PRÁCTICA 2: CONCURRENCIA Y CONSISTENCIA DE MEMORIA

ARQUITECTURA DE COMPUTADORES

Curso 2015-2016

Grado en Ingeniería Informática

Universidad Carlos III de Madrid

PRIETO CEPEDA, JAVIER: **100 307 011**

JUÁREZ PUERTA, SANDRA: **100 303 528**

ÍNDICE

TABLAS DE PRUEBAS

ÍNDICE DE GRÁFICAS

1 Introducción

En este documento se procede a la explicación de la práctica 2, "Concurrencia y consistencia de memoria" de la asignatura Arquitectura de Computadores del grado en Ingeniería Informática de la Universidad Carlos III de Madrid. La realización de la misma la han llevado a cabo los alumnos Javier Prieto Cepeda y Sandra Juárez Puerta.

En primer lugar, se realizará una descripción del código. Esta descripción consistirá en la explicación de las 3 versiones del código, es decir, la versión secuencial, la versión concurrente con cerrojos y la versión libre de cerrojos. Tras la descripción del código, se indicarán las pertinentes decisiones de diseño tomadas.

Una vez descrito el código, se realizará una batería de pruebas para comprobar el correcto funcionamiento de la práctica. Tras esto, se procederá a comentar un análisis de los resultados obtenidos y del rendimiento conseguido, explicando los motivos de los mismos y comparando las diferencias entre la versión concurrente con cerrojos y la versión concurrente libre de cerrojos.

Por último, se realizarán una serie de valoraciones y conclusiones sobre la realización de la práctica.

2 Descripción del código

En este apartado, se procede a explicar las diferentes implementaciones de la práctica. En primer lugar, se va a realizar la explicación del código fuente proporcionado. Después, se realizará la explicación de las dos implementaciones concurrentes, tanto la versión con cerrojos como la versión sin cerrojos.

En la explicación de las versiones concurrentes, únicamente se explicarán las modificaciones necesarias para la implementación de la concurrencia, sin volver a explicar las partes del código comunes con la versión secuencial, las cuales ya han sido anteriormente descritas.

El funcionamiento del código, se basa en un sistema productor/consumidor con un buffer circular, en el cual el sistema productor, se encarga de insertar una serie de números en el buffer mientras este se encuentre con espacio suficiente. Por otro lado, el sistema consumidor, se encargará de sacar los elementos del buffer mientras en éste existan elementos. Ambos sistemas funcionarán siempre y cuando no se llegue al elemento que determina el final del sistema.

Cada elemento insertado en la cola, contará con 2 atributos: El valor del carácter a insertar en el buffer, y un booleano que indica si es el último elemento a insertar/sacar del sistema.

Además, el sistema cuenta con 2 funcionalidades. La primera, se encarga de encontrar en una serie de números, los números con mayor y menor valor que son insertados en el buffer. La segunda, se encarga de contar el número de caracteres que conforman todos los números insertados en el buffer, de forma que si se insertan 10 números de 2 cifras, el número devuelto serán 10*2=20.

Para la primera funcionalidad, la serie de números puede ser generada aleatoriamente mediante un generador de números aleatorios implementado en el código fuente, en el cuál se generan números de forma aleatoria entre un máximo y mínimo establecido. También se puede utilizar una serie de números contenida en un fichero, el cual ha de ser pasado por parámetro en la ejecución del código.

Para la segunda funcionalidad, la serie de números se obtiene de un fichero pasado por parámetro en la ejecución del código.

Para la ejecución de todas las funcionalidades, hay que determinar un tamaño máximo de buffer. Éste, contará con un puntero de lectura, que apuntará a la próxima posición a realizar la lectura, y un puntero de escritura, que apuntará a la próxima posición para realizar la escritura.

Todas las funcionalidades, llamarán a una función llamada *run_task()*, la cual se encuentra templarizada. Esta función, recibe 3 parámetros por referencia: generator, reducer y buffer. El primero, será el objecto encargado de generar los números, tanto aleatorios como de fichero, indicando además si es o no el último elemento a generarse. El segundo, será el objeto encargado de la realización de las funcionalidades, es decir, se encargará de contar el número de caracteres o de identificar el máximo número y el mínimo número dentro de la serie introducida en el buffer.

A continuación, se explicará la implementación de esta función en las 3 distintas versiones, puesto que en ella se encuentran las principales diferencias.

2.1 Versión secuencial

En esta versión, no se implementa ningún tipo de concurrencia, por lo que no se realiza la creación de hilos, todo el programa discurre en un único proceso e hilo. En la función *run_task()*, se realiza todo el proceso de inserción y vaciado del buffer, de manera que mientras el buffer no esté lleno, se insertarán elementos en él, de manera secuencial, es decir, se irán insertando elementos en el buffer hasta que la posición siguiente a la del próximo elemento a insertar, coincida con la posición del próximo elemento a leer ((next_write+1)==next_read). En ese momento, se dejarán de añadir elementos al buffer para comenzar a vaciar éste, de manera que se irá sacando elemento a elemento del buffer hasta que la posición del próximo elemento a leer sea igual a la del próximo elemento a insertar, momento en el que se dejará de vaciar el buffer, puesto que no habrá más elementos a sacar.

Este proceso, se repetirá hasta que se genere el último elemento a ser insertado en el buffer, y posteriormente, sea leído, momento en el que finalizará la función.

Para el tratamiento del buffer, existen una serie de métodos propios del buffer. Desde *run_task()*, se llamará en primer lugar, al método

full(), que devuelve un booleano que indica si el buffer está o no lleno.

Para la inserción de elementos en el buffer, se llamará al método *put()*, al cual se le pasa por parámetro el elemento a añadir y un booleano que indica si es o no el último elemento.

Para comprobar si el buffer está vacío, es decir, que no existan más elementos en el buffer para leer, se utiliza el método *empty()*, el cual devuelve un booleano indicando si está o no vacío.

Para realizar las lecturas (sacar elementos) de los elementos del buffer, se utiliza el método *get()*, el cual se encarga de ir sacando elementos del buffer.

Por tanto, para la implementación de la concurrencia, observamos que es necesario que sea implementada en las funcionalidades que ofrece el buffer para el tratamiento de éste.

2.2 Versión concurrente con cerrojos

En esta versión, la función *run_task()* es dividida en dos hilos, un hilo productor y un hilo consumidor. El primero, será el encargado de realizar la inserción de los elementos en el buffer, mientras que el segundo, será el encargado de realizar las lecturas de elementos del buffer. Por tanto, al existir concurrencia en el llenado/vaciado del buffer, es necesario implementar control en la concurrencia.

Las posibles condiciones de carrera que pueden darse, es a la hora de insertar/sacar elementos del buffer. Esto se debe, a la compartición de los punteros para escribir y leer, los cuales son modificados cada vez que se escribe o lee un elemento, y son compartidos entre los hilos productor y consumidor para comprobar que el buffer se encuentra lleno (en el caso del hilo productor) y para comprobar que el buffer se encuentra vacío (en el caso del hilo consumidor).

Por tanto, para resolver este problema, se ha decidido implementar un sistema de control de concurrencia que consta de un mutex y dos variables condicionales. Estas variables condicionales, sirven para controlar si el buffer se encuentra lleno o vacío.

