Report of Project 2

卞思沅 518021910656

1. Introduction

In this project, I have implemented a new system call to set the memory limit, and then devise a new oom killer to kill the largest process whose user has exceeded its memory limit. In the report, I will

1. Explain how your system call works in detail.
2. Explain how the original OOM killer is triggered.
3. Explain how you design and implement your new OOM killer.
4. Show the results of my implementation
5. Explain my implementation of some extra parts.

2. System call

Generally speaking, I add system call according to the blog[1]. Here I will give a detailed explanation of my code.

In the sys_arm.c file, I define a new structure to help record the memory limit of each user. The struct is defined as below:

```
struct MMLimits {
    uid_t uid;                  // user id
    long mm_max;                // the memory limit
    struct list_head next;      // point to next element
};
```

The notable point is that I use list_head to help us simplify the code. With the help of *list_head*, we do not need to use *kmalloc* to allocate kernel memory at first call, and we do not need to initiallize the whole array as well. We can add struct to the tail of the list one by one whenever necessary. In this way, we have saved a lot of memory space. And there is no need to worry about whether the array is long enough or whether the space is enough to store the array. So the number of possible problems drastically decreases.

We define a global `list_head` variable *mm_limit_head*. In later part, we will refer to memory limits in my new *oom_killer* through this variable.

In the system call, we first check whether the current user has already has a limit. If so, we just update the limit. Otherwise, we create a new *MMLimits* struct and add the struct to the tail to the *list_head*. Here is the code:

---

[1] http://blogsmayan.blogspot.com/p/adding-simple-system-call.html

```c
int sys_set_mm_limit(uid_t uid, unsigned long mm_max) // newly added
{
    struct MMLimits *mmp;
    int flag = 0;

    mutex_lock(&init_mutex);
    list_for_each_entry(mmp, &mm_limit_head, next) {
        // check if uid has already has a limit
        if(mmp->uid == uid) {
            mmp->mm_max = mm_max;
            flag = 1;
            break;
        }
    }

    if(flag == 0){
        // if uid has no limit before, add a new one
        struct MMLimits *tmp = NULL;
        tmp = (struct MMLimits*)( kmalloc(sizeof(struct MMLimits), GFP_KERNEL) );
        if(!tmp) {
            // if allocation fails
            printk("allocation fail");
            mutex_unlock(&init_mutex);
            return 1;
        }
```

```c
        tmp->uid = uid;
        tmp->mm_max = mm_max;

        list_add_tail(&tmp->next, &mm_limit_head);
    }

    list_for_each_entry(mmp, &mm_limit_head, next) {
        printk("uid=%d, mm_max=%ld\n", mmp->uid, mmp->mm_max);
    }
    mutex_unlock(&init_mutex);

    return 0;
}
```

3. Original oom killer

The original oom_killer is triggered by *__alloc_pages_may_oom()* in *page_alloc.c*. *__alloc_pages_may_oom()* will call *out_of_memory()* when the memory is exhausted. Then *out_of_memory()* will then kill one process according to its designed algorithm.

In *out_of_memory()*, it first do some checks to make sure that we have no other options but kill one process. Then it will call *select_bad_process()* to find a victim process. If the selected process is killable, it will use *oom_kill_process()* to kill it, and return to normal routine.

The *select_bad_process()* function chooses the victim process according to point of each process scored by *oom_badness()*. Specifically, it will call *oom_badness()* to rate a point from 1 to 1000, and the process with the highest score will be selected as the victim process. Function *oom_badness()* rates the score mostly by the amount of memory one process has used, but it also takes other factors into account. For example, if the process belongs to a root user, it will get lower score. And *p->signal->oom_score_adj* is also used to avoid kill important processes.

4. New oom killer

To check and kill a process when the user memory limit has been exceeded, we devise a new oom killer. Here are detailed explanations.

In *__aloc_pages_nodemask()*, we add a new line, in which we call the new function: *new_oom_killer()*. And we add a mutex to protect it.

In *new_oom_killer()*, we first use *find_largest_p()* to check whether there exists a user whose memory usage exceeds its limit. If so, *find_largest_p()* will just return the largest process of that user, and we kill that process by *new_oom_kill_process()*.

new_oom_killer() is quite similar to the function *out_of_memory()* from oom_kill.c, except that I discard some unnecessary checks, because in our new oom killer, we only want to kill the exact out-of-memory process, and there is no need to check whether other processes are able to be killed and so on. And *new_oom_kill_process()* is also quite similar to *oom_kill_process()*. I just also discard some unnecessary parts. The details can be refered in the code.

We use *find_largest_p()* to help us retrive the largest process whose user exceeds memory limit. If no user exceeds it memory limit, it will just return NULL, otherwise, it returns the needed process described above. In the function, we just traverse the MMLimits list, and check the users one by one.

