Dynamic Select Approach for Memory Allocation

Jyoti Raina Bakaya
M.TECH. Scholar,
Computer Science & Engineering
Kautilya Institute of Technology & Engineering
Email: raina.jyoti81@gmail.com
Mobile: 9850209827

Abstract: When we need to use Memory allocation for relatively huge datasets, then we may have a possibility to encounter the exception that is OutOfMemoryException. This exception shows that memory is not available for the allocation. But exception does not occur due to limited memory system, it usually occurs due to non availability of virtual address space for that byte of data. This issue is because of the current implementation of memory allocation which uses single array byte as backing store. When the data set is huge the backing store of memory allocation space also requires more contiguous memory than that is available in the virtual address space. If there is no contiguous memory available for the process then it encounters the exception of OutOfMemoryException even there is enough space available but not continuous. This research proposed an approach for dynamically selecting the best memory allocator for every application. The proposed approach does not need any type of contiguous memory for storing the data in stream. This approach uses a dynamic list of small chunks as backing storage that are allocated on demand when the stream is used. If there is no contiguous memory available in the Stream then memory allocation can be done from these small chunks of memory with no OutOfMemoryException.

*****

I. INTRODUCTION

An operating system has memory management unit (MMU) to manage primary memory of the system. It is the responsibility of the memory management unit to maintain records of each and every memory location that is memory management should be well informed about a memory location whether it is allocated to a process or processes or it is available for allocation to any process.

MMU checks how much memory is required to be allocated to a process or processes, and then decides which process will be allocated memory and at what time.[1] It has to keep track of memory, when memory if freed by a process or processes, memory allocated to processes and correspondingly it has to change the status of memory. Memory management provides this feature by using two registers, a base register and a limit register. The base register keeps the smallest physical memory address and the limit or restrict register specifies the size of range. Instructions and data for the memory addresses can be accessed in following ways:

Compile Time –Compile time is pre-determined time and the binding at compile time is used to create the absolute code.

Load Time – Load time is not known statically at compile time where the process will be processed in memory, the compiler creates re-locatable code.

Execution Time – When a process can be easily moved while it is executing from one memory segment to another memory segment, then binding must be delayed to be executed at run time.

Types of Memory Management

Memory management is divided into three parts, although the distinctions are fuzzy:

- Hardware Memory Management
- Operating System Memory Management
- Application Memory Management

The above mentioned Memory Management methods are present in almost most of the computer systems; to some extent form layers between the user's program and actual memory hardware [5]. The Memory Management mostly deals with the application layer memory management.

Memory Management Problems

The basic problem in managing memory is to know when to hold the data that it contains, and when to throw this data away so that memory can be used again. This sounds easy and simple, but it is a matter of study in its own capacity. In the ideal world scenario, most programmers do not have to worry about memory management problems as it is taken care of by memory management of the system. Unfortunately, there are several ways in which lack of a good memory management may affect the performance and speed of user programs, in both manual and automatic memory management situations.

The contribution of this work can be summarized as the performance of Memory allocation process of this approach over the default allocation method by operating system. This approach uses 4kb blocks of unused memory to allocate the process. When a process is requested for the contiguous memory then these small blocks are used for allocating them.

In this paper it is found that performance and capacity of memory allocation process has improved.

II. LITERATURE SURVEY

In related work which is found in literature to propose the problems of Memory Management, a number of papers have been published regarding the improvement of memory management problems.

OnurUlgen and MutluAvci in 2015 in their research that entitled as "The intelligent memory allocator selector" proposed an approach for “the dynamically selecting the best memory allocator for each application”. In their approach they executed each process with several memory allocators. When the execution was done they choose an efficient memory allocator according to situation of operating system (OS). If the system running out of memory exception occurred, then it selected the memory efficient allocator for processes which are newest. When most of the CPU power had been consumed, then it chooses the faster allocator. If it is not selected, then the balanced allocator is selected. As per the execution results, the proposed approach offered up to 59% less fragmented memory, and around 90 percent faster memory operations. Even in the average case fragmented memory is less and memory operations are faster. These results also prove the proposed approach is more reliable. This research proposed a technique that is dynamic and efficient approach for the memory fragmentation issue but their approach of solution did not solve the problem for contiguous allocation [7].

In 2013 German Molto, Miguel Caballer and others in their research entitled as "Elastic Memory Management of virtualized Infrastructures for Applications with Dynamic Memory Requirements" focused on “automatic dynamic memory management to fit dynamically at runtime for the computing infrastructure in the application, therefore adapting the memory size of the virtual machine pattern of the application.”

