Report for Project 2

惠宇龙 518030910059 qinchuanhuiyulong@sjtu.edu.cn

June 11, 2020

1. Introduction

1.1 Objectives

1. Compile the Android kernel.
2. Familiarize with the information-sharing.
3. Familiarize with Android OOM killer.
4. Implement a new OOM killer.

1.2 Implementation Steps

1. Compile the Linux kernel.
2. Add a new system call.
3. Design and implement a new OOM killer.

2. Syscall

2.1 Information-sharing

I choose to deal with this problem by global variables. I make some prototype in a ".h" file, and define it in a ".c" file. Then the kernel can directly share these variables by include the ".h" file.

2.2 Data structure

Because there may be several users' information to store, we have to use a list or an array to store these information. In Linux kernel, there is a struct called list_head, so I can create my own list using this struct as follows.

```
#ifndef _LINUX_MMLIMIT_STRUCT_H
#define _LINUX_MMLIMIT_STRUCT_H

#include <linux/types.h>
#include <linux/list.h>

struct MmlimitStruct{
    uid_t uid;
    unsigned long mm_max;
    struct list_head list;
};

extern struct MmlimitStruct mmlimit_struct;

#endif
```

2.3 Implementation

To add a new system call, we should do the following changes:

1. In file: `arch/arm/include/asm/unistd.h`

   Add code: `#define __NR_set_mm_limit (__NR_SYSCALL_BASE+385)`

2. In file: `arch/arm/kernel/calls.S`

   Add code: `CALL(sys_set_mm_limit)`

3. In file: `include/linux/syscalls.h`

   Add code: `asmlinkage long sys_set_mm_limit(void);`

4. In file: `kernel/sys.c`

   Detailed implementation is as follows:

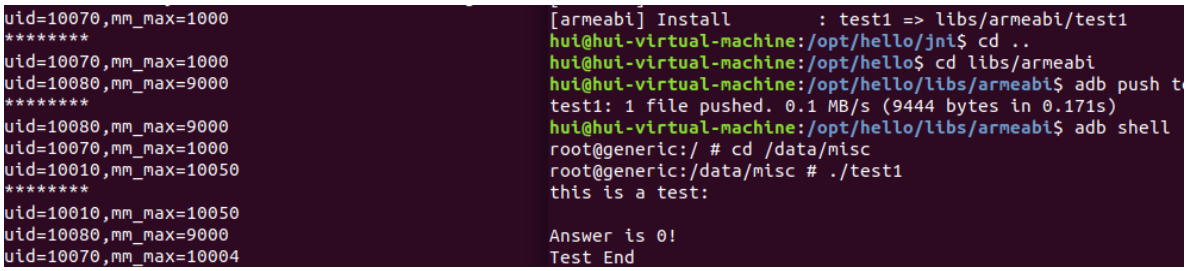
```
SYSCALL_DEFINE2(set_mm_limit, uid_t , uid , unsigned long , mm_max )
{
    struct MmlimitStruct *mm;
    struct MmlimitStruct *new_tmp;
    bool is_exist=0;

    // allocate the space for the new struct
    new_tmp=kmalloc(sizeof(struct MmlimitStruct),GFP_KERNEL);
    if(!new_tmp){
        printk("memory error");
        return -1;
    }
    //set the value
    new_tmp->uid=uid;
    new_tmp->mm_max=mm_max;
    INIT_LIST_HEAD(&new_tmp->list);

    //if it is existing, just update the data and print.
    list_for_each_entry(mm,&mmlimit_struct.list,list){

        if(uid==mm->uid) {
            is_exist=1;
            mm->mm_max=mm_max;
        }
        printk("uid=%d,mm_max=%lu\n",mm->uid,mm->mm_max);
    }

    // if it is not existing, add it to the list
    if(!is_exist)
    {
        list_add(&new_tmp->list,&mmlimit_struct.list);
        printk("uid=%d,mm_max=%lu\n",new_tmp->uid,new_tmp->mm_max);
        printk("*********\n"); // this is a separator for each syscall.
    }
    return 0;

}
```

2.4 Result

The test program is as follows:

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("this is a test:\n\n");
    int i=syscall(385,10070,1000);
    i=syscall(385,10080,9000);
    i=syscall(385,10010,10050);
    i=syscall(385,10070,10004);
    printf("Answer is %d!\n",i);
    printf("Test End\n");
    return 0;
}
```

And the result of this program is:



In the first three call, we just add a new user. And at the end, we change the information of uid=10070.

So, the result is correct, this system call is correct.

3. Original OOM Killer

Memory is managed in the form of pages in Linux. When allocating pages, the system will call the function `__alloc_pages_nodemask`. If there is not enough memory to use, the system will call the function `__alloc_pages_slowpath`. The `__alloc_pages_slowpath` function will trigger some memory reclamation and memory regulation mechanisms.

After that, if the available memory is still not enough, then the system will call `__alloc_pages_may_oom` function to do some checks for oom killer. If it meet the condition, the system will call `out_of_memory` to select a worst process and kill it through `oom_kill_process`.

(More detailed implementation is showed later, throughout my new OOM killer).

4. New OOM Killer

4.1 When to kill

In Linux kernel, when allocating pages, the system will call the function `alloc_pages_nodemask`. Since the project requires us to check the user's memory after every allocating, I just call my OOM-function (including checking and killing) at the end of the `alloc_pages_nodemask` function.

4.2 What to kill

Traverse the limit-list and get the information for every user. If the allocated memory is less than the limit, then there is nothing to be kill for this user. If the allocate memory exceeds the limit, then we should find the process with largest RSS and kill it. (The complex choosing mechanism will be introduced in the bonus part)

4.3 How to kill

The method to kill the process is based on the original function `oom_kill_process`. And there are several checking before the true killer (`do_send_sig_info`) in case there is some new memory available. More detailed implementation is showed as follow and it is explained in comment.

```
void new_oomkiller(void){
    // some variables used to traverse proc-list
    struct task_struct* task ;
    uid_t curr_uid;
    uid_t needed_uid;
    pid_t curr_pid;
    unsigned long curr_size;
    // used to traverse the user-limit
    struct MmlimitStruct *mm_limit;

    unsigned long tot_size;
    pid_t max_pro_pid;
    unsigned long max_pro_size;
    struct task_struct* max_pro;
    struct task_struct* victim;
    struct mm_struct *mm;
    int killed=0;

    //traverse the space_limit  for every set user
    list_for_each_entry(mm_limit,&mmlimit_struct.list,list)
    {
        needed_uid=mm_limit->uid;

        tot_size = 0;
        max_pro = NULL;
        max_pro_pid =0 ;
        max_pro_size = 0;

        /*
        * check whether exceed
        */
        // traverse the proc and calculate total-current-size
        task = &init_task ;
        for_each_process (task){
            curr_uid = task->cred->uid;
            if(curr_uid == needed_uid)
            {
                curr_pid = task->pid;
                if(task->mm)
                    curr_size = get_mm_rss(task->mm)<<PAGE_SHIFT;
                else
                    curr_size = 0;
                //update the information
                tot_size += curr_size;
                if(curr_size > max_pro_size){
                    max_pro = task;
                    max_pro_size = curr_size;
                    max_pro_pid = curr_pid;
                }
            }
```

```
        }

        /*
        * if exceed the limit
        */

        if(tot_size >  mm_limit->mm_max ){

                // if current process is existing
                if (fatal_signal_pending(current) || current->flags &
PF_EXITING) {
                        set_thread_flag(TIF_MEMDIE);
                        return;
                    }

                read_lock(&tasklist_lock);

                //kill max_pro

                // if max_pro is exiting
                if (max_pro->flags & PF_EXITING) {
                        set_tsk_thread_flag(max_pro, TIF_MEMDIE);
                        return;
                    }

                /*
                * 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
                */
                victim = find_lock_task_mm(max_pro);
                if(!victim)
                    return;

                killed=1;
                mm = victim->mm;
                printk(KERN_INFO "NEW-OOM-Killer  Choosen process info: uid=%d,"
                    "uRSS=%lu, mm_max=%lu, pid=%d, pRSS=%lu", needed_uid,
tot_size,                           mm_limit->mm_max, victim->pid, get_mm_rss(mm)
<<PAGE_SHIFT);
                task_unlock(victim);

