International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 4 Issue 02, February-2015

Efficient Handling of Low Memory Situations in Linux

Prashant B. Patare
Department of Computer Science and Engineering
NIT Calicut
Kerala, India

V. K. Govindan
Department of Computer Science and Engineering
NIT Calicut
Kerala, India

***Abstract*—Low memory situations in computing systems is always a daunting issue for operating systems. When a requested memory requirement by an application can not be fulfilled, the system calls out-of-memory (OOM) killer to kill one or more processes so as to free some memory. In the current paper we are proposing an approach where the system sends signals to currently running processes so that the processes by themselves can free some memory. Hence, most of the times we get sufficient free memory without calling the OOM killer. This proposed approach can be used along with the approach of considering the application's past usage history. In addition, in case if the low memory condition still persists after sending the low memory signals to processes, then lower priorities for being killed, are given for the process which has freed some memory. We will also be suggesting some better ways for controlling the OOM killer management for some of the processes.**

Keywords—Out of memory killer; danger signal; signal threshold; improviser module; obeying factor; Linux operating system.

I. INTRODUCTION

Every computer system has a system memory that we call RAM, which holds applications for running. System memory is available in limited amount only. Therefore, optimal use of it is very essential for good system performance.

Applications can ask for more memory when they need it using system calls like malloc. However, applications usually do not use all of its requested memory. Hence, considering this fact in mind, the Linux operating system allows over-commitment of memory, in which the operating system allows more memory to be granted to processes than available. The actual memory assignment is differed till actual usage of that memory page.

Virtual memory allows more memory to be used than available physical memory. Virtual memory uses the combination of actual physical memory and the swap space present on secondary storage holding memory pages.

Out of Memory (OOM) condition is unavoidable due to over-commitment policy of operating system. In OOM condition, the system can not allocate memory to a requesting application and hence, it needs to kill one or more processes from the system; this is the job of OOM killer. It uses heuristic approach to find the process that must be killed.

OOM killer first calculates the badness value for each running process. The badness value indicates the likelihood of the process to be killed. While calculating the current memory usage of the process, total running time and some user defined parameters are taken into consideration. The detailed algorithm is explained in [9].

Embedded systems have less memory and no swap area. In this case there is more possibility of OOM condition to occur more frequently compared to the high-end computer system.

The rest of the paper is organized as follows: Section II briefly reviews some of the related work in the topic of research, and brings out lacunae of out of memory killer to handle low memory situations. The proposed new approach of handling low situations is presented in Section III. The system design incorporating the low memory improviser module is presented in Section IV. Section V and VI respectively deal with the algorithm for handling low memory situations, and method for testing system performance with the improviser module. Finally, the paper is concluded in Section VII with proposal for future work.

II. RELATED WORKS

Some of the related research works addressing the low memory scenarios are briefly reviewed in this section.

Mauricio Lin et al. [1] proposed an approach for swapless embedded system memory management. There are two thresholds proposed, MAT and ST. When the memory requirement by the process exceeds ST, a signal to that process is sent telling it to try freeing some memory. The proposed approach in this paper is per process based threshold, while we are implementing a single threshold for whole system.

Rajesh Prodduturi et al. [2] has suggested an improvement in OOM killer for Android operating system. It prioritizes applications based on its past usage history, hence an application is likely not to be considered for killing if it has been used several times in the past. Along with this approach, the work also suggests fine tuning of the parameters related to OOM killer of Android for better performance.

K. Y. Sim et al. [3] have discussed a new optimal test method for fuzzing the out-of-memory killer. It is based on Adaptive Random Approach. This test method requires fever number of samples to reveal any flaws in the OOM. Fuzzing is the black box testing technique which is used for crashing any software under use.

Goldwyn Rodrigues [5] reviews the concept of OOM killer and discusses approach for moving the OOM responsibility to user space. In Linux, a user can change the value of specific system variable in order to change the priority of that process for the OOM killer function.

Jonathan Corbet [7] suggested some approaches for improving the OOM killer, which includes creating a common and global OOM notification mechanism, also making OOM functionality to be available to loadable modules for better control and new framework for describing policy to kernel.