En el método *put()*, que se encarga de la inserción de elementos en el buffer, se intenta tomar el mutex al comienzo del mismo. Este mutex, utiliza la estructura de unique_lock, basada en la entrada de un único hilo en la sección crítica. En caso de obtenerse el acceso a la

sección crítica, se comprueba que el buffer no se encuentre lleno. En caso de encontrarse lleno, se realiza un "wait" de la variable de condición que indica que el buffer se encuentra lleno, quedándose el hilo dormido esperando que se le despierte una vez el buffer esté disponible para la inserción de un nuevo elemento. Una vez puede insertar, se realiza la inserción, y una vez insertado el elemento en la nueva posición, se modifica el puntero de siguiente posición a escribir, apuntando a pos+1 en caso de no ser la última posición del buffer, o a la primera posición del buffer en caso de ser la última posición del buffer, ya que es un buffer circular. Una vez insertado el elemento y modificado el puntero de inserción, se realiza la liberación del mutex, y posteriormente, se realiza la notificación de que el buffer no está vacío a través de la variable de condición que comunica si el buffer está o no vacío. El tipo de notificación elegida es notify_one, puesto que solo se le debe de comunicar a un hilo en el peor caso (hilo productor).

En el método *get()*, que se encarga de la extracción de elementos del buffer, se intenta tomar el mutex al comienzo del mismo. Este mutex, utiliza la estructura de unique_lock, basada en la entrada de un único hilo en la sección crítica. En caso de obtenerse el acceso a la sección crítica, se comprueba que el buffer no se vacío. En caso de encontrarse vacío, se realiza un "wait" de la variable de condición que indica que el buffer se encuentra vacío, quedándose el hilo dormido esperando que se le despierte una vez el buffer esté disponible para la extracción de un nuevo elemento. Una vez puede extraer, se realiza la extracción, y una vez extraído el elemento, se modifica el puntero de siguiente posición a extraer, apuntando a pos+1 en caso de no ser la última posición del buffer, o a la primera posición del buffer en caso de ser la última posición del buffer, ya que es un buffer circular. Una vez extraído el elemento y modificado el puntero de extracción, se realiza la liberación del mutex, y posteriormente, se realiza la notificación de que el buffer no está lleno a través de la variable de condición que comunica si el buffer está o no lleno. El tipo de notificación elegida es notify_one, puesto que solo se le debe de comunicar a un hilo en el peor caso (hilo productor).

2.3 Versión concurrente libre de cerrojos

Al igual que en la versión anterior de concurrencia con cerrojos, en el código que nos proporcionan ya viene implementado el lanzamiento de dos hilos, uno actuará como productor y el otro como consumidor. Estos dos hilos podemos verlos en la función *run_task()*, el primer hilo será el encargado de meter datos en el buffer y el segundo será el que consuma dichos datos.

Como vemos en la versión anterior, esto nos produce condiciones de carrera debido al acceso y modificación de dos hilos diferentes a los mismos punteros del buffer. Dichos punteros son next_*write*_ y *next_read*, encargados de apuntar las nuevas posiciones para escribir (producir) y leer (consumir) datos en el buffer respectivamente.

Las tareas que realiza cada hilo están bien definidas, siendo el hilo productor el que llama al método *put()* del buffer y el hilo consumidor el que invoca a *get()*. Una vez aclarado esto podemos determinar en una primera observación que, en *put()* sólo se lee y modifica el puntero *next_write_*, y en *get()* ocurre lo mismo pero, en este caso, con el puntero *next_read_*. Llegados a este punto podríamos decir que cada hilo no accede a ninguna variable que pueda ser modificada por ningún otro hilo, pero es aquí cuando nos damos cuenta que en *put()* se hace una llamada al método *full()* que comprueba si el buffer está lleno, y en *get()* ocurre lo mismo pero llamando a *empty()* que comprueba si el buffer está vacío.

Teniendo lo anterior, se ve que *full()* realiza una lectura al puntero *next_read_* y *empty() a next_write_*. Por tanto después de estudiar el código en profundidad, vemos que tanto un hilo como el otro acceden los dos a los dos punteros de nuestro buffer, por tanto existen posibles condiciones de carrera sobre los punteros al realizar las diferentes acciones de cada. En cada uno de ellos se accede y modifica estos punteros compartidos y es posible que en un momento determinado un hilo lea un valor inadecuado de uno de estas variables (punteros) lo que puede producir que los accesos a nuestro buffer circular sean incorrectos.

Después de analizar los posibles problemas que se pueden dar al ejecutar la parte atómica nos proponemos a explicar la solución encontrada e implementada para esta tercera parte de la paráctica.

La implementación de una solución con atómicos, por tanto libre de cerrojos, puede ser variada dado que se pueden elegir diferentes

modelos de consistencia. En este caso, la solución dada cuenta con un modelo de consistencia secuencial memoria frente a otros posibles más relajados, tal y como se pide expresamente en el enunciado. Esta implementación con atómicos evita el uso de cerrojos y por el contrario hace uso de esperas activas.

Lo que se pretende es garantizar un ordenamiento entre las diferentes operaciones con las que se modifica a la misma posición de memoria para evitar que se produzca una carrera de datos. Con esto queremos garantizar que la lectura de una variable por un hilo y la escritura en esta misma variable por otro hilo devuelva o bien el valor previo antes de realizar la escritura o el valor escrito, por tanto queremos que sólo se den estos dos casos y no otro comportamiento no definido. Para hacer posible la anterior explicación haremos uso de tipos atómicos de datos.

Por tanto, las dos variables compartidas a las que modifican los dos hilos creados se han definido como variables de tipo int atómicas. Después de hacer lo anterior se ha tenido que modificar la forma en la que se realizan todas las lecturas y las escrituras en estas variables. Es necesario utilizar el método *load()* para realizar una lectura y el método *store()* para una escritura. Para el uso de ambos métodos es preciso especificar el ordenamiento de memoria aunque si no es especificado nada se tomará por defecto *memory_order_seq_cst* (consistencia secuencial). Para el uso de *store()* se necesita pasar por parámetro el nuevo valor que se quiere que se escriba desde la variable que lo invoque.

Para la realización de esta tercera parte se ha elegido, como se comentaba en párrafos anteriores, un ordenamiento secuencial de memoria frente a otros más relajados, para llevar a cabo este ordenamiento se ha especificado en los parámetros de *load()* y *store()* dicho orden con *memory_order_seq_cst*.

En este orden de memoria se establecen "barreras" entre las diferentes operaciones, permitiendo que éstas se ejecuten sin riesgos ni reordenamiento desde un punto de vista secuencial, con dicho orden, el programador no se encontrará "efectos secundarios" en las llamadas a *load()* o *store().* Memory:order_seq_cst puede llegar a provocar sincronizaciones hardware muy costosas.

Por otra parte además de lo anterior se han sustituido las excepciones con esperas activas con las cuales se estará comprobando repetidamente si se verifica una determina condición.

En el método *put()* se hace una comprobación para comprobar si el buffer está o no lleno, si se da que éste está lleno, el productor no puede producir más y, por tanto, el hilo productor se quedará computando continuamente y verificando cuando el buffer deja de estarlo, se genera, así, una espera activa de la que se sale cuando éste por fin ha dejado de estar lleno.

Al contrario que en el método anterior, ahora en *get()* se comprobará si el buffer está vacío. Cuando se de tal condición el consumidor no podrá sacar más elementos del buffer y por tanto deberá quedarse en una espera activa, computando todo el tiempo hasta que el buffer por fin deja de estar vacío y, por tanto, hay elementos en él y éstos pueden ser consumidos.

Dicho lo anterior, contamos con una solución libre de cerrojos en la que se cambian las excepciones por esperas activas y con un modelo de consistencia secuencial.