This research explained architecture, combined with the proof of implementation that dynamically adapts the memory size of the virtual machine to avoid thrashing while reducing the excess of free memory of virtual memory. In the test case, where a synthetic benchmark is applied that regenerate different memory consumption patterns which arise on actual scientific applications. The test cases results prove that vertical elasticity, in dynamic memory management that enables to mitigate memory over the provisioning with managed application performance penalty [8].

III. PROPOSED APPROACH

This approach does not require contiguous memory to allocate the data that the memory stream has. This approach uses a dynamic list of small blocks as the backing store which is decided by the user, which are allocated to process on demand when a process requests memory.

Our approach is also derived from the Stream class but the allocation process is different from the normal process of allocation. This approach allocates small chunks of memory as continuous memory to a process. This is capable of initializing from array of a byte

1. When a process requires memory, it request for the memory then the allocation of blocks is done on demand either for the operation read or write. The Position is checked with respect to the Length before a read operation takes place, to make sure the read operation is performed within the limit of the stream. Length is just to check if the position is below the length size and not for the allocation amount of memory, setting the Length size does not allocate memory to the process, rather it allows reads to proceed on the data.

```
//A new object of class: where length=0, position=0, and no memory allocated

Memory_msps ds = new Memory_msps();

//returns -1, no memory allocated

int data = ds.ReadByte();

//Length now becomes 10000 bytes, but no memory allocated

ds.SetLength(10000);

//three blocks of memory are allocated now,

//but data1 is undefined

int data1 = ds.ReadByte();
```

2. Memory is allocated in sequential blocks that make the continuous memory which is required for the process. That is, if the first block requests to access the third block, then first and second blocks are automatically allocated.

IV. RESULTS ANALYSIS

4.1Performance Metrics

There are two parameters for analysis which algorithm behaves in different scenarios. The two parameters are performance and storage capacity. On the basis of these two parameters the real implementation of both algorithms can be determined.

It is difficult to predict performance both in terms of storage capacity and speed, of default class and my approach. Performance is dependent on a number of factors, one of the most significant being the fragmentation and memory usage of the current process, a process which allocates a lot of memory will use up large contiguous sections faster than one that does not – even though it is possible to get an idea of the relative performance characteristics of the two approaches by taking measurements in controlled conditions.

The tables below compare the capacity and access times of default and my approach. In all cases the process instance has tested only the target stream.

Storage Capacity

To check storage capacity, an operation is performed in which a loop write the contents of 1MB array to the target stream over and over until the stream throw an OutOfMemoryException, that was caught and the total number of write executions before the exception were returned.

TABLE 4.1 CAPACITY OF CLASSES

Stream	Average Stream Length Before Exception (MB)
Default Class	785
My_Class	2272

Speed (Access Time)

To test speed or access time using default and my approach, a set of data was written to perform the operation, then a read operation on the stream. The data was written in lengths between 1KB to 1MB, to and from a 1MB byte array. A Stopwatch is used to calculate the amount of time it takes to write, then read, the specified amount of data.

The test was executed five (5) times by each process on the same data. The variation in the results shows the time taken by stream every time for allocating memory and accessing memory, which vary every time.

Access Time with 10MB Data

Here the operations are performed over 10MB data and results are calculated on different scenarios.

TABLE 4.2

ACCESS TIME WITH 10MB DATA

	Stream Test Execution Times (ms)			
Amount Written and Read (10 MB)	Default	My_Class (128KB Block)	My_Class (512KB Block)	My_Class (1MB Block)
Execution 1	11	14	10	8
Execution 2	4	6	4	4
Execution 3	4	7	4	3
Execution 4	4	6	3	3
Execution 5	5	6	3	3
Average	5.6	7.8	4.8	4.2

The above table shows the results performed on 10MB data with different approaches. It is calculated by the average of these operations to find the variations between all the scenarios.

Here it is found that when My_Class is applied with 128KB block then it takes too much time and when size of block is increased, then it takes less time as compared to default class.

