CS307 Operating Systems

Project 2: Memory Limit for Applications

Fan Wu

Department of Computer Science and Engineering

Shanghai Jiao Tong University

Spring 2020

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.

Enviroment

- Implementation
  - AVD (Android Virtual Devices)
    - SDK version r24.4.1
- Development
  - Linux (64-bits)
    - Ubuntu (recommended)
    - Debian
    - Fedora

What to Submit

- A “tar” file of your DIRECTORY, containing:
  - All *.c, *.h files you have changed in Linux kernel.
  - Any “readme” or “.pdf” files asked for in the project
  - Screen captures of your test
    - If you cannot get your program to work, submit a run of whatever you can get to work as you can get partial credit

- DO NOT SUBMIT your object or executable files, REMOVE them before packing your directory.

How to Submit

- Pack your code in a project directory

```
tar –cvf  Prj2+StudentID.tar  project2
```

- Submit your Prj2+StudentID.tar file on Canvas
  - Ex. Prj2+115XXXXXXXXX.tar

Statement

- Word in blue means it is variable or function.
- Word in red means it is an absolute path in system.
- Word in green means it is the path to kernel files.
- Word in blue means it is an application.

Problem Description

- 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
  2. is not important

Problem (cont.)

- 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 being killed.

Problem (cont.)

- 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:
  - Implement a new system call to set per-user memory limit. (recall that each Android app is represented as a unique user)
  - Study how memory is allocated and how the original OOM killer works.
  - 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.

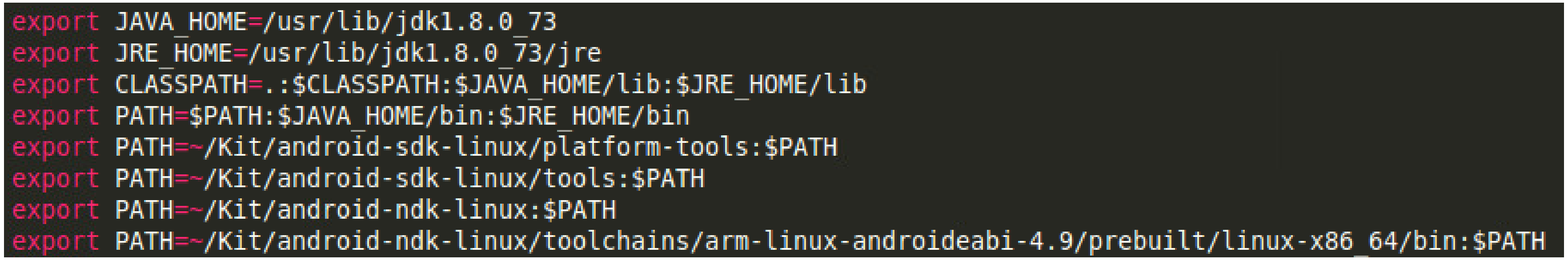
Roadmap

- Step 1:
  - Recompile the Kernel
- Step 2:
  - Add a New System Call
- Step 3:
  - Design and Implement a New OOM Killer

