Report for Project 2, CS356

Ziqi Zhao 518030910211
bugenzhao@sjtu.edu.cn

June 18, 2020

Contents

1 Introduction

1.1 Objectives

In this project, we were asked to learn more about Linux kernel development, the necessary techniques for hacking around the kernel, the basic methods of information sharing within the kernel, as well as the principle of OOM killer. Based on these, we were asked to implement a new OOM killer for user-level memory limitations which should performs as efficiently as possible.

1.2 Report Organization

The remaining parts of the report are organized as follows.

- In Section 2, I explain how my system call works in details. In particular,
    - I implement the **full functionalities** for system call to add, delete, and update the memory limitations and the allowed exceeding time.
    - I use **linked list** to store the data, allowing arbitrary number of limitations.
    - I apply **rwlock** carefully to avoid hazards of concurrency, which performs well in this producer-consumer pattern.
- In Section 3, I explain how the original OOM killer is triggered.
- In Section 4, I explain the design and implementation of my new OOM killer and provide the results of performance benchmark. In particular,
    - I apply a **three-step strategy** in the OOM killer, which ensures the success of kill while avoiding the overhead of repeated killing.
    - I design a **heuristic killing strategy** and **hybrid trigger** by adopting the idea of time trigger to the allocation trigger, which results in a much better real-time performance than the basic allocation triggered one, but only brings the overhead of **less than 2%** for limitation-enabled programs.
    - I implement the killer very cleanly as an **optional feature** in `Kbuild`, with only 2 lines of code added to the original kernel source `page_alloc.c` in total.
- In Section 5, I introduce some additional and useful development techniques learned in this project. In particular,
    - I use `rsync` to sync my hacking code to the kernel and make the workspace much cleaner, and write a `Makefile` to build, run, and test automatically.
- In Section 6, I record some of my feelings about the project and the challenges I've overcome, as well as the acknowledgements.
- In the last section, I list the references for this project.

Special Notes There are several special notes for testing or requirements in the report like me. Please pay more attention to them. Thanks!

2 System Call

2.1 Necessary Preparations

2.1.1 Data Structure

Linked list is designed to store arbitrary number of data entries while providing good performance for adding and deleting operations. The limitation data of my OOM killer is designed as a linked list using Linux kernel's list API, which is located at kernel/mm/bz_mm_limits.h.

Special Notes For better consistency with the original kernel code, I've made some slight changes on the name of variables. For example, I've renamed MMLimits to mm_limit_struct.

```
/* Struct definition of mm_limit's */
struct mm_limit_struct {
    uid_t uid;                       /* user id */
    unsigned long mm_max;            /* memory limit for this user in bytes */
    int paused;                      /* is killer for this user paused? */
    struct timer_list* timer;        /* killer timer for this user */
    unsigned long time_allow_exceed; /* time allowed to exceed the limit */
    unsigned long last_mm;
    unsigned long last_time;
    struct list_head list;           /* for linked list */
};
```

Some fields of the struct may be intuitive. For the other ones,

- timer: when a timer is set, the killer should be temporarily disabled for this user, which will occur if
  - a kill signal has just been sent, so we may ignore the user and wait for the process to exit in a period of time.
  - the heuristic part of killer thinks that the limitation is "safe" in next short time, so we may disable the checking for the user to *reduce the overhead*.
  - the user has exceeded the limitation, but an allowed exceeding time is set.
- paused: the indicator of whether the timer is paused for expiration, which is atomic under the protection of rwlock for the list.
- last_mm, last_time: the previous status record of the user, which is used by the heuristic to decide the "safety" of some user.
- list: the necessary field for kernel linked list.

The head of the linked list is defined as init_mm_limit in kernel/mm/bz_mm_limits.c, which stores nothing but is used as the entry point for list traversal.

```
/* initialization for the head of mm_limit list */
#define INIT_MM_LIMIT(mm_limit) { .list = LIST_HEAD_INIT(mm_limit.list) }
/* the head of mm_limit list */
struct mm_limit_struct init_mm_limit = INIT_MM_LIMIT(init_mm_limit);
```

For user space, I also create an mm_limit_user_struct for getting the info of limitation and allowed time by get_mm_limit system call, which is also located in kernel/mm/bz_mm_limits.h.

```
/* mm_limit info for user */
struct mm_limit_user_struct {
    unsigned long mm_max;
    unsigned long time_allow_exceed_ms;
};
```

2.1.2 Lock

It's really important to apply locks onto the data structure in kernel, since a procedure might be preempted, and even be called concurrently in a multiprocessor system. Many synchronization techniques and examples were mentioned in our lectures, and there are several provided by the Linux kernel, such as spinlock, mutex, and rwlock.

The main part we want to protect is the linked list and its contents. The operations on the list follow the pattern of *producer-consumer problem* — some kernel threads are traversing/reading the list, while the other ones are writing to the list. Linux kernel's rwlock is designed specifically for this pattern. Instead of mutex which only allows exactly one thread to access the data, rwlock allows multiple readers accessing the data simultaneously, which improves the performance much a lot.

Definition of a static rwlock is quite easy in kernel/mm/bz_mm_limits.c. Since the system call will access the list as well, I export the symbol mm_limit_rwlock to the kernel modules.

```
DEFINE_RWLOCK(mm_limit_rwlock);
EXPORT_SYMBOL(mm_limit_rwlock);
```

To lock or unlock rwlock, the kernel also provides several functions which look like <read/write>_[un]lock[_irq]. read_[un]lock.. is designed for reader, and write_[un]lock.. for writer. A very important point is that only the functions with suffix _irq disable the IRQ's during the critical section, which should always be chosen in the killer to **avoid deadlocks** in case that some other thread with higher priority traps into the handler of page fault simultaneously.

2.2 Basic Functionalities

I implement the **full functionalities** for system call to add, delete, and modify the memory limitations and the allowed exceeding time. In common/syscall_num.h, there are 3 macros defined for the # of system calls.

```
#define __NR_mm_limit      381
#define __NR_mm_limit_time 382
#define __NR_get_mm_limit  383
```

Specifically,

- to add a basic limitation for user with uid:
  `syscall(__NR_mm_limit, uid, limit_in_bytes);`

- to add a limitation for user with `uid` with allowed exceeding time:
  `syscall(__NR_mm_limit_time, uid, limit_in_bytes, allowed_time);`

- to modify the limitation or allowed exceeding time for user with `uid`:
  `syscall(__NR_mm_limit_time, uid, new_limit, new_time);`

- to remove the limitation for user with `uid`, using `ULONG_MAX` to represent "no limitation":
  `syscall(__NR_mm_limit, uid, ULONG_MAX);`

- to remove the allowed exceeding time for user with `uid`, using `0` to represent "no allowed exceeding time":
  `syscall(__NR_mm_limit_time, uid, limit_in_bytes, 0);`

- to get the limitation and the allowed exceeding time for user with `uid`:

  ```
  struct mm_limit_user_struct {
      unsigned long mm_max;
      unsigned long time_allow_exceed_ms;
  } buf;
  syscall(__NR_get_mm_limit, uid, &buf);
  ```

The system call `mm_limit` is actually a convenient sugar of `mm_limit_time`, which is reserved considering the compatibility of tester.

2.3 Init & Exit

For the sake of "clean", I decide to provide system calls by module again. Since the module may replace the system call functions on several different computer architectures, a more robust way to do this is find the `syscall_table` during the runtime instead of hard-coding the address.

Here is the implementation of my `find_syscall_table` function, which can successfully obtain the address `0xc000d8c4` by locating the `sys_close` by `__NR_close`.

```
/* find the syscall table address */
static unsigned long **find_syscall_table(void) {
    unsigned long offset;
    unsigned long **sct;

    /* find syscall table by __NR_close */
    for (offset = PAGE_OFFSET; offset < ULLONG_MAX; offset += sizeof(void *)) {
        sct = (unsigned long **)offset;
        if (sct[__NR_close] == (unsigned long *)sys_close) {
            printk(KERN_INFO "Found syscall table: %p\n", sct);
            return sct;
        }
    }

    printk(KERN_WARNING "Failed to find syscall table, use default value %p\n",
           DEFAULT_SYSCALL_TABLE);
    return (unsigned long **)DEFAULT_SYSCALL_TABLE;
}
```

After getting the address of the system call table, I save and replace 3 system calls in function mm_limit_init in the same way as Project 1. Besides, when the module exits, all system calls should be restored to the original one in function mm_limit_exit.

2.4 Setting Limitation Only

The set_mm_limit system call is simply a convenient sugar of set_mm_limit_time with allowed exceeding time set to 0, whose implementation is introduced in next section.

```
/* convenience: set_mm_limit system call, without time_allow_exceed */
static int set_mm_limit_syscall(uid_t uid, unsigned long mm_max) {
    return set_mm_limit_time_syscall(uid, mm_max, 0);
}
```

2.5 Setting Limitation and Allowed Time

The system call function set_mm_limit_time_syscall is the "heart" of setting the limitations. After it is called, it will do the following things:

- The motivation of the killer is to limit used memory by some Android App, which has a uid larger than 10000. So check whether the value of uid is legal first.

```
/* avoid illegal calling */
if (uid < 10000) {
    printk(KERN_ERR
           "*** Attempted to limit user with uid < 10000. Aborted. ***\n");
    return -EACCES;
}
```

- Then, we should check whether the limitation for user is already in the list. If so, we may need to *update* or *remove* the limitation and allowed time. To *remove*, carefully release the memory for the timer and the entry.

  Since we are gonna traverse the list and may modify it, we should get the lock using write_lock_irq(&rwlock).

