In Search of an Efficient OOM Killer

A Kernel + User Hybrid Approach for Imposing Per-User Memory Limit

ZHANG Chi (aka. Alex Chi) 518021910395
iskyzh@sjtu.edu.cn

Date: June 18, 2020

Contents

1 Introduction

In this new OOM killer, we achieved:

- **A Kernel + User Hybrid Approach** By combining a reliable but slow-response kernel timer trigger, and an unreliable but fast-response user-space trigger, we made an OOM killer which impacts little on kernel performance.
- **Leverage Kernel Event Tracing Facilities** By exposing RSS change events through trace pipe, we reduced possible slowdown in kernel, and the user-space notifier listen for events passively. This enables the notifier to efficiently decide when to fire the OOM killer.
- **Modify the Last Defense of Linux Kernel** The original OOM killer will still be fired when the system is out of memory, but it will kill process with per-user limit at first.
- **Robust OOM Trigger** By offloading some of the works from kernel to user space, we made kernel modification very concise and easy to verify. The three-trigger scheme can handle the case of partial trigger failure.
- **Focus on Kernel Performance** We used data structures such as hash table in kernel, as well as unordered map in user space. We preferred lock-free structures to mutex. In this way, we successfully reduced lock contention and achieved higher performance.
- **Soft Limit Technique** By early bootstrapping to kill a process that may exceed limit, we can reduce the case of over-committing.
- **Transactional Memory Allocation** User-space programs may use a newly-introduced `tmalloc` to coordinate with other processes of same user. In this way, they will collaborate to not exceed the memory limit.
- **Intensive Benchmark** We run benchmark in two aspects: memory allocation speed and memory over-committing. We observed comparable performance to the original Linux kernel. In this way, new OOM killer impacts very little on performance.

2 Motivation

2.1 Current Linux Out-Of-Memory Killer Implementation

The OOM killer is called on insufficient memory. The call path of current OOM killer is as follows.

```
__alloc_pages
-> __alloc_pages_nodemask
  -> __alloc_pages_slowpath
    -> __alloc_pages_may_oom
      -> out_of_memory
```

When a kernel thread requires new pages, it will call `__alloc_pages`. Then, Linux kernel will first try allocating a page from per-CPU free page list. If this cannot be done, it will go into slow path and allocate a page from global buddy allocator.

If it turns out that we are out of memory, the kernel will first try to reclaim some pages. Failed reclaiming attempts will finally call into `__alloc_pages_may_oom`.

Inside that, we will finally have some checks, and call the OOM killer. In this way, we can reclaim some memory and allocate new pages.

In summary, the original OOM killer will be triggered when we failed to allocate new pages. The trigger of the original OOM killer is memory (or page) allocation.

2.2 Other Trivial Approaches

One common approach of implementing an OOM killer is to hook the `alloc_pages` code path. In this way, every time a page is allocated, the OOM killer will be called. As OOM killer locks the task struct, no new process can be created in this period. Furthermore, one call of OOM killer is very time-consuming. This will cause significant performance regression.

Another approach is to use a kernel timer or a user-space daemon, which calls OOM killer periodically. This implementation suffers greatly from over-commiting. An evil process may eat up the memory even before the OOM killer is triggered. If the timer is scheduled of small interval, performance may suffer. If, on the other hand, the OOM killer is triggered on large interval, evil process may over-commit and eats up all resource of the operating system.

Our new OOM killer offloads page allocation trigger into user space by exposing RSS change via kernel event tracing facility, and runs a kernel timer to call OOM killer periodically. When the system really goes out of memory (instead of exceeding per-user limit), the original Linux OOM killer will kill those with limit first. This combination leads to a high-performance and safe OOM killer.

