# OOM-killer Per User - Report of Project2

Zhicun Chen 518030910173 Zhicun_Chen@sjtu.edu.cn

## 1. Introduction

In this report, I will first introduce you that how to use my implementation of this OOM killer. Then I will show you the details of my implementation by four steps. Actually, the new OOM-killer is just the final output, to accomplish this project, I also need to learn how to compile the Android kernel, get familiar with how information can be shared in the kernel and how Android original OOM-killer works.

## 2. How to Use

### Running Environment

I test all the programs on the Ubuntu 18.04 LTS. I think it will work well on all Linux-based OS, however, I have not tested them on other systems, so you are highly recommended to change to Ubuntu 18.04 LTS if you  encountered any problem during testing on other systems. For the tool chains, you need to install JDK and NDK properly.

### File Structure

```
.
├── kernel_change                 // list all kernel files that I modified
│   ├── init_task.c
│   ├── kernel.change
│   ├── mm.h
│   ├── page_alloc.c
│   └── signal.c
├── oom_killer_daemon              // implement new oom-killer by daemon
│   └── jni
│       ├── Android.mk
│       └── oom_killer_daemon.c
├── oom_killer_module             // implement new oom-killer by hacking the kernel
│   ├── Makefile
│   └── my_oom_killer.c
├── set_mm_limits                 // add the syscall to set my_mm_limits
│   ├── Makefile
│   └── set_mm_limits.c
├── test_oom_killer_daemon          // test the oom-killer implemented by daemon
│   └── jni
│       ├── Android.mk
│       └── daemon_test.c
├── test_oom_killer_kernel          // test the oom-killer implemented by hacking the kernel
│   └── jni
│       ├── Android.mk
│       └── prj2_test.c
└── test_set_mm_limits            // test the syscall set_mm_limits()
```

```
└── jni
    ├── Android.mk
    └── test_set_mm_limits.c
```

Above is the file tree of my project.

> - In the ***kernel_change*** directory, it lists all kernel files that I modified to finish this project, please replace files listed in ***kernel.change*** with files in this directory. After you do the replacement, please run **make -j4** for faster recompiling Android kernel.
> - For the remain directory, if it contains a ***jni*** directory, you need to run **ndk-build** in the terminal, and push generated files into the virtual device.
> - For the directory without ***jni*** directory, you need to run **make** to compile a new kernel module and push them into the virtual device.

For detailed purpose of each program file, you can check the head comments to se what exactly that program wants to do. For implementation details, you are welcome to read my source codes with necessary comments that I gave.

## Some Pre-defined Variables

There are some pre-defined variables that affect how program works. First, **MY_MM_LENGTH** is defined as 10 in *line 37* of ***mm.h*** ,*line 2448* of ***page_alloc.c*** and *line 22* of ***set_mm_limits.c*** , if you want to change the number of slots in the **my_mm_limits** , you can change the definition of all **MY_MM_LENGTH**. Then there are three pre-defined in line 2803-2806, it decides which type of oom-killer will be used, if want to change, just comment the other define statements. By the way, if you would like to add some new omm-killer, you can create a new function and add another define statement, which is highly welcomed. The last pre-defined variables is **OOM_KILLER_TYPE** in the line 25 of ***my_oom_killer.c***, and there are comments to show how this variable works.

## How to Test

- To test new oom_killer implemented in Linux kernel, you need first choose oom_killer type by changing the comments of:
  - **OMM_KILLER_HIGHEST_RSS** in *line 2803* of ***page_alloc.c***
  - **OOM_KILLER_LONGEST_RUN_TIME** in *line 2804* of ***page_alloc.c***
  - **OOM_KILLER_WORST** in *line 2805* of ***page_alloc.c***

  Then you need to re-compile the kernel. After you booting with the new kernel, you can find ***set_mm_limits.ko*** and install this module with command **insmod**. After installing, you can check the module by running **test_set_mm_limitsARM**. And then you can running **oom_test_kernelARM** to test oom-killer you chose. One proper example is:

  **./oom_test_kernelARM u0_a70 100000000 20000 40000 20000 30000 200000 100000000**

  Notes: you must first switch to user u0_a70 before you run this command.
- To test new oom-killer implemented as a daemon program and module, you need first comment the define statements of **OMM_KILLER_HIGHEST_RSS**, **OOM_KILLER_LONGEST_RUN_TIME**, **OOM_KILLER_WORST** to close oom-killer in the Linux kernel. And change **OOM_KILLER_TYPE**'s value to which type you want to test in *line25* of ***my_oom_killer.c***. Then:
  - Re-compile Android kernel and ***my_oom_killer.ko***
  - **insmod *set_mm_limits.ko*** and ***my_oom_killer.ko*** after you booting with the new kernel
  - Running ***oom_daemonARM*** to run the new oom-killer with the given time intervals to execute, one proper example is:

        - **./oom_daemonARM 5**
    - Running **oom_test_daemonARM** to test the oom-killer you chose. One proper example is:
        - **./oom_test_daemonARM u0_a70 100000000 20000 40000 20000 30000 200000 100000000**

    Also note that you must switch to u0_a70 before running this command.

# 3. Implementation Details

## Change the Compile Config

This part I just need to install a GUI library and follow the steps in the guidance file **CS356-Project.pdf** .

## Add a Syscall named set_mm_limits()

- First, I define my struct my_mm_limits in ***mm.h*** and initialize it in ***init_task.c*** .
    - Definition in ***mm.h***

```
#define MY_MM_LENGTH 10
// MY_MM_LENGTH is the number of slots that can storage mm limit information
// You can change it into fit number, however, remember to change MY_MM_LENGTH.
struct MMLimits
{
    int uid[MY_MM_LENGTH];
    int mm_max[MY_MM_LENGTH];
};

extern struct MMLimits my_mm_limits;
```

- Second, I implement a syscall **set_mm_limits(int uid, int mm_max)** as a kernel module. The module will update all slots by LRU algorithm. The more detailed information you can check the source codes and comments below.
    - implementation of **set_mm_limits(int uid, int mm_max)** in ***set_mm_limits.c*** .

```
static int set_mm_limits(int uid,int mm_max){
    int i;
    int full=0;
    /*
     * The whole my_mm_limits is updated by LRU algorithm.
     * The smaller index is, the less recently it is used.
     */
    for(i=0;i<MY_MM_LENGTH;i++){
        if(my_mm_limits.uid[i]!=-1){
            if(my_mm_limits.uid[i]!=uid){
                continue;
            }
            else{
                // to update my_mm_limits by LRU algorithm.
                int j;
                for(j=i;(j<(MY_MM_LENGTH-1))&&
(my_mm_limits.uid[j+1]!=-1);j++){
```

```
                    my_mm_limits.uid[j]=my_mm_limits.uid[j+1];
                    my_mm_limits.mm_max[j]=my_mm_limits.mm_max[j+1];
                }
                my_mm_limits.mm_max[j]=mm_max;
                my_mm_limits.uid[j]=uid;
                break;
            }
        }
        else{
            // this slot have not been initialized before.
            my_mm_limits.uid[i]=uid;
            my_mm_limits.mm_max[i]=mm_max;
            break;
        }
    }
    if(i==MY_MM_LENGTH){
        full=1;
        printk(KERN_ERR "The MMLimits is full! We will delete the
earliest limits!\n");
    }
    /*
     * find a victim limits by LRU algorithm,
     * and update limits by LRU algorithm.
     */
    if(full){
        for(i=0;i<MY_MM_LENGTH-1;i++){
            my_mm_limits.uid[i]=my_mm_limits.uid[i+1];
            my_mm_limits.mm_max[i]=my_mm_limits.mm_max[i+1];
        }
        my_mm_limits.uid[9]=uid;
        my_mm_limits.mm_max[9]=mm_max;
    }

    // print all mm limits
    for(i=0;i<MY_MM_LENGTH;i++){
        if(my_mm_limits.uid[i]!=-1){
            printk(KERN_INFO
"uid=%d,mm_max=%d",my_mm_limits.uid[i],my_mm_limits.mm_max[i]);
        }
    }

    printk(KERN_INFO "\n");

    return full;
}
```

