Operating Systems Principles: Segmented Memory Allocator

COSC1114 A1 REPORT

Chuying Zhang S3718057
RMIT University

Table of Contents

Contents

Memory Allocator Design

Linked List data structure and implementation

The underlying data structure establishing the system of managing conceptual memory blocks in the memory allocator is a *doubly-ended* linked list. The C++ container std::list under the header file <list> was a suitable candidate for this context with the required member functions to emulate the semantics of a memory allocator.

A block of memory is represented as a struct of type *block* consisting of:

```
struct block {

        size_t size;
        void* memoryAddress;
        bool isFree;

};
```

Inclusion of the *size* field is required during the allocation process to determine if the size requested to the memory allocator is within range of the space currently available in the block of memory.
During deallocation, the field *memoryAddress* enables a currently allocated block to be uniquely identified in the allocated list and subsequently transferred to the free list. The boolean field *isFree* provides additional validation upon deallocation that the block can indeed be freed.

The *SegmentedMemoryAllocator.h* file outlines the functions involved in memory block generation, allocation and deallocation, and *SegmentedMemoryAllocator.cpp* provides the definitions of these functions.

As the operations involved in generating a new block of memory are common throughout several conditions, (including when the allocated list is initially empty or when a remainder block requires construction after splitting from a larger block) the process of generating a new block is suitable to be written as a separate, private function. This function, *createBlock(size_t size)* is called within all implementations of the allocation algorithms; *First Fit*, *Best Fit* and *Worst Fit*.

The *push_back()* member function of *std::list* is used to insert newly constructed memory blocks into the allocated or deallocated list. This aligns with the process of adding a new node to the linked list data structure; with insertion occurring at the end of the list. Another *std::list* member function, *splice()* transfers the block at the current iterator position (when searching the allocated list for the block to free), to the deallocated list. In effect, the memory blocks are ***transferred, not copied*** from one list to the other.

The function signature of the *alloc(size_t size, int strategy)* function contains an additional formal parameter of type int to specify the allocation approach to be applied. This enables the *alloc* function to be called universally for all three allocation implementations and does not require every strategy to be written as a distinct function. In particular, the code involved in traversing the list using an iterator is common between all three approaches.

Experimental Design

Analysing the performance of the First, Best and Worst strategies

In order to analyse the efficiency of the memory allocator implementation, a series of timed experiments were executed on the RMIT Core teaching UNIX server with application of the First, Best and Worst schemes respectively. Each sub-experiment utilised a specific load capacity and data type in addition to the allocation schema.

These factors became the independent variables against the amount of memory space occupied by the process and the process's completion time.
The data types int, double and char* were specifically selected as the types to be varied in each sub-experiment as they have differences in bit width on the Linux 64-bit application: 4 and 8 bytes respectively. A further aim of the experiment was to emulate situations in the real world where the memory allocator would be included in programs to store variables. It can be deduced that the int*, double* and char* types are the among most commonly used data types in these scenarios; and as such this provides a reasonable indication of the allocator's expected performance in practical applications.

Statistics were gathered using the function getrusage for physical memory space held and the chrono header file to record the current interval of time taken during a stage in the test.

Experimental Setup

A ***set*** of tests consists of the structure:
1. <Load Capacity>, First Fit, <Data Type> – 3 repeat runs
2. <Load Capacity>, Best Fit, <Data Type> – 3 repeat runs
3. <Load Capacity>, Worst Fit, <Data Type> – 3 repeat runs

Load Capacity

High Load = 50,000 allocations and deallocations
Free memory-block generation, allocation and deallocation are executed for each test in this respective sequence with the *High Load* capacity. For every interval of 5000 iterations during each stage of the test, the current RRS and time in milliseconds were recorded.

Medium Load **=** 20,000 allocations and deallocations
Free memory-block generation, allocation and deallocation are executed for each test in this respective sequence with the *Medium Load* capacity. For every interval of 2000 iterations during each stage of the test, the current RRS and time in milliseconds were recorded.

Low Load **=** 7,000 allocations and deallocations
Free memory-block generation, allocation and deallocation are executed for each test in this respective sequence with the *Medium Load* capacity. For every interval of 700 iterations during each stage of the test, the current RRS and time in milliseconds were recorded.

In total, there were ***9 distinct sets of tests*** in the experiment, and each set was a sub-experiment to assess the independent variable: the memory allocation schema.

For each set, the factors <Load Capacity> and <Data Type> were kept constant in order to observe the outcome of switching allocation strategies alone, on the memory space used and the total time taken for the test to run to completion. Once all sets of tests were completed, the load capacity and data type factors could then be compared across the sets to recognise patterns. ***e.g.***

Set 1	*Set 2*
1. High Load, *First Fit*, int – 3 repeat runs	1. Low Load, *First Fit,* double – 3 repeat runs
2. High Load, *Best Fit,* int – 3 repeat runs	2. Low Load, *Best Fit,* double – 3 repeat runs
3. High Load, *Worst Fit,* int – 3 repeat runs	3. Low Load, *Worst Fit,* double – 3 repeat runs

The differences in the memory used and time taken can be attributed to only varying the allocation strategy in each example set above. Then, the effect of running the test using a Low Load or a double type can also be compared across the two example sets.

Using Vectors

Dynamic vectors were used as the data structure to store the variables created in the tests. This choice was for the convenience of helper methods such as *size()* to track the length of the data collection instead of allocating further space and operations to determine the size manually which may interfere with test results.

Free Block Generation

A function named *requestFreeBlock(size_t size)* in the implementation of the Segmented Memory Allocator is written for the convenience of running tests by generating variable-sized 'free' memory blocks without allocating them first and then subsequently deallocating them. Hence, allocations can be immediately made for a data type with existing free blocks on starting the test. This function is used solely for experimentation purposes.