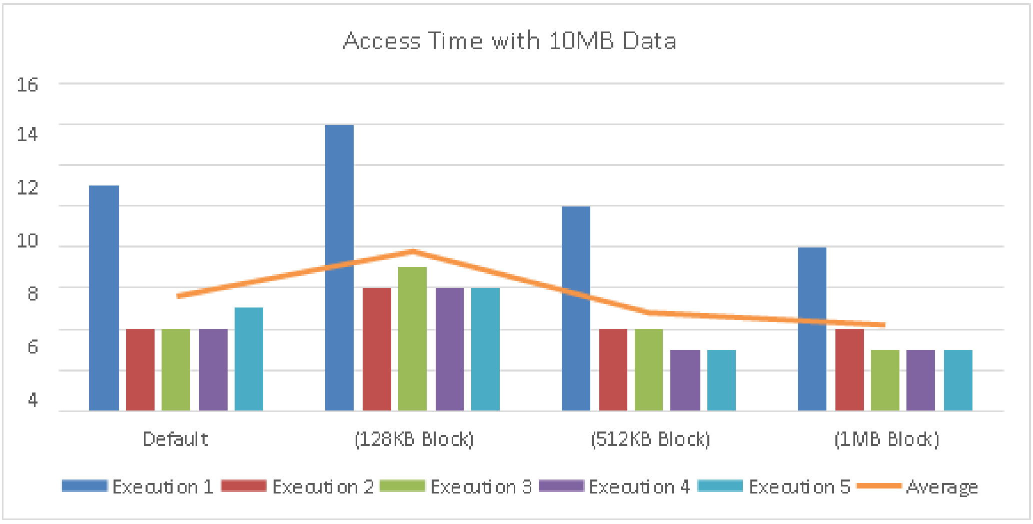

Figure 4.1: Access Time with 10MB Data

In the above figure results are compared for the operations performed on 10MB data with different approaches. It is calculated by the average of these operations to find the variations between all the scenarios.

Access Time with 100MB Data

Here the operations are performed over 100MB data and results are calculated on different scenarios.

TABLE 4.3

ACCESS TIME WITH 100MB DATA

	Stream Test Execution Times (ms)			
Amount Written and Read (100 MB)	Default	My_Class (128KB Block)	My_Class (512KB Block)	My_Class (1MB Block)
Execution 1	105	150	97	57
Execution 2	39	56	36	40
Execution 3	39	54	37	39
Execution 4	40	53	36	40
Execution 5	39	52	36	40
Average	52.4	73	48.4	43.2

From the above scenario it can be said that when the block size is of 1MB then access time is less as compared to other cases where block size is less than 1MB and operations are performed on the 100MB data. Here it is found that when My_Class is applied with 128KB block then it takes too much time and when increase the size of this block then it takes less time as compare to default class.

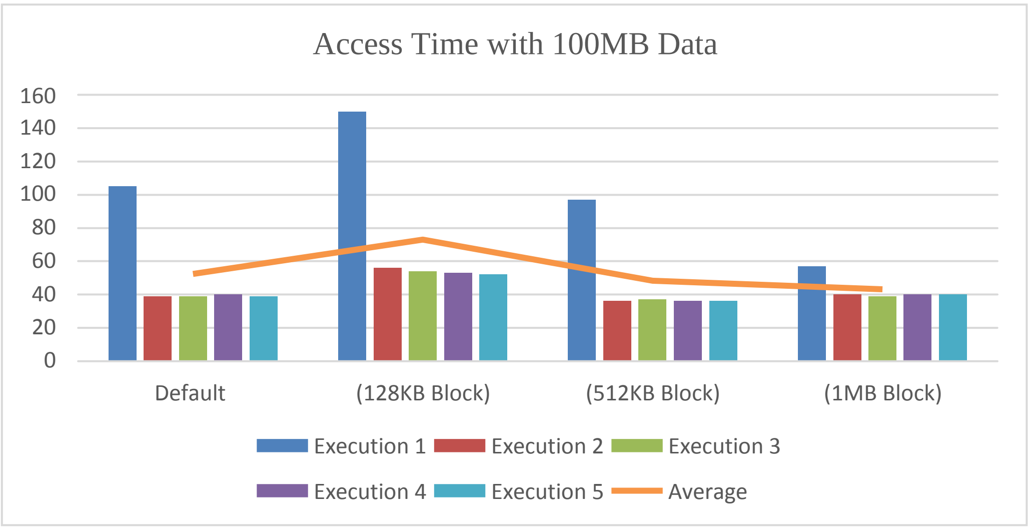

Figure 4.2: Access Time with 100MB Data

Access Time with 500MB Data

Here the operations are performed over 500MB data and results are calculated using different block sizes of my_class.