3 Design

3.1 Architecture Overview

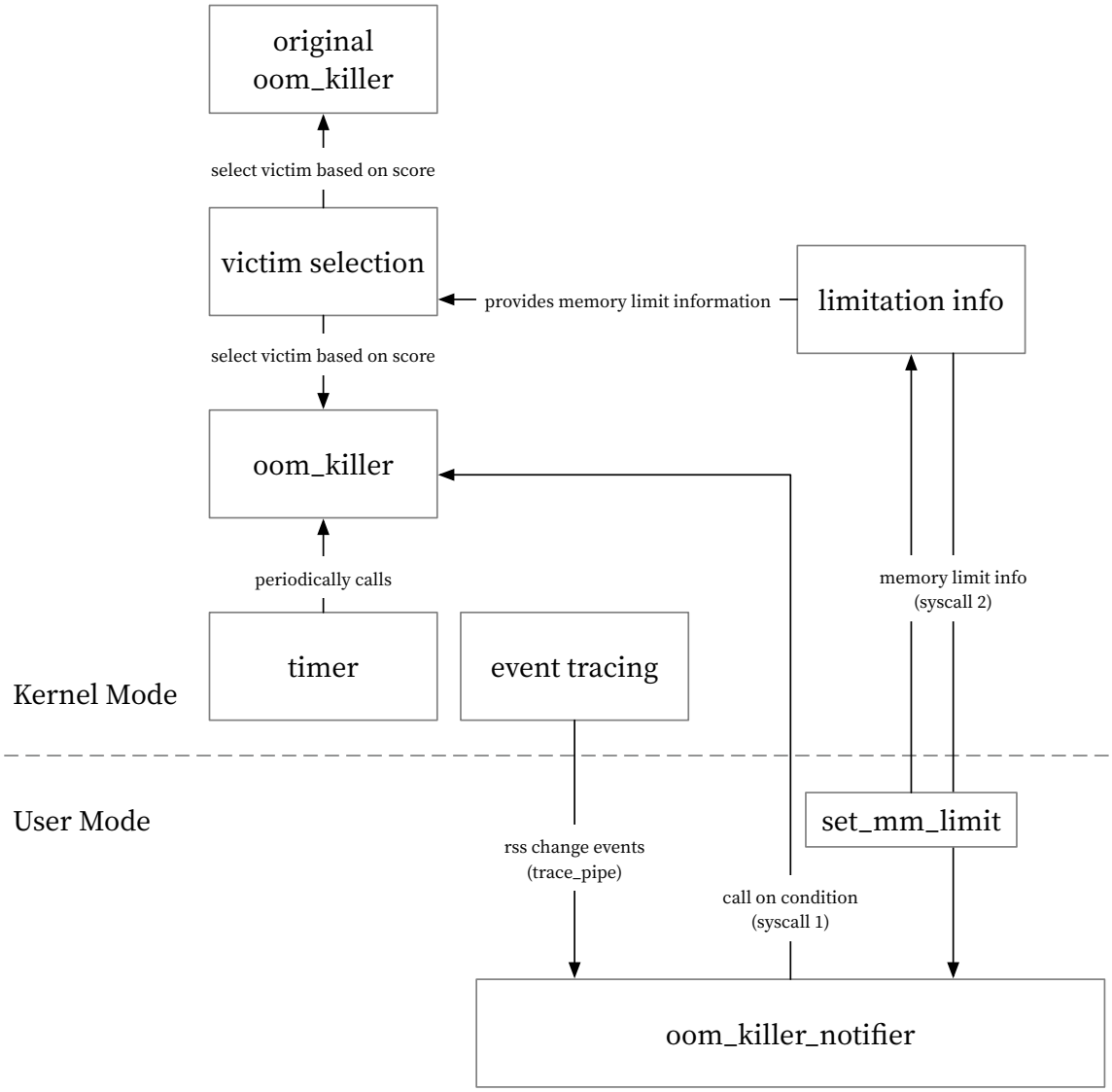

Figure 1: Architecture Overview

As seen in Figure 1, the new out-of-memory killer is composed of several components.

The OOM killer is triggered by 3 sources.

- **Timer in Kernel Space** In kernel, we placed a timer that triggers OOM kill in given interval. This interval was set to be 200ms. This trigger is reliable, but responds to memory events slowly.
- **Notifier in User Space** The *OOM Killer Notifier* runs in user-space. It reads RSS change events from kernel trace pipe, processes all events, and decides whether to trigger OOM killer or not. A user-space program may crash and act wrongly, but it won't affect the kernel. A user-space notifier is unreliable, but responds

to memory allocation very quickly. And we can use complex structures in user-space programs.

- **Original Linux OOM Killer** This is the last defense of the OOM killer. If our user-space notifier crashes, and the evil process eats up memory too quickly, original Linux OOM killer will be triggered. We have changed the victim selection process, so process with memory limit will be killed first.

By combining a **reliable but slow-response** kernel timer trigger, and an **unreliable but fast-response** user-space trigger, plus modifying last defense of original Linux OOM killer, we can kill evil process quickly, as well as boost efficiency.

We designed 3 syscalls for the OOM killer.

- **set_mm_limit** Users may set per-user memory limit using syscall.
- **get_mm_limit** *OOM Killer Notifier* may get current limits in system using syscall.
- **run_oom_killer** The notifier can trigger OOM killer using this syscall.

To expose RSS change events to *OOM Killer Notifier*, we added event tracing facilities in kernel RSS-related functions.

Victim selection component scans all processes, and selects the process to kill.

3.2 User-Space Trigger: RSS Event Tracing

Tracepoints can be used without creating custom kernel modules to register probe functions using the event tracing infrastructure. [6] By leveraging tracepoints in kernel, we can obtain kernel information efficiently from trace pipe without blocking the kernel, and process them in user space daemon.

Tracepoints can be created very easily. They can be created in kernel with `TRACE_EVENT` macro. After that, we can use trace functions to place tracepoints anywhere in code. For example, kernel memory subsystem (kmem) provides many tracepoints such as `mm_page_alloc` and `mm_page_free`. These two tracepoints are called respectively on page allocation and page free.

Tracepoints impact little on performance. The internal structure of storing logged events is a per-CPU lockless ring-buffer. [2] This makes tracepoints a very high-performance facility to log kernel information.

Linux event tracing provides a simple plain-text format for parsing. Here is an example.

```
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
    silly-thread-5377  [000]  1802.845581: me_silly: time=4304842209 count=10
    silly-thread-5377  [001]  1802.845581: me_silly: time=4304842209 count=10
    silly-thread-5377  [002]  1802.845581: me_silly: time=4304842209 count=10
```

In this way, the new OOM killer place tracepoints on RSS change from kernel space.

User-space notifier then listens for these events, and decides when to fire the kernel-space OOM killer.

By leveraging the event-tracing facility in Linux, the *OOM Killer Notifier* reacts very quickly to memory allocation. On one side, it won't block kernel operations, as putting log in per-CPU ring buffer is very fast. On the other side, every RSS change event can be observed almost immediately in *OOM Killer Notifier*. Thus, if any process exceeds the memory limit, the notifier will fire OOM killer in a flash.

Writing user-space program is much more trivial than writing kernel module. User-space program provides higher-level APIs such as C++ standard template library. We use `unordered_map` or hash map to store all necessary information in the notifier. Meanwhile, we do not need to consider memory allocation, interrupts, locking, concurrency and process scheduling. This makes programming easier, and prevent possible kernel panic if we wrote a kernel module. Another benefit of a user-space notifier is portability. We can easily target the notifer at another platform, such as x86, without massive refactor in kernel.

However, user-space daemon is unreliable. User program may crash, may be killed by kernel, and may act maliciously. Thus, we can not trust all information from user-space program. The design of new OOM killer only allows the user-space daemon to fire the OOM killer. The victim process selection process and the killing procedure all occur in kernel.

In conclusion, the user-space *OOM Killer Notifier* reacts quickly to memory allocation, but it is unreliable. By offloading the trigger into user-space, we can eliminate kernel panic and achieve high performance.

3.3 Interval Trigger: Kernel Timer

Timers are used to schedule execution of a function (a timer handler) at a particular time in the future. [5] As user-space notifier is not reliable enough, we must run OOM killer periodically to check if there is any process exceeding memory limit, in case of notifier failure.

