Design and Implementing my new OOM Killer

Summary

The Linux memory management code does its best to ensure that memory will always be available when some part of the system needs it. But in order to improve the efficiency of memory usage, the Linux kernel adopts the method of over-commit memory, which causes excessive strain on physical memory. So when a system has reached a point where no memory is available, things can grind to a painful halt, with the only possible solution (other than rebooting the system) being to kill off processes until a sufficient amount of memory is freed up. That grim task falls to the out-of-memory (OOM) killer.

In order to improve the OOM Killer's kill process mechanism (that is to say, to make the OOM Killer smarter), we first learn the basic ideas of how a OOM Killer is triggered and how does it select a specific process and kill it. Then we design and implement a new OOM Killer which use different killing strategy to make up the drawbacks of the original one when the system encountered malicious applications.

Firstly, using the knowledge we've learned in project 1 about adding a kernel module, we add two new system calls (syscall 382 and syscall 383). Here we define a new data structure (struct MMLimits) in sched.h and define two global variables (my_mm_limits and my_current_RSS) , one (my_mm_limits) store the information about MM_max limits we set for each user while the other (my_current_RSS) store the current resident set size for each user. We initialize these global variables in init_task.c , applying **EXPORT_SYMBOL** function on them and use **extern** declaration each time we use them. We use syscall 382 to set MM_max limits for each user , and syscall 383 to check the triggered condition of our new OOM Killer.

Secondly, we investigate these four functions: __alloc_pages_nodemask() | __alloc_pages_slowpath() | __alloc_pages_may_oom() | out_of_memory() to learn how does the original OOM Killer is triggered. Then I add the trigger code in page_alloc.c and main working code in oom_kill.c. The trigger code cooperate with the set MM_max limit system call to check if there exists any user that run out of its memory limit. When triggered, new OOM Killer select the highest RSS process of that user and kill it. Note that we need to output some information every time we kill a process, we modify the function interface of both trigger code and working code. When the program call the trigger function, it pass the necessary information to the killing function, which greatly reduce the time for re-finding the necessary information in killing process.

In the end, for bonus points, we design a new reasonable rule to choose a process which should be killed. This modification makes the OOM Killer smarter and lower the case for killing important processes that user consider.

Design and Implementing my new OOM Killer

June 13,2020

Contents

1 Introduction

1.1 Problem Background

A system may become out-of-memory (OOM) when a buggy or malicious process uses a lot of physical memory or when the system has too many processes running. Although Linux implements an OOM killer to selectively kill some processes to avoid crashing the whole system, it has one security flaw that it ignores which users created the processes when selecting the process to kill. A malicious user can thus game the OOM killer by creating a large number of processes. While the aggregated amount of memory allocated to this user's processes is huge, each process owns only a small amount of memory, and the OOM killer may not notice these processes and kill the other users' processes.

1.2 My Work

My work is consisted of four main parts. First part, add a new System Call of setting MM_max limit for each user and add a new System Call of checking the trigger condition. Second part, understand and explain how the original OOM Killer is triggered. Third part, design and Implement a new OOM Killer. Fourth part, design a new rule to select the 'best' process for OOM Killer and modify the select_badness_function to implement it.

2 Implemented Details

2.1 Global variables

First let's talk something about global variables. I define two global variables in init_task.c . Why choose this file? Actually, there are many files we can put our global variables, but I think it's more clear to define them in a file with less code and this file should be compiled and executed before all the files we defined other work functions. Because all the other files defined by myself will use these two global variables. So I choose this file. In Figure 1 , we can see that after definition, we call **EXPORT_SYMBOL** function in order to enable these two variables to be used in other kernel modules or source files.

```

/*
 * Initial MMLimit here
 * Try whether it successfully init another global variable here
 */
struct MMLimits my_mm_limits;               // store the limit entry for each user
struct MMLimits my_current_RSS;         // store the cunrrent RSS for each user, it's needed in select_bad_process
/*
my_mm_limits.Current_Size=0;
my_mm_limits.Max_Size=500;
my_mm_limits.MMLimit_list= (struct MMLimit_entry*) kmalloc(sizeof(struct MMLimit_entry)* 500,GFP_KERNEL);
*/
EXPORT_SYMBOL(my_mm_limits);
EXPORT_SYMBOL(my_current_RSS);

// define two flag to indicate whether the module is installed, if not, we can't use the new OOM_killer
int MM_Limit_Module_Ready;
int Check_RSS_Module_Ready;
EXPORT_SYMBOL(MM_Limit_Module_Ready);
EXPORT_SYMBOL(Check_RSS_Module_Ready);

