CS356 Operating System Projects

Project 2: Android Memory Management

> *KylinChen(陈麒麟), 517030910155, Spring 2019*

> *Instructed by professor.Wu*

1 | Project Objectives

- Compile the Android kernel.
- Familiarize Android page replacement algorithm
- Get the target process's virtual address and physical address.
- Implement a page replacement aging algorithm.

2 | Runing Environment

- Ubuntu 18.04.2 LTS
- Android Virtual Devices

3 | Solving Statement

Problem 1: Compile the kernel

> Description

The major aim of this project is to hacking the Android kernel and learn how to program at Linux kernel layer, therefore how to use *ncurses-dev* to compile kernel is the first task we should do because once we modify the *goldfish* kernel we should repeat compiling it to make the changes effective.

> Modify

Once we need to visit the remapped address of page table in Problem 2 and Problem 3, We need a simple but effective way to traverse the remapped memory area. Then I find the walk function *walk_page_range()* in Linux, which can recursively walk the page table for the memory area, you can visit Bootlin to get more details in Linux. However, unfortunately, the function is defined as private-used, that's to say we should modify some kernel file before we modify the kernel.

- Modify **include/linux/mm.h** to make *walk_page_range()* an external function, so that we can call it in our modules.

```
ubuntu ~/goldfish:@ vim include/linux/mm.h
```

Then modify the 945-th lines, adding *extern* before the function definition, just like this:

```
extern int walk_page_range(unsigned long addr, unsigned long end,
    struct mm_walk *walk);
```

- Modify **mm/pagewalk.c** in order to make it possible to call it by reference function name, just open the aim .c file:

```
ubuntu ~/goldfish:@ vim mm/pagewalk.c
```

Then add an macro definition at file head

```
#include <linux/export.h>
```

In order to avoid declaration conflicts, we add the *EXPORT_SYMBOL* in the file tail, like this:

```
EXPORT_SYMBOL(walk_page_range);
```

These steps help us have function call *walk_page_range* in our module programming, then we can compile the Android kernel to generate a mirror we can drive by AVD.

> Compile

- 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_6
4/bin
```

(NDK file locations vary from OS, and depende on your download or extract)

- Open Makefile in **KERNEL_SOURCE/goldfish/**, and find these:

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

Change it to:

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

(In fact, it has been changed if we download the kernel from Jiao Cloud)

- Execute the following command in terminal to set compiling config:

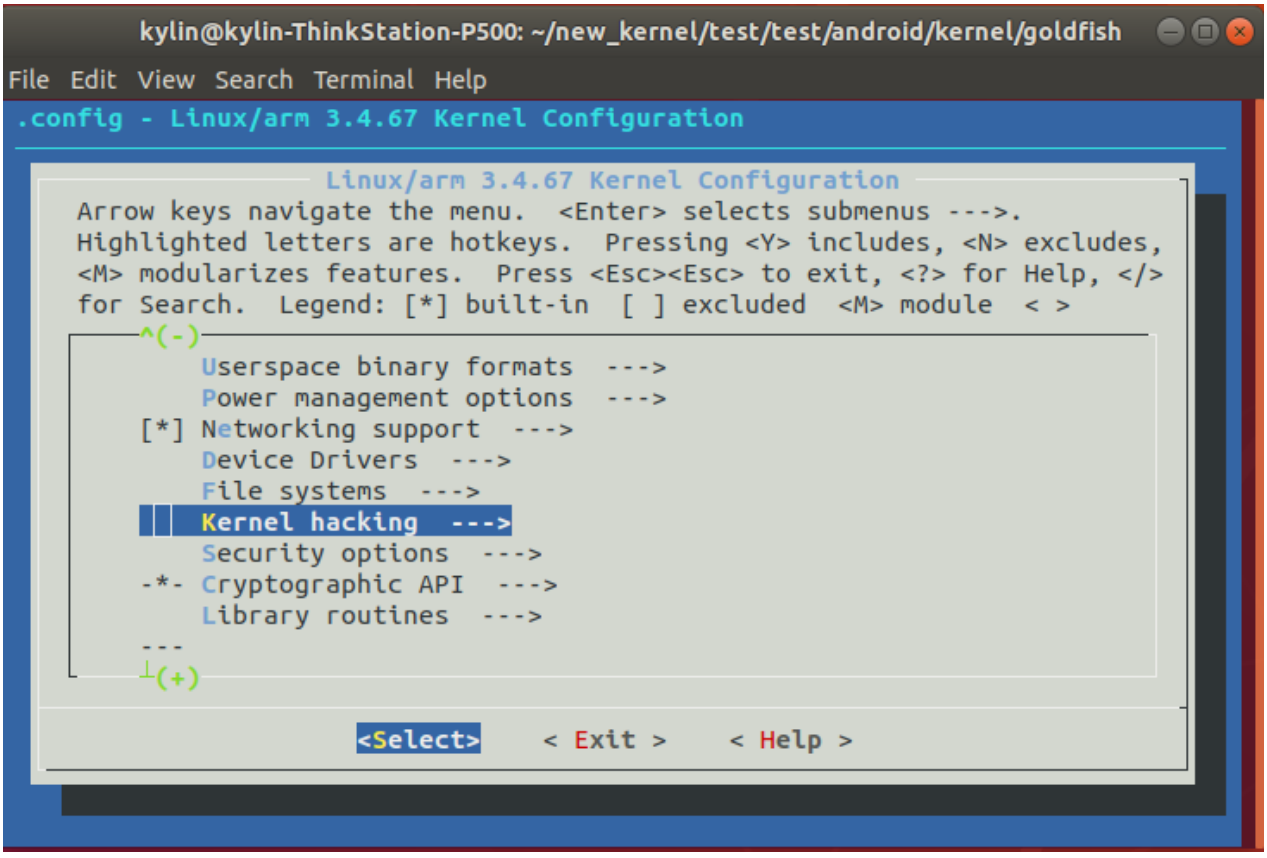
```
make goldfish_armv7_defconfig
```

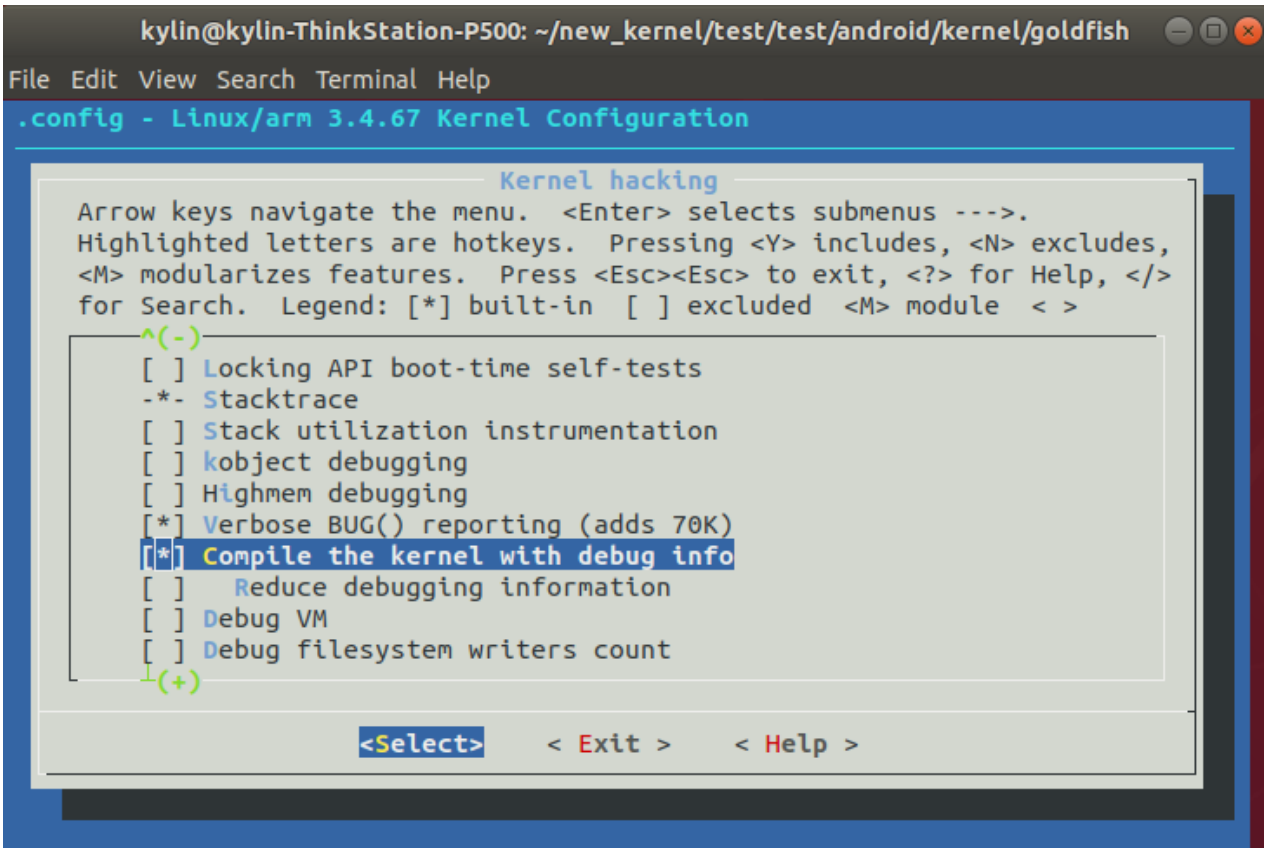
- Modify compiling config:

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

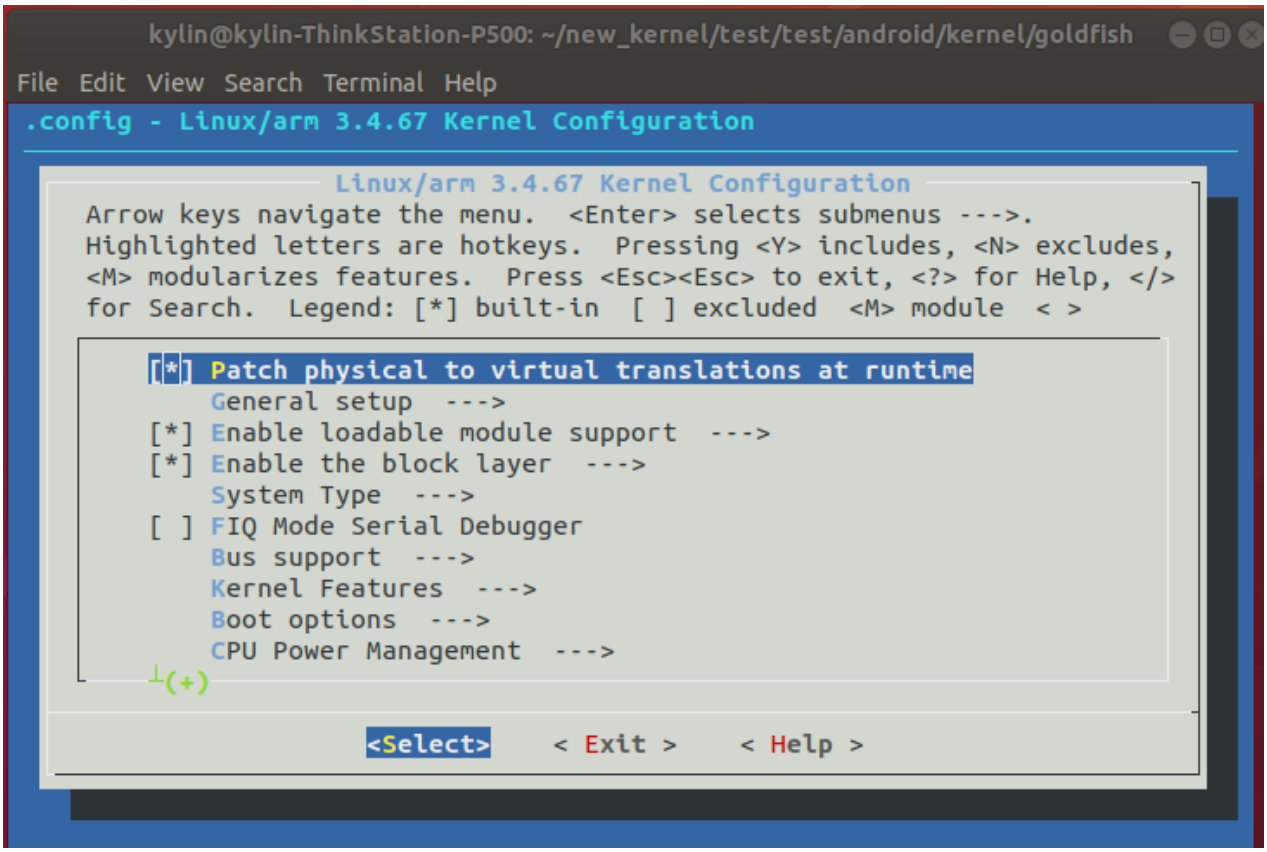
Then we will install the *ncurses-dev*, which help us hack the kernel. And we will see the configuration GUI replace the terminal, just like this:

We select the **Kernel hacking -> Compile the kernel with debug info**





And select **Enable loadable module support -> {Forced module loading, Module unloading, Forced module unloading}**



save and exit the GUI configuration menu.

- Compile the kernel

