

Operating System Project 2

Android Kernel Hacking

518030910188
Yimin Zhao
doctormin@sjtu.edu.cn

July 5, 2020

Contents

1	Introduction	3
1.1	Project Results	3
1.2	Road Map	4
1.3	Modification to Kernel	4
2	System Call	5
2.1	Implementation	5
2.1.1	Prototype	5
2.1.2	Declare Syscall	5
2.1.3	Define Syscall	5
2.2	Global Variable	6
2.2.1	Struct	6
2.2.2	Initialize Global Variable	6
2.2.3	Use Global Variable	7
2.3	Mutex	7
2.4	Timer	7
2.4.1	Define Timer	7
2.4.2	Initialize Timer	7
2.4.3	Timing Function	7
3	Android OOM Killer	9
3.1	Trigger Mechanism	9
3.2	Details of Killing Processes	10
4	Yimin's OOM Killer	10
4.1	Overview	10
4.2	Event Tracing Module	11
4.3	Dynamic Timer	12
4.4	Allow Exceeding Limits for a Pre-set Time	14
4.5	Details of Killing Processes	14
5	Makefile and Tricks	15
5.1	Modify Kernel Makefile	15
5.2	Makefile for Auto-complete	16
5.2.1	Hacking List	16
5.2.2	Test on One-click	17
6	Conclusion	18
7	Feelings	18
8	Reference and Acknowledgement	18