## The Trigger mechanism of the Original OOM killer

As mentioned in guidance file, the function chain is like:

```
__alloc_pages()                         //invoked when allocating pages
  |-->__alloc_pages_nodemask()
    |--> __alloc_pages_slowpath()
      |--> __alloc_pages_may_oom()
        |--> out_of_memory()              //trigger the OOM killer
```

Actually, function **__alloc_pages()** just simple calls **__alloc_pages_nodemask()**, which is the "heart" of the zoned buddy allocator. What **__alloc_pages_nodemask()** do is to allocate physical memory in the kernel. It tries many ways to allocate physical memory, and **__alloc_pages_slowpath()** is included. And in**__alloc_pages_slowpath()**, if we have tried all ways to allocate pages, but still don't work, then it is only possible that out_of_memory happens, so we need to call **out_of_memory()** .

For the details of **out_of_memory()**, if current process waiting for allocating pages is killable, it will just kill current process.

```
if (sysctl_oom_kill_allocating_task &&
        !oom_unkillable_task(current, NULL, nodemask) &&
        current->mm) {
        oom_kill_process(current, gfp_mask, order, 0, totalpages, NULL,
                 nodemask,
                 "Out of memory (oom_kill_allocating_task)");
        goto out;
    }
```

Else, it will choose a "worst" process by calling **select_bad_process()**, and kill it.

```
    p = select_bad_process(&points, totalpages, NULL, mpol_mask,
                   force_kill);
    /* Found nothing?!?! Either we hang forever, or we panic. */
    if (!p) {
        dump_header(NULL, gfp_mask, order, NULL, mpol_mask);
        read_unlock(&tasklist_lock);
        panic("Out of memory and no killable processes...\n");
    }
    if (PTR_ERR(p) != -1UL) {
        oom_kill_process(p, gfp_mask, order, points, totalpages, NULL,
                 nodemask, "Out of memory");
        killed = 1;
    }
```

For **select_bad_process()** it will go through all process and compute their "badness" by calling **oom_badness()**. It will choose the one that have the largest badness points. The details of **oom_badness()** is like below:

```
unsigned int oom_badness(struct task_struct *p, struct mem_cgroup *memcg,
              const nodemask_t *nodemask, unsigned long totalpages)
{
    long points;

    if (oom_unkillable_task(p, memcg, nodemask))
        return 0;

    p = find_lock_task_mm(p);
    if (!p)
        return 0;

    if (p->signal->oom_score_adj == OOM_SCORE_ADJ_MIN) {
        task_unlock(p);
        return 0;
    }

    /*
```

```
     * 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 is the proportion of RAM that each
 * task's rss, pagetable and swap space use.
 */
    points = get_mm_rss(p->mm) + p->mm->nr_ptes;
    points += get_mm_counter(p->mm, MM_SWAPENTS);

    points *= 1000;
    points /= totalpages;
    task_unlock(p);

    /*
     * Root processes get 3% bonus, just like the __vm_enough_memory()
     * implementation used by LSMs.
     */
    if (has_capability_noaudit(p, CAP_SYS_ADMIN))
        points -= 30;

    /*
     * /proc/pid/oom_score_adj ranges from -1000 to +1000 such that it may
     * either completely disable oom killing or always prefer a certain
     * task.
     */
    points += p->signal->oom_score_adj;

    /*
     * Never return 0 for an eligible task that may be killed since it's
     * possible that no single user task uses more than 0.1% of memory and
     * no single admin tasks uses more than 3.0%.
     */
    if (points <= 0)
        return 1;
    return (points < 1000) ? points : 1000;
}
```

## The Design and Implementation of My OOM Killer

For the basic oom-killer using highest RSS rule to kill process, you can check the function **void oom_killer_highest_rss(void)** in the *line 2450* of ***page_alloc.c*** . I add extremely detailed comments to show how I implement this oom-killer.

This function will go through all the process that exists in the Android, and try to find a match between one specific process and the limits information in the **my_mm_limits**. If we don't find a match, we just skip this process and do the operation on the next process. If we find a match, we will initialize the record of process that has the largest RSS and the number of this RSS (condition is that this slot have not been set yet) or we will compare the RSS of this process with the record process and update the information. Finally, if we find a match, we will update the physical memory have been allocated to this user.

