Memory Management Approach for Swapless Embedded Systems

Mauricio Lin

Ville Medeiros

Raoni Novellino

Ilias Biris

Edjard Mota

Abstract

This article presents a strategy for managing memory allocation in swapless, embedded systems to help you avoid system slowness and the dreaded Out-of-Memory killer exception.

The Linux kernel Out of Memory (OOM) killer is not usually invoked on desktop and server computers, because those environments contain sufficient resident memory and swap space, making the OOM condition a rare event. However, swapless embedded systems typically have little main memory and no swap space. In such systems, there is usually no need to allocate a big memory space; nevertheless, even relatively small allocations may eventually trigger the OOM killer.

Experiments with end-user desktop applications show that when a system has low memory—that is, it is about to reach the OOM condition—applications could become nonresponsive due to system slowness. System performance is affected when physical memory is about to reach the OOM condition or is fully occupied. System slowness should be prevented as such behaviour brings discomfort to end users.

Furthermore, the process selection algorithm used by the kernel-based OOM killer was designed for desktop and server computer needs. Thus, it may not work properly on swapless embedded systems, because at any moment it can kill applications that a user may be interacting with.

In this article, we present an approach that employs two memory management mechanisms for swapless embedded systems. The first is applied to prevent system slowness and OOM killer activation, by refusing memory allocations based on a predefined memory consumption threshold. Such a threshold should be determined and calibrated carefully in order to optimize memory usage while avoiding large memory consumption that may lead to system delay and invocation of the OOM killer. We call it the Memory Allocation Threshold (MAT).

The second mechanism employs an additional threshold value known as the Signal Threshold (ST). When this threshold is reached, the kernel sends a low memory signal (LMS), which should be caught by user space, triggering memory release before crossing the MAT. Both thresholds are implemented by a kernel module, the Low Memory Watermark (LMW) module. We offer some experimental results that point out situations when our approach can prove useful in optimizing memory consumption for a class of embedded systems.

Memory Management Approach

System performance is degraded when the memory requirements of active applications exceed the physical memory available on a system. Under such conditions, the perceived system response can be significantly slow. On swapless devices, application memory needs can drive the system to such conditions often, because system internal main memory is low and the chance of applications occupying the whole physical memory is high.

Memory resources should be managed differently on such devices to avoid slow system responsiveness. The memory allocation failure mechanism can be applied to prevent slowness. Preventing system slowness makes OOM killer invocation rare. Thus, such a mechanism also can reduce the chances of triggering the OOM killer, whose process selection algorithm may choose an unexpected application to be killed on devices with low memory and no swap space.

Memory allocation failure means refusing memory allocations requested by applications. It is carried out according to a MAT value that is set based on experimentation with various use cases of end-user applications. MAT should be set sufficiently high to allow applications to allocate necessary memory without affecting overall system performance, but its value should be well defined to guarantee memory allocation failure when necessary to prevent extreme memory consumption.

Before memory allocation failure occurs, however, process termination can be performed to release allocated memory. It can be triggered by transmitting the LMS from kernel space to user space to notify applications to free up memory. LMS is dispatched according to ST value. ST should be smaller than MAT, as shown in Figure 1, because the LMS should occur well before memory allocation failure.

