User-Space Out-of-Memory(OOM) Killer

Sunny Wadkar
Virginia Polytechnic Institute and State University

1 Introduction to the Problem

Linux's Out-of-Memory(OOM) Killer has always been the topic of discussion in multiple forums. It is one of the most controversial subsystems of Linux which has lead to some contradicting views on how it should be designed. There are still many ongoing discussions on memory accounting of the kernel, the kernel's allowance of over-commitment of memory and how the kernel should behave in the situation of out of memory.

Physical memory(RAM) is a precious resource in the computer systems. In embedded systems which can be swap-less at times, it is even more critical resource and needs to be used judiciously. In the current Linux kernel, over-commitment of the memory to a process is possible and controlled by `overcommit_memory`. Memory overcommit is based on the assumption that not all assigned memory is needed by running applications. This means that kernel can guarantee a process, the amount of memory which it cannot provide. On the memory overcommitted system, malloc and other memory allocators usually never fail. However, if an application dereferences the returned pointer and the system has run out of the memory, then the kernel is forced to call upon the OOM Killer. This is the situation in which either the system on a whole can progress by killing some processes or the system will crash due to lack of memory.

While the OOM killer is seldom called on desktop environments which are equipped with large physical memories, it is frequently called on systems with low memories such as Android. The OOM killer kills one or more processes based on system memory heuristics that it maintains. The OOM killer makes its best effort to kill a 'good' process so that system can progress without crashing. The good process with respect to OOM is the process which can be killed to free maximum amount of memory with minimal or no damage to the system as a whole [2]. Killing the processes can sometimes be a slow task since kernel can take an unbounded amount of time reclaiming the memory from the dying processes. This can make the system unresponsive when OOM operations are in progress. Linux OOM killer also has the tendency to kick in too late resulting in system entering the livelock for an indeterminate and unresponsive state.

The current OOM implementation is that it's kind of unpredictable [4]. OOM Killer might choose an unexpected process that is important to the user as a victim to sacrifice without any notification to the user-space application. The Linux's OOM Killer primarily focuses of kernel's concern and its heuristics does not take into consideration some of new statistical tools such as Pressure Stall Information [6]. Besides this, the current kernel does not declare the OOM condition as long as reclaimable pages exist [3]. The problem is that there are no real bounds on how long it might take for "reclaimable" pages to actually be reclaimed, for a number of reasons. Additionally, the memory allocator can conceivably find itself endlessly retrying if a single page is reclaimed, even if that page cannot be used for the current allocation request. As a result, the kernel can find itself hung up in allocation attempts that do not succeed, but which do not push the system into OOM handling. Hence, the heuristics employed by the kernel in detecting OOM situations are not reliable in all of the cases.

From the above discussion, we can see that the problem of default Linux Kernel OOM mechanism is that it kicks in too late to kill a process when the system is under memory pressure, resulting in unresponsive behavior. Its another problem is that the victim selection process is unpredictable and does not consider some of the recent memory metrics.

2 Motivation

As stated above, the victim selection process of the OOM is quite unpredictable at times. It is difficult to determine which processes will be selected by the OOM in the user-space. This is especially evident in server-based environments where high memory demand user-space processes are running concurrently. Once memory pressure reaches the maximum limit, then kernel OOM is invoked to kill the process with the highest `oom_score`. The OOM also kills the children processes of the victim processes. The OOM prefers to kill user-space processes over the kernel space processes to maintain the stability of the system. This could result in killing of applications necessary for the user which are otherwise deemed insignificant by the system. The problem is also relevant on embedded systems with low memories and/or without swap memories such as smartphone operating systems. Since these systems don't have a swap, high memory pressure situations invoke OOM killer more frequently clearing out background processes. If the reclaimed memory from the background user processes is still insufficient for the system, then foreground user processes are also terminated thus causing user inconvenience. Since the killing starts too late, it results in the system becoming unresponsive when the system memory pressure reaches critical thresholds and thus hindering the user-experience.

There have been numerous discussions about moving OOM handling to user-space in various Linux forums. Some have questioned about the reliability of the user-space OOM killer and whether it would be able to accomplish the task when the situation arises. Mel Gorman, speaking in the LSFMM Summit 2013, has provided following directions for the kernel to facilitate user-space OOM [1]:

1. Create a global OOM notification mechanism that can be used by processes like the Android low-memory killer.

2. Create some sort of internal OOM hook in the kernel that would be available to loadable modules or, perhaps, SystemTap scripts. Administrators could then load whatever policy suited them best.