2.4 Estudio de la estructura de datos

La estructura de datos implementada, se trata de un buffer circular. Cada elemento del buffer, consta de dos elementos: valor y last. El primero, almacena el número del elemento almacenado en la posición, y el tipo al estar templarizado, es dependiente del tipo del elemento pasado. El segundo, es un booleano que indica si el elemento es el último del sistema, es decir, el elemento de finalización del sistema.

El buffer, es un array incluído dentro de un puntero inteligente (unique_ptr). La ventaja de este tipo, reside en que una vez se sale del ámbito en el cual se crea el unique_ptr, este se destruye al igual que el objeto al cual apunta, evitando posibles fugas de memoria. Por tanto, te asegura que el buffer se destruya (el array que lo representa). Además del array, el buffer consta con una variable de tipo entero que indica su tamaño. Las otras dos variables que conforman el buffer, son dos enteros. El primer entero, **next_read**, almacena la posición dentro del buffer, del próximo elemento a ser extraído. El segundo entero, **next_write**, almacena la posición dentro del buffer del próximo elemento a ser insertado.

Esta estructura, cuenta además con una serie de métodos que facilitan su uso. Entre ellos se encuentran:

- seq_buffer(): Es el constructor de la clase, recibe por parámetro el tamaño del buffer y crea éste.
- -~seq_buffer(): Es el destructor de la clase, el cual se encarga de destruir el buffer liberando la memoria ocupada por éste.
- size(): Se encarga de devolver el tamaño del buffer como un entero.
- empty(): Se encarga de devolver el booleano que determina si el buffer está vacío. La comprobación se encarga realizando la siguiente comparación: next_read==next_write. Es decir, si la próxima posición a ser leída es igual que la posición a realizar la próxima lectura, no existe ningún elemento disponible en el buffer para ser leído.
- full(): Se encarga de devolver el booleano que determina si el buffer está lleno. La comprobación se encarga realizando la siguiente comparación: (next_write+1)==next_read. Es decir, si tras la inserción en el buffer del nuevo elemento la nueva posición de escritura sería igual que la próxima posición de lectura, no inserto en el buffer el elemento ya que éste se encuentra "lleno". Esto nos determina que el **número máximo de elementos en el buffer simultáneamente sea de n-1**, siendo n el tamaño del buffer.
- put(): Se encarga de insertar un nuevo elemento en el buffer. Para ello, comprobará que el buffer no esté lleno. Si el buffer está lleno, devolverá una excepción, no insertando el elemento. Si el buffer está disponible, insertará el nuevo elemento en la posición apuntada por la variable *next_write_*, modificando después la variable, de forma que ésta apunte a la siguiente posición.
- get(): Se encarga de leer el elemento más antiguo disponible en el buffer. Para ello, comprobará que el buffer no esté vacío. Si el buffer está vacío, devolverá una excepción, no extrayendo ningún elemento. Si el buffer está disponible, leerá elemento de la posición apuntada por la variable *next_read_*, extrayéndolo del buffer, devolviendo el par de datos que conforman el elemento y

modificando después la variable *next_read_*, de forma que ésta apunte a la siguiente posición a ser leída.

- next_position(): Se encarga de devolver la siguiente posición en el buffer circular a una recibida por parámetro. Para ello, se le aplica el módulo del tamaño del buffer, siendo el resultado la posición siguiente resultante.

2.5 Decisiones de diseño

En el siguiente apartado, se procede a explicar las decisiones de diseño adoptadas más relevantes. Entre ellas, debemos destacar las siguientes:

- Dos variables de condición: En la versión *locked*, se ha decido realizar la implementación con dos variables de condición. La primera, se encarga de controlar que el buffer se encuentra lleno, mientras que la segunda, se encarga de controlar que el buffer está vacío. Sería posible realizarlo con una única variable de condición, pero por simplicidad y comprensión del código, se ha decidido el uso de estas dos variables.
- No usar los métodos *empty* y *full* de la clase buffer: Por simplicidad en el código, en la versión *locked* se ha decidido no realizar las llamadas a los métodos full y *empty* de la clase buffer. En su lugar, el código de la comprobación ha sido añadido en el bucle de comprobación de los método *get* y *put*.
- Se ha decidido realizar la implementación de la versión relajada sin cerrojos, implementando consistencia secuencial frente a un modelo más relajado. El motivo de ésta implementación, ha sido para asegurar el correcto funcionamiento de la versión del código. Se ha de destacar, que una implementación más relajada, podría producir menor coste en esperas. El inconveniente, sería no poder llegar a una solución válida que asegurase el correcto funcionamiento en todas las ejecuciones.
- No alinear las variables atómicas: Para un mejor rendimiento en tiempos, sería válido realizar una alineación en memoria del tamaño de la línea de caché. Esto se debe a que al modificar la línea y modificar el estado a inválido, si cada variable atómica se encuentra en una línea de caché diferente realizando la alineación, evita la invalidación de toda la línea con la consecuente penalización que supone. No se ha decidido realizar esta implementación por el coste en memoria que

supone, puesto que el resto de espacio de la línea, sería desperdiciado, y por tanto, hemos decidido obtener mejores resultados en memoria que en tiempo, ya que con un número grande de elementos, se desperdicia un tamaño considerable de memoria.

3 Batería de pruebas

En este apartado vamos a presentar nuestra batería de pruebas para tratar de demostrar el correcto funcionamiento del sistema. El objetivo de esta sección es corroborar que se cumple el funcionamiento básico de las dos versiones concurrentes. Para ello, se ha implementado una impresión por pantalla que indica cómo se realizan las lecturas/escrituras del buffer. Las pruebas de rendimiento se corresponden con la sección 4: Evaluación del rendimiento: Concurrencia con cerrojos VS libre de cerrojos.

3.1 Comprobación random

3.1.1 Atómicos

PRUEBA 1	
Objetivo	Comprobar el correcto funcionamiento de la función random con atómicos para un buffer más pequeño que el número de elementos.
Entradas	Random, 9 elementos y 3 de tamaño de buffer.
Salida	La salida se puede apreciar en la captura de pantalla. Dicha salida viene proporcionada por diferentes trazas creadas por nosotros de la siguiente manera: Put/Get X: in/at Y. Donde X es el elemento con el cual se realiza put/get e Y es la posición del buffer en la que se ha realizado dicho put/get.
Conclusión	Podemos concluir que el resultado es el esperado y por tanto este caso se realiza correctamente.