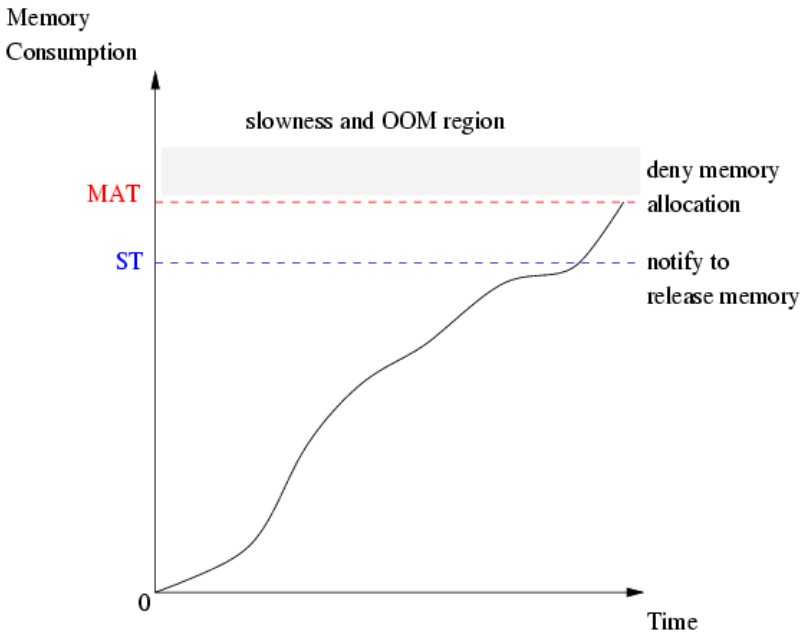

Figure 1. Signal Threshold is smaller than Memory Allocation Threshold.

If the LMS dispatch is successful and memory is released by receiving the signal, a possible memory allocation failure will be prevented. A useful scenario could involve running some window-based applications, A, B and C, consuming chunks of memory, while their window frames can superimpose one another (assuming the use of a simple window manager environment such as Matchbox). Assuming that application A is the one the user is interacting with at the moment MAT is reached, instead of denying memory allocation to A, it would be preferable to attempt to free up memory allocated by applications B and C, which are not visible to the user. Doing this would allow the user to continue working with application A.

However, memory allocation failure could be unavoidable for some application use cases. For instance, such a case could involve a single window-based application, consuming memory at a constant rate, that the user is interacting with. Releasing memory from other applications would not be as desirable in this situation, because there may be no other window-based applications from which memory could be released. Therefore, a more desirable solution would be to fail memory allocation requested by the guilty application, selecting it as a candidate for termination.

In our proposal, the kernel should provide two mechanisms to deal with management of memory in extreme cases of low memory levels:

- Failure of brk(), mmap() and fork() system calls: deny memory allocation requests to prevent system slowness and kernel OOM killer invocation according to a previously calibrated MAT level.
- Low memory signal: Kernel Event Layer signal sent by the kernel to a user-space process terminator, which should employ a process selection algorithm that works based on a specified ST.

Using these mechanisms, it would be possible to identify when memory can be released or when to deny further allocations. Denying memory allocations should happen only when memory release attempts cannot be successful.

Low Memory Watermark (LMW) Module

LMW is a kernel module based on the Linux Security Module (LSM) framework. It implements a heuristic to check the physical memory consumption threshold for denying memory allocation and notifying user space to free up memory. A user-space process terminator can be employed to free up memory. Formulas for low memory watermark thresholds are as follows:

- deny_threshold = physical_memory * deny_percentage
- notify_threshold = physical_memory * notify_percentage

physical_memory is the system's main memory and is represented by the kernel global variable totalram_pages. deny_percentage and notify_percentage are tunable kernel parameters, and the value of these can be altered through the sysctl interface. These parameters are bound to the /proc filesystem and can be written to and read from, using standard commands such as echo and cat. These parameters may be handled as follows:

```
$ echo 110 > /proc/sys/vm/lowmem_deny_watermark

$ echo 90 > /proc/sys/vm/lowmem_notify_watermark

$ cat /proc/sys/vm/lowmem_deny_watermark
110

$ cat /proc/sys/vm/lowmem_notify_watermark
90
```

The LWM architecture is illustrated in Figure 2. Basically, LWM overrides the kernel default overcommit behaviour by setting the vm_enough_memory function pointer field in the security_operations structure to point to the function low_vm_enough_memory(). low_vm_enough_memory() implements a

heuristic based on the formula described earlier. Binding vm_enough_memory to low_vm_enough_memory() permits interception of all requests for allocation of memory pages in order to verify whether the committed virtual memory has reached the MAT or ST watermarks. Listing 1 presents how the MAT and ST watermarks are implemented in the low_vm_enough_memory() function.