1 Introduction

1.1 Project Results

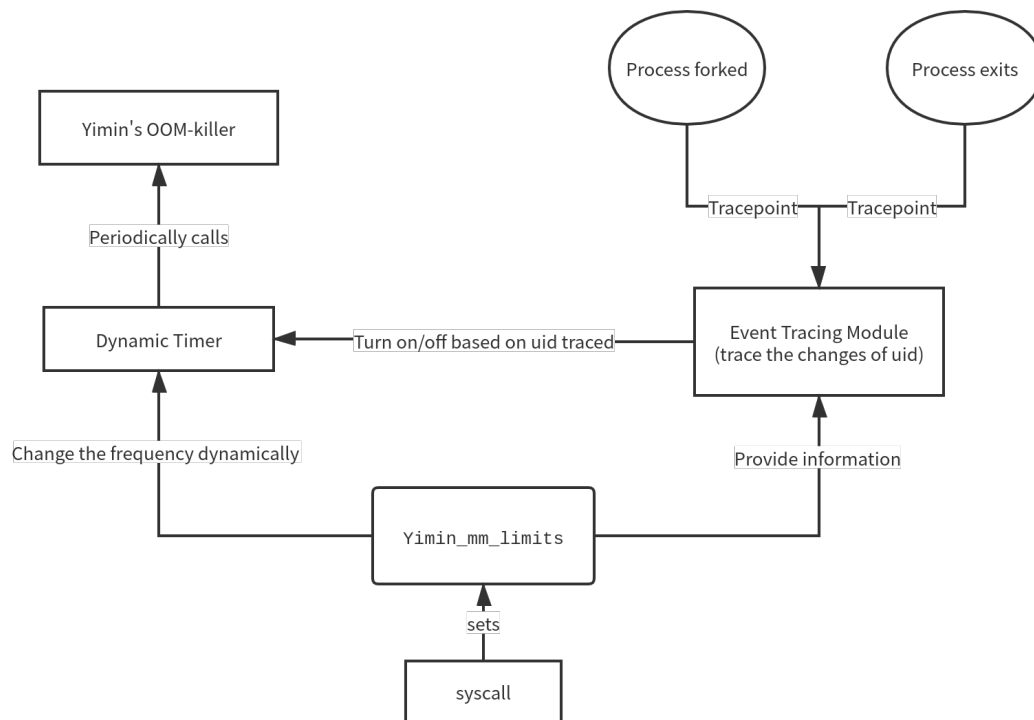


Figure 1: Overview

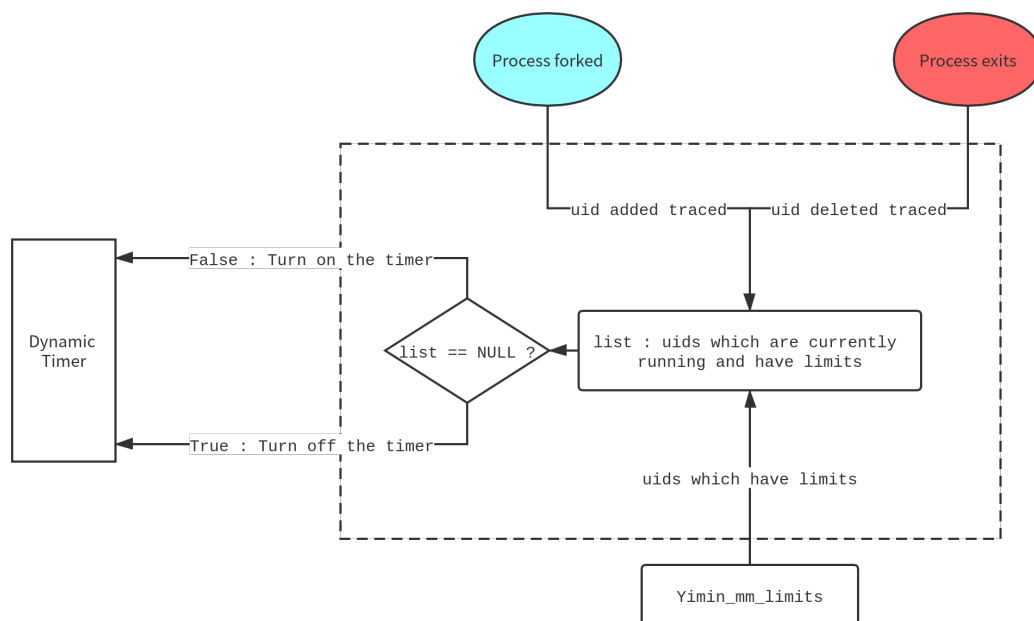


Figure 2: Details of the Event Tracing Module

1.2 Road Map

- Implement a custom system call
- Define a global variable to store UID and RSS information
- Implement OOM-killer triggered by `__alloc_pages_nodemask` → [baseline version](#)
- (Bonus) Implement OOM-killer triggered by timer
- (Bonus) Allow users to exceed memory limits for a pre-set time period
- (New Feature) Make timer dynamic to further decrease overhead
- (New Feature) Support "passive detection" of changes of uid to further decrease overhead of the OOM-killer
- (New Feature) Support using "make menuconfig" to add Yimin's oom-killer feature in GUI

1.3 Modification to Kernel

Please refer to [5.2.1](#).

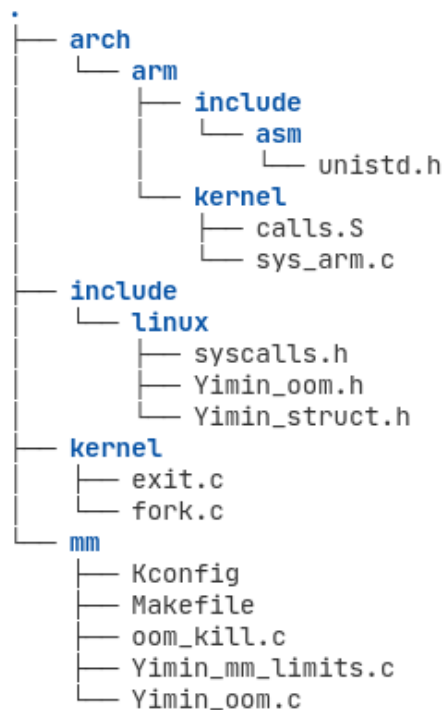


Figure 3: Modified / Added Files (the root directory is goldfish/)

2 System Call

2.1 Implementation

2.1.1 Prototype

We need a system call in this prototype

```
1 asmlinkage long sys_set_mm_limits(uid_t uid, unsigned long mm_max, unsigned long  
    ↪ time_allow_exceed);
```

- uid is the ID of a user on whom we want to set memory limits.
- mm_max is the limit we set.
- time_allow_exceed is the amount of time (in nanosecond) the user is allowed to exceed the memory limit

2.1.2 Declare Syscall

First we need to declare it in **unistd.h** and **calls.S**. I choose the syscall number to be 59 because that is shown to be available in **call.S**. After testing, at least in my ubuntu 18.04 machine, it works well.

goldfish/arch/arm/include/asm/unistd.h #line 87

```
1 #define __NR_set_mm_limit    (__NR_SYSCALL_BASE+ 59)
```

goldfish/arch/arm/kernel/calls.S #line 71

```
1 CALL(sys_set_mm_list)
```

2.1.3 Define Syscall


The implementation of this system call is in **goldfish/arch/arm/kernel/sys_arm.c**. You can refer to the codes start from #line 140 in that file. Here I'll show what the system call does at the logical level.

Prerequisites (Details covered in the following three sub-sections)

- A **global variable** is needed to store these limits we set on users
- A **mutex** is needed during reading and writing the global variable
- A **timer** is needed to be initialized in the syscall.

Logical Steps

1. Initialize the timer for only **once** (use a static flag to skip this step when the syscall is called more than once). The details of the initialization of the timer is shown in [2.4](#).
2. Use a for loop to traverse every entry of the global variable (**struct Yimin_struct Yimin_mm_limits** ↪)
 - (a) If the entry for the target uid already exists, update the newly set memory limit. And set the flag "**updated**" to be true.

3. Use a for loop to traverse every entry of the global varibal (**struct** Yimin_struct Yimin_mm_limits ) again.
 - (a) Print every entry which is valid
 - (b) If a invaild entry is met and the flag "updated" is false
 - i. Update that entry to be the newly set
 - ii. Print this renewed entry
 - iii. Set the flag "updated" to be true.
4. If the flag "updated" is still false, then it means that the upper bound of the number of entries has been reached.

2.2 Global Variable

2.2.1 Struct

We need a global variable to save the information of memory limits for users set by our system call because the oom killer must refer to this information every time it is evoked. So before implementing the syscall, we need to define and initialize a global variable first. The data type of this global variable is a very simple struct defined as follow.

goldfish/include/linux/Yimin_struct.h

```

1 #define e_enum 200
2 struct Yimin_struct {
3     /**
4      * uid           = Yimin_mm_limits.mm_entries[i][0];
5      * memory limit   = Yimin_mm_limits.mm_entries[i][1];
6      * valid bit      = Yimin_mm_limits.mm_entries[i][2];
7      * time_allow_exceed = Yimin_mm_limits.mm_entries[i][3];
8      */
9     unsigned long mm_entries[e_enum][4];
10 };

```

The `e_enum` is a macro used globally to define the maximum number of entries that can be saved. For example, if `e_enum` is set to be 200, then we can set memory limits for up to 200 different users. And if we want to increase the maximum number of entries, we just need to modify `e_enum` in this header file.

