Shanghai Jiao Tong University

CS356
Operating System

Project2 Report

Name: Yiheng Zhang
Student ID: 515030910216
E-mail: seanthem@sina.com/
hankzhangcb@gmail.com

Abstract

This is a report for project 2 of CS356 course in SJTU. Basically, it gives each problem a corresponding illustration and analysis.

The project is aimed to help students has a better understanding of Linux operating system in a high-level scale, as well as how to write a user module with system call efficiently. Furthermore, the project requires students try to implement a new method of Linux page replacement algorithm, which is both challenging and exciting.

For the report, it consists of 4 parts, each of which explains how I finish the tasks in my way. It may contain the description of original problems, source code of my solutions, the analysis of how my code functions, the test outcomes, some key points and what I thought and learnt when working on the project.

Last but not least, since my code is well self-explaining, there is not many comments in my program. If any part of my programs looks confusing, please contact me and I will be willing to explain.

keywords: operating system, Linux module, memory explorer, Linux page replacement algorithm,

CS356 Project2 Report

Yiheng Zhang 515030910216

Contents

1 Problem 1

1.1 Compile the Linux kernel

There are not many words for this part, as it is not required in the project description.

To compile a new Linux kernel, ensure the kernel path has been added to the environment variables, and the ARCH and COMPILE configuration in Makefile is modified as required.

In terminal, enter the local path of Linux kernel and start compiling with command make goldfish_armv7_defconfig. To invoke a GUI config interface, install n-curses.dev and enter make menuconfig.

Note:

Type *Y* to include selected option and *N* to exclude it. At first I wasted some time in this part since I failed to find out how to set specific flag of one option to true, orz.

Set config as instructed, and enter make -j*number_of_cores* to compile a new Linux kernel.

2 Problem 2

2.1 Investigate page table layout

Problem Description:

The first is to investigate the page table layout. You need to implement a system call to get the page table layout information of current system.

Since our Android emulator is 32-bit, the page table has two layers only. Thus, the PGD and PMD information should be the same.

we use the following user defined C structure to store the page table information

```
1    struct pagetable_layout_info
2    {
3        uint32_t pgdir_shift;
4        uint32_t pmd_shift;
5        uint32_t page_shift;
6    };
```

The shift indicates the number of bits we should right shift to find the entry of next layer's page index.

Furthermore, the interface of our system call looks like:

```
1    int get_pagetable_layout
2    (
3        struct pagetable_layout_info __user * pgtbl_info,
4        int size
5    );
```

pgtbl_info: user address to store the related information

size : the memory size reserved for *pgtbl_info*

Source Code:

```
1    #include <linux/init.h>
2    #include <linux/init_task.h>
3    #include <linux/kernel.h>
4    #include <linux/list.h>
5    #include <linux/mm.h>
6    #include <linux/mman.h>
7    #include <linux/module.h>
8    #include <linux/pid.h>
9    #include <linux/rwlock_types.h>
10   #include <linux/rwlock.h>
11   #include <linux/spinlock.h>
12   #include <linux/sched.h>
13   #include <linux/slab.h>
14   #include <linux/string.h>
15   #include <linux/unistd.h>
16   #include <linux/uaccess.h>
17   #include <asm/pgtable.h>
18   #include <asm/page.h>
19   #include <asm/memory.h>
20
21   #include "pagetable_layout_info.h"
22
23   MODULE_LICENSE("Dual BSD/GPL");
24   #define __NR_get_pagetable_layout 378
25   DEFINE_RWLOCK(buf_lock);
26
27   static int (*oldcall)(void);
28
29   //definition of system call get_pagetable_layout
30   static int get_pagetable_layout
31   (
```

```
32      struct pagetable_layout_info __user * pgtbl_info,
33      int size
34      )
35      {
36          struct pagetable_layout_info kernel_info;
37
38          if (size < sizeof(struct pagetable_layout_info))
39          {
40              printk(KERN_INFO "THE SIZE OF PGTBL_INFO IS TOO SMALL!\n");
41              return -EINVAL;
42          }
43          kernel_info.pgdir_shift = PGDIR_SHIFT;
44          kernel_info.pmd_shift = PMD_SHIFT;
45          kernel_info.page_shift = PAGE_SHIFT;
46          if(copy_to_user(pgtbl_info, &kernel_info, size))
47          {
48              printk(KERN_INFO "COPY BACK TO USER SPACE FAIL!\n");
49              return -EFAULT;
50          }
51          return 0;
52      }
53
54      static int addsyscall_init(void)
55      {
56          long *syscall = (long*)0xc000d8c4;
57          oldcall = (int(*)(void))(syscall[__NR_get_pagetable_layout]);
58          syscall[__NR_get_pagetable_layout] = (unsigned long)
                get_pagetable_layout;
59          printk(KERN_INFO "GET_PAGETABLE_LAYOUT MODULE LOAD!\n");
60          return 0;
61      }
62
63      static void addsyscall_exit(void)
64      {
65          long *syscall = (long*)0xc000d8c4;
66          syscall[__NR_get_pagetable_layout] = (unsigned long )oldcall;
67          printk(KERN_INFO "GET_PAGETABLE_LAYOUT MODULE EXIT!\n");
68      }
69
70      module_init(addsyscall_init);
71      module_exit(addsyscall_exit);
```

Analysis:

The functional structure of this system call is relatively very simple, lying between line 32 and line 52. All we have to do is to fetch three macros defined in Linux kernel file pagetable.h named PGDIR_SHIFT, PMD_SHIFT(should be the same with PGDIR_SHIFT due to the 32-bit structure) and PAGE_SHIFT. Then copy them back from kernel space to our page_layout_info struct in user space.

Outcome:

Fig.2.1 shows the test result of the two-level page table structure.

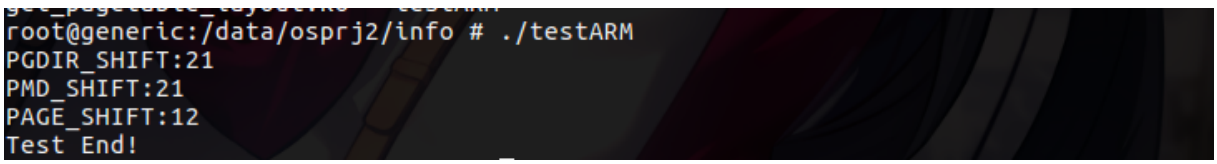


Figure 2.1: Outcome of system call page_table_layout_info

2.2 Target process page table remapping

Problem Description:

There is a bit of changes to this part. Because of the nonexistence of the second-level page table, all the variables related with *pmd* will be discarded.