The free blocks initially generated range from 5 – 34 bytes in size and are determined from the loop iteration's index.

Repeating Tests

A set consists of three distinct tests using the three allocation strategies and each test was repeated three times to increase the reliability of results. Due to the experiment being conducted on the Core teaching server, it's possible for the shared bandwidth to cause slight deviations in the results. Hence, repeating the same test multiple times accounts for this factor in the experiment and increases the certainty that the results obtained are correct.

Strings and Input Data

Experiments which analysed the behaviour of strings used the data type char* (more specifically char* *const*) instead of the container std::string. All strings in the experiment are read into the program from an input text file called *Collins Scrabble Words (2015).txt* containing a newline delimited list of immutable strings.

As the length of strings are never modified in this experiment, there is no need for library functions to process the strings, and as such the char* data type is best suited in this purpose for simplicity. Utilising the same dataset of strings in all tests involving the string data type is required to maintain consistency in the results as opposed to randomising the length of strings; which is subject to variation in each test.

Recording the RSS against Time

Establishing a relationship between the RSS (Resident Set Size) and running time forms a reasonable understanding of the rate of memory consumption in the heap during the execution of each test involving a different load and data type. The choice of using this statistic allows the three allocation strategies to be graded on efficiency; by the degree of memory usage and total time taken. The RSS can also be compared with the final heap size to estimate the extent of fragmentation in a test (if any).

Experimental Data

Valgrind, getrusage and Chrono time statistics

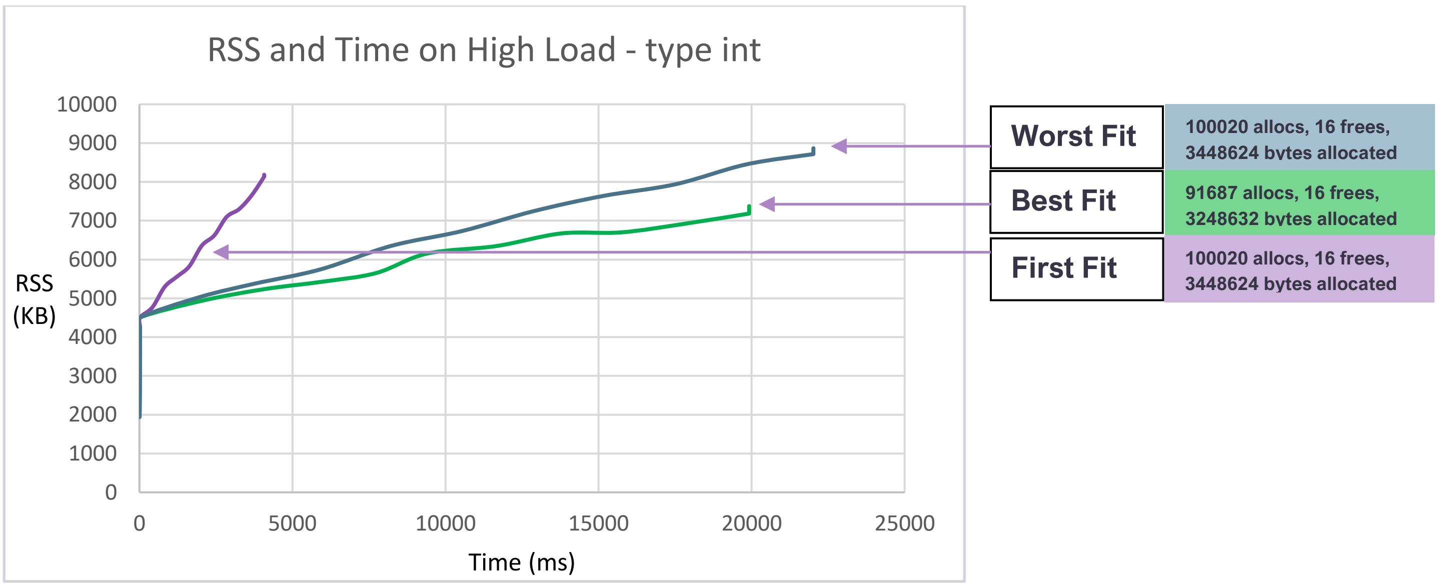
For each test in a set as defined previously, the recorded memory use and total time taken for the test to run to completion was the basis for judging the efficiency of the memory allocation algorithm. Another aspect contributing to an algorithm's definition of effectiveness was the extent of memory fragmentation.

As such, the final RSS reported by the operating system was also compared with the actual heap size reported by *Valgrind* to gain a reasonable insight to this degree of fragmentation. The efficiency of each allocation schema would also take into account whether the outcomes observed in a test set are also reflective of those in other test sets using a different data type or load capacity.

