CS356 Operating System Projects Spring 2020

Project 2: Memory Limit for Applications

Objectives:

- Compile the Android kernel.
- Familiarize with how information can be shared in the kernel.
- Familiarize with Android OOM killer.
- Implement a new OOM killer that support memory limit for applications.

Make sure your system is a 64-bit system.

Problem Statement:

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. To avoid crashing the whole system including all processes, Linux implements an OOM killer to selectively kill some processes and reclaim the physical memory allocated to these processes. The OOM killer selects processes to kill using a variety of heuristics, typically targeting the process that 1) uses a large amount of physical memory and 2) is not important.

Although this OOM killer works reasonably well, it has one security flaw: it ignores which users created the processes. 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. This security flaw has implications on Android as well. In Android, each app is represented as a user. When a user runs, it can create more than one process. Thus, a malicious user can create a large number of processes, causing other users' processes be killed.

In this project, you will fix the security flaw by implementing a new OOM killer that supports per-user memory limit. Specifically, you will do the following works:
1) Implement a new system call to set per-user memory limit. (recall that each Android app is represented as a unique user)
2) Study how memory is allocated and how the original OOM killer works.
3) Add a new OOM killer in the Android kernel to kill a process owned by a user when the user's processes run out of the user's memory quota.

Roadmap:

1. **Compile the Linux kernel.**
    a. Make sure that you have added the following path into your environment variable. ANDROID_NDK_HOME/toolchains/arm-linux-androideabi-4.6/prebuilt/linux-x86_64/bin
    b. Open Makefile in KERNEL_SOURCE/goldfish/, and find these:

```
ARCH            ?=  $(SUBARCH)
CROSS_COMPILE   ?=
```

Change it to:

```
ARCH            ?=  arm
CROSS_COMPILE   ?=  arm-linux-androideabi-
```

Save it and exit.

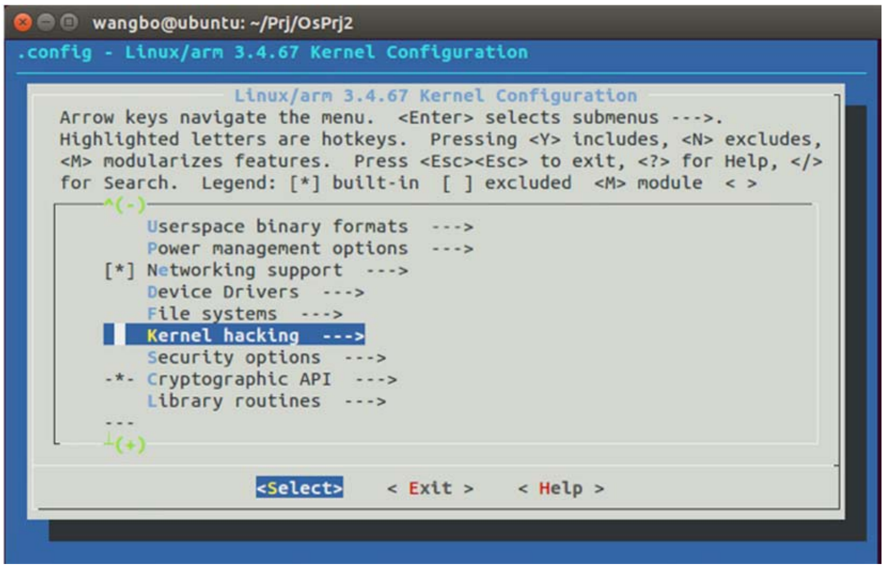
c. Execute the following command in terminal to set compiling configuration:

```
make goldfish_armv7_defconfig
```

d. Modify compiling configuration:

```
sudo apt-get install ncurses-dev
make menuconfig
```

