



PERFORMANCE ANALYSIS OF MULTITHREADED MEMORY ALLOCATION STRATEGIES



AUTHOR: JEREMY QUINTANA

STUDENT NUMBER: S3719476

DATE: 17/10/2019



CONTENTS

Abstract	pg. 2
Introduction	pg. 2
Methodology	pg. 4
Results	pg. 5
Discussion	pg. 9
Conclusion	pg. 10
Appendix	pg. 11

ABSTRACT

This report will follow ahead from the previous report "Performance Analysis of Memory Allocation Strategies". In particular this report will delve into the effect of turning the previously experimented allocation strategies to multithreaded programs. Multithreading is the term of turning a sequential program into one that may run in parallel along with other parts of the program creating concurrency as parallel segments progress together. An experiment will be conducted identifying the overall performance in terms of speed and fragmentation of memory. Through these results we will be able to determine the effects of multithreading on the different allocation strategies of first, best and worst fit, and also observe the effects of Amdahl's law on the program duration in relation to concurrency.

INTRODUCTION

As we had previously covered the critical resource of memory on the previous report, we will now focus primarily on speed and resource utilisation within a given program. As processes are given resources as efficiently as possible allowing them to utilise the CPU, the next step is to create a more efficient process that essentially ensures maximum CPU and resource utilisation, reducing the amount of time dedicated by the CPU to a given process.

Multi-threading is used within a program to ensure that all resources are being utilised as effectively as possible to reduce time dedicated to the given process. At its essence, multi-threading splits a given process allowing multiple parts of the code to be executed out of sequential order. Overall, the use of threads increases the responsiveness of a system as the process is no longer working sequentially, hence removing potential waits and decreasing wait time of the process in queue. This responsivity is due to concurrency, as the system will support more than one task enabling the process to potentially switch when waiting on a thread to ensure the CPU is utilised at all times.

Multi-threading is also extremely scalable when implemented with multicore processor as a thread can now execute at the same moment as another thread which is termed parallelism, meanwhile a single core processor will need to context switch between threads as only one thread can run at one time. Parallelism speeds up the process although according to **Amdahl's law** there is a maximum speed up for a program relative to the amount of cores. Overall, multi-threading encourages resource sharing improving the overall economy of the process by removing the need to allocate more memory to create a new process.

Parallelism and concurrency have similar meanings and similar effects when viewed but are subtly different. Parallelism refers to the ability to run more than one task simultaneously and is possible, as mentioned before, through multicore systems. Meanwhile concurrency is the ability to support more than one task making progress, with the CPU switching between given tasks of the process. These two can be confused due to the speed tasks are context switched in non-parallel concurrent systems, enabling it to appear parallel as each task switches and runs too fast to notice.

A context switch was previously mentioned and is essentially the switch between a running thread on the CPU with another thread waiting in queue. Context switching requires the storage of information of the current thread and acquiring information on the next thread to run although this is still relatively cheap. Although, while cheap, it has the potential to accumulate with a large number of switches due to their being too many threads running and waiting within the queue.

Overall, issues can arise with context switching such as starvation of a thread meaning that it may never run due to other threads taking priority. Although, this is entirely dependant on whether the switch is pre-emptive or non-preemptive, in which a thread relinquishing utilisation is determined by itself or by another process. Additionally, the scheduling algorithm used for determining the next thread to run can determine whether starvation occurs and when a thread is run relative to the entire process.

As multi-threading creates multiple tasks that hold a shared resource, a process needs to be wary when creating new threads as this resource may be altered by one thread and another thread may alter it essentially simultaneously creating an incorrect result, this is known as a race condition. To prevent this race condition, public locks that can be viewed by all threads are required to protect the resource from being altered by a thread when another thread is using it.

One of the most recognised locks are the read write locks. A write lock is performed when the shared resource is to be edited in any way and must ensure that no other thread holds any form of access on the shared resource essentially serialising the access. With a read lock, it allows any thread to read from the shared resource and must ensure that there is no current thread holding a write lock over the data.