```

Figure 1: Define global variables

Since we need to save the memory limit information (uid and mm_max) in proper data structures so that the OOM killer can use this information to determine whether a user has run out of its memory quota, we define our data structure MMLimits and MMLimit_entry in sched.h as task_struct's definition. The global variable's type is MMLimits and it has a limit entry table whose type is MMLimit_entry. It has current_size and max_size data members to help maintain the limit entry table. You can see the definitions in Figure 2.

sched.h > MMLimits

```c
};

/*
* Add new struct here
*
*/
struct MMLimit_entry{
    long uid;
    unsigned long mm_max;
};

struct MMLimits{
    struct MMLimit_entry * MMLimit_list;
    int Max_Size;
    int Current_Size;

/*
After thoughtful thinking, these functions should not be defined inline function,
I change their definition in system call .c file
```

Figure 2: Define data structure

Secondly, let's see the initialization of these two global variables. There are several ways to initialize the global variables, but I think it's more appropriate to initialize them when we need to use them. So I choose to put their initialization in module init function. When we install the system call module, it means that we want to use these variables, so we init them with allocating the limit entry table array memory using kmalloc(). When we uninstall the system call module, it means that we don't need to use these variables so that we recycle them using kfree(). Here are several key points when I implementing the system call.

(i) choose the system call number carefully. At first I choose system call 356 and 357 but it's continuously be invoked by kernel. So I change them to 382 and 383.

(ii) using global flag variables (MM_Limit_Module_Ready and Check_RSS_Module_Ready)to indicate the module is ready and can be used by the kernel. We set their value separately to 382 and 383 in the module init function , when we uninstall the module we set them to 0 which means the module can't be used by the kernel. Using these two variables, we can let the new OOM Killer be triggered only when the necessary modules are ready. Otherwise, without the necessary components, the kernel will call the original OOM Killer, which absolutely not hurt the correctness.

2.2 System Call

I add two new system calls: MM_limit_system_call (system call number: 382) and Check_Condition_system_call (system call number: 383). We use syscall 382 to set MM_max limits for each user , and syscall 383 to check the triggered condition of our new OOM Killer.

(i)The main idea of MM_limit_system_call.

The idea is to add MM_max limit entry for users when we can't find one, or update the user's limit entry if there already exist one. When things develops normally, the system call return 0, otherwise it return -1 and print the error information. The main function is set_mm_limit(), with four worker function: find_uid(), add_entry(), update_entry(), print_infor(). As you can see in Figure 3, set_mm_limit first call the find_uid() function to find whether there exists a limit entry for the uid, if not it returns -1, call the add_entry() to add a new limit entry; if yes, it returns the index of the entry and call update_entry() to update. Each time when we add a new limit entry, we output the information about all the existing entries, this job is done by print_info() function. If all the things go well, set_mm_limit() return 0, otherwise it return -1.

```
int set_mm_limit(long my_uid, unsigned long mm_max){
    int index;
    int i;
    printk("this is my system set_mm_limit call!\n");
    index=find_uid(&my_mm_limits,my_uid);
    if(index==-1){
        index=add_entry(&my_mm_limits,mm_max,my_uid);   // if add successfully, return 0, else , return -1
        if(index==-1){
            printk("the MMLimit_list is full, add entry failed!\n");
        }
        else{
            print_info(&my_mm_limits);       //traverses and outputs all existing memory limit entries
        }
        return index;       // if set limit successfully, return 0, else , return -1
    }
    i=update_entry(&my_mm_limits,index,mm_max);
    if(i==-1){
        printk("index error, update entry failed!\n");
    }
    else{
        printk("update the %d th entry\n",index);
    }
    return i;
}
```

Figure 3: the structure of MM_limit_system_call

(ii)The main idea of Check_Condition_system_call.