The second call is mapping a target process's Page Table into the current process's address space. After successfully completing this call, *page_table_addr will* contain part of page tables of the target process. To make it efficient for referencing the re-mapped page tables in user space, your syscall is asked to build a *fake pgd*. The *fake pgd* will be indexed by *pgd_index(va)*.(where *va* is a given virtual address).

the interface of our system call looks like:

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

pid: pid of the target process you want to investigate,

fake_pgd: base address of the fake pgd

*fake_pmd*s: base address of the fake pmds

page_table_addr: base address in user space the ptes mapped to

[*begin_vaddr*, *end_vaddr*]: remapped memory range of the target process

Source Code:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/unistd.h>
#include <linux/uaccess.h>
#include <linux/mm.h>
#include <linux/mm_types.h>
#include <linux/slab.h>
#include <asm/errno.h>
#include <asm/pgtable.h>

#include "pagetable_layout_info.h"

MODULE_LICENSE("Dual BSD/GPL");
#define __NR_expose_page_table      379
#define PGD_PAGE_COUNT              3

static int (*oldcall)(void);

struct walk_record
{
    unsigned long *pgd;
    unsigned long pte_base;
};

//originally defined in /include/linux/mm.h line 1225
//an interface for user to implement custom walk method
int user_pmd_entry
(
    pmd_t *pmd,
    unsigned long addr,
    unsigned long next,
    struct mm_walk *walk)
{
    struct walk_record *record = (struct walk_record*)walk->private;
    unsigned long index = pgd_index(addr);
    int pfn;
    int err;
```

```
40
41          struct vm_area_struct *vma = find_vma(current->mm, record->
                pte_base);
42
43          //argument check
44          if(!vma)
45          {
46              printk(KERN_INFO "FAIL TO FIND VMA!\n");
47              return -EFAULT;
48          }
49          if(!pmd)
50          {
51              printk(KERN_INFO "INVALID PMD!\n");
52              return -EINVAL;
53          }
54          if(pmd_bad(*pmd))
55          {
56              printk(KERN_INFO "BAD PMD!\n");
57              return -EINVAL;
58          }
59
60          pfn = page_to_pfn(pmd_page((unsigned long)*pmd));
61          if (!pfn_valid(pfn))
62          {
63              printk(KERN_INFO "BAD PFN!\n");
64              return -EINVAL;
65          }
66
67          err = remap_pfn_range(vma, record->pte_base, pfn, PAGE_SIZE, vma
                ->vm_page_prot);
68          if(err)
69          {
70              printk(KERN_INFO "COPY PTE PAGE TO USER SPACE FAIL!\n");
71              return -EFAULT;
72          }
73
74          //we use the index of va to trace the pa
75          record->pgd[index] = record->pte_base;
76          record->pte_base += PAGE_SIZE;
77          printk(KERN_INFO "pgd[%lu]:\t 0x%08lx\n", index, record->pte_base
                );
78
79          return 0;
80      }
81
82      void print_out_all_vma(struct task_struct* task)
83      {
84          struct vm_area_struct *vma;
85
86          printk(KERN_INFO "========== Virtual Addresses ==========\n");
87          for (vma = task->mm->mmap; vma; vma = vma->vm_next)
88          {
89              printk(KERN_INFO "0x%08lx \t 0x%08lx\n", vma->vm_start, vma->
                    vm_end);
90          }
91      }
92
93      static int expose_page_table(
94          pid_t pid,
95          unsigned long fake_pgd,
96          unsigned long page_table_addr,
97          unsigned long begin_vaddr,
98          unsigned long end_vaddr)
99      {
100         struct walk_record record;
101         struct mm_walk walk;
102         struct task_struct *target;
103         struct pid *pid_tmp;
104
105         //check argument
106         if (!fake_pgd)
107         {
108             printk(KERN_INFO "INVALID FAKE_PGD\n");
```

```
109             return -EINVAL;
110         }
111
112         if (!page_table_addr)
113         {
114             printk(KERN_INFO "INVALID PAGE_TABLE_ADDR\n");
115             return -EINVAL;
116         }
117
118         if(begin_vaddr >= end_vaddr)
119         {
120             printk(KERN_INFO "INVALID BEGIN AND END VA\n");
121             return -EINVAL;
122         }
123
124         //find targer task struct by pid
125         pid_tmp = find_get_pid(pid);
126         if (!pid_tmp)
127         {
128             printk(KERN_INFO "INVALID PID!\n");
129             return -EINVAL;
130         }
131         target = get_pid_task(pid_tmp, PIDTYPE_PID);
132         printk(KERN_INFO "Target process name:%s\n", target->comm);
133
134         //target process page table lock and read
135         down_write(&target->mm->mmap_sem);
136         print_out_all_vma(target);
137
138         //set up my walk
139         record.pgd = (unsigned long*)kmalloc(PGD_PAGE_COUNT * PAGE_SIZE,
                GFP_KERNEL);
140         if (!record.pgd)
141         {
142             printk(KERN_INFO "RECORD PGD SPACE ALLOCATE FAIL!\n");
143             return -EFAULT;
144         }
145         memset(record.pgd, 0, PGD_PAGE_COUNT * PAGE_SIZE);
146         record.pte_base = page_table_addr;
147         walk.pgd_entry = NULL;
148         walk.pud_entry = NULL;
149         walk.pmd_entry = user_pmd_entry;
150         walk.pte_entry = NULL;
151         walk.pte_hole = NULL;
152         walk.hugetlb_entry = NULL;
153         walk.mm  = target->mm;
154         walk.private = (void*)(&record);
155
156         walk_page_range(begin_vaddr, end_vaddr, &walk);
157         up_write(&target->mm->mmap_sem);
158
159         // copy the level-one page table to fake pgd
160         if (copy_to_user((void*)fake_pgd, record.pgd, PGD_PAGE_COUNT *
                PAGE_SIZE))
161         {
162             printk(KERN_INFO "COPY PGD PAGE TO USER SPACE FAIL!\n");
163             return -EFAULT;
164         }
165
166         kfree(record.pgd);
167         return 0;
168     }
169
170     static int addsyscall_init(void)
171     {
172         long *syscall = (long*)0xc000d8c4;
173         oldcall = (int(*)(void))(syscall[__NR_expose_page_table]);
174         syscall[__NR_expose_page_table] = (unsigned long)
                expose_page_table;
175         printk(KERN_INFO "EXPOSE_PAGE_TABLE MODULE LOAD!\n");
176         return 0;
177     }
178
```