So in conclusion many approaches were proposed for optimal handling of low memory conditions and OOM killer in Linux, most of them were reactive in nature and few were proactive in nature. But here we are proposing a new proactive way to tackle the problem using a single system wide threshold.

III. PROPOSED APPROACH

The issues identified in current out of memory killer can be summarized as follows:

1. OOM killer may eventually kill some important and innocent processes.
2. OOM killer uses system resources for its execution
3. OOM killer is too frequently called in embedded systems with low memory.
4. No opportunity is given to the application for recovery of itself on its own.
5. No past usage history of application is considered while deciding to priority for killing.

The last problem was touched by [2], explaining the approach of using a past usage based strategy for selecting bad process. The present work proposes an approach in which the system sends signals to currently running processes so that the processes can free some unused memory.

Usually when the OOM condition occurs the operating system calls the OOM killer to kill some of the low priority processes. A different approach was proposed in [1] for swap-less embedded systems. The swap-less embedded system does not have any swap area to save the swapped out pages, hence all its system memory is equal to physical memory which is available only in little quantity. In that method, a fixed threshold is decided for every process in the system and a system only can request for that much memory only. If any process asks for more memory than memory access threshold (MAT) then the process will be eligible for killing. But, there is another threshold called signal threshold (ST), slightly lower than MAT, when this ST is reached the application is given a signal indicating that it is reaching its limit of memory usage. In response to this signal, the application may start releasing some of the memory held by it.

The issue with approach in [1] is that it uses a per process based threshold and not all processes require the same amount of memory. Hence, choosing proper values of MAT and ST for processes is a problem. Our current approach in this paper uses a single system wide threshold which is not a per process based.

Hence, our proposed approach can be summarized as:

1. A specific threshold called *Danger Threshold* is decided for the system
2. Whenever the current memory usage goes beyond the Danger threshold, a *danger_signal* is given to each of the processes in the system, telling them to release some of the memory.

The following subsection gives more insight into our proposed approach and discusses it in detail.

A. *Approaches for handling the danger_signal*

The process after receiving the *danger_signal* must release some of its memory. This is possible because every application whenever asks for more memory also keeps the record of its usage and has the links to those memory locations. In most of the cases, the application requires memory to store a large number of independent inputs; for example, the image browser application will require memory to hold many independent images, web browser requires memory to hold each independent page opened in a tab. So, most applications have some independent small memory sections in use. Hence, an application can free some of these memory sections, but this will definitely affect their performance and responsiveness. However, this option is always better than the whole application getting killed by OOM killer. Hence, this is a trade off here. There can be some applications for which freeing the memory is not at all possible. For example, some computational program that may use extra memory but each of these memory may be needed for proper function of the application.

B. *Working of the improviser module*

For the implementation of our proposed approach, we are designing a kernel module called the *Low Memory Improviser module* which will be handling the logic. Whenever a process frees some memory the statistics about that event will be stored by this module in a specific data structure. This information will then help the OOM killer in future for selecting bad processes.

There can be some applications which may not be handling the *danger_signal* at all. So, we will first group all the applications based on the criteria of whether they handle the *danger_signal or not*, we will also consider the type of process (like kernel process , user process etc.) , its user set priority etc. This grouping of applications will make easy to manage the system.

C. *Use of past application usage history in OOM killer*

As proposed in [2], the frequency of an application usage is also an important factor while selecting a process to kill from the system. In the work [2], the author has proposed an approach where the OOM killer takes into consideration the particular applications past usages. For example, the gallery application is more often used than a calender application. But by default, the system gives same priority to both of the applications. So, in any situation where we have a choice in selecting either of these two applications for killing under Low Memory condition, we must never select the gallery application.

Many times, it is better to involve the actual user in deciding the bad process, because even the application history also can not tell the exact preference that the user might have at that time instance. For example, at some time , the calender may be more important to the user than the gallery. So, it will be great if we are able to implement some solution for this.