When implementing locks though it is possible that deadlocks can occur which is essentially one thread waiting for another thread to relinquish their lock over a shared resource whom never relinquishes it for an arbitrary reason. There is a criterion that enables these deadlocks to occur, and if one criterion is broken, then a deadlock may not occur.

Locks enable multi-threading and hence greater CPU utilisation, although it comes with risks to the process functionality and large overhead if not used in an effective and efficient manner.

METHODOLOGY

The analysis will delve into the performance regarding time and overall performance difference from the initial single threaded version of the following strategies:

- First fit, using the first found chunk of sufficient size
- Best fit, searching the entire list of free chunks to find the chunk that is the closest to the size needed
- Worst fit, searching the entire list of free chunks to find the chunk that contains the most difference between the needed size and size of the chunk

Using the previous C++ program created to test the strategies, the program will be adapted to ensure that it is completely thread safe. In particular the allocation and deallocation functions will maintain the core functionality but must include read and write locks and unlocks in the appropriate areas where the function is accessing the shared resource, allocated list and free list.

These read and write locks will be implemented to ensure that no deadlocks occur. It will be implemented on a new Locker class which will hold public functions for read and write locks and unlocks. These locks will be determined by class variables that hold the current state of how threads are accessing the data. Read locks will ensure that it may only pass the condition when no other writers are writing or in queue. And the write lock will ensure that it only passes its condition if there are no threads reading, or that there is no other thread writing. These locks will be implemented using the pthread library in Locker.cpp which will prevent race conditions on the variables of the Locker class by sequentialising the access to the Locker class functions.

In terms of the experiment, it will be an adapted version of the non-threaded performance analysis experiment of the memory allocation strategies. The experiment will be separated into iterations with each iteration having 1000 sequences of an allocation followed by a deallocation, running on a passed in number of threads. We will differ from the original experiment to mimic the real-world use of a memory allocator which may allocate and deallocate simultaneously on separate threads rather than just a series of allocations followed by a series of deallocations.

Using this experiment, we will record how threads affect the speed of the test having a constant of 10 iterations, tested with 1-10 threads on each fit algorithm three times to attain an average.

We will also determine if this multithreaded version using the optimal thread count will affect the speed of each fit algorithm over iterations incrementing by 10 starting from 10-50.

Lastly, we will determine if there are any changes of fragmentation for each fit differing from observations in the single threaded version of the experiment using iterations 10, 50, 100 with the optimal thread amount being used as a constant.

RESULTS

Fig. 1: First Fit Time(NS) over threads 1-10 and 20

	1	2	3	4	5	6	7	8	9	10	20
Test 1	1.27E+09	9.74E+08	8.87E+08	8.24E+08	9.85E+08	1.058E+09	1.1E+09	1.14E+09	1.22E+09	1.23E+09	1.39E+09
Test 2	1.24E+09	9.68E+08	9.05E+08	8.34E+08	9.81E+08	1.071E+09	1.13E+09	1.16E+09	1.24E+09	1.23E+09	1.37E+09
Test 3	1.25E+09	9.72E+08	8.85E+08	8.21E+08	9.73E+08	1.079E+09	1.11E+09	1.16E+09	1.22E+09	1.25E+09	1.41E+09
Average	1.25E+09	9.71E+08	8.92E+08	8.26E+08	9.8E+08	1.069E+09	1.11E+09	1.15E+09	1.22E+09	1.24E+09	1.39E+09