Linux provides `mod_timer` function, which allows us to schedule a subroutine at a specified time. That is to say, if we schedule the timer once, the subroutine corresponding to the timer will only run once. To run the OOM killer periodcally, we have to schedule the timer continously.

Interval of OOM killer timer trigger is carefully selected to accompany user-space notifier and backup when notifier fails. If we run the timer too quickly, performance of Linux kernel will suffer great penalty. If the timer is triggered in long period, it may not kill out-of-memory process, and may cause resource drain. In this way, we select the interval to be 200ms.

Another point worth mentioning is that, only one thread will enter OOM killer

triggered by timer. To ensure this property, timer was scheduled at the end of each OOM epoch.

3.4 Last Defense: Linux Out-Of-Memory Killer

It is still possible that a process eats up memory so quickly, that none of above trigger is fast enough to response. In this way, the original OOM killer will be triggered.

The original OOM trigger is modified to kill those with per-user limit first.

3.5 Victim Selection Policy

First of all, we find the first user who exceed memory limit. And then, we scan all processes of this user, to find the process with largest RSS.

After that, we mark this process as "to be killed", and try to kill this process. As there is latency between sending kill signal and actually being killed, we will check if that process is "to be killed" before killing it.

4 Implementation

4.1 Syscall Design

In this new OOM killer, we have in total 3 syscalls.

- `set_mm_limit(uid_t uid, unsigned long mm_max)` Set memory limit for a user.
- `get_mm_limit(uid_t *uid, unsigned long *mm _max, unsigned int max_element)` Get all limits in system. Save them into `uid` array and `mm_max` array. This syscall will return `-EFAULT` if current limitation imposed is larger than `max_element`
- `run_oom_killer` Trigger OOM killer from user space.

4.2 User-space Notifier

The user space *OOM Killer Notifier* mainly does two things. It parses RSS change events from trace pipe, and decides when to fire the OOM killer.

As described in previous section, we can parse the trace event by splitting the string. This can be easily done with C standard library.

```
sscanf(&buffer[17], "%d", &pid);
sscanf(&buffer[48], "%[a-zA-Z_]: member=%d size=%ld", event_name, &member, &size);
```

In this way, we can obtain which process has RSS changed, and the RSS of that process after change. The internal of OOM killer notifier can be seen in Figure 2.

To count memory usage of every user, we must obtain corrsponding user ID of the process. The `proc` peusdo file system provides a "file" to read per-process information. in `/proc/pid/status`, we can find a line begins with `Uid`. In this way, we can find the relationship between pid and uid.

The pid-to-uid information is then stored in a hashmap cache, in order to boost performance. As Linux allocates process ID incrementally, and re-use previous PID only when we are running out of that (e.g. used up 32767 PIDs), this cache can be valid for a long time. The cache is flushed every 10000 RSS events.

Then we can update current memory usage of every user. This is done incrementally. That is to say, we only update memory usage of related user, instead of summing up all usage information. We compare previous RSS and current RSS to get a delta, and add that delta up to user usage. As user usage is stored in hash table, we can update it in O(1) time. In this way, one event only modifies one user's usage. This makes the notifier very efficient.

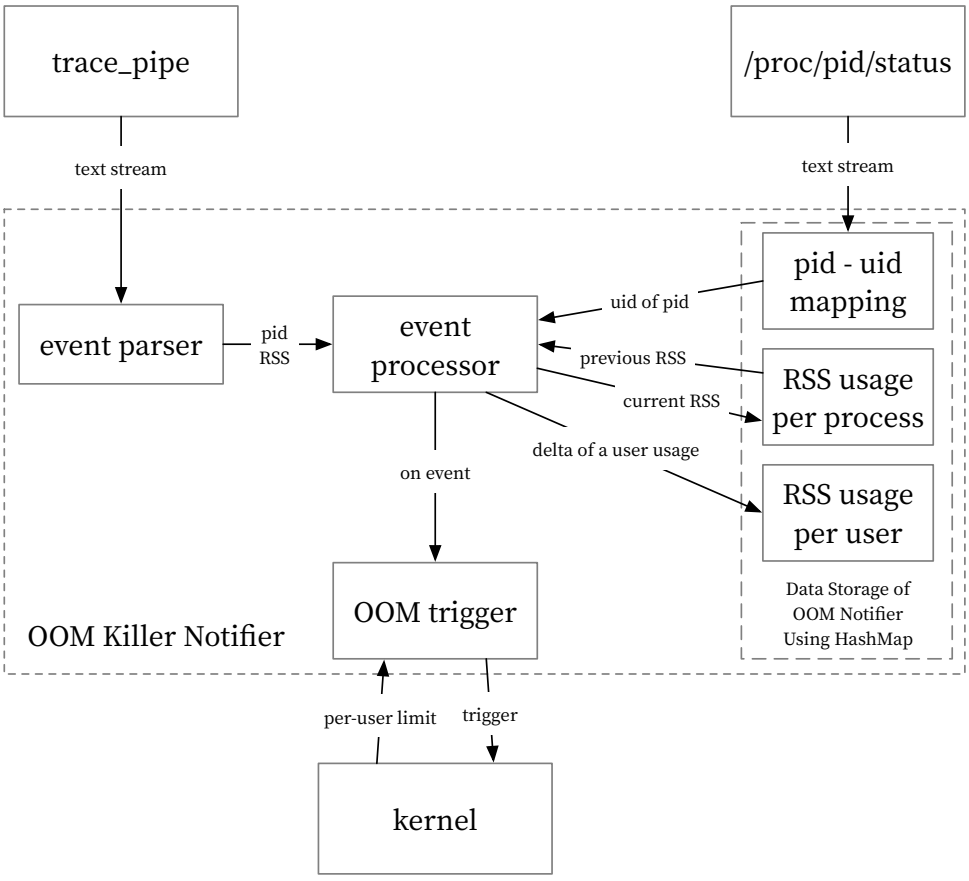

Figure 2: OOM Notifier Internal Structure

The per-user limit in kernel is updated every 10000 RSS events. As this information doesn't change frequently, and is only useful when there is actually memory allocation, this update frequency is acceptable in experiments.