            // Kill all user processes sharing victim->mm in other thread
groups, if any.
                for_each_process(max_pro)
                    if(max_pro->mm==mm && !same_thread_group(max_pro, victim) &&
!                           (max_pro->flags & PF_KTHREAD)){
                        if(max_pro->signal->oom_score_adj == OOM_SCORE_ADJ_MIN)
                            continue;
                        task_lock(max_pro); /* Protect ->comm from prctl() */
                        pr_err("Kill process %d (%s) sharing same memory\n",
                            task_pid_nr(max_pro), max_pro->comm);
                        task_unlock(max_pro);
                        // send signal to kill
                        do_send_sig_info(SIGKILL, SEND_SIG_FORCED, max_pro,
true);
```

```
                    }

                set_tsk_thread_flag(victim, TIF_MEMDIE);
                 // send signal to kill
                do_send_sig_info(SIGKILL, SEND_SIG_FORCED, victim, true);

                read_unlock(&tasklist_lock);

                if (killed && !test_thread_flag(TIF_MEMDIE))
                {
                    schedule_timeout_uninterruptible(1);
                }
        }
        }
}
```

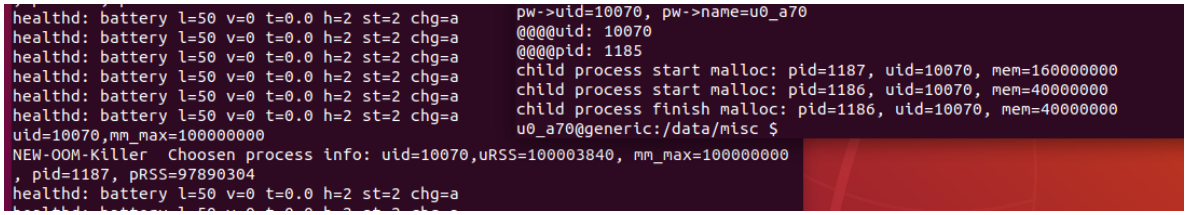
4.4 Result

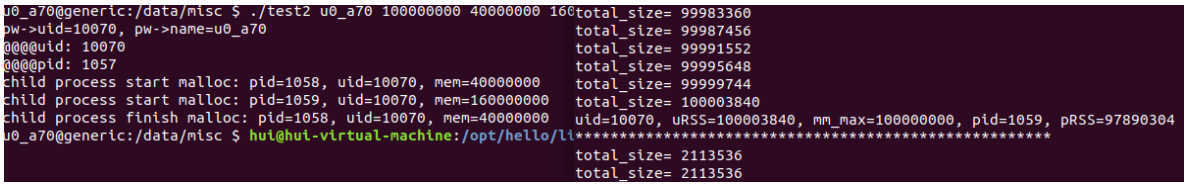
The test program is just as same as the offered one.

The result is as follows:

4.4.1 One Test

./test2 u0_a70 100000000 40000000 160000000:



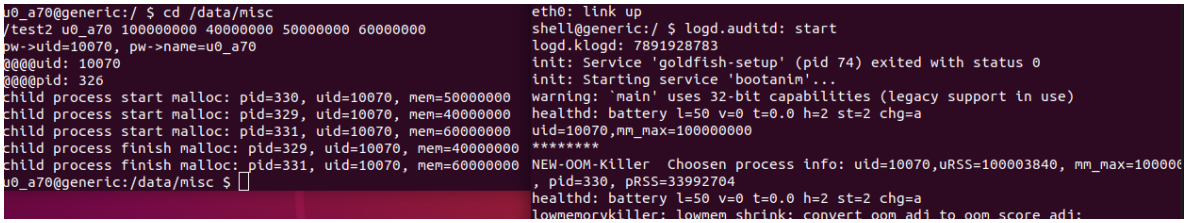


As you can see, in the first picture, the right part shows that the process with pid of 1186 was finished, but the 1187 one not. In the left part, we can see the NEW-OOM-Killer kill the 1187 one.

In order to show the effect of killing, I change the code and do the same test. In the second pictuare, the right part shows that the total-size decreased by 9789034 after killing.

4.4.2 Another Test

./test2 u0_a70 100000000 40000000 50000000 60000000



In this test, I make a more complicated input, and also get the correct result. Specially, when 60000000-process is allocating memory, there is new available space released from 40000000-process, so it doesn't exceed the limit.

4.4.3 Summary

Therefore, the implementation for the OOM Killer is correct.

5. Bonus

There are 3 advised bonus tips:

1. OOM killer should be awaken periodically.
2. Allow temporarily exceeding.
3. Design a new rule to choose a process.

And I implement them as a whole.

5.1 Advanced Setting Syscall

We should set allowed time for every user, so the struct and the syscall should be changed.

The code for struct propotype is as follows. I add some new variables to it.

