

A New OOM Killer

Hanzhang Yang 518030910022

June 13, 2020

In this project, I designed a new OOM killer based on the linux operating system. However, something I have to mention first is that I used three modules and a user application to implement the oom killer. Therefore, I haven't change any files in linux kernel.

1 How the system call `set_mm_limit` works

There are two modules works for the system call: `mylimit` and `set_mm_limit`.

1.1 `mylimit`

There is a struct `MMLimit` which contains the information of user's uid, maximum memory limit and its time stamp (used in bonus part).

There are two global variables are defined. The first one is the number of limits: `limit_nr` and the second is a struct array contains all limits set for different users.

Then I use function `EXPORT_SYMBOL()` to make the two variables be global variables which are shared in the kernel.

The other part in this module are space application and variable initialization

```
18 typedef struct
19 {
20     uid_t uid;
21     unsigned long mmm_max;
22     unsigned int time_stamp;
23 } MMLimits;
24
25
26 MMLimits *my_mm_limits;
27 int limit_nr;
28 EXPORT_SYMBOL(limit_nr);
29 EXPORT_SYMBOL(my_mm_limits);
```

Fig. 1. global variables

1.2 set_mm_limit

In this module, I implement a new system call which is to set memory limit for users.

```
int set_mm_limit(uid_t uid,unsigned long mm_max,unsigned int time_allow_exceed)
```

Fig. 2. syscall interface

At first, we still need to extern define the two variables to use the global variables defined in the first module. Then the makefile should add a new words to pass the compiling: **export KBUILD_EXTRA_SYMBOLS**.

```
typedef struct
{
    uid_t uuid;
    unsigned long mmm_max;
    unsigned int time_stamp;
} MMLimits;

extern int limit_nr;
extern MMLimits *my_mm_limits;
```

Fig. 3. syscall interface

For the user with uid 0 is root (we have no wrights to kill its process), during every loop we set a new limit if the uid is 0 or update the limit if it find the same uid contained in the global variable.

```
int i=0;
for(i=0; i<3000; i++)
{
    if(uuid==my_mm_limits[i].uuuid)
    {
        my_mm_limits[i].mmm_max=mm_max;
        my_mm_limits[i].time_stamp=time_allow_exceed;
        break;
    }
    if(my_mm_limits[i].uuuid==0)
    {
        my_mm_limits[i].uuuid=uid;
        my_mm_limits[i].mmm_max=mm_max;
        my_mm_limits[i].time_stamp=time_allow_exceed;
        limit_nr++;
        break;
    }
};
```

Fig. 4. implementation

Everytime after adding a new limit, it will print the information of all limits in kernel.

2 How the original OOM killer is triggered

When kernel triggers OOM killer, it will call a function called **out_of_memory()**, which is called as following steps.

```

__alloc_pages //will be called when memory allocation
| - - > __alloc_pages_nodemask
| - - > __alloc_pages_may_oom
| - - > out_of_memory //triggered

```

In above functions, it will check the value of **oom_killer_disabled** before calling **__alloc_page_may_oom()**.

However, memory is managed by pages in linux. So no matter how to applicate memory, it will call **alloc_page()** and then call **out_of_memory()** when memory is not enough.

Linux allows users to applicate more memory than the physical which called overcommit. So allocating memory to users can not directly trigger OOM killer, for the memory is still enough in actual use. Only when allocating physical pages and the physical is not enough can it triggers OOM killer.

Here is the steps that original OOM killer works.

1. When the kernel detects that the system memory is insufficient, the function is triggered to execute.
2. When killing process to free memory, it will choose the "worst" process. It will rate all running processes and choose the process with the highest scores.
3. Finally, kill process (it may kill more than one processes instead of killing only the worst process).

3 The design and implementation of my oom killer

I implement a new syscall with a parameter which controls the interval time of oom killer works. Then I use another program to create a daemon process to use system call at regular intervals.

3.1 daemon process creation

In user program, using function `daemon` to make current process to be a daemon process which is always working.

```
daemon(0,0);
```

Fig. 5. daemon function

Then in the following part, I use a loop to use system call periodically. The sleeptime is the parameter which has been set by users.

```

while(1)
{
    syscall(382,sleeptime);
    sleep(sleeptime);
}

```

Fig. 6. using system call

3.2 system call of oom killer

This is the main part to finish a oom killer work. Everytime using this system call, it will traverse all process ,collect the information, compair with the limit set before and decide whether to kill some process.