Step 1:
Recompile the Kernel

Compile the Linux Kernel

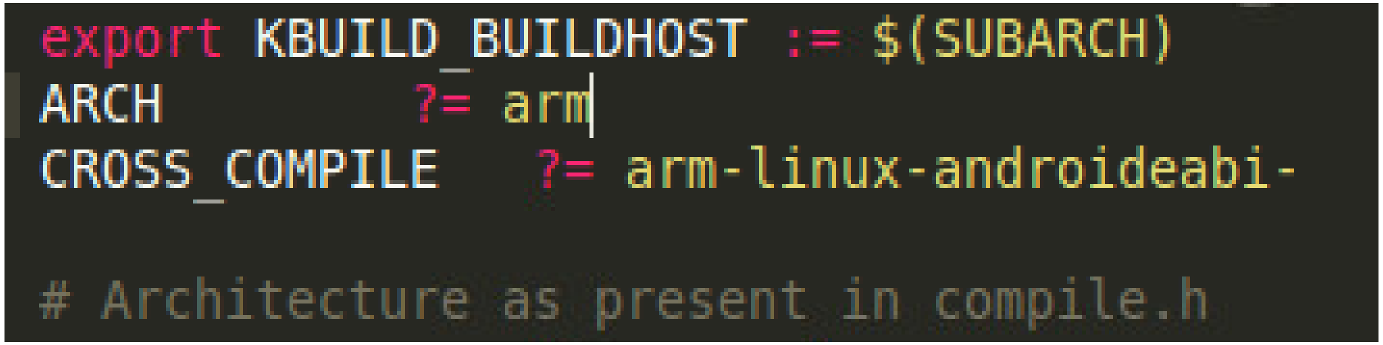
- Make sure that your environment variable is correct.



```
export JAVA_HOME=/usr/lib/jdk1.8.0_73
export JRE_HOME=/usr/lib/jdk1.8.0_73/jre
export CLASSPATH=.:$CLASSPATH:$JAVA_HOME/lib:$JRE_HOME/lib
export PATH=$PATH:$JAVA_HOME/bin:$JRE_HOME/bin
export PATH=~/Kit/android-sdk-linux/platform-tools:$PATH
export PATH=~/Kit/android-sdk-linux/tools:$PATH
export PATH=~/Kit/android-ndk-linux:$PATH
export PATH=~/Kit/android-ndk-linux/toolchains/arm-linux-androideabi-4.9/prebuilt/linux-x86_64/bin:$PATH
```

Compile the Linux Kernel (cont.)

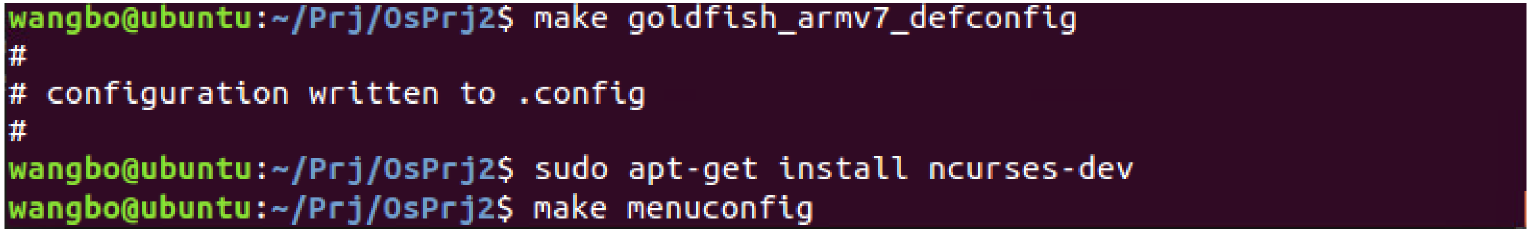
- Modify Makefile in the kernel
  - Change
    - ARCH ?= $(SUBARCH)
    - CROSS_COMPILE ?=
  - To



```
export KBUILD_BUILDHOST := $(SUBARCH)
ARCH            ?= arm
CROSS_COMPILE   ?= arm-linux-androideabi-

# Architecture as present in compile.h
```

Compile the Linux Kernel (cont.)