```
179     static void addsyscall_exit(void)
180     {
181         long *syscall = (long*)0xc000d8c4;
182         syscall[__NR_expose_page_table] = (unsigned long )oldcall;
183         printk(KERN_INFO "EXPOSE_PAGE_TABLE MODULE EXIT!\n");
184     }
185
186     module_init(addsyscall_init);
187     module_exit(addsyscall_exit);
```

Analysis:

The system call is composed with two main parts, one being find the target process by pid and investigate its all valid virtual addresses, and the other begin using kernel function page_range_walk to remap all the page table to given physical address in user space.

As kernel function find_task_by_vpid is reserved in the kernel files, we have to find another way round to get target process by pid. Luckily, we have find_get_pid and get_pid_task. We employ them in line 125 and line 131. One thing needs paying attention to is, once we want to investigate the target process, we have to lock the semaphore of which stored in mm->mmap_sem (line 135). To fetch all the continues valid virtual address blocks, we defined a new function named print_out_all_vma between line 82 and line 90 to help us display information of some process's va and it is used in line 136.

Now we start the remapping. However, a powerful kernel function named page_range_walk is not exported, so we have to modified some files of our kernel, adding EXPORT_SYMBOL(*function name*) to make function available to modules. In the core of page_range_walk, we can find a interface struct reserved for users called mm_struct. If we offers out methods to it, it will automatically invoke our methods when entering specific level of page table. Since we want to copy 2-level page table to user space and trace them by building fake_pgds, we have to allocate a new array to store fake_pgds and write our own pmd_entry function.

In our pmd_entry function from line 29 to line 80, remap_pfn_range is called to remap one whole second-level page table to user space. Furthermore, we record the base physical address of remapped page table to corresponding virtual address's pgd_index in our fake_pgd array if remapping succeeds(line 75 to line 77).

At last, we finish our system call by copying the fake_pgd array from kernel space to user space(line 160 ot line 163).

Outcome:

We choose the **initial process** 1(with pid =1) to be the target process to expose page table layout. First, we print out all the continues valid virtual addresses' begin and end virtual addresses of this process, then remap the second level page table to user space with kernel function and store the base physical addresses of second-level page tables in the corresponding blocks of fake first-level page table.

Fig.2.2 shows the our expose result.

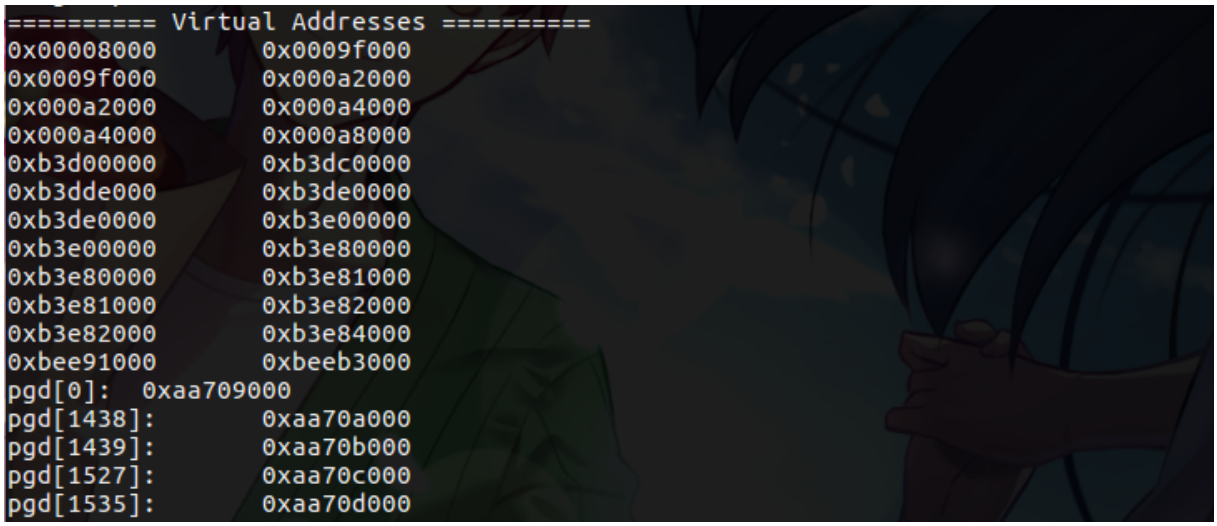


Figure 2.2: Outcome of exposing **initial process**.

2.3 VATranslate

Problem Description:

Besides these two system calls, you should develop an application named VATranslate to test the system calls. The application should run on AVD. You can get the physical address via command "./VATranslate *VA pid*".

Source Code:

```
1   /*=====================
2   Creator: Renjie Gu
3   StudentID: 515030910086
4   Problem: VATranslate
5   =====================*/
6   #include <stdio.h>
7   #include <sys/syscall.h>
8   #include <unistd.h>
9   #include <string.h>
10  #include <stdlib.h>
11  #include <errno.h>
12  #include <sys/mman.h>
13  #include "pagetable_layout_info.h"
14
15  #define PGD_COUNT 1528
16  #define PGD_PAGE_COUNT 3
17
18  #define PGDIR_SHIFT             21
19  #define pgd_index(addr)         ((addr) >> PGDIR_SHIFT)
20  #define pte_index(addr)         ((addr >> PAGE_SHIFT) & 511)
21  #define page_offset(addr)       (addr & 4095)
22
23  #define PHYS_MASK 0xFFFFF000
24
25  int main(int argc, char **argv)
26  {
27      struct pagetable_layout_info info;
28      int pid, ret;
29      char *ptr;
30      void *addr, *fake_pgd_addr;
31      unsigned long begin_vaddr = 0;
32      unsigned long end_vaddr = 0;
33      unsigned long page_table_addr, fake_pgd, index;
34      unsigned long **fake_pgd_new;
35
36      if (argc == 3)
37      {
```

```
38              begin_vaddr = strtoul(argv[2], &ptr, 16);
39              end_vaddr = begin_vaddr + 1;
40          }
41          else if (argc < 2 || argc > 4)
42          {
43              printf("USAGE: ./VATranslate pid va\n");
44              return -1;
45          }
46
47          if(syscall(378, &info, sizeof(struct pagetable_layout_info)))
48          {
49              printf("GET PAGE TABLE LAYOUT INFO FAIL!\n");
50              return -1;
51          };
52
53          printf("pgdir_shift: %u\t", info.pgdir_shift);
54          printf("pmd_shift: %u\t", info.pmd_shift);
55          printf("page_shift: %u\n", info.page_shift);
56
57          int ctr = 0;
58
59          pid = atoi(argv[1]);
60
61          addr = mmap(0, PGD_COUNT * PAGE_SIZE, PROT_READ,
62          MAP_SHARED | MAP_ANONYMOUS, -1, 0);
63
64          if (addr == MAP_FAILED)
65          {
66              printf("MMAP ADDR FAIL: %s\n", strerror(errno));
67              return -1;
68          }
69
70          page_table_addr = (unsigned long) addr;
71
72          fake_pgd_addr = mmap(0, PGD_PAGE_COUNT * PAGE_SIZE,
73          PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0);
74
75          if (fake_pgd_addr == MAP_FAILED)
76          {
77              printf("MMAP FAKE FAIL: %s\n", strerror(errno));
78              goto free_addr;
79          }
80
81          fake_pgd = (unsigned long) fake_pgd_addr;
82
83          if (syscall(379, pid, fake_pgd,
84          page_table_addr, begin_vaddr, end_vaddr))
85          {
86              printf("EXPOSE PAGE TABLE FAIL: %s\n", strerror(errno));
87              goto free_all;
88          }
89
90          fake_pgd_new = (unsigned long **) fake_pgd_addr;
91
92          index = pgd_index(begin_vaddr);
93
94          printf("virtual address:0x%08lx ====> PA: 0x%08lx\n", begin_vaddr
                ,
95          (*(fake_pgd_new[index] + pte_index(begin_vaddr)) & PHYS_MASK)
96          + page_offset(begin_vaddr));
97
98
99      free_addr:
100         munmap(addr, PGD_COUNT * PAGE_SIZE);
101         return -1;
102
103     free_fake_pgd:
104         munmap(fake_pgd_addr, PGD_PAGE_COUNT * PAGE_SIZE);
105         return -1;
106
107     free_all:
108         munmap(addr, PGD_COUNT * PAGE_SIZE);
109         munmap(fake_pgd_addr, PGD_PAGE_COUNT * PAGE_SIZE);
```