Fig. 2: Best Fit Time(NS) over threads 1-10 and 20

	1	2	3	4	5	6	7	8	9	10	20
Test 1	1.24E+09	1.12E+09	8.8E+08	8.19E+08	1.01E+09	1.035E+09	1.11E+09	1.17E+09	1.23E+09	1.26E+09	1.37E+09
Test 2	1.25E+09	1.15E+09	8.64E+08	8.33E+08	9.99E+08	1.024E+09	1.15E+09	1.16E+09	1.23E+09	1.26E+09	1.39E+09
Test 3	1.27E+09	9.63E+08	8.88E+08	8.35E+08	9.82E+08	1.043E+09	1.13E+09	1.2E+09	1.24E+09	1.27E+09	1.38E+09
Average	1.26E+09	1.08E+09	8.77E+08	8.29E+08	9.96E+08	1.034E+09	1.13E+09	1.18E+09	1.23E+09	1.27E+09	1.38E+09

Fig. 3: Worst Fit Time(NS) over threads 1-10 and 20

	1	2	3	4	5	6	7	8	9	10	20
Test 1	1.28E+09	9.93E+08	9.08E+08	8.35E+08	9.89E+08	1.053E+09	1.1E+09	1.17E+09	1.21E+09	1.24E+09	1.46E+09
Test 2	1.28E+09	9.84E+08	9.03E+08	8.53E+08	1.05E+09	1.053E+09	1.12E+09	1.19E+09	1.2E+09	1.23E+09	1.38E+09
Test 3	1.29E+09	9.93E+08	9E+08	8.52E+08	1.06E+09	1.066E+09	1.12E+09	1.18E+09	1.24E+09	1.26E+09	1.39E+09
Average	1.28E+09	9.9E+08	9.03E+08	8.47E+08	1.03E+09	1.057E+09	1.11E+09	1.18E+09	1.21E+09	1.24E+09	1.41E+09

Fig. 4: All fits average values

FITS	1	2	3	4	5	6	7	8	9	10	20
First Fit	1.25E+09	9.71E+08	8.92E+08	8.26E+08	9.8E+08	1.069E+09	1.11E+09	1.15E+09	1.22E+09	1.24E+09	1.39E+09
Best Fit	1.26E+09	1.08E+09	8.77E+08	8.29E+08	9.96E+08	1.034E+09	1.13E+09	1.18E+09	1.23E+09	1.27E+09	1.38E+09
Worst Fit	1.28E+09	9.9E+08	9.03E+08	8.47E+08	1.03E+09	1.057E+09	1.11E+09	1.18E+09	1.21E+09	1.24E+09	1.41E+09

