COSC1112/1114

Operating System Principles

Semester 2, 2019

Assignment 2 – Multithreaded Allocator

Name: Duncan Do

Student Number: s3718718

Contents Page

Solution Design

Locking and Concurrency

To ensure all running threads can access and manipulate the contents of the list without conflicts between them, locks must be put on sections of code to limit the amount of threads that can access said code section at any given time.

Read Lock

Read locks are placed around the loop in which the freed list is iterated over to find a suitable “memory chunk”. A simple solution would be to lock at the beginning of the search and unlock once the “memory chunk” had been found, for thread safety. However, this hurts the concurrency of the solution as a single thread would lock the whole list down until it finds its “memory chunk”, this would create a queuing system for the list. Thus, the threads would go one at a time searching through the list. However, the only restriction needed on a read lock is preventing any manipulation to the list as the reading occurs. Therefore, multiple readers should be able to read the list at any given time. To ensure this, a variable was created to keep track of the number of active writers at any given time (same goes for readers); furthermore, a condition was made that reading could only occur when there were no writers active. No such condition is set for readers. This logic was extracted into a function that is called before any thread is about to read; which maximises reading concurrency while keeping the threads safe from each other’s writing.

Write Lock

In the same vein, logic for determining whether a thread can write to a list was extracted into a function. A similar condition is created that forces threads to wait when the condition for writing is not met. In this case, the condition is no active readers or writers. This is because, as stated above, readers and writers cannot be concurrently active, as well as the fact multiple writers should not be active at the same point in time. Say for example, multiple writers attempt to delete the same object in memory, the second thread to delete would find that their target is gone but would still try to delete it; resulting in run time errors (double free).

Deadlocking and Signalling

The above reading and writing conditions keep the threads safe by forcing threads to wait while others carry out their tasks such as writing; avoiding threads from clashing. However, these “waiting threads” need to be “woken up” at some point, otherwise you’d run into a deadlock; where threads are still active but cannot continue due to their “continue condition” never being met.

To prevent deadlocks, a third function was created to “wake up” threads after a write was completed. The “wake up” condition would be like the initial read and write conditions. A thread can “wake up” and read when there are no writers active, and a thread can “wake up” and write when there are no readers active (No need to check for writers since a write was just completed when this function was called). This is however still not enough, that the final thread finishes writing but still tries to wake up another, non-existent thread. To cover these cases, another pair of variables were created to keep track of when readers and writers are made to wait. The “wake up” conditions also include the question if there are any reading or writing threads left waiting.

Deallocation Locks

Similar read and write locks can be placed on the deallocation code. However, the implementation simply locks the whole method down for each thread. Meaning, rather than each thread being able to search the freed list concurrently to find the “to be deleted” object, they must go sequentially. This choice was made due to time constraints and the added complexity of implementing separate locks on the deallocation code. Since allocation and deallocation occur in isolation, when working with multiple threads, the case has been covered that another thread could begin reading/writing between locks of allocation and deallocation. Concurrency was sacrificed to maintain thread safety (Correct thread safe locks on deallocation would’ve taken time not available).

Further improvements

Further concurrency could’ve been added where only one list (allocation/freed) needed to be locked at a time, allowing another thread to utilise the other list. This again, would require several safety measures to ensure no other threads slip through the cracks and start manipulating the list between locks. This was sacrificed in place of a lock over the entire reading algorithm to ensure thread safety

Experiment Choices

To experiment the time differential of multiple threads working on the same list, the initial allocation experiment was run on 3 number of threads; 1, 5 and 10. With the amount of allocations scheduled for each test, these test numbers will yield enough data to display the affect of multiple threads on the allocation process. Furthermore, in order to see how the time complexity growth of multiple threads, these thread groups were run on 2 different allocation amounts (30,000 and 60,000). This was done to demonstrate the affect of more processes on time.

Greater thread group sizes and allocation amounts were not used because of the factorial growth nature of adding more threads or allocations. For example, doubling the allocation amount to 120,000 created a test run so long that it took multiple hours to process. This could also come down to the efficiency of the solution design (Mainly due to the deallocation being non concurrent), however the experiment must be constructed around the limits of the implementation.

