An Adaptive Out-of-Memory Killer for Embedded Linux Systems

Katrina Siegfried, Dylan Fox, Ravindra Mangar
University of Colorado Boulder, Boulder, Colorado, USA

ABSTRACT

Embedded systems manifest themselves in a wide variety of forms, and many embedded systems have a limited amount of memory available to them. The Out-of-Memory (OOM) killer is a kernel process that aims to free memory when a system is over-allocated by terminating user processes. The current heuristic used by Linux's OOM killer is not suited for many possible use cases where embedded systems are deployed, including data-critical and safety-critical applications. The current algorithm may result in systems either becoming unresponsive or critical processes being killed when the system is over-subscribed. While a user can set a process to be "unkillable" by the OOM Killer, this may result in the system perpetually remaining unresponsive if an "unkillable" process is causing the memory pressure. Here we propose an alternative OOM killer that is deployment environment aware and can execute a user specified graceful failure procedure if memory pressure still exists after all "unkillable" processes are killed.

I. INTRODUCTION

One of the greatest limitations in embedded systems is system memory [5]. This necessitates that memory is appropriately managed, and out of memory conditions (OOM) are immediately addressed. The last line of defense in memory management by the Linux kernel is the OOM killer, a kernel process which seeks to identify the process causing the OOM condition and terminate it in order to preserve kernel function. Shortcomings of the current OOM killer can cause the Linux kernel to remain in the OOM condition and become non-responsive [1].

The OOM killer used by the Linux kernel for memory management assigns a score to each user space process. Scores are influenced by several factors including the proportion of RAM used, page table, swap space use, and the user set OOM adjusted score variable [4]. Processes with a higher score are more likely to be terminated when the system requires more memory. Linux provides a variable, called the OOM adjusted score, that can be adjusted to make a process less likely to be killed. If a user assigns a large enough negative value to the OOM adjusted score, the process will be "unkillable" by the OOM killer.

This behavior may not always be desired. There are many situations in which an abrupt termination of a process is unacceptable but a controlled shutdown may be acceptable. This is the case for data-critical and safety-critical systems where failures can have "significant and far-reaching consequences" [5]. For example, If a programmer expects data loss to occur if a specific process is killed by the OOM killer mid execution, they may set the process to be unkillable. However, this process might be perfectly acceptable to kill if the data is transmitted to a remote server or written to a disk first. In the case of safety critical processes, it may be vital to neither kill a critical process, nor end up in an unresponsive state. In these cases, our graceful failure option could allow the system to notify the user and shutdown the entire system in a safe manner.

II. RELATED WORK

Limits on memory available have caused problems for programmers since the dawn of the computer era. Most previous efforts to alleviate problems caused by running out of memory have focused on increasing the memory available or decreasing the memory utilization. The linux OOM killer takes the second approach, and kills processes till the memory is no longer overcommitted [3]. Killing arbitrary processes until enough memory is regained is obviously not the optimal approach for many situations.

In cloud computing, previous works have devised systems to add memory on demand or move processes to remote nodes, reducing the memory pressure on the current node. VM migration allows the migration of a running VM to a different physical machine with more memory [11]. Nswap and other networked memory technologies allow the memory space to be extended across a network to remote nodes, enabling a single node to push memory pages to remote machines and avoid calls to the OOM killer [13]. However, VM migration and networked memory simply are not options for many embedded systems. Few embedded systems are connected to a network of other physical machines that they can utilize and latency critical applications may not tolerate the network latency caused by moving to a different physical node connected via a network.

Other works in cloud computing have focused on seamlessly adding disk swap space to nodes that are overcommitted. CUDSwap allows a node to add more swap space (on dsk) to a node with overcommitted memory, avoiding calls to the OOM killer [12]. While CUDSwap may be suitable for an embedded system with connection to a large, fast hard drive, it would not be suitable for a system with a small or particularly slow hard drive. CUDSwap also generally suffers a performance hit compared to using memory alone, so it may be unsuitable for latency critical applications.

On the Android OS, Kim et al developed a procedure for killing processes not based on their OOM score, but rather based on the likelihood a user would want to utilize the process [14]. This system collects data about the user usage of different processes and kills the process with the least usage time [14]. However, this procedure may have deleterious effects if applied to embedded systems. Embedded systems tend to have their hardware relatively well tuned to the common level of resource utilization for the system, and are less likely to experience spikes in memory demand when compared to a mobile phone. In the embedded domain, memory pressure due to a programmer’s mistakes, or a fundamental mismatch of hardware and load are more common. One could imagine a situation in which a memory leak occurred on a relatively frequently used process, and if fed through the algorithm proposed by

Kim et al, would result in all of the less frequently used processes being killed before reaching the process causing the memory pressure.