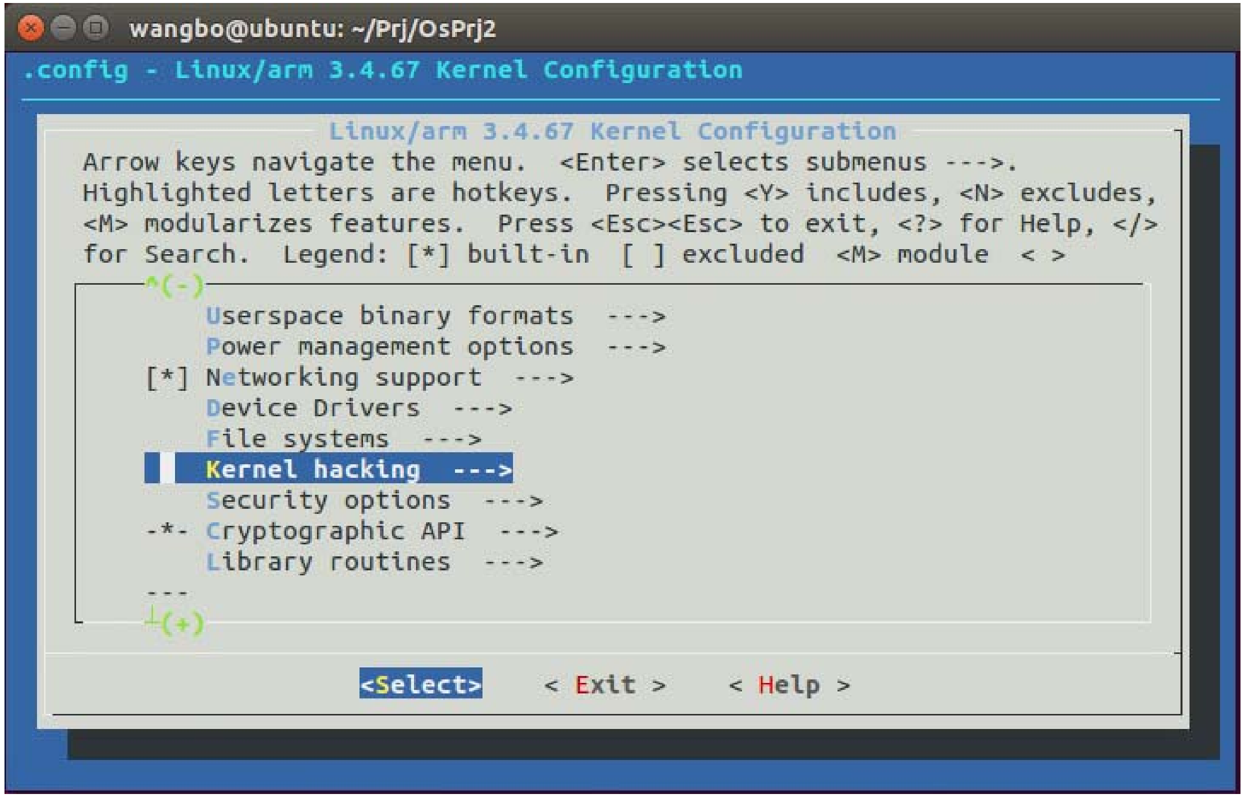
- Execute the following command:



```
wangbo@ubuntu:~/Prj/OsPrj2$ make goldfish_armv7_defconfig
#
# configuration written to .config
#
wangbo@ubuntu:~/Prj/OsPrj2$ sudo apt-get install ncurses-dev
wangbo@ubuntu:~/Prj/OsPrj2$ make menuconfig
```

Compile the Linux Kernel (cont.)