2.2.2 Initialize Global Variable

The global variable should be statically initialized in a c file. So I creat a source file in the kernel to do this. And to add source files in kernel, we need to modify the **Makefile** in this directory (i.e. /mm/Makefile). The details for **Makefile** are show in section [5.1](#)

goldfish/mm/Yimin_mm_limits.c

```

1 struct Yimin_struct Yimin_mm_limits = {
2     .mm_entries = {{0}}
3 };

```

2.2.3 Use Global Variable

We need to get access to this global variable in the source file where the syscall is defined. So this is needed in the code of the syscall definition:

```
1 extern struct Yimin_struct Yimin_mm_limits;
```

2.3 Mutex

In theory, global variables can be accessed by all parts of the kernel simultaneously. So, since we need to access a global variable in the syscall, a mutex is required. Because we will also read and write this global variable (`Yimin_mm_limits`) in oom killer, the most convenient way is to also define a global mutex — which I name `Yimin_mutex`.

It should be noted, however, [statically defining a mutex should be done in a c file rather than in a header file](#). Because if the header file is included in several translation units, it will raise a duplicate definition error in the linking stage. In the kernel, there is a macro provided to statically define a mutex — `DEFINE_MUTEX(Yimin_mutex)`.

2.4 Timer

2.4.1 Define Timer

In my code, I define a timer which I call `Yimin_timer` in the source file `Yimin_oom.c`.

`goldfish/mm/Yimin_oom.c`

```
1 struct timer_list Yimin_timer;
```

2.4.2 Initialize Timer

Timer can be initialized by a macro — `init_timer`. But this macro is essentially a function as shown in `timer.h`.

`goldfish/include/linux/timer.h`

```
1 #define init_timer(timer)\
2     init_timer_key((timer), NULL, NULL)
3 -----
4 void init_timer_key(struct timer_list *timer,
5                     const char *name,
6                     struct lock_class_key *key);
```

So this is not a static initialization which means that we need to use `init_timer` inside our syscall function — `init_timer(&Yimin_timer);`. And of course we need to declare this timer using "`extern ↪ struct timer_list Yimin_timer;`" in the syscall function because it's defined in other source file.

2.4.3 Timing Function

A timer is essentially a countdown device. After the timer starts, once the countdown is over, a specified function is triggered. So the timer can realize the function of timing triggering function which will be used for our oom-killer.

Define the Triggered Function

Struct `timer_list` has a element named `function` which is function pointer. So we just need to assign the name of our function (i.e. the oom-killer) to this element.

```
1 /**
2  * It should be noted that to be a function which can be
3  * triggered by the timer, the function must be like:
4  * void xxx(unsigned long)
5  */
6 void Yimin_oom_killer(unsigned long);
7 -----
8 Yimin_timer.function = Yimin_oom_killer;
```

Define the Countdown Time

Struct `timer_list` has a element named `expires` which stand for the time point when the timer will be stopped. Its unit is the number of system interrupts whose frequency is defined by a macro — `HZ`. It's 100 in our case, which means that the the accuracy of our timer is $\frac{1}{100} = 0.01s$. The countdown number can be defined as follow:

```
1 #define KILLER_TIMEOUT 3 //The unit is 0.01s
2 -----
3 Yimin_timer.expires = jiffies + KILLER_TIMEOUT;
```

It should be noted that `jiffies` is the number of "ticks" (system interrupts) since the system booted. So `expires = jiffies + KILLER_TIMEOUT` means that the timer will countdown `KILLER_TIMEOUT` times of system interrupt before triggering the function pointed by `Yimin_timer.function`.

Add Timer to System Timer List

Timer initialization only needs to be done once. But for the periodic countdown. We need to `add_timer(&Yimin_timer)` and `del_timer(&Yimin_timer)` periodically, too. When countdown finish, the function is triggered. We need to delete timer from the system timer list and add it again. In this way, the timer will start the countdown for another round.

The triggered function

```
1 void Yimin_oom_killer(unsigned long data)
2 {
3     //The real oom-killer function
4     __Yimin_oom_killer();
5
6     //reset Yimin_timer -> start another round of countdown
7     del_timer(&Yimin_timer);
8     Yimin_timer.function = Yimin_oom_killer;
9     //BIAS is for "dynamic timer" which will be covered in this report later
10    Yimin_timer.expires = jiffies + KILLER_TIMEOUT + BIAS;
11    add_timer(&Yimin_timer);
12 }
```