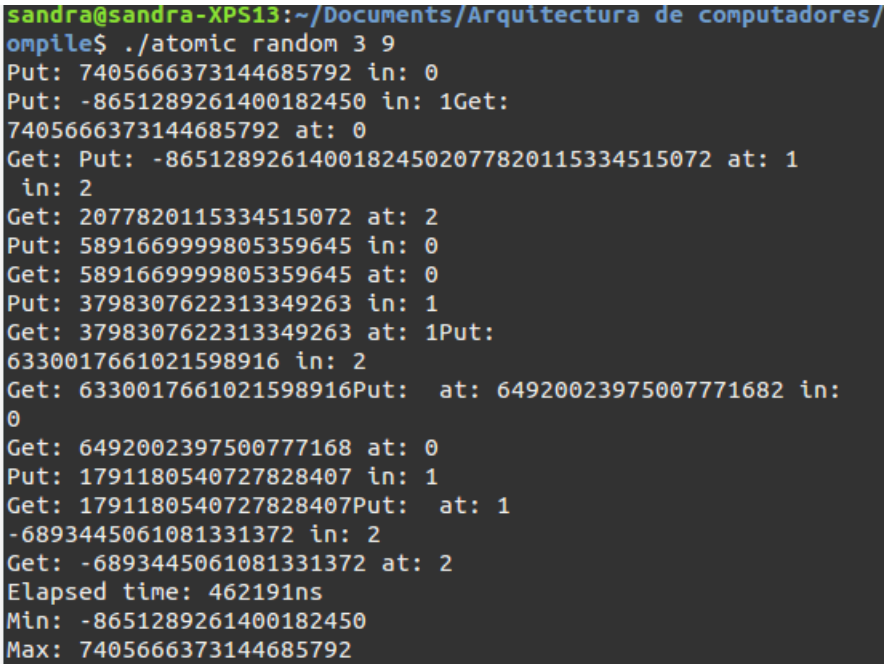
PRUEBA 1	
Captura de pantalla	

Tabla 1: Pruebas Random - Atómicos con buffer mayor que elementos

PRUEBA 2	
Objetivo	Comprobar el correcto funcionamiento de la función random con atómicos para un buffer del mismo tamaño que el número de elementos.
Entradas	Random, 9 elementos y 10 de tamaño de buffer.
Salida	La salida se puede apreciar en la captura de pantalla. Dicha salida viene proporcionada por diferentes trazas creadas por nosotros de la siguiente manera: Put/Get X: in/at Y. Donde X es el elemento con el cual se realiza put/get e Y es la posición del buffer en la que se ha realizado dicho put/get.
Conclusión	Se ha elegido para el tamaño del buffer una unidad más de la cantidad especificada en elementos, ya que el programa, a parte de los elementos especificados, vuelve a realizar otro put y otro get del último elemento. Por tanto podemos concluir que el resultado es el esperado y la solución funciona correctamente.

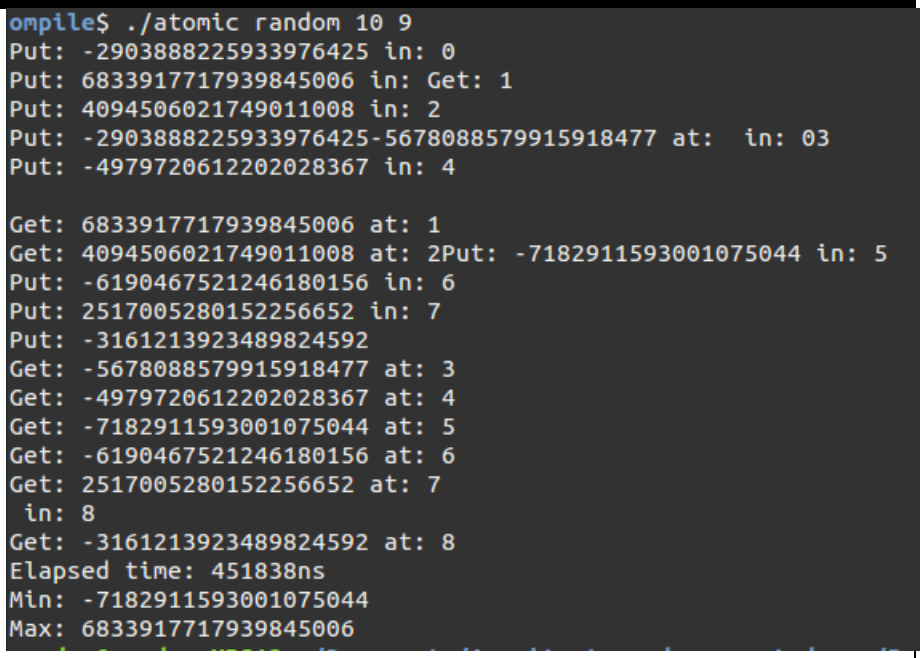
PRUEBA 2	
Captura de pantalla	

Tabla 2: Pruebas Random - Atómicos con buffer igual que elementos

PRUEBA 3	
Objetivo	Comprobar el correcto funcionamiento de la función random con atómicos para un tamaño de buffer más grande que el número de elementos.
Entradas	Random, 9 elementos y 3 de tamaño de buffer.
Salida	La salida se puede apreciar en la captura de pantalla. Dicha salida viene proporcionada por diferentes trazas creadas por nosotros de la siguiente manera: Put/Get X: in/at Y. Donde X es el elemento con el cual se realiza put/get e Y es la posición del buffer en la que se ha realizado dicho put/get.
Conclusión	Podemos concluir que el resultado es el esperado y por tanto este caso se realiza correctamente.

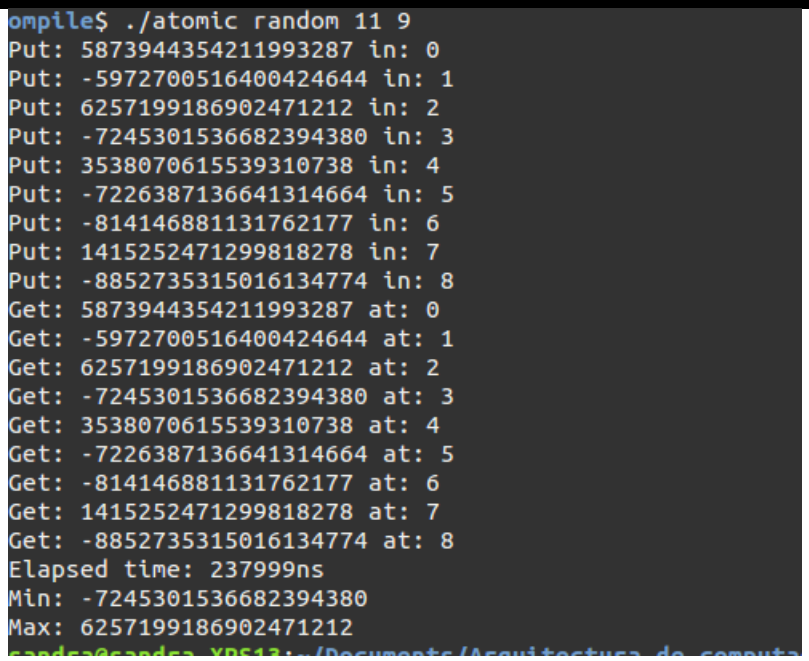
PRUEBA 3	
Captura de pantalla	

Tabla 3: Pruebas Random - Atómicos con buffer menor que elementos

3.1.2 Cerrojos

PRUEBA 4	
Objetivo	Comprobar el correcto funcionamiento de la función random con cerrojos para un buffer más pequeño que el número de elementos.
Entradas	Random, 9 elementos y 3 de tamaño de buffer.
Salida	La salida se puede apreciar en la captura de pantalla. Dicha salida viene proporcionada por diferentes trazas creadas por nosotros de la siguiente manera: Put/Get X: in/at Y. Donde X es el elemento con el cual se realiza put/get e Y es la posición del buffer en la que se ha realizado dicho put/get.
Conclusión	Podemos concluir que el resultado es el esperado y por tanto este caso se realiza correctamente.