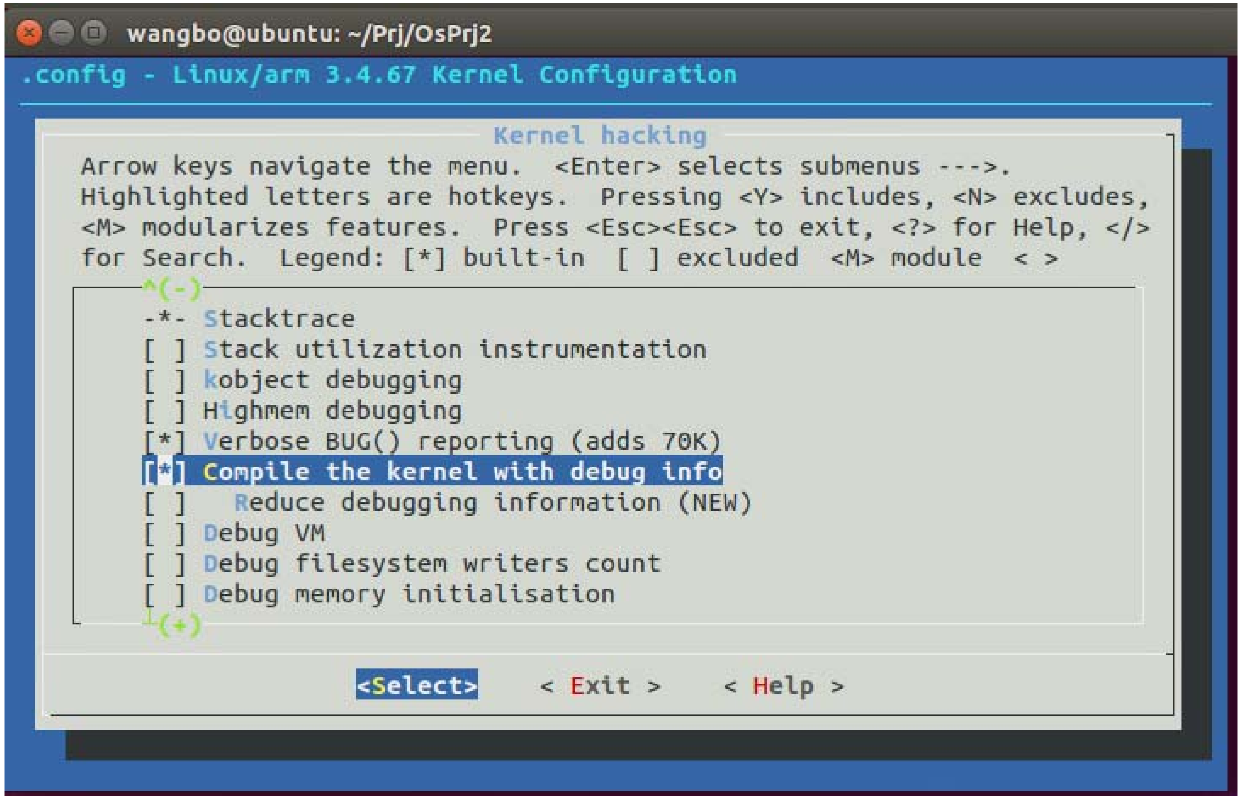
- Then you can see a GUI configuration dialog:



```
.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  --->
      ---
      v(+)

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

Compile the Linux Kernel (cont.)

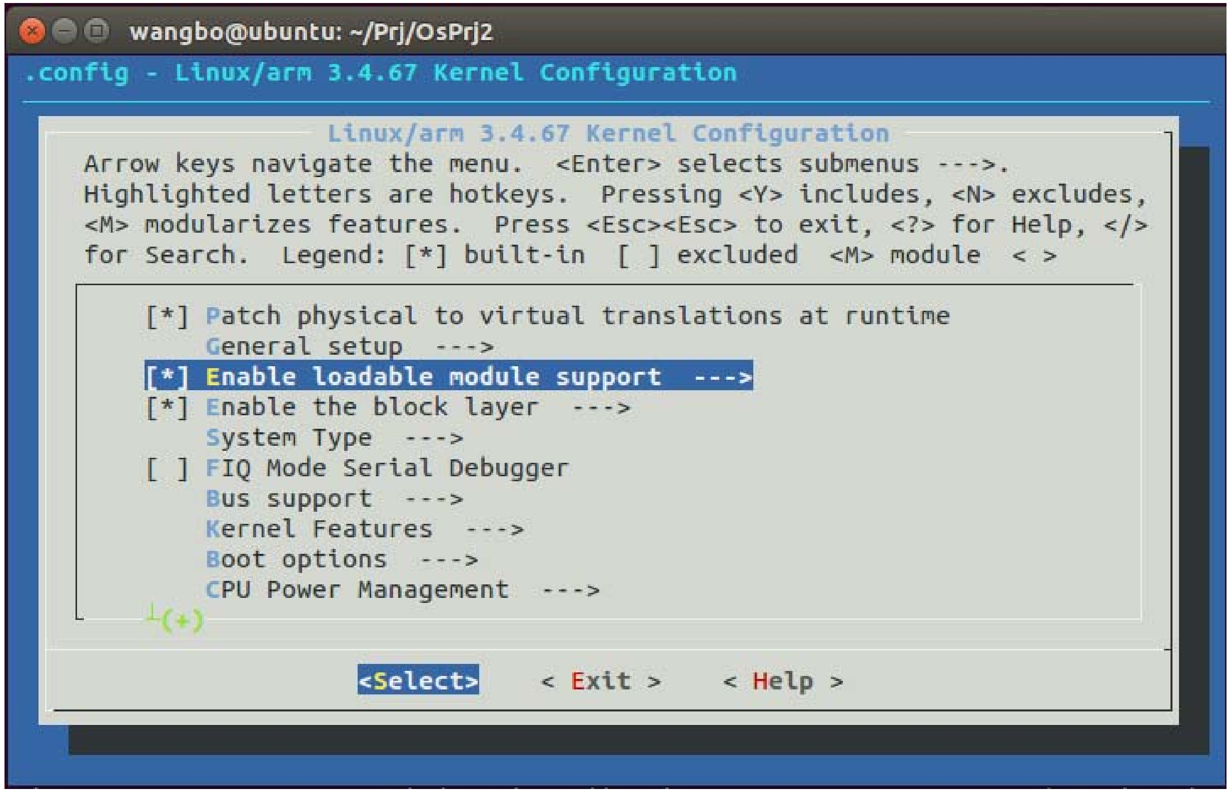
- Open the *Compile the kernel with debug info* in *Kernel hacking*:



```
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
      v(+)

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

Compile the Linux Kernel (cont.)

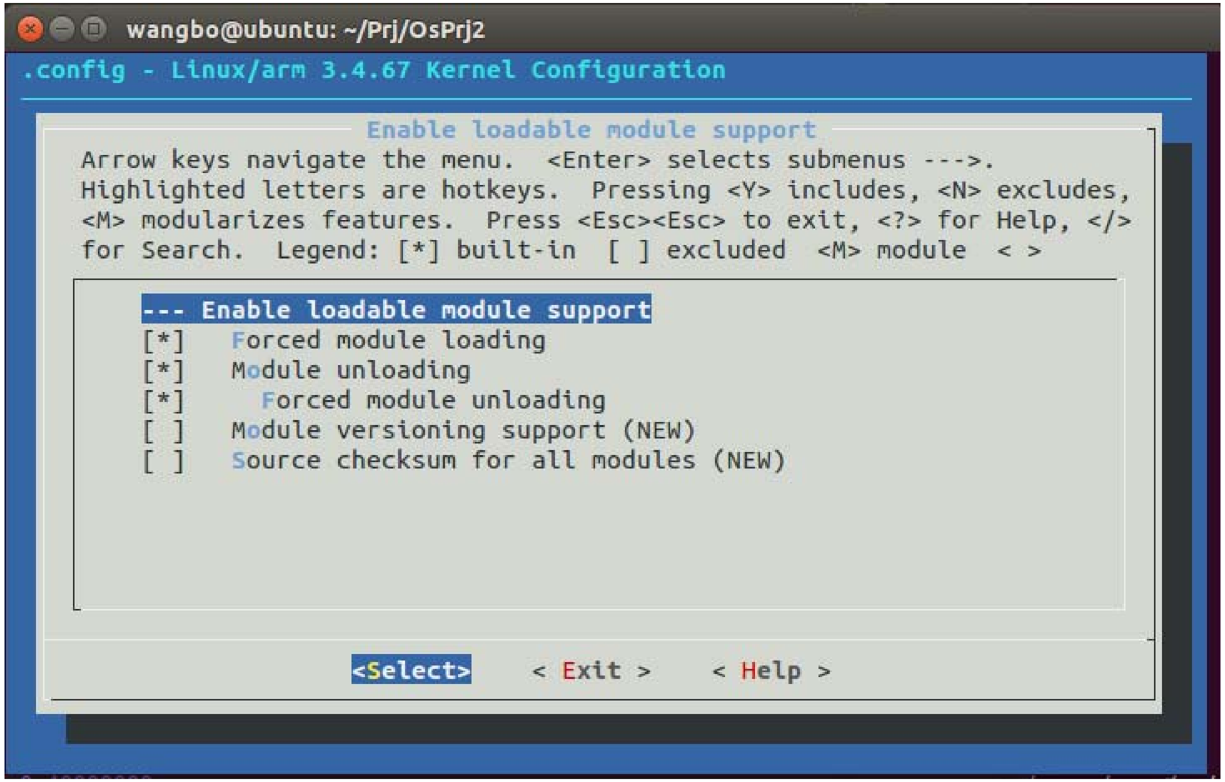
- *Enable loadable module support* with Forced module loading, Module unloading and Forced module unloading in it:



```
.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 >
```

Compile the Linux Kernel (cont.)