Fig. 5: Worst fit visualisation of time over threads

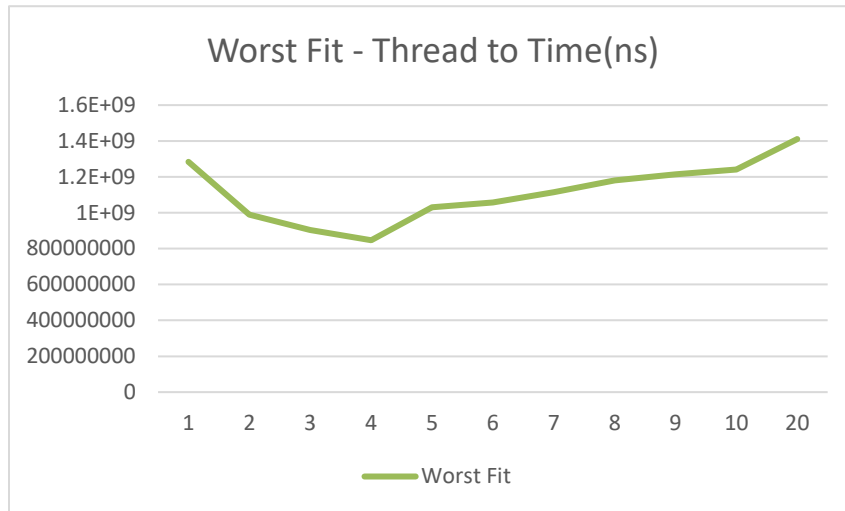


Fig. 6: Best Fit visualisation of time over threads

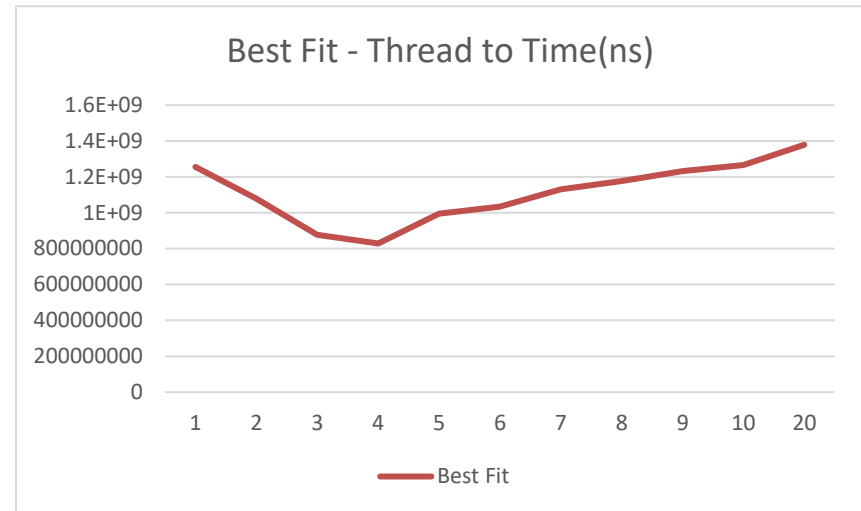


Fig. 7: Worst fit visualisation of time over threads

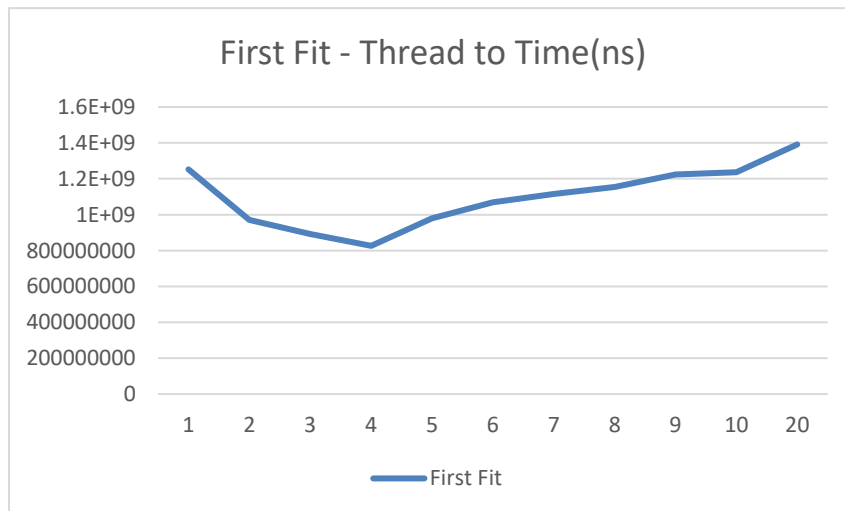


Fig. 8: All fits visualisation of time over threads

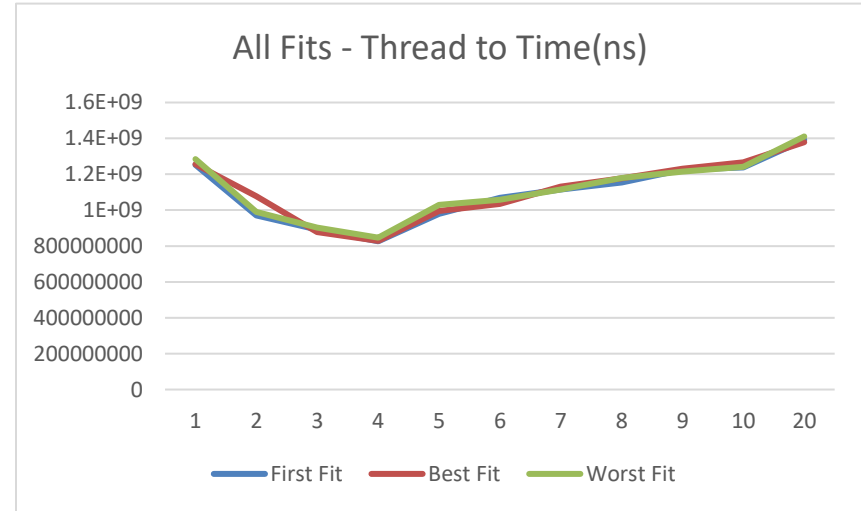


Fig. 8: First Fit Time(NS) over iterations 10-50 and 100 with 4 threads

	10	20	30	40	50	100
Test 1	8.49E+08	1.97E+09	4.15E+09	5.25E+09	8.26E+09	2.352E+10
Test 2	8.59E+08	1.96E+09	3.45E+09	5.24E+09	7.39E+09	2.253E+10
Test 3	8.18E+08	2.04E+09	3.4E+09	5.24E+09	7.44E+09	2.512E+10
Average	8.42E+08	1.99E+09	3.67E+09	5.24E+09	7.7E+09	2.373E+10

Fig. 9: Best Fit Time(NS) over iterations 10-50 and 100 with 4 threads

	10	20	30	40	50	100
Test 1	8.56E+08	1.94E+09	3.43E+09	5.13E+09	7.32E+09	2.306E+10