```
struct MmlimitStruct{
    uid_t uid;
    unsigned long mm_max;
    unsigned int time;   // the allowed time
    struct list_head list;
    struct timer_list user_timer; // to set a timer in kernel
    bool mask; //to avoid to be killed during allowed_time
    bool flag; //to judge whether it's released from the mask-time or not.
};

extern struct MmlimitStruct mmlimit_struct;
```

As for the implementation of the advanced Syscall, I just add some instructions to set the allowed time and initialize the other variables.

```
SYSCALL_DEFINE3(set_mm_limit, uid_t , uid , unsigned long , mm_max, unsigned
int, time ){
    ....
    //set the value
    new_tmp->uid=uid;
    new_tmp->mm_max=mm_max;
    new_tmp->time=time;
    new_tmp->mask=0;
    new_tmp->flag=0;
    init_timer(&new_tmp->user_timer);
    INIT_LIST_HEAD(&new_tmp->list);
    ....
}
```

5.2 Daemon, new Syscall and Timer

To trigger the OOM-killer periodically, we should create a daemon and use a new syscall in it.

And this syscall will transfer the control to the kernel system, which will use timer to trigger OOM killer periodically.

The syscall with number 386 is as follows.(start_time should be ..ms)

```
SYSCALL_DEFINE1(check_oom_per_time, unsigned int , start_time){
    oom_killer_per_time(start_time);// the triggering will loop per start_time
    return 0;
}
```

Then we can create a daemon with the following code.

```
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    // return 0 for success
    if(daemon(0, 0))
    {
        printf("daemon error\n");
        exit(1);
    }
    syscall(386,100);
    return 0;
}
```

In the kernel Mode, we should use a timer to call `new_oomkiller` periodically as follows.

```
void new_oomkiller(void){
    struct MmlimitStruct *mm_limit;

    //traverse the space_limit  for every set user
    list_for_each_entry(mm_limit,&mmlimit_struct.list,list)
        handle_each_user(mm_limit);  //call  a handling function
}

struct timer_list my_timer;

static void my_function(unsigned long data){
    new_oomkiller();
    mod_timer(&my_timer,jiffies+ data*HZ/1000);
}

void oom_killer_per_time(unsigned int t){
    printk("check_oom_per_time is called\n");
    init_timer(&my_timer);
    // check once per t-ms
    my_timer.expires=jiffies+ t*HZ/1000;
    my_timer.data=t;
    my_timer.function = my_function;
    add_timer(&my_timer);
}
```

5.3 Select Process

Another thing different from the basic one is that we should design a method to choose a reasonable process.

I use some features (rss, pagetable, swap space and oom_socre_adj) from the original android OOM-killer. Also, I add some new features like switch-times and the age of process, which is explained in the comment.

```
struct task_struct* select_worst(struct MmlimitStruct *mm_limit ){
    uid_t needed_uid;
    uid_t curr_uid;
    int points=0;
    int max_points=0;
    struct task_struct* res=NULL;
    struct task_struct* p;

    needed_uid = mm_limit->uid;

    p = &init_task ;
        for_each_process (p){
            curr_uid = p->cred->uid;
            if(curr_uid == needed_uid)
            {
                if(!p->mm)
                    points=0;
                else
                {
                    //The baseline 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 *= 100;
                }
                // consider oom_socre_adj
                points += p->signal->oom_score_adj;

                // if p has lots of switches (measured by nvcsw + nivcsw),
                // it may be frequently executed for long time, which we should
not kill
                points -= p->nvcsw + p->nivcsw;

                // utime and stime measure the age of a process;
                // the earlier created p should be killed.
                points += (p->utime + p->stime)/HZ;
            }

            if (points>max_points)
                {
                    max_points=points;
                    res=p;
                }
        }
        return res;
}
```

5.4 Detailed Implementation

The important parts have been introduced above. However, we also need to make sure that during the allowed exceeding time, we should stop the useless checking and handling. Therefore, we should mask the `handle_each_user` function and unmask it after the allowed time. The unmasking function is as follows.