In Check_Condition_system_call, we need to traverse through all the process and collect the RSS value of the same user. Then we compare the current RSS value of each user with the user's MM_max limit in limit entry table. If there exists a user whose current RSS exceeds its limit value, the system call returns the uid of the user, otherwise it returns -1 to indicate that no user exceed the limit. There are two worker functions called by main work function check_OOM_condition() : TAC() and CUL(). TAC() function's job is : Traverse And Collect while CUL() function's job is Check User Limits. You can see their relationship in Figure 4.

```
// check the condition and decide whether to trigger new OOM killer, if satisfies, return uid;else return -1
long check_OOML_condition(int* limit_entry_index, int* RSS_entry_index ){
    int index;
    long uid_result;
    my_current_RSS.Current_Size=0;      // each time we start to check, we need to use the new my_current_RSS,
    index=TAC();
    if(index!=0){
        printk("Traverse And Collect my_current_RSS entries failed!\n");
    }
    uid_result=CUL(limit_entry_index,RSS_entry_index);
    return uid_result;
}
```

Figure 4: the structure of Check_Condition_system_call

For TAC() function, it's much like the set_mm_limit in MM_limit_system_call. We use do_each_thread() { contents……} while_each_thread() to visit all the process one by one and use get_mm_rss() function to get RSS from current process. Note that one page in Android is 4096 bytes, so after we get the RSS we need to multiply 4096 to get the correct memory usage in bytes. For CUL() function, we just check for every current user's total RSS in my_current_RSS's entry table that whether there exist a limit entry in my_mm_limits's limit entry table. If we find one, compare their value and see whether the user's total RSS has exceeded the memory limit. If yes, return uid; else it return -1 to its caller: check_OOML_condition(). Here is an explanation for check_OOML_condition() and CUL()'s function interface. Because we need to output the user RSS value and user MM_max limit value in OOM Killer's killing function, so we use two variables limit_entry_index and RSS_entry_index to indicate the position of the user entry. In later use, we can use these index to directly get information from entry table without go through the whole table again. see how they are used in Figure 5.

```c
// Check User Limits
long CUL(int* limit_entry_index, int* RSS_entry_index){
    int index;
    int i;
    long tmp_uid;
    long process_RSS;
    struct MMLimit_entry* tmp=my_current_RSS.MMLimit_list;
    struct MMLimit_entry* Limit_Table=my_mm_limits.MMLimit_list;
    for(i=0;i<my_current_RSS.Current_Size;++i){
        tmp_uid=tmp[i].uid;
        index=find_uid(&my_mm_limits,tmp_uid);
        if(index==-1) continue;              // it means no limit for this user
        else{
            process_RSS=tmp[i].mm_max;
            if(process_RSS>Limit_Table[index].mm_max){
                *limit_entry_index=index;
                *RSS_entry_index=i;
                return tmp_uid;        // it means user's current RSS is greater than its mm _limits
            }
            else continue;
        }
    }
    return -1;          // if no user exceed, return -1;
}
```

Figure 5: Function interface of CUL() function.

2.3 OOM Killer

(i) How the original OOM killer is triggered

The original OOM Killer is triggered when the system has run out of memory. It's trigger process starts from **__alloc_pages_nodemask**(). This function is trying to allocate physical memory for each request sender. At first it try to find free pages big enough for the requested size. If it can't find any free pages big enough, it will call the **__alloc_pages_slowpath**() function. The main role for __alloc_pages_slowpath() is to trigger memory recycle, memory regular reformation mechanism to try to make room for allocating. If there is nothing that __alloc_pages_slowpath() can help, it will call **__alloc_pages_may_oom**(). __alloc_pages_may_oom() is the OOM Killer enter interface, from here we formally start to get into OOM Killer part. This function make a judgment whether to call **out_of_memory**(), which is the main working function for OOM Killer. The judgment is mainly decided by the following two conditions:

A. The allocate request is not higher order allocation. Because OOM Killer will not help higher order allocs.

B. The OOM Killer will not kill tasks for low memory.

If the system exhausted what can be done then it has to call out_of_memory() function. This function is the kernel part of OOM Killer. It can be divided into two parts: 1. select the 'best' process to kill 2. Kill the specific process and recycle the memory. For the first part, it call select_bad_process() function to choose which process to be killed. **select_bad_process**() function traverse through all the process and calculate each process's points by **oom_badness**() function. It returns the process with the highest points as the 'best' process. In oom_badness() function, the badness score is mainly calculated by adding each task's RSS, page table and swap space size and multiply them with a factor. Then the score is added with oom_score_adj to make some adjustments. More important the process is and longer time the process has been running, lower the oom_score_adj. In the end, the select_bad_process() return the process with the highest score.