3 Android OOM Killer

3.1 Trigger Mechanism

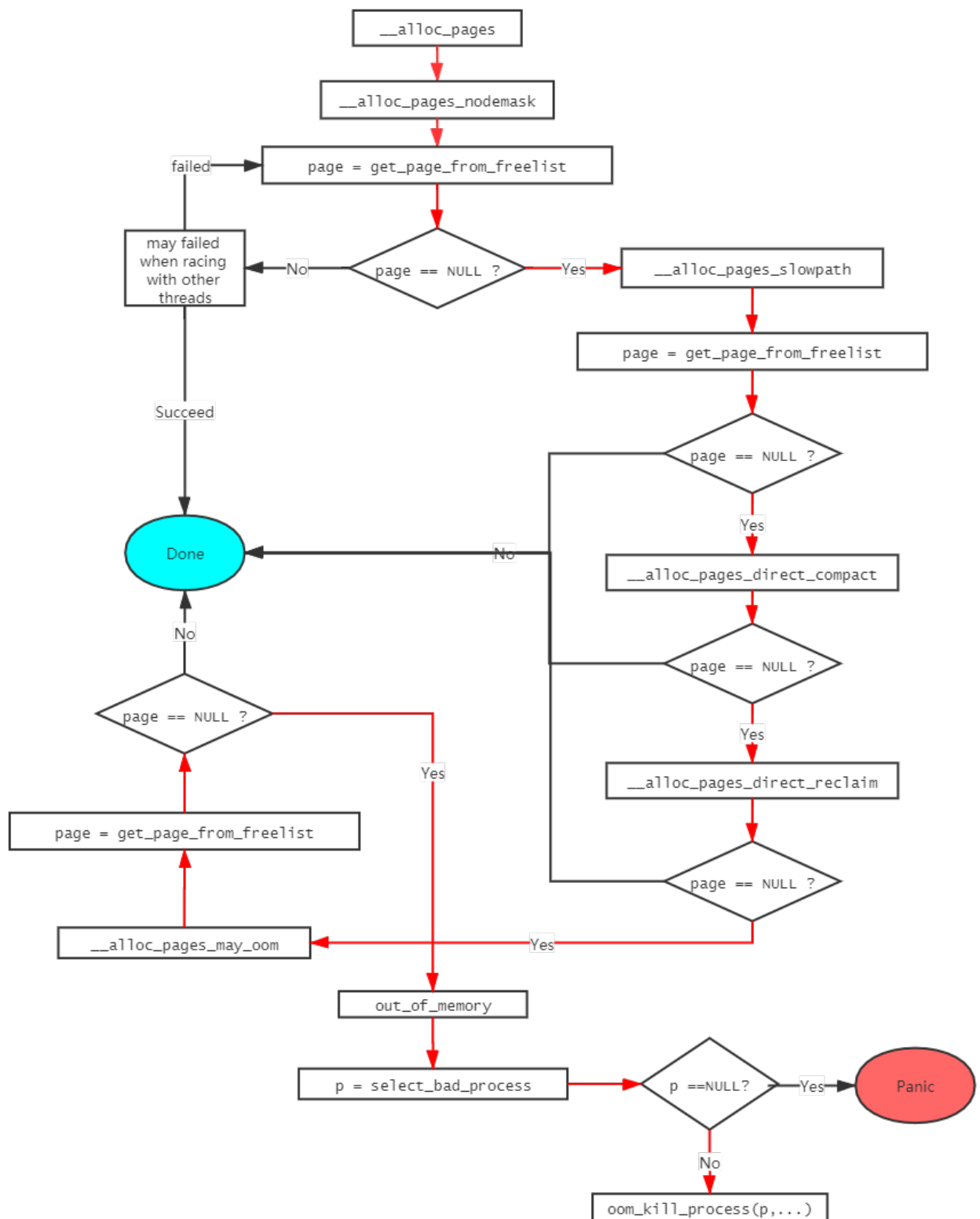


Figure 4: Trigger Mechanism of Android OOM killer

In general, it is possible to trigger the OOM-Killer every time the memory is allocating a page, and there are actually many steps of verification before the actual killing process is carried out. Due to limited time, my OOM-killer doesn't have so many steps of verification before killing, instead focusing on reducing steps to optimize performance.

3.2 Details of Killing Processes

As shown in the figure above, the "killer" function is `oom_kill_process`. This function is responsible for killing a process, but its implementation logic is actually not that simple. The direct way of killing a process is using:

```
1 do_send_sig_info(SIGKILL, SEND_SIG_FORCED< victim, true);  
2 //victim is the killed process
```

But rather than just killing the process, the android OOM killer actually does a lot before that step.

- Step 1 - If the task is already exiting, nothing brutal need to be done.
- Step 2 - Check its children. If any one of them has higher `oom_badness()` score, this child will be killed instead of the parent (`victim := this child`).
- Step 3 - Kill all user processes which shares the `victim` process's mm struct in other thread groups.
- Step 4 - Kill the `victim`.

4 Yimin's OOM Killer

4.1 Overview

The simplest idea is to use a trigger mechanism similar to the Android OOM-Killer which is triggering the OOM-killer in `__alloc_pages_nodemask`. The implementation method is therefore very simple, simply calling the OOM-killer function in the appropriate location in `__alloc_pages_nodemask`.

However, the biggest fallback of the trivial version is the large overhead caused by triggering the OOM-killer everytime a page is allocated. So, to reduce overhead, it is much better to **trigger the OOM-killer regularly**.