  There is also a function named del_timer_sync which waits for the timer handler on other CPUs to return and then delete the timer. Considering that our AVD is not configured SMP, sleeping is not allowed in interrupt context, and there's no need to wait for the handler to return in this case, I choose to use del_timer instead.

```
/* check if the limit is already in the list */
write_lock_irq(&mm_limit_rwlock);
list_for_each_entry(p, &init_mm_limit.list, list) {
    if (p->uid == uid) {
        if (mm_max == ULONG_MAX) {
            /* remove */
            list_del(&p->list);
            del_timer(p->timer);
            kfree(p->timer);
            kfree(p);
            printk(KERN_INFO "*** Removed: uid=%u ***\n", p->uid);
        } else {
            /* update mm_max and time_allow_exceed */
            p->mm_max = mm_max;
            p->time_allow_exceed = time_allow_exceed;
            /* reset last record for heuristic */
            p->last_time = jiffies;
            p->last_mm = 0;
            printk(KERN_INFO
                    "*** Updated: uid=%u, mm_max=%lu, time_allow_exceed="
                    "%lu ***\n",
                    p->uid, p->mm_max, p->time_allow_exceed);
        }
        ok = 1;
        break;
    }
}
write_unlock_irq(&mm_limit_rwlock);
```

- If not found, we need to append an mm_limit_struct to the list. Note that we must use kmalloc to allocate memory for the new entry, and manually initialize each field in mm_limit_struct.

```
/* limit not found in list, add it */
if (!ok && mm_max != ULONG_MAX) {
    /* allocate a new mm_limit_struct */
    struct mm_limit_struct *tmp =
        kmalloc(sizeof(struct mm_limit_struct), GFP_KERNEL);
    if (!tmp) { return -ENOMEM; }

    tmp->uid = uid;       /* user id */
    tmp->mm_max = mm_max; /* memory limit */
    tmp->paused = 0;      /* killer paused flag */

    /* reset the last record for heuristic  */
    tmp->last_mm = 0;
    tmp->last_time = jiffies; /* NOW */

    /* init timer */
    tmp->timer = kmalloc(sizeof(struct timer_list), GFP_KERNEL);
    if (!tmp->timer) {
        kfree(tmp);
        return -ENOMEM;
    }
    init_timer(tmp->timer);
    tmp->time_allow_exceed = time_allow_exceed;

    /* add it to list */
    write_lock_irq(&mm_limit_rwlock);
    list_add(&tmp->list, &init_mm_limit.list);
    write_unlock_irq(&mm_limit_rwlock);

    printk(KERN_INFO
            "*** Added: uid=%u, mm_max=%lu, time_allow_exceed=%lu ***\n",
            uid, mm_max, time_allow_exceed);
    ok = 1;
}
```

2.6 Getting Limitation and Allowed Time

Sometimes it's useful to reveal the limitation and allowed exceeding time applied to some user. I implement a "getter" system call to provide this functionality, where the user gives the uid and the pointer to a buffer of mm_limit_user_struct.

The procedure is similar to the last section. The main differences are to use read_[un]lock_irq instead of write_[un]lock_irq to reduce the overhead, and to use copy_to_user to write the results back into user space. Please refer to mm_limit_syscall /mm_limit_syscall.c for more details.

2.7 Tests

2.7.1 Functionalities

I write a test program at killer_test/jni/killer_test.c to test the correctness of system calls, based on the assertion macro mentioned in the report of project 1.

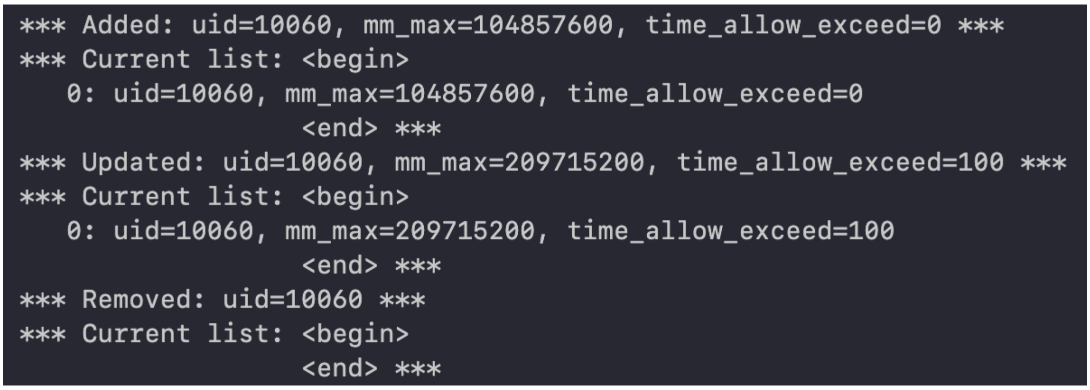
```
/* test assertion utility */
#define bugen_assert(case, lhs, op, rhs, format)                          \
    if (!((lhs)op(rhs))) {                                                \
        fprintf(stderr,                                                   \
                "%s:%d: Test case '" case "': ERROR: " #lhs " == " format \
                                          "\n",                           \
                __FILE__, __LINE__, (lhs));                               \
        exit(-1);                                                         \
    }
```

In syscall_test, I test several use cases for all system calls. For example,

```
/* test set */
ret = syscall(__NR_mm_limit, 10060, TEN_MB * 10);
bugen_assert("set", ret, ==, 0, "%d");

/* test get 1 */
ret = syscall(__NR_get_mm_limit, 10060, &buf);
bugen_assert("get 1", ret, ==, 0, "%d");
bugen_assert("get 1", buf.mm_max, ==, TEN_MB * 10, "%lu");

...
printf("Syscall test: all tests passed\n");
return 0;
```

By running with argument syscall, the tester reports that all tests are passed. At the same time, we can also check the logs from the system call module for the correctness, see Figure 1.

Special Notes Since I add more functionalities like *update*, *remove*, and the support of time_allow_exceed, I insert some verbs like '*Added*', '*Updated*', and '*Removed*' to the



```
*** Added: uid=10060, mm_max=104857600, time_allow_exceed=0 ***
*** Current list: <begin>
   0: uid=10060, mm_max=104857600, time_allow_exceed=0
                  <end> ***
*** Updated: uid=10060, mm_max=209715200, time_allow_exceed=100 ***
*** Current list: <begin>
   0: uid=10060, mm_max=209715200, time_allow_exceed=100
                  <end> ***
*** Removed: uid=10060 ***
*** Current list: <begin>
                  <end> ***
```

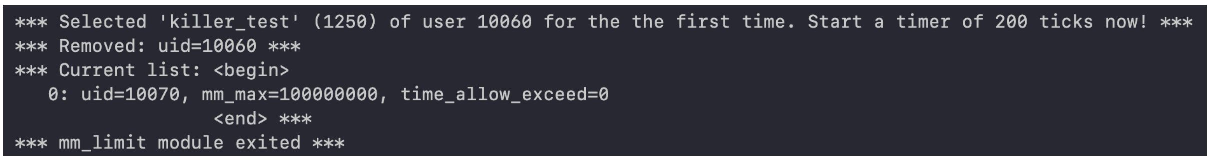
Figure 1: Tests for System Calls

2.7.2 Race

Recall that by setting the limit for a user to ULONG_MAX, we remove the entry from the list and are supposed to release its resources. Consider a situation where the corresponding timer is running but we are about to free the data entry — the pointer to the callback function might be an unknown value, which results in undefined behaviors. Thus, we must carefully delete the timer before freeing the memory.

I write a test again in killer_test/jni/killer_test.c, which can be run with argument race. First we attach a limitation with allowed exceeding time of 2000 ms, then reach it by malloc but remove the limitation after 1 second. The result is tested to be expected that the kernel does not check and kill the process after another 1 second, and the program can print “passed” correctly then exit with 0. The kernel output in Figure 2 also claims this.

```c
#define LIMIT (TEN_MB * 5)
char *p;

void race_timer_handler(int sig) {
    static int count = 0;
    if (count == 0) {
        syscall(__NR_mm_limit, getuid(), ULONG_MAX);
        alarm(2);
        count += 1;
    } else {
        printf("Race test: passed\n");
        exit(0);
    }
}

int race_test(void) {
    syscall(__NR_mm_limit_time, getuid(), LIMIT, 2000);
    p = malloc(LIMIT);
    memset(p, 0x88, LIMIT);

    signal(SIGALRM, &race_timer_handler);
    alarm(1);
    while (1)
        ;
}
```



```
*** Selected 'killer_test' (1250) of user 10060 for the the first time. Start a timer of 200 ticks now! ***
*** Removed: uid=10060 ***
*** Current list: <begin>
   0: uid=10070, mm_max=100000000, time_allow_exceed=0
                  <end> ***
*** mm_limit module exited ***
```

Figure 2: Race Tests in Kernel Output

2.8 Answers to Questions

1. Where to define the struct?

   In a header file `kernel/include/linux/bz_mm_limits.h`. Remember to add the sources to Makefile and export them to the module through `EXPORT_SYMBOL()`.

2. How to make and use a linked list in kernel?

   Add a field of `struct list_head` to the struct, and make a head of the list. Use it with `list_add`, `list_del`, `list_for_each_entry` and so on.

3 Android OOM Killer

3.1 Why to Kill

Some programs using large amount of memory may cause the system out-of-memory, where the kernel must kill them in time to avoid crashing itself. The OOM killer checks it the system is truly out of memory, and there exists a process to be the victim, then kill it to get more memory spaces.