3. Add a framework by which a policy could be described to the kernel, similar to how firewall rules or packet filters are handled in the network stack.

Similar mechanism exists in MacOS/iOS where a thread continuously monitors the memory pressure of the system and kills the processes based on priority. The Jetsam also kills the process which exceeds its memory consumption. Prior to killing a process, however, the OOM killer does allow a process to "redeem itself", and avoid untimely termination, by getting the OOM thread to first send a kernel note to processes which are "candidates" for termination [5].

This project aims to solve the problem of the system going into livelock state due to fact of kernel OOM killer kicking in too late. This project also aims to solve the problem of the unpredictability in the OOM victim selection process with the intervention from user-space.

3 Proposed Solution for the Problem

The above stated problem can be solved by shifting the OOM killing policy to user space. This would allow the users to decide on the important processes that would be exempted by the OOM mechanism. It would also allow the system administrators to deploy their custom OOM policies depending on the workloads.

In my solution, the kernel will provide a mechanism to send a low memory signal(LMS) to the possible user-space OOM daemon. On receiving the LMS signal, the user-space OOM daemon will select a victim process based on its heuristics and victim selection policies. The user-space daemon will then kill a user-space victim to reduce the memory pressure. The user-space daemon will also provide a notification to the victim user-space process about the memory pressure. This will also provide the latter a chance to either help relieving some memory for system stability or save the process state before it is killed for the future use.

The user-space OOM can deploy its own heuristics to find a victim thus providing a flexibility in choosing the victim. This user-space OOM killer will work in conjunction with kernel-level OOM killer. The LMS signal sent by the kernel will be received by user-space OOM killer. The user-space OOM heuristics can take into consideration a user-defined policy for the victim selection. This policy can be created by the user based on his/her requirements. Adjusting the policy as per the requirements can make victim selection process more predictable and deterministic. One of the policy can be group exclusion policy where a group of processes are prevented from OOM killer and other less important user-space processes are sacrificed. If the user-space OOM is unable to reclaim any memory in a specified time-period, then the kernel level OOM can go ahead and kill the victim process according to its heuristics. The heuristics of the kernel level OOM killer can be adjusted further to select the victim more appropriately when user-level OOM daemon fails.

4 System Design

A block diagram of the proposed system is presented in Figure 1. The Memory Heuristics Kernel Module requires the

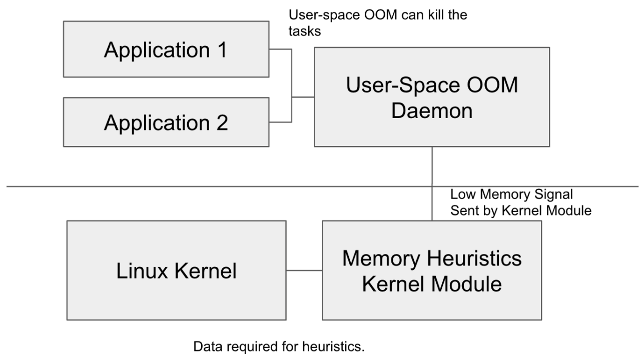

User-space OOM can kill the tasks
Application 1
Application 2
User-Space OOM Daemon
Low Memory Signal Sent by Kernel Module
Linux Kernel
Memory Heuristics Kernel Module
Data required for heuristics.

Figure 1: Block Diagram for User-Level OOM

following data from the kernel.

1. Total System Memory - available from `totalram_pages()`
2. Available System Memory - available from `si_mem_available()`
3. Total Swap Memory - available from `si_swapinfo()`
4. Free Swap Memory - available from `si_swapinfo()`
5. Page Size - available from `PAGE_SIZE` macro

The kernel module constantly monitors memory levels and sends a signal to user-space OOM daemon when it detects memory pressure. The user-space OOM daemon polls for the data sent by kernel module. The user-space daemon also monitors pressure stall information to detect memory pressure by itself. The pressure stall information is available from `/proc/pressure/memory` file. The Pressure Stall Information(PSI) feature identifies and quantifies the disruptions caused by resource crunches and the time impact it has on complex workloads or even entire systems [6]. PSI provides statistics for cpu, io and memory. For the memory statistics, the '`some`' part indicates the time when some resources are stalled while the '`full`' part indicates the time share when all of the resources are stalled. It can be used to indicate a thrashing scenario where system on whole is not making any progress due to constant page reclamation. Integrating the PSI metrics in the OOM heuristics can help in detecting the OOM conditions early.