Memory Allocation Experiment Results

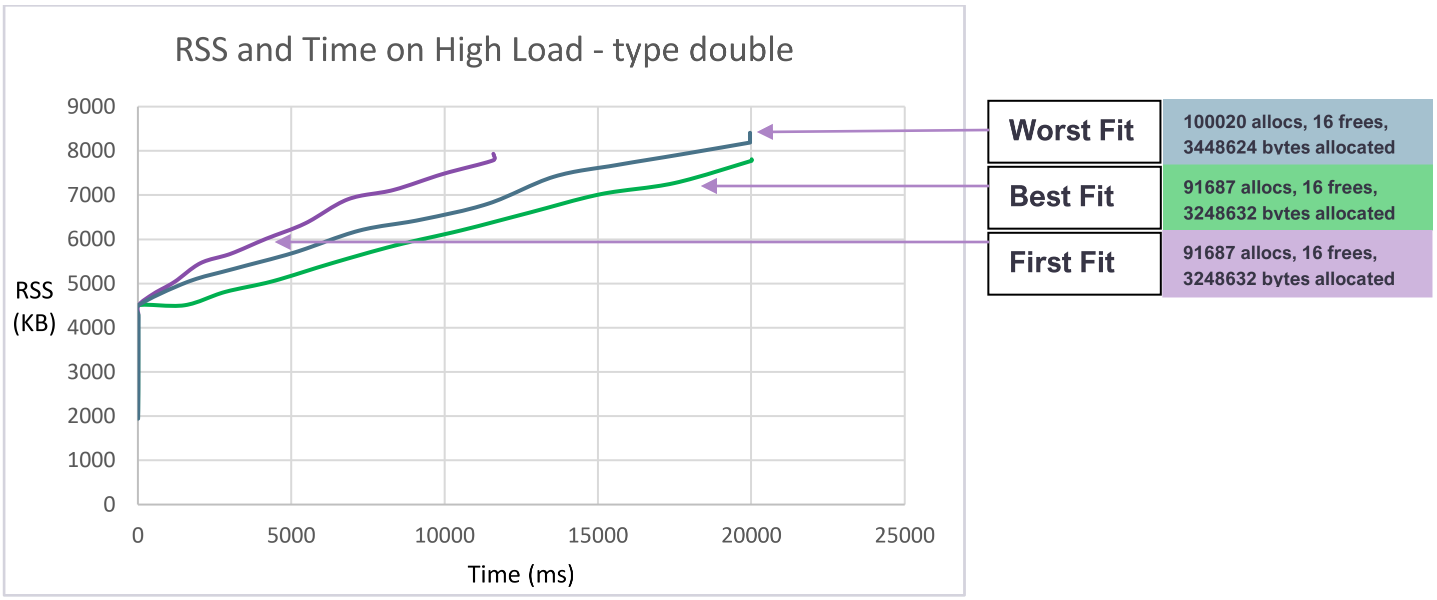
Set 1

Test number	Test name	Average total RSS (KB) across three runs	Average total time (ms) across three runs
1	High Load, *First Fit*, int	8188	4145
2	High Load, *Best Fit*, int	7376	20060
3	High Load, *Worst Fit*, int	8868	21860



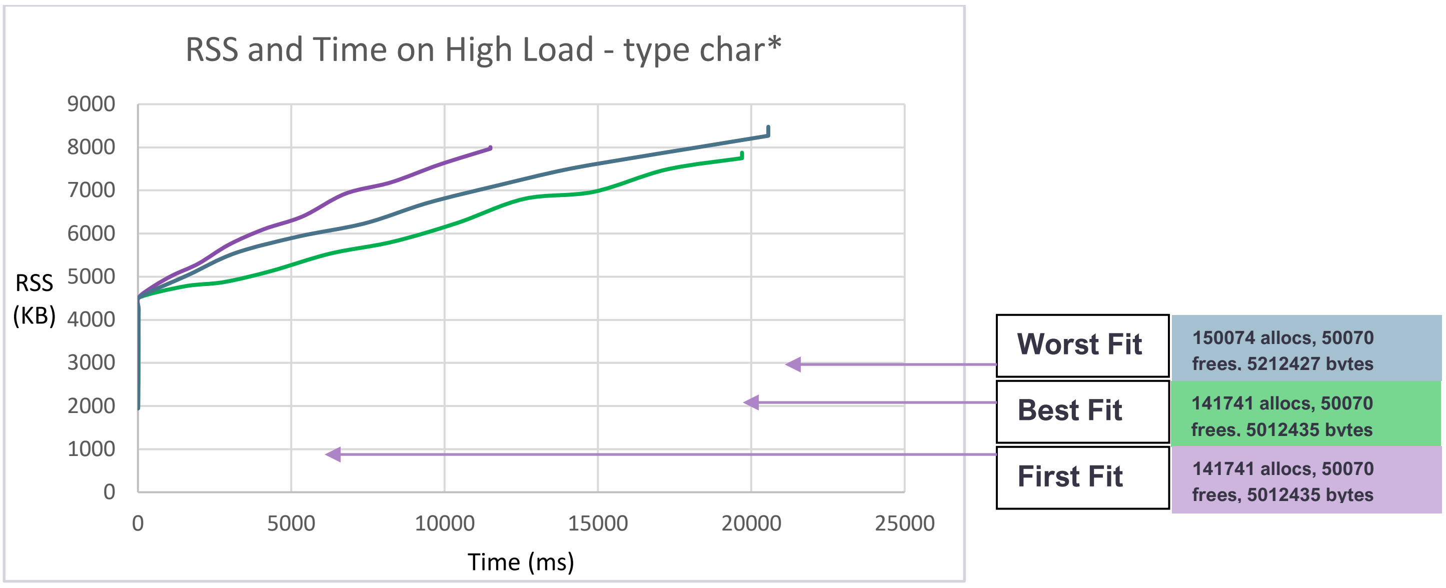
Set 2

Test number	Test name	Average total RSS (KB) across three runs	Average total time (ms) across three runs
1	High Load, *First Fit*, double	7932	11496
2	High Load, *Best Fit*, double	7808	19998
3	High Load, *Worst Fit*, double	8408	20146