```
make -j8
```

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

Then we can compile our module and drive AVD by this kernel. And every time we change the previous kernel file in *goldfish* (except .h header file), we should re-compile it to make it effect.

Problem 2: Get the physical address of a process.

> Description

In the Linux kernel, the page table is broken into multiple levels. For ARM64-based devices, a three-level paging mechanism is used, it's named page directory (pgd), page middle directory (pmd), page table.

In order to finish this problem, we should program two system calls before we write our module.

The first is to investigate the page table layout, which gets the page table layout information of current system.

```
static int get_pagetable_layout(struct pagetable_layout_info __user*
pgtbl_info,int size)
```

The second call is mapping a target process's Page Table into the current process's address space.

```
int expose_page_table(pid_t pid,unsigned long fake_pgd,unsigned long
fake_pmds,unsigned long page_table_addr,unsigned long begin_vaddr,unsigned long
end_vaddr)
```

After we change the *goldfish* kernel file, we can use the external function *walk_page_range* if we include **<linux/mm.h>** macro at the system call file header, which will make our work easier to complete.

We make the call number 356 (CS356 class number) to system call *get_pagetable_layout* and call number 357 to system call *expose_page_table*. Moreover, we add some info **KERN_INFO** to print Debug information so that we can find the error while we use the system call the the project.

> Inplementation

```
// expose_page_table

int expose_page_table(pid_t pid,unsigned long fake_pgd,unsigned long
fake_pmds,unsigned long page_table_addr,unsigned long begin_vaddr,unsigned long
end_vaddr)
{
    // init the struct(s) we used
    struct pid* pid_struct;
    struct task_struct *target_process;
```

```
struct vm_area_struct *vm_tmp;
struct mm_walk walk={};
struct walk_info record_data={};

// check the memory address
if(begin_vaddr>=end_vaddr)
{
    printk(KERN_INFO"Invalid Mapped Address!\n");
    return -1;
}

// get pid from expose_page_table parameter
pid_struct=find_get_pid(pid);
if(!pid_struct)
{
    return -1;
}

// get pid task by get_pid_task() method and printk the name.
target_process=get_pid_task(pid_struct,PIDTYPE_PID);
printk(KERN_INFO "Target Process -> %s \n", target_process->comm);