4.3 Prevent Over-Committing with Early Bootstrapping

As every event changes memory usage of only one user, the *OOM Killer Notifier* only have to check one user on every event. If this user exceeds current memory limitation, we must notify the kernel to run OOM killer.

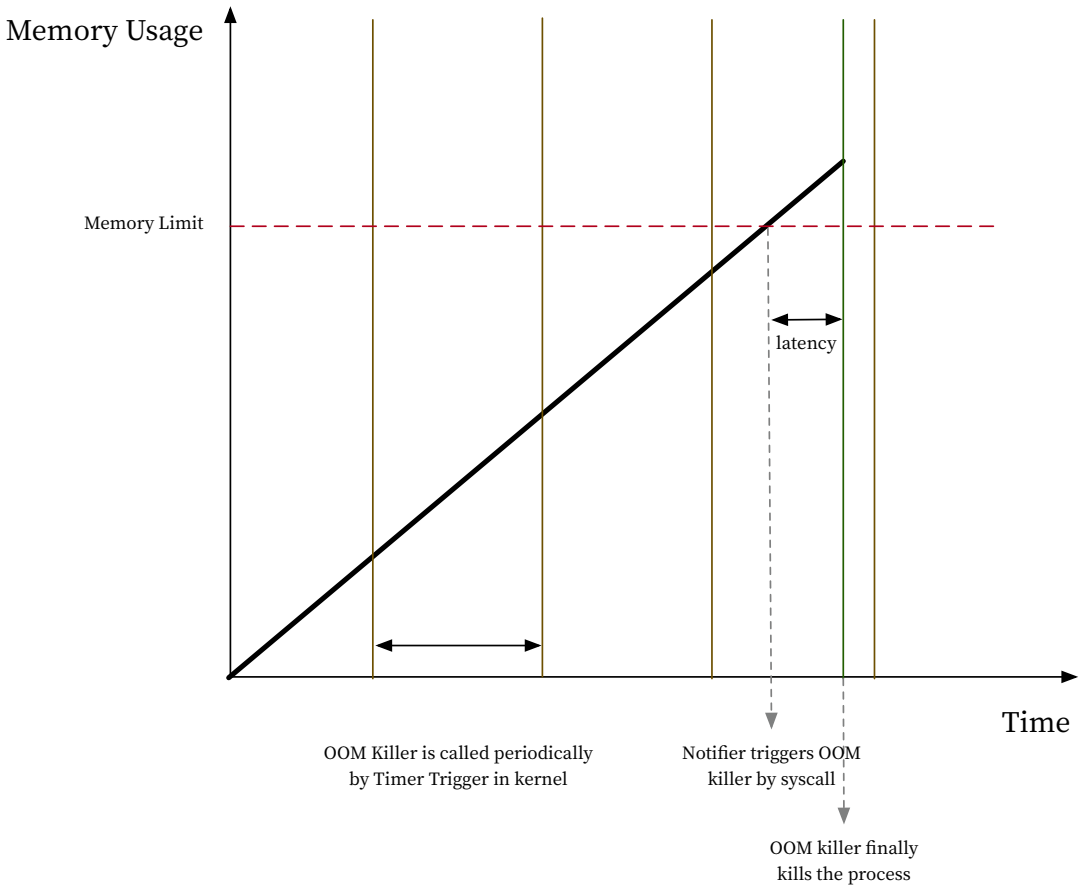

Figure 3: Evil Process Exceeding Memory Limit on Killing

This scheme sounds good at first, but it suffers from over-committing. As seen in Figure 3, at the time we notify the kernel to run OOM killer, the user may already have used up its limit, and is allocating more memory. When the kernel actually kills that evil process, it may have already eaten up too much memory above limitation.

There are many reasons behind this latency.

1. **Trace Pipe Latency** It may take some moments before the trace events get flushed into the pipe, and read by *OOM Killer Notifier*.
2. **Process Scheduling** *OOM Killer Notifier* may be put into sleep as it is a user-space process.
3. **Syscall Latency** Inside syscall, there is possibility that page allocation causes the process to sleep.

To solve this problem, we introduce a **soft limit** in *OOM Killer Notifier*.

