Volume 4, Issue 6, June 2014 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

An Intelligent Memory Allocator with in-built Dynamic Memory Leak Detection and Garbage Collection for Distributed Environment

Shubhra Garg*
Scholar, M. Tech (Comp. Sci. & Engg.)
Yamuna Institute of Engineering & Technology,
Gadholi, Yamunanagar, India

Ravinder Chauhan
Asst. Professor, Dept. of Computer Science,
Yamuna Institute of Engineering & Technology,
Gadholi, Yamunanagar, India

Abstract— Memory Leak and Memory Corruption pose a big challenge to the programmers since the very beginning of the programming era. Programming languages such as C and C++ do not support for the detection of Memory leak, memory corruption and garbage collection as other languages, such as JAVA, do. In this paper, we present an independent, self-sufficient Memory Analyzer and Allocator that dynamically detects memory leak in the course of allocating the requested memory to the program, thereby performing garbage collection on the way. This algorithm is a modified version of the traditional Mark & Sweep algorithm for garbage collection. The modified algorithm shows marked performance improvement over the traditional counterpart. It would also provide the user with all statistics in terms of amount and location of memory leak found, amount of corrupted memory and amount of memory successfully collected.

Keywords— Memory Leak, Garbage Collector, Memory Analyzer, Mark & Sweep, Garbage Detector

I. INTRODUCTION

The memory management subsystem is one of the most important parts of the operating system. Since the early days of computing, there has been a need for more memory than that exists physically in a system. Strategies have been developed to overcome this limitation and the most successful of these is virtual memory. Virtual memory makes the system appear to have more memory than it actually has by sharing it between competing processes, as they need it. [1].

Memory management is a complex field of computer science and there are many techniques being developed to make it more efficient. Some platforms have specific problems in dealing with memory. Memory management is usually divided into three areas: hardware, operating system, and application, although the distinctions are a little fuzzy. In most computer systems, all three are present to some extent, forming layers between the user's program and the actual memory hardware. The Memory Management is mostly concerned with application memory management. [2]

A. What is Memory Leak?

In computer science, a memory leak is a particular kind of unintentional memory consumption by a computer program where the program fails to release memory when no longer needed. The term is meant as a humorous misnomer, since memory is not physically lost from the computer. Rather, memory is allocated to a program, and that program subsequently loses the ability to access it due to program logic flaws. As is noted below, a memory leak has similar symptoms to a number of other problems, and generally can only be diagnosed by a programmer with access to the program source code. [3]

B. Causes of Memory Leak

One of the joys of working with managed code is not to worry about memory management and letting the Garbage Collector do its job. There are situations, however, when applications need to take a more active role in memory management. Three common causes of memory leaks in managed applications:

1) Holding references to managed objects: When an application holds references longer than necessary, performance counters can show a steady increase in memory consumption and may, eventually, lead to an OutOfMemoryException. This situation can arise when variables never leave active scope.

2) Failing to release unmanaged resources: Writing managed code largely frees you from needing to be concerned about memory management. This is true when you are using only managed objects.

3) Failing to dispose Drawing objects: A third common cause of memory leaks in managed applications is actually a manifestation of the previous cause. It is important to call the object's Dispose method when you no longer need the object. This is important because, while these are managed objects, they contain references to unmanaged resources. These references are cleaned up (and the memory used is freed) when the object is disposed.

C. Consequences and effects of Memory Leak

If a program has a memory leak and its memory usage is steadily increasing, there will not usually be an immediate symptom. Almost all systems have a certain amount of available memory, which can be consumed by programs. Eventually, the available RAM may run out. This may have one of two effects

- On systems where all memory is in RAM, there will be an immediate failure
- Most modern operating systems on general-purpose computers use a hard disk to provide virtual memory. The effect once RAM has run out is increasing use of hard disk. The hard disk can in turn eventually run out, but usually the performance of the application and/or system will become so slow that they will be considered to have failed before that point.