// init record info and alloc memory then check
record_data.page_table_addr=page_table_addr;
record_data.fake_pgd=fake_pgd;
record_data.copied_pgd=kcalloc(PAGE_SIZE,sizeof(unsigned long),GFP_KERNEL);
if(!record_data.copied_pgd)
{
    printk(KERN_INFO"Memory Error!\n");
    return -1;
}

// init walk struct
walk.mm=target_process->mm;
walk.pgd_entry=&callback_pgd;
walk.private=&record_data;

// print target's virtual memory address
printk(KERN_INFO"Virtual Memory:\n");
//unlock semaphore.
down_write(&target_process->mm->mmap_sem);
for(vm_tmp=target_process->mm->mmap;vm_tmp;vm_tmp=vm_tmp->vm_next)
{
    printk(KERN_INFO"%08X->%08X\n",vm_tmp->vm_start,vm_tmp->vm_end);
}
//unlock semaphore.
up_write(&target_process->mm->mmap_sem);

current->mm->mmap->vm_flags|=VM_SPECIAL;
```

```
    //unlock semaphore.
    down_write(&target_process->mm->mmap_sem);
    // walk the page table
    walk_page_range(begin_vaddr,end_vaddr,&walk);
    //lock semaphore.
    up_write(&target_process->mm->mmap_sem);

    //copy the temp fake pgd to userspace.
    if(copy_to_user(fake_pgd,record_data.copied_pgd,sizeof(unsigned
long)*PAGE_SIZE))
    {
        return -1;
    }

    kfree(record_data.copied_pgd);
    return 0;
}
```

```
//  get_pagetable_layout

static int get_pagetable_layout(struct pagetable_layout_info __user*
pgtbl_info,int size)
{
    struct pagetable_layout_info layout;
    printk(KERN_INFO "Syscall get_pagetable_layout() invoked!\n");

    // exception handling
    if (size < sizeof(struct pagetable_layout_info))
        return -EINVAL;

    // get the pagetable layout information
    layout.pgdir_shift = PGDIR_SHIFT;
    layout.pmd_shift = PMD_SHIFT;
    layout.page_shift = PAGE_SHIFT;

    // copy the acquired information to user space
    if (copy_to_user(pgtbl_info, &layout, sizeof(struct pagetable_layout_info)))
        return -EFAULT;
    return 0;
}
```

```
// VAtranslate .c code file

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/types.h>
```

```
#include<sys/mman.h>

#define __NR_get_pagetable_layout 356
#define __NR_expose_page_table 357

// given info struct
struct pagetable_layout_info
{
    uint32_t pgdir_shift;
    uint32_t pmd_shift;
    uint32_t page_shift;
};

void print_usage(int para_num)
{
    printf("============================================================\n");
    printf("Parameters Wrong!\n");
    printf("Need 2 parameters, only %d detected!\n",para_num-1);
    printf("The Correct Format Can be:  ");
    printf("./begin_vaddrTranslate #PID #VA\n");
    printf("============================================================\n");
}

void display_pagetable_layout(struct pagetable_layout_info* layout)
{
    printf("============================================================\n");
    printf("Android Pagetable Layout:\n");
    printf("  pgdir_shift = %d\n", layout->pgdir_shift);
    printf("  pmd_shift = %d\n", layout->pmd_shift);
    printf("  page_shift = %d\n", layout->page_shift);
}

void display_pagetable_expose(pid_t pid, unsigned long *fake_pgd_addr, unsigned
long *page_table_addr, unsigned long begin_vaddr, unsigned long end_vaddr,
unsigned page_size)
{
    unsigned long pgd_ind,pte_ind,phy_addr,mask=page_size-1;
    unsigned long *phy_base;

    // call the expose fuction
    syscall(__NR_expose_page_table, pid, fake_pgd_addr, 0, page_table_addr,
begin_vaddr, end_vaddr);

    //get pgd index.
    pgd_ind = (begin_vaddr >> 21) & 0x7FF;
    pte_ind = (begin_vaddr >> PAGE_SHIFT) & 0x1FF;

    //get physical base of page table.
    phy_base=fake_pgd_addr[pgd_ind];
```

```
    if(phy_base)
    {
        //get the entry in the table.
        phy_addr=phy_base[pte_ind];

        //mask the lower bit of the entry.
        phy_addr=phy_addr&~mask;
        if(phy_addr)
        {
            phy_addr=begin_vaddr&mask|phy_addr;
            printf("Virtual Address Translation:\n");
            printf("  virtual address = 0x%08lx\n", begin_vaddr);
            printf("  physical address = 0x%08lx\n", phy_addr);

printf("============================================================\n");
        }
        else
        {
            printf("virtual address:0x%08lx not locate in the
memory.\n",begin_vaddr);

printf("============================================================\n");
        }
    }
    else
    {
        printf("virtual address:0x%08lx not locate in the
memory.\n",begin_vaddr);

printf("============================================================\n");
    }

}

int main(int argc,char **argv)
{
    // catch the para-fault_infoor exception
    if(argc!=3)
    {
        print_usage(argc);
        return -1;
    }

    pid_t pid;  // parameter PID
    unsigned long begin_vaddr, end_vaddr;  // parameter begin_vaddr
    struct pagetable_layout_info pagetable_info; // info struct

    unsigned long *page_table_addr;
```

```
    unsigned long *fake_pgd_addr;
    unsigned page_size;

    pid=atoi(argv[1]);
    begin_vaddr=strtoul(argv[2],NULL,16);
    end_vaddr=begin_vaddr+1;

    // get the info struct call get layout.
    syscall(__NR_get_pagetable_layout,&pagetable_info,4*3);

    //print the layout offset.
    display_pagetable_layout(&pagetable_info);

    //get address
    page_size=1<<(pagetable_info.page_shift);
    page_table_addr=mmap(NULL,page_size,PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
    fake_pgd_addr=malloc(sizeof(unsigned long)*page_size);

    ////////////////////////
    display_pagetable_expose(pid, fake_pgd_addr, page_table_addr, begin_vaddr,
end_vaddr, page_size);