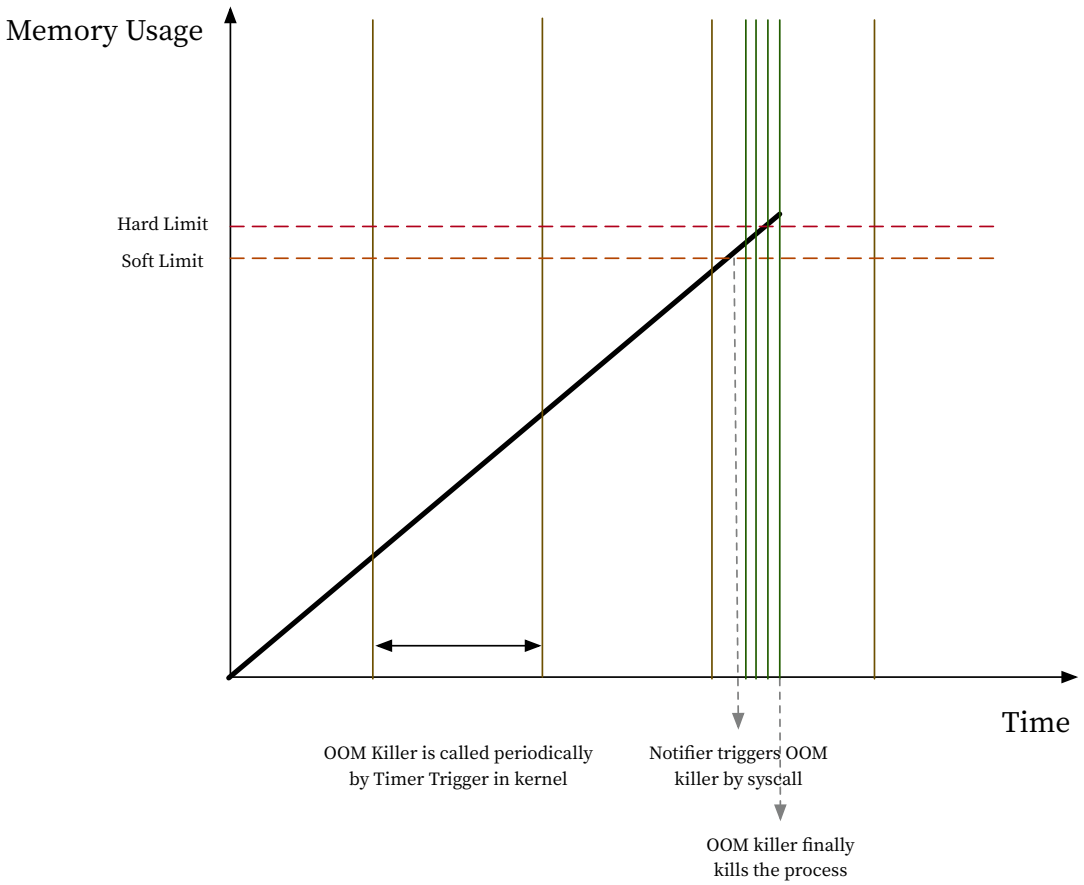

Figure 4: Use Soft Limit to Prevent Over-Committing

As seen in Figure 4, the notifier internally use soft limit to determine whether to trigger the OOM killer. If any user exceeds its soft limit, the notifier will trigger OOM killer in short interval.

Currently, soft limit is set to be 90% of hard limit. Hard limit (or memory limit) is directly set by user using "set_mm_limit" command or syscall. Soft limit is automatically calculated, and cannot be set by user.

If one user exceeds its soft limit, OOM killer is triggered every 10 RSS change events. By using soft-limit technique, the system only suffers performance regression when one process is to be killed. In this way, we can reduce the case of over-committing.

4.4 Kernel Hacking

New syscalls are added into kernel, and most of the OOM killer functionality is implemented in `sys_arm.c`.

4.5 Reduce Memory Footprint

When summing up memory usage of a user, we have to allocate an array or map to save current sum. However, this can be smartly eliminated by reusing memory.

```
struct mm_limit_struct {
  int alloc_pages;
  int alloc_pages_ticket;
  atomic_t alloc_pages_limit;
  uid_t uid;
  struct hlist_node hash;
};
```

Inside `mm_limit_struct`, we use a *ticket* to indicate whether the summed pages is from current OOM call. If not, we set it to zero, and update the ticket. Otherwise, we can add current process RSS to alloc pages.

4.6 Reduce Lock Contention

We use hash table, fine-grained lock and atomic variable in kernel space, to reduce possible lock contention. In this way, we can reduce overhead for calling the OOM killer.

One example is how OOM killer in kernel saves per-user memory limit. This is done by using a hash table.

```
struct mm_limit_struct {
  int alloc_pages;
  int alloc_pages_ticket;
  atomic_t alloc_pages_limit;
  uid_t uid;
  struct hlist_node hash;
};

DEFINE_HASHTABLE(mm_limit_struct_hash, 8);
```

Also, by setting `alloc_pages_limit` as an atomic variable, we can allow fearless concurrent read and write to this value.

5 Transactional Memory Allocation

5.1 Design Overview

The introduction of new OOM killer brings a new issue: users are unaware of their memory usage.

The original Linux kernel will return NULL to malloc function, if a user used up all memory (when over-commit option off). To make the user aware that they used up memory, we can also implement something like malloc, that will return NULL if the user exceeds new OOM killer limit.

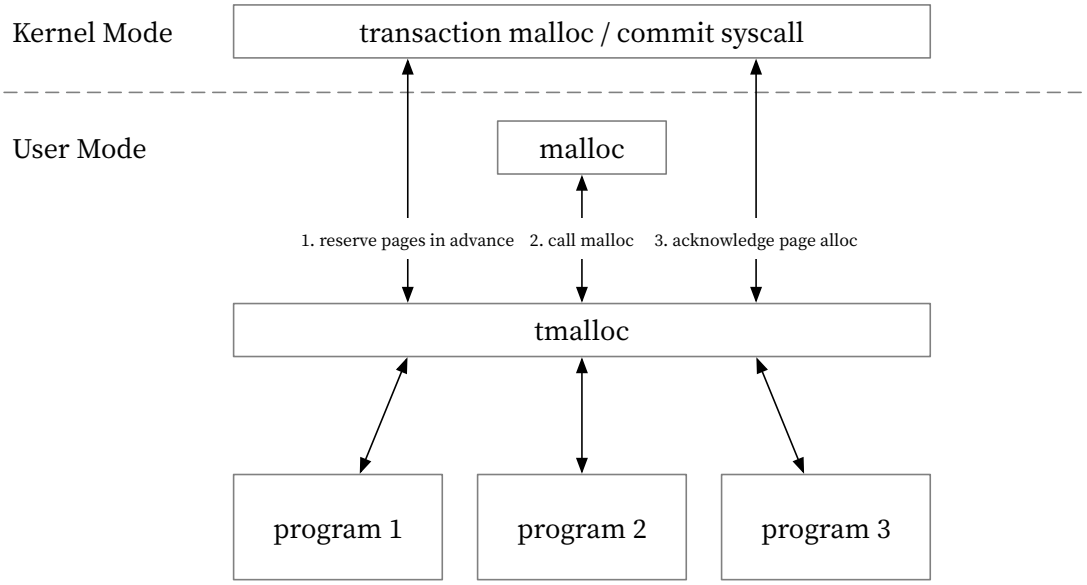

Figure 5: Transactional Memory Allocation Architecture

As seen in Figure 5, the implementation involves changes in both user space and kernel space.