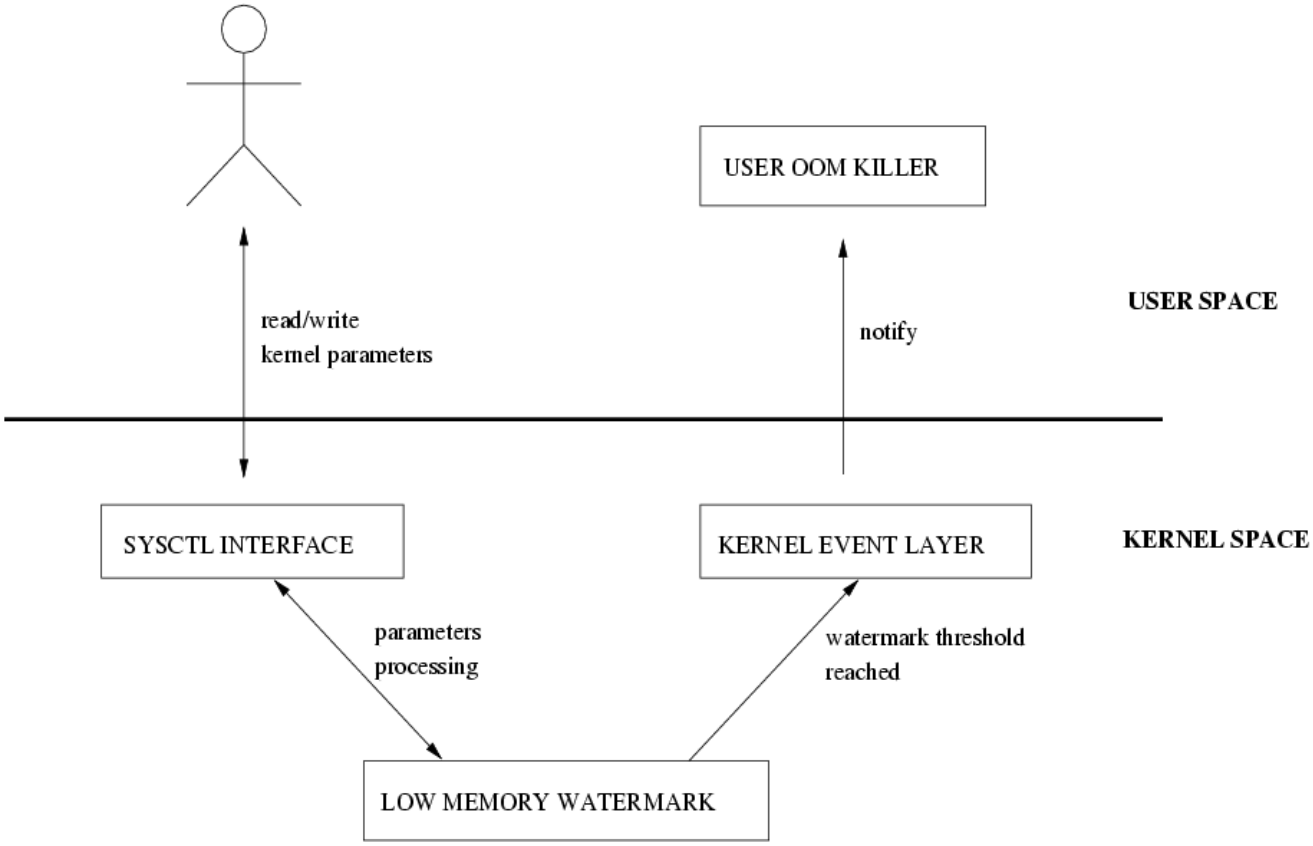

Figure 2. Low Memory Watermark Architecture

Listing 1. Algorithm of MAT and ST Watermarks Heuristic

```
1 static int low_vm_enough_memory(long pages)
2 {
3   unsigned long committed;
4   unsigned long deny_threshold, notify_threshold;
5   int cap_sys_admin = 0;
6
7   if (cap_capable(current, CAP_SYS_ADMIN) == 0)
8     cap_sys_admin = 1;
9
10  if (deny_percentage==0||notify_percentage==0)
11    return __vm_enough_memory(pages,cap_sys_admin);
12
13  deny_threshold=
14       totalram_pages*deny_percentage/100;
15  notify_threshold=
16       totalram_pages*notify_percentage/ 100;
17
18      vm_acct_memory(pages);
19      committed = atomic_read(&vm_committed_space);
20      if (committed >= deny_threshold) {
21              enter_watermark_state(1);
22              if (cap_sys_admin)
23                      return 0;
24              vm_unacct_memory(pages);
25              return -ENOMEM;
26      } else if (committed >= notify_threshold) {
27              enter_watermark_state(1);
28              return 0;
29      }
30      enter_watermark_state(0);
31      return 0;
32 }
```

The code in Listing 1 is explained below:

- Lines 7, 8: verify whether the current process has root privileges.

- Lines 10, 11: if MAT or ST watermarks are zero, perform the default overcommit behaviour.
- Lines 13–16: calculate the low memory watermark thresholds.
- Line 18: the pages are committed to update the amount of vm_committed_space.
- Line 19: the amount of committed memory is acquired.
- Line 20: verify whether committed memory has reached the MAT watermark.
- Line 21: set a flag state to 1 if MAT has been reached—state=1 means any (or both) of the two thresholds was reached.
- Lines 22, 23: do not deny memory allocation for root programs—allocation is successful for these.
- Line 24: uncommit the current committed pages since MAT was reached.
- Line 25: return no memory available message.
- Line 26: verify whether committed memory has reached the ST watermark.
- Lines 27, 28: set the state to 1, and allocation has succeeded.
- Line 30: set the state to 0 (if no threshold was reached).