Implementing a daemon can meet this requirement. But the idea of daemon is not based on the baseline version. In fact, I would like to add something to the baseline version to achieve wanted functionality instead of just starting over. So, I choose to use `timer`. Basic information about the timer has been covered in section 2.4. The logic steps of using a timer is as follow:

- Define and initialize a timer in our syscall.
- Specify the timer's trigger function as our "OOM-killer wrapper" function.
- Inside the "OOM-killer wrapper":
 - Trigger the "real killer" function.
 - Reset timer to make it ready for the next round of triggering.

Moreover, even with triggering the OOM-killer regularly, we still have to suffer a certain overhead. Can we decrease the overhead further more? The answer is yes. First, let's make it clear that the intermediate purpose of an OOM-killer is to find a user who has exceeded the memory limit in time. So when the risk is low, the OOM-killer need not to be triggered as frequently as when the risk is high. At the same time, the "risk" mentioned above can be quantified in terms of the amount of

memory left and the frequency of the triggering can also be quantified by the countdown number of the timer. Therefore, it's actually intuitive to use a "dynamic timer" to change the frequency due to the "risk".

Furthermore, we can note that the timer's repetitive effort is meaningless if the UIDs we have set memory limits on are not currently running. So it will be great if we can do the following things:

- When a UID that we have set limits on begin running in the system, we start the timer(i.e. start triggering the OOM-killer over and over)
- When all UIDs that we have set limits on are not running in the system, we stop the timer(i.e. stop triggering the OOM-killer)

Therefore, I set two "tracepoint" in `do_fork` and `do_exit`. Almost every process starts with `do_fork` and terminates with `do_exit`. In this way, when a new process starts or an old process exits, I can get informed. And this detection is "passive" and takes almost no overhead!

4.2 Event Tracing Module

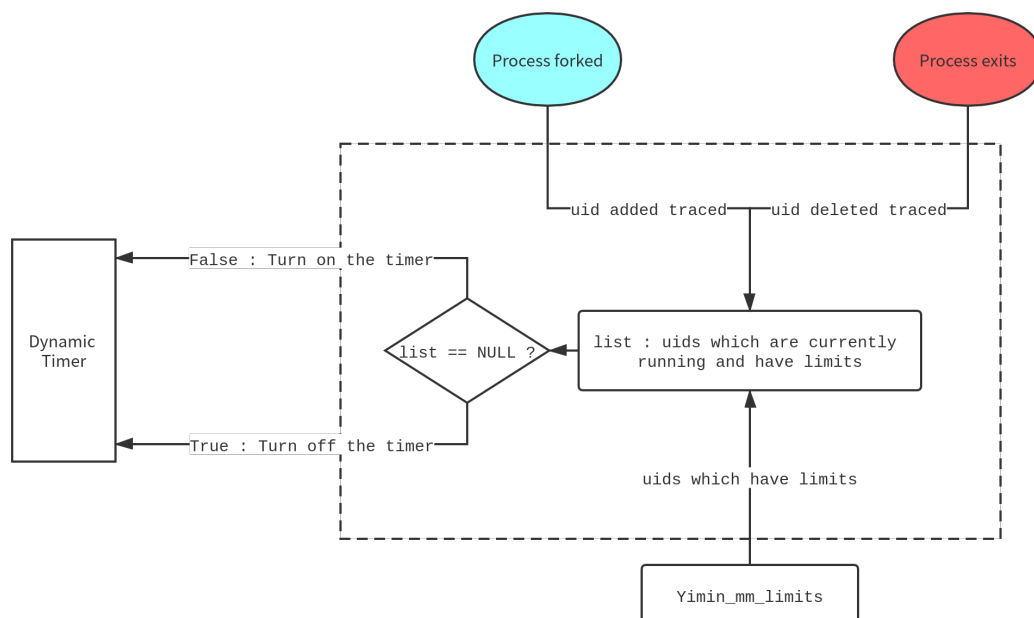


Figure 5: Details of the Event Tracing Module

Using the "tracepoint" technique, we need to write a module acting as a notifier. "Tracepoint" allows us to insert a "point" into somewhere of the kernel. Everytime the "point" is run through, our "probe" function (defined in the module) will be triggered in which we can do a lot of things.

I use the "tracepoint" to trace two events — `do_fork()` and `do_exit()`. In this way, we can achieve **passive detection of processes (also for UID) creation and termination**. First we need to insert two "points" into these two functions, here is an example:

goldfish/kernel/fork.c

```

1 //inside 'do_fork' function:
2 /**
3  * When this points is reached(every time do_fork() is called)
4  * it will trigger a probe function : Yimin_eventDoFork(uid_t uid)

```

```

5  * and pass 'p->cred->uid' as the parameter to our probe function
6  * The probe function is defined in our 'trace_module'
7  */
8  trace_Yimin_eventDoFork(p->cred->uid);

```

As for the probe function. The aim of it is to record the changes of UID which is passed to it as an parameter by the "point". The probe function will maintain a list ([watch_table](#)) recording all UIDs which are currently running and have memory limits set by our syscall.