Then the function out_of_memory() get into next part: call **oom_kill_process**() to kill the selected process. In oom_kill_process(), it first try to kill the selected process. Then it traverse through the children list of this process to find if any child has a different mm and is eligible for kill. If we have found, the one with the highest oom_badness() score is sacrificed for its parent. In the end it try to kill all the processes which satisfies these three condition:

1. user processes(note that it's not kernel thread)
2. sharing victim->mm
3.in other thread groups

For the fact that these processes don't get access to memory reserves, though, to avoid depletion of all memory. The whole process above is how the original OOM Killer is triggered to kill some specific processes. On the basics of these knowledge, I start to design and implement my own OOM Killer.

(ii) How to design and implement my new OOM killer

The key idea of my design is to '**help**', not to '**replace**'. Because before the necessary module is loaded, the kernel is lack of necessary information to check the condition that whether there exists a process has exceeded its memory limit, we haven't set it yet! So after the system is informed that the modules are ready, it start to use the new OOM Killer's check trigger condition function. You can see Figure 6 to get a clear understanding.

```
/*
*Check new OOM killer condition here,
*These two flag variables' declaration is on line 2435, and it's defined on init_task.c
*check_OOML_condition() is defined on line 2548
*/
//printk("THis is alloc_page, the MM_Limit_Module_Ready is %d, and the Check_RSS_Module_Ready is %d \n",MM_Limit_Module_Ready,Check_RSS_
if(Check_RSS_Module_Ready==383){
    // when two module are both installed, we can use new OOM killer here, otherwise, we just use the original OOM Killer
    long tmp_uid;
    int R_index;
    int L_index;
    tmp_uid=check_OOML_condition(&L_index,&R_index);
    if(tmp_uid==-1) goto MY_RETURN;             // not satisfy trigger condition
    else{
        // trigger new OOM killer
        printk("Start new my_out_of_memory!\n");
        my_out_of_memory(zonelist, gfp_mask, order, nodemask, false,tmp_uid,L_index,R_index);
    }
}
```

Figure 6: Use new OOM Killer condition in __alloc_pages_nodemask()

What's more, there is another important reason that we shouldn't replace the original OOM Killer. Because if we set the MM_max limit for each user large enough, the user may never reach this memory limit. So the new OOM Killer will never be triggered, but the system has run out of memory now. Thus, the original OOM Killer is triggered to deal with this case.

My design idea is that when start to use new OOM Killer, we call the Check_Condition_system_call functions to check the trigger condition. Because we are now in kernel code, so we don't directly call a kernel module (syscall) to do the job. So I copy all the worker functions from Check_Condition_system_call to page_alloc.c As we've already explain the role of these worker function ,we don't repeat here. When **check_OOML_condition**() has found the exceed limit user and return its uid, we start to call **my_out_of_memory**() function, which is the key part of the new OOM Killer. As you can see in Figure 6, my_out_of_memory() has two more parameters than out_of_memory(). These two parameters are limit_entry_index and RSS_entry_index to indicate the position of the user entry as we've talked above.

I add all the new OOM Killer work code in oom_kill.c as the original one. I add four new functions: my_out_of_memory(), my_select_bad_process(), my_oom_kill_process(),my_oom_badness(). The main difference of these function's goal with the original ones is: the new OOM Killer is aimed to kill the process of the exceed limit user with the highest RSS. So the process must be chosen among the specific user processes and the only criteria to select is the process RSS. So that's the main idea in **my_select_bad_process**() and **my_oom_badness**(). In **my_select_bad_process**(), as you can see in Figure 7, add a judgment statement : **if(p->cred->uid!=chosen_uid)** when selecting the killing process.

```

// define new OOM Killer 's select bad process here
static struct task_struct * my_select_bad_process(unsigned int *ppoints,
        unsigned long totalpages, struct mem_cgroup *memcg,
        const nodemask_t *nodemask, bool force_kill,long chosen_uid)
{
    struct task_struct *g, *p;
    struct task_struct *chosen = NULL;
    *ppoints = 0;

    do_each_thread(g, p) {
        unsigned int points;
        if(p->cred->uid!=chosen_uid)
            continue;
        if (p->exit_state)
            continue;
        if (oom_unkillable_task(p, memcg, nodemask))
            continue;
        /*
```

