12 DE DICIEMBRE DE 2014

ARQUITECTURA DE COMPUTADORES

LABORATORIO 2: PROGRAMACIÓN LIBRE DE CERROJOS EN C++.

Álvaro Gómez Ramos 100307009

Carlos Contreras Sanz 100303562

Contenido

Descripción del código

En este apartado se detallará el diseño de las colas de datos implementadas en el código. Deberá contener, al menos, la explicación de los mecanismos de concurrencia utilizados, así como la justificación de los mismos: por qué son necesarios, qué problemas resuelven, etc.

En esta práctica se nos pedía implementar un método main desde el que ejecutar las distintas implementaciones de cola y pila. Este main es el mismo dentro de cada estructura de datos, y varia muy ligeramente de una a otra (en llamadas y tipos de datos básicamente).

Este método main primero comprueba que tenemos todos los parámetros necesarios, después toma tiempo de inicio, lanza los hilos, hace join de los mismos, y por ultimo calcula tiempo total empleado. Las operaciones de push/pop o enqueue/dequeque se realizan sobre una pila o cola declarada de forma global en el método main de cada caso, para poder acceder todos los hilos métodos a la misma estructura de datos dentro de cada ejecución.

Los hilos se lanzan mediante un for que itera tasks_per_thread veces, lanzando un push/enqueue en iteraciones pares y un pop/dequeue en iteraciones impares. También podría lanzarse un push/enqueue y un pop/dequeue en cada iteración e iterar la mitad de veces, quizás mejorando algo en rendimiento.

Antes de empezar con las dos implementaciones diferentes de la cola, nos hemos planteado las principales cosas a tener en cuenta en cada una de las operaciones:

Para la operación de enqueue es importante comprobar si es el primer objeto que entra o no, para actualizar el head en consecuencia (en caso de estar la cola vacía, este nuevo nodo seria head y last de forma simultánea). Además al encolar se deberá siempre añadir el objeto al final de la cola, como nuevo last, actualizando el puntero del last que teníamos antes, de forma que apunte a este nuevo nodo que introducimos.

Para la operación de desencolar, deberemos de actualizar el head, siendo el nodo al que apunta del next del actual head. Devolveremos los datos contenidos en el objeto que estamos desencolando, los datos del viejo head.

Tanto en la implementación de la cola con mutex, como en la de atómicos, se usan los mismos datos. Se usa una estructura nodo, que contiene: datos, puntero a nodo siguiente en la fila (que llego después que yo) y el propio constructor de la estructura.

Además se tiene el mutex para la implementación de mutex, y el puntero al head y al last (normalmente conocido como tail de la cola).

Implementación de la cola con mutex

Para implementar la cola con mutex, se hace uso de un mutex. El método enqueue lo primero que haces es crear el nuevo nodo, con los datos que nos han pasado como parámetros. Tras eso cogemos el mutex, y en la sección protegida: si la cola está vacía digo que este nuevo nodo es head, si la cola ya tenía elementos, apunto next del last a este nuevo nodo para encolarlo. En cualquier caso digo que este nuevo nodo es last. Tras eso libero el mutex.

Para el método desencolar, lo primero que se hace es reservar el mutex, y en la sección protegida hacemos: primero guardamos puntero a la cabeza actual, si resulta que no había cabeza, no puedo desencolar, luego libero mutex y devuelvo objeto vacío. Si resulta que sí podemos desencolar, actualizo la nueva cabeza como el next de la antigua. Libero el mutex y devuelvo los datos de la antigua head, que es la que he desencolado.

Gracias a la utilización de mutex, podemos proteger secciones enteras del código, asegurándonos de que nadie ejecute a la vez que nosotros, y evitando que se den condiciones de carrera o inconsistencias en los datos. Con esto no nos tenemos que preocupar de que otro hilo modifique la cabeza o la cola mientras que estamos realizando operaciones.

Si no usásemos métodos de control podría darse el caso, por ejemplo, de que dos hilos hiciesen dequeue sobre una cola con un elemento, y ambos devolviesen elemento.

Resolver ese problema es la razón de que se usen métodos de control o sincronización.

En nuestro caso, protegemos todo lo susceptible de dar errores si se hiciese de forma concurrente, esto es, básicamente proteger todos los cambios.