Resident Set Size(VmRSS) is the indication of memory held by the process in the physical memory. This is also used in selection of the victim process for the OOM. This can also be used to detect if the kernel has overcommited the memory for a particular process. The user-space OOM daemon will use `VmRSS` along with `oom_score` to select the victim process. The resident set size is available from `/proc/pid/statm` file. If `VmRSS` is to obtained in kernel module, then `get_mm_rss()` function can be used.

The user-space daemon uses `oom_score` and `VmRSS` to select the victim process. It then sends `SIGTERM` signal to the victim and waits for a bounded amount of time for the process to respond to `SIGTERM`. If the victim does not drop the memory after receiving `SIGTERM`, it proceeds to send `SIGKILL` signal. The user-space OOM daemon is not allowed to kernel processes or the `init` process for the system stability purposes.

5 Implementation

The source code for the proposed design is available at https://github.com/SunnyWadkar/User-Space-OOM. The implementation involves two subsystems.

1. `memory_info` kernel module
2. `user_space_kill` daemon

The `memory_info` kernel module is created to detect the OOM conditions early. It takes into account all of the data mentioned above. It constantly runs in background and monitors these parameters to check whether they are in an allowable range. In the `memory_info` kernel module, a `kthread` is created to monitor available memory and available swap memory. This thread monitors the memory levels at every 500ms. The module also defines thresholds for the memory to notify user-space. Two thresholds are defined as follows:

1. `KILL_THRESHOLD` - When memory reaches below this threshold, a process has to be killed.
2. `WARN_THRESHOLD` - When memory reaches below this threshold but is above KILL_THRESHOLD, a process is to be warned to take care of its allocated memory.

The kernel module creates a `/proc/oom_notifier` entry to communicate with the user-space. It updates the entry periodically based on the available memory and available swap memory. The kthread is killed when the module is removed and it is started when the module is inserted. The module outputs the available memory percentage and available swap memory to the kernel log and can be viewed with `dmesg`.

The user-space daemon continuously scans the `/proc/oom_notifier` file to detect any memory pressure.If the memory level reaches below any one of the thresholds stated above, it finds a victim user-space process to kill. For this, it scan the whole `/proc` directory for the process information. It extracts `oom_score`, `oom_score_adj` and `VmRSS` for each of the process. The process with largest `oom_score` and largest `VmRSS` is selected for the sacrifice. If `VmRSS` is 0 for some process, it indicates that the process is a kernel process and it is not considered in victim selection process. It also parses `/proc/pid/stat` for each of the process to determine if the process is active or not. The daemon first sends a `SIGTERM` signal using the `kill()` system call and then after some time, it sends `SIGKILL`.

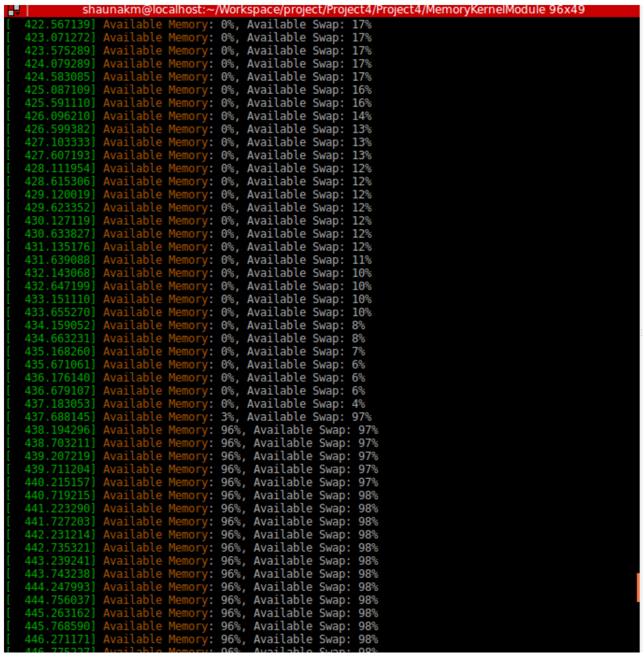
6 Evaluation

In this section, we will evaluate the proposed design on the basis of following questions.