If a system crashes simply because an application has continually asked for more memory, this would be considered poor system design or a bug. What normally happens is that the program eventually asks for memory and be refused, either because there is no more or because a per-application limit has been reached. What happens next depends on the program, and can include:

- The application terminates itself, perhaps with an error message.
- The application tries to recover from the error. In fact this is usually unsuccessful, because memory would be needed to do that, and the application has by definition lost track of how it could free the leaked memory. However, some applications start by reserving a safety area of memory, which is freed if a failure occurs to allow some limited recovery.
- The application assumes that the request for memory has succeeded, and continues on this basis. This will typically result in an access violation but in some cases may result in damaging information belonging to this or (in primitive systems) some other application.

II. RELATED WORK

Many researchers and programmers have demonstrated varied techniques to accomplish the task of garbage collection. Willard and Frieder [4] came up with autonomous garbage collection, an extension of conservative garbage collection that effectively mitigates performance degeneration due to memory leaks in the applications. Veiga and Ferreira [5] have developed a complete distributed garbage collection algorithm, based on: (i) a reference-listing algorithm; and (ii) a centralized algorithm that complements the previous one by detecting distributed cycles of garbage. Xiangrong et. al. [6] went a step further to relate software memory leakage to Neuroscience Principles that govern human memory behaviour. They have mapped the Cue, Recognition and Recall used in Kahana's neuroscience method to the similar memory elements of operating system, and applied Yule's Q equation to accurately pinpoint the memory leak in the source code. Feng et. al. [7] proposed a tool called SafeMem to detect memory leaks and memory corruption on-the-fly during production-runs. This tool makes a novel use of existing ECC memory technology and exploits intelligent dynamic memory usage behaviour analysis to detect memory leaks and corruption. Timothy et. al [8] describe a run-time scheme that detects and removes memory leaks with minimal performance overhead and with no modifications to application source code. The scheme consists of a first stage where a pattern recognition technique proactively detects subtle memory leaks, followed by a more resource-intensive second stage that scans the memory space of an application and removes detected memory leaks.

A. Available Tools

There are many tools available today to detect memory leaks. Two of them are DMALLOC [9] and VALGRIND [10]. These tools successfully detect memory leaks and give detailed information about memory leaks in an executing application. The main drawback of these tools is that they detect memory leaks of applications reactively (i.e. they detect memory leak only after it occurs). Other tools are too expensive in terms of efficiency. Some tools are better in efficiency but they take more memory, others are low on memory consumption but they slow down the process under observation. Study shows that a garbage collector makes a process 30 – 40% slower. Hence, it becomes important for the memory leak detection and garbage collection algorithm to balance these two aspects: time as well as space.

III. CONCEPTUALIZATION AND METHODOLOGY

Our design approach is built upon the traditional Mark & Sweep garbage collection algorithm. After studying the literature we found that Mark and Sweep algorithm is a good candidate for detecting memory leaks and collects garbage for the C programs.

Mark and Sweep has the following advantages:

- Mark and Sweep garbage collection algorithm traces out the set of objects accessible from the roots, it is able to correctly identify and collect garbage even in the presence of reference cycles. This is the main advantage of mark-and-sweep over the reference counting technique.

- Secondary benefit of the mark-and-sweep approach is that the normal manipulations of reference variables incur no overhead.

One disadvantage of the Mark and Sweep is that it incurs some performance overhead. Mark phase of Mark and Sweep algorithm is mainly the cause of the performance degradation. Figure 1 shows the working of Mark and Sweep algorithm. The algorithm keeps a book-keeping list to keep track of the allocated memory. However, it is expensive in terms of performance.

The suggested new approach will improve the performance of Mark and Sweep algorithm. It modifies the traditional algorithm by introducing the concept of Normalize Factor. The memory allocator module always returns the address which is a multiple of Normalize Factor. When the algorithm is in Mark Phase, it first scans the stack elements for the multiple of Normalize Factor (Fig. 2). Only those values are compared to the book-keeping list and marked as live, if found. The flow of the algorithm is shown in figure 3.