    //free memory space.
    free(fake_pgd_addr);
    munmap(page_table_addr,page_size);
    return 0;
}
```

Complete code is attached in the folder.

> Running Result

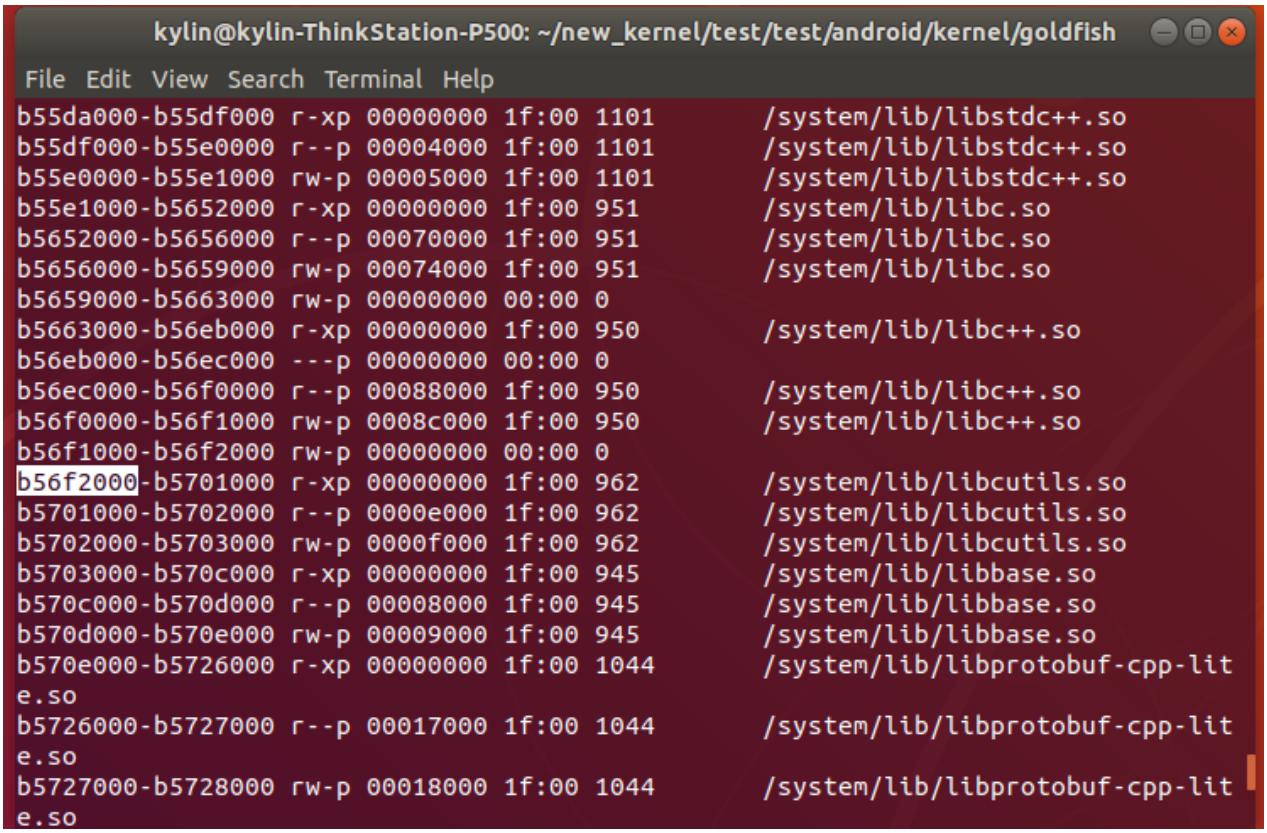
We first start the AVD by the new compiled kernel, then load the *sys_pagetable.ko*root we make in our Linux environment.

```
root@generic:/data/misc # insmod sys_pagetable.ko
```

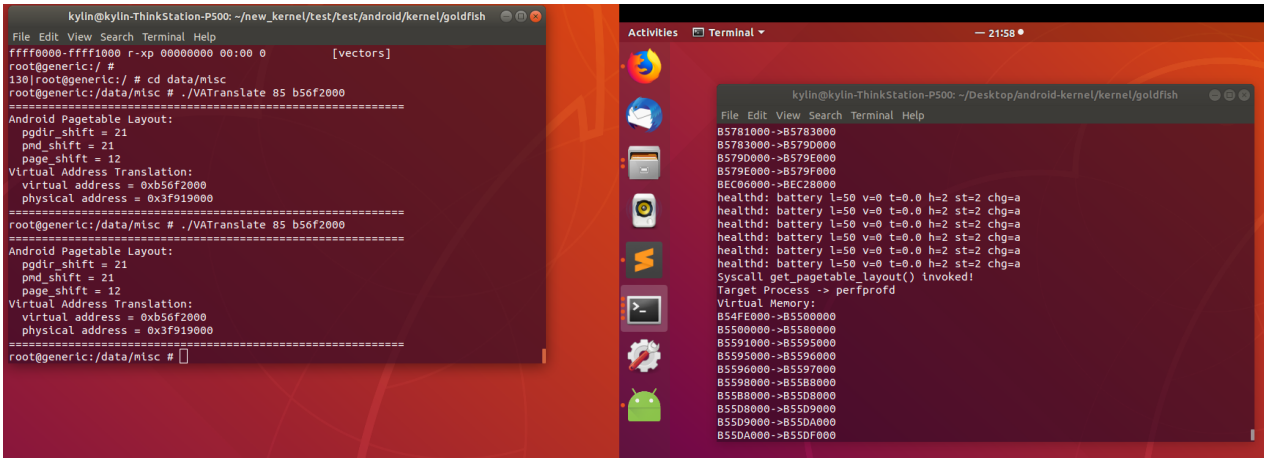
Then we should know some *pid* and its *virtual address* to use VAtranslate to calculate the physical address, so we check the **~/proc** and check the pid **85**:

```
root@generic: # cat proc/85/maps
```

Then we can find the VA from returning value:



Then we choose the PID 85 with VA b56f2000 to run the module:



Left part is the adb shell we run the command in **/data/misc**, and its returning information about its page and physical address:

```
root@generic:/data/misc # ./VATranslate 85 b56f2000
```

And the Right part is the kernel debug information.

Problem 3: Investigate Android Process Address Space

> Description

Implement a program called *vm_inspector* to dump the page table entries of a process in given range. To dump the PTEs of a target process, you will have to specify a process identifier "pid" to vm_inspector.

Try open an Android App in your Device and play with it. Then use vm_inspector to dump its page table entries for multiple times and see if you can find some changes in your PTE dump.

Use *vm_inspector* to dump the page tables of multiple processes, including Zygote. Refer to **/proc/pid/maps** in your AVD to get the memory maps of a target process and use it as a reference to find the different and the common parts of page table dumps between Zygote and an Android app.

For this problem, its just a stronger edition of problem 2, what we need to do is to find the virtua address start and end from **/proc** of our aim pid and then allocate enough free space for page table and fake pgd, release them at the end of module. Of course, we will use the system call 356 and 357 in problem 1 to transfer the address to finish this problem.

> Implementation

```
// vm_inspector.c code file
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/mman.h>

#define __NR_get_pagetable_layout 356
#define __NR_expose_page_table 357

#define pgd_index(va,loinfo) ((va)>>loinfo.pgdir_shift)
#define pte_index(va,loinfo) (((va)>>loinfo.page_shift)&((1<<(loinfo.pmd_shift-
loinfo.page_shift))-1))

// given info struct
struct pagetable_layout_info{
    uint32_t pgdir_shift;
    uint32_t pmd_shift;
    uint32_t page_shift;
};

void print_usage(int para_num)
{

printf("==========================================================================
===\n");
    printf("Parameters Wrong!\n");
    printf("Need 3 parameters, only %d detected!\n",para_num-1);
    printf("The Correct Format Can be:  ");
    printf("./vm_inspector PID BEGIN_VADRR END_VADRR\n");
```

```
printf("================================================================================
===\n");
}


int main(int argc,char **argv)
{
    pid_t pid;
    unsigned long begin_vaddr,current_va;
    unsigned long end_vaddr;
    unsigned long *table_addr,*fake_pgd_addr;
    struct pagetable_layout_info loinfo;
    unsigned long page_size;
    unsigned long rd_begin,rd_end;
    unsigned long mask;
    unsigned long page_nums;

    //check the arguments.
    if(argc!=4)
    {
        print_usage(argc);
        return -1;
    }

    pid=atoi(argv[1]);
    begin_vaddr=strtoul(argv[2],NULL,16);
    end_vaddr=strtoul(argv[3],NULL,16);
    current_va=begin_vaddr;

    //call the layout
    syscall(__NR_get_pagetable_layout, &loinfo,4*3);

    //calculate page size and calculate mask.
    page_size=1<<(loinfo.page_shift);
    mask=page_size-1;

    rd_begin=begin_vaddr&~mask;
    rd_end=(end_vaddr+mask)&~mask;

    //calculate needed pages.
    page_nums=pgd_index(rd_end-1,loinfo)-pgd_index(rd_begin,loinfo)+1;

    //allocate enough memory space.

table_addr=mmap(NULL,page_size*page_nums,PROT_READ|PROT_WRITE,MAP_SHARED|MAP_ANO
NYMOUS,-1,0);
    fake_pgd_addr=malloc(sizeof(unsigned long)*page_size);
```

```
    if(!table_addr||!fake_pgd_addr)
    {
        printf("Memory Error!\n");
        return -1;
    }


syscall(__NR_expose_page_table,pid,fake_pgd_addr,0,table_addr,begin_vaddr,end_va
ddr);