TABLE 4.4

ACCESS TIME WITH 500MB DATA

	Stream Test Execution Times (ms)			
Amount Written and Read (500 MB)	Default	My_Class (128KB Block)	My_Class (512KB Block)	My_Class (1MB Block)
Execution 1	520	396	297	242
Execution 2	172	228	190	175
Execution 3	173	192	160	172
Execution 4	173	193	157	173
Execution 5	172	193	158	173
Average	242	240.4	192.4	187

Here this table proves the result variation between various blocks used for the memory storage and 1 MB size block takes very less time.The operations are performed with the 500MB data. Here again it is found that when My_Class is applied with 128KB block size then it takes more time than default class. And when block size is either 512KB or 1MBtoo the default class takes more time

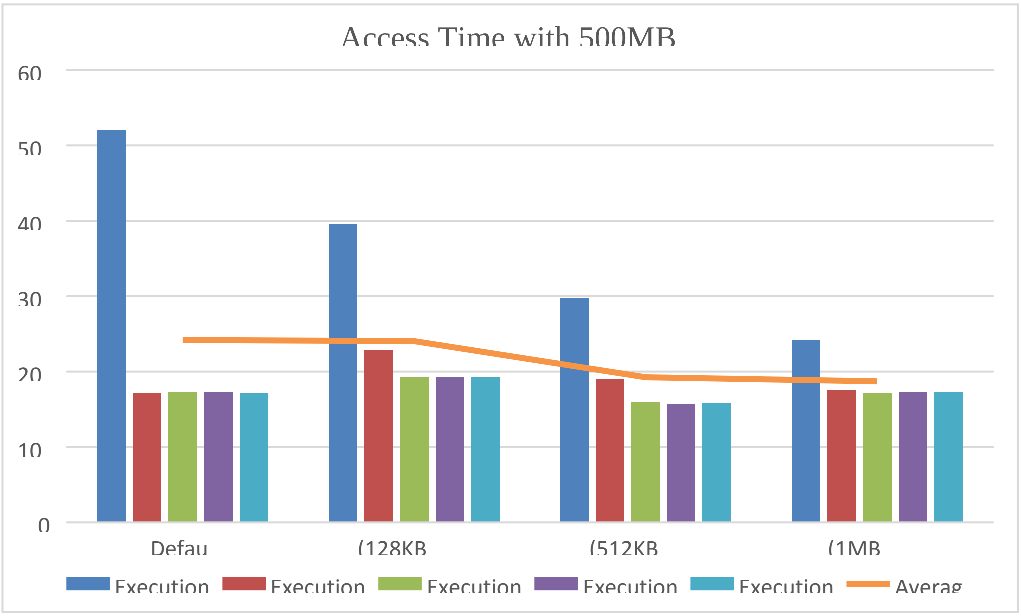

Figure 4.3: Access Time with 500MB Data

Here this figure shows the results variation between the various blocks used for the memory storage and 1 MB size block takes very less time. These operations are performed with the 500MB data. Here it found that when My_Class is applied with 128KB block then it takes too much time and when increase the size of this block then it takes less time as compare to default class.

Access Time with 1000MB Data

Here the operations are performed over 1000MB data and results are calculated on different scenarios.

TABLE 4.5

ACCESS TIME WITH 1000MB DATA

	Stream Test Execution Times (ms)			
Amount Written and Read (1000 MB)	Default	My_Class (128KB Block)	My_Class (512KB Block)	My_Class (1MB Block)
Execution 1	1210	1595	817	490
Execution 2	356	556	403	351

Execution 3	356	570	359	350
Execution 4	345	568	357	349
Execution 5	342	567	356	349
Average	521.8	771.2	458.4	377.8

Here this table proves the result variation between the various blocks used for the memory storage and 1 MB size block takes very less time. These operations are performed with the 1000MB data. Here we found that when My_Class is applied with 128KB block then it takes too much time and when the size of this block is increased then it takes less time as compare to default class.