Test 2	8.14E+08	1.96E+09	3.38E+09	5.16E+09	7.41E+09	2.183E+10
Test 3	8.13E+08	2.02E+09	3.49E+09	5.21E+09	7.35E+09	2.192E+10
Average	8.28E+08	1.97E+09	3.43E+09	5.17E+09	7.36E+09	2.227E+10

Fig. 10: Worst Fit Time(NS) over iterations 10-50 and 100 with 4 threads

	10	20	30	40	50	100
Test 1	8.57E+08	1.99E+09	3.47E+09	5.34E+09	7.62E+09	2.446E+10
Test 2	8.25E+08	2.03E+09	3.49E+09	5.33E+09	7.7E+09	2.603E+10
Test 3	8.66E+08	1.97E+09	4.32E+09	5.49E+09	7.64E+09	2.415E+10
Average	8.49E+08	1.99E+09	3.76E+09	5.38E+09	7.65E+09	2.488E+10

Fig. 11: All Fits Avg Time(NS) over iterations 10-50 and 100 with 4 threads

Time (ns) Threads 4	10	20	30	40	50	100
First Fit	8.42E+08	1.99E+09	3.67E+09	5.24E+09	7.7E+09	2.373E+10
Best Fit	8.28E+08	1.97E+09	3.43E+09	5.17E+09	7.36E+09	2.227E+10
Worst Fit	8.49E+08	1.99E+09	3.76E+09	5.38E+09	7.65E+09	2.488E+10

Fig. 12: All Fits Visualisation of Iterations to Time(NS)