After we go through all processes, we have information that record process that has maximum of RSS of each slot and this maximum. Then we will compare the physical memory allocated to this user and its **mm_max** information, if exceeding, then we need to do some operations to kill the process that is recorded to have the maximum RSS, and also we need to kill all processes that have the same **mm**.

Finally, we will check after we do the kill operations whether all **mm_max** limits are satisfied, if not, go to the beginning part and do whole procedures one more time, until it fits all **mm_max** limits for each user.

For more detailed implementation details, you can refer to my codes and comments.

```
#define MY_MM_LENGTH 10
// same as described in mm.h
void oom_killer_highest_rss(void){
    struct task_struct *p;
    int mm_alloc[MY_MM_LENGTH];
    struct task_struct* max_rss_process[MY_MM_LENGTH];
    int max_rss[MY_MM_LENGTH];
    int i,found,index;

    read_lock(&tasklist_lock);
my_rss_oom_begin:
    // initialize all useful variables
    for(i=0;i<MY_MM_LENGTH;i++){
        mm_alloc[i]=0;
        max_rss_process[i] = NULL;
        max_rss[i]=0;
    }
    // go through all process
    for_each_process(p) {
        found=0;
        // go through all my_mm_limits slots to find fit slot
        for(i=0;i<MY_MM_LENGTH;i++){
            if (p->cred->uid==my_mm_limits.uid[i]){
                // if this slot have not been set, then set
                if (max_rss_process[i] == NULL){
            // this condition aims to avoid NULL pointers, since some kernel
            // processes do not have mm pointer.
                    if (p->mm){
                        max_rss_process[i]=p;
                        max_rss[i]=get_mm_rss(p->mm) * 4096;
                    }
                }
                else{
                    if (p->mm){
            // if this process has larger rss than previous process, then update
                        if ((get_mm_rss(p->mm) * 4096)>max_rss[i]){
                            max_rss_process[i]=p;
                            max_rss[i]=get_mm_rss(p->mm) * 4096;// FIX
                        }
                    }
                }
                found=1;
                index=i;
                break;
            }
```

```
        }
        if (found){
            if (p->mm){
                // if we find a process fit , update the corresponding mm_alloc
                mm_alloc[index]+=get_mm_rss(p->mm) * 4096;
            }
        }
    }
    // check whether out-of-memory happens on each slot
    for(i=0;i<MY_MM_LENGTH;i++){
        // if this slot have not set information, then omit.
        if (my_mm_limits.uid[i]==-1){continue;}
        else{
            // if do not exceed limits, then omit
            if (mm_alloc[i] <= my_mm_limits.mm_max[i]){continue;}
            else{
        /*
        * If the task is already exiting, don't alarm the sysadmin or kill
        * its children or threads, just set TIF_MEMDIE so it can die quickly
        */
                if (max_rss_process[i]->flags & PF_EXITING) {
                    set_tsk_thread_flag(max_rss_process[i], TIF_MEMDIE);
                    return;
                }
                task_lock(max_rss_process[i]);
                    printk(KERN_ERR
"uid=%d,\tuRSS=%d,\tmm_max=%d,\tpid=%d,\tpRSS=%d\n",
                        max_rss_process[i]->cred-
>uid,mm_alloc[i],my_mm_limits.mm_max[i],max_rss_process[i]->pid,max_rss[i]);
                task_unlock(max_rss_process[i]);

        /*
        * Kill all user processes sharing victim->mm in other thread groups, if
        * any.  They don't get access to memory reserves, though, to avoid
        * depletion of all memory.  This prevents mm->mmap_sem livelock when an
        * oom killed thread cannot exit because it requires the semaphore and
        * its contended by another thread trying to allocate memory itself.
        * That thread will now get access to memory reserves since it has a
        * pending fatal signal.
        */

                for_each_process(p) {
                    if (p->mm == max_rss_process[i]->mm &&
!same_thread_group(p,max_rss_process[i])
                        && !(p->flags & PF_KTHREAD)) {
                            if (p->signal->oom_score_adj == OOM_SCORE_ADJ_MIN)
                                continue;

                            task_lock(p);
                            printk(KERN_ERR "Kill process %d (%s) sharing same
memory\n",p->pid, p->comm);
                            task_unlock(p);
                            do_send_sig_info(SIGKILL, SEND_SIG_FORCED, p, true);
                        }
                }

                set_tsk_thread_flag(max_rss_process[i], TIF_MEMDIE);
```

```
                do_send_sig_info(SIGKILL, SEND_SIG_FORCED, max_rss_process[i],
true);
                mm_alloc[i] -= max_rss[i];
            }
        }
    }
    /*
     * Check whether each slot satisfies the limits,
     * if not , do the whole procedure again.
     */
    for (i=0;i<MY_MM_LENGTH;i++){
        if (my_mm_limits.uid[i]==-1){continue;}
        else{
            if (mm_alloc[i] > my_mm_limits.mm_max[i]){
                goto my_rss_oom_begin;
            }
        }
    }
    read_unlock(&tasklist_lock);

    return;
}
EXPORT_SYMBOL(oom_killer_highest_rss);
```