IV. SYSTEM DESIGN

The Fig.1 shows a very abstracted view of our proposed system. It is showing some processes running in a system. The process requests some memory by using system calls, but the memory fulfillment will be done only after consulting to our low Memory Improviser module. If the system is burdening on low memory then the Improviser module will send the Low on Memory (i.e. *danger_signal*) to all the processes.

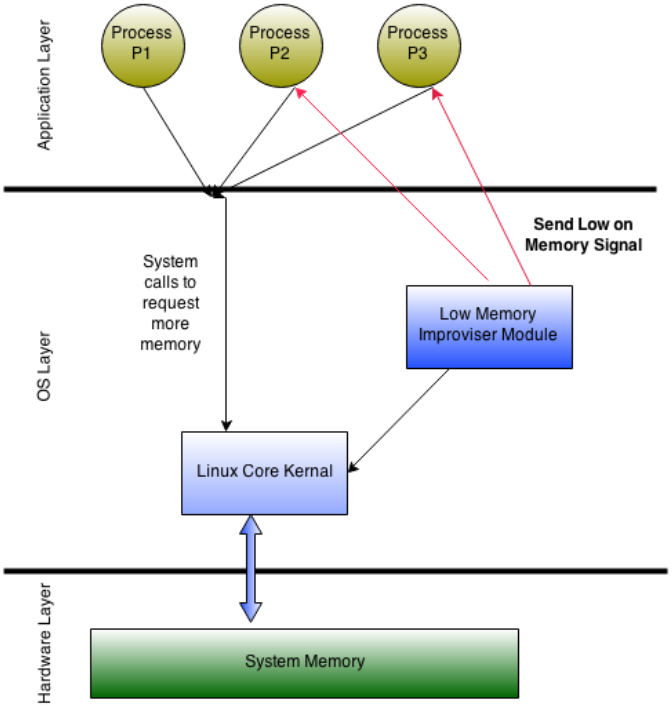

Fig. 1. System diagram showing low memory improviser interacting with processes and Kernel

The Low memory improviser Module is the only addition to the existing system. Also some modification in the kernel code is expected.

V. IMPROVED LOW MEMORY HANDLING ALGORITHM

The Fig. 2 shows the Flow Graph for the proposed system. Initially, the *danger_threshold* will be set by the system. Now, new memory requests are fulfilled till *the danger_threshold* is not crossed. Once it crosses, a system generated signal will be sent to all the processes in the system.

The processes in response to this signal will start freeing some memory which was allocated to it. At the same time the module will record the amount of memory freed by each process as Obeying Factor.

Later, the threshold will be checked again, and if the threshold is getting crossed then we have to compulsorily call the OOM killer. But, this time the OOM killer will use the Obeying Factor along with the application usage history for finding the candidate to be killed.