- When [do_fork\(\)](#) event happens, the UID of the newly created process will be passed to the probe function:
 - Check if the UID of the newly created process need to be added to [watch_table](#)
 - * If yes, add it
 - * Else, do nothing
 - Check if the [watch_table](#) is empty:
 - * If it's not empty and the timer is currently not running, then start the timer by calling [add_timer\(&Yimin_timer\)](#).
- When [do_exit\(\)](#) event happens, the UID of the newly ended process will be passed to the probe function.
 - Check if the UID is already in the [watch_table](#)
 - * If yes, delete it from the [watch_table](#)
 - Check if the [watch_table](#) becomes NULL
 - * If yes and the timer is currently running, then stop the timer by calling [del_timer\(&Yimin_timer\)](#)

The probe function is defined in `trace_module.c`. To decrease overhead, this function will scan the whole task list in the system only once when the module is installed. After this traversal, the function has the UIDs of all the currently running processes. Therefore, **there is no need to traverse the task list again, because every change of the UIDs (create process and end process) will be detected by the probe function, and the current information only needs to be modified by one item a time.**

4.3 Dynamic Timer

The countdown time of the timer will be dynamically adjusted due to the "risk" of exceeding the memory limits. So we need to define a way to compute "risk" first.

Here I take a very straightforward and simple approach. I define a variable "safety":

$$safety = \min\{(memory\ limits\ for\ uid - memory\ already\ taken\ by\ uid)\ among\ all\ uids\}$$

"safety" is computed every time the OOM-killer is called. We can see that the smaller the "safety", the more likely will the memory limits be exceeded. So when "safety" is small, the countdown time of the timer should be small as well.

To change the countdown time of the timer, I define another variable "BIAS":

$$countdown = 0.02s + BIAS \times 0.01s$$

In the code, this change is implemented in this way:

```

1 #define KILLER_TIMEOUT 2
2 /**
3  * Yimin_timer triggers the OOM-killer regularly
4  * and BIAS is computed everytime the OOM-killer run.
5  */
6 Yimin_timer.expires = jiffies + KILLER_TIMEOUT + BIAS

```

Now, what left to do is to establish a functional relationship between *BIAS* and *safety*. After some tests, I find that it will take about 1.2 second to allocate 256MB in avd running on ubuntu machine. If we set the upper-bound of *BIAS* to be 20 (i.e. 0.2s), then in one countdown round, the system can allocate $\frac{0.2+0.02}{1.2} \times 256MB \approx 49213166B$. If *BIAS* = 0, then in one countdown round, the system can allocate $\frac{0.02}{1.2} \times 256MB \approx 4473924B$.

So it should be reasonable to set the following relationship:

```

1 /*
2  * determine 'BIAS' based on 'safety'
3  * 0 <= safety <= mm_max
4  * 0 <= BIAS <= BIAS_UPP
5  */
6 if(safety >= 49213166 * 3)
7     BIAS = BIAS_UPP;
8 else
9 {
10     if(safety <= 4473924 * 3)
11         BIAS = 0;
12     else
13         /*
14          * float/double is not supported well, so have to change
15          * BIAS = safety / ((49213166-4473924)*3) * BIAS_UPP
16          * to the blow...
17          */
18         BIAS = safety / (44739 * 3) * BIAS_UPP / 1000;
19 }

```

The result turns out good, we can see a demo:

safety: Memory allowance

BIAS: The bias added to the timer's countdown number (countdown number without bias is 2 in this demo)

```

uid=10070,      mm_max=100000000B,      time_allow_exceed=0ns
safety = 8832640, BIAS = 3
safety = 8595072, BIAS = 3
safety = 8361600, BIAS = 3
safety = 8128128, BIAS = 3
safety = 3012224, BIAS = 0
safety = 1812096, BIAS = 0
safety = 0, BIAS = 0
Yimin's oom killer : uid = 10070, uRSS = 13082624B, mm_max = 100000000B, pid = 1094, pRSS = 3424256B
safety = 0, BIAS = 0
Yimin's oom killer : uid = 10070, uRSS = 15151104B, mm_max = 100000000B, pid = 1092, pRSS = 4014080B
safety = 0, BIAS = 0
Yimin's oom killer : uid = 10070, uRSS = 12677120B, mm_max = 100000000B, pid = 1093, pRSS = 4304896B
safety = 0, BIAS = 0
safety = 308864, BIAS = 0
safety = 4748928, BIAS = 2
safety = 9041536, BIAS = 4

```

Figure 6: Demo of Dynamic Timer

4.4 Allow Exceeding Limits for a Pre-set Time

This feature allows apps/users temporarily exceed the memory limit. The prototype of the system call needs to be changed, as already shown in 2.1.1.

I implement this feature by introducing another data struction into the OOM-killer. This data struction will save the information of "UID" and "the timestamp when then UID starts exceeding the limit". The logic steps are shown below:

Every time the OOM-killer is called

- The data struction will be refreshed(delete entries that are not currently running)
- traversing the memory limits set by syscall
 - If rss exceeds, refer to the data structure for the UID
 - * If not inside, then add it to the data struction and set the `time_begin_exceed` to be the current system time
 - * If inside, then get the current system time and compute the time interval($interval = current_time - time_begin_exceed$). If the time interval is longer than `time_allow_exceed` \hookrightarrow , then kill one process of this user.
 - Else, refer to the data structure for the UID
 - * If inside, delete it from the data structure
 - * Else, do nothing

4.5 Details of Killing Processes

When the timer countdown is over, it will trigger `Yimin_oom_killer`, which will call `__Yimin_oom_killer` \hookrightarrow and then reset the timer. Inside `__Yimin_oom_killer`, the UIDs which have exceeded the memory limits will be passed to `__Yimin_kill`.