In user space, all programs that intend to collaborate without exceeding limit uses `tmalloc` as their default memory allocator. In kernel space, a set of syscall `transaction_reserve` and `transaction_commit` is added, so as to support this kind of transactional memory allocation.

Another way to prevent process from allocating memory exceeding new user limit is to modify the `sbrk` syscall. As we don't have enough time in building this project, we choose to add this transactional syscall to achieve this goal.

5.2 Memory Allocation Transaction

When a user program requires new memory space (using heap), they will now call `tmalloc`. The new `tmalloc` does the following things.

1. Query kernel if it is possible to allocate N pages. If possible, the `transaction_reserve` syscall will return 0, and that page is reserved for this process. This is an atomic operation.
2. Call `malloc`.
3. Notify kernel the new page has been allocated with `malloc`, and now other process can begin to allocate new pages.

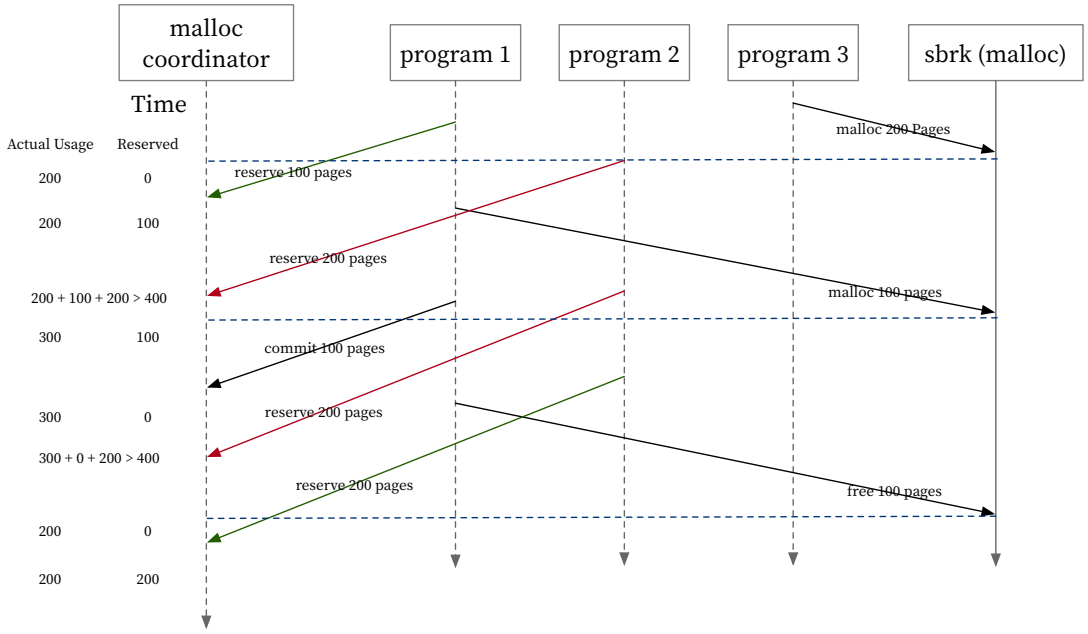

Figure 6: Transactional Memory Allocation Example

Figure 6 shows an example of new transactional memory allocation procedure. In the kernel is the *memory allocation coordinator*. It interacts with user process with the two new syscalls. All 3 programs belong to same user.

Program 1, program 2 collaborates not to exceed memory limit by using `tmalloc`. Program 3 doesn't support `tmalloc`, and uses the bare `malloc` to allocate memory.

When *memory allocation coordinator* receives a reserve request, it firstly sums up current memory usage of this user. Then, it sums up all reserved space of this user. If the sum of these two value exceeds per-user memory limit, this request won't be granted. Otherwise, the memory will be reserved.

For example, the first request from program 2 is rejected. That is because, all processes of this user already owns 200 pages (malloc by program 3), and program 1 has reserved 100 pages (but we don't know if it has actually been allocated 100 pages). If we reserve 200 pages for program 2, it is highly possible that this user will exceed memory limit, and a process will be killed.

This process is very similar to deadlock detection algorithm. And this implementation has nearly no overhead on normal memory allocation.

6 Evaluation

6.1 Environment Set-up

The evaluation is done on Ubuntu 18.04 virtual machine running on a macOS host. Inside that, we run an Android emulator with custom kernel. We select the following 4 cases for comparison.

- **Unmodified Kernel** We compiled original goldfish 3.4 from AOSP source, with the configuration provided in lab handout.
- **OOM Killer Disabled** In this case, the kernel is a modified version. We added all elements described in previous sections such as tracing facilities and OOM killer syscalls. However, the *OOM Killer Notifier* is not enabled.
- **OOM Killer Enabled** In this case, we started up *OOM Killer Notifier*, and set memory limit for several users.
- **Trivial OOM Killer** The kernel is modified to call OOM killer every time a new page is allocated.

We didn't do evaluation on the kernel shipped with Android emulator, because we suspected that Google has fine-tuned the kernel building parameters, so that the performance is blazing fast. In this way, we compiled the original goldfish 3.4 by ourselves.

When Android boots, it spawns several processes to initialize the Java virtual machine. Therefore, we must wait until this process ends, and evaluation is done 5 minutes after booting the emulator.

6.2 Speed of Memory Allocation

We benchmark speed of memory allocation. The evaluation program just allocates some size of memory, and fills it with zero. We used UNIX time utility to get real, user and system time of the benchmark program. We run several test cases as described below.

- 1 process, 4MB per process, 100 iterations
- 1 process, 32MB per process, 50 iterations
- 1 process, 64MB per process, 20 iterations
- 1 process, 128MB per process, 10 iterations
- 4 processes, 1MB per process, 100 iterations
- 4 processes, 4MB per process, 50 iterations
- 4 processes, 16MB per process, 20 iterations
- 4 processes, 64MB per process, 10 iterations
- 8 processes, 1MB per process, 50 iterations
- 8 processes, 4MB per process, 20 iterations
- 8 processes, 16MB per process, 10 iterations

We summed up time of all iterations, get the average of one iteration, and use system time plus user time as benchmark result.