There are three structs. **MMLimit** is used to keep the limit information in global variables. **prinfo** is used to store the information of all user process in their corresponding user's process array. **useinfo** is used to store user's information. **cur_mm** is used to sotre current RSS of this user. **proarray** is an array sotring all processes's information belong to this user.

```

typedef struct
{
    uid_t uid;
    unsigned long mmm_max;
    unsigned int time_stamp;
} MMLimits;

extern int limit_nr;
extern MMLimits *my_mm_limits;

struct prinfo
{
    pid_t pid;
    unsigned long rss;
};
struct useinfo
{
    uid_t uid;
    unsigned long cur_mm;
    int pro_nr;
    struct prinfo *proarray;
};

```

Fig. 7. several structs

At the first of this system call, I use **for_each_process** to traverse all processes. However, in this way some process belongs to kernel process. So it's illegal to visit these process's (mm) struct. Thus, I use (get_task_mm) to check the legality. If it's legal, I store the information in their corresponding user. To get the rss of process, I use function **get_mm_rss** to get current rss of process.

```
for_each_process(task)
```

```
{
    isfind=false;
    mm = get_task_mm(task);
    if (!mm)
        continue;
    tmprss=get_mm_rss(mm)<< (PAGE_SHIFT);
    for(i=0; i<use_nr; i++)
    {
        if(task->cred->uid==mymm[i].uid)
        {
            mymm[i].cur_mm+=tmprss;
            mymm[i].proarry[mymm[i].pro_nr].pid=task->pid;
            mymm[i].proarry[mymm[i].pro_nr].rss=tmprss;
            mymm[i].pro_nr++;
            isfind=true;
            break;
        }
    }
    if(isfind==true)
        continue;
    mymm[use_nr].uid=task->cred->uid;
    mymm[use_nr].cur_mm=tmprss;
    mymm[use_nr].proarry[0].pid=task->pid;
    mymm[use_nr].proarry[0].rss=tmprss;
    mymm[use_nr].pro_nr=1;
    use_nr++;
};
```

Fig. 8. collect information of all processes

After collecting the information of all users, I compare them with existing limits and check whether there are some users' rss is more than the limit.

```
for(i=0; i<limit_nr; i++)
{
    if(my_mm_limits[i].time_stamp>=time_step)
    {
        my_mm_limits[i].time_stamp-=time_step;
        break;
    };
    checkout(my_mm_limits[i].uuid,my_mm_limits[i].mmm_max);
}
```

Fig. 9. for exiting limit, check whether oom happened

Here is the **checkout()** function. When finding a user's rss is more than its limit, it will find the process of this user with maximum rss until the current rss of this user is under its limit. I use **kill_pid()** to kill process in kernel space by sending a signal to target process to kill it. After killing a process, it will print the information and update the information of user.

```

while(mymm[i].cur_mm>=limit)
{
    maxrss=0;
    //printk("%d\n",mymm[i].pro_nr);
    for(j=0; j<mymm[i].pro_nr; j++)
    {
        //printk("%u\t%lu\n",mymm[i].proarry[j].pid,mymm[i].proarry[j].rss);
        if(mymm[i].proarry[j].rss>maxrss)
        {
            maxpro=j;
            maxrss=mymm[i].proarry[j].rss;
        }
    }
    kill_pid(find_get_pid( mymm[i].proarry[maxpro].pid), SIGTERM, 1);
    printk("A process has been killed:\n uid=%u, uRSS=%lu, mm_max=%u, pid=%u, prSS=%lu\n", uid,
        mymm[i].cur_mm, limit, mymm[i].proarry[maxpro].pid, maxrss);
    mymm[i].cur_mm-=maxrss;
    mymm[i].pro_nr--;
    for(j=maxpro; j<mymm[i].pro_nr; j++)
    {
        mymm[i].proarry[j].pid=mymm[i].proarry[j+1].pid;
        mymm[i].proarry[j].rss=mymm[i].proarry[j+1].rss;
    }
}

```

Fig. 10. deal with the limit

4 Bonus part

4.1 Daemon process

As I explained before, I use a daemon process to use the system call periodically with a parameter T instead of checking memory after memory operation like the original OOM killer. After execute the OOM killer, you can use **ps** to check that the process is still exiting.

```

root      364      1      2384    304    hrtimer_na b669a4e0 S ./oom

```

Fig. 11. us ps command to check oom killer

4.2 time_allow_exceed

Maybe it is too strict to set a memory limit that can never be exceeded. So I do some changes to my OOM killer so that it allows the apps/users temporarily exceed the memory limit. Different **time_allow_exceed** can be set for different users.