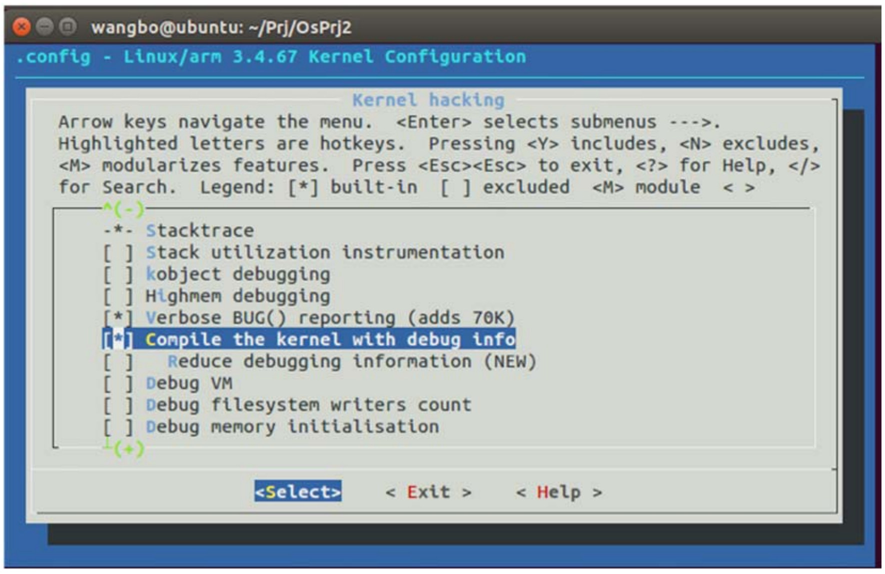
Then you can see a GUI config.



```
wangbo@ubuntu: ~/Prj/OsPrj2
.config - Linux/arm 3.4.67 Kernel Configuration
                    Linux/arm 3.4.67 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >
      ^(-)
          Userspace binary formats  --->
          Power management options  --->
      [*] Networking support  --->
          Device Drivers  --->
          File systems  --->
          Kernel hacking  --->
          Security options  --->
      -*- Cryptographic API  --->
          Library routines  --->
      ---
      └(+)

              <Select>    < Exit >    < Help >
```

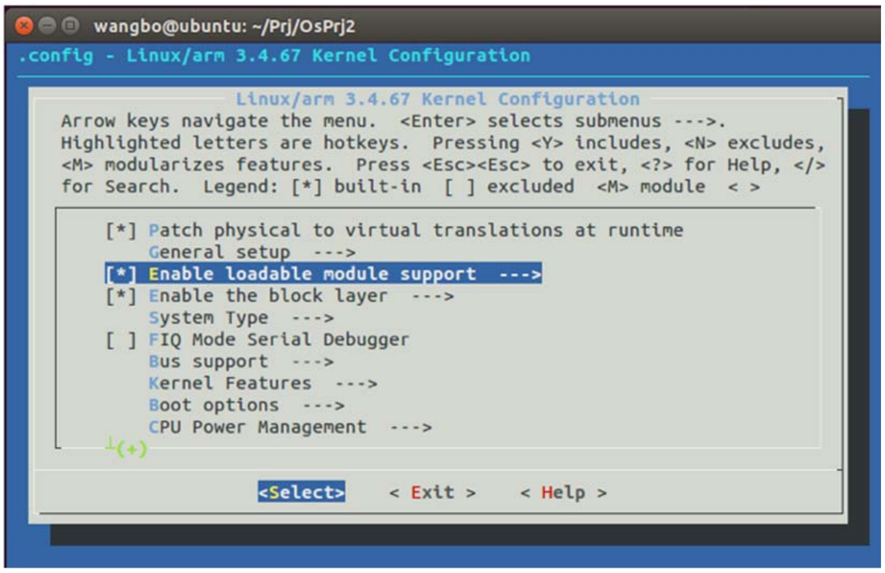
Open the *Compile the kernel with debug info* in *Kernel hacking*.



```
wangbo@ubuntu: ~/Prj/OsPrj2
.config - Linux/arm 3.4.67 Kernel Configuration
                           Kernel hacking
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >
      ^(-)
      -*- Stacktrace
      [ ] Stack utilization instrumentation
      [ ] kobject debugging
      [ ] Highmem debugging
      [*] Verbose BUG() reporting (adds 70K)
      [*] Compile the kernel with debug info
      [ ]   Reduce debugging information (NEW)
      [ ] Debug VM
      [ ] Debug filesystem writers count
      [ ] Debug memory initialisation
      └(+)

              <Select>    < Exit >    < Help >
```