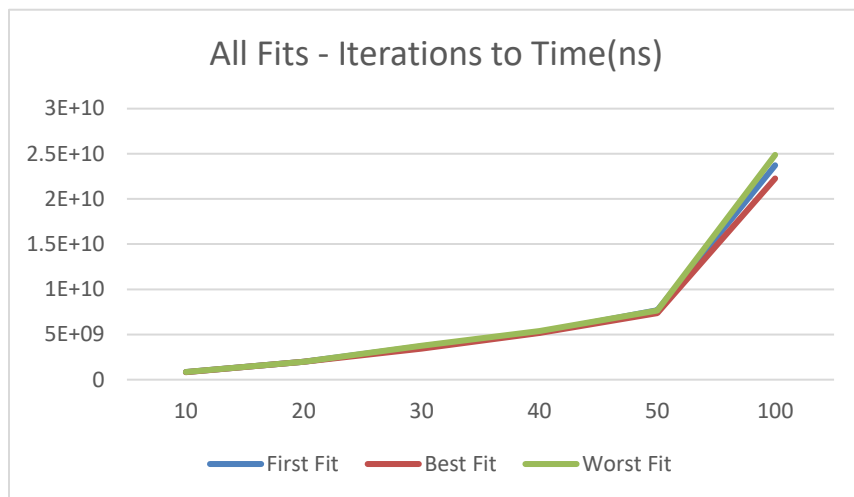


Fig. 13: All Fits Visualisation of Iterations to Fragmentation

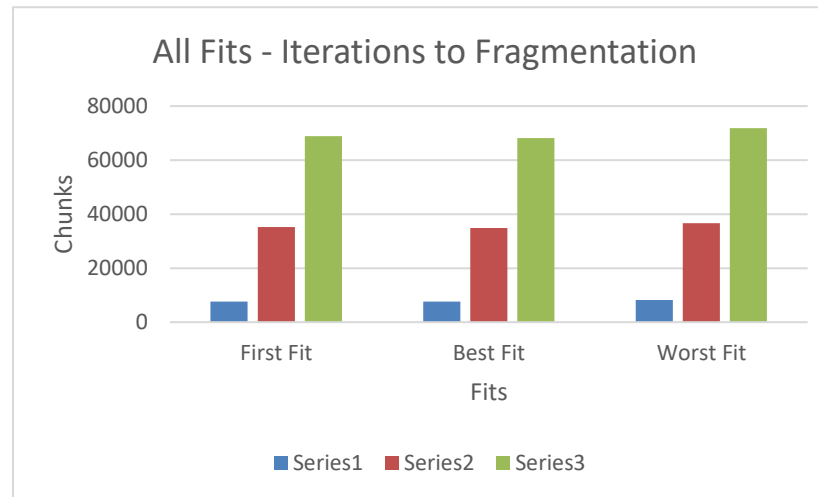


Fig. 14: All Fits data of Fragmentation over Iterations 10, 50, 100

Chunks	10	50	100
First Fit	7655	35200	68816
Best Fit	7633	34894	68192
Worst Fit	8233	36590	71849

DISCUSSION

Through the allocation and deallocation functions, testing was conducted to delve into the effects of multi-threading for each fit algorithm. This was conducted to determine the best algorithm to use for a computing system that allows multi-threading, and also how multi-threaded results differ for each algorithm from the non multi-threaded implementation.

It was expected that Amdahl's law would apply to each of the fit algorithms after multi-threading as it is a fundamental law that portrays the overall time efficiency of multi-threading processes. As we can see from fig. 5, 6 and 7 the experiment runtime decreases over a given number of threads and after the optimal thread amount gradually increases the more threads used. Using this information we are able to justify that Amdahl's law is applied to our multithreaded process due the decrease then gradual increase in experiment runtime over an increasing number of threads as this would convey that there is an optimum amount of threads to use for the given algorithms before runtime begins to slow down.

Looking at Fig. 8 we can see that there is marginal difference between how each fit algorithm is affected after applying multi-threading. It is to be noted that the number of threads that are most optimal for each of the fits would be **four** as it is the turning point, before constant escalation of experiment duration. This result may be traced back to Amdahl's law, where the optimal thread to be found is determined by the amount of parallelism within a process. As each fit algorithm holds no independent parallel components within the function, the number of parallel components is determined by the allocation and deallocation functions alone making the amount of parallelism for each fit equal. As the parallelism is equal in each fit, results that display speedup over amount of threads such as Fig. 5, 6 and 7 will yield marginal differences.

Within the results of Fig. 12 it held results of experiment duration differences on different fit strategies over escalating iterations. With these results we can see there are marginal differences between each fit strategy similar to what was found in the non-multithreaded implementation. To summarise what was found in the non-multithreaded implementation, it was found that each algorithm matched similarly in terms of time efficiency despite the first fit implementation being expected to be faster.