```
110         return -1;
111
112         return 0;
113     }
```

Analysis:

After finishing expose_page_table system call, VATranslate is easy to implement. We just need to invoke expose_page_table system call with input *pid* set to the same of VATranslate's input, and the begin and end addresses set to *va* and *va*+1(line 39 and line 83). Then we can easily trace to the *pa* of input *va* in user space by rules defined in the kernel(here I copied them out from kernel to the heading part in line 18 to line 21).

If input virtual address is invalid, VATranlate will throw a error information.

Outcome:

Fig.2.3 shows the test result with valid input with pid = 1

```
pgdir_shift: 21 pmd_shift: 21   page_shift: 12
virtual address:0x0009f000 ====> PA: 0x3fc9e000
```

Figure 2.3: Outcome of valid input with pid = 1.

Fig.2.4 shows the test result with invalid input with pid = 1

```
255|root@generic:/data/osprj2 # ./VATranslate 1 0x01234567
pgdir_shift: 21 pmd_shift: 21   page_shift: 12
Segmentation fault
```

Figure 2.4: Outcome of invalid input with pid = 1.

3 Problem 3

3.1 vm_inspector

Problem 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.

Based on cross-referencing and the output you retrieved, can you tell what are the shared objects between an Android app and Zygote?

Source Code:

```
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/mman.h>

#define PGD_COUNT                       1528
#define PGD_PAGE_COUNT                  3
#define PGDIR_SHIFT                     21
#define PHYS_MASK 0xFFFFF000

#define pgd_index(page_table_addr)      ((page_table_addr) >>
    PGDIR_SHIFT)

void print_out(unsigned long *addr, int pgd_index, int pte_index,
unsigned long begin_vaddr, unsigned long end_vaddr)
{
    unsigned long pte;
    unsigned long va;
    unsigned long phys_addr;
    unsigned long pgd_address;


    if (addr == NULL)
        return;

    pte = *addr;

    pgd_address = pgd_index * 512 * PAGE_SIZE;
    va = pgd_address + pte_index * PAGE_SIZE;

    phys_addr = pte & PHYS_MASK;

    if (phys_addr == 0)
        return;

    if (va > end_vaddr)
        return;

    if (va + (1 << 12) >= begin_vaddr)
        printf("0x%x\t0x%08lx\t0x%08lx\n", pgd_index, va, phys_addr);
}

```

```c
void print_pte_table(unsigned long *addr, int index,
unsigned long begin_vaddr, unsigned long end_vaddr)
{
    int i;

    if (addr == NULL)
        return;

    for (i = 0; i < 512; i++)
        print_out(addr++, index, i, begin_vaddr, end_vaddr);
}

int main(int argc, char **argv)
{
    int pid;
    char *ptr;
    void *addr, *fake_pgd_addr;
    unsigned long begin_vaddr = 0;
    unsigned long end_vaddr = 0;
    unsigned long page_table_addr, fake_pgd, index;
    unsigned long **fake_pgd_new;
    int verbose = 0, i = 0;

    if (argc == 4)
    {
        begin_vaddr = strtoul(argv[2], &ptr, 16);
        end_vaddr = strtoul(argv[3], &ptr, 16);
    }
    else if (argc == 3 || argc < 2 || argc > 4)
    {
        printf("USAGE: ./vm_inspector pid va_begin va_end\n");
        return -1;
    }

    if (begin_vaddr > end_vaddr)
    {
        printf("INVALID BEGIN_VADDR AND END_VADDR\n");
        return -1;
    }

    pid = atoi(argv[1]);

    addr = mmap(0, PGD_COUNT * PAGE_SIZE, PROT_READ,
    MAP_SHARED|MAP_ANONYMOUS, -1, 0);

    if (addr == MAP_FAILED)
    {
        printf("ERROR: %s\n", strerror(errno));
        return -1;
    }

    page_table_addr = (unsigned long) addr;

    fake_pgd_addr = mmap(0, PGD_PAGE_COUNT * PAGE_SIZE,
    PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    if (fake_pgd_addr == MAP_FAILED)
    {
        printf("ERROR: %s\n", strerror(errno));
        goto free_addr;
    }

    fake_pgd = (unsigned long) fake_pgd_addr;

    if (syscall(379, pid, fake_pgd, page_table_addr, begin_vaddr,
        end_vaddr))
    {
        printf("ERROR: %s\n", strerror(errno));
        goto free_all;
    }