- Line 31: memory allocation has succeeded.

The enter_watermark_state() function determines whether the low memory watermark condition has been reached and eventually sends the LMS to user space. A global boolean variable, lowmem_watermark_reached, marks the state of entering or exiting from low memory watermark conditions, being assigned to a value of 1 or 0, respectively. LMS is dispatched whenever a change in the value of this variable occurs.

Listing 2. Algorithm of Entering Watermark States

```
1 static void enter_watermark_state(int new_state)
2 {
3   int changed = 0, r;
4
5   spin_lock(&lowmem_lock);
6   if (lowmem_watermark_reached != new_state) {
7     lowmem_watermark_reached = new_state;
8     changed = 1;
9   }
10  spin_unlock(&lowmem_lock);
11  if (changed) {
12    printk(KERN_DEBUG MY_NAME ": changed to %d\n",
13           new_state);
14    r = kobject_uevent(&kernel_subsys.kset.kobj,
15                       KOBJ_CHANGE,
16                       &low_watermark_attr.attr);
17    if (r < 0)
18      printk(KERN_ERR MY_NAME
19             ": kobject_uevent failed: %d\n", r);
19    }
20  }
```

Listing 2 illustrates how the state is changed, and the LMS is sent to user space. Intuitively, the code works as follows:

- Line 5: lock to avoid a race condition.
- Line 6: verify whether the new state is different from the old one.
- Lines 7, 8: update the lowmem_watermark_reached and changed variable.
- Line 10: unlock to leave the critical region.
- Line 11: verify whether the state was changed.
- Lines 12–16: log that the state was modified and send the signal using the Kernel Event Layer mechanism.
- Lines 17–19: log a message if an error occurred.

Tuning Memory Consumption Parameters

Tuning MAT can be done empirically based on some use cases. Tuning of the ST watermark is not presented here, but it is usually done in the same manner as MAT. Applications used in the scenarios involved should succeed in filling the memory totally, thus overloading the system. Doing this can trigger system slowness and kernel OOM killing, thus ensuring a valid use case for tuning the MAT watermark.