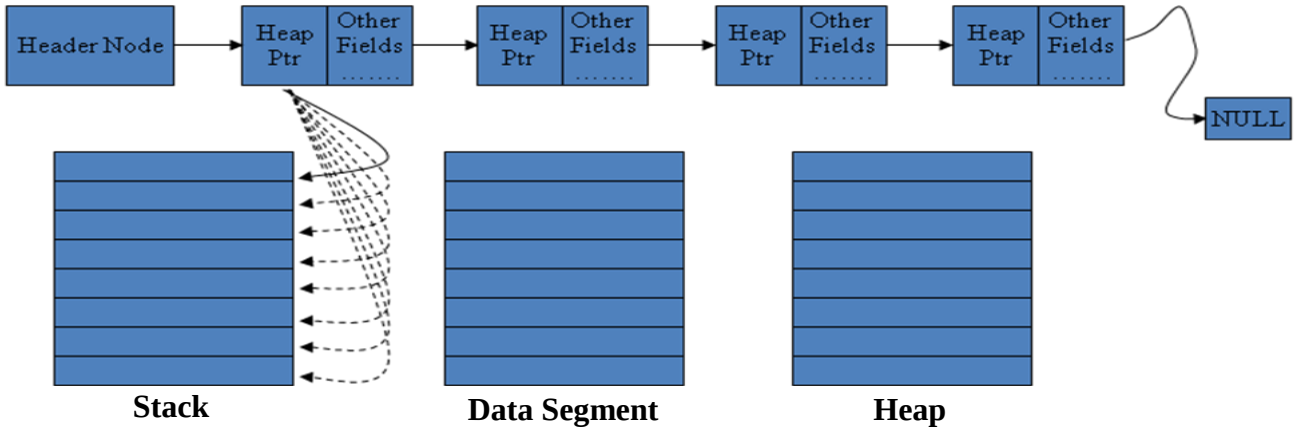

Fig. 1: Working of traditional Mark & Sweep

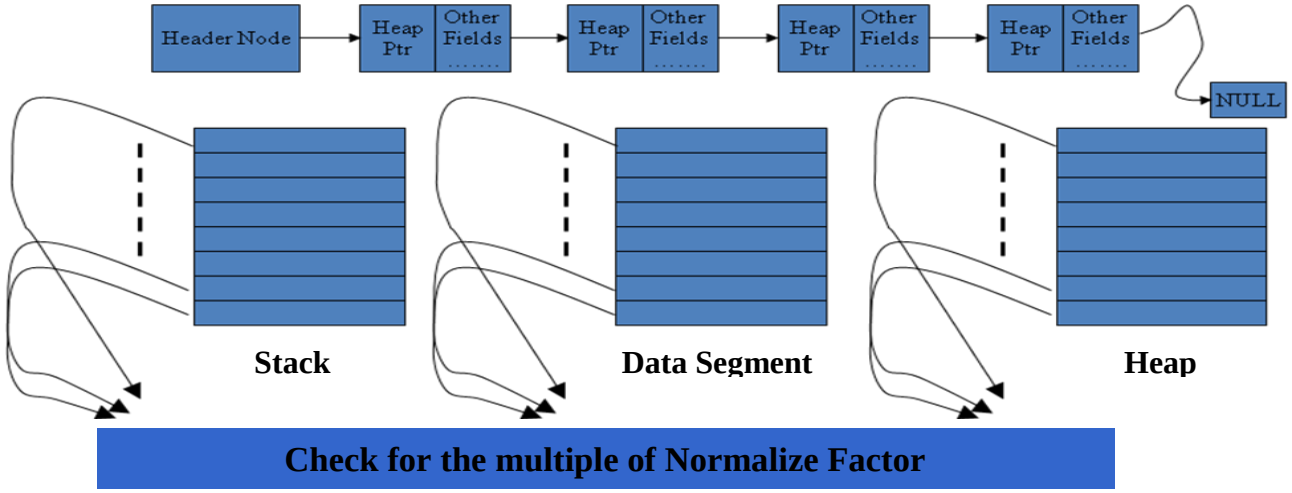

Fig. 2: Modified Mark and Sweep algorithm

IV. Discussion of Results

To compare the performance matrices of the traditional Mark and Sweep with the modified Mark and Sweep algorithm, let us consider the number of entries in the book-keeping list to be 50 and the number of elements in stack to be 1000. In order to search for the references in the stack, it would be traversed 50 times, amounting to 50000 comparisons. Same number of comparisons are performed for the data and heap segment. The modified algorithm beats the traditional algorithm in the best and average cases. Table 1 compares both the algorithms.