1. Does the user-space OOM daemon kills the right process?
2. Does the user-space OOM daemon selects the right victim process?
3. Does the user-space application receive the `SIGTERM` signal before getting killed?
4. Does the intended user-space application gets killed?

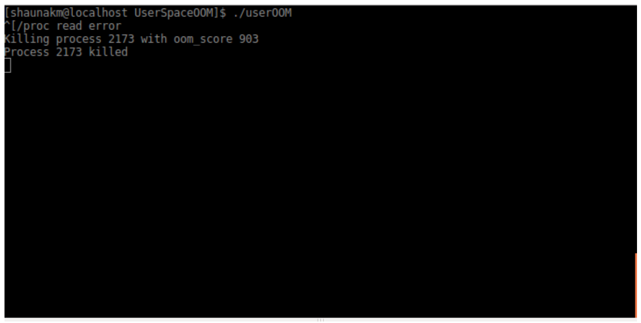
5. Does the user-space OOM daemon kick in at right time to prevent the system from invoking kernel OOM?

6. Does the kernel module monitor the memory levels properly?



```
shaunakm@localhost:~/Workspace/project/Project4/Project4/MemoryKernelModule 96x49
[  422.567139] Available Memory: 0%, Available Swap: 17%
[  423.071272] Available Memory: 0%, Available Swap: 17%
[  423.575289] Available Memory: 0%, Available Swap: 17%
[  424.079289] Available Memory: 0%, Available Swap: 17%
[  424.583085] Available Memory: 0%, Available Swap: 17%
[  425.087109] Available Memory: 0%, Available Swap: 16%
[  425.591110] Available Memory: 0%, Available Swap: 16%
[  426.096210] Available Memory: 0%, Available Swap: 14%
[  426.599382] Available Memory: 0%, Available Swap: 13%
[  427.103333] Available Memory: 0%, Available Swap: 13%
[  427.607193] Available Memory: 0%, Available Swap: 13%
[  428.111954] Available Memory: 0%, Available Swap: 12%
[  428.615306] Available Memory: 0%, Available Swap: 12%
[  429.120019] Available Memory: 0%, Available Swap: 12%
[  429.623352] Available Memory: 0%, Available Swap: 12%
[  430.127119] Available Memory: 0%, Available Swap: 12%
[  430.633827] Available Memory: 0%, Available Swap: 12%
[  431.135176] Available Memory: 0%, Available Swap: 12%
[  431.639088] Available Memory: 0%, Available Swap: 11%
[  432.143068] Available Memory: 0%, Available Swap: 10%
[  432.647199] Available Memory: 0%, Available Swap: 10%
[  433.151110] Available Memory: 0%, Available Swap: 10%
[  433.655270] Available Memory: 0%, Available Swap: 10%
[  434.159052] Available Memory: 0%, Available Swap: 8%
[  434.663231] Available Memory: 0%, Available Swap: 8%
[  435.168260] Available Memory: 0%, Available Swap: 7%
[  435.671061] Available Memory: 0%, Available Swap: 6%
[  436.176140] Available Memory: 0%, Available Swap: 6%
[  436.679107] Available Memory: 0%, Available Swap: 6%
[  437.183053] Available Memory: 0%, Available Swap: 4%
[  437.688145] Available Memory: 3%, Available Swap: 97%
[  438.194296] Available Memory: 96%, Available Swap: 97%
[  438.703211] Available Memory: 96%, Available Swap: 97%
[  439.207219] Available Memory: 96%, Available Swap: 97%
[  439.711204] Available Memory: 96%, Available Swap: 97%
[  440.215157] Available Memory: 96%, Available Swap: 97%
[  440.719215] Available Memory: 96%, Available Swap: 98%
[  441.223290] Available Memory: 96%, Available Swap: 98%
[  441.727203] Available Memory: 96%, Available Swap: 98%
[  442.231214] Available Memory: 96%, Available Swap: 98%
[  442.735321] Available Memory: 96%, Available Swap: 98%
[  443.239241] Available Memory: 96%, Available Swap: 98%
[  443.743238] Available Memory: 96%, Available Swap: 98%
[  444.247993] Available Memory: 96%, Available Swap: 98%
[  444.756037] Available Memory: 96%, Available Swap: 98%
[  445.263162] Available Memory: 96%, Available Swap: 98%
[  445.768590] Available Memory: 96%, Available Swap: 98%
[  446.271171] Available Memory: 96%, Available Swap: 98%
[  446.775227] Available Memory: 96%, Available Swap: 98%
```