Figure 7: select process from exceed limit user in my_select_bad_process()

In Figure 8 you can see that in my_oom_badness() function, we calculate the points of a process only use the RSS value.

```
     * The memory controller may have a limit of 0 bytes, so avoid a divide
     * by zero, if necessary.
     */
    if (!totalpages)
        totalpages = 1;

    /*
     * The baseline for the badness score only has something to do with process's RSS, we
     * don't consider other factors here.
     */
    points = get_mm_rss(p->mm);
    points *= 1000;
    points /= totalpages;
    task_unlock(p);

    if (points <= 0)
        return 1;

    return (points < 1000) ? points : 1000;
}
```

Figure 8 : calculate process points in my_oom_badness()

Finally, in my_oom_kill_process() , we output the information each time we kill a process. Here we use the two index variable (**RSS_entry_index** and **limit_entry_index**) we get from CUL() function which is passed through the whole OOM Killer process to get the mm_max and uRSS information from corresponding information table.

```c

    task_lock(p);
    pr_err("%s: Kill process %d (%s) score %d or sacrifice child\n",
        message, task_pid_nr(p), p->comm, points);

    // print information when kill the process
    process_RSS=get_mm_rss(p->mm);      //get_mm_rss() defined in mm.h
    printk("uid=%d, uRSS=%ld, mm_max=%ld, pid=%d, pRSS=%ld",p->cred->uid,
    R_tmp[RSS_entry_index].mm_max,L_tmp[limit_entry_index].mm_max, task_pid_nr(p),process_RSS);

    task_unlock(p);
```

Figure 9: Output information in my_oom_kill_process()

3 Bonus part

4 Results

This part I will show you the result of this project, which including the following three parts: system call results, basic OOM Killer results, bonus results.

4.1 System call results:

(1)

```
root@generic:/data/misc # chmod 777 mmcallerARM
root@generic:/data/misc # chmod 777 prj2testARM
root@generic:/data/misc # chmod 777 CCcallerARM
root@generic:/data/misc # ./mmcallerARM
Here is the mm_caller, start testing the system call!
root@generic:/data/misc # ./CCcallerARM
This is my Check_Condition_caller
There is no user exceed MM_Limit
root@generic:/data/misc # ./CCcallerARM
This is my Check_Condition_caller
There is no user exceed MM_Limit
root@generic:/data/misc #
```

Figure 10: Execute the system call caller: mmCallerARM, CCcallerARM

```
module load!succeed
my_mm_limits's limit table is NULL, initialized in module init
module load!succeed
my_current_RSS's limit table is NULL, initialized in module init
this is my system set_mm_limit call!
uid=10070, mm_max=9870
this is my system set_mm_limit call!
uid=10070, mm_max=9870
uid=10071, mm_max=9871
this is my system set_mm_limit call!
uid=10070, mm_max=9870
uid=10071, mm_max=9871
uid=10072, mm_max=9872
this is my system set_mm_limit call!
uid=10070, mm_max=9870
uid=10071, mm_max=9871
uid=10072, mm_max=9872
uid=10073, mm_max=9873
this is my system set_mm_limit call!
uid=10070, mm_max=9870
uid=10071, mm_max=9871
uid=10072, mm_max=9872
uid=10073, mm_max=9873
uid=10074, mm_max=9874
this is my system set_mm_limit call!
uid=10070, mm_max=9870
uid=10071, mm_max=9871
uid=10072, mm_max=9872
uid=10073, mm_max=9873
uid=10074, mm_max=9874
uid=10075, mm_max=9875
```

Figure 11: Part of result of testing set_mm_limit system call (test add)