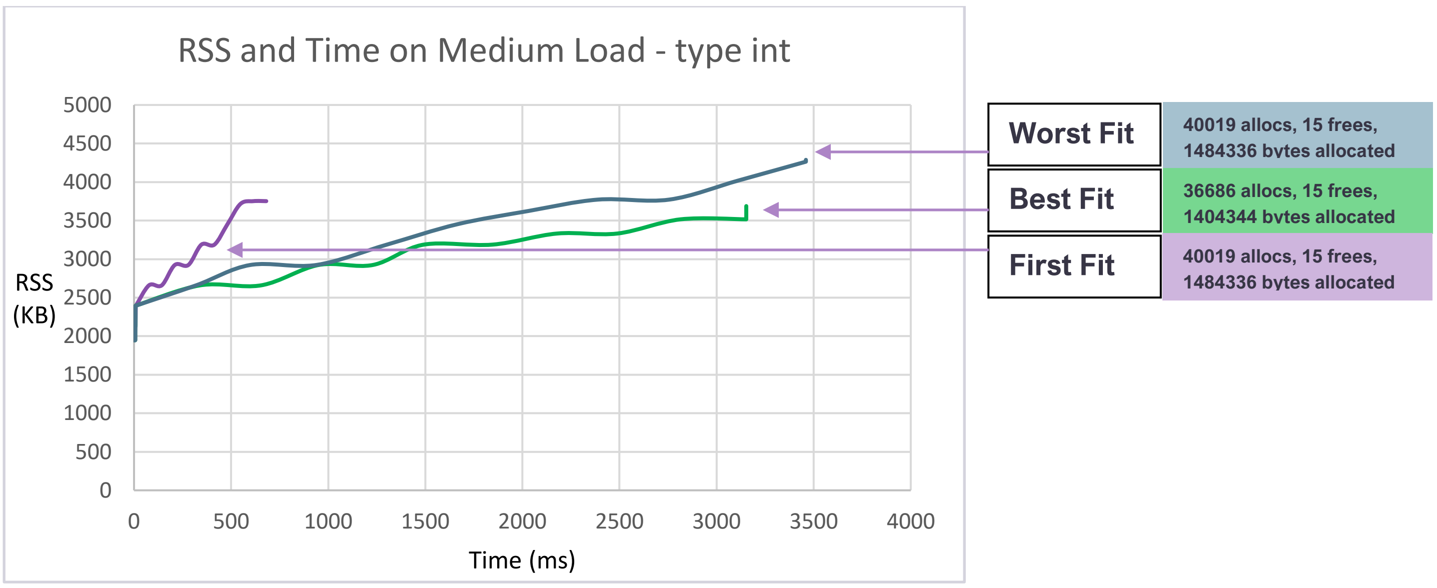
Set 3

Test number	Test name	Average total RSS (KB) across three runs	Average total time (ms) across three runs
1	High Load, *First Fit*, char*	8004	11522
2	High Load, *Best Fit*, char*	7876	19936
3	High Load, *Worst Fit*, char*	8476	20552



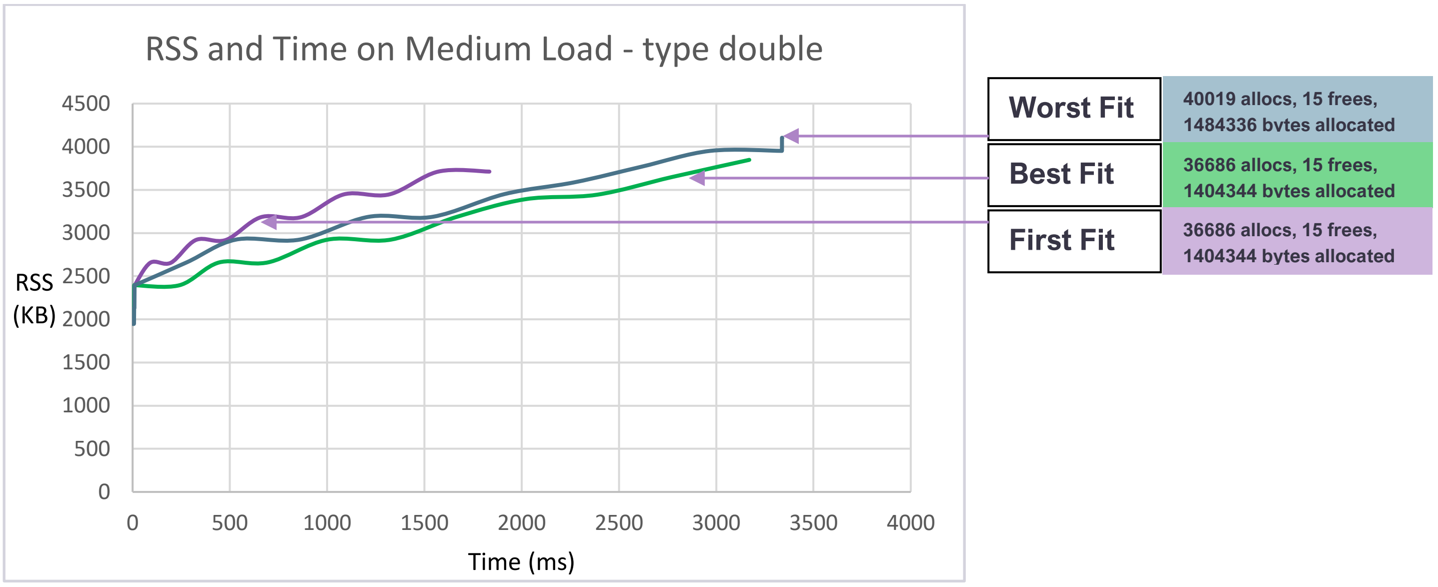
Set 4

Test number	Test name	Average total RSS (KB) across three runs	Average total time (ms) across three runs
1	Medium Load, *First Fit*, int	3752	677
2	Medium Load, *Best Fit*, int	3688	3178
3	Medium Load, *Worst Fit*, int	4288	3517