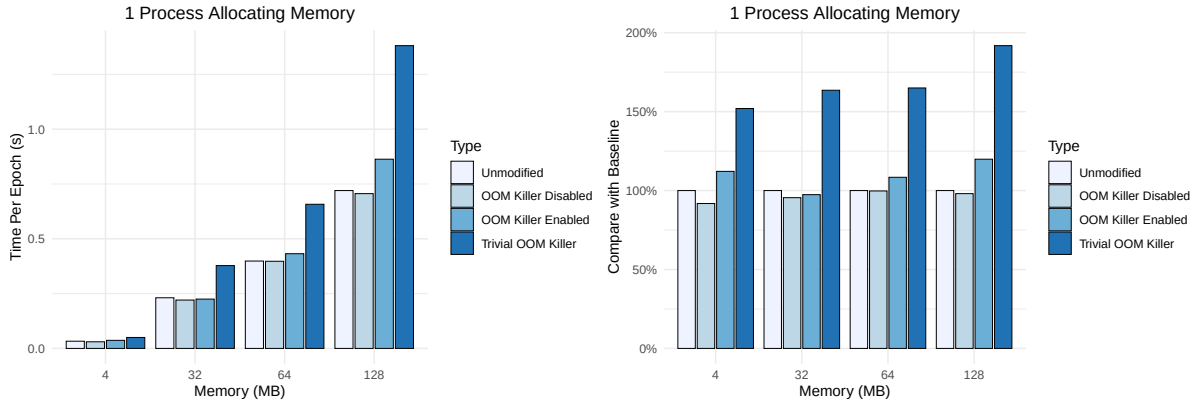

Figure 7: Evaluation of 1 Process Allocating Memory

As seen in Figure 7, our new OOM killer has comparable performance with the original kernel. That is to say, when there is only one memory-intensive process on system, there is no significant performance regression from the original kernel. And our OOM killer outperforms the trivial implementation by 50%.

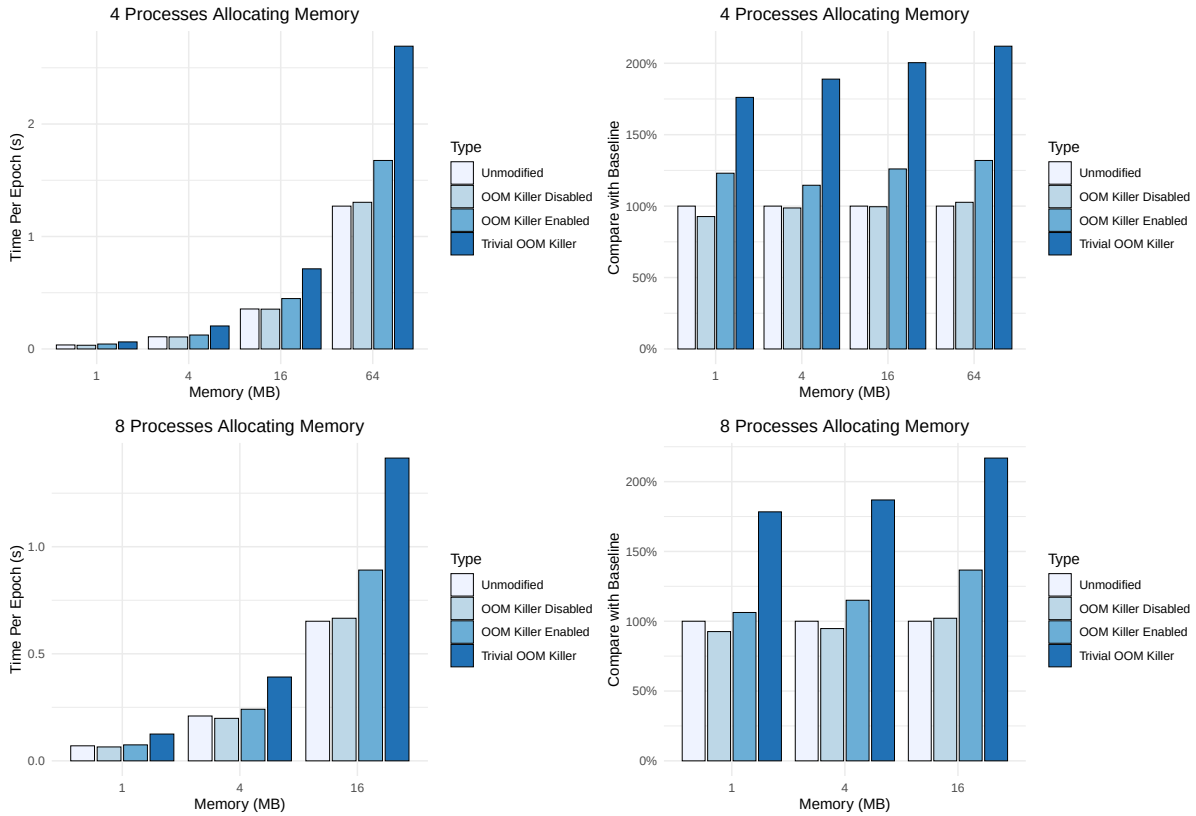

Figure 8: Evaluation of Multiple Processes Allocating Memory

When there are multiple memory-bound processes, as seen in Figure 8, We observe only 20% performance regression from the original kernel. Meanwhile, the new OOM killer has 2x speed up than the trivial implementation.

6.3 Memory Over-Committing

Using a user-space notifier or timer may cause latency. The cause under this latency includes several aspects, as analyzed before. With the introduction of a user-space *OOM Killer Notifier*, we can reduce this latency significantly.

The benchmark contains 4 cases: 1 or 4 processes, and 64MB or 128MB limits. When the OOM killer kills a process, we get the "over-commit" value by *Current User RSS – User Limit*.