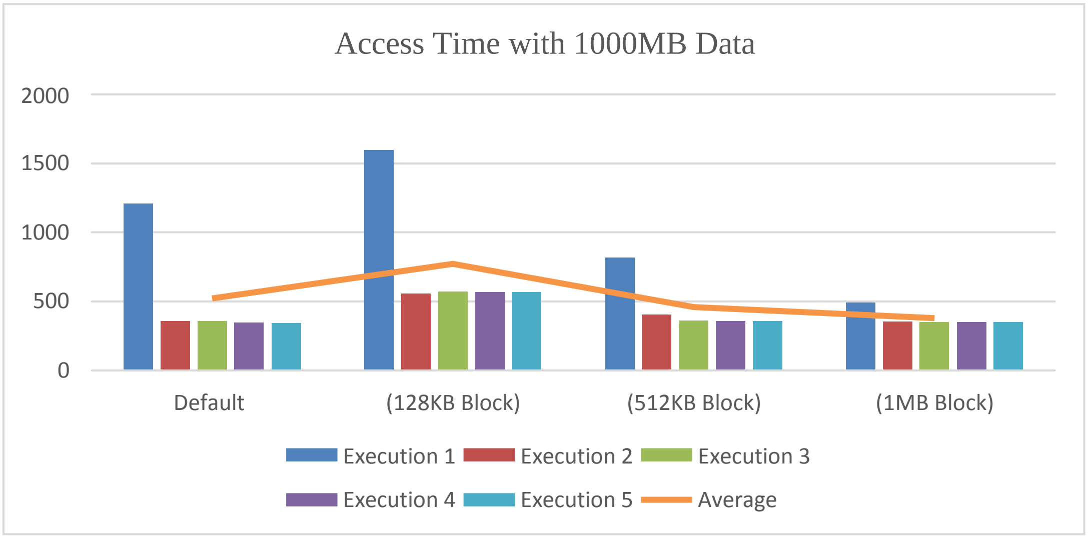

Figure 4.4: Access Time with 1000MB D

Figure 4.4 presents the result variation between the various blocks used for the memory storage and 1 MB size block takes very less time. These operations are performed with the 1000MB data. Here it found that when My_Class is applied with 128KB block then it takes too much time as compared to default class and when the size of the block is increased then it takes less time as compare to default class.

Average Access Time with Different Data

TABLE 4.6

AVERAGE ACCESS TIME WITH DIFFERENT DATA

	Stream Test Execution Times (ms)			
Amount Written and Read on Data	Default	My_Class (128KB Block)	My_Class (512KB Block)	My_Class (1MB Block)
10 MB	5.6	7.8	4.8	4.2
100 MB	52.4	73	48.4	43.2
500 MB	242	240.4	192.4	187
1000 MB	521.8	771.2	458.4	377.8

Above table summarizes the average of tests performed on different data that is 10MB, 100MB, 500 MB and 1000MB data by default class and My_class with different block. Results are shown in visual format below.