`__Yimin_kill` will choose the process which has the largest UID and meanwhile is not vital to kill.

Detect Vital Process (Unkillable)

```
1 if (is_global_init(killed_process) || (killed_process->flags & PF_KTHREAD))
2 {
3     // don't kill
4 }
```

In the kernel, the following code can kill a process:

```
1 do_send_sig_info(SIGKILL, SEND_SIG_FORCED, killed_process, true);
```

5 Makefile and Tricks

5.1 Modify Kernel Makefile

The Linux Kernel takes the "layered make" approach which is commonly used in large projects. A top-level Makefile in **goldfish/** will trigger other Makefiles in children directories. Because we add source files to the kernel, we need to modify some Makefiles to add our new translation units into linking process. To comply with Linux's "layered make" logic, I will modify the Makefile in **goldfish/mm/** because all the source files added are in that directory.

Consider adding a **compilation option** (i.e. supporting "make menuconfig") to our new OOM features, I decide to add a configuration variable for **Yimin_oom.c**. So both **mm/Kconfig** and **mm/Makefile** need to be modified. The following code need to be added to the end of the two files.

goldfish/mm/Kconfig

```
1 config YIMIN_SETTING
2     bool "Yimin's oom killer"
3     default y
4     help
5         This is a oom killer implemented by Yimin Zhao in Prj2 of CS356-2
```

goldfish/mm/Makefile

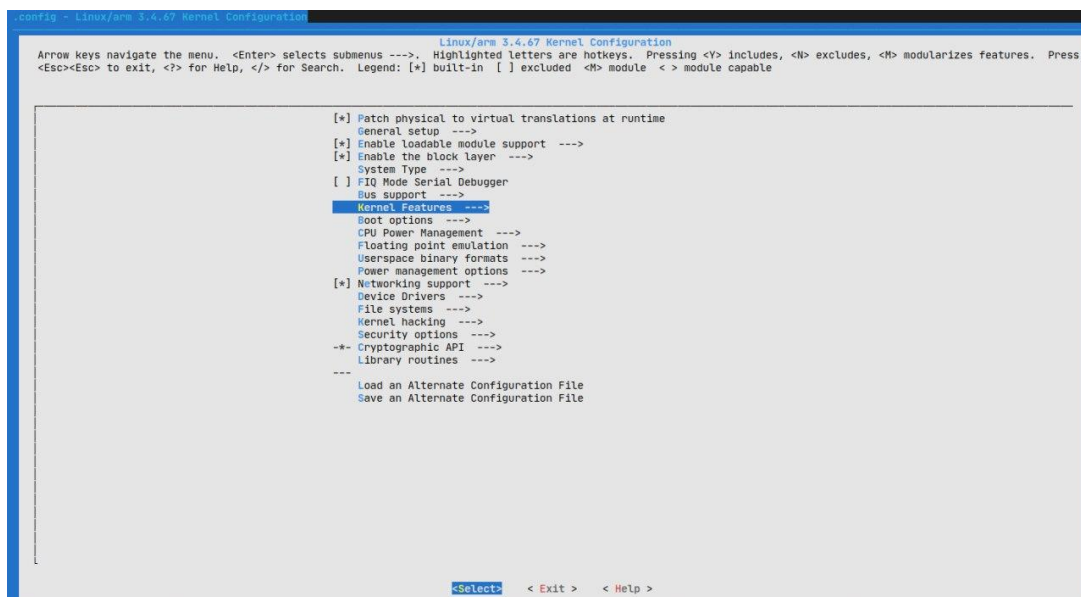
```
1 obj-$(CONFIG_YIMIN_SETTING) += Yimin_mm_limits.o
2 obj-$(CONFIG_YIMIN_SETTING) += Yimin_oom.o
```

Demo of "make menuconfig"

Step 1 - type "make menuconfig" in the terminal

```
→ OperatingSystemPrj2 git:(Allow_time_exceed) X make menuconfig|
```

Step 2 - choose "Kernel Features"



Step 3 - Yimin's oom killer is added as an option

```
[*] Tickless System (Dynamic Ticks)
[*] High Resolution Timer Support
    Memory split (3G/1G user/kernel split) --->
    Preemption Model (Preemptible Kernel (Low-Latency Desktop)) --->
[ ] Compile the kernel in Thumb-2 mode (EXPERIMENTAL)
[*] Use the ARM EABI to compile the kernel
[ ] Allow old ABI binaries to run with this kernel (EXPERIMENTAL)
[*] High Memory Support
[ ] Allocate 2nd-level pagetables from highmem
    Memory model (Flat Memory) --->
[ ] Allow for memory compaction
[ ] Enable KSM for page merging
(4096) Low address space to protect from user allocation
[ ] Enable cleancache driver to cache clean pages if tmem is present
[*] Yimin's oom killer
[ ] Use kernel mem{cpy,set}() for {copy_to,clear}_user() (EXPERIMENTAL)
[ ] Enable seccomp to safely compute untrusted bytecode
[ ] Enable -fstack-protector buffer overflow detection (EXPERIMENTAL)
[ ] Provide old way to pass kernel parameters
[ ] Force flush the console on restart
```

5.2 Makefile for Auto-complete

To make it easier to manipulate AVD and compile the kernel, I write a makefile that **makes everything one-click**.