Some pictures can clearly show the results will be showed in **Section5. Results**.

# 4. Bonus part

After I finishing the basic part, I continued to investigate this project freely. For the bonus part,

- I accomplish two more ways to set the rule of new oom-killer, they are:
    - OOM killer by longest running time
    - OOM killer by the highest "badness" points

      Here the idea of "badness" is highly inspired by the function **oom_badness()** in ***oom_kill.c***.
- I design a new kernel module to add oom-killer as a syscall and write a daemon program to execute the oom-killer syscall with the given time intervals.

## OOM Killer by Longest Running Time

The implementation is as below, which is quite similar to basic one.

```
void oom_killer_longest_run_time(void){
    /*
     * The implementation details of this function is
     * actually very similar to oom_killer_highest_rss().
     * If you get confused, you can refer to the comments
     * of oom_killer_highest_rss().
     */
    struct task_struct *p;
    int mm_alloc[MY_MM_LENGTH];
    struct task_struct* max_run_time_process[MY_MM_LENGTH];
    int rss_max_rtime_process[MY_MM_LENGTH];
    int i,found,index;
```

```
    read_lock(&tasklist_lock);
my_rtime_oom_begin:
    for(i=0;i<MY_MM_LENGTH;i++){
        mm_alloc[i] = 0;
        max_run_time_process[i] = NULL;
        rss_max_rtime_process[i] = 0;
    }
    for_each_process(p) {
        found = 0;
        for(i=0;i<MY_MM_LENGTH;i++){
            if (p->cred->uid == my_mm_limits.uid[i]){
                if (max_run_time_process[i] == NULL){
                    if (p->mm){
                        max_run_time_process[i] = p;
                        rss_max_rtime_process[i] = get_mm_rss(p->mm) * 4096;
                    }
                }
                else{
                    if (p->mm){
                        if((p->stime + p->utime) > (max_run_time_process[i]-
>stime + max_run_time_process[i]->utime)){
                            max_run_time_process[i] = p;
                            rss_max_rtime_process[i] = get_mm_rss(p->mm) * 4096;
                        }
                    }
                }
                found = 1;
                index = i;
                break;
            }
        }
        if (found){
            if (p->mm){
                mm_alloc[index] += get_mm_rss(p->mm) * 4096;
            }
        }
    }
    for(i=0;i<MY_MM_LENGTH;i++){
        if (my_mm_limits.uid[i] == -1){continue;}
        else{
            if (mm_alloc[i] <= my_mm_limits.mm_max[i]){continue;}
            else{
                if (max_run_time_process[i]->flags & PF_EXITING) {
                    set_tsk_thread_flag(max_run_time_process[i], TIF_MEMDIE);
                    return;
                }
                task_lock(max_run_time_process[i]);
                    printk(KERN_ERR
"uid=%d,\tuRSS=%d,\tmm_max=%d,\tpid=%d,\tpRSS=%ld,\ttotal_run_time=%ld\n",
                        max_run_time_process[i]->cred->uid, mm_alloc[i],
my_mm_limits.mm_max[i],
                        max_run_time_process[i]->pid,
get_mm_rss(max_run_time_process[i]->mm) * 4096,
                        max_run_time_process[i]->utime +
max_run_time_process[i]->stime);
                task_unlock(max_run_time_process[i]);

                for_each_process(p) {
```

```
                    if (p->mm == max_run_time_process[i]->mm &&
!same_thread_group(p, max_run_time_process[i])
                        && !(p->flags & PF_KTHREAD)) {
                            if (p->signal->oom_score_adj == OOM_SCORE_ADJ_MIN)
                                continue;

                            task_lock(p);
                            printk(KERN_ERR "Kill process %d (%s) sharing same
memory\n",p->pid, p->comm);
                            task_unlock(p);
                            do_send_sig_info(SIGKILL, SEND_SIG_FORCED, p, true);
                        }
                }

                set_tsk_thread_flag(max_run_time_process[i], TIF_MEMDIE);
                do_send_sig_info(SIGKILL, SEND_SIG_FORCED,
max_run_time_process[i], true);
                mm_alloc[i] -= (get_mm_rss(max_run_time_process[i]->mm) * 4096);
            }
        }
    }
    for (i=0;i<MY_MM_LENGTH;i++){
        if (my_mm_limits.uid[i]==-1){continue;}
        else{
            if (mm_alloc[i] > my_mm_limits.mm_max[i]){
                goto my_rtime_oom_begin;
            }
        }
    }
    read_unlock(&tasklist_lock);

    return;
}
EXPORT_SYMBOL(oom_killer_longest_run_time);
```