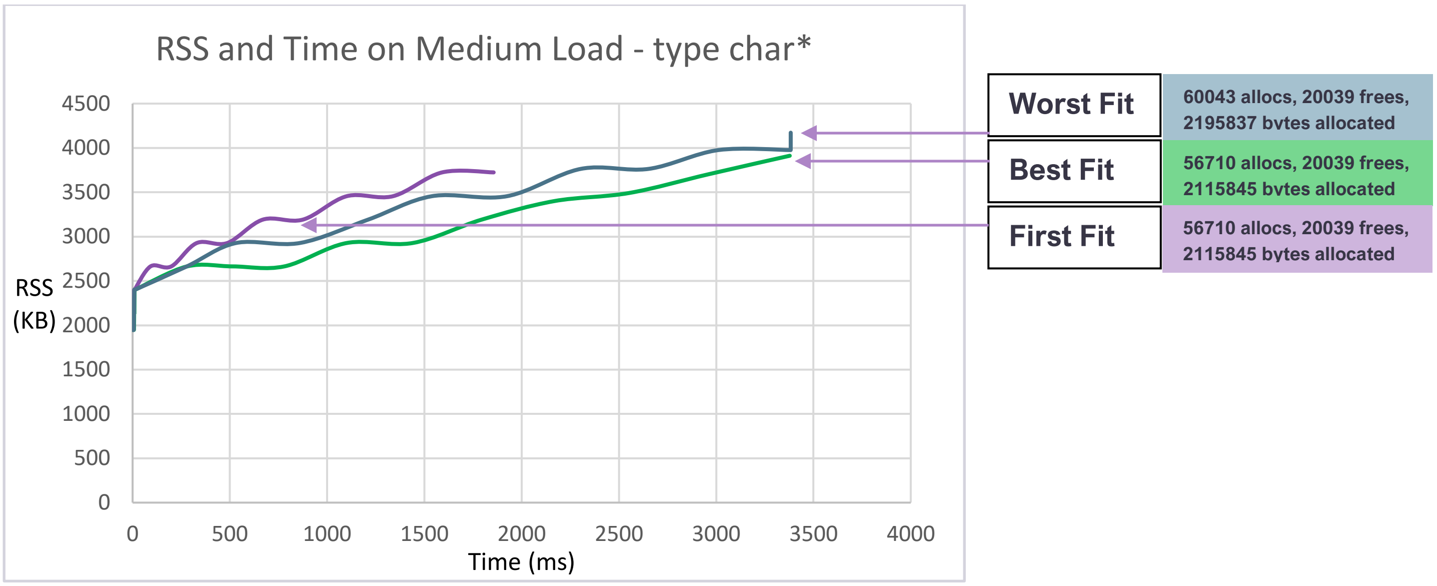
Set 5

Test number	Test name	Average total RSS (KB) across three runs	Average total time (ms) across three runs
1	Medium Load, *First Fit*, double	3716	1875
2	Medium Load, *Best Fit*, double	3848	3209
3	Medium Load, *Worst Fit*, double	4104	3296



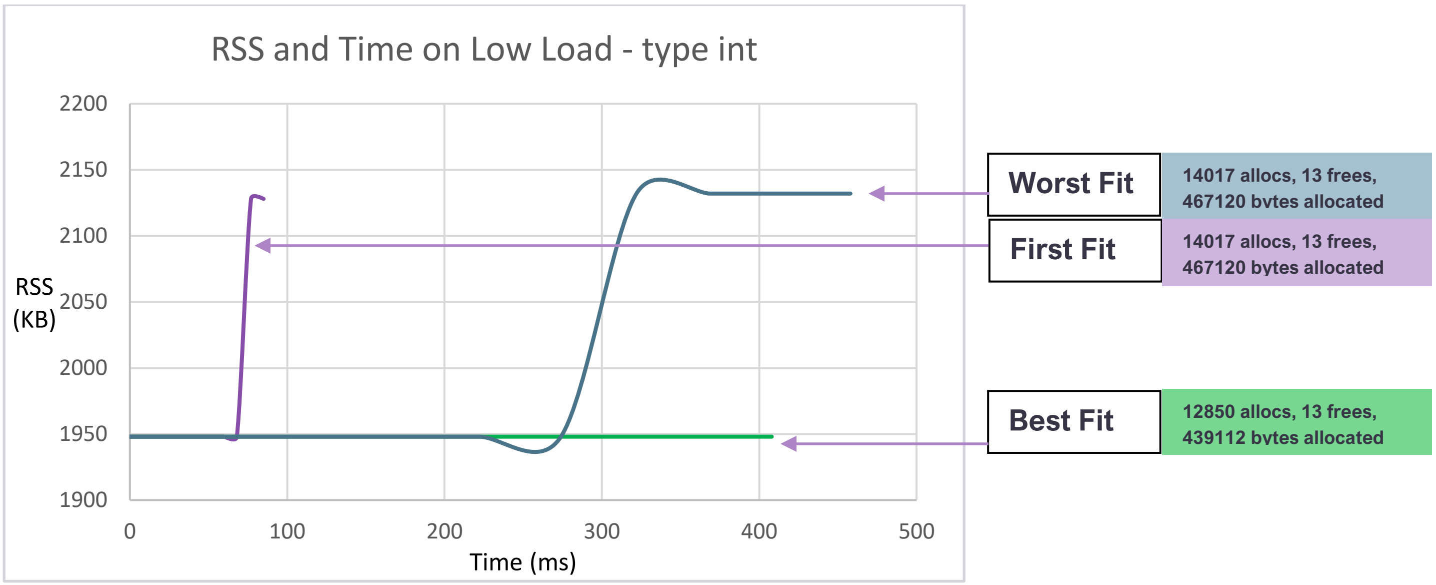
Set 6

Test number	Test name	Average total RSS (KB) across three runs	Average total time (ms) across three runs
1	Medium Load, *First Fit*, char*	3724	1914
2	Medium Load, *Best Fit*, char*	3892	3333
3	Medium Load, *Worst Fit*, char*	4172	3334