5.2.1 Hacking List

The kernel has tens of thousands of files, and if you modify them directly in the kernel directory, locating them can be a huge headache. So I take advantage of the shell's **rsync** directive, which allows me to manipulate all the kernel files I modify or add in a **separate folder**. I call this directory "hacking/", and this is the tree structure of this directory.

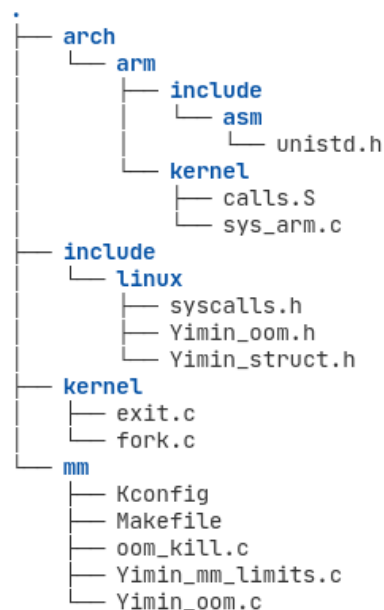


Figure 7: Structure of "hacking/"

The following instructions in Makefile can **update the files in the kernel according to the hacking list in one-click**.

OperatingSystemPrj2/Makefile

```
1  HACKING_LIST=      \
2  arch/arm/include/asm/unistd.h  \
3  arch/arm/kernel/calls.S      \
4  arch/arm/kernel/sys_arm.c    \
5  include/linux/syscalls.h      \
6  include/linux/Yimin_oom.h     \
7  include/linux/Yimin_struct.h  \
8  mm/Kconfig                  \
9  mm/Makefile                  \
10 mm/page_alloc.c              \
11 mm/Yimin_mm_limits.c         \
12 mm/Yimin_oom.c               \
13
14 rsync:
15   for file in $(HACKING_LIST); do \
16     rsync -u hacking/$$file kernel/goldfish/$$file; \
17   done
```

Demo of one-click → update files in the kernel

```
→ OperatingSystemPrj2 git:(Allow_time_exceed) ✖ make rsync
for file in arch/arm/include/asm/unistd.h arch/arm/kernel/calls.S arch/arm/kernel/sys_a
rm.c include/linux/syscalls.h include/linux/Yimin_oom.h include/linux/Yimin_struct.h mm
/Kconfig mm/Makefile mm/Yimin_mm_limits.c mm/Yimin_oom.c ; do \
    rsync -u hacking/$file kernel/goldfish/$file; \
done
→ OperatingSystemPrj2 git:(Allow_time_exceed) ✖ |
```

5.2.2 Test on One-click

- Files updating (from hacking list) and kernel compilation can be done using **make kernel**
- The Emulator can be booted using **make emulator**
- The trace_module can be compiled and loaded into avd in one-click using **make module**
- Test programme can be compiled, pushed and run in one-click using **make test**

Please place kernel/goldfish into the directory of this project, and place the text programme into ./test/jni. Then input **make kernel** → **make emulator** → **make module** → **make test**.

6 Conclusion

In this project, I have implemented a new OOM-killer for the android system. This OOM-killer allows user mode programmes to set memory limits for specific users by using a syscall. This OOM-killer has very small overhead for using passive detection technique and dynamic timer technique. And the OOM-killer also supports memory exceeding for a pre-set time.

Passive detection is implemented with tracepoint technique — a "point" in functions we want to detect (in this case, "do_fork" and "do_exit") and a module which responds to the events traced. Only when UIDs in the limits list start to run, will the OOM-killer be triggered regularly. And once no UIDs in the limits list is running, the OOM-killer will not be triggered.

Dynamic Timer means that timer responsible for regularly calling the OOM-killer will be dynamically adjusted according to the degree of "risk".

Allowing the memory limits to be exceeded is implemented with an extra data structure in the OOM-killer which saves the timestamp when an UID starts exceeding the limits. When the OOM-killer is called again, the items on this data structure will be checked, if any of them has exceeded the pre-set time, this process will be killed.

In general, I've implemented bonus parts and other new features in addition to the basic functionality.

7 Feelings

This part is not made public.

8 Reference and Acknowledgement

Reference

Robert Love (2010). Linux Kernel Development Third Edition
[Blog-spot: adding simple system call](#)
[Stack Overflow: how to add a new source file for kernel build](#)
[Stack Overflow: how to use su command over adb shell](#)
[Stack Overflow: getting a user id and a process group id from...](#)
[draapho.github.io: kernel-makefile](#)
[Kernel.org: Kconfig](#)
[ruanyifeng.com: makefile tutorial](#)
[github.com/skyzh/notes: os project makefile](#)
[Ask Ubuntu: how can I kill a process in kernel](#)
[Cnblogs: timer in kernel](#)
[YouTube: kernel symbol table](#)

Acknowledgement

Thanks Chi Zhang (skyzh) for providing this [link](#) and his [notes](#) and inspiring me to use "tracepoint".

Thanks Bugen Zhao (a.k.a Ziqi Zhao) for inspiring me about using a hacking list rather than directly messing up the kernel files.