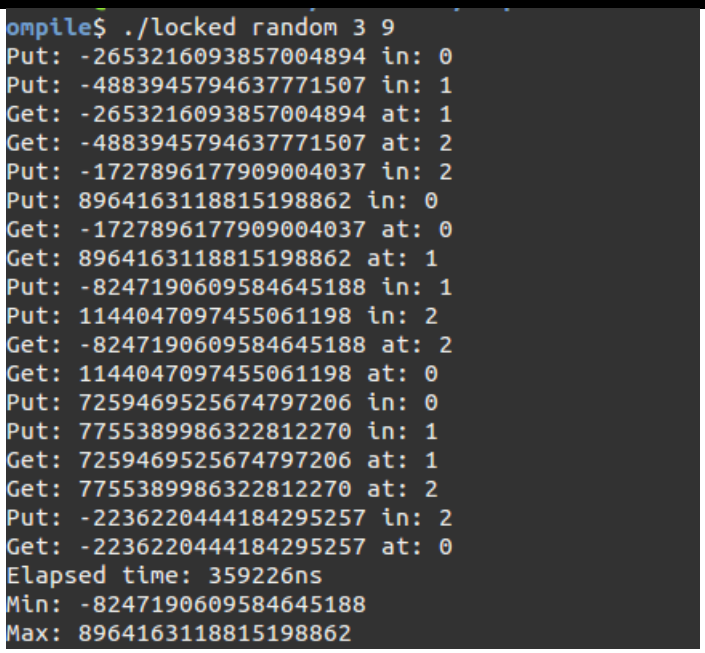
PRUEBA 4	
Captura de pantalla	

Tabla 4: Pruebas Random - Cerrojos con buffer menor que elementos

PRUEBA 5	
Objetivo	Comprobar el correcto funcionamiento de la función random con cerrojos para un buffer del mismo tamaño que el número de elementos.
Entradas	Random, 9 elementos y 10 de tamaño de buffer.
Salida	La salida se puede apreciar en la captura de pantalla. Dicha salida viene proporcionada por diferentes trazas creadas por nosotros de la siguiente manera: Put/Get X: in/at Y. Donde X es el elemento con el cual se realiza put/get e Y es la posición del buffer en la que se ha realizado dicho put/get.
Conclusión	Se ha elegido para el tamaño del buffer una unidad más de la cantidad especificada en elementos, ya que el programa, a parte de los elementos especificados, vuelve a realizar otro put y otro get del último elemento. Por tanto podemos concluir que el resultado es el esperado y nuestra solución funciona correctamente.

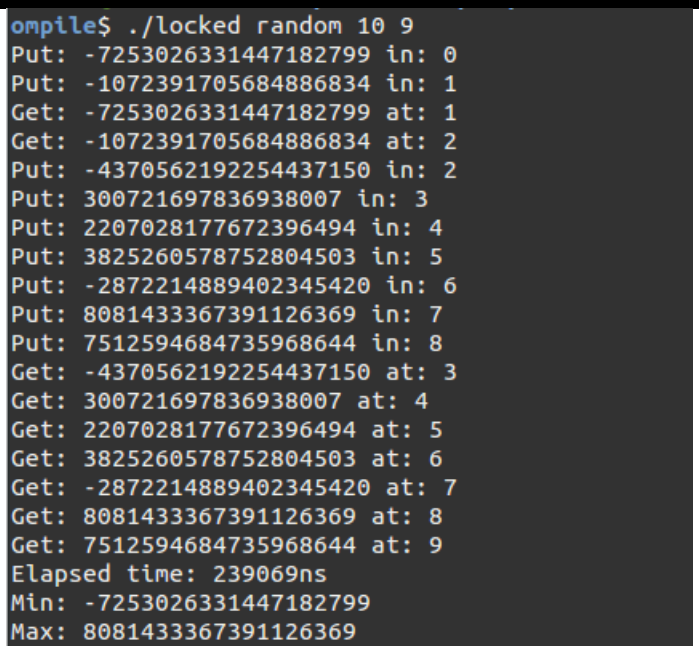
PRUEBA 5	
Captura de pantalla	

Tabla 5: Pruebas Random - Cerrojos con buffer igual que elementos

PRUEBA 6	
Objetivo	Comprobar el correcto funcionamiento de la función random con cerrojos para un tamaño de buffer más grande que el número de elementos.
Entradas	Random, 9 elementos y 11 de tamaño de buffer.
Salida	La salida se puede apreciar en la captura de pantalla. Dicha salida viene proporcionada por diferentes trazas creadas por nosotros de la siguiente manera: Put/Get X: in/at Y. Donde X es el elemento con el cual se realiza put/get e Y es la posición del buffer en la que se ha realizado dicho put/get.
Conclusión	Al tener un tamaño de buffer mayor que el número de elementos, la ejecución del programa debería realizarse sin preocupación por el llenado total del buffer, lo que implica que podrían realizarse todos los put de una vez y posteriormente los get de todos los elementos anteriores. Sin embargo observamos que el caso anterior no se muestra nuestra salida pues se intercalan put y get.

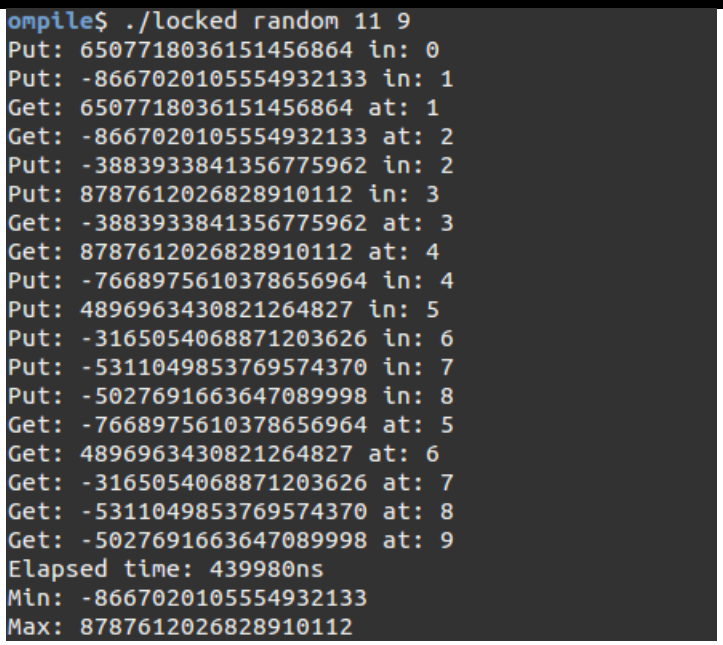
PRUEBA 6	
Captura de pantalla	

Tabla 6: Pruebas Random - Cerrojos con buffer mayor que elementos

3.2 Comprobación count

3.2.1 Atómicos

PRUEBA 7	
Objetivo	Comprobar el correcto funcionamiento de la función count con atomicos para un buffer más pequeño que el número de elementos.
Entradas	count, archivo con 9 elementos de un carácter cada uno y 3 de tamaño de buffer.
Salida	La salida se puede apreciar en la captura de pantalla. Dicha salida viene proporcionada por diferentes trazas creadas por nosotros de la siguiente manera: Put/Get X: in/at Y. Donde X es el elemento con el cual se realiza put/get e Y es la posición del buffer en la que se ha realizado dicho put/get.
Conclusión	Podemos concluir que el resultado es el esperado y por tanto nuestra solución es correcta.