    fake_pgd_new = (unsigned long **) fake_pgd_addr;

```

```
117         index = pgd_index(page_table_addr);
118
119         printf("================== Beginning Dump ==================\n");
120
121         for (i = 0; i < PGD_COUNT; i++)
122         {
123             if (fake_pgd_new[i] != NULL)
124             print_pte_table(fake_pgd_new[i], i, begin_vaddr, end_vaddr);
125         }
126
127         munmap(addr, PGD_COUNT * PAGE_SIZE);
128         munmap(fake_pgd_addr, PGD_PAGE_COUNT * PAGE_SIZE);
129
130     free_addr:
131         munmap(addr, PGD_COUNT * PAGE_SIZE);
132         return -1;
133
134     free_fake_pgd:
135         munmap(fake_pgd_addr, PGD_PAGE_COUNT * PAGE_SIZE);
136         return -1;
137
138     free_all:
139         munmap(addr, PGD_COUNT * PAGE_SIZE);
140         munmap(fake_pgd_addr, PGD_PAGE_COUNT * PAGE_SIZE);
141         return -1;
142
143         return 0;
144     }
```

Outcome:

In this part, **only one example result will be shown**.

The first column is the index, the second is the virtual address and the last is the physical address.

Fig.3.1 shows the test result with invalid inputs

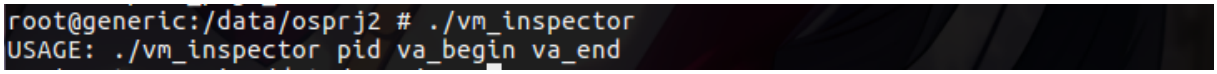


Figure 3.1: Outcome of invalid inputs.

Fig.3.2 shows the test result with valid dump from 0x00000000 to 0x0000ffff (pid = 1)

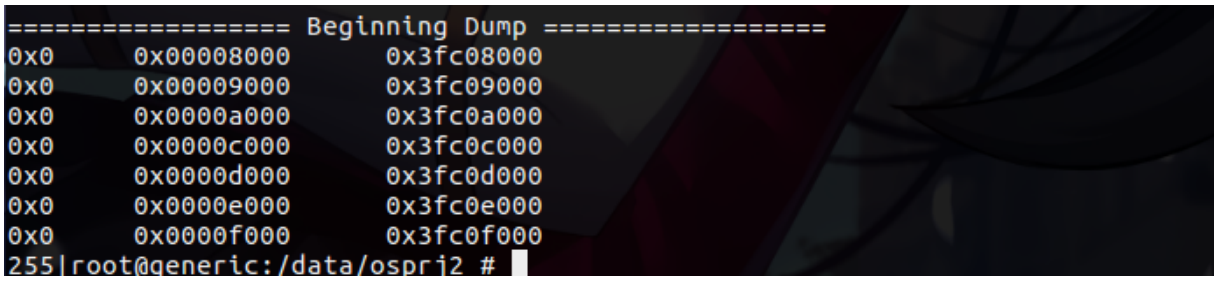


Figure 3.2: Outcome of 0x00000000 to 0x0000ffff.

Investigation:

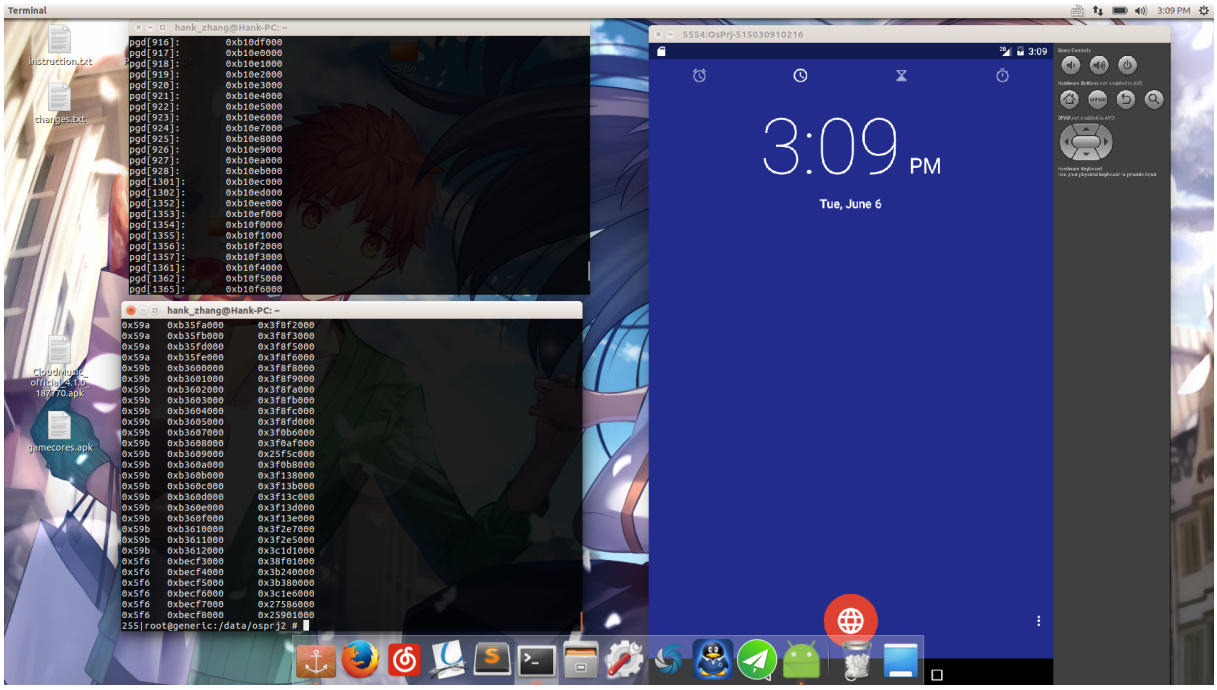


Figure 3.3: Screen shot of zygote page table investigation

At first I tried to install some apps I always use on my phone on the emulator. However, I found it difficult since all the tools we got is adb command line in Linux terminal only. Moreover, the Android version of our emulator is too high for many of current popular apps to run on perfectly.

Thus, I turned to focus on some system processes. Using the vm_inspector I dumped the page tables for several system processes belonging to : System calendar, System Clock, System settings and Zygote in the end.

For the first three system applications, I printed their page table out in the terminal and played with them, then investigate their pages again and compared them with the original outcome. I found the some new pages in the second time's result. Since I added several alarm clocks and changed some system settings, the appearance of new pages is completely logically consistent with what we've learnt during the lecture.

For the Zygote, I searched its function on google and it was said that "A zygote process is one that listens for spawn requests from a master process and forks itself in response. Generally they are used because forking a process after some expensive setup has been performed can save time and share extra memory pages."

In short words, Zygote manages many duplicate requests for the same physical resources. So, we should be able to find some common physical addresses in application's pages and Zygote pages.

And I succeeded. I found a great many of libraries that are shared across processes originate from Zygote for their pages all point to the same physical frames. The following are some of them:

/system/lib/libjnigraphics.so

/system/lib/libjpeg.so

/system/lib/liblog.so

/system/lib/libmedia
/system/lib/libmedia.so
/system/lib/libmemtrack.so
/system/lib/libpng.so
/system/lib/libpowermanager.so
/system/lib/librs

4 Problem 4

4.1 Implement new Linux page replacement algorithm

Problem Description:

Linux implements an Approximate LRU Algorithm as page replacement algorithm. All pages in a zone are split into two parts according to whether they were referenced recently. Tow lists, active_list and inactive_list, are used to store the page status in a zone.

In this project, you need to change the page replacement algorithm. You should add a new referenced variable to record the last time when this page was referenced. If a page is referenced by process, the referenced value of it should be set to 0. Otherwise, the referenced value increases 1 for every period until it is referenced. You should check these two lists periodically, move the page, whose reference vale is 0, to active list, and move the page, whose referenced value larger than a threshold that defined by yourself to inactive list.

Source Code:

mm_type.h

```
1     struct page {
2         //my code
3         unsigned long my_PG_ref;
4         /* First double word block */
5         unsigned long flags;         /* Atomic flags, some possibly
6         * updated asynchronously */
7         struct address_space *mapping;  /* If low bit clear, points to
```

vm_scan.c

