*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 process that aims to free memory when a system is over-allocated by terminating processes. The current heuristic used by Linux’s OOM killer is not suited for many possible use cases where embedded systems are deployed. The current algorithm may result in safety critical systems either becoming unresponsive or 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.

**Introduction**

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 Linux 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. If a user sets a large negative value to the OOM adjusted score variable at process initiation, this process will be very unlikely or impossible to kill. 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. 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.

Previous efforts have been made to optimize the response time of the OOM killer [2]. However, for real time systems and especially safety-critical systems simply optimizing the time necessary to pick a process to kill may not be enough.

**Evaluation**

We plan to analyse the performance of the Linux’s current OOM killer amongst a variety of use cases, ranging from sensor equipement to safety-critical systems.

Industry standard tools for benchmarking embedded systems include Cyclictest (rt-tests), and Tracers could be used to ensure performance is not degraded when the OOM killer is changed to allow graceful failures [4]. In order to test the OOM killer, we will need to introduce memory pressure to the system. Fuzzing various sizes of memory requests in an endless loop has been shown to create conditions in which the OOM killer will become active [1]. We may need to add on to this scheme to create conditions in which multiple processes are killed sequentially, avoiding simply killing the single test driver requesting memory.

**IV. Lack of Graceful Shutdown Procedures in Embedded OS**

One of the greatest limitations in embedded systems is system memory [5]. This necessitates that memory is appropriately managed, and out of memory 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 mitigate these risks in data-critical and safety-critical systems where failures can have “significant and far-reaching consequences” [5].

***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 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 hard real-time systems dynamic memory use is often discouraged or not used at all [8].

Swap space is used as an extension of physical RAM memory can be written out to swap space in order to free up more memory temporarily. If the writing of memory requires additional allocations 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 pagetable [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 marks 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, which can result in multiple processes being killed before the out of memory condition is finally resolved. This behavior does not guarantee to leave the system in any particular state, but it does aim to leave the kernel operational [7].

***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 out to memory will be lost if the process is killed [9]. If this data loss is unacceptable the system is considered to have failed.

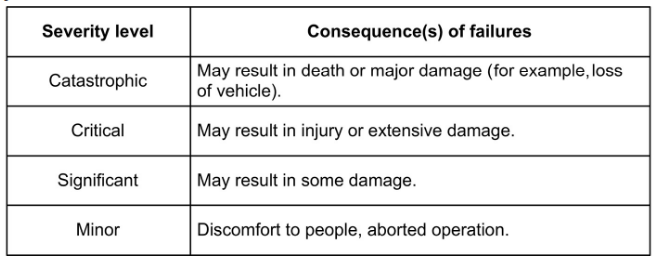
***Memory Error Induced System Failure***

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 regards to desired system operation.

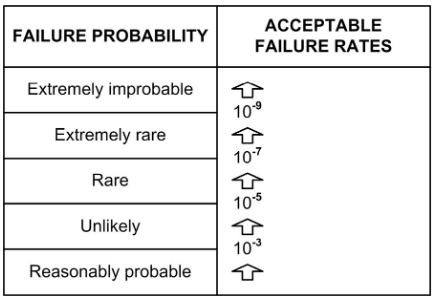
***Risk Assessment of Out Of Memory Errors***

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 way that leaves the device and the user(s) in a safe state.

For example, the threshold for acceptable loss of data from a military reconnaissance radar is extremely low; loss of any data is extremely undesirable and can compromise the success of the mission [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.

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

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

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

**III. Related Work and Background [DYLAN] - massage what we have and incorporate the paper shiv shared with us**

**IV. Description of Topic [KATRINA]**

**-----------------------------------------------------**

**V. Design**

**A. Preempting Data Loss [RAVI]**

**B. Handling Process / Device Shutdown**

**Preliminary Design**

1. **How to associate graceful fail safe - spawn another process to handle shutdown? Does it need access to the dying process? Best if done at user level if possible? [RAVI]**
2. **Memory allocation, do a check before each new memory allocation? Performance of adaptive checking vs. start with a standard check [KATRINA]**
3. **Reliably creating OOM memory leaks (Fuzzing paper) [DYLAN]**
4. **Use of Traces or Cyclictest (rt-tests) to benchmark our system, optimization could be to grow out list of benchmarks [DYLAN]**

**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] <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] <https://www.embedded.com/safety-critical-operating-systems/>

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

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

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