    // in loop variable
    unsigned long table_number;
    unsigned long page_frame_number;
    unsigned long pgd_ind,phy_addr,vir_addr;
    unsigned long *phy_base;
    unsigned long begin_table_number=rd_begin>>loinfo.page_shift,
end_table_number=rd_end>>loinfo.page_shift;

    //print location va,pa,pn,fn

printf("==============================Translation==============================
===\n");
    printf("Virtual Address\t\tPhysical Address\tPage Number\tFrame Number\n");

for(table_number=begin_table_number;table_number<end_table_number;++table_number
)
    {
        vir_addr=table_number<<loinfo.page_shift;
        pgd_ind=pgd_index(vir_addr,loinfo);
        phy_base=fake_pgd_addr[pgd_ind];
        if(phy_base)
        {
            phy_addr=phy_base[pte_index(vir_addr,loinfo)];
            page_frame_number=phy_addr>>loinfo.page_shift;
            if(page_frame_number)
            {

printf("0x%08lx\t\t0x%08lx\t\t0x%08lx\t0x%08lx\n",vir_addr,phy_addr,table_number
,page_frame_number);
            }
        }
    }

printf("=======================================================================
===\n");

    //free memory space.
    free(fake_pgd_addr);
    munmap(table_addr,page_size*page_nums);
```

```
    return 0;
}
```

The complete file is attached in the folder

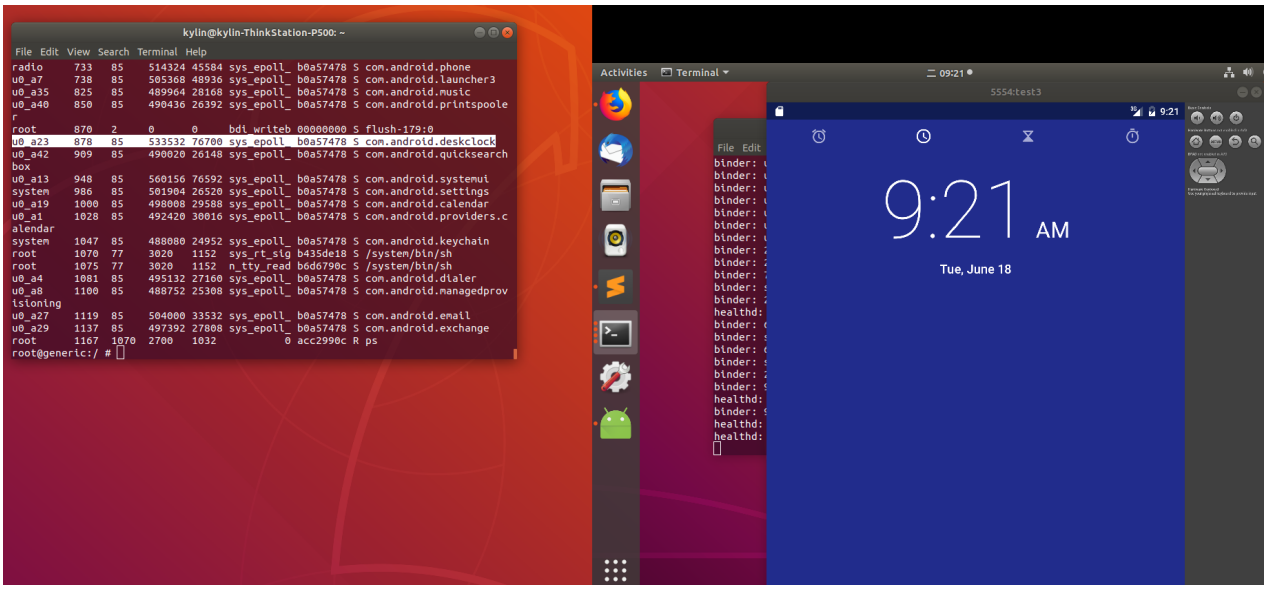
> Running result

(You should execute insmod to load the system call we program before)

Open the App **Deskclock** and use *ps* to find the pid of this application:

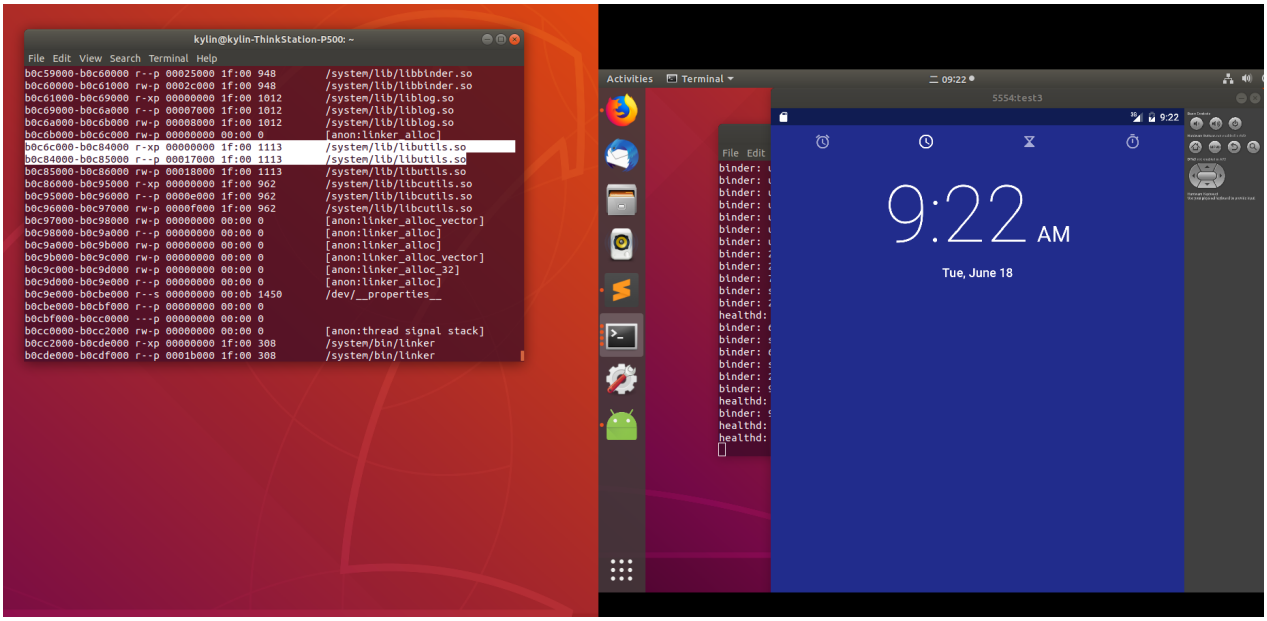
```
root@generic: # ps
```

Then we can get the result of PID 878



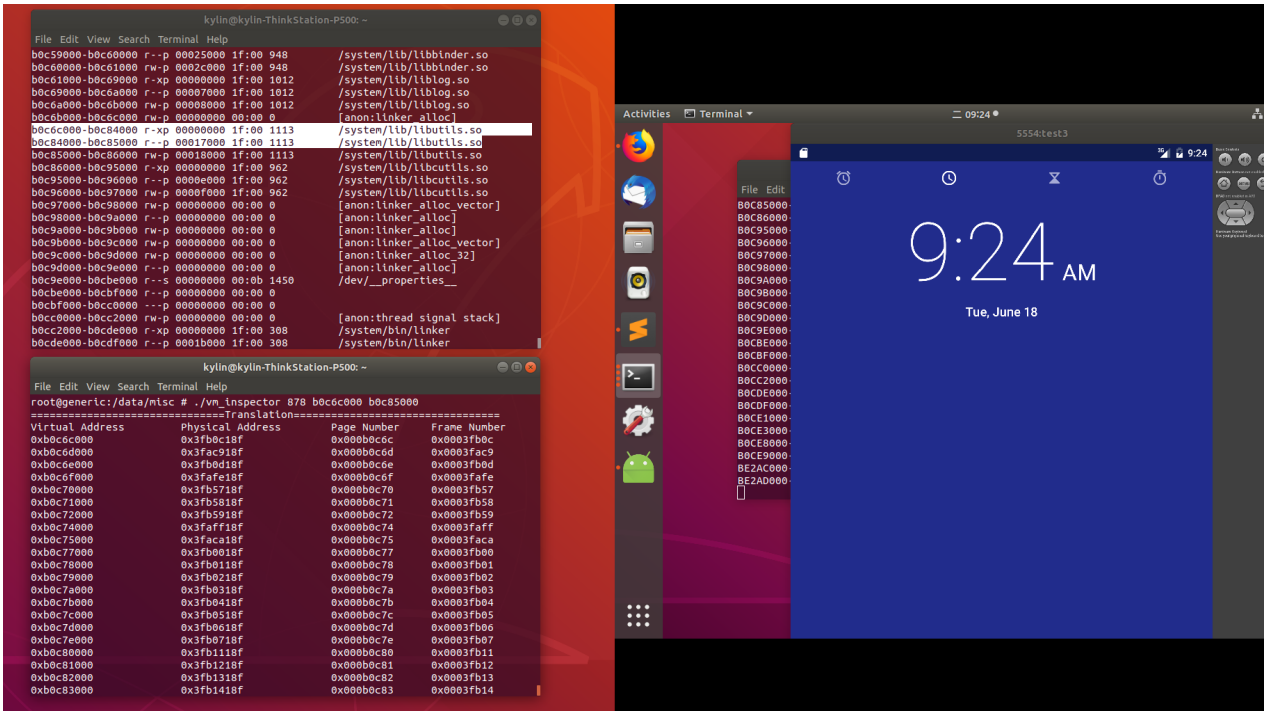
Use **cat** /proc/878/maps command to find a VA of App Deskclock:

```
root@generic: # cat proc/878/maps
```



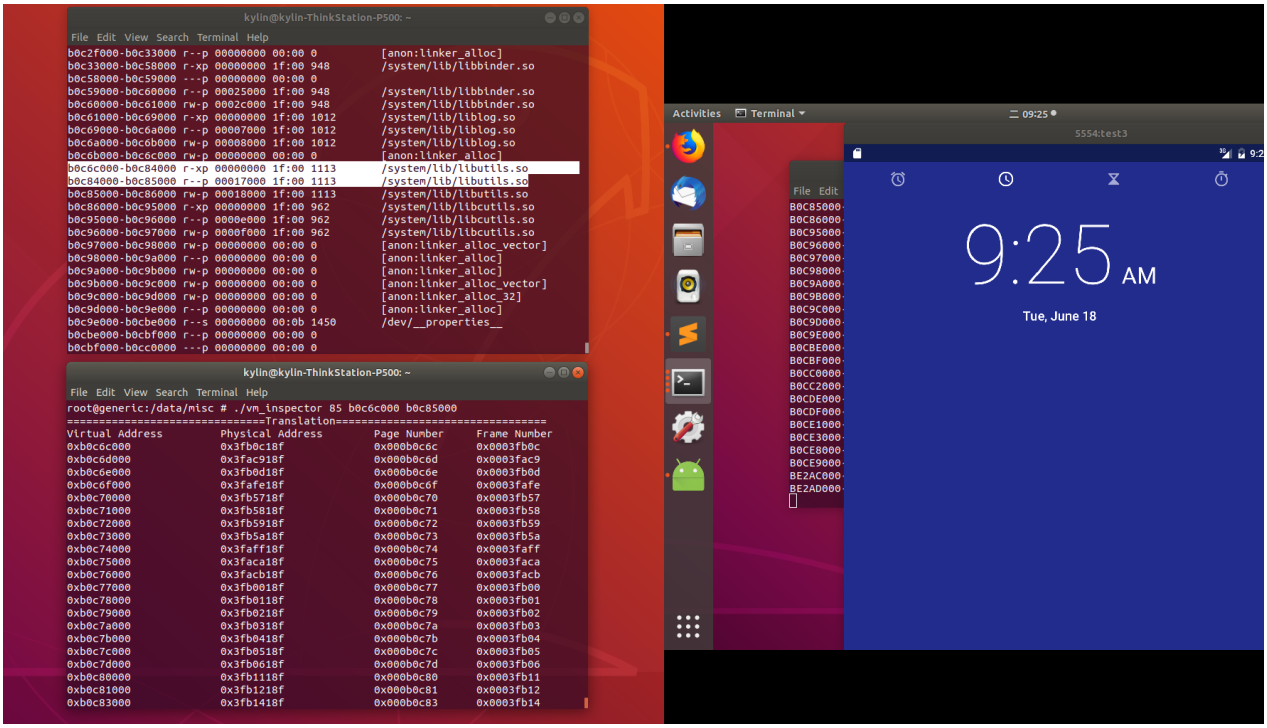
Then execute the vm_inspector module to find the address information

```
root@generic:/data/misc # ./vm_inspector 878 b0c6c000 b0c85000
```



Finally we can use cat to find the page information of Zygote, and execute vm_inspector as well like this:

```
root@generic:/data/misc # ./vm_inspector 85 b0c6c000 b0c85000
```



Conclusion

If we play the app **Deskclock** we can find the PTE changes for some operation, I think thiese operation may reference the memory which didn't load while approach. And for Zygote and Deskclock app, we can find there are many sharing in proc/pid/maps, and these memory area are read-only. Therefore, I think Zygote may be the father process, which pre-load many memory in order to make the approach of apps faster.

Problem 4: Change Linux Page Replacement Algorithm

> Description

In this project, you need to change the page replacement algorithm. You should add a new referenced variable to reflect the importance of a page. If a page is referenced by process, the referenced value of it should be shifted 1 bit to the right and added by 2^k which is defined by yourself. Otherwise, the referenced value shifts 1 bit to the right for every period.

To achieve this Algorithm replacement, we just need to modify the file in *include/linux/mm_types.h* which define a struct **page**, in which has the check information LRU, it means that we can add more variable **RankMyDefine** to give the priority problem described.

```
// in the kernel file: goldfish/include/linux/mm_types.h
struct page{
/* ----------- ------------- */
unsigned long lastRefTime;
/* ----------- ------------- */
};
```

Then we can modify the kernel file mm/swap.c (need compiling after modification), in this file, it have a recently-called function *mark_page_accessed(struct page *page)*, which can be called once asserting is happening. Accoring to the problem description, its priority **RankMyDefine** must add 2.