Results

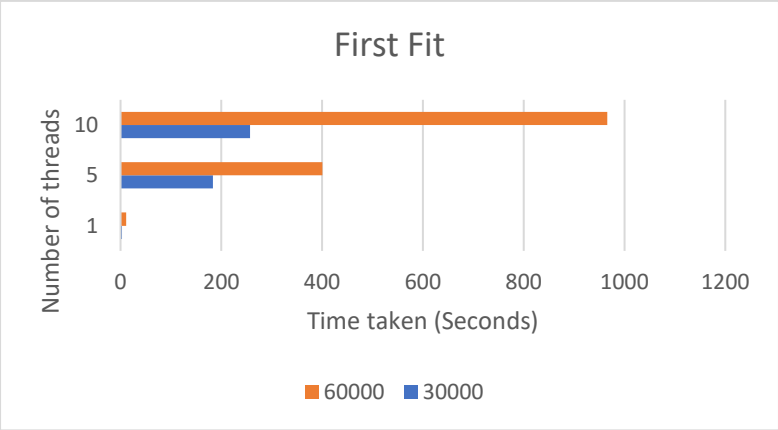
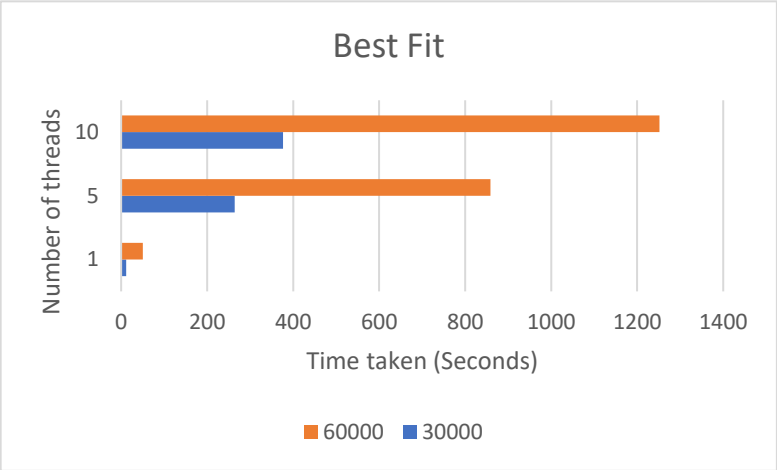


Table results in Appendix 1

In comparison to the other strategies, multithreading affects first fit's time complexity the least. This is to be expected when first fit has the earliest "early exit condition" of finding the first "memory chunk" that fits.

Table results in Appendix 2



Best fits time growth due to multithreading is like worst fit as both follow similar logical steps, however best fit does run faster on average compared to worst fit (in all three thread groups). This is due to best fits early exit condition where it finds a "memory chunk" of the exact size desired. This doesn't always happen, and best fit can run to the end of the freed list like worst fit. Hence, the similar yet better time results.

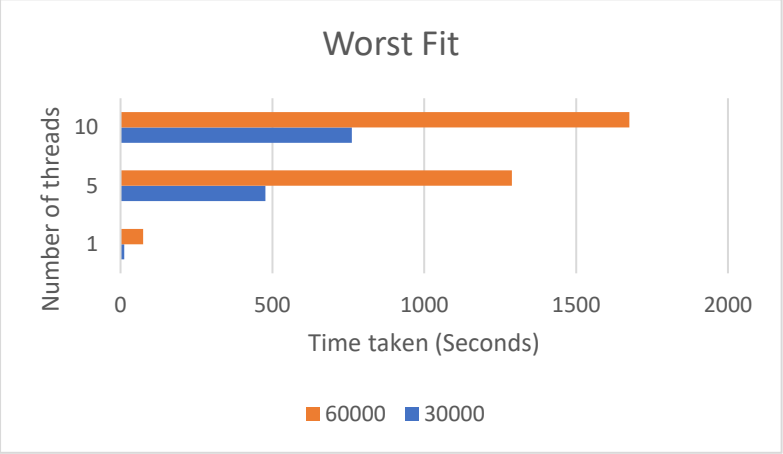


Table results in Appendix 3

From the analysis of all 3 graphs; it is highlighted that worst fit's time grows the largest when more threads are added. This is to be expected as the "longer nature" of worst fit compared to the other strategies would be compounded by multiple runs of the strategy for each thread.

Discussion

Benefit vs Detriment of multithreading the software

Overall, it is interesting how adding multiple threads to search the list would theoretically improve the time complexity of the software, it ended up hindering the time taken, and by a large factor.

This most likely can be attributed to the fact that the actual modification of list elements must be done in isolation of each other, eliminating the possibility for concurrency.

In different software or a higher quality allocation implementation, multithreading could benefit the software; both in terms of time complexity and resource utilisation.

Effect of load amounts

It is evident that when increasing the amount of allocations by a factor of 2, the time complexity raised by a factor far higher.

The solution design is most likely the culprit for the egregious time increase. Even though all the readers can search concurrently, the amount of readers does increase linearly in relation to amount of allocations, thus the time does as well. On top of that, the queue like nature of writing to the lists results in increases in the time taken per allocation. These two factors compounded with each thread allocating the same amount; drastically increases the time complexity.

Conclusion

Multithreading is a practice that utilises more of the hardwires processing power to speed up processes. This experiment resulting in disproving that for a single case. Further testing could've been done to analyse the software's behaviour under different circumstances. Such as, increasing the thread amount on a static allocation amount to analyse the multi-threading improvement on a single test size, rather than observing the time growth of multiple threads.

Appendix

Appendix 1

FIRSTFIT

Amount Allocated	Thread Count		
	1	5	10
30000	2.86	183.24	256.83
60000	11.15	400.56	965.91
120000	91.21		

Appendix 2

BESTFIT

Amount Allocated	Thread Count		
	1	5	10
30000	11.83	264.24	376.71
60000	50.2	858.57	1252.03
120000	541.27		

Appendix 3

WORSTFIT

Amount Allocated	Thread Count		
	1	5	10
30000	12.32	476.73	761.1
60000	74.32	1288.64	1674.97
120000	557.79		