Figure 2: memory_info module output

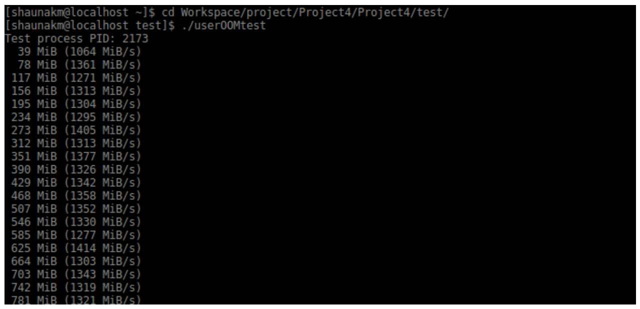


```
[shaunakm@localhost UserSpaceOOM]$ ./userOOM
^[/proc read error
Killing process 2173 with oom_score 903
Process 2173 killed
```

Figure 3: user_space_kill output

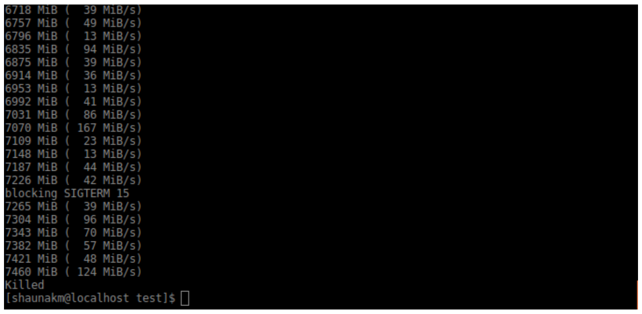
6.1 Experimental Setup

To test the proposed design, a situation of memory pressure on the system was created. For that purpose, a process was created which keeps on allocating memory in an infinite loop. The process allocated 10 pages in a single iteration and wrote a garbage value to those pages so that kernel actually allocated those pages in the main memory. This memory consumer process was also made to ignore the `SIGTERM` signal so that it kept allocating memory after receiving a warning. The `WARN_THRESHOLD` was kept as 20% and `KILL_THRESHOLD` was kept at 10%. The PID of the test process was noted to check whether the user OOM daemon killed the right process.



```
[shaunakm@localhost ~]$ cd Workspace/project/Project4/Project4/test/
[shaunakm@localhost test]$ ./userOOMtest
Test process PID: 2173
  39 MiB (1064 MiB/s)
  78 MiB (1361 MiB/s)
 117 MiB (1271 MiB/s)
 156 MiB (1313 MiB/s)
 195 MiB (1304 MiB/s)
 234 MiB (1295 MiB/s)
 273 MiB (1405 MiB/s)
 312 MiB (1313 MiB/s)
 351 MiB (1377 MiB/s)
 390 MiB (1326 MiB/s)
 429 MiB (1342 MiB/s)
 468 MiB (1358 MiB/s)
 507 MiB (1352 MiB/s)
 546 MiB (1330 MiB/s)
 585 MiB (1277 MiB/s)
 625 MiB (1414 MiB/s)
 664 MiB (1303 MiB/s)
 703 MiB (1343 MiB/s)
 742 MiB (1319 MiB/s)
 781 MiB (1321 MiB/s)
```

Figure 4: memory consumer process output 1



```
6718 MiB (  39 MiB/s)
6757 MiB (  49 MiB/s)
6796 MiB (  13 MiB/s)
6835 MiB (  94 MiB/s)
6875 MiB (  39 MiB/s)
6914 MiB (  36 MiB/s)
6953 MiB (  13 MiB/s)
6992 MiB (  41 MiB/s)
7031 MiB (  86 MiB/s)
7070 MiB ( 167 MiB/s)
7109 MiB (  23 MiB/s)
7148 MiB (  13 MiB/s)
7187 MiB (  44 MiB/s)
7226 MiB (  42 MiB/s)
blocking SIGTERM 15
7265 MiB (  39 MiB/s)
7304 MiB (  96 MiB/s)
7343 MiB (  70 MiB/s)
7382 MiB (  57 MiB/s)
7421 MiB (  48 MiB/s)
7460 MiB ( 124 MiB/s)
Killed
[shaunakm@localhost test]$ 
```