De esta forma nos aseguramos de que no habrá errores, pero a cambio aumentamos el coste temporal del programa. Para comprobarlo podemos incluir impresiones por pantalla de cada operación, estando seguro de que si lo introducimos en el main, aparecerán en el orden en que se ejecutan.

Implementación de la cola con atómicos

La implementación de la cola con atómicos resulta más complicada que con mutex, y resulta mucho más difícil de comprobar su funcionamiento. Esto es porque en este caso no se protegen secciones, si no que se realizan operaciones atómicas, en las que se asegura que no hay interrupciones durante su ejecución, pero no podemos asegurar que no habrá interrupciones durante toda una sección (podríamos usar por ejemplo test_and_set para implementar un mutex, pero eso no cumpliría la condición de no usar cerrojos.).

Hemos tenido por tanto que usar el mecanismo atómico de compare_exchange, combinado con bucles para asegurar que se realizaban determinadas operaciones.

Si analizamos el código, lo primero es crear el nuevo nodo, e incluir los datos que nos pasan por parámetros. Después guardamos punteros tanto al viejo head, como al viejo last. Esto es para usar con los compare_exchange, quienes irán actualizando estas referencias en cada una de las iteraciones del while que veremos más abajo.

Tras crear dichos punteros, tenemos un while que se ejecutara siempre que no hay ninguna cabeza. Este bucle se usara para, mientras que la cola este vacía, diremos que el nodo que encolamos es la cabeza también.

Después de este bucle, tenemos otro while cuya condición es un compare_exchange, que se usa para intentar cambiar el last hasta que sea posible. Tras eso, y siempre que exista el puntero que habíamos creado al viejo last, diremos que el next de ese viejo last, apunta a nuestro nuevo nodo.

El método dequeue lo primero que hace es crear puntero al viejo head. Tras eso, tenemos un while que siempre que haya una cabeza intentará hacer que su next se convierta en cabeza. Cuando lo consiga, o bien devuelve los datos que contenía el viejo head, o si no existía, devuelve el objeto no creado.

Tal y como tenemos implementada la cola atómica, hay un error que puede llegar a ser bastante importante, que no hemos sabido solucionar: si se diese el caso de que otro hilo ejecutase operaciones de encolar o desencolar entre la comprobación de que no hay cabeza y la actualización del last, en una cola en la que solo haya un nodo, resultaría una cola sin head, de la que si que se podría encolar, pero en la que no se podría desencolar.

En las numerosas pruebas que hemos realizado no hemos detectado esto en ninguno de los casos, suponemos por tanto que es altamente improbable que se de esta situación. Otro problema similar seria dos nodos encolando a la vez, y que uno se ejecute entre los dos while del otro.

Algunas de las posibles soluciones que se nos han ocurrido para este problema han sido comprobar que existe cabeza a la vez que se realiza al comprobación de que se puede actualizar el last con el nuevo valor, o bien volver a realizar la comprobación de que existe cabeza después de actualizar el last, pero en todos los casos o bien no sabíamos implementarlo, o bien la solución creaba otras situaciones potencialmente problemáticas.

Es por esto que, a pesar de ser conscientes de este error, debemos presentar el código con este fallo incluido, habiendo sido incapaces de solucionarlo.

Ejercicios a realizar

Ejercicio 1

Explique cómo funcionan las dos pilas proporcionadas, prestando especial atención a las funciones push y pop.

Ambas pilas proporcionadas hacen uso de una estructura para guardar los datos del nodo, y un puntero al nodo que se sucede en la pila. El que se sitúa debajo de él. También se hace uso de un puntero a la cabeza, en el caso de la implementación con mutex, se usa uno.

Implementación con mutex

La función push lo primero que hace es crear e inicializar el nuevo nodo con los datos obtenidos por parámetros. Tras eso entramos en la sección protegida por el mutex, en la que se dice que le next del nuevo nodo es la cabeza vieja, y que la nueva cabeza es éste nuevo nodo. Después simplemente liberamos el mutex.