TABLE 1

NUMBER OF COMPARISONS IN TRADITIONAL AND MODIFIED MARK & SWEEP ALGORITHM

	Number of Comparisons in		
	Best Case	**Average Case**	**Worst Case**
Traditional Mark & Sweep	50000	50000	50000
Modified Mark & Sweep	1000 + 1000 = 2000 (If 20 entries are multiple of Normalize Factor)	1000 + 25000 = 26000 (If 50%, i.e. 500 entries are multiple of Normalize Factor)	1000 + 50000 = 51000 (If all entries are multiple of Normalize Factor)

V. CONCLUSION

Memory leak occurs when a programmer allocates the memory but fails or forgets to free that part of memory when it is no longer required. Memory leak is a silent killer because there is no immediate symptom shown by the program. If a program has memory leak and it increases gradually, then at some point of time it'll consume the whole available physical memory and the user can only know when the system runs out of memory. If the system contains only the physical memory then the system will fail immediately after all the memory is exhausted. But if a system uses virtual memory, then thrashing (high paging activity) is performed that degrades the performance of the system considerably.

The suggested memory leak detection and garbage collection algorithm efficiently overcomes the hurdle of program failure due to memory leak. It introduces a modified version of the traditional Mark & Sweep algorithm that accomplishes the task even more optimally. It analyzes the memory dynamically, allocates the requested memory to the program, detects memory leak and collects garbage to recover the lost memory. It also reports to the user about the memory leak details that have occurred. This algorithm can be easily embedded with application programs and ease the task of the programmers to a great extent.