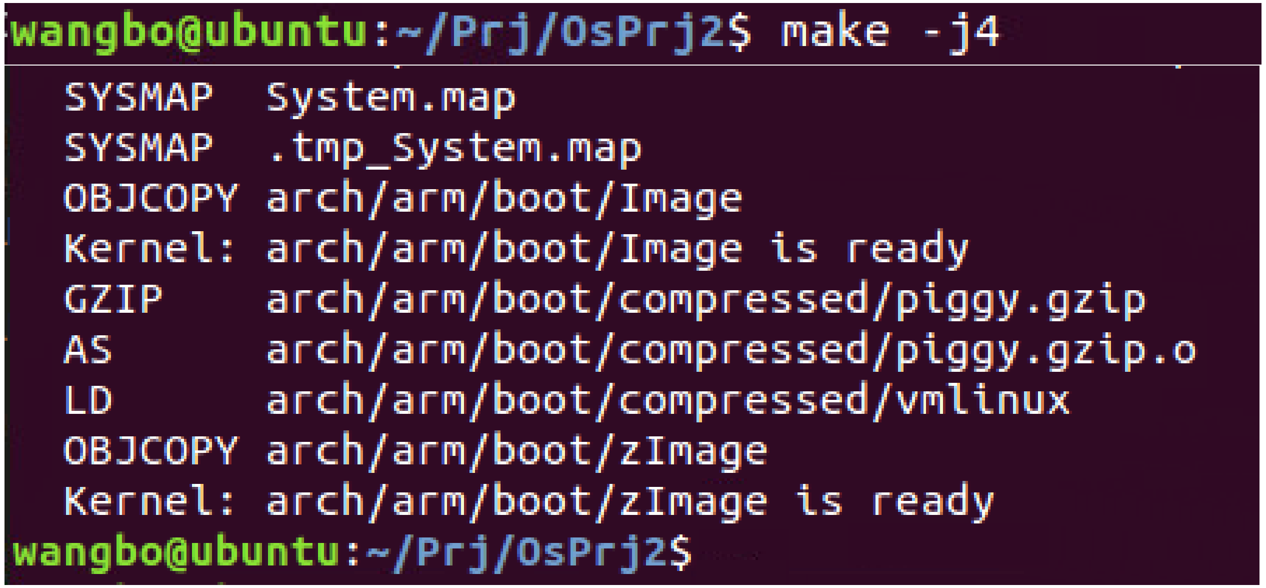
- Enable loadable module support with *Forced module loading*, *Module unloading* and *Forced module unloading* in it:



```
.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 >
```

Compile the Linux Kernel (cont.)

- Compile it



```
wangbo@ubuntu:~/Prj/OsPrj2$ make -j4
  SYSMAP  System.map
  SYSMAP  .tmp_System.map
  OBJCOPY arch/arm/boot/Image
  Kernel: arch/arm/boot/Image is ready
  GZIP    arch/arm/boot/compressed/piggy.gzip
  AS      arch/arm/boot/compressed/piggy.gzip.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
wangbo@ubuntu:~/Prj/OsPrj2$
```

Step 2:
Add a New System Call

Add a New System Call

- You have already learned how to add a system call in Project 1.
- This time, you need to design and add a new system call, which will cooperate with the new OOM killer.
- Signature of the new system call:
  - 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)

User and uid

- The Android system represents each app as a unique user. A user has its own username and uid. For an app, its username is in the form of “ux_ay”, where x and y are two integers.
- The username “ux_ay” is constructed using uid:
  - $x = \lfloor uid \,/\, 100000 \rfloor$
  - $y = (uid \; mod \; 100000) - 10000$
- “ps” command can list all processes with their corresponding username.
- 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 an app 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.

Data Sharing in the Linux Kernel

- 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).