```
void set_unmask(unsigned long uid){
    struct MmlimitStruct *mm_limit;

    //traverse and find the user
    list_for_each_entry(mm_limit,&mmlimit_struct.list,list)
    {
        if(mm_limit->uid == uid){
            mm_limit->mask=0;
            return;
        }
    }
    printk("no correct uid to set mask\n");

}
```

The `handle_each_user` function is as follow, which should set masking and set a timer for `set_unmask` :

```
void handle_each_user(struct MmlimitStruct *mm_limit){

        // if it has been masked, just do nothing.
        if(mm_limit->mask)
            return;

        .....

        /*
        * if exceed the limit
        */
        if(tot_size >  mm_limit->mm_max ){


                if(!mm_limit->flag)
                        {
                            // if it is not released from the allowd masking.

                            printk("Firstly find the oom for uid=%u,
                                   "and wait some time\n",needed_uid);
                            mm_limit->flag=1;
                            //set the masking;
                            mm_limit->mask=1;

                            // check once per delay_time ms
                            delay_time=mm_limit->time;
                            // set a timer to release the user from mask
                            mm_limit->user_timer.expires=jiffies+
delay_time*HZ/1000;
                            mm_limit->user_timer.data=needed_uid;
                            mm_limit->user_timer.function =set_unmask;
                            add_timer(&mm_limit->user_timer);
```

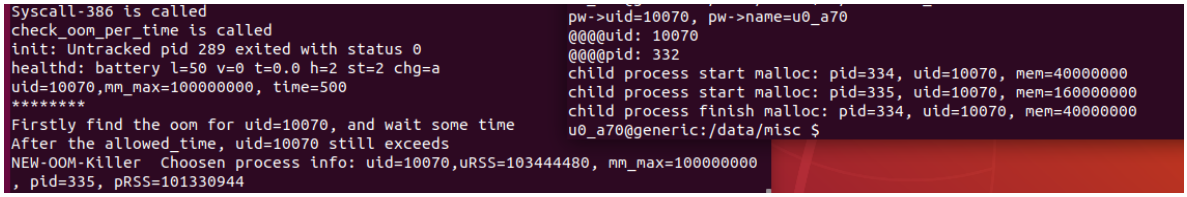
```
                            return;
                        }

                printk("After the allowed_time, uid=%u still
exceeds\n",needed_uid);

                // select a  worst process
                max_pro=select_worst(mm_limit);

                /*
                * kill max_pro
                */

                .....

        }
```

5.5 Test and Result.

The test program is the offered one, and I just add a time variable when using the 385 system call.

5.5.1 One Test

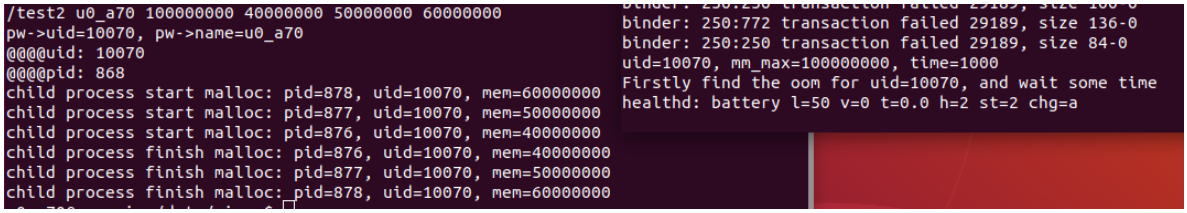
./test2 u0_a70 100000000 40000000 160000000:



As you can see, the result above is correct, and it shows the allowed time period.

5.5.2 Another Test

./test2 u0_a70 100000000 40000000 50000000 60000000



In this case, because there is some allowed time, all of the processes can be finished.

In the right part of picture, we can see: we find the oom at first, but after some time, there is new available space, which means we don't need to kill any process.

5.5.3 Summary

Therefore, the implementation is correct.

6. Conclusion

I have correctly finished the whole project and deal with some bonus parts.

I would like to thank Prof. Wu and TAs for careful guidance on the project.