```
1     static void shrink_active_list(unsigned long nr_to_scan,
2         struct mem_cgroup_zone *mz,
3         struct scan_control *sc,
4         int priority, int file)
5     {
6         ......
7         //my code
8         //if the pafe_ref is smaller than the threshold, keep it in the
              active list
9         if(++(page->my_PG_ref) <= 2)        //threshold
10        {
11            list_add(&page->lru, &l_active);
12            continue;
13        }
14
15        ClearPageActive(page);  /* we are de-activating */
16        list_add(&page->lru, &l_inactive);
17        printk("%ld ===> inactive list (threshold)\n",page->index);
18
19        ......
20    }
21    ......
22    static enum page_references page_check_references(struct page *page,
23        struct mem_cgroup_zone *mz,
24        struct scan_control *sc)
25    {
26        referenced_ptes = page_referenced(page, 1, mz->mem_cgroup, &
              vm_flags);
27        //referenced_page = TestClearPageReferenced(page);
28        //my code
29        referenced_page = (page->my_PG_ref)++;
30        printk("%ld: ref + 1\n",page->index);
31        ......
32        if (referenced_ptes) {
33        {
```

```
            page->my_PG_ref--;

            if (referenced_page < 1 || referenced_ptes > 1)
            return PAGEREF_ACTIVATE;
        }
        ......
        if (referenced_page < 1 && !PageSwapBacked(page))
        return PAGEREF_RECLAIM_CLEAN;
        ......
    }
```

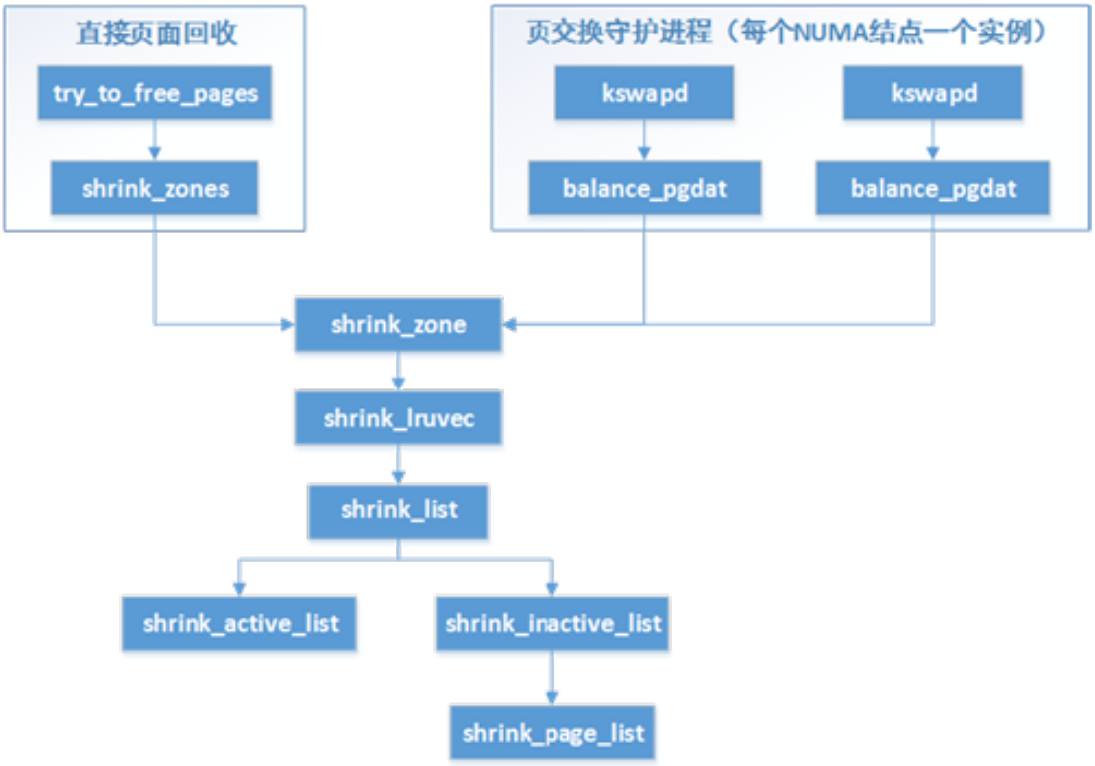
swap.c

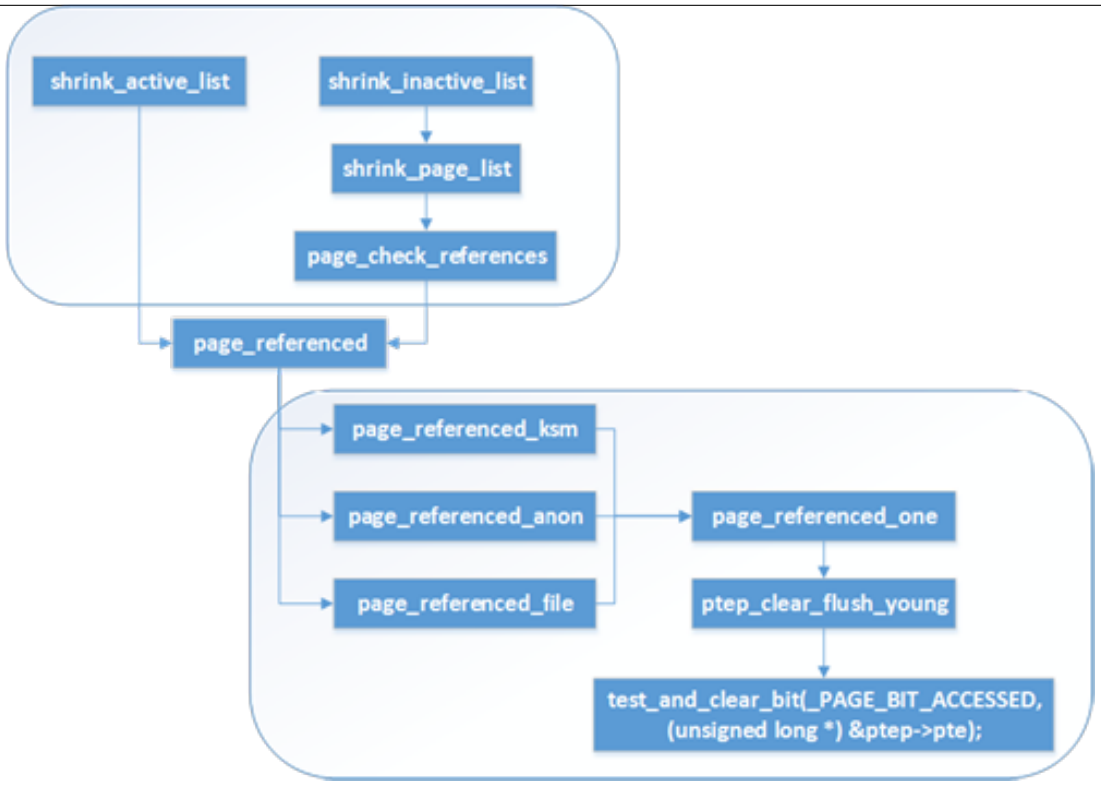
```