La función push realiza todo en una sección protegida por el mutex. Primero crea puntero a la vieja cabeza, después, si no había head, devuelve objeto no creado, pero en caso de que si hubiese, dice que la nueva cabeza en el next de la vieja. Tras eso libera el mutex y devuelve los datos de la vieja cabeza.

Implementación con atómicos

En el push de la implementación con atómicos lo primero que se hace es crear a inicializar le nodo que recibimos como parámetros. Tras eso cargamos el viejo head en el next del nodo nuevo.

Entonces, por medio de un compare_exchange_weak en un while, realizamos el cambio de head, y terminamos la función.

En este caso, el problema que teníamos nosotros con la cola atómica no se produce, ya que: si tenemos dos nodos A y B que realizan push simultáneos, en el caso de que B se iniciase y completase entre la creación del puntero de A y su while (es decir, next de A apunta a head original, y no ha llegado a cambiar el head), al fallar el compare_exchange se actualiza dicha referencia y se apilan en el orden correcto, con los punteros correctos.

En la función pop, lo primero que se crea es el puntero al head viejo. Después se intenta cambiar la vieja cabeza por el next de la vieja cabeza, siempre que exista dicho head (se usa una vez mas el compare_exchange_weak). Por ultimo devolvemos o bien el data del head viejo, o bien un objeto sin crear si no hay head.

¿Por qué la función push utiliza la función de la biblioteca de atómicos compare_exchange_weak, en lugar de compare_exchange_strong?

Como hemos mencionado se usa un compare_exchange_weak en lugar de usar un compare_exchange_strong. Esto se debe a que el weak es más eficiente en cuanto a tiempo: el débil puede verse interrumpido por fallos espurios (fallos debidos a cambios de contexto, por ejemplo, en lugar de debidos a escrituras concurrentes, que sería el fallo real), mientras que el strong no. Esto se consigue añadiendo comprobaciones en el strong que no posee el weak, haciendo que tarde más el fuerte. Ya que los fallos espurios son poco frecuentes y que se van a ejecutar cinco millones de push y cinco millones de pop, ese tiempo que perdemos con cada strong respecto al weak haría que variase el tiempo total de ejecución del programa.

Usamos por tanto weak dentro de un bucle, asegurando que en caso de que se produzcan fallos espurios, no salgamos de dicho bucle hasta que no se haya completado satisfactoriamente el cambio.

Ejercicio 2

Ejecute un programa en el que utilice las dos pilas con diverso número de threads. Como mínimo deberá probar con 2, 4, 8, 16 y 32 threads, trabajando simultáneamente sobre cada una de las pilas. Debe garantizar que entre todos los hilos se ejecutan siempre 5000000 llamadas a push y 5000000 llamadas a pop. Represente los tiempos de forma gráfica y comente los resultados.

Tal y como tenemos implementado el main, a la hora de ejecutar tenemos que usar como parámetros el número de hilos que deseemos, y el número de tareas por hilo, de tal forma que el producto de ambos de como resultado los 10 millones de operaciones totales pedidas. Es decir se ejecutará con las siguientes cifras:

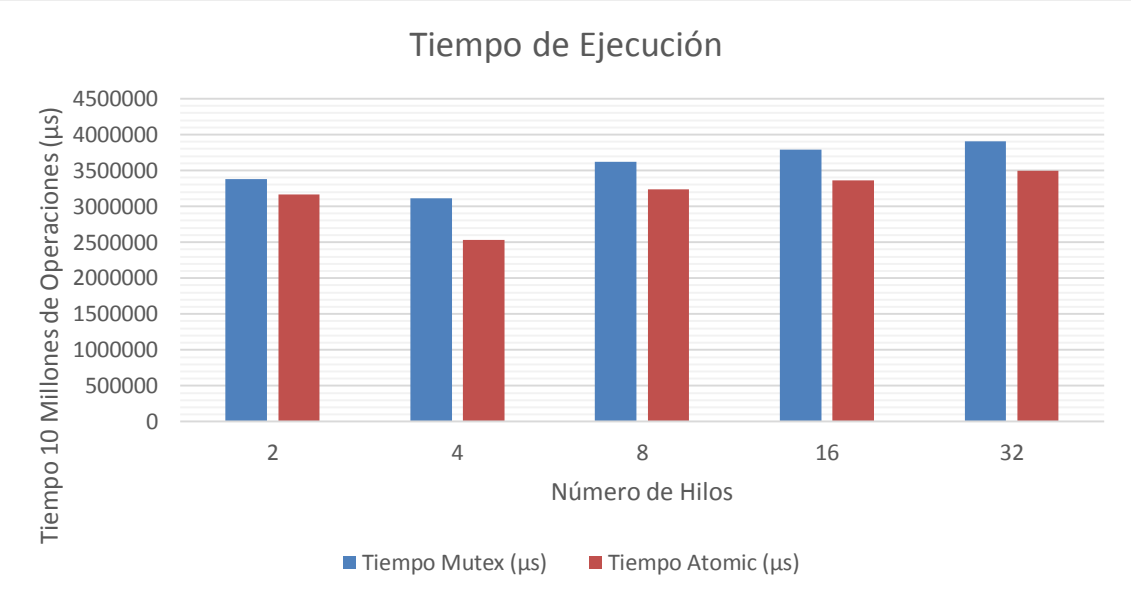
Hilos	Operaciones por hilo
2	5.000.000
4	2.500.000
8	1.250.000
16	625.000
32	312.500