## OOM Killer by Largest "Badness" Points

First, I need to design a function to compute the "badness" points, which is as follows:

```
unsigned int compute_badness(struct task_struct* p){
    /*
     * This function will compute the "badness" of one process
     * inspired by function "badness" in "oom_kill.c".
     */
    long badness = 0;
    int total_pages = totalram_pages + total_swap_pages;

    task_lock(p);
    if (!(p->mm) || (p->signal->oom_score_adj == OOM_SCORE_ADJ_MIN)){
        task_unlock(p);
        return 0;
    }

    if (!total_pages)
        total_pages=1;

    badness = get_mm_rss(p->mm) + p->mm->nr_ptes;
```

```
    badness += get_mm_counter(p->mm, MM_SWAPENTS);

    badness *= 1000;
    badness /= total_pages;
    task_unlock(p);

    if (has_capability_noaudit(p, CAP_SYS_ADMIN))
        badness -= 30;

    badness += p->signal->oom_score_adj;

    if (badness <= 0)
        return 1;
    return badness;
}
```

Then the following implementation is just similar to the other two.

```
void oom_killer_worst(void){
    /*
     * The implementation details of this function is
     * actually very similar to oom_killer_highest_rss().
     * If you get confused, you can refer to the comments
     * of oom_killer_highest_rss().
     */
    struct task_struct *p;
    int mm_alloc[MY_MM_LENGTH];
    struct task_struct* worst_process[MY_MM_LENGTH];
    int rss_worst_process[MY_MM_LENGTH];
    int i,found,index;

    read_lock(&tasklist_lock);
my_badness_oom_begin:
    for(i=0;i<MY_MM_LENGTH;i++){
        mm_alloc[i] = 0;
        worst_process[i] = NULL;
        rss_worst_process[i] = 0;
    }
    for_each_process(p) {
        found = 0;
        for(i=0;i<MY_MM_LENGTH;i++){
            if (p->cred->uid == my_mm_limits.uid[i]){
                if (worst_process[i] == NULL){
                    if (p->mm){
                        worst_process[i] = p;
                        rss_worst_process[i] = get_mm_rss(p->mm) * 4096;
                    }
                }
                else{
                    if (p->mm){
                        if(compute_badness(p) >
compute_badness(worst_process[i])){
                            worst_process[i] = p;
                            rss_worst_process[i] = get_mm_rss(p->mm) * 4096;
                        }
                    }
                }
```

```
                found = 1;
                index = i;
                break;
            }
        }
        if (found){
            if (p->mm){
                mm_alloc[index] += get_mm_rss(p->mm) * 4096;
            }
        }
    }
    for(i=0;i<MY_MM_LENGTH;i++){
        if (my_mm_limits.uid[i] == -1){continue;}
        else{
            if (mm_alloc[i] <= my_mm_limits.mm_max[i]){continue;}
            else{
                if (worst_process[i]->flags & PF_EXITING) {
                    set_tsk_thread_flag(worst_process[i], TIF_MEMDIE);
                    return;
                }
                task_lock(worst_process[i]);
                    printk(KERN_ERR
"uid=%d,\tuRSS=%d,\tmm_max=%d,\tpid=%d,\tpRSS=%ld,\tbadness=%d\n",
                        worst_process[i]->cred->uid, mm_alloc[i],
my_mm_limits.mm_max[i],
                        worst_process[i]->pid, get_mm_rss(worst_process[i]->mm)
* 4096,
                        compute_badness(worst_process[i]));
                task_unlock(worst_process[i]);

                for_each_process(p) {
                    if (p->mm == worst_process[i]->mm && !same_thread_group(p,