In the following part, I will just show part of important codes.

```
     */
    if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
        goto retry_cpuset;


    mutex_lock(&new_oom_killer_mutex);

    new_oom_killer(zonelist, gfp_mask, order, nodemask);

    mutex_unlock(&new_oom_killer_mutex);

    return page;
}
EXPORT_SYMBOL(__alloc_pages_nodemask);
```

```c
/*
 * Traverse the global variable mm_limit, check whether user has exceeded the limit.
 * If one user has exceeded its limit, return the "largest" process of that user.
 */
struct task_struct* find_largest_p(unsigned long totalpages,
        struct mem_cgroup *memcg, const nodemask_t *nodemask)
{
    uid_t uid;
    int flag = 0;
    long limit, now_mem, max_mem;
    struct task_struct *p = NULL;
    struct MMLimits *mmp = NULL;

    list_for_each_entry(mmp, &mm_limit_head, next) {
        // Traverse the MMLimits to check whether one user exceeds its limit
        uid = mmp->uid;
        limit = mmp->mm_max;
        now_mem = get_total_mm_rss(uid);

        if(now_mem * 4096 > limit){
            // The user has exceeded memory limit
            p = _find_largest_p(uid, &max_mem, totalpages, memcg, nodemask);
            flag = 1;
        }
        if(flag == 1) break;
    }
```

```c
    if(p != NULL){
        task_lock(p);
        printk("uid=%d,uRSS=%ld,mm_max=%ld,pid=%ld,pRSS=%ld\n",
            (int)(uid), now_mem, limit, p->pid, max_mem);
        task_unlock(p);
    }
    if(flag == 1 && p == NULL)
        printk("User memory limiit exceeded, but no process is killable\n");

    return p;
}
```

```c
/*
 * Check whether one user has exceeded its memory limit
 * If yes, kill the largest process of that user
 *
 * The inplementation is quite similar to out_of_memory,
 * I just delete some unnecessary checking and make some minor changes
 */
void new_oom_killer(struct zonelist *zonelist, gfp_t gfp_mask,
        int order, nodemask_t *nodemask)
{
    const nodemask_t *mpol_mask;
    struct task_struct *p;
    unsigned long totalpages;
    unsigned int points;
    enum oom_constraint constraint = CONSTRAINT_NONE;

    /*
     * Check if there were limitations on the allocation (only relevant for
     * NUMA) that may require different handling.
     */
    constraint = constrained_alloc(zonelist, gfp_mask, nodemask,
                    &totalpages);
    mpol_mask = (constraint == CONSTRAINT_MEMORY_POLICY) ? nodemask : NULL;
    check_panic_on_oom(constraint, gfp_mask, order, mpol_mask);

    read_lock(&tasklist_lock);
```

```c
    // Check whether need to kill one process
    p = find_largest_p(totalpages, NULL, mpol_mask);
    points = 1000;

    if (!p) {
        // No user ot of limit, or no process is killable
        read_unlock(&tasklist_lock);
        return;
    }
    if (PTR_ERR(p) != -1UL) {
        new_oom_kill_process(p, gfp_mask, order, points, totalpages, NULL,
                nodemask, "User memory limit exceeded");
    }

    read_unlock(&tasklist_lock);
}
```

5. Results

I have set the memory limit for many users, and the new oom killer still makes good result. Here are the screen shots.

AVD shell:

```
^C
130|root@generic:/data/misc # su 10060
u0_a60@generic:/data/misc $ ./testARM u0_a60 100000000 160000000
pw->uid=10060, pw->name=u0_a60
@@@@uid: 10060
@@@@pid: 416
child process start malloc: pid=418, uid=10060, mem=160000000
u0_a60@generic:/data/misc $
```

Kernel:

```
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
set_mm_limit startsuid=10060, mm_max=100000000
uid=10070, mm_max=100000000
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
type=1400 audit(1592581552.690:6): avc: denied { module_request } for pid=76 c
omm="netd" kmod="netdev-dummy0" scontext=u:r:netd:s0 tcontext=u:r:kernel:s0 tc
lass=system permissive=0
type=1400 audit(1592581552.700:7): avc: denied { sys_module } for pid=76 comm=
"netd" capability=16 scontext=u:r:netd:s0 tcontext=u:r:netd:s0 tclass=capabili
ty permissive=0
uid=10060,uRSS=24415,mm_max=100000000,pid=418,pRSS=23900
Killed process 418 (testARM) total-vm:166224kB, anon-rss:95452kB, file-rss:148
kB
type=1400 audit(1592581571.750:8): avc: denied { write } for pid=459 comm="fin
gerprintd" name="/" dev="mtdblock1" ino=2 scontext=u:r:fingerprintd:s0 tcontex
t=u:object_r:system_data_file:s0 tclass=dir permissive=0
```

The kernel also prints much debug information below, that is because I just call the new_oom_killer, and keeps many debug outputs. But they are not important. And between the lines are some other debug information, they have nothing to do with our project either, and it seems that the netd process whose pid=76 always has some problems, no matter whether we implement the new oom killer or not. And we should note that the unit of mm_max is byte, and the unit for RSS is page, which may cause some misunderstanding.

Note that the string *"child process finish malloc:"* is not printed in AVD shell, that is because before the finish of memory allocation of that child process, it has been killed by our now oom killer.