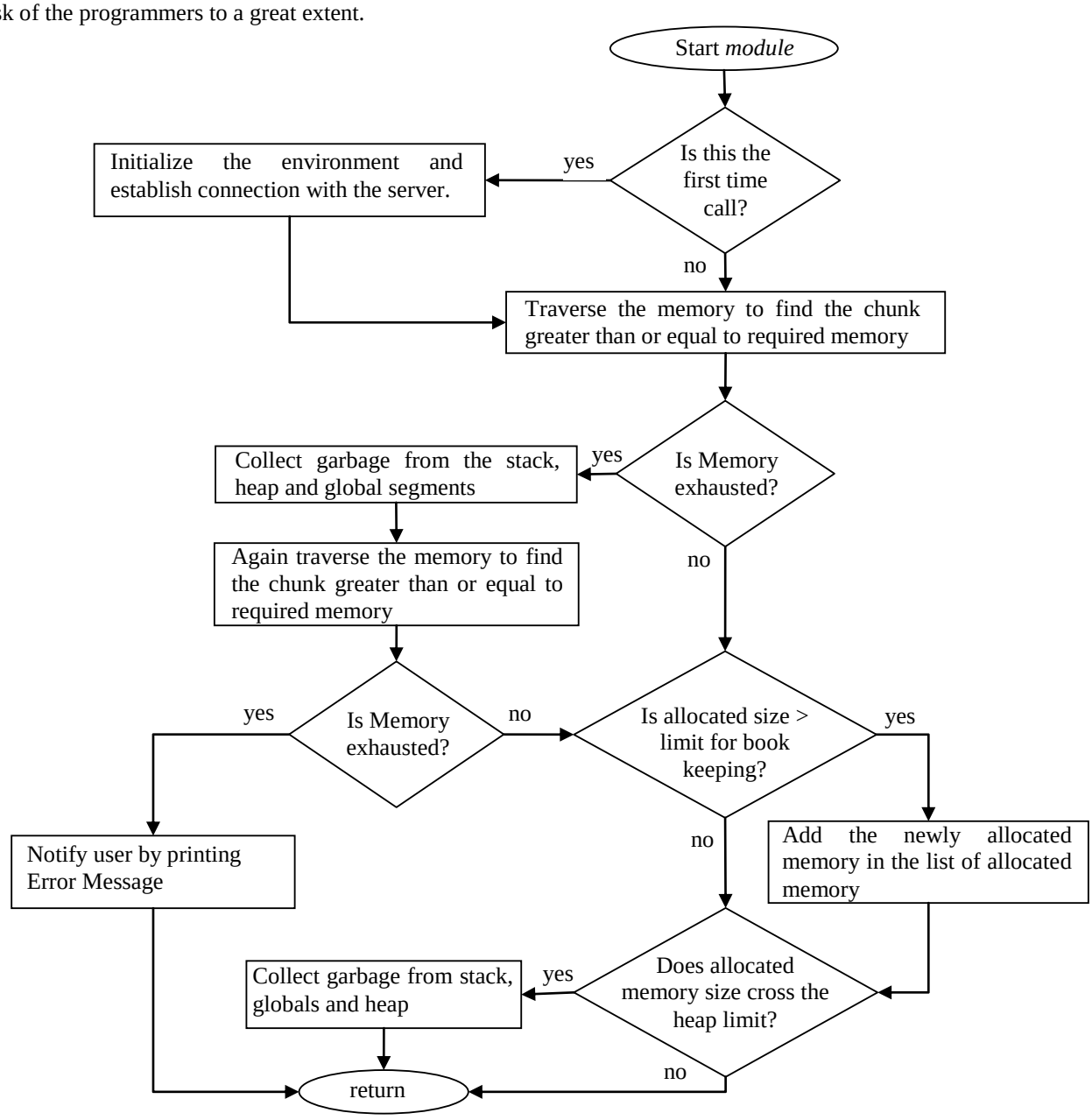

Fig. 3: Flowchart of the algorithm

ACKNOWLEDGEMENT

The authors are thankful to all the individuals under whose generous guidance and assistance this work has been carried out. A special thanks to Mr. Pritam Pal, Sr. Software Engineer, Novell and Mr. Mohit Ved, Sr. Technical Officer, CDAC for their consistent support and apt guidance. We wish to thank all those, who generously gave their time and knowledge, and facilitated an environment conducive to the development of innovative ideas.

REFERENCES

[1] Memory Management page on The Linux Development Project website [Online]. Available: http://www.tldp.org/LDP/tlk/mm/memory.html

[2] Memory Management Reference website [Online]. Available: http://www.memorymanagement.org/articles/begin.html

[3] Memory Leak article on Wikipedia [Online]. Available: http://en.wikipedia.org/wiki/Memory_leak

[4] Willard, B. and Frieder, O., "Autonomous garbage collection: resolving memory leaks in long running network applications" in *Proc. ICCCN-1998*, 1998 pp. 886 – 896.

[5] Veiga, L. and Ferreira, P., "Complete distributed garbage collection: an experience with Rotor," in *IEEE Proc.*- Volume 150, Issue 5, 2003, pp 283 – 290

[6] Wang, X., Xu, J. and Pham, C.H., "An effective method to detect software memory leakage leveraged from neuroscience principles governing human memory behavior," in *Proc. ISSRE 2004*, 2004, pp. 329 – 339

[7] Feng Qin, Shan Lu and Yuanyuan Zhou, "SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs," in *HPCA-11*, 2005, pp. 291 – 302

[8] Tsai, T., Vaidyanathan, K. and Gross, K., "Low-Overhead Run-Time Memory Leak Detection and Recovery," in PRDC '06, 2006 pp. 329 – 340

[9] Debug Malloc website [Online]. Available: http://dmalloc.com

[10] Valgrind website [Online]. Available: http://valgrind.org