In the realm of embedded systems, work has been done to decrease the time spent choosing which process to kill when out of memory but we are unaware of any attempts to add graceful failure procedures to the OOM killer [2].

IV. UNGRACEFUL HANDLING OF OUT OF MEMORY CONDITIONS

Limited memory in embedded systems and the additional demands of data-critical and safety-critical systems necessitates that memory is appropriately managed, and OOM conditions immediately addressed. The risks of mishandling these memory errors is twofold – data can be lost, and the system can be left in an unsafe or unsatisfactory state as a result. The standard linux implementations for handling these errors do not effectively address these risks in the data-critical and safety-critical systems.

How Out of Memory Errors Occur

There are several ways in which an out-of-memory, or OOM, condition can occur. Most commonly OOM conditions are caused by overcommits, dynamic memory allocations, and inability to use swap space [6].

Linux will allow, by default, a process to allocate more memory than is physically available, as most processes allocate far more than they will ever use, referred to as overcommitting [7]. If processes cache in on their full memory allocations simultaneously an OOM condition will occur. This can be somewhat mitigated by reducing the size of requested memory allocations by programs (common in embedded) and by adjusting the overcommit ratio to reduce the amount of overcommitting the system will permit in /proc/sys/vm/overcommit_ratio [7].

Dynamic memory allocations can be unpredictable, and are typically discouraged in real-time (RT) embedded systems. At compile time, the pathway a particular program will follow in a runtime exception handling is not defined. If the amount of memory allocated in handling runtime exceptions exceeds the limited heap memory available, an OOM condition can result. In real-time systems dynamic memory use is often discouraged or not used at all [8].

Swap space is used as an extension of physical memory, or RAM. A process's memory can be written out to swap space in order to free up more physical memory temporarily. If the writing of memory requires additional allocations, such as for I/O, that is not available the system may deadlock, unable to invoke the write process that would free up physical memory, causing an OOM condition [6]. Additionally if a process requires writing larger pages of memory than the swap space would permit, the system is unable to utilize it and will fall into an OOM condition when enough physical memory is unavailable [6].

Default System Handling of Out of Memory Errors

When proactive mitigations fail to prevent an OOM condition, Linux invokes the OOM killer to free up memory on the system. This kernel process uses a heuristic to assign a score to all running user processes that is a function of the proportion of physical memory allocation, use of swap space, and use of the page table [3]. Scores range from 0 to 1000 for each process, where 0 is least likely to be killed and 1000 is most likely to be killed. Kernel processes are marked with a score of -1000 which essentially makes the process as unkillable [5]. Users can override the scores for a specific process by adjusting the value in /proc/<PID>/oom_score_adj.

The intent is to select the process that is the most active in the memory with the assumption that is the source of the out of memory condition. This is often, but not always true, and can result in multiple processes being killed before the out of memory condition is finally resolved. Further, this behavior does not guarantee to leave the system in any particular state, but it does aim to leave the kernel operational [7]. Depending on the constraints of the application, this would be considered device failure.

Memory Error Induced Data Loss

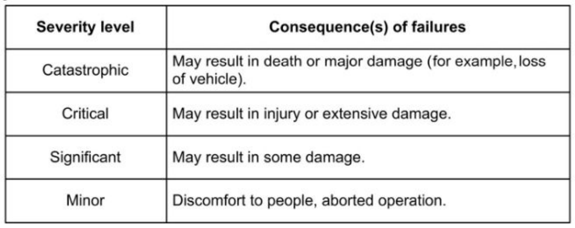
Data critical applications demand that the data being generated is successfully written to disk. If a process responsible for writing this data is hosting it in an in-process buffer and encounters an OOM condition any data in a buffer or cache for that process that has not yet been written to the disk will be lost if the process is killed [9]. If this data loss is unacceptable the system is considered to have failed.

Memory Error Induced System Failure

If a process is killed in a safety-critical system there are three options in responding to unrectified system faults – the system is fault tolerant and can continue operation, the system can continue operation with reduced service, or the system can be placed in a fail-safe state [7]. The default OOM killer behavior will result in a process being killed off without regard to desired system operation, which can leave the device in a sub-operational, and therefore failed state.