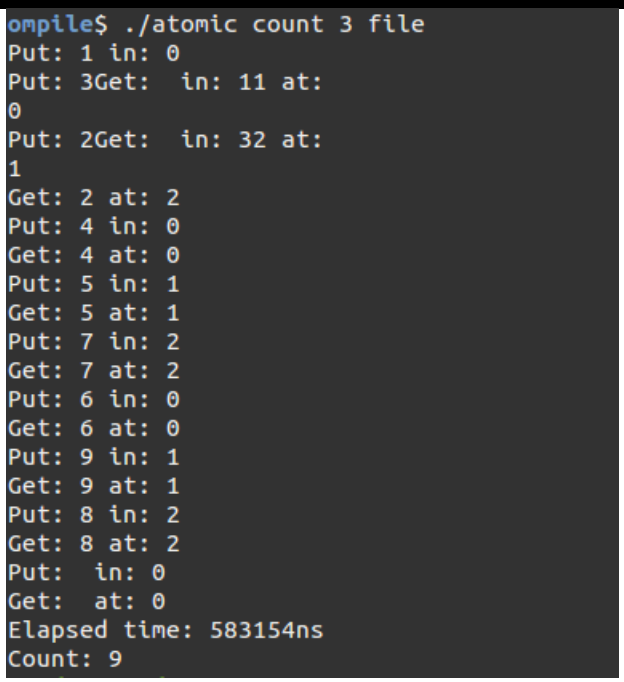
PRUEBA 7	
Captura de pantalla	

Tabla 7: Pruebas Count - Atómicos

3.2.2 Cerrojos

PRUEBA 8	
Objetivo	Comprobar el correcto funcionamiento de la función count con cerrojos para un buffer más pequeño que el número de elementos.
Entradas	count, archivo con 9 elementos de un carácter cada uno y 3 de tamaño de buffer.
Salida	La salida se puede apreciar en la captura de pantalla. Dicha salida viene proporcionada por diferentes trazas creadas por nosotros de la siguiente manera: Put/Get X: in/at Y. Donde X es el elemento con el cual se realiza put/get e Y es la posición del buffer en la que se ha realizado dicho put/get.
Conclusión	Podemos concluir que el resultado es el esperado y por tanto nuestra solución es correcta.

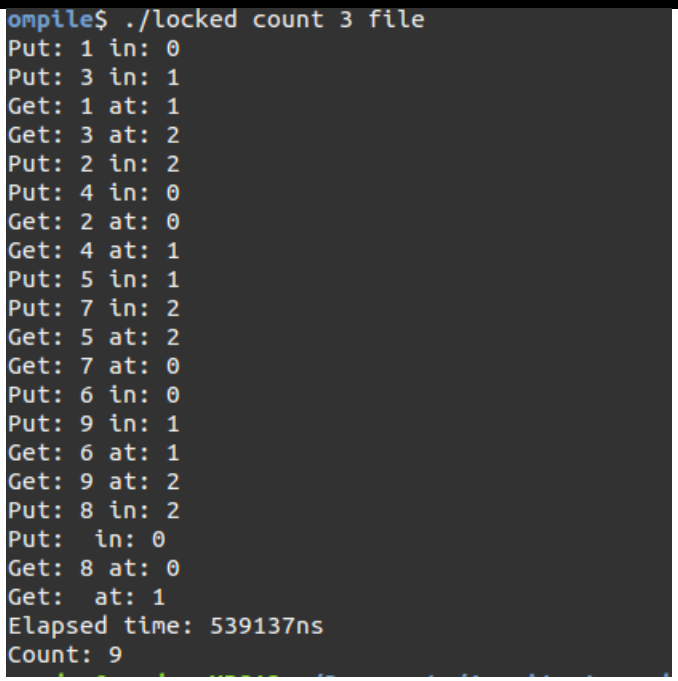
PRUEBA 8	
Captura de pantalla	

Tabla 8: Pruebas Count - Cerrojos

4 Evaluación del rendimiento: Concurrencia con cerrojos VS libre de cerrojos

En este apartado, vamos a proceder a evaluar el rendimiento de las 3 versiones, la versión secuencial y las 2 concurrentes implementadas. Para ello, hemos realizado evaluado 4 parámetros distintos. El primero de ellos, el número de cambios de contexto que se produce durante la ejecución. El segundo de ellos, es el uso de CPU que ejerce cada una de las versiones. El tercero, el número de migraciones de cpu que sufre el programa durante su ejecución. Y por último, el cuarto, el tiempo de ejecución final, mediante el cual podremos obtener el speed-up obtenido con respecto a la versión secuencial.

Para ello, hemos decidido mantener fijo el número de elementos en 10.000.000 de elementos, de forma que se analice el comportamiento en función del número de veces que se llena el buffer.

Las pruebas han sido realizadas mediante la herramienta perf junto con el parámetro stat, que devuelve la media de algunos registros de la CPU durante la ejecución del programa. No es una herramienta que pueda concluir con exactitud estos datos, pero si da una aproximación muy cercana a la real.

La máquina sobre la cual han sido realizadas las prácticas, ha sido un Intel core 2 duo-E7300 con 3.06 GHz, con 2 cores. Al disponer únicamente nuestro programa de 2 hilos (productor/consumidor) el uso máximo teórico de CPU será igual a 2.

4.1 Cambios de contexto

Mediante esta prueba, pretendemos observar cuál de las versiones concurrentes realiza un mayor número de cambios de contexto. Esto se debe, a que es un buena forma de medir el impacto en el tiempo que tienen los cambios de contexto sobre el resultado final:

Cambios de contexto				
Versión/tamaño buffer	**500**	**50000**	**500000**	**10000000**
Secuencial	792	617	700	963
Cerrojos	181689	160572	58461	8557
Atómicos	1970	2078	1986	2039

Tabla 9: Resultados cambios de contexto

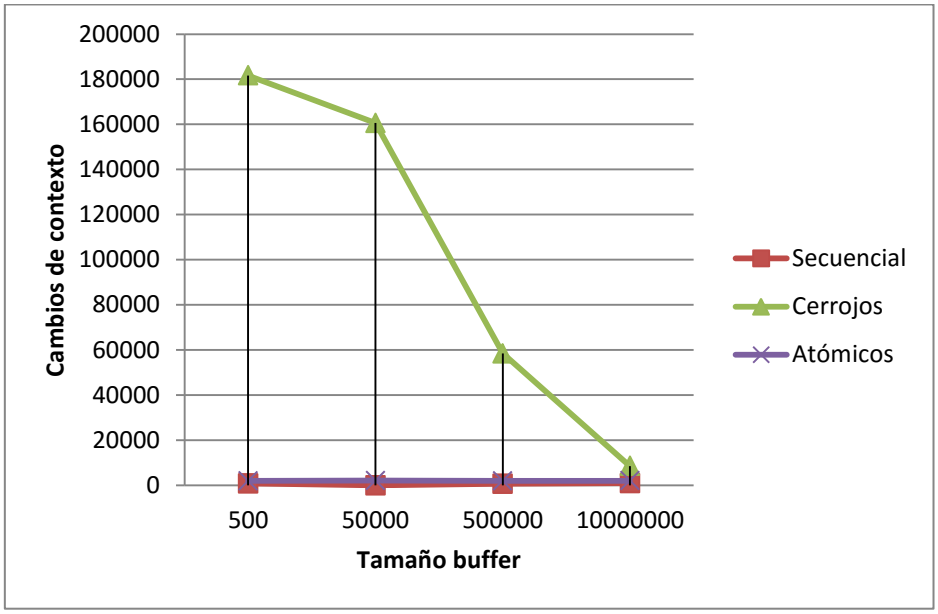


Ilustración 1: Gráfica cambios de contexto

Como podemos observar, la versión con cerrojos realiza con gran diferencia un mayor número de cambios de contexto. Esto se debe a la disputa entre los hilos por tomar el mutex, lo cual resulta muy costoso en cuanto a tiempo final de ejecución. Se puede observar, que a medida que se reduce el número de llenados del buffer, el número de cambios de contexto disminuye, lo cual se debe a un menor número de esperas por llenado de buffer. En cuanto a la versión de atómicos, se puede observar que se mantiene estable en el número de cambios de contexto, aunque al igual que la versión con cerrojos, estos disminuyen a medida que el número de llenados del buffer disminuye.