        void mark_page_accessed(struct page *page)
        {
            //my code
            page->my_PG_ref = 0;
            //too many info printed
            //printk("%ld ===> active list\n",page->index);
            if (!PageActive(page) && !PageUnevictable(page) && PageLRU(
                page))
            {
                activate_page(page);
            }
        }
```

Analysis:

Linux defines a struct called page to store all the information about one page. In order to implement our new algorithm, we are ought to add one more attribute to it(my_PG_ref)

Before we start to change the Linux page replacement algorithm, we have to look into it's structure to determine which functions should be modified. The following two pictures give a well-organized explanation.





From this, we know that, on a very base level, the Linux page replacement uses shrink_active_list and shrink_inactive_list to organize pages. Furthermore, the key function in these two is page_referenced.

Thus, we have to go down and down to the shrink_active_list and page_check_references to add our page replacement methods, and block the original one.

In mark_page_accessed, which sets my_PG_ref to zero each time a page is referred.

In shrink_active_list, we age corresponding page's my_PG_ref by one each time and compare it with the threshold (2 in the sample code). If it exceeds the threshold, put the page into inactive list.

In check_page_references, we age corresponding page's my_PG_ref by one each time and temporarily decrease it back to the old value to check whether it has been referred. If referred, return PAGEREF_ACTIVAT to put it into active_list.