As discussed previously, an optimal MAT value, the memory allocation refusal threshold, should be such so as to avoid system slowness and kernel

OOM killer execution. MAT value is given in terms of the percentage of memory that the kernel commits, possibly reaching values more than 100% due to Linux kernel's memory overcommit feature.

Basically, three behaviours need to be identified during experimentation: OOM killer execution, refusal of memory allocation and system slowness. The experiments were performed using a swapless device with 64MB of RAM memory and 128MB of Flash memory. The Flash memory is the secondary storage used as a block device to retain data.

The first use case involves reaching the MAT in a gradual manner, running the following applications (in the order they are listed): Web browser, e-mail client, control panel to configure the system and image viewer. First, the Web browser loads a Web page, followed by the e-mail client loading some 360 messages in the inbox, followed by the control panel, which is simply opened, and finally by the image viewer loading a number of image files, one after the other (only one image is loaded to memory at a time). Each image file is progressively larger than the previous one, all a few hundreds of KB, but one is about 2MB. Loading these files progressively can cause a different system behaviour according to different MAT values. Table 1 illustrates the results of this scenario when varying the MAT values.

Table 1. Table 1. MAT Value for Web Browser, E-mail Client, Control Panel and Image Viewer Use Case

MAT (%)	OOM Killer	Denied Memory	Slowness
120	2	0	3
119	r	0	1
115	5	0	0
112	2	7	1
111	0	5	0
110	0	5	0

A MAT threshold of 120% is not a good choice, because it allows OOM killing to occur twice while slowness occurs three times. The best MAT value, in this use case, is 111%, because at that level the system is able to deny all memory allocations preventing system slowness and kernel OOM killer execution.

In the use case described above, whenever the OOM killer occurs, it always kills the image viewer application. Slowness takes place when the image viewer tries to load the heavy image file of 2MB. During the experiment, it was perceived that the OOM killer is always started during the system slowness, and usually system slowness is so severe that waiting for OOM killing is not viable.

A second use case could try to reach the MAT threshold in a more direct manner. The following applications are started: Web browser, PDF viewer, image viewer and control panel. The Web browser loads a Web page, then the PDF viewer attempts to load a file of 8MB, followed by the image viewer loading an image file of 3MB and finally invoking the control panel.

In this use case, whenever the image viewer loads the image file, the PDF file of 8MB loaded previously is unloaded, because of the ST threshold being reached, causing a signal dispatch to user space in order to free up memory. The observed behaviour also involved the termination of the control panel application, which can be attributed to memory allocation denial due to having reached MAT. Table 2 presents the experimental results for this use case for different MAT values.

Table 2. Table 2. MAT Value for Web Browser, PDF Viewer, Image Viewer and Control Panel Use Case

MAT (%)	OOM Killer	Denied Memory	Slowness
120	0	0	5
113	0	0	5
112	0	1	4
111	0	2	3
110	0	5	0

This use-case scenario indicates a reliable MAT value of 110%. Slowness occurs for values above 110% when the control panel is started. Figure 3 illustrates how the MAT and ST behave in this use case. The memory consumption curve shown is assumed, but it does not in any way alter the aforementioned results.