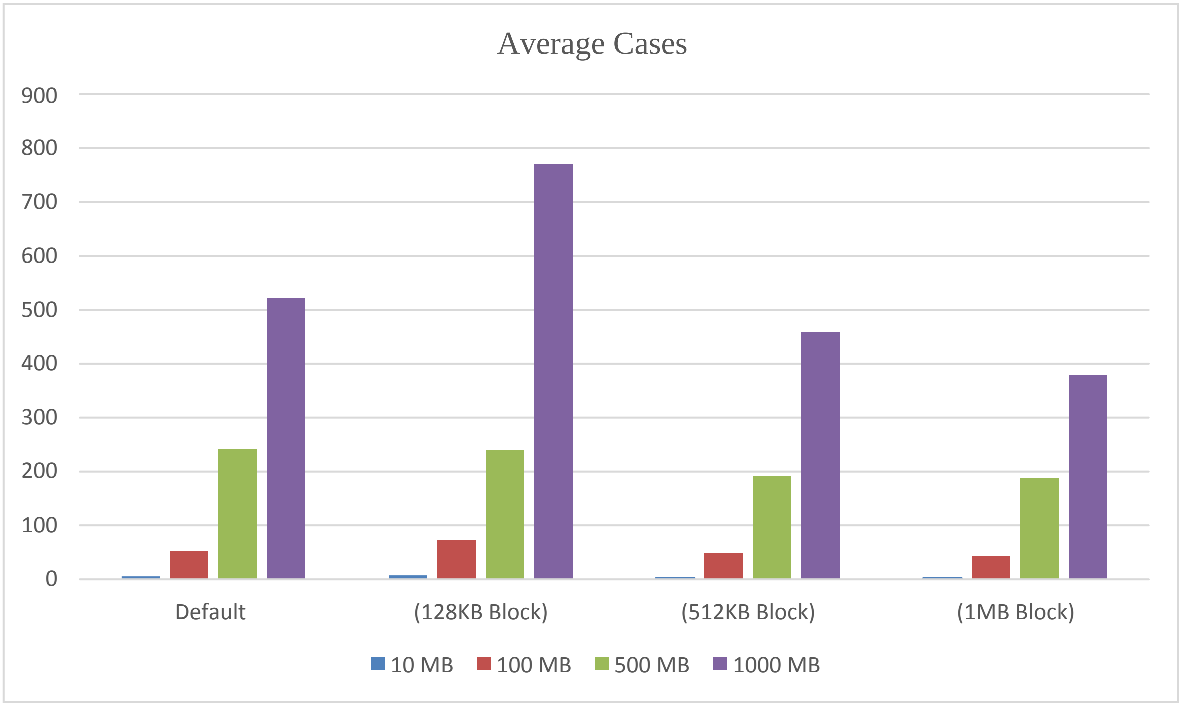

Figure 4.5: Average Access Time with Different Data

Here this figure presents the results variation between the various blocks used for the memory storage and 1 MB size block takes very less time. Here it found that when My_Class is applied with 128KB block then it takes too much time as compared to default class and My_class applied with 512 KB or 1MBnd block size.

V. CONCLUSION AND FUTURE SCOPE

Conclusion

The results indicate that My_Class can store more than double the data of Default in ideal conditions. The access times depend on the block size of the memory setting of My_Class; the initial allocations are margin faster than Default class but access times are similar. The smaller the block the more allocations must be done and got the best results with block of 1MB.

This shows the performance and access time are better in case of approach here implemented and it is able to allocate more process to memory when there is an exception encounters in the normal case.

Future Scope

This paper covers the limitation of data to the 1000MB data after this it slows down the system.

In the future there may be some chance of improvement and improvements can be done by increasing the virtual address size so that when a process requests for memory then memory can easily be allocated to that process.

REFERENCES

[1] Silberschatz A, GalvinPB, GagneG Operating system concepts Boston, MA Wiley.

[2] Tanenbaum AS, Woodhull AS Operating systems design and implementation.

[3] Evans J, Scalable memory allocation using jemalloc, 2011. Available at this URL ⟨http://j.mp/1H6zIm4⟩.

[4] E. Kalyvianaki, T. Charalambous, S. Hand, Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters, in: Proceedings of the 6th international conference on Autonomic computing - ICAC '09, ACM Press, New York, New York, USA, 2009, p. 117

[5] Tutorialspoint Team, "Introduction to Memory Management", Source Available at http://www.tutorialspoint.com, July 2015.

[6] "Why use CPUs without MMU?" Page source Available at: http://www.uclinux.org/pub/uClinux/archive/5762.html

[7] Germ´an Molt´, Miguel Caballer, Eloy Romero, Carlos de Alfonso, Elastic Memory Management of Virtualized Infrastructures for Applications with Dynamic Memory Requirements, International Conference on Computational Science, ICCS 2013

[8] OnurÜlgen, MutluAvci, The intelligent memory allocator selector, Computer Languages, Systems & Structures Vol 44, Pages 342–354, Year 2015.

[9] Hasan Y, Chang M. A study of best-fit memory allocators. Comput Lang SystStruct 2005; 31(1): 35– 48.

[10] Hasan Y, Chang M. A tunable hybrid memory allocator. J SystSoftw 2006; 79(8):1051–63.

[11] Garca-Martnez A, Fernández – Conde J, Viña Á Efficient memory management in video on demand servers. ComputCommun 2000; 23(3):253–66.

[12] Gustavo Duarte, "Page Cache, the Affair between Memory and Files". Available at: http://duartes.org/gustavo/blog/category/internals/

[13] JonesR,HoskingA,MossE.In:Thegarbagecollectionhandbook:theartofautomaticmemorymanagement.BocaRaton ,FL:Chapman &Hall/CRC;2011.

[14] "Process address space". Available at: http://kernel.org/doc/gorman/html/understand/understand007.html

[15] D. Williams, H. Jamjoom, Y.-H. Liu, H. Weatherspoon, Overdriver: handling memory overload in an oversubscribed cloud, ACM SIGPLAN Notices 46 (7) (2011) 205. doi:10.1145/2007477.1952709.

[16] W. Zhao, Z. Wang, Y. Luo, Dynamic memory balancing for virtual machines, ACM SIGOPS Operating Systems Review 43 (3) (2009) 37. doi:10.1145/1618525.1618530.