtener tiempos de ejecución menores que sus contrapartes bloqueantes.

6.2. Consideraciones

Se describen las ventajas y desventajas de la propuesta del presente trabajo como consideraciones:

6.2.1. Ventajas

- **Mejora en el rendimiento:** En general se aprecian mejoras en cuanto a tiempo de ejecución al emplear estos modismos. Existen excepciones.
- **Documentación:** Se realiza una descripción clara y concisa siguiendo el formato POSA [7] que permite ser consultada de forma fácil y oportuna por otros desarrolladores.
- **Ejemplificación:** Por su descripción en forma de patrón se puede portar a otros lenguajes como C++, que tiene utilidades similares para la manipulación atómica de variables.

6.2.2. Desventajas

- **Contexto limitado:** El contexto de empleo de estos modismos limitado. Es por ello que se deja al usuario de este conocimiento la evaluación y prueba en la solución de su problema específico.
- **Dificultad de aprendizaje:** No son ideas intuitivas, de manera que, al igual que para aprender a programar en paralelo requiere una nueva manera de pensar sobre la programación secuencial tradicional, la programación libre de bloqueos requiere una nueva manera de pensar sobre la programación paralela tradicional.

- **No *starvation-free*:** Su uso no garantiza la propiedad del programa de ser *starvation-free* cuando para continuar la ejecución de un hilo determinado hay que hacer espera de un resultado específico al modificar atómicamente una variable. Esto es una limitación propia de la programación libre de bloqueos.

6.3. Comparación del trabajo relacionado

Se han escrito numerosos trabajos sobre *spinlocks* en sus diversas implementaciones. Estos trabajos ahondan en las diferentes implementaciones buscando minimizar el uso del bus de datos o el impacto de mantener la coherencia de la memoria caché a la hora de hacer la espera activa. Se basan en un estudio a nivel de arquitectura de computadoras. La novedad del presente trabajo es aprovechar la evolución en las computadoras desde aquellos primeros artículos, así como las mejoras en los lenguajes de programación (Java en este caso) para identificar patrones de uso relacionados con mecanismos de sincronización y estructuras de datos libres de bloqueos que se presentan en los trabajos relacionados.
A partir de los trabajos relacionados del Capítulo 3 y otros, donde presentan estructuras de datos, como colas libres de bloqueos, basadas en la modificación atómica, se extraen los patrones que se presentan, en particular *Atomic Node*. De tal manera, el presente trabajo se distingue porque brinda mejor entendimiento en su uso a los tipos de datos con operaciones atómicas sobre ellos al ser descritos en sus posibles usos como patrones. Si bien los trabajos relacionados proponen implementaciones de estructuras de datos libres de bloqueo, estas están presentadas en pseudocódigo. El presente trabajo abre la posibilidad de implementar estructuras de datos, en lenguajes de programación modernos y de alto nivel como Java y C++, usando estos modismos. Por último, se considera como distinción del presente trabajo, la introducción de estas herramientas al mundo de los patrones de *software*.

6.4. Resumen de las contribuciones

La contribución principal del presente trabajo es mostrar que es posible identificar patrones de uso de las clases de la biblioteca de atómicos de Java (*java.util.concurrent.atomic*) como mecanismos de sincronización no bloquean-

te. A su vez, demuestra la pertinencia de usarlo en ciertos casos para obtener mejores rendimientos, así como para evitar los problemas asociados a la programación bloqueante.
Este resultado no se enmarca solo al lenguaje de programación Java, sino que son modismos que pueden ser portados a cualquier lenguaje que ofrezca funcionalidades similares. Por ejemplo, el lenguaje C++ incluye desde su versión del 2011 una biblioteca para el trabajo con tipos de datos manipulables de forma atómica.
Otras contribuciones son:

- Demostración práctica de la viabilidad tanto funcional, como de desempeño del uso de la programación libre de bloqueos en la implementación de patrones de diseño para la programación paralela.

- Comparación de desempeño entre implementaciones bloqueantes y libre de bloqueos de patrones de diseño.

- Descripción en forma de patrón POSA de los usos de algunas de las clases de atómicos de JAVA. Dejando de esta forma una documentación organizada sobre los mismos.