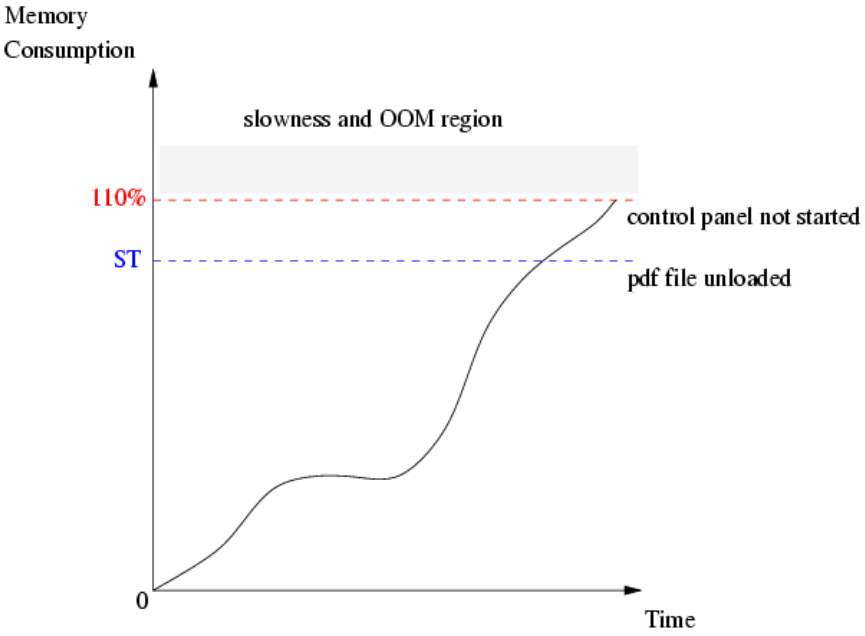

Figure 3. Low memory watermark graphic, based on Web browser, PDF viewer, image viewer and control panel use case.

During experimentation, it is important to verify whether the planned use cases are satisfactory for calibrating the MAT value, because there could be use cases that do not overload memory allocations. An example of such a scenario could be invoking the Web browser to download a file of 36MB in the background while playing a game at the same time. Our experiments indicated that this use case was not as useful in determining a realistic MAT value, because it worked successfully even with a MAT value of 120% or higher.

Some Additional Remarks

A useful approach in assisting the fast selection of processes to be killed, in order to release memory, could involve registering applications as killable or not. Applications considered killable could be registered on a list known as the Red List. Additionally, other applications, crucial for the correct functionality of the system, such as the X Window System, should not be killed under any circumstances and could be registered on a list known as the White List.

End users could be allowed to choose which applications should be registered on the Red or White Lists. However, this would require a security mechanism in place to ensure that applications on the Red List or White List do not cause any unexpected conditions or instabilities. If application A is the culprit by consuming tons of memory continuously, it cannot be on the White List. Likewise, if killing application B can break down overall system functionality, then it cannot be on the Red List. A heuristic could be employed for selecting in advance which applications can be registered on the Red List or White List. Preselected applications could then be presented to the user to be opted for registration on the respective list, thus improving user-friendliness while avoiding potential problems from choosing erratically.

The Red List and White List could be implemented in kernel space, with each list also reflected in the /proc filesystem. ST can be used to notify user space the moment when the Red and White Lists should be updated. Afterward, the kernel can start terminating applications registered on the Red List in order to release memory. Perhaps a ranking heuristic can be employed in kernel space to prioritise entries on the Red List. Figure 4 illustrates a possible architecture of OOM killer, based on Red List and White List approach. If it is not enough simply to kill processes on the Red List, other processes, not appearing on the White List, could be killed as well, as a last measure to ensure system stability.

It is interesting to maintain a mechanism based on having one heuristic for selection and termination of processes in user space and another one in kernel space, because each space can offer different pieces of information that may prove useful to the ranking criteria. For instance, in user space it is possible at any moment to know which window-based applications are active, that is, visible and used by the end user, but in kernel space such information is not as easily attainable. Hence, if there is a heuristic that needs to verify whether any window-based application is active or not, it should be implemented in the user space.