Risk Assessment of Out Of Memory Errors

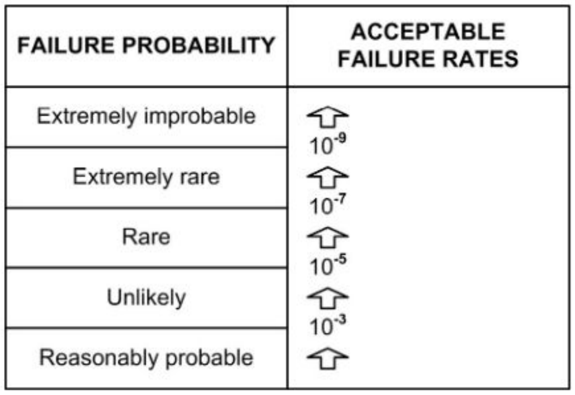
Given the available options to mitigate OOM conditions in embedded Linux operating systems, when used in combination with careful and thorough software development the occurrence of OOM errors can be mitigated. The acceptance level for the frequency of these types of errors will vary from application to application. In data-critical systems or safety critical systems these acceptance levels will generally be extremely low. When these errors do occur they need to be handled in a predictable way that leaves the device and the user(s) in a safe state.



Severity level	Consequence(s) of failures
Catastrophic	May result in death or major damage (for example, loss of vehicle).
Critical	May result in injury or extensive damage.
Significant	May result in some damage.
Minor	Discomfort to people, aborted operation.

Defining various levels of severity used in defining critical systems, borrowed from Jim Cooling's book Software Engineering for Real Time Systems [7].

For example, the threshold for acceptable loss of data from a military submarine radar is extremely low; loss of any data is extremely undesirable and can compromise the success of the mission and well-being of the sailors on-board [7]. In an airbag deployment system, the embedded device becoming unresponsive unknown to the user could result in catastrophic results if it failed to deploy when needed [7]. A suggested acceptable rate of occurrence for various applications is listed below.



FAILURE PROBABILITY	ACCEPTABLE FAILURE RATES
Extremely improbable	⇧ 10^{-9}
Extremely rare	⇧ 10^{-7}
Rare	⇧ 10^{-5}
Unlikely	⇧ 10^{-3}
Reasonably probable	⇧

Failure probability and associated failure rates, borrowed from Jim Cooling's book Software Engineering for Real Time Systems [7].

With a demand for such low occurrence rates for OOM errors in real time and near real-time systems, and a requirement to appropriately handle the errors when they do occur, a need exists for improved handling of OOM errors in embedded data critical and safety critical applications and systems.

V. DESIGN

A. Predicting Low Memory Scenarios

Amongst other factors, low memory allows a system to become unresponsive or induce unfavorable conditions such as a kernel panic or thrashing. Supplementing the problem of unresponsiveness is the tendency of the Linux kernel to overcommit memory to processes.

In several embedded systems, especially RT systems, the affordability of unresponsiveness is extremely low. As such we seek to design a preemptive mechanism which can predict when a system would enter a state of unresponsiveness due to a low memory issue. Such mechanism aims to leverage off the fact that many embedded systems engage in some form of redundant task.

Our proposed mechanism would collect relevant system information such as oom_scores, net memory usage statistics, and CPU load amongst others. Such features would be used to generate some heuristic which would be able to anticipate when a system may undergo issues relating to low memory such as unresponsiveness or a kernel panic.

Essentially, the process seeks to identify abnormal events in memory as well as the overall system which may enact undesirable behavior. The distribution of memory as described in the /proc/meminfo file, provides possible data sources which can be further explored. Additionally, tracing the system calls when a kernel panic is induced can allude to factors which may lead to out of memory problems.

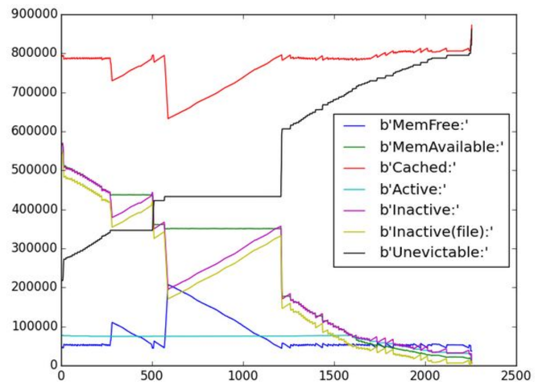
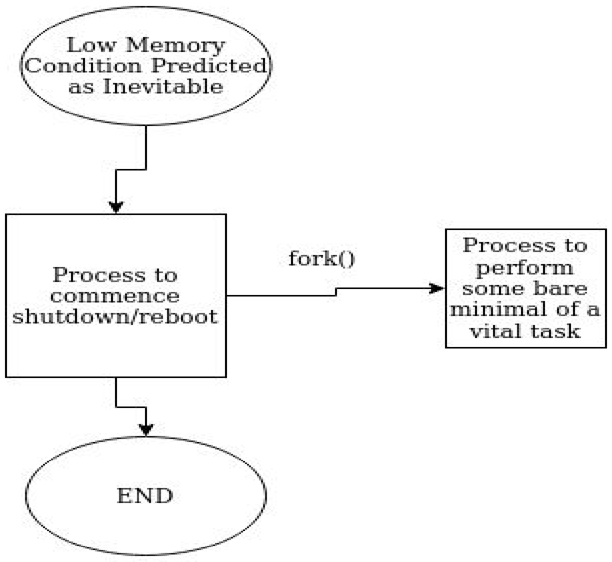