Hints for the Global Variable

- In Project 1, you must have already checked the definition of task_struct (can be found in goldfish/include/linux/sched.h).
- 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).
- To define your global variable my_mm_limits, you can refer to the definition and the initialization of another global variable init_task (can be found in goldfish/arch/arm/kernel/init_task.c).
- To learn how to use a global variable in source files, you can refer to the declaration of init_task (can be found in goldfish/include/linux/sched.h).

Hints for the Global Variable (cont.)

- Since the shared information in our case is quite simple, your struct MMLimits has no need to be as complicated as task_struct.
- For convenience, in this project, you are allowed to set a constant upper bound for the total number of memory limit entries. This enables you to use fixed-size data structures in your struct MMLimits.
- Don’t forget to initialize your global variable my_mm_limits after it is defined.
- The initialization of init_task is a complicated procedure. Luckily, you do not have to totally understand it if you just want to learn how to initialize your global variable my_mm_limits.
- It may be necessary for you to search for and refer to some other documents, so that you can understand how a global variable is defined, declared, initialized, and used in Linux kernel.

Other Requirements for the New System Call

- 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:
  - When set_mm_limit() is invoked to set a memory limit for a certain user, after successfully adding 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);
  - 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.
  - 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.)

Step 3:
Design and Implement a New OOM Killer

Investigate the Existing OOM Killer

- The existing out-of-memory (OOM) killer is triggered when the memory runs out.
- You can learn the trigger mechanism 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
```

- You should add your new OOM killer at a proper place in the file goldfish/mm/page_alloc.c.

Collect Processes Created by a Specific uid

- 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 first traverse all processes and collect the ones that are created by the same user.
- We have already mentioned how to translate username into uid.
- Note that 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)

Count the Allocated Physical Memory

- 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. Definition of mm_struct can be found in goldfish/include/linux/mm_types.h. Refer to the source code and use the search engine to find out how to get the RSS of a process.
- 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.
- To understand how the kernel allocates physical memory, read through the function __alloc_pages_nodemask() in goldfish/mm/page_alloc.c.

Kill the Chosen Process

- The function kill() cannot be used in the kernel, so you should investigate how to kill a process in your new OOM killer.
- You can refer to the implement in the existing OOM killer, i.e., the function oom_kill_process() in goldfish/mm/oom_kill.c.
- If the new OOM killer kill 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);

Hints on the New OOM Killer

- We have already mentioned that you can use your newly added system call set_mm_limt() to set a memory limit for a user (an app). If the limit is smaller than the RSS of the currently running app, a process with highest RSS of this app 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.)
- Actually, the OOM killer is not a process. It is a set of functions that can be triggered and invoked by the kernel.

Hints on the New OOM Killer (cont.)

- 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 memory quota of (≈)100MB for username u0_a70 (i.e., uid 10070), and then fork a new process which requests (≈) 160MB memory by running the following command:

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

PS: 100000000 bytes actually equals to 95.36MB.

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

Bonus

- Any extended ideas can be considered into the bonus!
- Here we provide some bonus features which are optional. Implementing them can bring you bonus points.
  - Instead of letting your new OOM killer be triggered by memory allocation events, make it become awaken periodically for some pre-set period T. For this feature, you may have to refer to documents about *Linux Daemon*.
  - Maybe it is too strict to set a memory limit that can never be exceeded. Do 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);
  - 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 a process which should be killed.

Report

- Explain how your system call works in detail.
- Explain how the original OOM killer is triggered.
- Explain how you design and implement your new OOM killer.

Deadline

Mid-night, June 19, 2020

Demo & Presentation

- Demo:
  - 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.
- Presentation:
  - You are encouraged to present your design of the project optionally. The presentation date will be announced later.

For Help?

- Teaching Assistant
  - Renjie Gu
    - Email: grj165@sjtu.edu.cn
  - Hongtao Lv
    - Email: lvhongtao@sjtu.edu.cn
- Some useful website
  - https://www.bing.com/
  - https://www.csdn.net/
  - https://stackoverflow.com/
  - https://developer.android.com/

For further questions regarding the project, you can:

- Raise the questions now
- Post the questions in the wechat group
- Send the questions to a TA through email

  Renjie Gu (grj165@sjtu.edu.cn)
  Hongtao Lv (lvhongtao@sjtu.edu.cn)

Thank you！