3.2 When to Kill

Every time a process traps in a page fault will cause the kernel to allocate a new page for it. The kernel serves the page fault with the procedure:

- `__alloc_pages`, which calls `__alloc_pages_nodemask`.
- If the first attempt by `get_page_from_freelist` fails, it calls `__alloc_pages_slowpath`.
- If it fails to make any progress reclaiming or getting a free page again, it may call `__alloc_pages_may_oom`.
- If it fails to get a page from free list eventually, it will call `out_of_memory`.
- In `out_of_memory`, the kernel will first go through the following checks, and kill a process only if
  - no enough swap space,
  - 10 failures in last 5 seconds,
  - and it failed within the last second.

Next, the kernel will select a victim process to kill.

3.3 Which to Kill

For each task, the kernel firstly checks whether it is not adequate as candidate victim task, for example a kernel thread, with function `oom_unkillable_task`. Then the kernel calculates the “badness” of each running task in `oom_badness`, which is basically based on the RSS — a process that is using a large amount of memory but is not that long lived is likely to be chosen.

Besides, there is also a score named `oom_score_adj` for each process to increase or decrease the value of the badness, or even disable the killer. Root processes or processes with `CAP_SYS_ADMIN` or `CAP_SYS_RAWIO` capabilities will get some bonus.

3.4 How to Kill

After the victim is selected, the kernel walks the task list again to find all processes that share the same `mm_struct` as the selected one, in `oom_kill_process()`. Then send a `SIGTERM` or `SIGKILL` to them, based on whether they are critical or require exiting cleanly.

4 Bugen's OOM Killer

4.1 Hybrid Trigger

A basic idea of when to trigger off the killer's worker function is whenever a page allocation is requested by some userspace program, i.e., add a calling to the killer in function `__alloc_pages()` at `hacking/mm/page_alloc.c`. This approach brings the best real-time — since we will check all running tasks and the limitation list every time a program's allocated memory is to be increased, we can detect the out-of-memory as soon as possible. However, this results in a poor allocation performance as well in the same reason.

Another idea is to check the allocated memory every period of time T, whose overhead is only up to T and independent of the frequency of the page allocation, by Linux daemon for example. This approach won't affect the performance very much, but may lead to poor real-time, since the time period T is hard to determine ahead of time. A large T will cause some processes never be detected, while a precise one requires Linux's *high resolution timer* and may brings a much heavier overhead.

Based on the ideas of both approaches, I design a **hybrid trigger** combining the allocation trigger with the time trigger. Generally, it can be triggered by both page allocation and timer expiration. A lot of optimizations are also applied to reduce the number of costly operations like traversing the task list every time. For example,

- The worker will only check the current user by `current->uid`. If it is not in the limitation list, the worker will return immediately.

```
/* check if limit exists, uid represents current->uid here */
found = find_lock_mm_limit_struct(uid);
if (found == NULL) { return 0; }
```

- Recall that every user in `mm_limit_struct` has a `timer` and a `paused` flag, which has been mentioned in Section 2.1.1. If the timer is running or `paused` is true due to

  - a `SIGKILL` signal has just been sent, so we may ignore the user and wait for the process to exit in a period of time.
  - the heuristic part of killer thinks that the limitation is "safe" in next short time, so we may disable the checking for the user to *reduce the overhead*.
  - the user has exceeded the limitation, but an allowed exceeding time is set.

  the worker will also return immediately.

```
/* check if killer should wait/ignore this time */
if (found->paused) {
    write_unlock_irq(&mm_limit_rwlock);
    return 0;
}
```

- The expiration of a timer may also trigger off the killer, for example, to recheck the status with a higher **strictness level** of some user after sending a `kill` signal.

```
/* timer callback: time to check if exactly killed */
void bz_oom_kill_expires(unsigned long _uid) {
    uid_t uid = (uid_t)_uid;
    printk(KERN_INFO
           "*** Time for checking if user %u is killed. ***\n", uid);
    set_mm_limit_paused(uid, 0);  /* reset paused flag */
    bz_oom_worker(uid, 0, 2);     /* retry with strict = 2 */
}
```

The key implementation of **hybrid trigger** can be described by two strategies:

- **Three-step Strategy** divides the killing behavior into 3 strictness levels, as the third argument of the worker function `int bz_oom_worker(uid_t uid, int order , int strict)`, which will result in different force level and paused times.

  The detailed explanation will be introduced in Section 4.3.

- **Heuristic Killing Strategy** calculates the "safety" of some user by combining its current memory usage, historical memory usage, and remaining memory limit. By setting the timer and the `paused` flag to pause the killer for this user in next short time, it greatly reduces the number of killer calling and improves the performance.

  The detailed explanation will be introduced in Section 4.4.

Before introducing these two strategies, let us make some preparations and create some helper functions to make the code of killer's worker much clearer.

4.2 Necessary Preparations

There're 4 helper functions in `hacking/mm/bz_mm_limits.c`.

- `find_lock_mm_limit_struct`: find the `mm_limit_struct` for the given `uid` and lock the list. Since the purpose of this function is general, we must assume that the caller will write to the list and use `write_[un]lock_irq`.

```
/* find the mm_limit_struct for the given uid and lock the list, if not found,
 * return NULL with list unlocked */
struct mm_limit_struct *find_lock_mm_limit_struct(uid_t uid) {
    struct mm_limit_struct *p;

    /* lock the list */
    write_lock_irq(&mm_limit_rwlock);
    list_for_each_entry(p, &init_mm_limit.list, list) {
        if (p->uid == uid) { return p; }
    }
    /* uid not found */
    write_unlock_irq(&mm_limit_rwlock);
    return NULL;
}
```

- `set_mm_limit_paused`: set the killer's paused flag for the given `uid` to `v`.

```
/* set the killer's paused state for the given uid to v */
int set_mm_limit_paused(uid_t uid, int v) {
    struct mm_limit_struct *p;

    /* lock the list */
    write_lock_irq(&mm_limit_rwlock);
    list_for_each_entry(p, &init_mm_limit.list, list) {
        if (p->uid == uid) {
            p->paused = v;
            write_unlock_irq(&mm_limit_rwlock);
            return 0;
        }
    }
    /* uid not found */
    write_unlock_irq(&mm_limit_rwlock);
    return -2;
}
```

- get_mm_limit_paused: get the killer's paused flag for the given uid. The code is very similar to the previous one and is omitted here.

- bz_start_timer: start a killer timer for the given uid. This function is also designed for general purpose — it will start a timer with the given callback function and the custom time. If custom time is zero, it will use the time in time_allow_exceed instead. Be careful that if the timer is already started, it mustn't be started again.

```
/* start a killer timer for the given uid, with the callback function, if
 * custom_time is 0, the default time_allow_exceed will be used;
 * return the expires time, or negative value if failed */
long long bz_start_timer(uid_t uid, void (*function)(unsigned long),
                         unsigned long custom_time) {
    struct mm_limit_struct *p;

    /* lock the list */
    write_lock_irq(&mm_limit_rwlock);
    list_for_each_entry(p, &init_mm_limit.list, list) {
        if (p->uid == uid) {
            /* either custom_time, or time_allow_exceed ? */
            unsigned time = custom_time ? custom_time : p->time_allow_exceed;

            if (p->paused) {
                /* the timer may have been started! */
                write_unlock_irq(&mm_limit_rwlock);
                return -3;
            }

            if (time == 0) {
                /* may not want to start the timer */
                write_unlock_irq(&mm_limit_rwlock);
                return -1;
            }

            init_timer(p->timer);                 /* init the timer */
            p->timer->expires = jiffies + time; /* set expiration time */
            p->timer->data = uid;               /* argument to the callback */
            p->timer->function = function;      /* callback */
            p->paused = 1;      /* killer should be paused from now */
            add_timer(p->timer); /* add timer to kernel */

            write_unlock_irq(&mm_limit_rwlock);
            return time;
        }
    }
    /* uid not found */
    write_unlock_irq(&mm_limit_rwlock);
    return -2;
}
```

4.3 Three-step Strategy

To improve the performance of killing and avoid repeated killing which is useless, I design a **three-step strategy** which divides the killing behavior into 3 strictness levels (`strict in [0,1,2]`) as an argument of the killer.