4.2 Migraciones de CPU

Mediante esta prueba, pretendemos ver el número de migraciones que realiza de CPU que realiza el sistema operativo sobre las distintas versiones, lo cual puede influir en el tiempo final de ejecución:

Migraciones de CPU				
Versión/tamaño buffer	**500**	**50000**	**500000**	**10000000**
Secuencial	11	10	8	10
Cerrojos	96	64	57	39
Atómicos	2	3	1	2

Tabla 10: Resultados migraciones de CPU

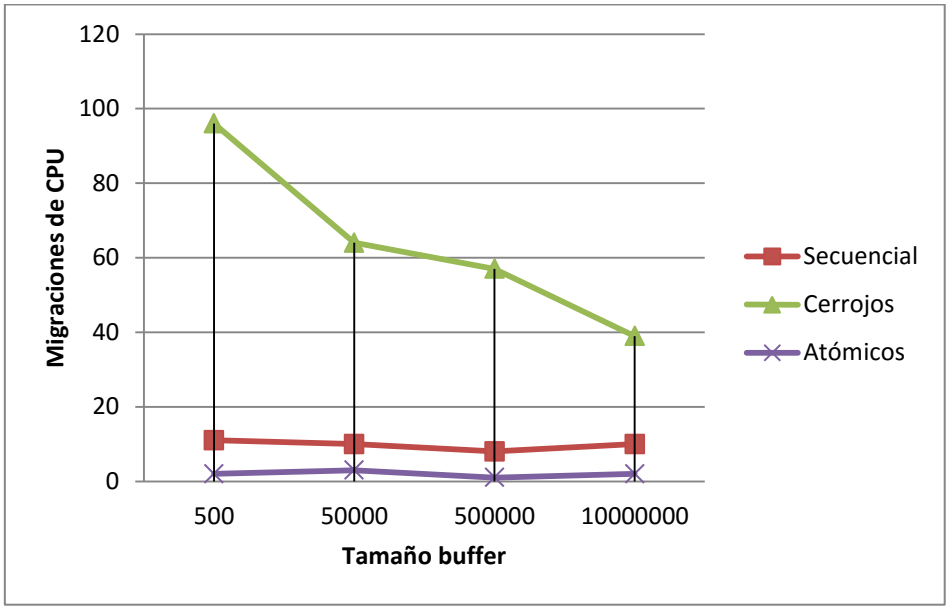


Ilustración 2: Gráfica migraciones de CPU

Como podemos observar, la versión con cerrojos realiza un mayor número de migraciones de CPU. Esto se debe que los hilos realizan una mayor disputa por el mutex, lo cual es bastante costoso y puede tener una repercusión importante en el tiempo final de ejecución. Además, el número de migraciones disminuye a medida que se reducen los llenados del buffer. Podemos observar que la versión con atómicos se mantiene bastante estable. Hay que decir, que no es un análisis concluyente puesto que el planificador del sistema operativo, puede realizar las migracioens de CPU en función de los procesos externos a la aplicación que se encuentren corriendo en el sistema.

4.3 Uso de CPU

Mediante esta prueba, pretendemos observar el aprovechamiento de las CPU's que realizan las versiones. Con ello, se puede determinar cuál de las versiones pierde un menor tiempo en la disputa por la sección crítica:

Uso de CPU				
Versión/tamaño buffer	**500**	**50000**	**500000**	**10000000**
Secuencial	0,999	0,999	0,999	0,999
Cerrojos	1,299	1,438	1,781	1,918
Atómicos	1,984	1,986	1,988	1,986

Tabla 11: Resultados uso de CPU

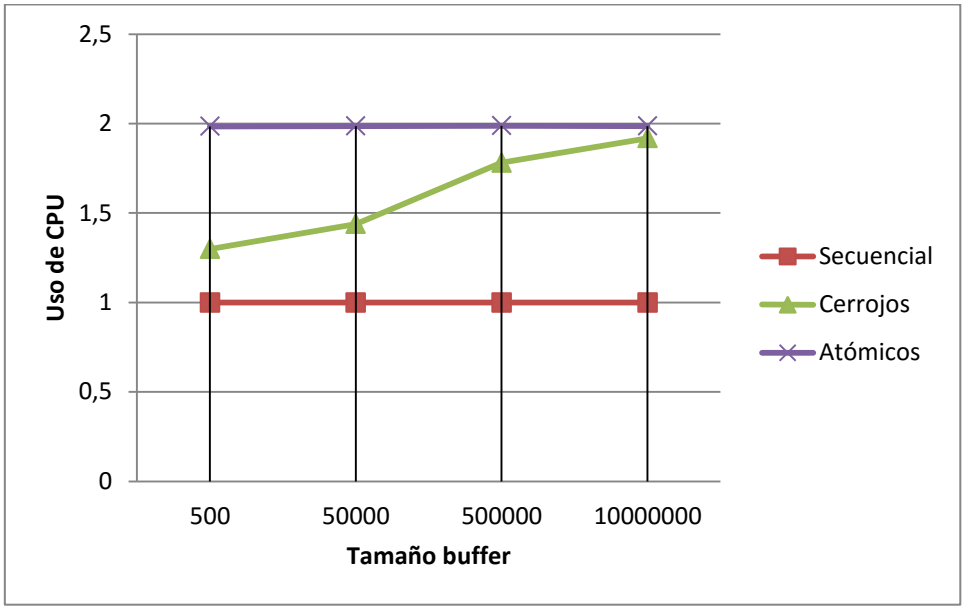


Ilustración 3: Gráfica uso de CPU

Como podemos observar, la versión secuencial tiene un uso completo de una CPU. Esto se debe a que esta versión únicamente consta de un proceso. Las dos versiones concurrentes, disponen de 2 hilos, por lo que su máximo teórico será de 2. Como podemos observar, la versión de atómicos realiza un mayor uso de CPU, puesto que constantemente utiliza 2 cpus (el máximo en nuestro caso). La versión de cerrojos, no acaba de aprovechar al máximo las CPU's, lo cual puede deberse a la disputa por los cerrojos con la pérdida de rendimiento que ello conlleva. Se puede observar como a medida que se reduce el número de llenados de buffer, el aprovechamiento de CPU de la versión con cerrojos mejora, lo cual se debe al menor número de esperas por llenado de buffer que va experimentando el sistema.