(2)

```
uid=10074, mm_max=9874
uid=10075, mm_max=9875
uid=10076, mm_max=9876
uid=10077, mm_max=9877
uid=10078, mm_max=9878
uid=10079, mm_max=9879
uid=10080, mm_max=9880
uid=10081, mm_max=9881
uid=10082, mm_max=9882
uid=10083, mm_max=9883
uid=10084, mm_max=9884
this is my system set_mm_limit call!
update the 0 th entry
this is my system set_mm_limit call!
update the 9 th entry
this is my system set_mm_limit call!
uid=10070, mm_max=1000834
uid=10071, mm_max=9871
uid=10072, mm_max=9872
uid=10073, mm_max=9873
uid=10074, mm_max=9874
uid=10075, mm_max=9875
uid=10076, mm_max=9876
uid=10077, mm_max=9877
uid=10078, mm_max=9878
uid=10079, mm_max=1000
uid=10080, mm_max=9880
uid=10081, mm_max=9881
uid=10082, mm_max=9882
uid=10083, mm_max=9883
uid=10084, mm_max=9884
uid=10099, mm_max=1000835
```

Figure 12: Part of result testing set_mm_limit system call (test update)

```
root@generic:/data/misc # ./CCcallerARM
This is my Check_Condition_caller
There is no user exceed MM_Limit
root@generic:/data/misc # ./CCcallerARM
This is my Check_Condition_caller
There is no user exceed MM_Limit
root@generic:/data/misc # ./CCcallerARM
This is my Check_Condition_caller
There is no user exceed MM_Limit
```

Figure 13: Result1 of testing check trigger condition system call

```
uid=0, Current_RSS=819200
update the 0 th entry, it's adding process_RSS is 667648
uid=0, Current_RSS=1486848
uid=1036, Current_RSS=1806336
uid=0, Current_RSS=1486848
uid=1036, Current_RSS=1806336
uid=1000, Current_RSS=1806336
update the 1 th entry, it's adding process_RSS is 1806336
update the 1 th entry, it's adding process_RSS is 1806336
update the 1 th entry, it's adding process_RSS is 1806336
update the 1 th entry, it's adding process_RSS is 1806336
update the 1 th entry, it's adding process_RSS is 1806336
update the 0 th entry, it's adding process_RSS is 2441216
update the 0 th entry, it's adding process_RSS is 2441216
update the 0 th entry, it's adding process_RSS is 2441216
update the 0 th entry, it's adding process_RSS is 2441216
update the 0 th entry, it's adding process_RSS is 335872
update the 0 th entry, it's adding process_RSS is 1155072
update the 2 th entry, it's adding process_RSS is 1097728
update the 2 th entry, it's adding process_RSS is 30793728
update the 2 th entry, it's adding process_RSS is 30793728
update the 2 th entry, it's adding process_RSS is 30793728
update the 2 th entry, it's adding process_RSS is 30793728
update the 2 th entry, it's adding process_RSS is 30793728
update the 2 th entry, it's adding process_RSS is 30793728
update the 2 th entry, it's adding process_RSS is 30793728
update the 2 th entry, it's adding process_RSS is 30793728
update the 2 th entry, it's adding process_RSS is 30793728
update the 2 th entry, it's adding process_RSS is 30793728
update the 2 th entry, it's adding process_RSS is 30793728
update the 2 th entry, it's adding process_RSS is 30793728
update the 0 th entry, it's adding process_RSS is 978944
uid=0, Current_RSS=13721600
uid=1036, Current_RSS=10838016
uid=1000, Current_RSS=372428800
uid=2000, Current_RSS=1171456
update the 0 th entry, it's adding process_RSS is 536576
update the 0 th entry, it's adding process_RSS is 536576
update the 0 th entry, it's adding process_RSS is 536576
update the 0 th entry, it's adding process_RSS is 536576
```

Figure 14: Part of result of testing check trigger condition system call

4.2 New OOM Killer results:

(1)

```
root@generic:/data/misc # su 10069
u0_a69@generic:/data/misc $ ./prj2testARM u0_a69 100000000 60000
pw->uid=10069, pw->name=u0_a69
@@@@uid: 10069
@@@@pid: 1151
child process start malloc: pid=1152, uid=10069, mem=60000
child process finish malloc: pid=1152, uid=10069, mem=60000
u0_a69@generic:/data/misc $
```