6.5. Trabajo futuro

Se recomienda abordar como trabajos futuros los siguientes temas:

- Diseñar e implementar estructuras de datos libres de bloqueos empleando los modismos presentados: basados en los ejemplos del presente trabajo, es posible desarrollar una serie de estructuras de datos concurrentes y libres de bloqueos. Por ejemplo, estructuras de datos secuenciales como vectores, colas doblemente terminadas, listas enlazadas, etc.

- Portar a otros lenguajes los modismos descritos en el presente trabajo. Por ejemplo, C++ cuenta con una biblioteca *atomic* en el que se encuentran clases y operaciones similares a las presentadas en Java.

Bibliografía

[1] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[2] Gregory R Andrews. *Concurrent programming: principles and practice*. Benjamin/Cummings Publishing Company, 1991.

[3] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2015.

[4] Blaise Barney. Introduction to parallel computing. `https://computing.llnl.gov/tutorials/parallel_comp/`, 2018.

[5] Paul E. Black. ”dictionary of algorithms and data structures”. `https://xlinux.nist.gov/dads/`, 2018.

[6] Clay Breshears. *The art of concurrency: A thread monkey's guide to writing parallel applications*. O'Reilly Media, Inc., 2009.

[7] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

[8] Oracle Corporation. Documentación Java JDK 9. `https://docs.oracle.com/javase/9/index.html`, 2018.

[9] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, pages 185–192. IEEE, 2010.

[10] Edsger W Dijkstra. Cooperating sequential processes. In *The origin of concurrent programming*, pages 65–138. Springer, 1968.

[11] Allen Downey. *The little book of semaphores*. Allen B. Downey, 2016.

[12] Real Academia Española. Diccionario de la lengua española. `http://dle.rae.es`, Diciembre 2017.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[14] Fayez Gebali. *Algorithms and parallel computing*, volume 84. John Wiley & Sons, 2011.

[15] Per Brinch Hansen. *The origin of concurrent programming: from semaphores to remote procedure calls*. Springer Science & Business Media, 2013.

[16] Stephen J Hartley. *Concurrent programming: the Java programming language*. Oxford University Press, Inc., 1998.

[17] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, revised first edition*. Morgan Kaufmann, 2012.

[18] Charles Antony Richard Hoare. Monitors: An operating system structuring concept. In *The origin of concurrent programming*, pages 272–294. Springer, 1974.

[19] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[20] Timothy G Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Pearson Education, 2004.

[21] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[22] Florian Negele, Felix Friedrich, Suwon Oh, and Bernhard Egger. On the design and implementation of an efficient lock-free scheduler. In *Job Scheduling Strategies for Parallel Processing*, pages 22–45. Springer, 2015.

[23] Jorge Luis Ortega-Arjona. *Una Introducción a los Patrones de Software.* 2006.

[24] Jorge Luis Ortega-Arjona. *Patterns for parallel software design.* John Wiley & Sons, 2010.

[25] Jayesh Patel. Trie datastructure explanation and simplified dictionary implementation in java.

[26] Jayesh Patel. ”trie datastructure explanation and simplified dictionary implementation in java.”. `http://javabypatel.blogspot.com/2015/07/trie-datastructure-explanation-and-applications.html`, 2015.

[27] Michael L Pinedo. *Scheduling: theory, algorithms, and systems.* Springer, 2016.

[28] Aleksandar Prokopec. Cache-tries: Concurrent lock-free hash tries with constant-time operations. *SIGPLAN Not.*, 53(1):137–151, February 2018.

[29] Oliver Sinnen. *Task scheduling for parallel systems*, volume 60. John Wiley & Sons, 2007.

[30] Andrei Smirnov. The missing article about qt multithreading in c.

[31] Andrew S Tanenbaum. *Sistemas operativos modernos.* Pearson Educación, 2009.

[32] Niklaus Wirth. *Algorithms+ Data Structures= Programs Prentice-Hall Series in Automatic Computation.* Prentice Hall, 1976.

[33] Albert Y Zomaya. *Parallel computing for bioinformatics and computational biology: models, enabling technologies, and case studies*, volume 55. John Wiley & Sons, 2006.