```
// in the kernel file: goldfish/mm/swap.c
void mark_page_accessed(struct page *page)
{
    //just activate it and set our variable to be zero.
    page->RankMyDefine += 2;
      if (!PageActive(page) && !PageUnevictable(page) && PageLRU(page)) {
        activate_page(page);
        ClearPageReferenced(page);
    } else if (!PageReferenced(page)) {
        SetPageReferenced(page);
    }
    //printk(KERN_INFO"%ld:Found referenced in mark_page_accessed.\n",page-
>index);
}
```

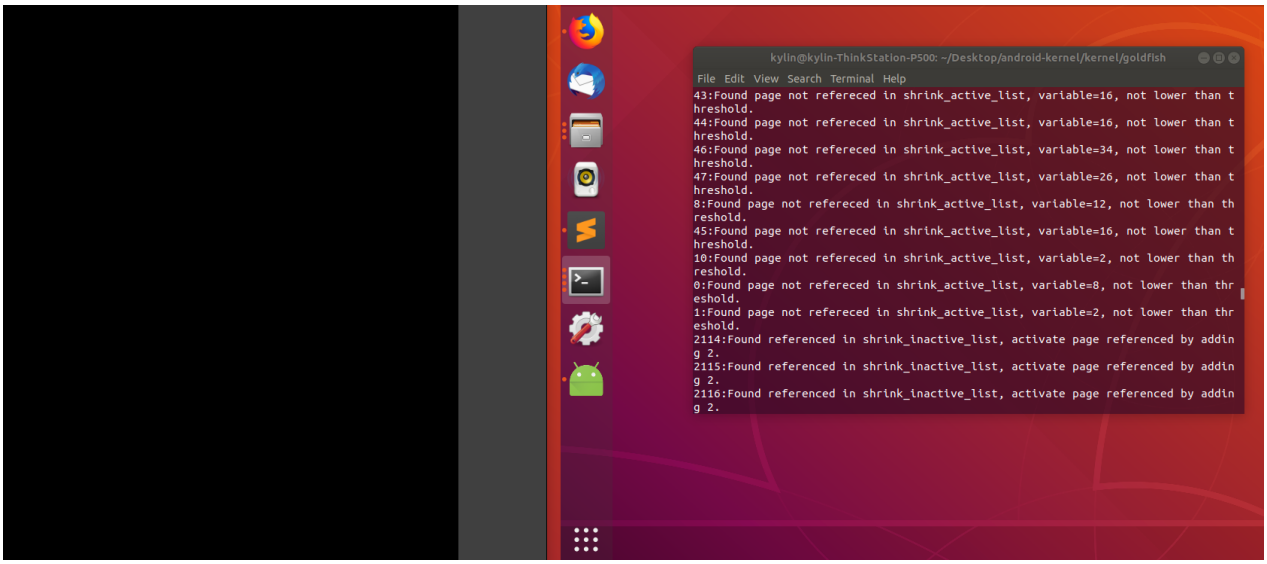
Then we modify other inner module file *mm/vmscan.c* in several locations to define the threshold with 2 and modify it with several changes to make sure every time the priority **RankMyDefine** will right shift 1 bits.

```
// in the kernel file: goldfish/mm/vmscan.c

//  -------------1-th Modification-----------------
#define MY_THRES    2

//  -------------2-th Modification-----------------
static enum page_references page_check_references(struct page *page,
                          struct mem_cgroup_zone *mz,
                          struct scan_control *sc)
{
      /* ----------- ------------- */
    page->RankMyDefine /= 2;
      /* ----------- ------------- */
        SetPageReferenced(page);
        page->RankMyDefine += 2;
        if (tmp_RankMyDefine>2 || referenced_ptes > 1)
            return PAGEREF_ACTIVATE;
        /* ----------- ------------- */
}

//  -------------3-th Modification-----------------
static void shrink_active_list(unsigned long nr_to_scan,
                   struct mem_cgroup_zone *mz,
                   struct scan_control *sc,
                   int priority, int file)
{
    /* ----------- ------------- */

    while (!list_empty(&l_hold)) {
          /* ----------- ------------- */
       if (page_referenced(page, 0, mz->mem_cgroup, &vm_flags)) {
            //let our variable be zero.
            printk(KERN_INFO"%ld:Found page referenced in shrink_active_list.
Add RankMyDefine 2.\n",page->index);
      page->RankMyDefine += 2;
            nr_rotated += hpage_nr_pages(page);
            /* ----------- ------------- */
            list_add(&page->lru, &l_active);
            continue;
        }
        //if our variable is still not larger than the set threshold, it can be
given one more chance.
          else if((page->RankMyDefine) >= MY_THRES ){
            printk(KERN_INFO"%ld:Found page not refereced in shrink_active_list,
variable=%u, not lower than threshold.\n",page->index,page->RankMyDefine);
```

```
            list_add(&page->lru, &l_active);
                continue;
        }

        printk(KERN_INFO"%ld:Found page not refereced in shrink_active_list,
variable=%u, shrinked add to inactive.\n",page->index,page->RankMyDefine);
        ClearPageActive(page);  /* we are de-activating */
        list_add(&page->lru, &l_inactive);
    }
  /* ----------- ------------- */
}

//  -------------4-th Modification-----------------
static unsigned long shrink_list(enum lru_list lru, unsigned long nr_to_scan,
                 struct mem_cgroup_zone *mz,
                 struct scan_control *sc, int priority)
{
    int file = is_file_lru(lru);
    shrink_active_list(nr_to_scan, mz, sc, priority, file);
    return shrink_inactive_list(nr_to_scan, mz, sc, priority, file);
}