Figure 15: Test the OOM Killer not kill the normal process (not exceed MM_max limit)

```
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
this is my system set_mm_limit call!
uid=10070, mm_max=1000834
uid=10071, mm_max=9871
uid=10072, mm_max=9872
uid=10073, mm_max=9873
uid=10074, mm_max=9874
uid=10075, mm_max=9875
uid=10076, mm_max=9876
uid=10077, mm_max=9877
uid=10078, mm_max=9878
uid=10079, mm_max=1000
uid=10080, mm_max=9880
uid=10081, mm_max=9881
uid=10082, mm_max=9882
uid=10083, mm_max=9883
uid=10084, mm_max=9884
uid=10099, mm_max=1000835
uid=10069, mm_max=100000000
```

Figure 16: Result of testing the OOM Killer not kill the normal process

(2)

```
u0_a68@generic:/data/misc $ ./prj2testARM u0_a68 100000000 160000000
pw->uid=10068, pw->name=u0_a68
@@@@uid: 10068
@@@@pid: 1159
child process start malloc: pid=1160, uid=10068, mem=160000000
u0_a68@generic:/data/misc $
```

Figure 17: Test the OOM Killer kill the exceed MM_max limit process

```
[   76]  2000    76      755      287  0     -17        -1000 sh
[   77]     0    77     1445      131  0     -17        -1000 adbd
[   78]     0    78     3527      475  0     -17        -1000 netd
[   79]     0    79      984      410  0     -17        -1000 debuggerd
[   80]  1001    80     1950      398  0     -17        -1000 rild
[   81]  1019    81     2866      912  0     -17        -1000 drmserver
[   82]  1013    82    30554     2081  0     -17        -1000 mediaserver
[   83]     0    83      684      288  0     -17        -1000 installd
[   87]  1017    87     1421      496  0     -17        -1000 keystore
[   88]     0    88   118959    12190  0     -17        -1000 main
[   89]  1000    89     1310      450  0     -17        -1000 gatekeeperd
[   92]     0    92      688      297  0     -17        -1000 perfprofd
[   93]  1000    93     1564      461  0     -17        -1000 fingerprintd
[  186]     0   186      755      288  0       0            0 sh
[  239]  1000   239   153713    23043  0     -14         -941 system_server
[  572]  1023   572     1560      469  0     -17        -1000 sdcard
[  673] 10013   673   143520    25937  0     -10         -705 ndroid.systemui
[  696] 10032   696   125307     9193  0       1          117 putmethod.latin
[  711]  1001   711   128558    10851  0     -10         -705 m.android.phone
[  719] 10007   719   134111    18954  0       0            0 droid.launcher3
[  747] 10006   747   122278     6422  0      11          764 externalstorage
[  762] 10002   762   124409     8574  0       9          647 d.process.acore
[  826] 10035   826   122498     7052  0      11          764 m.android.music
[  856] 10005   856   123703     8467  0       7          529 d.process.media
[  924] 10023   924   123667     7729  0       9          647 droid.deskclock
[  948] 10042   948   122512     6541  0       9          647 .quicksearchbox
[  967] 10004   967   123790     6788  0      11          764 .android.dialer
[  981]  1000   981   122027     6239  0      11          764 ndroid.keychain
[  996] 10001   996   123618     7602  0       7          529 viders.calendar
[ 1014] 10040  1014   122616     6598  0       9          647 id.printspooler
[ 1037] 10008  1037   122195     6330  0      11          764 gedprovisioning
[ 1055] 10019  1055   125277     7446  0       7          529 ndroid.calendar
[ 1079] 10027  1079   126297     8537  0       7          529 m.android.email
[ 1096] 10029  1096   124089     6946  0       7          529 ndroid.exchange
[ 1116]  1000  1116   125483     6616  0       9          647 ndroid.settings
[ 1156] 10068  1156      755      286  0       0            0 sh
[ 1159] 10068  1159      596      229  0       0            0 prj2testARM
[ 1160] 10068  1160    41556    14550  0       0            0 prj2testARM
Out of memory: Kill process 1160 (prj2testARM) score 57 or sacrifice child
uid=10068, uRSS=123412480, mm_max=100000000, pid=1160, pRSS=14550
Killed process 1160 (prj2testARM) total-vm:166224kB, anon-rss:58052kB, file-rss:148kB
Kille victim, uid=10068, uRSS=123412480, mm_max=100000000, pid=1160, pRSS=14550
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
```