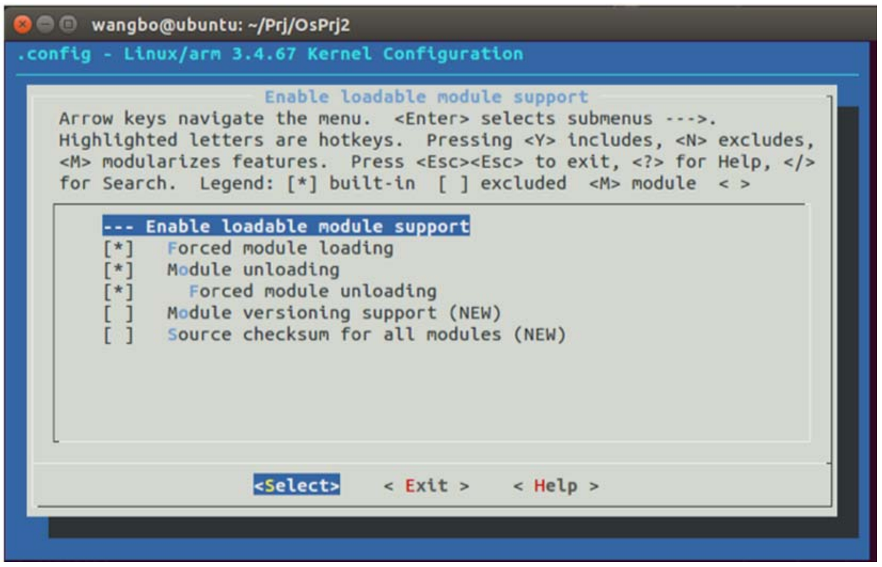
Enable loadable module support with *Forced module loading, Module unloading and Forced module unloading* in it.



```
wangbo@ubuntu: ~/Prj/OsPrj2
.config - Linux/arm 3.4.67 Kernel Configuration
                    Linux/arm 3.4.67 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

      [*] Patch physical to virtual translations at runtime
          General setup  --->
      [*] Enable loadable module support  --->
      [*] Enable the block layer  --->
          System Type  --->
      [ ] FIQ Mode Serial Debugger
          Bus support  --->
          Kernel Features  --->
          Boot options  --->
          CPU Power Management  --->
      └(+)

              <Select>    < Exit >    < Help >
```



```
wangbo@ubuntu: ~/Prj/OsPrj2
.config - Linux/arm 3.4.67 Kernel Configuration
                    Enable loadable module support
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

      --- Enable loadable module support
      [*]   Forced module loading
      [*]   Module unloading
      [*]     Forced module unloading
      [ ]   Module versioning support (NEW)
      [ ]   Source checksum for all modules (NEW)

              <Select>    < Exit >    < Help >
```

Save it and exit.

e. Compile

make -j4

The number of -j* depends on the number of cores of your system.

2. Add a New System Call

You need to design and add a new system call, which will cooperate with the new OOM killer. The signature of the new system call should be:

int set_mm_limit(uid_t uid, unsigned long mm_max);

The system call has two parameters:

uid: the id of the user that we want to set memory limit for

mm_max: maximum amount of physical memory the user can use (in bytes).

You should save the memory limit information (i.e., 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. To enable data sharing between the system call and the OOM killer, you can define a new global variable my_mm_limits in the Linux kernel to store the shared data (you are also encouraged to explore other data sharing methods). You can refer to task_struct to learn how to define a struct in C. Then define your own struct MMLimits for your global variable in a proper location (in a proper file).

For your system call set_mm_limit(), besides its basic function which is to set the memory limit, you should also implement the following features:

1) When set_mm_limit is invoked to set a memory limit for some user, after it successfully adds the memory limit, it traverses and outputs all existing memory limit entries using printk(). The output format of each entry is:

```
printk("uid=%d, mm_max=%d", uid, mm_max);
```

2) When set_mm_limit() is invoked, if it finds out that there already exists a memory limit for that uid, it should update the old memory limit entry instead of adding a new one.

3) If all works of set_mm_limit() has been successfully done, it returns 0. (You can make the system call return other values if errors are detected.)

3. Design and Implement A New OOM Killer

You should add your new OOM killer at a proper place in the file *goldfish/mm/page_alloc.c*. After your OOM killer is triggered, it should check whether there is a user who has exceeded the memory limit. In order to do the check, your OOM killer has to traverse all processes, and collects the ones that are created by a specific user.

To count the total amount of physical memory allocated to a user, use the Resident Set Size (RSS), which tracks the amount of physical memory currently allocated to a process. RSS is stored in struct mm_struct in task_struct. If a user has run out its memory quota, your OOM killer should kill the process that has the highest RSS among all processes belonging to the user. The function kill() cannot be used in the kernel, so you should investigate how to kill a process in your new OOM killer.