Fig. 12 displays the overall speed after an optimal amount of threads (four) and comparing this towards results in the non-multithreaded implementation it can be seen that the multithreaded implementation is only faster under a large enough number of allocations. This may be due to the amount of overhead required to run the lock functions that ensures the implementation is thread safe. As the amount of allocations increases the CPU utilisation compared to the non-

multithreaded implementation becomes larger, resulting in the locking overhead becoming marginal making the multithreaded implementation, faster on larger number of allocations.

What yields particularly interesting results is Fig. 13 which is a bar graph displaying the amount of fragmentation over a series of iterations. These results convey that despite the fit algorithm, it will hold the same amount of fragmentation when having the same number of allocations. This greatly differs from non-multithreaded results which displayed exponentially increasing difference between each algorithm with an increasing amount of allocations. It was to be expected that we would find similar results with the multi-threaded implementation although this is not the case.

A possible reason to this skew may be the alteration to the experiment to simulate for real world use of memory allocators. This alteration enforced a sequential allocation and deallocation per thread rather than a series of allocations followed by a series of deallocations. As a result, given threads may be able to reuse a chunk that had just been deallocated, rather than a series of allocations that share a set number of chunks. Overall, we are unable to prove that multi-threading affects fragmentation between different fit algorithms due to this discrepancy between the tests. Although, it can be deduced that in a real-world scenario where multiple processes are constantly allocating a deallocating memory, fragmentation between each algorithm is little concern as they all hold similar end results.

Overall, with the results we are able to conclude that under a large enough number of allocations all fits in the multithreaded implementation will run more efficiently than its non-multithreaded counterpart. We can also see how multi-threading does not affect each fit algorithm independently showing the same optimal thread value.

CONCLUSION

This report was intended to analyse how multithreading a memory allocator changes the overall performance of the fit strategies of first, best and worst. When analysing these strategies, it was intended to determine whether multi-threading this particular process would improve time efficiency and CPU utilisation and if it does, how do the amount of threads affect this. It was also to be determined if a multi-threaded implementation would alter fragmentation results from the non-multithreaded implementation.

Overall using the results, it can be determined that turning the memory allocator into a multi-threaded process will improve time and CPU utilisation over a "large" number of allocations. It was also found that Amdahl's law could be seen within the implementation of the strategies, displaying that four is the optimal amount of threads to use for the implemented memory

allocator. It was also noted that all fit strategies held the same optimal thread value, and it was concluded that this was due to their being no change in the amount of parallel and sequential components between each fit strategy implementation.

On an interesting contradictory note, the multithreaded implementation held results that shows each fit strategy having the same fragmentation for a given number of allocations. This differs greatly from the initial non-multithreaded implementation as those results display the exponential decrease between fit strategies over an increasing number of allocations. Nothing can be concluded from this evidence though as the experiment had been slightly altered to simulate more realistic usage of a memory allocator.

For future tests we recommend that we hold the experimental process of performing a series of allocation followed by a series of deallocations for an iteration. This is due to each allocation having a larger scope of chunks to select from in the current experiment as opposed to the original experiment which forced a series of allocation to share a set number of allocations. By having a consistent test between the two experiments we can hold have a justified conclusion on whether multithreading affects fragmentation.

It is recommended in terms of the functionality of the memory allocator, that it only runs in a multithreaded format given a large enough number of allocations. This is to avoid the overhead of the locking mechanisms that forces the multi-threaded implementation to run slower on a lower number of allocations.

Currently with these findings, we can finalise that multi-threading using four threads will improve CPU utilisation and runtime reduction given a large enough number of allocations. With a lower number of allocations regardless of thread amount, forcing the multithreaded implementation to run slower than a non-multithreaded implementation.

APPENDIX

Locker.cpp – contains all read and write locks and unlocks