Figure 5: memory consumer process output 2

6.2 Results

The kernel module produced the output shown in figure 2 in `dmesg`. It shows that when the memory consumer process was running, the available memory and available swap size was decreasing. When both the memories reached below threshold, a victim process was killed to regain the memory and keep system from crashing. The memory levels increased after the process was killed. This also shows that kernel OOM was invoked and the system remained stable. Figure 2 also shows that the module is reporting memory changes correctly. Figure 3 shows the output of `user_space_kill` daemon when it killed the process with PID 2173. Figure 4 shows the output of memory consumer process where it displays its PID as 2173. This confirms that the user-space OOM daemon calculated a right victim process and also killed a right process. Finally, figure 5 shows that the memory consumer process blocked the `SIGTERM` signal and got killed after some time receiving `SIGKILL`. The overall experiment shows that the user-space OOM daemon kicked in at right time to kill the victim process and to prevent kernel OOM invocation.

7 Conclusion

Linux's Out of Memory Killer has always been a divisive component due to the fact that it's not deterministic in selecting victim process and it is invoked very late in the time of memory crunch. In this project, I propose a design of user-space OOM killer that works in conjunction with the kernel OOM killer. The user-space OOM killer continuously monitors memory pressure on the system and kills user-space victim process before the memory pressure reaches critical limits, thus providing system stability. The user-space OOM killer also allows for the deployment of custom victim selection policies as per the requirement. The evaluation of the proposed design shows that the user-space OOM killer kills an appropriate victim, preventing the kernel OOM invocation and maintaining the stability of the system.

8 Future Work

The current implementation only considers the parameters such as total system memory, available system memory, total swap memory, free swap memory and resident set size of a process. It doesn't consider the pressure stall information(PSI) that was discussed earlier for memory heuristics. The current implementation also lacks the ability to integrate a custom victim selection policy. The current implementation was also not tested on a system with no swap memory. Currently, the kernel level OOM heuristics perform in an unpredictable manner in some of the cases and select an unwanted process as a victim. There is a need for extensive research on making the kernel level OOM heuristics more deterministic. This will ensure that the kernel OOM selects a right victim in the scenario that user-space OOM fails to terminate a process. Following are the tasks planned as future work on the proposed design:

1. Integrate PSI monitor into the user-space OOM daemon.
2. Implement an interface to accept custom victim selection policies.
3. Test the implementation on a swapless system.
4. Research on making kernel level OOM heuristics more deterministic.

9 Goals of the Project

The goals of the project were as follows:

1. 75% : Create a mechanism in the kernel to send the notification of memory pressure to the user space.
2. 100% : Design a user-space OOM daemon that monitors the system memory and terminates appropriate user-space processes when required.
3. 125% : Make the user-space OOM daemon accept custom policies for the selection of the victim processes.

10 Overview of Project Development Status

The current implementation achieves the 75% goal. There is a notification mechanism designed to send the status of memory pressure through `/proc` file system.The current implementation also achieves the 100% goal. The implementation includes a user-space OOM killer that receives the notification, selects a victim process and sends the kill signal. Following are the tasks completed towards achieving 100% goal:

1. Found out the parameters required for the statistics calculation.
2. Created a kernel module to extract Total System Memory, Available System Memory, Total Swap Memory, Free Swap Memory.
3. Designed a notification system to send low memory signal(LMS) to user-space.
4. Designed a user-space OOM daemon that can select a victim process based on `oom_score` and `VmRSS` of each process.
5. Created a memory consumer user-space application to test the user-space OOM killer.

The current implementation does not achieve 125% goal since it does not accept custom victim selection policies and does not integrate PSI heuristics. That goal will be achieved as the part of future work.

References

[1] Jonathan Corbet. Lsfmm: Improving the out-of-memory killer, April 2013. https://lwn.net/Articles/548180/.

[2] Jonathan Corbet. Toward reliable user-space oom handling, June 2013. https://lwn.net/Articles/552789/.

[3] Jonathan Corbet. Toward more predictable and reliable out-of-memory handling, December 2015. https://lwn.net/Articles/668126/#reaper.

[4] Jonathan Corbet. Improving the oom killer, April 2016. https://lwn.net/Articles/684945/.

[5] Jonathan Levin. Handling low memory conditions in ios and mavericks, March 2013. http://newosxbook.com/articles/MemoryPressure.html.

[6] Johannes Weiner. Pressure stall information, April 2018. https://www.kernel.org/doc/html/latest/accounting/psi.html.