- **Strictness level 0** occurs on the first calling from `__alloc_pages_nodemask`. If the killer for this user is not `paused` and does detect an OOM,
  - One with `time_allow_exceed` set will be given the time, and the killer will be paused for the time. After that, killer will check the memory again with strictness level 1.
  - One without `time_allow_exceed` set will be sent `SIGKILL` by `send_sig` immediately, and the killer will be paused for a constant time period `WAIT_FOR_KILLING_TIME` ticks. After that, killer will check whether the process is killed with strictness level 2.

```
/* we are not strict: check if time_allow_exceed is set */
if (!strict) {
    /* try to start a timer of time_allow_exceed */
    long long timer_time = bz_start_timer(uid, &bz_oom_time_expires, 0);

    /* timer started successfully, ignore killing this time */
    if (timer_time > 0) {
        printk(KERN_INFO
                "*** Selected '%s' (%d) of user %u for the the first time. "
                "Start a timer of %lld ticks now! ***\n",
                selected->comm, selected->pid, uid, timer_time);
        return 0;
    }
    /* else, time_allow_exceed may not set, kill now! */
}

/* BEGIN KILLING */
```

Otherwise, if the user does not exceed the limitation, we will start the heuristic part to evaluate the “safety” and pause the killer for some time.

```
/* check if limit is not exceeded */
if ((sum_rss << PAGE_SHIFT) <= mm_max || !selected) {
    if (strict) {
        /* we are strict, but no task selected: might be killed already */
        printk(KERN_INFO "*** No process to select for user %u now. ***\n",
                uid);
    } else {
        /* heuristic part */
    }
    return 0;
}
```

- **Strictness level 1** occurs on the time expiration of users with `time_allow_exceed` set. The killer for this user must not be `paused` now, and if it does detect an OOM *again*,
  - The process will be sent `SIGKILL` by `send_sig` immediately, and the killer will be paused for a constant time period `WAIT_FOR_KILLING_TIME` ticks. After that, killer will check whether the process is killed with strictness level 2.

```
if (strict <= 1) {
    /* strict <= 1 means it is the first time we try to kill it */
    printk(KERN_ERR
            "*** Killing '%s' (%d) of user %u since it is selected. ***\n",
            selected->comm, selected->pid, uid);
    printk(KERN_ERR
            "*** uid=%u, uRSS=%lupages=%lubytes, mm_max=%lubytes; pid=%u, "
            "pRSS=%lupages=%lubytes ***\n",
            uid, sum_rss, sum_rss << PAGE_SHIFT, mm_max, selected->pid,
            max_rss, max_rss << PAGE_SHIFT);
    set_tsk_thread_flag(selected, TIF_MEMDIE);

    /* send SIGKILL */
    send_sig(SIGKILL, selected, 0);

    /* start a timer to wait for the selected to be killed and set paused
     * flag. I.e., during the next WAIT_FOR_KILLING_TIME, the killer will
     * ignore this user, and after that killer will immediately check if it
     * is killed */
    bz_start_timer(uid, bz_oom_kill_expires, WAIT_FOR_KILLING_TIME);
} else {
    /* ... */
}
```

- **Strictness level 2** occurs in `WAIT_FOR_KILLING_TIME` ticks after `SIGKILL` is sent to some user. The killer for this user must not be `paused` now, and if it *still* detects an OOM, which may imply the process is not successfully killed,
  - The process will be sent `SIGKILL` by `force_sig` immediately, and the killer will be paused for a constant time period `WAIT_FOR_KILLING_TIME` ticks. After that, killer will repeat checking whether the process is killed with strictness level 2, until there's no OOM for this user.

```
if (strict <= 1) {
    /* ... */
} else {
    /* strict = 2, we have tried to kill it but it fails, retrying... */
    printk(KERN_ERR "*** Force-killing '%s' (%d) of user %u. ***\n",
            selected->comm, selected->pid, uid);

    /* send a force SIGKILL */
    force_sig(SIGKILL, selected);

    /* start a timer to wait for the selected to be killed and set paused
     * flag. I.e., during the next WAIT_FOR_KILLING_TIME, the killer will
     * ignore this user, and after that killer will immediately check if it
     * is killed */
    /* in this case, the killer will keep retrying every after the period
     * until the process is killed successfully */
    bz_start_timer(uid, bz_oom_kill_expires, WAIT_FOR_KILLING_TIME);
}
```

By attaching a strictness level, **three-step strategy** ensures that every process causing OOM is eventually killed, and avoid the useless repeated killings as well as reduce the performance overhead.

4.4 Heuristic Killing Strategy

In last section, we have argued that there's no need to repeat killing after a `SIGKILL` is sent. On the other hand, It's also quite unnecessary to check the memory every time `__alloc_pages_nodemask` is called. Suppose one user has a very high memory limitation but always use a little memory at a time, then the frequent checks are totally useless under this circumstance.

I design a **heuristic killing strategy** which is able to tune the frequency of the killer calling *dynamically*, according to current memory usage, historical memory usage, and remaining memory limit. The heuristic will evaluate the "safety" of some user, and again, start a timer with dynamic time period `wait_time` then pause the killer **for him**. Only after the timer expires, will the killer be re-enabled and evaluate again.

The basic principles of **heuristic killing strategy** is listed as the follows.

- Referring to the idea of the CPU burst prediction in Short-Job-First scheduling algorithm, we may estimate the memory allocation speed by the amount of memory allocation over time, and then estimate the expected `wait_time`. To simplify the computing procedure in kernel, `wait_time` is calculated by:

$$\texttt{wait\_time} = \frac{\texttt{current\_time} - \texttt{last\_time}}{\texttt{current\_mem} - \texttt{last\_mem}} \times (\texttt{limit} - \texttt{current\_mem})$$

  Note that in order to get the right result, a 64-bit division function `do_div(a, b)` is required in kernel.

- Based on the allocation speed from experimental measurements, `wait_time` should be lower than `(limit - current_time) >> 22`.

- Finally, we give `wait_time` an upper bound of `HZ / 2 = 500ms` for safety.

Here is the core code of the heuristic killing strategy. It is very simple in a way, but is also proved to be very efficient.

```
/* check if limit is not exceeded */
if ((sum_rss << PAGE_SHIFT) <= mm_max || !selected) {
    if (strict) {
        /* ... */
    } else {
        /* we are not strict: start the heuristic part to decide the next
            * checking time*/
        /* expr: 128 MB, 0.8 secs; set: 4 MB, 1 tick = 0.01 secs */
        unsigned long wait_time = HZ / 2;
        unsigned long mm_rem = mm_max - (sum_rss << PAGE_SHIFT);
        unsigned long elapsed = jiffies - last_time;

        /* if memory is increasing: decide wait_time based on its speed */
        if ((sum_rss << PAGE_SHIFT) > last_mm) {
            unsigned long long a = (unsigned long long)mm_rem * elapsed;
            unsigned long long b = (sum_rss << PAGE_SHIFT) - last_mm;
            do_div(a, b); /* 64-bit division */
            wait_time = (unsigned long)a;
        }

        /* bound the wait_time, should never be too large */
        wait_time = min3(wait_time, (unsigned long)(mm_rem >> 22),
                        (unsigned long)HZ / 2);

        if (wait_time > 0) {
            /* start a timer with callback which reset paused */
            bz_start_timer(uid, &bz_oom_reset_paused, wait_time);
            printk(KERN_INFO
                    "*** Pause checking for user %u for %lu ticks. ***\n",
                    uid, wait_time);
        }
    }
    return 0;
}
```

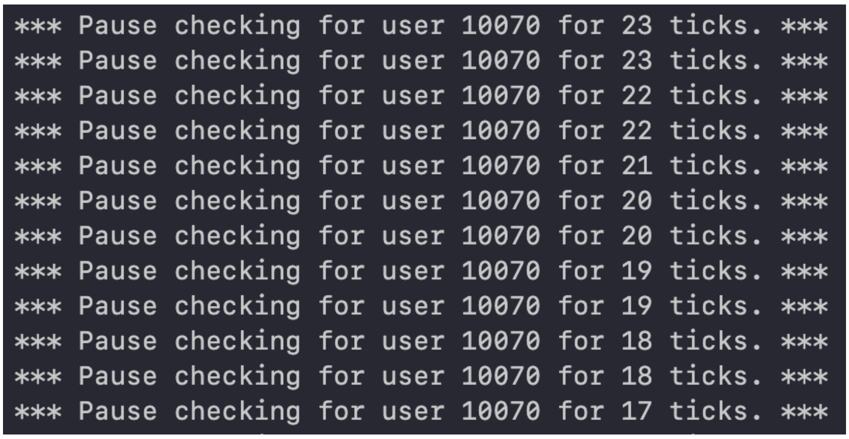
After the `wait_time` is decided to be larger than 0, a timer with the callback which resets the `paused` field will be started. Therefore, in the next `wait_time` ticks, this user will be completely ignored by the killer.

By pausing the killer for some user on demand, the killer's overhead for memory-limited user decreases from 30% of naive version to **less than 2%**. For more detailed comparison, please refer to Section 4.6.3.

Special Notes Logs for the heuristic part are frequent and thus disabled by default for the sake of performance.

4.5 Cleanliness First

The build of the Linux kernel is well divided into modules, which can be configured by `Kbuild` and `Kconfig` provided by the Kernel Build System. Therefore, the kernel encourages us to add new functionalities as optional modules or features to the kernel



```
*** Pause checking for user 10070 for 23 ticks. ***
*** Pause checking for user 10070 for 23 ticks. ***
*** Pause checking for user 10070 for 22 ticks. ***
*** Pause checking for user 10070 for 22 ticks. ***
*** Pause checking for user 10070 for 21 ticks. ***
*** Pause checking for user 10070 for 20 ticks. ***
*** Pause checking for user 10070 for 20 ticks. ***
*** Pause checking for user 10070 for 19 ticks. ***
*** Pause checking for user 10070 for 19 ticks. ***
*** Pause checking for user 10070 for 18 ticks. ***
*** Pause checking for user 10070 for 18 ticks. ***
*** Pause checking for user 10070 for 17 ticks. ***
```

Figure 3: Heuristic Logs

sources in order to make the structure as clear as possible.

To follow the modularization of the kernel, let us add a new config entry to the top-level `Kconfig` file at `kernel/Kconfig` for the sake of simplicity.

```
# bugen's oom killer feature
config BUGEN_OOM_KILLER
    bool "Bugen's OOM Killer Feature"
    default y
```

Then, add the only 2 lines of code calling the killer in `__alloc_pages_nodemask` function at `kernel/mm/page_alloc.c:2437`. Note that we can compile this functionality optionally by checking the `CONFIG_BUGEN_OOM_KILLER` macro.

```
...
/* the only hacking code to the original kernel file */
#ifdef CONFIG_BUGEN_OOM_KILLER
    extern int bz_oom_worker(uid_t uid, int order, int strict);
    bz_oom_worker(current->cred->uid, order, 0);
#endif
...
```

Finally, add the list of additional sources to `kernel/mm/Makefile`. Again, make sure that these sources will be compiled and linked only if the feature of killer is enabled.