De esta forma cada hilo hará ese número de operaciones, y sabiendo que los hilos pares hacen push y los impares pop, tenemos el número de operaciones requeridas.

Las tablas de datos son los siguientes:

Hilos	Op push/pop	Tiempo Mutex (µs)	Tiempo Atomic (µs)
2	5 M	3380458	3165468
4	2,5 M	3112829	2534702
8	1,25 M	3624422	3239449
16	625 K	3791716	3359156
32	312,5 K	3904178	3496417

Donde tenemos recogidos los tiempos que tarda en ejecutarse el código en microsegundos, para cada número de hilos. Si los representamos de forma gráfica obtenemos:



¿Cuál es la evolución de rendimiento? ¿Por qué? ¿Qué diferencias hay entre las dos tecnologías? ¿Cómo afectan estas diferencias al rendimiento?

Si nos fijamos en los datos de forma global, vemos una clara tendencia ascendente, aunque no demasiado acusada. Vemos que con 4 hilos, la ejecución mejora ligeramente el tiempo de la de 2 hilos. A partir de aquí aumenta, teniendo en el cambio de 4 a 8 su salto más grande, y siendo los siguientes de menos magnitud.

La bajada del tiempo de 2 a 4 la achacamos a que es más fácil repartir tareas en hilos físicos, ya que con 2 hilos, teníamos uno que siempre hace push, y otro que siempre hace pop. Posiblemente, con cuatro hilos, el reparto de tareas se hace de forma más eficiente, y es posible paralelizar mejor las partes que se pueda (partes no protegidas por el mutex ni los compare_exchange).

Esta mejora a la hora de reparto de tareas y paralelización se ve anulada por el hecho de que demasiadas creaciones de hilos y demasiados cambios de contexto empeoran el rendimiento, ya que con cada una de esas operaciones se pierde tiempo. No es realmente

beneficioso pasar de 3-4 hilos en una máquina de dos núcleos, ya que el reparto óptimo de las tareas sucedería con ese número de hilos (y optimizamos la paralelización), y todo lo que sea mayor implicaría perder tiempo, además de que habría un techo de paralelización, ya que hay secciones u operaciones que no se podrían paralelizar, por lo que no se podría superar cierta mejora.

Si analizamos más detenidamente los datos, parece que este aumento de tiempo del que hablamos se estabiliza, es decir, aumenta más de 4 a 8, que de 8 a 16, que de 16 a 32.

No hemos sido capaces de explicar este comportamiento, no hemos podido encontrar la causa. Ya hemos dicho que al tener 4 hilos sobre 2 procesadores/núcleos, puede que organicemos mejor y paralelicemos mejor. Hemos dicho que el paralelizar tiene un límite, y que aumentar hilos lo único que haría sería aumentar el tiempo que tarda en ejecutarse, ya que el tiempo que deberíamos ganar al paralelizar no es posible ganarlo, porque no es posible paralelizar tanto y porque nos penalizan los cambios de contexto y creación de hilos.