worst_process[i])
                        && !(p->flags & PF_KTHREAD)) {
                            if (p->signal->oom_score_adj == OOM_SCORE_ADJ_MIN)
                                continue;

                            task_lock(p);
                            printk(KERN_ERR "Kill process %d (%s) sharing same
memory\n",p->pid, p->comm);
                            task_unlock(p);
                            do_send_sig_info(SIGKILL, SEND_SIG_FORCED, p, true);
                        }
                }

                set_tsk_thread_flag(worst_process[i], TIF_MEMDIE);
                do_send_sig_info(SIGKILL, SEND_SIG_FORCED, worst_process[i],
true);
                mm_alloc[i] -= (get_mm_rss(worst_process[i]->mm) * 4096);
            }
        }
    }
    for (i=0;i<MY_MM_LENGTH;i++){
        if (my_mm_limits.uid[i]==-1){continue;}
        else{
            if (mm_alloc[i] > my_mm_limits.mm_max[i]){
                goto my_badness_oom_begin;
            }
```

```
        }
    }
    read_unlock(&tasklist_lock);

    return;
}
EXPORT_SYMBOL(oom_killer_worst);
```

At last, here I use some pre-compile skills to decide which oom-killer will be used.

```
/*
     * Using some pre-compile skills and values defined before
     * to decide which oom_killer will be used.
     */
    #ifdef OOM_KILLER_HIGHEST_RSS
    oom_killer_highest_rss();
    #endif

    #ifdef OOM_KILLER_LONGEST_RUN_TIME
    oom_killer_longest_run_time();
    #endif

    #ifdef OOM_KILLER_WORST
    oom_killer_worst();
    #endif
```

If you want to use particular oom-killer, you just need to do a define statement.

## OOM Killer Syscall and Daemon

For the implementation of this syscall, since each oom-killer is written as a function in the kernel, so I just need to call particular oom-killer according to the given type. Here is some detailed codes and comments.

```
// 0 means highest RSS, 1 means longest run time , 2 means worst.
#define OOM_KILLER_TYPE 2

static int my_oom_killer(void){
    /*
     * Just call corresponding oom_killer by OOM_KILLER_TYPE.
     * As you can see, I use a switch statement here, so if
     * we design another efficient oom_killer, it's convinient
     * to add it to this syscall.
     */
    switch (OOM_KILLER_TYPE){
        case 0:
            oom_killer_highest_rss();
            break;
        case 1:
            oom_killer_longest_run_time();
            break;
        case 2:
            oom_killer_worst();
            break;
        default:
            oom_killer_highest_rss();
```

```
            break;
    }

    return 0;
}
```

For the daemon program, we need to convert arguments to the time intervals. And call **daemon(0,0)** to make this program always works in the background. Then we just need to call the syscall to call oom-killer we defined before.

Here is the source codes:

```
int main(int argc, char **argv){
    // this parameter can control how long the syscall will be called once.
    int sleepTime = atoi(argv[1]);
    printf("The time interval to run the oom killer is %d.\n", sleepTime);

    // To make this program to a daemon program.
    daemon(0,0);

    while(1){
        sleep(sleepTime);

        if (syscall(382) != 0){
            printf("oops, oom killer crashed, please rerun this daemon.");
            break;
        }

    }

    return -1;
}
```

***Note:*** since I write the oom-killer in a modularized way, it is very convenient to add some new oom-killers.