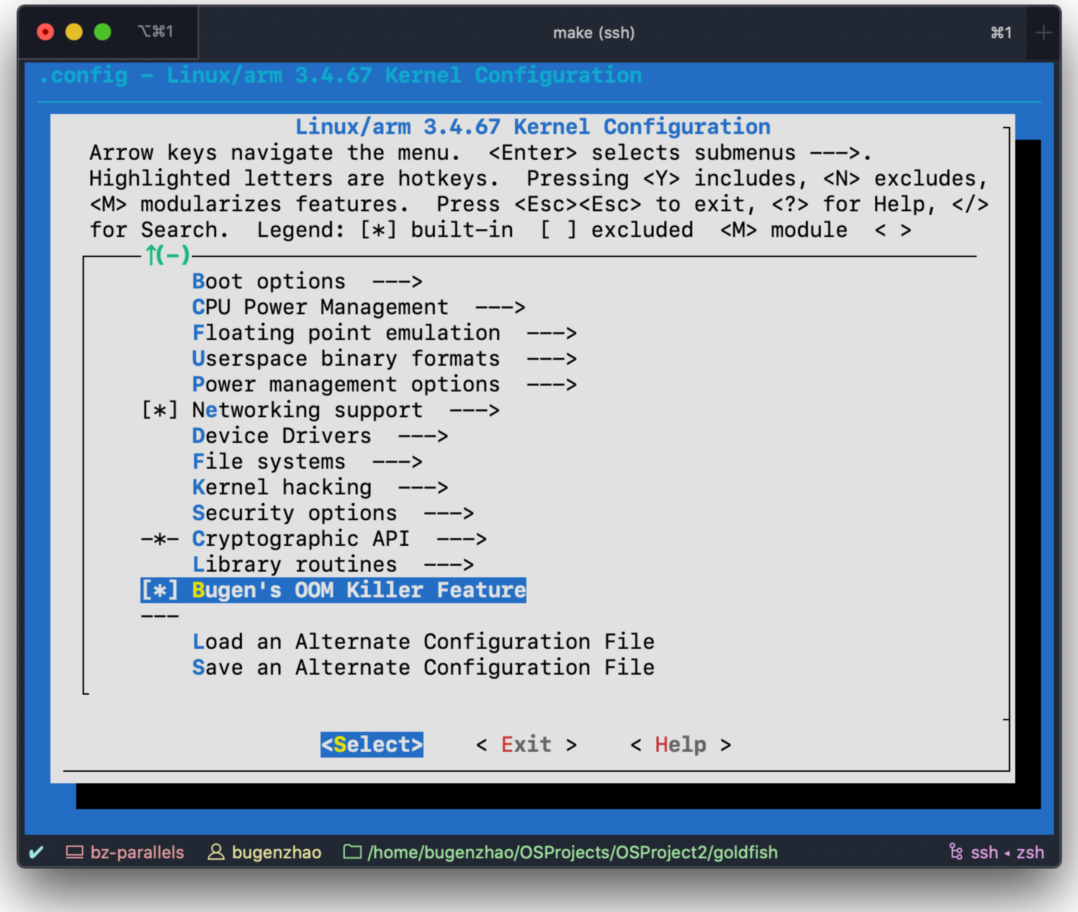
```
# bugen's oom killer feature
obj-$(CONFIG_BUGEN_OOM_KILLER) += bz_mm_limits.o bz_oom_killer.o
```

After these configurations, we complete adding the new killer cleanly to the kernel with little or no damage, as almost all of the code is *additive*. Finally, run `make menuconfig` to enable the new feature. (See Figure 4)

4.6 Tests

4.6.1 prj2_test

As Figure 5 shows, the killer succeeds to pass the tests in `prj2_test` with three different configurations — the child processes which make the user exceed the limitation



```
.config - Linux/arm 3.4.67 Kernel Configuration

                  Linux/arm 3.4.67 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >
 ┌───↑(-)──────────────────────────────────────────────────────────────┐
 │          Boot options  --->
 │          CPU Power Management  --->
 │          Floating point emulation  --->
 │          Userspace binary formats  --->
 │          Power management options  --->
 │      [*] Networking support  --->
 │          Device Drivers  --->
 │          File systems  --->
 │          Kernel hacking  --->
 │          Security options  --->
 │      -*- Cryptographic API  --->
 │          Library routines  --->
 │      [*] Bugen's OOM Killer Feature
 │      ---
 │          Load an Alternate Configuration File
 │          Save an Alternate Configuration File
 └

                  <Select>    < Exit >    < Help >
```

Figure 4: `make menuconfig`

are killed and fail to print `finish malloc`.

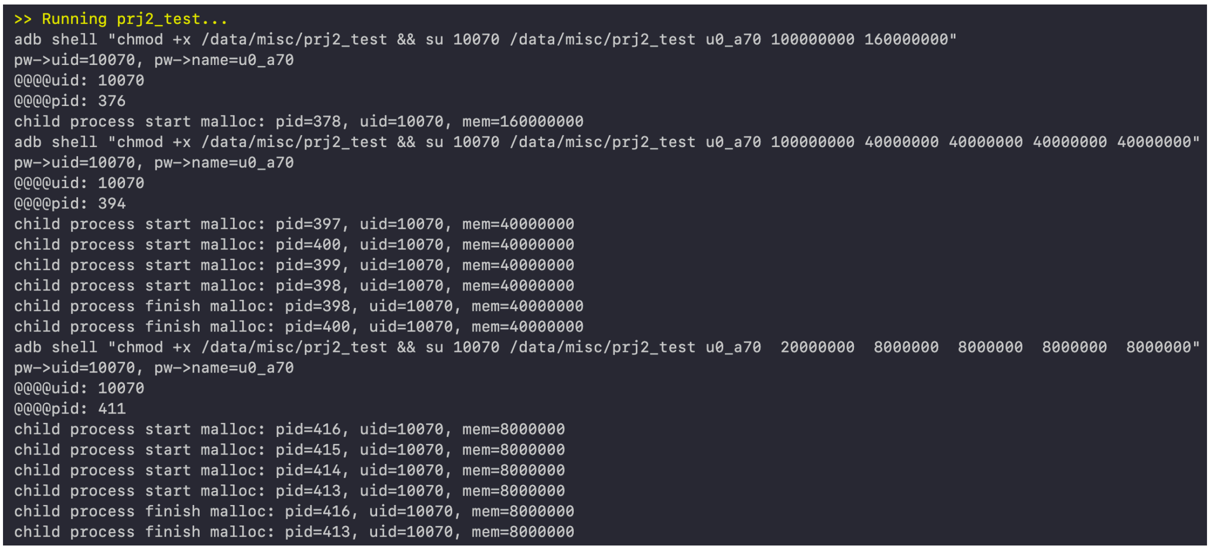
On the other hand, the logs in kernel output show that the **three step strategy** works. No redundant killings are performed or printed, see Figure 6.

Special Notes I've added the prefix and postfix '***' to make the output more visible. Besides, I make the required output in two units: *page* and *bytes*, for clearer comparison with other data.

4.6.2 Allowed Exceeding Time

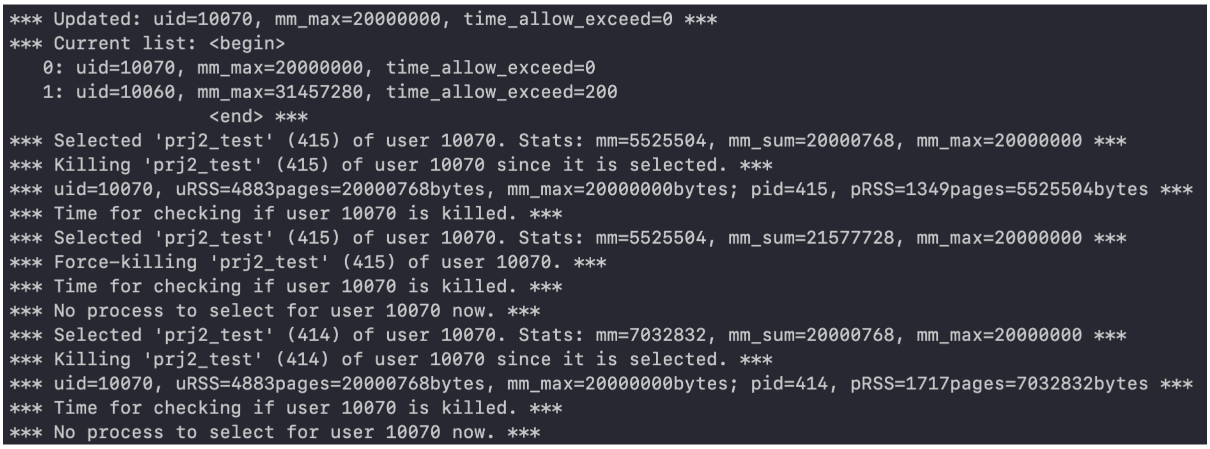
The original tester provided with the project handout does not test the behaviors with allowed exceeding time set. In my tester `killer_test`, I also write some tests for these cases.