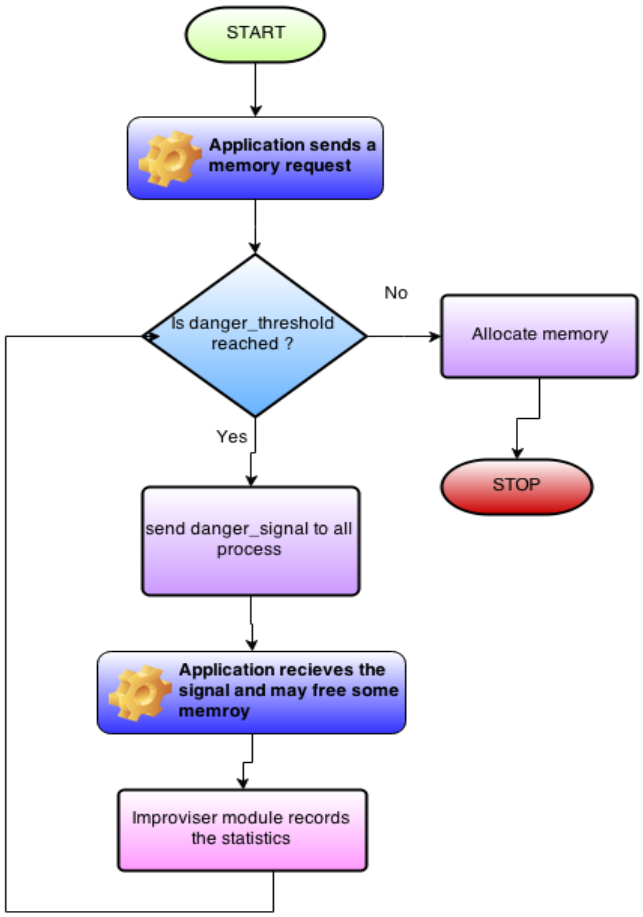

Fig. 2. Flow graph for memory request/ allocation

VI. PERFORMANCE METRICS

Performance of computer system depends on effective and efficient execution of the applications on the system. The system will be tested with some set of dummy processes say 50, that will try to eat off the memory. The system will also have some innocent process. With time OOM will occur and hence in that situation all the modules prescribed here will start functioning.

The number of times the OOM killer is called is also a performance metric, because it is a very resource intensive task. Also, how many processes were killed in the system for proper operation. Minimum number of processes to be killed is desirable. And, at final, the responsiveness of applications would also be considered as performance metric to calculate overall system efficiency.

VII. CONCLUSION

Applications can shrink and grow their memory usage dynamically depending on the current memory pressure on system. An improved algorithm for low memory handling in Linux is proposed in this paper. With the proposed approach we are providing a mechanism by which the applications would be informed dynamically when the system is low on memory. This avoids calling OOM killer frequently and hence improves the performance. The applications usage history will also be used when deciding its priority for killing by OOM.

Users or system administers do not have the ability to properly tune the OOM killer according to their demands. The tools that provide administers the control over OOM killer have very limited functionality. The future work may consider the involvement of the users in deciding the smarter OOM policies.

ACKNOWLEDGMENT

We would like to acknowledge Dr. Krishna Kumar, (former faculty, Department of computer Science and Engineering, NIT Calicut) for his help and support during this work.

REFERENCES

1. Mauricio Lin and Ville Medeiros , " Memory Management Approach for Swap-less Embedded Systems", ACM Linux journal Archives, Issue #140 Dec, 2005
2. Rajesh Prodduturi and Deepak Phatak , " Effective Handling of Low Memory scenarios in Android " , IIT Bombay Master Thesis, June 2013 , available at http://www.it.iitb.ac.in/frg/wiki/images/8/8c/113050076_Rajesh_Prodduturi_Stage-II_report.pdf
3. K. Y. Sim, F.-C. Kuo, and R. Merkel., "Fuzzing the out-of-memory killer on embedded Linux: an adaptive random approach. ", Proceedings of the 26th Annual ACM Symposium on Applied Computing (SAC 2011), TaiChung, Taiwan, 21-24 March 2011, pp. 387-392
4. "OOM Killer for embedded Linux System. " , Available at *http://www.mnis.fr/en/ support/ doc/ oomkiller/*
5. Goldwyn Rodrigues , " Taiming the OOM Killer" Feb. 2009 , Article #317814 Available at *http://lwn.net/Articles/317814/*
6. " Error Notification Support for SIGDANGER signal. " IBM Knowledge center, Available at *http://www01.ibm.com/support/knowledgecenter/SSPHQG_7.1.0/com.ibm.powerha.insgd/ha_install_error_sigdanger.htm*
7. Jonathan Corbet , " LSFMM : Improving the out-of-memory killer." , April 2013 , LSFMM Summit 2013, Article, Available at *http://lwn.net/Articles/548180/*
8. Corbet, " Respite from the OOM Killer.", Article, Sept. 2004 , Available at *http://lwn.net/Articles/104179/*
9. Mel Gorman, " Out of Memory Management. ", Book " Understanding the Linux Virtual Memory Manager" , Available at *https://www.kernel.org /doc/ gorman/html/understand/understand016.html*
10. *David Rientjes* , Source code for Out of memory management in Linux , oom_kill.c , Available at *http://lxr.free-electrons.com/source/mm/oom_kill.c*