But in fact, there are still some unsatisfying parts in my implementation. I have found that the new oom killer will be called every time a new page is allocated. That influences the speed of memory allocation. I think that should be improved with further effort.

6. Extra parts

Because time limit, I fail to complete bonus part 1(trigger new OOM killer periodically). I successfully finished **bonus 2 and 3**. I have changed my code to take into account the time limit of exceeding memory limit. And I also add a new_oom_badness function to design a new reasonable rule to choose a victim process.

First, to record the time limit, I change the MMLimits struct and add a new oom_time_record struct to record the start time of memory exceeding as below:

```c
struct MMLimits {
    uid_t uid;                      // user id
    long mm_max;                    // the memory limit
    long time_limit;                // how long it can exceed its memory limit
    struct list_head next;          // point to next element
};

struct oom_time_record{
    uid_t uid;
    long estart_t;                  // start time of memory exceeding
    struct list_head next;
};
```

When we start a new oom killer, if the memory limit has been exceeded for the first time, we will record the start time of memory exceeding. Then when a second time a new oom killer is triggered, if the time limit for the user is still exceeded, we will check the time limit. But if the memory gets back to within the limit, then we will just remove the start time of memory exceeding of that user. If one user has exceeded the memory limit longer than the time limit, then one process will be killed.

The new_oom_badness function follows the old oom_badness function in that they all account whether the process is a root process and how large the oom_score_adj of a process is. But new_oom_badness evaluates more factor. It accounts both the start time of a process and the space one process takes. The earlier one process starts, the less likely it will be killed. That definitely decreases the chance of killing some important tasks. The processing time of a user is obtained by p->start_time.

Here are the key parts.

```c
/*
 * Evaluate how bad a process is, take into account:
 * 1. space
 * 2. processing time
 * 3. Other factors such as root process and unkillable and oom_score_adj
 */
unsigned int new_oom_badness(struct task_struct *p,  long t_limit, long m_limit,
        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;
```

```c
m_limit = m_limit / 4096; // change the unit
if(m_limit == 0) m_limit = 1;
points = get_mm_rss(p->mm) * 800 / m_limit;

/*
 * 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;

/*
 * Check the time, and adjust the point
 */
struct timespec now_time, start_t;
time_t ntime, stime;
getnstimeofday(&now_time);
ntime = now_time.tv_sec;
start_t = p->start_time;
stime = start_t.tv_sec;
```

```c
    if(t_limit == 0) t_limit = 1;
    long time_point = (ntime - stime) / t_limit;
    // The longer one process has existed, the less likely ot will be killed.
    time_point = (time_point > 50) ? 0 : (200 - time_point * 4);
    points += time_point;

    task_unlock(p);
    /*
     * 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;
    points = (points > 1000) ? 1000 : points;
    return points;
}
```

```c
/*
 * Set the start time of oom for one user.
 * If has already been set, then check whether time limit has been exceeded.
 * If time limit has been exceeded, then return -1;
 * If error occurs, return -2;
 */
int set_oom_time(uid_t uid, unsigned int t_limit)
{
    struct oom_time_record *rp;
    struct timespec now_time;
    time_t ntime;

    getnstimeofday(&now_time);
    ntime = now_time.tv_sec;

    list_for_each_entry(rp, &time_limit_head, next) {
        // check if uid has already has a limit
        if(rp->uid == uid) {
            if(ntime - rp->estart_t >= t_limit){
                // Time limit exceeded
                list_del(&rp->next);
                kfree(rp);
                return -1;
            }
            // else, return total exceeding time
            else {
                return (ntime - rp->estart_t);
            }
```

```c
    // If not in list, add a new element
    struct oom_time_record *tmp = NULL;
    tmp = (struct oom_time_record*)( kmalloc(sizeof(struct oom_time_record), GFP_KERNEL) );
    if(!tmp) {
        // if allocation fails
        printk("allocation fail\n");
        return -2;
    }

    tmp->uid = uid;
    tmp->estart_t = ntime;
    list_add_tail(&tmp->next, &time_limit_head);

    return 0;
}
```

Result of bonus part: (I have set the default time limit to be one second)

```
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
set_mm_limit starts
uid=10070, mm_max=100000000, time_limit=1
uid=10060, mm_max=10000000, time_limit=1
uid=10060,pid=600, points=1000
uid=10060,uRSS=2442,mm_max=10000000,pid=600,pRSS=1927, time_limit=1
Killed process 600 (testARM) total-vm:18768kB, anon-rss:7560kB, file-rss:148kB
```

```
u0_a60@generic:/data/misc $ ./testARM u0_a60 10000000 16000000 1000000
pw->uid=10060, pw->name=u0_a60
@@@@uid: 10060
@@@@pid: 598
child process start malloc: pid=600, uid=10060, mem=16000000
child process start malloc: pid=601, uid=10060, mem=1000000
child process finish malloc: pid=601, uid=10060, mem=1000000
u0_a60@generic:/data/misc $
```

The time limit is 1s. Because the allocation takes quite much time (1s or nearly 1s), so the total badness points get to 1000 for the first process, and thus the first child process is killed.