Es posible que el estancamiento del aumento de tiempos se deba a que el tiempo perdido por cambios de contexto y creación de hilos se ve eclipsado por la inmensa cantidad de operaciones a realizar y por el tiempo que tardan las partes no paralelizables.

Si nos fijamos ahora en los datos de cada una de las dos tecnologías utilizadas, vemos además una clara diferencia en rendimiento (ambos con las tendencias antes explicadas). En todos los casos la medida equivalente de atómicos (para el mismo número de hilos) es mejor que la de mutex, y además el estancamiento del tiempo es más apreciable en los atómicos que en los mutex.

Esto tiene su explicación en las tecnologías usadas en cada caso. Mientras que el mutex protege secciones enteras, con el número de operaciones que queramos, el atómico realiza operaciones atómicas, que nos aseguraremos que se llevan a cabo.

Esto implica que el mutex está mucho más tiempo bloqueando la ejecución del programa, mientras que con los atómicos no está realmente bloqueado nunca. Puede darse el caso de que se intente hacer varias operaciones atómicas, y solo una de ellas se hará cada vez.

Esto hace que la implementación con mutex sea más lenta (no podemos hacer push y poco a la vez), pero más sencilla, y que la implementación con atómicos sea más rápida (si podemos hacer push y pop a la vez) pero más difícil de implementar.

Ejercicio 3

Realice de nuevo el anterior apartado, esta vez ejecutando 10 ejecuciones de cada configuración. Represente gráficamente la media de tiempos de cada configuración y la desviación típica. Analice la evolución del rendimiento en cada una de las soluciones.

En este caso hemos realizado diez medidas por cada una de las que realizamos en el ejercicio anterior, los resultados son los siguientes:

tabla tiempos MUTEX STACK (µs)									
Operaciones	2 Hilos	Operaciones	4 Hilos	Operaciones	8 Hilos	Operaciones	16 Hilos	Operaciones	32 Hilos
5 Millones	3450804	2,5 Millones	3168954	1,25 Millones	3568451	625000	3782290	312500	3893147
	3195461		3167545		3604294		3809814		3896977
	3250147		3240004		3582729		3795859		3974377
	3346673		3161416		3584256		3773984		3948233
	3307876		3164311		3598637		3809551		3864757
	3469973		3124236		3698145		3788547		3896158
	3302292		3128331		3585009		3851350		3936246
	3392382		3153331		3582329		3792945		3872323
	3309980		3190007		3563053		3978153		3936777
	3264067		3187300		3568214		3804539		3875138

Tabla tiempos Atomic STACK (µs)									
Operaciones	2 Hilos	Operaciones	4 Hilos	Operaciones	8 Hilos	Operaciones	16 Hilos	Operaciones	32 Hilos
5 Millones	3028074	2,5 Millones	2863491	1,25 Millones	3294215	625000	3315296	312500	3415386
	3123879		2607849		3349927		3421793		3437736
	3209941		3170316		3201372		3368961		3475156
	2869432		3153963		3232586		3327291		3373145
	3154568		2963401		3272893		3301303		3452088
	3056654		2774119		3224549		3434004		3422889
	3036763		2728155		3295042		3410841		3453847
	3146926		2859592		3240037		3401011		3458615
	3175251		2572425		3215471		3392286		3404273
	3207728		2692951		3303096		3407347		3476578

Si calculamos las medias de dichas medidas, y las desviaciones típicas obtenemos:

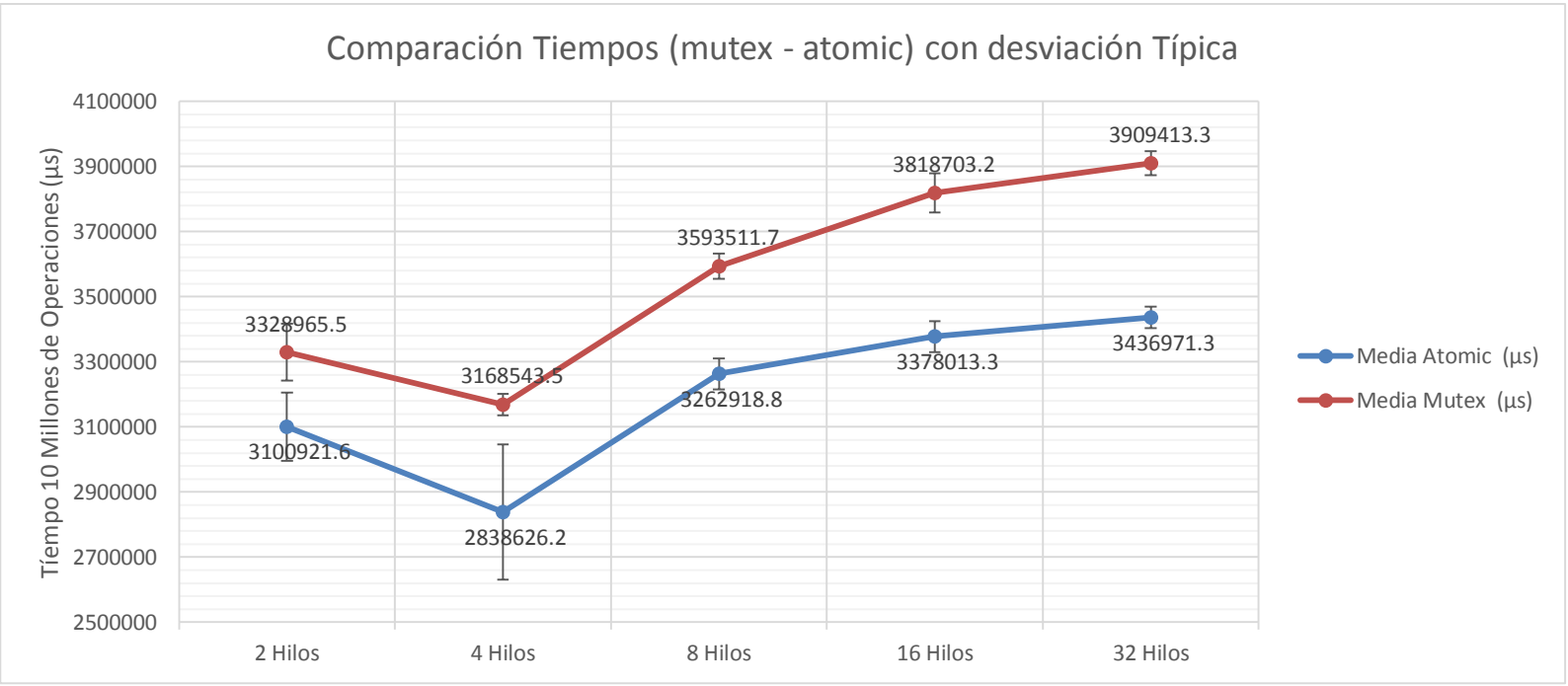
	Media Mutex (µs)
2 Hilos	3328965,5
4 Hilos	3168543,5
8 Hilos	3593511,7
16 Hilos	3818703,2
32 Hilos	3909413,3

	Desviacion tipica Mutex (µs)
2 Hilos	87449,83181
4 Hilos	32999,31292
8 Hilos	38995,27345
16 Hilos	59883,14936
32 Hilos	37021,98449

	Media Atomic (µs)
2 Hilos	3100921,6
4 Hilos	2838626,2
8 Hilos	3262918,8
16 Hilos	3378013,3
32 Hilos	3436971,3

	Desviacion tipica Atomic (µs)
2 Hilos	105044,5587
4 Hilos	207575,8277
8 Hilos	47434,95012
16 Hilos	47369,67423
32 Hilos	33025,76793

Si lo representamos en forma de gráfica obtenemos:



¿Qué solución es más estable? ¿Cuál es el impacto del incremento del número de hilos en la estabilidad de cada solución? ¿Por qué?

Como es lógico las medias tienen el mismo comportamiento de lo expuesto en el ejercicio anterior. En ambas soluciones el tiempo aumenta a medida que aumentamos el número de hilos (a partir de 4, de 2 a 4 disminuye). Además dicho aumento es cada vez menos acusado.