For example, let us set the limitation and the time to be 50 MB and 1000 ms. When the allocation is finished, I start a timer with `SIGALRM` handler to print current elapsed time. I free some memory at 600 ms, thus the process isn't killed at 1000 ms. At 1200 ms I allocate it again, then the process is eventually killed at (1200+1000) ms. Figure 7 and 8 show the output of both process and kernel.



```
>> Running prj2_test...
adb shell "chmod +x /data/misc/prj2_test && su 10070 /data/misc/prj2_test u0_a70 100000000 160000000"
pw->uid=10070, pw->name=u0_a70
@@@@uid: 10070
@@@@pid: 376
child process start malloc: pid=378, uid=10070, mem=160000000
adb shell "chmod +x /data/misc/prj2_test && su 10070 /data/misc/prj2_test u0_a70 100000000 40000000 40000000 40000000 40000000"
pw->uid=10070, pw->name=u0_a70
@@@@uid: 10070
@@@@pid: 394
child process start malloc: pid=397, uid=10070, mem=40000000
child process start malloc: pid=400, uid=10070, mem=40000000
child process start malloc: pid=399, uid=10070, mem=40000000
child process start malloc: pid=398, uid=10070, mem=40000000
child process finish malloc: pid=398, uid=10070, mem=40000000
child process finish malloc: pid=400, uid=10070, mem=40000000
adb shell "chmod +x /data/misc/prj2_test && su 10070 /data/misc/prj2_test u0_a70  20000000  8000000  8000000  8000000  8000000"
pw->uid=10070, pw->name=u0_a70
@@@@uid: 10070
@@@@pid: 411
child process start malloc: pid=416, uid=10070, mem=8000000
child process start malloc: pid=415, uid=10070, mem=8000000
child process start malloc: pid=414, uid=10070, mem=8000000
child process start malloc: pid=413, uid=10070, mem=8000000
child process finish malloc: pid=416, uid=10070, mem=8000000
child process finish malloc: pid=413, uid=10070, mem=8000000
```

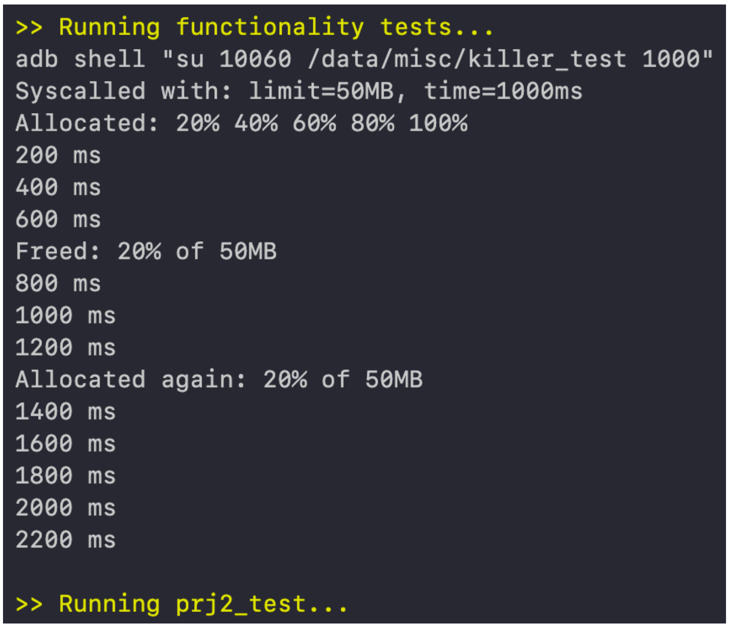
Figure 5: `prj2_test` in Standard Output



```
*** Updated: uid=10070, mm_max=20000000, time_allow_exceed=0 ***
*** Current list: <begin>
   0: uid=10070, mm_max=20000000, time_allow_exceed=0
   1: uid=10060, mm_max=31457280, time_allow_exceed=200
                  <end> ***
*** Selected 'prj2_test' (415) of user 10070. Stats: mm=5525504, mm_sum=20000768, mm_max=20000000 ***
*** Killing 'prj2_test' (415) of user 10070 since it is selected. ***
*** uid=10070, uRSS=4883pages=20000768bytes, mm_max=20000000bytes; pid=415, pRSS=1349pages=5525504bytes ***
*** Time for checking if user 10070 is killed. ***
*** Selected 'prj2_test' (415) of user 10070. Stats: mm=5525504, mm_sum=21577728, mm_max=20000000 ***
*** Force-killing 'prj2_test' (415) of user 10070. ***
*** Time for checking if user 10070 is killed. ***
*** No process to select for user 10070 now. ***
*** Selected 'prj2_test' (414) of user 10070. Stats: mm=7032832, mm_sum=20000768, mm_max=20000000 ***
*** Killing 'prj2_test' (414) of user 10070 since it is selected. ***
*** uid=10070, uRSS=4883pages=20000768bytes, mm_max=20000000bytes; pid=414, pRSS=1717pages=7032832bytes ***
*** Time for checking if user 10070 is killed. ***
*** No process to select for user 10070 now. ***
```

Figure 6: `prj2_test` in Kernel Output

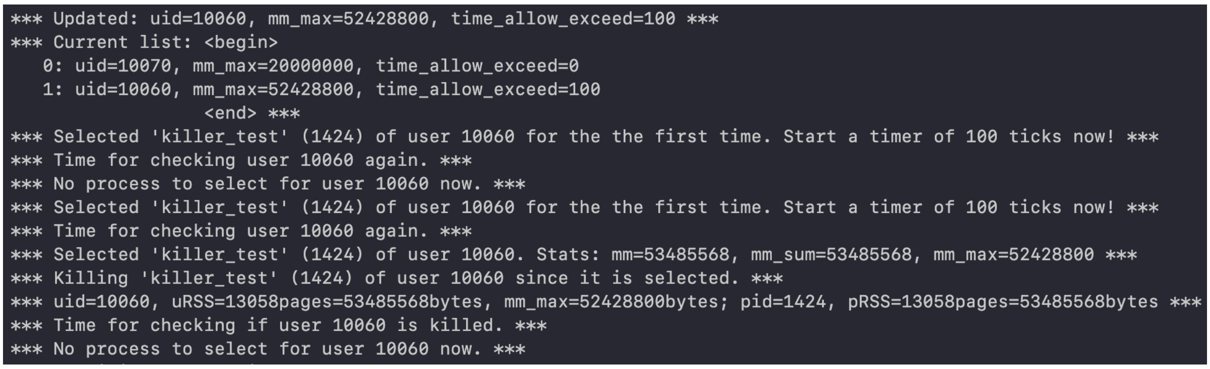
```c
#define LIMIT (TEN_MB * 5)
char *p;

void timer_handler(int sig) {
    static int count = 0;
    printf("%d ms\n", ++count * 200);
    if (count == 3) {
        free(p);
        printf("Freed: 20%% of %dMB\n", LIMIT >> 20);
    }
    if (count == 6) {
        p = malloc(LIMIT / 5);
        memset(p, 0x88, LIMIT / 5);
        printf("Allocated again: 20%% of %dMB\n", LIMIT >> 20);
    }
}

int main(int argc, char **argv) {
    int i, time = 0;
    struct itimerval timer;
    int count = 0;

    ...
    timer.it_value.tv_sec = 0;
    timer.it_value.tv_usec = 200000;
    timer.it_interval.tv_sec = 0;
    timer.it_interval.tv_usec = 200000;
    signal(SIGALRM, &timer_handler);

    /* start testing */
    printf("Allocated: ");
    for (count = 1; count <= 5; count++) {
        p = malloc(LIMIT / 5);
        memset(p, 0x88, LIMIT / 5);
        printf("%d%% ", count * 20);
    }
    printf("\n");
    setitimer(ITIMER_REAL, &timer, NULL);

    while (1)
        ;

    printf("SHOULD NEVER REACH HERE: Bye\n");
    exit(0);
}
```



```
>> Running functionality tests...
adb shell "su 10060 /data/misc/killer_test 1000"
Syscalled with: limit=50MB, time=1000ms
Allocated: 20% 40% 60% 80% 100%
200 ms
400 ms
600 ms
Freed: 20% of 50MB
800 ms
1000 ms
1200 ms
Allocated again: 20% of 50MB
1400 ms
1600 ms
1800 ms
2000 ms
2200 ms

>> Running prj2_test...
```

Figure 7: Time Tests in Standard Output

Special Notes The system may be slow for a while after it just starts up. As the allowed exceeding time is actually the natural time instead of CPU time, you may need to wait a moment in order to make sure to get the expected behaviors shown in the figures.



```
*** Updated: uid=10060, mm_max=52428800, time_allow_exceed=100 ***
*** Current list: <begin>
   0: uid=10070, mm_max=20000000, time_allow_exceed=0
   1: uid=10060, mm_max=52428800, time_allow_exceed=100
                  <end> ***
*** Selected 'killer_test' (1424) of user 10060 for the the first time. Start a timer of 100 ticks now! ***
*** Time for checking user 10060 again. ***
*** No process to select for user 10060 now. ***
*** Selected 'killer_test' (1424) of user 10060 for the the first time. Start a timer of 100 ticks now! ***
*** Time for checking user 10060 again. ***
*** Selected 'killer_test' (1424) of user 10060. Stats: mm=53485568, mm_sum=53485568, mm_max=52428800 ***
*** Killing 'killer_test' (1424) of user 10060 since it is selected. ***
*** uid=10060, uRSS=13058pages=53485568bytes, mm_max=52428800bytes; pid=1424, pRSS=13058pages=53485568bytes ***
*** Time for checking if user 10060 is killed. ***
*** No process to select for user 10060 now. ***
```

Figure 8: Time Tests in Kernel Output

4.6.3 Performance

To measure the performance of killer, I also write some benchmarks in `killer_test/jni/killer_test.c`, which can be run with argument `performance`. Note that for the sake of accuracy, I use `clock()` to get the CPU time instead of natural time.