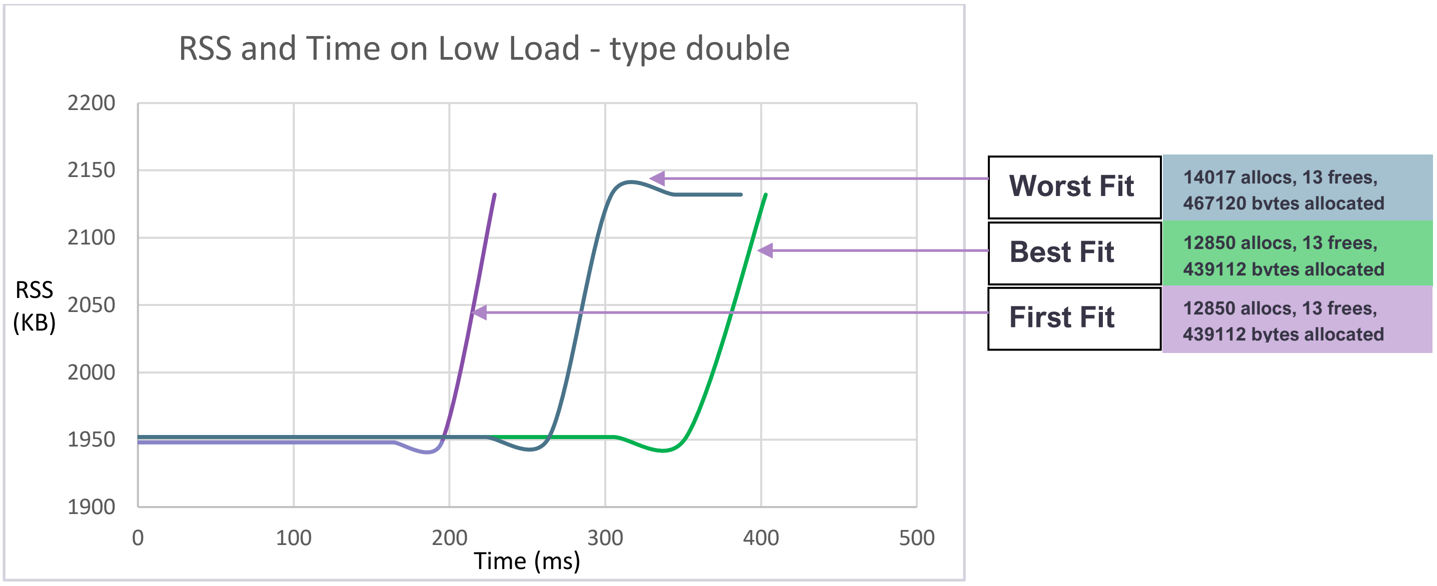
Set 7

Test number	Test name	Average total RSS (KB) across three runs	Average total time (ms) across three runs
1	Low Load, *First Fit*, int	2132	88
2	Low Load, *Best Fit*, int	1948	414
3	Low Load, *Worst Fit*, int	2132	451