If the new OOM killer kills a process, it should output some information using printk(). The output format is:

```
printk("uid=%d, uRSS=%d, mm_max=%d, pid=%d, pRSS=%d", uid,
user_rss_before_killing, mm_max, killed_process_pid, killed_process_rss);
```

If the limit we set is smaller than the RSS of the currently running user, a process with the highest RSS of this user should be killed upon the next physical memory allocation request. Basically, to maintain the memory limit, your new OOM killer should be triggered to check the memory usage after every physical memory allocation. (We also encourage you to propose some other trigger schemes, so that the new OOM killer can be more efficient.)

To help your debugging, we have created a test program *prj2_test.c*. It takes two or more parameters. The first parameter specifies the username of the user you want to set memory quota for. The second parameter specifies the memory quota (in bytes). Each parameter, from the third to the last, causes the test program to fork a process and allocate the specified amount of memory (in bytes). For example, you can first set a (≈)100MB memory quota for username u0_a70 (i.e., uid 10070), and then fork a new process which requests (≈)160MB memory by running the following command (100000000 bytes actually equals to 95.36MB):

```
./prj2_test u0_a70 100000000 160000000
```

You can compile it by yourselves, and run it in your AVD.

Implementation Details:

1. You can use "adb shell" to login as root user, and use "su" command in adb shell to switch to a specific user. For example, if currently a user with username u0_a70 is running, you can type "su 10070" (10070 is the uid of user u0_a70) in adb shell to switch to that user.
2. It may be necessary for you to search for and refer to an existing global variable such as init_task and some other documents, so that you can understand how a global variable is defined, declared, initialized, and used in Linux kernel.
3. You can learn how the existing out-of-memory (OOM) killer is triggered when the memory runs out. You can learn from the following function chain in goldfish/mm/page_alloc.c:
```
__alloc_pages()    //invoked when allocating pages
    |-->__alloc_pages_nodemask()
        |--> __alloc_pages_slowpath()
            |--> __alloc_pages_may_oom()
                | --> out_of_memory()      //trigger the OOM killer
```
4. For a process, its corresponding uid can be found in its task_struct. (Use the search engine to find out how to get uid from task_struct)
5. To understand how the kernel allocates physical memory, read through the function __alloc_pages_nodemask() in *goldfish/mm/page_alloc.c*.
6. Refer to *goldfish/include/linux/mm.h* and use the search engine to find out how to get RSS for a process.
7. To learn how to kill the chosen process, you can refer to the implementation in existing OOM killer, i.e., the function oom_kill_process() in *goldfish/mm/oom_kill.c*.
8. Any extended ideas can be considered for the bonus! Here we provide some bonus features which are optional. Implementing them can bring you bonus points.
   1) Instead of letting your new OOM killer be triggered by memory allocation events, making it become awaken periodically for some pre-set period T. For this feature, you may have to refer to documents about Linux Daemon.
   2) Maybe it is too strict to set a memory limit that can never be exceeded. You can make some changes to your OOM killer, so that it allows the apps/users temporarily exceed the memory limit. Different time_allow_exceed can be set for different users. Your system call should be changed into:
   ```
   int set_mm_limit(uid_t uid, unsigned long mm_max, unsigned int
   ```

time_allow_exceed);

3) Instead of killing the process with the highest RSS, you can refer to the existing OOM killer and design a new reasonable rule to choose the process to be killed.

Stuffs to be submitted:

1. Compress the source code of the programs into **Prj2+StudentID.tar** file. It contains all *.c, *.h files you have modified in Linux kernel. Use meaningful names for the file so that the contents of the file are obvious. Enclose a README file that lists the files you have submitted along with a one sentence explanation. Call it **Prj2README**.
2. Only internal documentation is needed. Please state clearly the purpose of each program at the start of the program. Add comments to explain your program. (-5 points, if insufficient.)
3. Test runs: Screen captures of the scheduler test to show that your scheduler works.
4. A project report of the project2. In this report, you should:
   1) Explain how your system call works in details.
   2) Explain how the original OOM killer is triggered.
   3) Explain how you design and implement your new OOM killer.
5. Submit your **Prj2+StudentID.tar** file on Canvas.
6. Due date: **June 19, 2020**, submit on-line **before midnight**.
7. Demo slots: The demo date will be announced later. Demo slots will be posted in the WeChat group. Please sign your name in one of the available slots.
8. You are encouraged to present your design of the project optionally. The presentation date will be announced later. Please pay attention to Canvas and the course website.