Diagram above depicts a graphical log of /proc/meminfo [10]

B. Handling Process Shutdown & Preemptive Data Loss

In order to keep data loss to a minimum we induce graceful shutdown via two processes. A parent process which commences the shutdown process and a child process which performs some bare minimal task which can be classified as highly vital. Such a task may involve writing data to a file which can then be processed or backfilled upon resumption of normal operating conditions for the system. A process diagram of our technique can be seen in the ensuing diagram.


Low Memory Condition Predicted as Inevitable
Process to commence shutdown/reboot
fork()
Process to perform some bare minimal of a vital task
END

C. Evaluation

We first plan to artificially create out of memory conditions to force a process to enter the graceful failure state. This can be accomplished simply by writing an endless loop of memory requests. This will allow us to ensure the graceful failure state works as expected.

Secondly, we plan on analyzing the performance of our enhanced OOM killer to ensure it does not degrade performance of the kernel when implemented. We will use industry standard tools for benchmarking embedded systems, like Cyclictest (rt-tests), to ensure performance is not degraded with the new OOM killer [4].

References

[1] Sim, K. Y., Kuo, F., & Merkel, R. (2011). Fuzzing the out-of-memory killer on embedded Linux. *Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11*. doi:10.1145/1982185.1982268

[2] Kook, J., Hong, S., Lee, W., Jae, E. (2016).Optimization out of memory killer for embedded Linux environments. *Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11*. . doi:10.1145/1982185.1982324

[3] *Linux Source Code*, Retrieved from: https://github.com/torvalds/linux/blob/master/mm/oom_kill.c

[4] Reghenzani, F., Massari, G., Fornaciari, W. (2019) The Real-Time Linux Kernel: A Survey on PREEMT_RT. *ACM Compt. Surv.* 52, 1 Article 18 (February 2019), doi: 10.1145/3297714

[5] Vaduva, Alexander, et al. *Linux: Embedded Development*. Pact Publishing, 2016.

[6] Wiki written by linux developers on the memory management mailing list https://linux-mm.org/OOM

[7] Cooling, Jim. *The Complete Edition - Software Engineering for Real-Time Systems: a Software Engineering Perspective toward Designing Real-Time Systems*. Packt Publishing, 2019.

[8] Electronics Media site run by division of Arrow Electronics https://www.embedded.com/safety-critical-operating-systems/

[9] Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2006.

[10]Bvolkmer, Out of Memory with Unused Memory, Unix Stackexchange, Retrieved from: https://unix.stackexchange.com/questions/194225/out-of-memory-with-unused-memory-and-swap

[11] Choudhary, A., Govil, M., Singh, G. et al. A critical survey of live virtual machine migration techniques. J Cloud Comp 6, 23 (2017). https://doi.org/10.1186/s13677-017-0092-1

[12] J. A. Navas-Molina and S. Mishra, "CUDSwap: Tolerating Memory Exhaustion Failures in Cloud Computing," 2014 International Conference on Cloud and Autonomic Computing, London, 2014, pp. 15-24, doi: 10.1109/ICCAC.2014.12.

[13] T. Newhall, et al. “Nswap: A Network Swapping Module for Linux Clusters,” International Conference on Parallel and Distributed Computing, 2003.

[14] J. Kim, J. Sung, S. Hwang and H.-J. Suh, "A Novel Android Memory Management Policy Focused on Periodic Habits of a User" in Ubiquitous Computing Application and Wireless Sensor, Netherlands:Springer, vol. 331, pp. 143-149, 2015.

[*] https://elinux.org/Memory_Management – good related works

** Agreed to use 18.04 LTS with kernel 5.4 for development and testing in VM