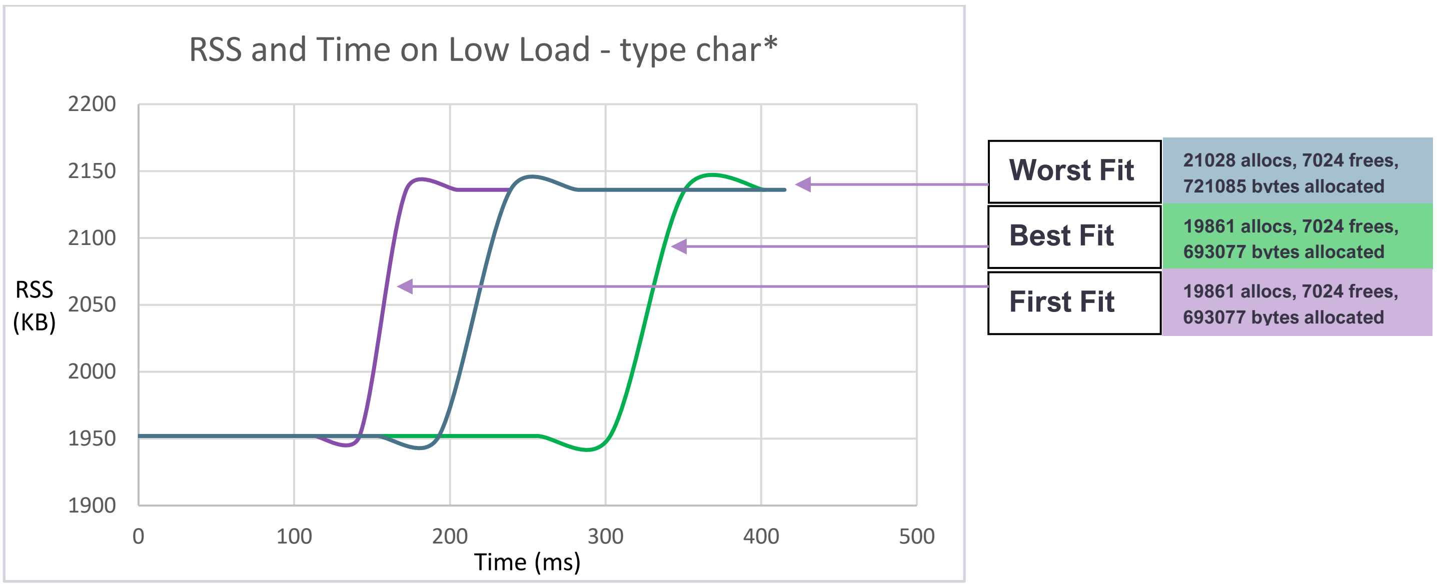
Set 8

Test number	Test name	Average total RSS (KB) across three runs	Average total time (ms) across three runs
1	Low Load, *First Fit*, double	2132	231
2	Low Load, *Best Fit*, double	2128	397
3	Low Load, *Worst Fit*, double	2132	388



Set 9

Test number	Test name	Average total RSS (KB) across three runs	Average total time (ms) across three runs
1	Low Load, *First Fit*, char*	2136	235
2	Low Load, *Best Fit*, char*	2136	418
3	Low Load, *Worst Fit*, char*	2136	411



Evaluation

Comparing the efficiency of the allocation algorithms

By graphing the total time taken for running a test to completion against the incurring RSS, the behaviour of each allocation strategy can be reasonably visualised; and a verdict can be made on its overall efficiency with regard to the data collected. The results also reveal the drawbacks of each memory allocation algorithm; in which potential enhancements can be suggested to reduce their incurring memory costs and time.

The experimental design may also contain flaws resulting in some surprising statistics. These experimental errors may be reduced with a few modifications to the design of the tests.

Efficiency of the Allocation Algorithms

In comparing the graphs produced across all test sets, it becomes evident that a similar pattern of RSS increase per time can be observed for sets under the same load capacity. When comparing test sets under a different allocation capacity, the RSS increase patterns differentiate.

For instance, ***Set 8*** and ***Set 9*** follow a reasonably homogenous trend of RSS growth against time as both sets reach a maximum threshold of around 2132 – 2136 KB and time of 388 ms and 411 ms respectively to run the process at a *Low Load capacity*.

This is in contrast to ***Set 2*** running at a *High Load capacity* and reaching a maximum RSS of 8408 KB, spanning a time of 20146 ms. It is thus reasonable to affirm that the number of allocations tasked to the memory allocator will indeed affect its overall performance. ***That is; as the number of allocations increases, the performance of the memory allocator decreases.***

Conceptually, this adheres to the underlying data structure of the linked list in the memory allocator's implementation. Traversing through the linked list requires linear time of ***O(n)*** complexity and this exacerbates as the lists grow in size. When a call is made to search the deallocated list for a target block containing either the largest or smallest available size; the operation of comparing values stored in these blocks incur greater time than finding the very first suitable block as would occur in the *First Fit* approach.

Deallocation is also achieved with linear time of ***O(n)*** complexity to search the list for the requested memory address. The results indicate that deallocation completes within 1 millisecond of the test's completion which is a significantly lower time compared to allocation. However, this is due to additional operations occurring during memory allocation which includes block construction, splitting and searching for the most suitable block in the approach used.

Considering the allocation strategies involved, the resulting data obtained across all sets of tests suggest that:

- The *First Fit* algorithm completes the test with the greatest speed; however consumes a greater amount of memory than the *Best Fit* algorithm, but not as significant as *Worst Fit.*

- The *Best Fit* algorithm completes the test with a time similar to, but slightly less than the *Worst Fit* algorithm and significantly greater than the time taken by the *First Fit* algorithm. However, *Best Fit* generally consumes the least memory out of the two other algorithms.

- The *Worst Fit* algorithm requires the longest amount of time to complete the tests across all sets and also incurs the greatest memory usage out of the two other algorithms.

The suggestion that the *First Fit* strategy executes *"with the greatest speed"* is likely to hold, as tests implementing this strategy are the first to run to completion across all sets. The *Best Fit* strategy also *"consumes the least memory out of the two other algorithms"* as evidenced in the majority of sets, and in particular ***Set 1*** under a *Low Load* where the total RSS differs by 1492 KB between the *Best Fit* and *Worst Fit* algorithms.

However, in sets subjected to a lower number of allocations, the advantage of the Best Fit approach becomes less obvious, with a significant decrease in memory usage differences compared to *First Fit* and

Worst Fit. This is observed in ***Sets 7, 8*** and **9,** where the RSS spans only between 1948 – 2136 KB and is likely due to the fact not every allocation results in a request to the OS for more space.
The standard library is able to pool some process memory, extending the pool only if no sufficient sized block could be found.

Hence it's possible that the total space required in *Low Load* tests are still within the memory allocated to the program at the start. ***Using the Best Fit approach consumes less memory and decreases the likelihood of memory fragmentation; however this advantage mainly applies to contexts involving a significant number of allocations to be made.*** The algorithm actively seeks blocks with an available space closest to the size requested; thus reducing the degree of splitting larger blocks reserved for larger requests. This occurs with the trade-off of time, which is a disadvantage of the *Best Fit* strategy.

Finally, the suggestion that the *Worst Fit* algorithm *"incurs the greatest memory usage out of the two other algorithms"* should be subjected to further analysis. While this is the observed trend across all tests and the expected conceptual outcome; the resulting RSS of tests implementing *Worst Fit* differ only slightly to that of *Best Fit.*

It may be the case that both these strategies require the longest time to complete the task and the statistics in some tests may have been skewed due to other factors such as interference from utility libraries used in the tests themselves.