At first, I add a new member which records how much time remaining to exceed the memory limit.

```

for(i=0; i<limit_nr; i++)
{
    if(my_mm_limits[i].time_stamp>=time_step)
    {
        my_mm_limits[i].time_stamp-=time_step;
        break;
    }
}

```

Fig. 12. us ps command to check oom killer

Everytime the daemon process uses the system call, oom killer decreases the time stamp. Only when the stamp is smaller than the time step can this process been checked for memory limit. And here I do some changes to the print format that I also print the time stamp.

4.3 the rest

Compared to the basic version, I kill processes until the user's current physical memory is under limit instead of killing only one process with highest RSS.

And the way I used to implement an oom killer is without any modifying on kernel codes. I think it's meaningful in linux kernel program.

5 Test

- 1) Compile the three modules and the oom.c file in the folder oom by using the makefile files (you may need to do some changes for the path set in makefile). And you must install module1 (mylimit.ko) first, for it stores the global variables which will be used in the other two modules.
- 2) Move these executable files to android virtual machine.
- 3) Install modules and execute the oom killer (the oom killer has a parameter which is the intervals of working). The parameter is not suitable to set too small which may cause some problems in kernel space (After trying, 3 might be suitable).

```
linqinluli@ubuntu:~/osprj2/test/jni$ adb devices
List of devices attached
emulator-5554    device

linqinluli@ubuntu:~/osprj2/test/jni$ adb push /home/linqinluli/osprj2/module1/mylimit.ko /data/misc
140 KB/s (44996 bytes in 0.312s)
linqinluli@ubuntu:~/osprj2/test/jni$ adb push /home/linqinluli/osprj2/module2/set_mm_limit.ko /data/misc
235 KB/s (44492 bytes in 0.184s)
linqinluli@ubuntu:~/osprj2/test/jni$ adb push /home/linqinluli/osprj2/module3/oom.ko /data/misc
155 KB/s (54644 bytes in 0.343s)
linqinluli@ubuntu:~/osprj2/test/jni$
linqinluli@ubuntu:~/osprj2/test/jni$ adb push /home/linqinluli/osprj2/oom/libs/armeabi/oom /data
63 KB/s (9444 bytes in 0.145s)
linqinluli@ubuntu:~/osprj2/test/jni$ adb push /home/linqinluli/osprj2/test/libs/armeabi/test /data
368 KB/s (9444 bytes in 0.025s)
linqinluli@ubuntu:~/osprj2/test/jni$ adb shell
root@generic:/ # cd data
root@generic:/data # cd misc
root@generic:/data/misc # insmod mylimit.ko
root@generic:/data/misc # insmod set_mm_limit.ko
root@generic:/data/misc # insmod oom.ko
root@generic:/data/misc # cd /
root@generic:/ # cd data
root@generic:/data # ./oom 3
oom begin to work
root@generic:/data # lsmod
Module              Size  Used by
oom                  1934   0
set_mm_limit         1084   0
mylimit              819   2 oom,set_mm_limit
```

Fig. 13. move and install

After these steps, the oom killer is working now.

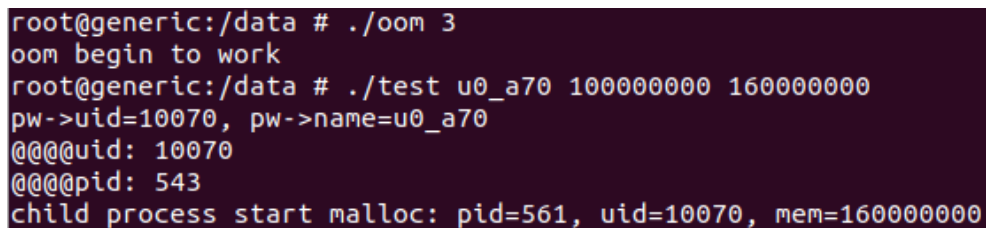
`syscall(381,uid_t uid,unsigned long mm_max,unsigned int time_allow_exceed)` can be used to set a limitation `mm_max` (Byte) for users with `uid`. If the user uses physical memory more than `mm_max`, the oom killer will work. And the last parameter is used to set a time that the user can exceed the memory limit.