4.4 Tiempo ejecución

Mediante esta prueba, pretendemos ver el tiempo final de ejecución de las distintas versiones, pudiendo obtener conclusiones acerca de cuál de ellas es más rápida. El tiempo de ejecución obtenido, es en segundos, siendo los resultados los siguientes:

Tiempo de ejecución				
Versión/tamaño buffer	**500**	**50000**	**500000**	**10000000**
Secuencial	8,6522	6,8208	7,6967	10,5790
Cerrojos	16,9848	14,0774	11,3029	11,2411
Atómicos	7,8625	7,9589	7,7858	7,7648

Tabla 12: Resultados tiempos de ejecución

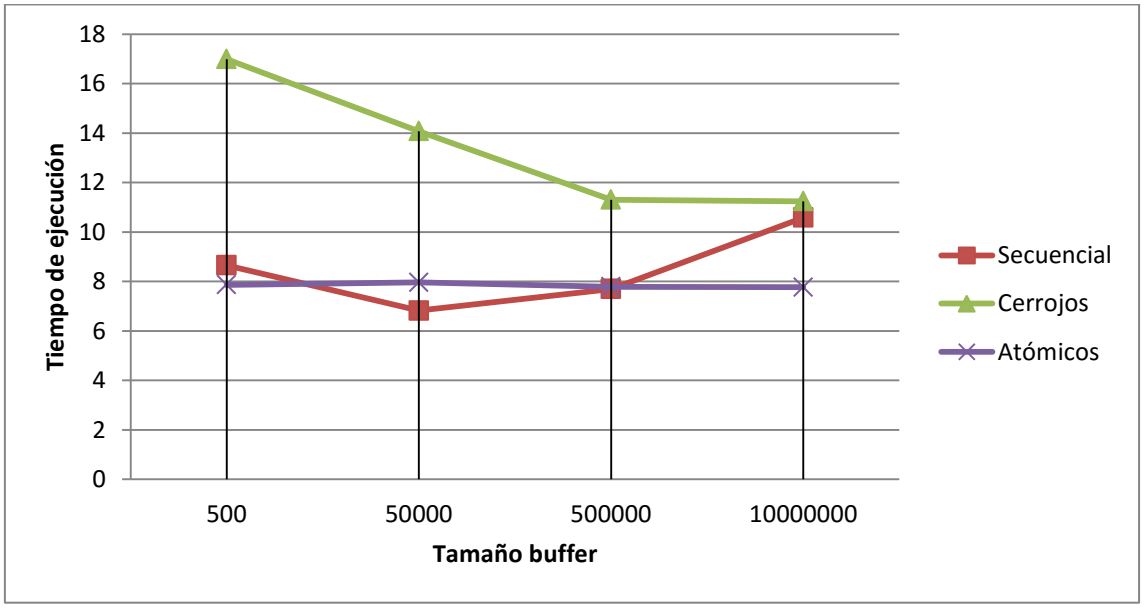


Ilustración 4: Gráfica tiempos de ejecución

Como podemos observar, la versión secuencial se mantiene estable en cuanto al tiempo de ejecución, lo cual se debe a que al mantenerse fijos el número de elementos, la variación del tamaño del buffer no le afecta, al solo disponer de un único proceso. La versión con cerrojos, a medida que el número de llenados aumenta, empeora en tiempos, debido a que existe un menor aprovechamiento de la concurrencia. La versión de atómicos, se mantiene estable e incluso mejora al existir menor número de llenados del buffer. Esto se debe al uso de una consistencia secuencial.

Por tanto, en último lugar, vamos a mostrar el speed-up obtenido en las versiones concurrentes con respecto a la versión secuencial:

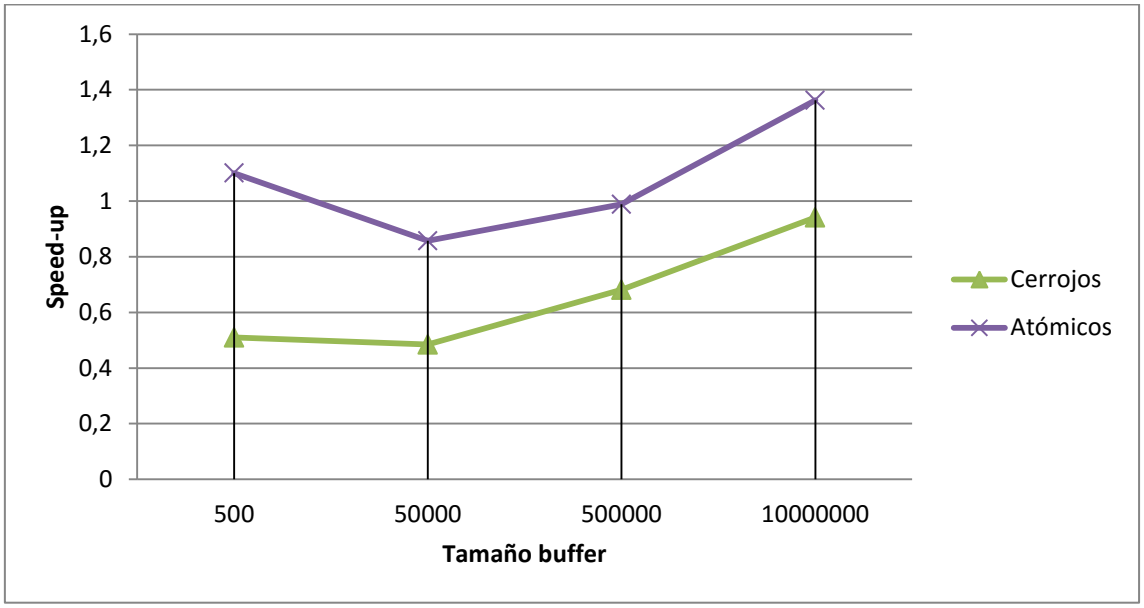


Ilustración 5: Gráfica speed-up

Por tanto, podemos concluir, que la versión que logra mejorar a la versión secuencial inicial, es la versión libre de cerrojos. Esto se debe entre otras cosas, como hemos visto en el análisis previo, a un menor número de cambios de contexto y migraciones de CPU, lo cual provoca un mayor aprovechamiento del uso de CPU con la consiguiente mejora en tiempos. La versión con cerrojos, mejora a medida que el número de llenados del buffer disminuye, debido a que se produce un menor número de esperas activas por llenado y vaciado del mismo. Por tanto, en la versión de atómicos, al evitar las condiciones de carrera por tomar el mutex entre los hilos, se obtiene un mejor rendimiento, con la mejora en tiempo de ejecución que conlleva.

5 Conclusiones

Una vez finalizada la práctica, han de considerarse una serie de aspectos importantes a destacar. Entre estos aspectos, se encuentra la ventaja en rendimiento del sistema de atómicos. El impacto es significativo frente al uso de sistemas de sincronización entre hilos como los mutex y las variables condicionales, debido al menor número de cambios de contexto que se obtiene con los atómicos, ya que esto es muy costoso. Además, el evitar las condiciones de carrera por tomar el mutex, nos da una mejora considerable en tiempos y rendimiento.

Otro aspecto importante, es la utilización de un buffer circular. No es tan llamativo el uso del buffer, sino el modo en el cual se implementa, ya que tal vez habría sido más lógico el uso del último elemento de la serie como identificador del fin de la serie mediante el atributo *last* y no un elemento con datos "basura".

Habría sido interesante realizar una sincronización entre un mayor número de hilos productores/consumidores en lugar de este sistema SPSC (single productor/single consumer), puesto que el análisis con mayor número de elementos a sincronizar, habría supuesto una conclusión más clara sobre las ventajas e inconvenientes entre los dos sistemas de concurrencia. Esto se debe, ya que teóricamente, con un gran número de hilos, el sistema de mutex se mantiene estable en bloqueos frente al modelo relajado, ya que al existir un mayor número de hilos atacando a una misma variable atómica, existe un mayor número de bloqueos que hacen ineficiente al sistema.

En definitiva, resulta interesante indagar en las mejoras que supone el uso de atómicos en este tipo de problemas, puesto que su utilización puede suponer una mejora considerable en rendimiento en ciertos tipos de problemas. Además, resulta útil saber en que situaciones o contextos resultan más eficientes cada uno de los modelos implementados en la práctica.