También vemos la diferencia entre las dos implementaciones en cuanto al rendimiento (explicada en el problema anterior). Si nos fijamos en las desviaciones típicas, vemos que son menores en la implementación con mutex. Esto puede deberse a que al ser la mayor parte el código sección protegida por mutex, hay poco que se pueda paralelizar, hay poco margen para variar los tiempos de ejecución (si todo se hace casi secuencialmente, por estar casi todo protegido, no habrá diferencias de tiempo grandes, será básicamente el tiempo de las ejecuciones “secuenciales” más una pequeña parte que si podría variar).

Con los atómicos en cambio, no hay secciones protegidas, sino que hay operaciones atómicas. Es por esto que podemos jugar mucho más con el orden de las cosas y su paralelización. Si en unas ejecuciones paralelizamos más, bajaremos el tiempo, si en unas ejecuciones tenemos muchos fallos en los compare_exchange, perderemos tiempo intentando una y otra vez lo mismo. Habrá ejecuciones que se ejecuten los compares a la primera, y otras en las que los compares fallaran la inmensa mayoría y tocara volver a hacerlos.

Cuanto más aumenten los hilos, más fácil será que un hilo ocasione que otro tenga que repetir un compare_exchange (más tarda), mientras que con los mutex, como ya hemos dicho, más hilos no implican más ejecuciones de nada, son las mismas, en orden casi secuencial, eso no variará.

Al igual que en el ejercicio anterior lo que no conseguimos explicar totalmente es lo que ocurre con 4 hilos (lo del rendimiento sí que se podría explicar si fuesen 4 núcleos y no 2). Resulta extraña la varianza tan alta en los atómicos de 4 hilos (las medidas son muy diferentes unas de otras), y lo único que se nos ha ocurrido es una combinación de la "variabilidad" de los atómicos más el estado puntual del computador.

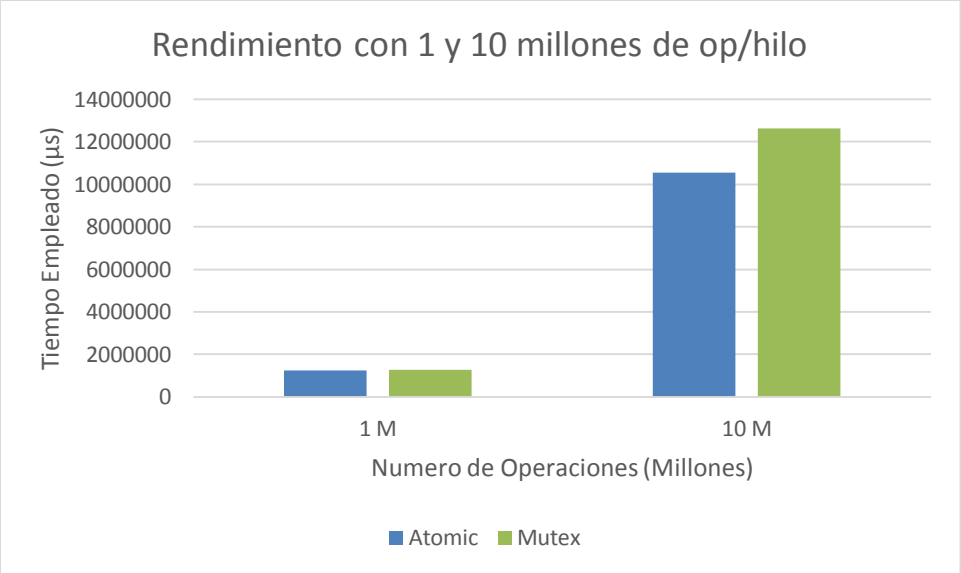
Según los datos, la más estable parece la del mutex a pocos hilos. Quizá la de atómicos se vuelva más estable con más de 32 hilos.

Ejercicio 4

Ejecute un programa con 4 hilos, modificando el número de tareas que ejecuta cada hilo. Las configuraciones serán 1000000 acciones por hilo y 10000000 acciones por hilo. Represente los tiempos obtenidos de forma gráfica y comente los resultados.

Si ahora ejecutamos con 4 hilos y 1 millón y 10 millones de acciones por hilo, obtenemos las medidas y la gráfica:

	1 M	10 M
Atomic	1253963	10538802
Mutex	1268183	12616832



¿Cuál es la evolución de rendimiento? ¿Por qué? Basándose en las diferencias explicadas anteriormente, justifique los resultados obtenidos.