After the oom killer begin to work, you can use test program to test the oom killer. The test program I used is based on the program provided by the professor and do some changes.

5.1 basic test

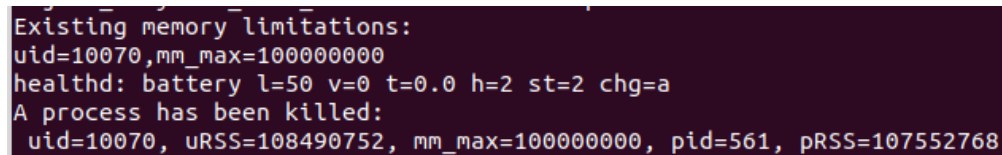
For basic functions, I implement an daemon process instead of calling oom killer after memory operations. When starting oom killer, there is a parameter which is the interval of oom killer works.

The test program is the same as the program provided on the canvas (I have changed the user in my program, so directly starting the test program is fine). There are two tests. Test 1 shows that oom killer can kill process when exceeding user's limit. Test 2 shows that it can kill process with highest RSS when exceeding user's limit and allow other process to finish there works.



```
root@generic:/data # ./oom 3
oom begin to work
root@generic:/data # ./test u0_a70 100000000 160000000
pw->uid=10070, pw->name=u0_a70
@@@uid: 10070
@@@pid: 543
child process start malloc: pid=561, uid=10070, mem=160000000
```

Fig. 14. test 1



```
Existing memory limitations:
uid=10070,mm_max=100000000
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
A process has been killed:
uid=10070, uRSS=108490752, mm_max=100000000, pid=561, pRSS=107552768
```

Fig. 15. result of test 1


```

root@generic:/data # ./test u0_a70 100000000 20000000 30000000 160000000
pw->uid=10070, pw->name=u0_a70
@@@uid: 10070
@@@pid: 1058
child process start malloc: pid=1061, uid=10070, mem=160000000
child process start malloc: pid=1060, uid=10070, mem=300000000
child process start malloc: pid=1059, uid=10070, mem=200000000
child process finish malloc: pid=1059, uid=10070, mem=200000000
child process finish malloc: pid=1060, uid=10070, mem=300000000

```

Fig. 16. test 2

```

Existing memory limitations:
uid=10070,mm_max=1000000000
A process has been killed:
uid=10070, uRSS=151519232, mm_max=1000000000, pid=1061, pRSS=150577152

```

Fig. 17. result of test 2

5.2 bonus test

This test mainly tests the `time_allow_exceed` function. Test program 1 will set a 20 seconds exceeding time and test program 2 set as 0. So the result is that program 1 can finish its memory allocation and program 2 cannot with the same parameters.

```

root@generic:/data # ./oom 3
oom begin to work
root@generic:/data # ./test1 u0_a70 100000000 160000000
pw->uid=10070, pw->name=u0_a70
@@@uid: 10070
@@@pid: 1138
child process start malloc: pid=1139, uid=10070, mem=160000000
child process finish malloc: pid=1139, uid=10070, mem=160000000
root@generic:/data # ./test2 u0_a70 100000000 160000000
pw->uid=10070, pw->name=u0_a70
@@@uid: 10070
@@@pid: 1140
child process start malloc: pid=1141, uid=10070, mem=160000000

```

Fig. 18. test of `time_allow_exceed`

```

module load!
module load!
module load!
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
Existing memory limitations:
uid=10070,mm_max=1000000000,time_stamp=20
Existing memory limitations:
uid=10070,mm_max=1000000000,time_stamp=0
A process has been killed:
uid=10070, uRSS=111218688, mm_max=1000000000, pid=1141, pRSS=110280704

```

Fig. 19. result of bonus test