Each case runs 10 iterations on each OOM killer implementation. One iteration may include multiple OOM kills, as there are more than 1 process. We benchmarked 3 implementations: "Trivial", which hooks the `alloc_pages` code path. "Timer", which uses a 200ms interval for OOM killing. And our new OOM killer.

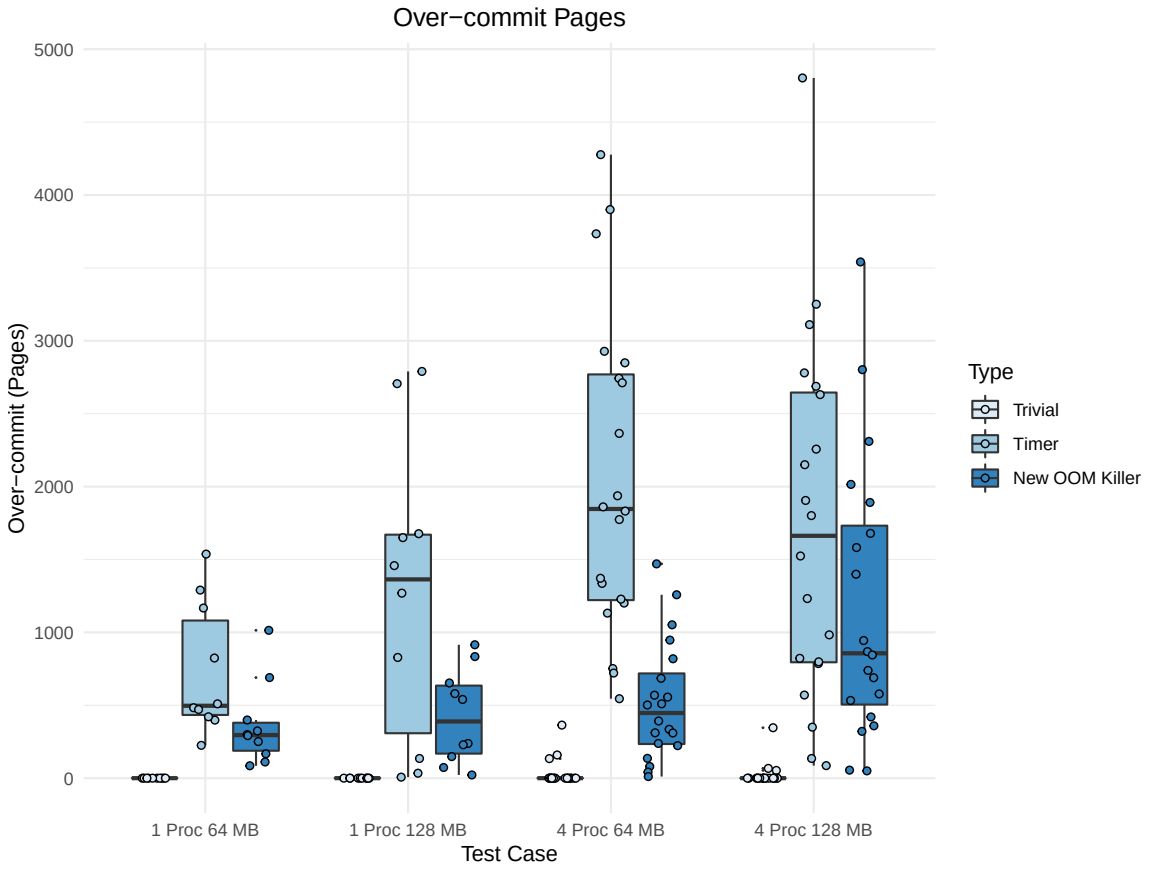

Figure 9: Evaluation of Memory Over-Committing

As seen in Figure 9, "Trivial" implementation observes nearly no memory over-commit. This is the ideal situation. "Timer" implementation sees large over-commit, and a large variance. Our OOM killer reduces latency, as well as improves reliability (small variance), compared with the "Timer" implementation.

7 Related Works

There have already been several attempts for implementing a user-space OOM killer.

Facebook's *OOMd* [3] leverages cgroupsv2 and PSI to monitor the system. This approach is similar to event-tracing in this project. When the system is under memory pressure, oomd will kill some victim processes.

Another user-space OOM killer daemon, called *earlyoom* [4], checks available memory for given interval. This is similar to the timer-only trivial implementation.

Nohang [1] is a user-space daemon to prevent kernel OOM killer from working, as it may kill some important process (e.g. Desktop Environment).

None of them is built specifically for limiting per-user memory usage.

8 Conclusion

In this project, we built a OOM killer with per-user memory limit support.

- **Technique** We combined an unreliable but low-latency user-space *OOM Killer Notifier*, a reliable but high-latency timer, and the original OOM killer, to reduce overhead and boost efficiency. From evaluation, we observed nearly no performance regression from original kernel, and the over-commit pages are very low.
- **Performance** By using Linux event-tracing facilities, atomic variables, hash tables and user-space data structures, we reduced lock contention and memory footprint.
- **Latency** We further reduced *OOM Killer Notifier* latency by early bootstrapping (soft limit).
- **User Interface** To make users aware of their memory usage, we introduced new `tmalloc`, or transactional memory allocation, which will return NULL if other user has already used up memory limit.

References

[1] Alexey Avramov. nohang. https://github.com/hakavlad/nohang.

[2] Jonathan Corbet. Low-level tracing plumbing. https://lwn.net/Articles/300992/.

[3] Facebook. oomd. https://github.com/facebookincubator/oomd.

[4] rfjakob. earlyoom. https://github.com/rfjakob/earlyoom.

[5] Alessandro Rubini and Jonathan Corbet. *Linux device drivers*. " O'Reilly Media, Inc.", 2001.

[6] Theodore Ts'o. The Linux Kernel Documentation event tracing. https://www.kernel.org/doc/html/v4.18/trace/events.html.