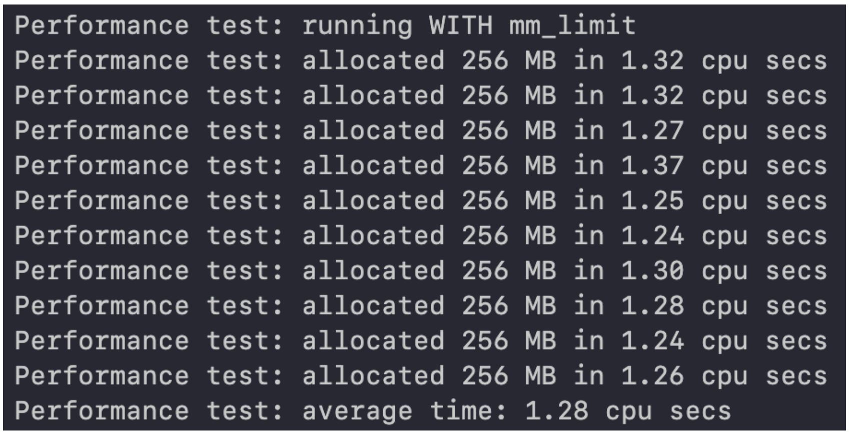
```c
/* test with mm_limit */
time_sum = .0;
syscall(__NR_mm_limit, getuid(), ULONG_MAX - 1);
printf("Performance test: running WITH mm_limit\n");
times = PERF_TIMES;
while (times--) {
    start = clock();
    p = malloc(PERF_SIZE);
    if (p != NULL) memset(p, 0x88, PERF_SIZE);
    end = clock();
    time_sum += (time = (end - start) / (double)CLOCKS_PER_SEC);
    printf("Performance test: allocated %u MB in %.2lf cpu secs\n",
            (PERF_SIZE >> 20), time);
    free(p);
}
printf("Performance test: average time: %.2lf cpu secs\n",
        time_sum / PERF_TIMES);
```

To test with the killer, make sure the "Bugen's OOM Killer Feature" is enabled in `Kconfig` and simply run `make testall`. To test without the killer, the feature should be disabled by `make -C goldfish menuconfig`, and then run the tests by `make nokiller`.

The results of performance test is listed as the follows:

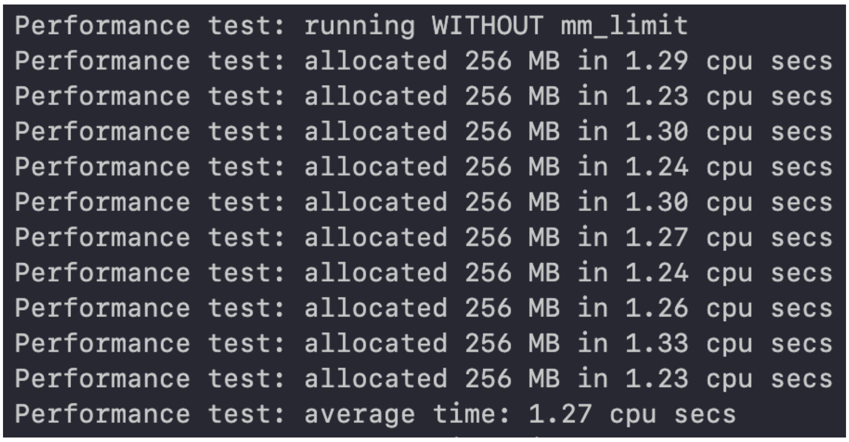
- Allocate with killer and limitation set for current user: Figure 9
- Allocate with killer but no limitation set for current user: Figure 10
- Allocate without killer: Figure 11

As the results show, thanks to the **Three-step Strategy** and **Heuristic Killing Strategy**, the overhead of adding the new OOM killer and limitation is very small, at around 2%. Compared to the naive version triggered whenever page allocation is requested, which has an overhead of 30%, our hybrid trigger really works.



```
Performance test: running WITH mm_limit
Performance test: allocated 256 MB in 1.32 cpu secs
Performance test: allocated 256 MB in 1.32 cpu secs
Performance test: allocated 256 MB in 1.27 cpu secs
Performance test: allocated 256 MB in 1.37 cpu secs
Performance test: allocated 256 MB in 1.25 cpu secs
Performance test: allocated 256 MB in 1.24 cpu secs
Performance test: allocated 256 MB in 1.30 cpu secs
Performance test: allocated 256 MB in 1.28 cpu secs
Performance test: allocated 256 MB in 1.24 cpu secs
Performance test: allocated 256 MB in 1.26 cpu secs
Performance test: average time: 1.28 cpu secs
```

Figure 9: With Killer, With Limitation



```
Performance test: running WITHOUT mm_limit
Performance test: allocated 256 MB in 1.29 cpu secs
Performance test: allocated 256 MB in 1.23 cpu secs
Performance test: allocated 256 MB in 1.30 cpu secs
Performance test: allocated 256 MB in 1.24 cpu secs
Performance test: allocated 256 MB in 1.30 cpu secs
Performance test: allocated 256 MB in 1.27 cpu secs
Performance test: allocated 256 MB in 1.24 cpu secs
Performance test: allocated 256 MB in 1.26 cpu secs
Performance test: allocated 256 MB in 1.33 cpu secs
Performance test: allocated 256 MB in 1.23 cpu secs
Performance test: average time: 1.27 cpu secs
```

Figure 10: With Killer, Without Limitation

4.7 Answers to Questions

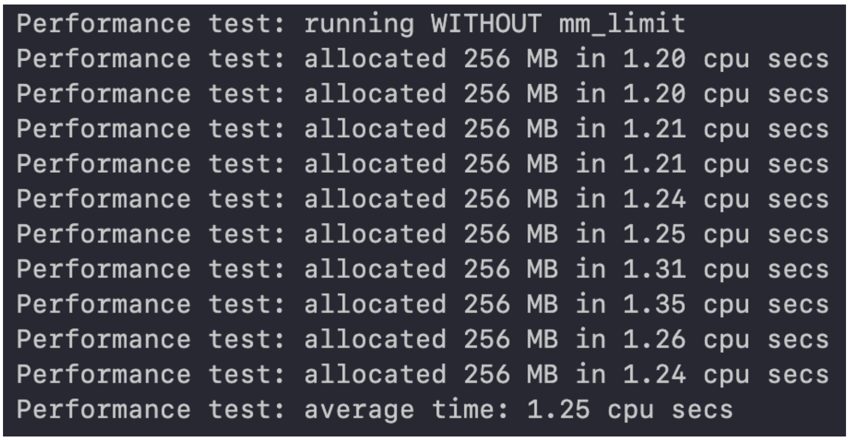
1. How to get the `uid`?

   For current user, get it through `current_cred->uid` or `current->cred->uid`. For some given `task`, get it through `task->cred->uid`.

2. How to get the RSS for a task?

   Remember to get the valid `mm_struct` through `find_lock_task_mm` first. RSS consists of several parts, and is in pages which should be left shift for `PAGE_SHIFT` bits to get the value in bytes.

```c
unsigned long rss;

/* for safer mm access */
struct task_struct *p = find_lock_task_mm(task);
if (!p) continue;

/* get rss of task */
rss = get_mm_rss(p->mm);                      /* basic */
rss += p->mm->nr_ptes;                        /* page table */
rss += get_mm_counter(p->mm, MM_SWAPENTS); /* swap */
```



```
Performance test: running WITHOUT mm_limit
Performance test: allocated 256 MB in 1.20 cpu secs
Performance test: allocated 256 MB in 1.20 cpu secs
Performance test: allocated 256 MB in 1.21 cpu secs
Performance test: allocated 256 MB in 1.21 cpu secs
Performance test: allocated 256 MB in 1.24 cpu secs
Performance test: allocated 256 MB in 1.25 cpu secs
Performance test: allocated 256 MB in 1.31 cpu secs
Performance test: allocated 256 MB in 1.35 cpu secs
Performance test: allocated 256 MB in 1.26 cpu secs
Performance test: allocated 256 MB in 1.24 cpu secs
Performance test: average time: 1.25 cpu secs
```

Figure 11: Without Killer

3. How to select the victim process?

   Just traverse all processes, get their RSS's, and select the one with the largest. Here I ignore the tasks with flag PF_EXITING, which may reduce the probability of repeated killing.

```c
/* get tasks of the user */
read_lock_irq(&tasklist_lock);
for_each_process(task) {
    if (task->cred->uid == uid) {
        /* for safer mm access */
        struct task_struct *p = find_lock_task_mm(task);
        if (!p) continue;

        /* get rss of task */
        rss = get_mm_rss(p->mm);                     /* basic */
        rss += p->mm->nr_ptes;                       /* page table */
        rss += get_mm_counter(p->mm, MM_SWAPENTS); /* swap */

        /* add the pages that are about to allocate */
        if (p->pid == current->pid) { rss += 1 << order; }

        /* valid if the task is not exiting */
        if (!(p->flags & PF_EXITING)) {
            sum_rss += rss;
            if (rss > max_rss) {
                selected = p;
                max_rss = rss;
            }
        }

        task_unlock(p);
    }
}
read_unlock_irq(&tasklist_lock);
```

4. How to kill the processes?

   Send a SIGKILL signal to some process through send_sig(SIGKILL, selected,

1) will kill it, where 1 means that the signal is from the kernel, i.e., SEND_SIG_PRIV . Or, send a forced signal by force_sig(SIGKILL, selected), which is a sugar of SEND_SIG_FORCED.