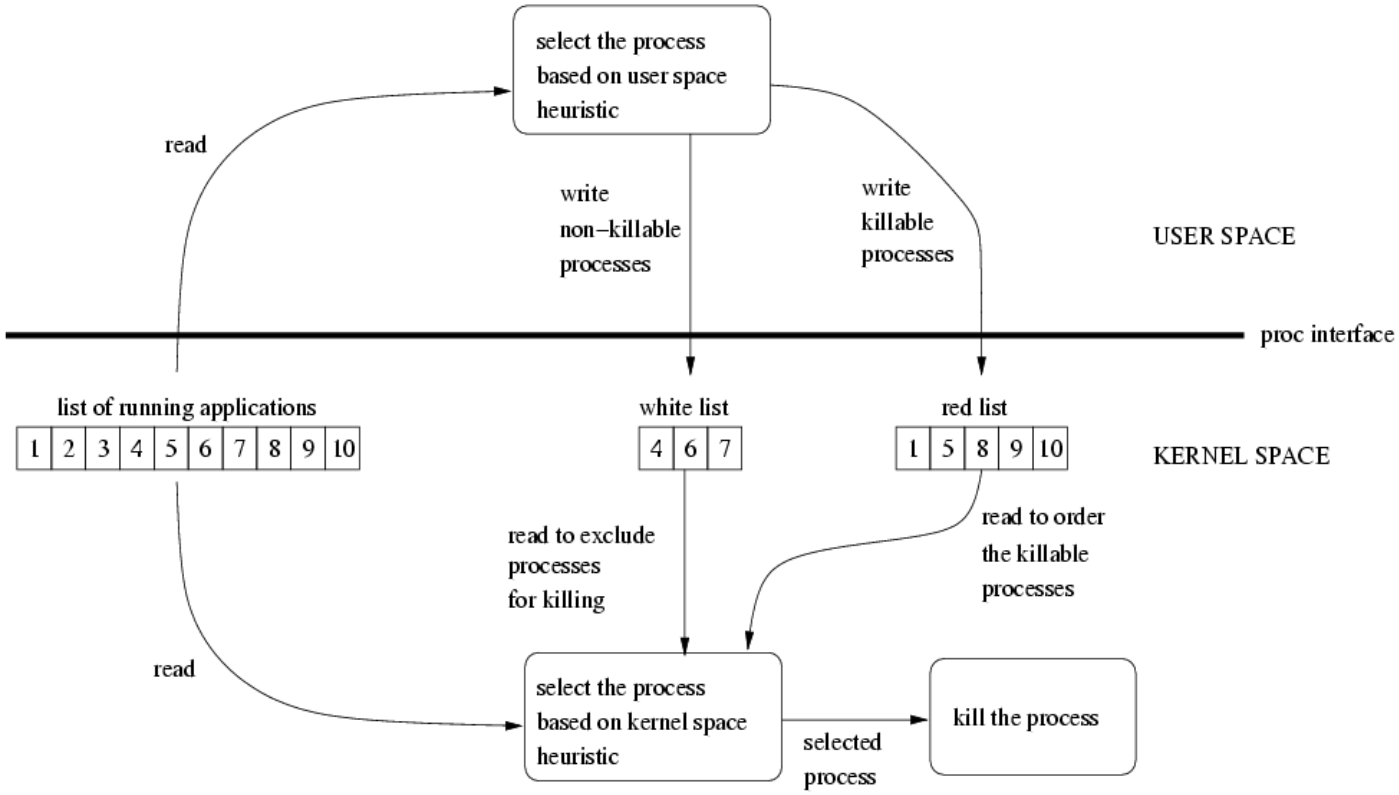

Figure 4. Architecture of OOM Killer Based on the Red List and White List Approach

Conclusion

Dealing with swapless embedded systems requires establishing an alternative memory management approach, in order to prevent slowness and to control OOM killer invocation and execution. The idea based on MAT and ST is simple yet practical and tunable on different swapless embedded devices, because the LMW kernel module provides the /proc and sysctl interfaces to change the MAT and ST values from user space as necessary.

Additional mechanisms can be implemented, such as the Red and White registration Lists. It is also interesting to design different selection criteria that take into account features related to swapless embedded devices.

Acknowledgements

We wish to offer our thanks to Kimmo Hämäläinen of Nokia Research Center (NRC) and Monica Nogueira for help on the organization and contents of this document, Juha Yrjölä of NRC for help with Low Memory Watermark coding and Fabritius Sampsa of NRC for providing us with the opportunity to develop this work.