Figure 18: Result of testing the OOM Killer kill the exceed MM_max limit process

(3)

```
139|root@generic:/data/misc # su 10088
u0_a88@generic:/data/misc $ ./prj2testARM u0_a88 1500000000 1200000
pw->uid=10088, pw->name=u0_a88
@@@@uid: 10088
@@@@pid: 1169
child process start malloc: pid=1170, uid=10088, mem=1200000
child process finish malloc: pid=1170, uid=10088, mem=1200000
u0_a88@generic:/data/misc $
```

```
u0_a88@generic:/data/misc $ ./prj2testARM u0_a88 150000000 190000000
pw->uid=10088, pw->name=u0_a88
@@@@uid: 10088
@@@@pid: 1171
child process start malloc: pid=1172, uid=10088, mem=190000000
u0_a88@generic:/data/misc $
```

Figure 19: Test the OOM Killer for at first normal then exceed limit process

```
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
this is my system set_mm_limit call!
uid=10070, mm_max=100000000
uid=10071, mm_max=9871
uid=10072, mm_max=9872
uid=10073, mm_max=9873
uid=10074, mm_max=9874
uid=10075, mm_max=9875
uid=10076, mm_max=9876
uid=10077, mm_max=9877
uid=10078, mm_max=9878
uid=10079, mm_max=1000
uid=10080, mm_max=9880
uid=10081, mm_max=9881
uid=10082, mm_max=9882
uid=10083, mm_max=9883
uid=10084, mm_max=9884
uid=10099, mm_max=1000835
uid=10069, mm_max=100000000
uid=10068, mm_max=100000000
uid=10088, mm_max=1500000000
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
```

Figure 20: Result for normal: Test for at first normal then exceed limit process

```
[ 1037] 10008  1037   122195     6330   0      11          764 gedprovisioning
[ 1055] 10019  1055   125277     7446   0       7          529 ndroid.calendar
[ 1079] 10027  1079   126297     8537   0       7          529 m.android.email
[ 1096] 10029  1096   124089     6946   0       7          529 ndroid.exchange
[ 1116]  1000  1116   125483     6616   0       9          647 ndroid.settings
[ 1168] 10088  1168      755      286   0       0            0 sh
[ 1171] 10088  1171      596      229   0       0            0 prj2testARM
[ 1172] 10088  1172    49748    25461   0       0            0 prj2testARM
Out of memory: Kill process 1172 (prj2testARM) score 100 or sacrifice child
uid=10088, uRSS=212795392, mm_max=150000000, pid=1172, pRSS=25461
Killed process 1172 (prj2testARM) total-vm:198992kB, anon-rss:101696kB, file-rss:148kB
Kille victim, uid=10088, uRSS=212795392, mm_max=150000000, pid=1172, pRSS=25461
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
```

Figure 21: Result for exceeding: Test for at first normal then exceed limit process

5 Advantages and Weaknesses

5.1 Advantages

- The idea is clear and can be quickly understood by the others
- The code has high readability with clear naming rules and detailed comments.
- The program logic structure is clear due to a good abstraction of main code's jobs into several worker function.

- Robustness. Our strategy is to help not to replace the original OOM Killer, and two global flag variables are added in order to ensure each time we use the new OOM Killer we have gained all the necessary information. So the system will not crash and its robustness is enhanced due to the help of new OOM Killer's trigger condition.
- Easy to test. I implement two system_call caller (in mm_caller.c and Check_Condition_caller.c) to help test the system calls' correctness. You can just compile them and execute them in android kernel. (Note that they don't need any parameters)

5.2 Weaknesses

- Do not use daemon to better improve the trigger strategy
- The memory limit is too strict, this part should be improved.

6 Discussions

From this project I leaned a lot about how to write the kernel code and enhanced the understanding about kernel module. But there are still a lot of knowledge for me to discover and a lot of brilliant idea of designing the kernel for me to learn. This new OOM Killer can be improved by adding **daemon process** to periodically trigger the OOM Killer and using **time_allow_exceed** strategy to allow the users temporarily exceed the memory limit, which is a proper memory limit for real world users. I think I will learn these technology in the future.