Tasks killed by OOM killer should be set the TIF_MEMDIE flag.

Tasks with PF_KTHREAD flag are kernel threads and should not be killed.

Besides, like the behaviors of original OOM killer, the tasks which share the same mm_struct with that of the victim should also be killed.

```
/* kill the processes sharing the same memory with 'selected' */
read_lock_irq(&tasklist_lock);
for_each_process(task) {
    if (task->mm == mm && !same_thread_group(task, selected) &&
        !(task->flags & PF_KTHREAD)) {
        task_lock(task);
        printk(KERN_ERR
                "*** Killing '%s' (%d) of user %u since it is sharing the "
                "same memory. ***\n",
                task->comm, task->pid, task->cred->uid);
        set_tsk_thread_flag(task, TIF_MEMDIE);
        send_sig(SIGKILL, task, 0); /* send SIGKILL */
        task_unlock(task);
        count++;
    }
}
read_unlock_irq(&tasklist_lock);
```

5. How to run commands as an other user?

   Simply run: su 10070 echo "Hello, world!".

6. How to do 64-bit division in the kernel?

   The kernel does not support raw '/' between 64-bit numbers, use r = do_div(a , b) instead. The remainder will be in r, and the quotient will be in a, which is quite like a division machine instruction.

5 Useful Techniques

5.1 rsync

There are too many source files in the kernel directory, so can we just focus on several files and make the workspace much cleaner? `rsync` is a utility for efficiently incremental file transfer, which can also detect the modification date between two files.

I make a symbolic link of the kernel to `./goldfish`, then create a new folder named `hacking` and put all of the modified or added sources into it with their original directory structure, where I mainly do my work. To synchronize the new sources to the kernel, simply run `rsync -u hacking goldfish` and then `make -C goldfish`. Or to be safer, list all modified sources explicitly and wrap the commands into the Makefile.

```
HACKING_LIST=                   \
Kconfig                         \
include/linux/bz_mm_limits.h    \
mm/bz_mm_limits.c               \
mm/Makefile                     \
mm/page_alloc.c                 \
mm/bz_oom_killer.c

rsync:
    for file in $(HACKING_LIST); do \
        rsync -u hacking/$$file goldfish/$$file; \
    done

kernel: rsync
    make -C goldfish -j4
```

Besides, organizing the code in this way is also great for submitting the work.

5.2 Makefile

It is complicated to sync, build, push, `insmod`, `chmod`, run and finally `rmmod` every time we make some minor changes on the sources. A Makefile is quite useful for this kind of fixed procedures.

```
emulator:
    emulator -avd ${AVD_NAME} -kernel ${KERNEL_ZIMG} -no-window -show-kernel

testall: clean
    make run | tee output.txt

build:
    @echo "\n\n\n\e[33m>> Building...\e[0m"
    make -C ${MODULE_DIR} KID=${KID}
    make -C ${KILLER_TEST_DIR}
    make -C ${PRJ2_TEST_DIR}

run: build upload
    @echo "\n\n\n\e[33m>> Running...\e[0m"
    adb shell "insmod ${MODULE_DEST} && lsmod"
    adb shell "chmod +x ${KILLER_TEST_DEST}"
    @echo "\n\e[33m>> Running syscall tests and performance tests...\e[0m"
    adb shell "su 10060 ${KILLER_TEST_DEST} test"
    ...
    @echo "\n\n\e[33m>> Cleaning...\e[0m"
    adb shell rmmod ${MODULE_DEST}
```

Also, ANSI escape color codes like `\e[33m` is also very useful in console logging `\e[0m`, especially there is a huge amount of info to check :-).

5.3 Bear

Bear (Build EAR, `https://github.com/rizsotto/Bear`) is a tool that generates a compilation database for clang tooling, which is also supported by mainstream editors like Visual Studio Code, for analyzing the structure and symbols of C or C++ project.

For example, to let Bear generate a compilation database for Linux kernel, simply add the word `bear` ahead of the `make` command, e.g. `bear make -j8`, and it will output a `json` file named `compile_commands.json`. The contents of the output are all actual building commands with their detailed arguments, like:

```
{
    {
        "arguments": [
            "arm-linux-androideabi-gcc",
            "-c",
            "-Wp,-MD,mm/.bz_oom_killer.o.d",
            "-nostdinc",
            ...
            "-o",
            "mm/bz_oom_killer.o",
            "mm/bz_oom_killer.c"
        ],
        "directory": "/home/bugenzhao/Android/kernel/goldfish",
        "file": "mm/bz_oom_killer.c"
    },
    {
        ...
    },
    ...
}
```

By offering this file to the editor, it will give a much more accurate navigation and auto-complete features, which lets us write kernel code without relying on separate cross references.

6 Conclusion

6.1 Challenges & Feelings

To be fair, it is not quite easy to get started with hacking on kernel for the first time — we are mainly talking about the operating system concepts in lectures, while Linux is a real, mature, huge, and industrial-use OS. Fortunately, the whole project is open-sourced and there are a lot of references for us to learn with and develop. As I delve deeper into the kernel sources, I gradually learn about some of the utilities and functions provided by the kernel, as well as a broader understanding of the sources' structure.

Developing in the kernel is much different from in the usual programs — we must keep in mind that there're lots of kernel threads running **asynchronically** and always be concern about the safety of the shared data, the deadlocks, potential memory leaks, and so on. Almost all of the challenges that I faced during the development is *kernel panic* or *kernel oops*, here are some of them.

- Kernel panic with "PREEMPT ARM".

  In fact, it's because I forgot to unlock the locks for the shared data. Since the critical section does not allow preemption, one who forgets to unlock might result in a "PREEMPT ARM" exception.

```
int set_mm_limit_paused(uid_t uid, int v) {
    struct mm_limit_struct *p;

    write_lock_irq(&mm_limit_rwlock);
    list_for_each_entry(p, &init_mm_limit.list, list) {
        if (p->uid == uid) {
            p->paused = v;
            /* I FORGOT TO UNLOCK HERE!!! */
            return 0;
        }
    }
    write_unlock_irq(&mm_limit_rwlock);
    return -2;
}
```

  This is a trivial bug, but is indeed really hard to discover and appears quite often, especially where the logic is complex and there are many `return` points. Most of the modern languages have proposed some solutions to avoid this problems.

  For example, some languages provides `defer` statement to ensure the `defer` block will be executed when execution leaves the current scope. This is a good practice for releasing resources like files, devices, or locks.

```
Resource.acquire()
defer { Resource.release() }

if foo { some(); return 0 }
other()
return 1
```

- Kernel panic with “Unable to handle kernel NULL pointer at va 00000140”

  It’s because we should not access the `mm` field of a task struct directly. The kernel provides a function `find_lock_task_mm` to return the real `mm_struct`.

```
/*
 * The process p may have detached its own ->mm while exiting or through
 * use_mm(), but one or more of its subthreads may still have a valid
 * pointer.  Return p, or any of its subthreads with a valid ->mm, with
 * task_lock() held.
 */
struct task_struct *find_lock_task_mm(struct task_struct *p)
{
    struct task_struct *t = p;

    do {
        task_lock(t);
        if (likely(t->mm))
            return t;
        task_unlock(t);
    } while_each_thread(p, t);

    return NULL;
}
```

- Kernel panic with “PC=0x00000000”

  It’s because I made the mistake of freeing `timer` directly without calling `del_timer` when removing a limitation. When the “freed” timer elapses, kernel may find that memory of the pointer to the handler function is freed and set to zero, so it calls 0x00000000 and results in a kernel panic.

```
if (mm_max == ULONG_MAX) {
    /* remove */
    list_del(&p->list);
    del_timer(p->timer); /* IMPORTANT */
    kfree(p->timer);
    kfree(p);
    printk(KERN_INFO "*** Removed: uid=%u ***\n", p->uid);
}
```

  Also, make sure `init_timer` is called with each timer when it is initialized, otherwise the function `del_timer` will cause a deadlock.

6.2 Acknowledgements

I’d like to thank Prof. Wu and TAs for the instructions on theoretical lectures and advice on the project.

I’d also like to thank my classmates Yimin Zhao and Chi Zhang for their help on kernel deadlocks and inspiration on killer’s trigger.

Finally, I’d like to thank my mother for providing me with a good learning environment.

7 References

[1] R. Love, *Linux Kernel Development*, ser. Developer's Library. Pearson Education, 2010. [Online]. Available: https://books.google.com/books?id=3MWRMYRwulIC

[2] Kracken, "Read/write lock for linux kernel module." [Online]. Available: https://stackoverflow.com/questions/38880604/read-write-lock-for-linux-kernel-module

[3] zer0stimulus, sarnold, and Angel, "kernel: how to add a new source file for kernel build?" [Online]. Available: https://stackoverflow.com/questions/8549992/kernel-how-to-add-a-new-source-file-for-kernel-build

[4] Linux, "Chapter 13 out of memory management." [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/understand016.html

[5] Google, "drivers/staging/android/lowmemorykiller.c - git at google." [Online]. Available: https://android.googlesource.com/kernel/msm/+/afd912bfd26fe96b271d9d52a058d486e1ef6b48/drivers/staging/android/lowmemorykiller.c

[6] M. L. Mitchell, "Linux system calls." [Online]. Available: https://www.informit.com/articles/article.aspx?p=23618

[7] ugoren, "Why write-lock-irq is used rather then write-lock for the tasklist-lock?" [Online]. Available: https://stackoverflow.com/questions/14719439/why-write-lock-irq-is-used-rather-then-write-lock-for-the-tasklist-lock