Con esto vamos a analizar el comportamiento y rendimiento al aumentar el número de operaciones que realiza cada hilo. En este caso se hacen primero 4 millones de operaciones, y después 40 millones de operaciones.

Si analizamos los datos, lo primero que nos llama la atención es que no hay demasiada diferencia entre las ejecuciones con atómicos y con mutex al hacerlo con 1 millón, pero que esta diferencia crece con 10 millones. Esto es lógico ya que pequeños retardos que tiene el mutex respecto del

atómico, pueden pasar desapercibidos si se acumulan pocas veces (1 millón de operaciones por hilo), pero cuando tenemos que eso sucede diez veces (multiplicamos por diez el número de operaciones por hilo), esos retardos se acumulan y se hacen evidentes.

Lo segundo que podemos destacar es que el rendimiento respecto del número de operaciones se comporta de forma lineal. Al multiplicar por 10 el número de operaciones por hilo, el tiempo se multiplica por 8.4 en el caso de los atómicos y por 9.9 en el caso de los mutex. Con esto vemos acentuada la diferencia de rendimiento entre las dos implementaciones posibles, y vemos que los atómicos efectivamente presentan una clara mejoría en el tiempo de ejecución respecto de los mutex, que quizá con menos operaciones no resultaba tan evidente. A medida que crezcan las operaciones, la diferencia de rendimiento entre ambos de hará mayor, ya que como hemos dicho, los mutex no permiten paralelizar secciones enteras (en nuestro caso la mayor parte del código) mientras que con atómicos tenemos muchas más posibilidades.

Ejercicio 5

Estudie detenidamente la pila de datos libre de cerrojos. ¿Qué problema presenta la implementación que se proporciona?

Si usamos valgrind para analizar los posibles problemas, vemos que figuran leaks, es decir, fugas (de memoria), luego un posible problema es la fuga de memoria que se produce al no liberar la que reservamos para cada nodo. No liberamos memoria ni al hacer pop ni al terminar la ejecución de todos los hilos.

Se presentan bloques que nos muestra valgrind como perdidos definitivamente (que nadie apunta a ellos) y perdidos indefinidamente (que a pesar de que hay gente que apunta a ellos, nadie apunta a esos).

Este problema hace que se pueda producir otro: si un nodo A hace la creación del nodo y la asignación del next, y antes de su compare_exchange, primero se hace pop de la cabeza, y luego se pushea un nodo con los mismos datos que la vieja cabeza (como la que tenía A guardada). Puede suceder entonces que el nodo A no se dé cuenta de que en realidad la cabeza que tiene guardada, no es la que tiene la pila.

Ejercicio 6

Plantee, de forma teórica, una posible solución al problema detectado en el apartado anterior.

La solución al problema de las fugas de memoria sería la de borrar nodos y los datos que contienen justo antes de devolver en el pop (guardando antes lo que devolvemos en una variable auxiliar), y la de borrar la pila justo antes de salir del main.

Para resolver el problema de que un nodo no se dé cuenta de que hayan cambiado algo, basta con borrar ese algo al quitarlo, luego al arreglar el problema de la pérdida de memoria se debería arreglar el otro (teniendo cuidado de cuando borrarla).

Conclusiones

Esta práctica empezó pareciendo relativamente sencilla, y ha acabado siendo bastante más difícil de lo que pensábamos. La parte relativa al error nos resulta bastante dudosa, ya que no extraña que se den tres puntos por un simple delete de memoria.

Por eso es que no sabemos si eso era lo que se nos pedía detectar, y en caso de que no lo sea, que era lo que se supone que debíamos ver.

Otra gran frustración en no haber conseguido implementar la cola atómica totalmente sin errores, ya que hay situaciones que, de producirse, ocasionarían comportamientos no correctos con la cola.

Sería interesante que los profesores de la asignatura respondiesen a estas cuestiones y aclarasen las dudas después de la entrega de la práctica.

En cuanto a si nos ha quedado clara la diferencia entre mutex y lo atómico, se ha conseguido satisfactoriamente, también nos ha ayudado a comprender el compare_exchange, además de realizar una introducción a C++.