//  -------------5-th Modification-----------------
static void kswapd_try_to_sleep(pg_data_t *pgdat, int order, int classzone_idx)
{
    long remaining = 0;
    DEFINE_WAIT(wait);
    if (freezing(current) || kthread_should_stop())
        return;
    prepare_to_wait(&pgdat->kswapd_wait, &wait, TASK_INTERRUPTIBLE);
    if (!sleeping_prematurely(pgdat, order, remaining, classzone_idx)) {
        remaining = schedule_timeout(HZ/10);
        finish_wait(&pgdat->kswapd_wait, &wait);
        prepare_to_wait(&pgdat->kswapd_wait, &wait, TASK_INTERRUPTIBLE);
    }
    if (remaining)
        count_vm_event(KSWAPD_LOW_WMARK_HIT_QUICKLY);
    else
        count_vm_event(KSWAPD_HIGH_WMARK_HIT_QUICKLY);
    // }
    finish_wait(&pgdat->kswapd_wait, &wait);
}
```

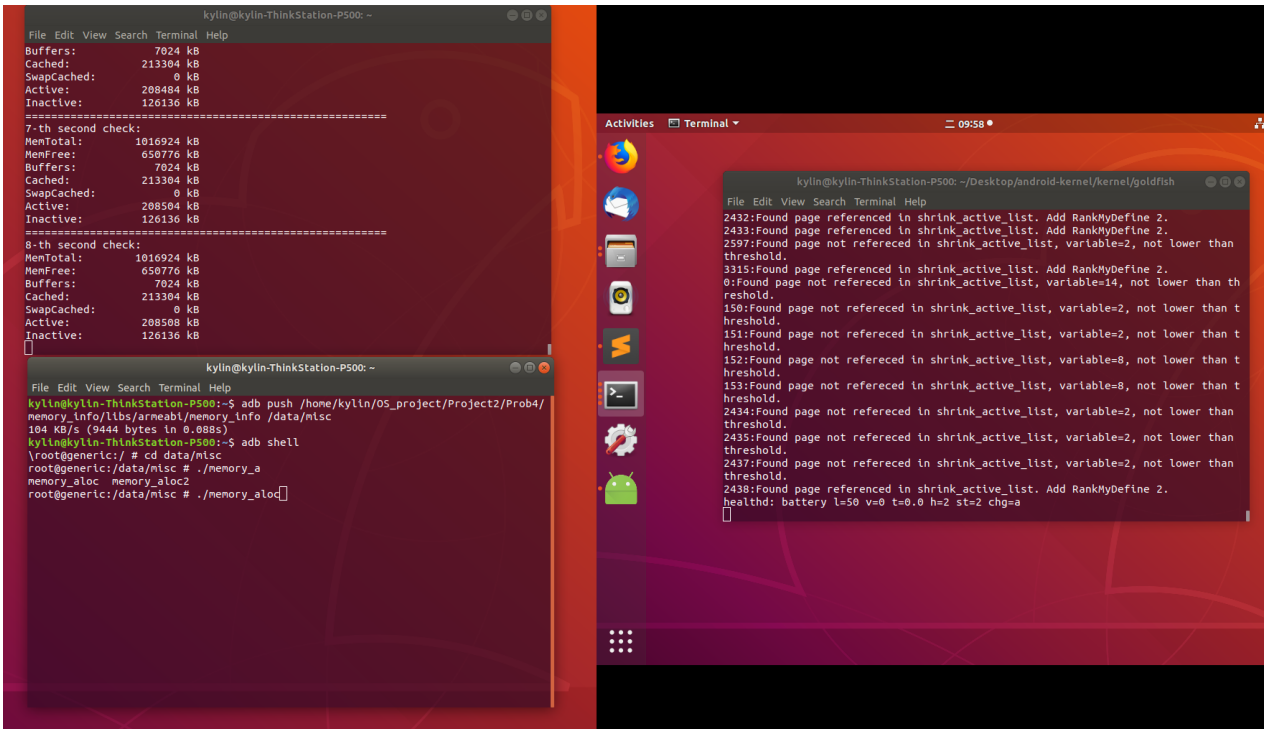
> Result

Once we restart the AVD, we can find that replacement algorithm happens when load system module, I take a screenshoot like this:



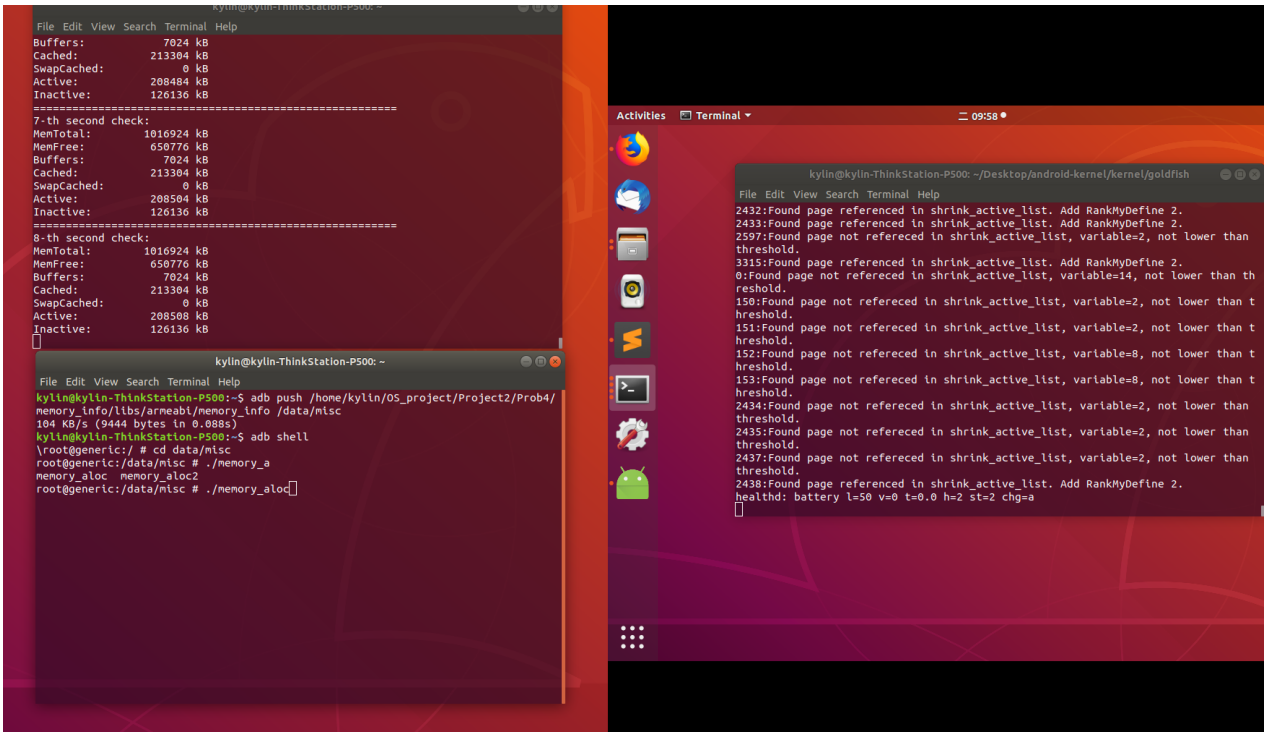
We write two program module **memory aloc** and **memory aloc2** witch will alloc 512M memory from free memory, but this two program will performs differently, because one will always catch hit while another will always miss with the new Algorithm, and we can use the third module **memory_info** to get the memory info every 1 second from *proc*.

At first, we run the *memory_info* module to get the memory information of AVD, it last 30 seconds, intervals 2 seconds. Only in this way, can we make sure the AVD has loaded all the system module. In steady state, we can run our memory alloc module to test new algorithm. In this creenshoot we get the steady state because of the information we get from **memory_info**:

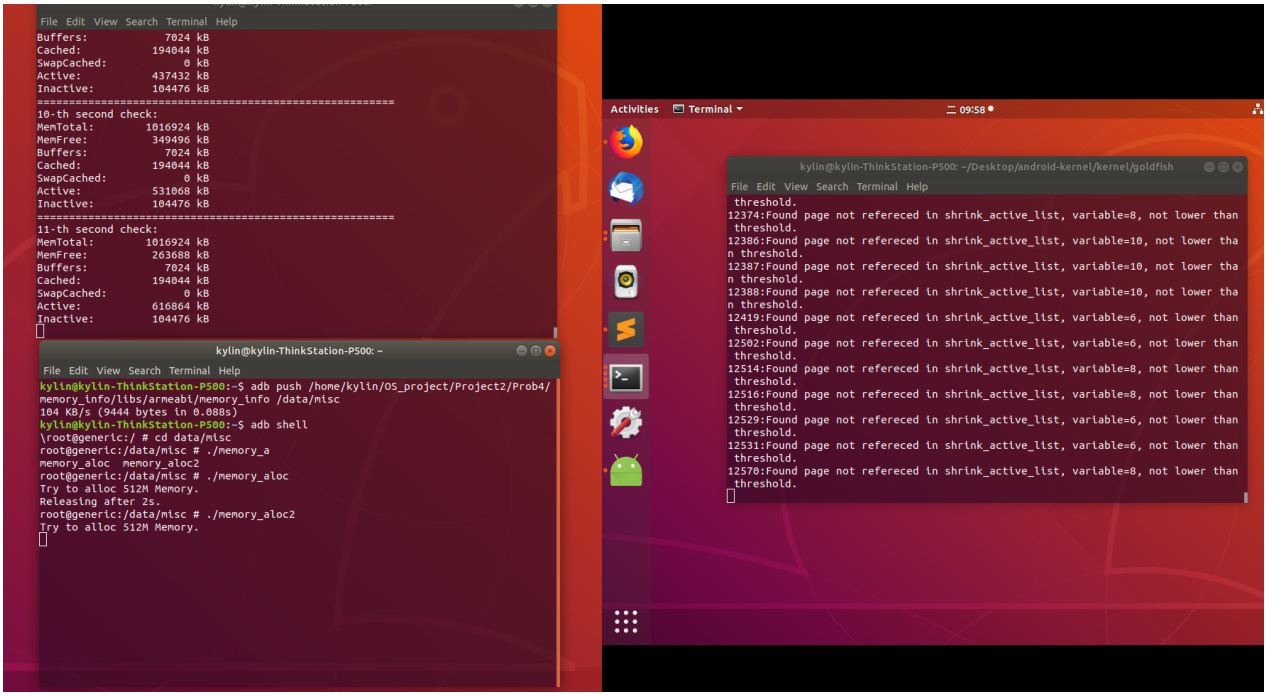


Finally, we run these two memory-alloc module **memory aloc** and **memory aloc2** to test the new algorithm. From the degugging information (contains priority value **RankMyDefine** value change, for example, add 2 or right shift) we can make sure the new algorithm make sense!

> Test memory_aloc1:



> Test memory_aloc2:



Conclusion

In this projects, I really get to know how to hack the kernel file and much Linux Knowledge, for example, kernel space and user space, page replacement algorithm. By reading the kernel code and trying to program my own kernel file, my programming skills really improved and know how to track a difficult problem in Linux. I indeed appreciate my gratitude to Prof.Wu and class TAs for so much help.