Memory Requested from the OS

The RSS provides a good approximation of the amount of memory allocated to the process in RAM by the OS, however it excludes memory that was potentially swapped out, exchanged for virtual memory and stored on the disk. There is a high possibility this occurred when executing the tests on the server, along-side longer running background processes in the machine, such as the Google Chrome browser and other applications to keep track of data.

This may account for overestimates in the amount of memory used; and as such recording the *VSZ* (the Virtual Memory Size involved in Linux memory management) as an alternative may provide a more comprehensive analysis of memory consumption in the experiment.

This includes the libraries involved in the tests which also accounts for some of the memory recorded, and could be the reason for surprising fluctuations such as in ***Set 5*** and ***Set 6,*** where the total RSS for the *Best Fit* strategy (3848 KB and 3892 KB respectively) appeared greater than that of the *Worst Fit* strategy (3716 KB and 3724 KB respectively).

Memory Fragmentation

One class of memory leak is often caused by heap fragmentation and can detected when the program's reported freed memory does not reflect the RSS reported by the OS. While this allocated memory may continue to be used by the program it is never released to be available for other processes to use, creating an overhead for higher priority operations. The issue is exacerbated when the program no longer runs sufficiently to consume the remaining block segments; which themselves may not be suitable for use by other processes, and continue to remain unused.

Within this experiment, the *Valgrind* heap summary was recorded and compared to the final RSS in each test. It becomes evident from these statistics that tests utilising the *First Fit* or *Worst Fit* strategy had possibly the highest degree of memory fragmentation.

For instance in ***Set 1,*** the final RSS recorded for *First Fit* was 8188 KB (8188000 bytes) which is significantly greater than the heap size reported by *Valgrind* at 3448624 bytes. In *Worst Fit* of the same set, a similar observation is made with an RSS of 8868 KB (8868000 bytes) compared to the same 3448624 allocated in the heap. *Best Fit* produced the least difference between the RSS (7376000 KB) and heap memory (3248632 bytes) from *Valgrind* out of the three tests.

Furthermore the load capacity also contributes to the degree of fragmentation. A greater number of allocations requires larger requests for memory from the OS compared to lower loads. ***Thus, the effects of memory fragmentation are best reduced when the*** *Best Fit* ***approach is selected in scenarios requiring a significant number of allocations. Fragmentation is also reduced when fewer allocations are involved.***

Potential enhancements

The efficiency of the memory allocator may be improved with a modified algorithm of the *First Fit* approach. Although *First Fit* (as implemented in this experiment) has shown to find a block of memory in the least amount of time, its approach is not feasible on larger scale tests requiring a significant number of allocations, as the first suitable block of memory encountered is not guaranteed to be the best fit. This results in a build-up of block fragments from repeated splitting and impedes on the performance of future processes.

According to ***McCreight [13, ex. 6.2.3.301]***, a theoretically more efficient implementation of the *First Fit* algorithm utilises a *"a height-balanced binary tree (i.e., an AVL tree) with each free block corresponding to a node in the tree"*.

McCreight further outlines that each node in this AVL tree would have a field reserved for the *"size of the largest free block corresponding to a node in the left subtree attached to the given node"* and that *"[a] practical implementation would probably maintain three additional fields (two pointers to left and right neighboring free blocks and an "up" pointer to avoid the need for a stack when traversing the tree).".*

The time complexity of this approach sits at ***O(log n)*** which is an improvement from ***O(n)*** of *First Fit* used in this experiment. However the disadvantage of this enhancement is that it is not suitable for scenarios involving smaller blocks on average, as the smallest block needs to be large enough to *"hold at least five fields (two pointers to left and right descendants, a balance factor indicating the difference in height between the left and right subtrees, and two size fields).".*

In terms of reducing the amount of memory used, the ability to merge neighbouring blocks into larger blocks when their constituent smaller blocks are not enough for the current request would theoretically lessen the occurrence of memory fragmentation.

An additional enhancement could be the *First Fit* approach utilising an address ordered free list to prevent blocks of memory on one end being used preferentially over blocks on the other end. As stated by *Johnstone and Wilson [1998, p. 35]*, *"[this] gives objects at the end of memory from which new blocks are not being allocated more time to die and merge with their neighbors."*. By coalescing freed memory this way, fragmentation may be reduced.

Conclusion

A verdict on the most efficient allocation algorithm

With consideration of the obtained results and their analysis in this experiment, it is reasonable to conclude that the most advantageous memory allocation algorithm with respect to requiring the minimum amount of time is *First Fit.*

On the other hand, if less memory fragmentation is desired, the *Best Fit* application operates best. In scenarios requiring fewer allocations to be made, the advantage of *Best Fit* wavers and *First Fit* is the preferred algorithm to complete a minimal number of allocations requiring less space in general, and in the least amount of time. In all contexts, the use of the *Worst Fit* algorithm should be avoided; generating the greatest degree of fragmentation in the heap and spanning the longest time to complete the allocations.
This verdict is based solely on observations and analysis drawn from the experimental results and as such cannot be entirely guaranteed true in practical cases; as the experimental design itself has potential errors.

References

1. R. P. Brent, 1989, *First-Fit Strategy for Dynamic Storage Allocation,* ACM 11, pp. 390 – 391

2. M. Johnstone, P. Wilson, 1998, *The Memory Fragmentation Problem: Solved?*, ACM, pp. 34 – 35

3. N. Elhage. *Three kinds of memory leaks*, Made of Bugs, viewed 3 September 2019, < https://blog.nelhage.com/post/three-kinds-of-leaks/>