Outcome:

For this part, I wrote a testing program which would occupy 1 GB space of our emulator theoretically. In fact, we only had to claim space from the system large enough to invoke the kswapd for calling page replacement algorithm.

As it should be, the testing program was usually killed by the system for too large space allocation causing a system panic.

The following screen shot shows the successful invoke of my new page replacement algorithm.

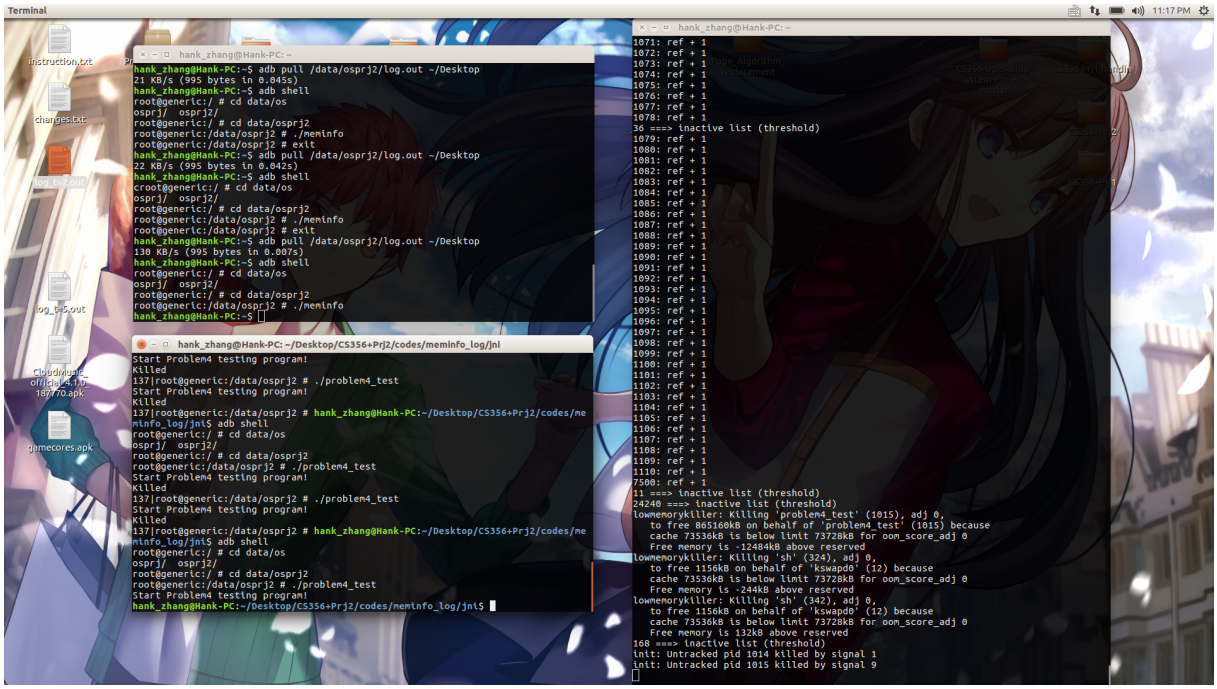


Figure 4.1: Screen shot of my page replacement algorithm.

Furthermore, I write a program to keep tracking the active and inactive list space consuming information, recorded into threshold corresponding log files.

Here comes the list of all my log files:

log_raw.out

log_t=1.out

log_t=2.out

log_t=5.out

log_t=10.out

lograw.out is the original outcome from the Linux page replacement algorithm.

Following figure gives an example of how the log file's structure looks like.

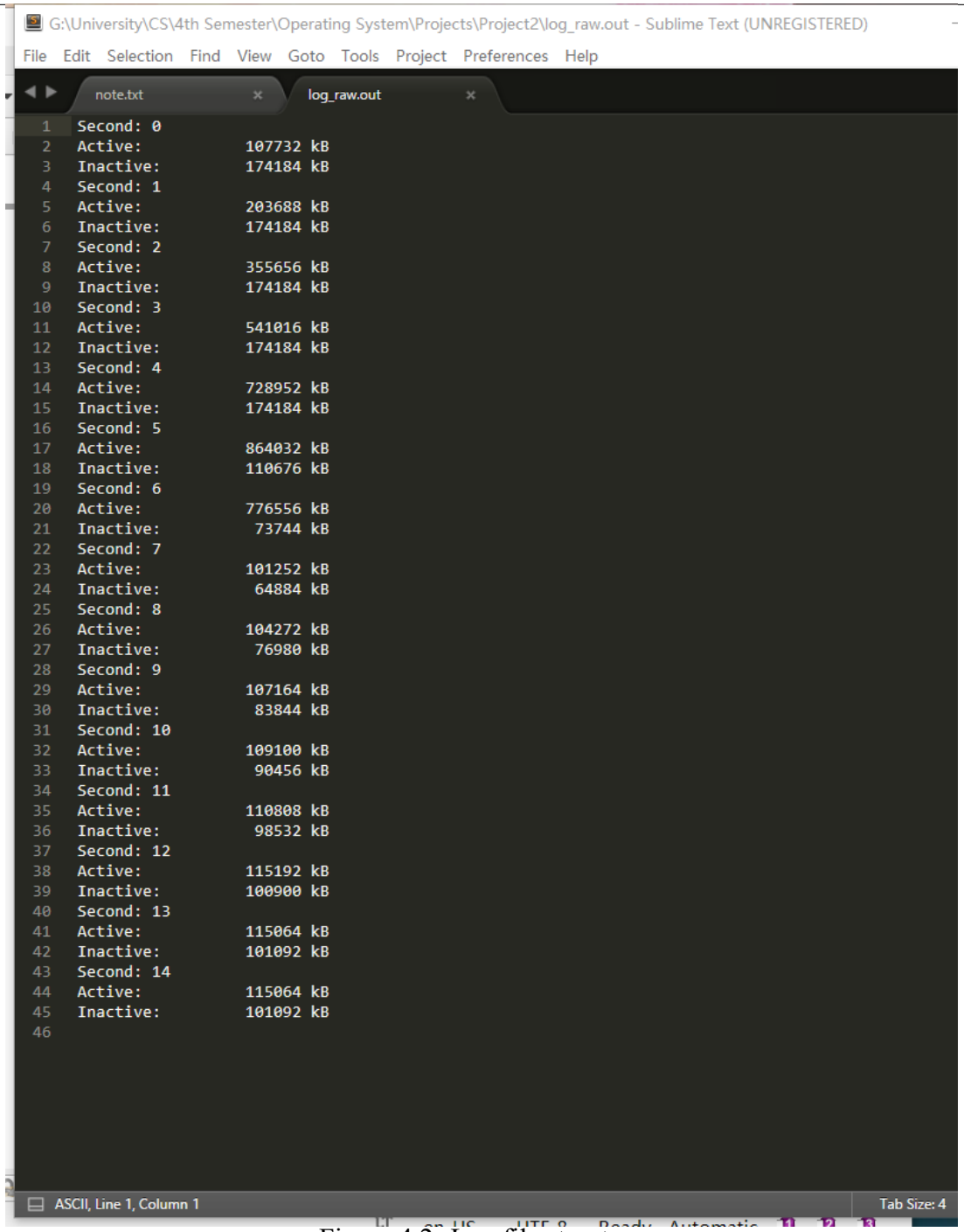


Figure 4.2: Log file structure.

For all the test result: origin Linux page replacement algorithm, my page replacement algorithm with threshold = 1/2/5/10, I created 3 line graphs to compare their outcome, with active page space, inactive page space and total page space being the y-axis while time being the x one.

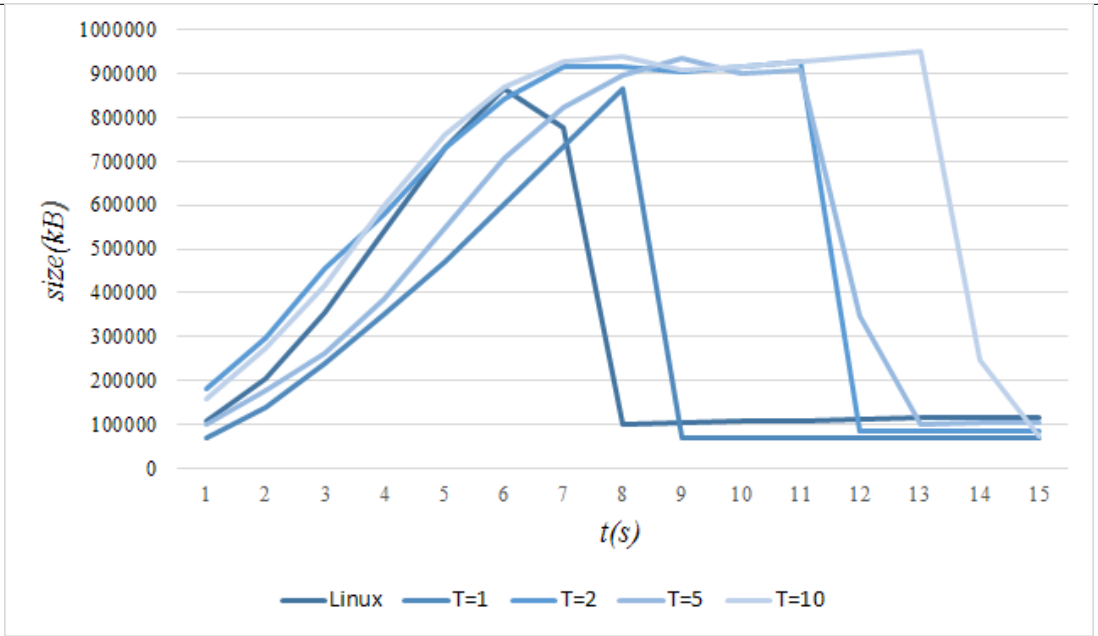


Figure 4.3: Active page list size-t graph

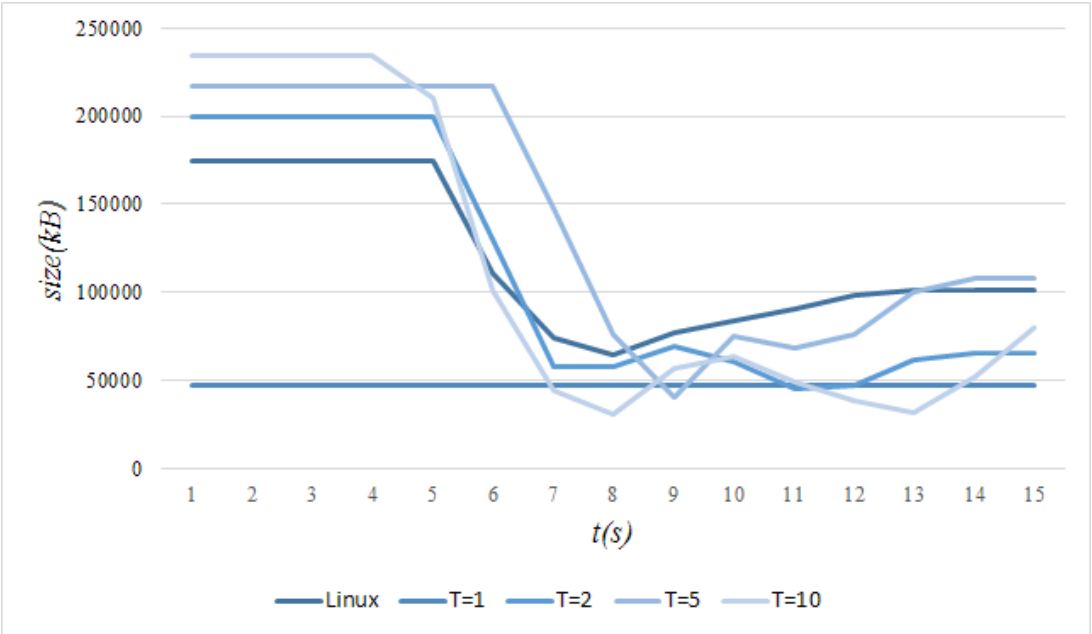


Figure 4.4: Inactive page list size-t graph

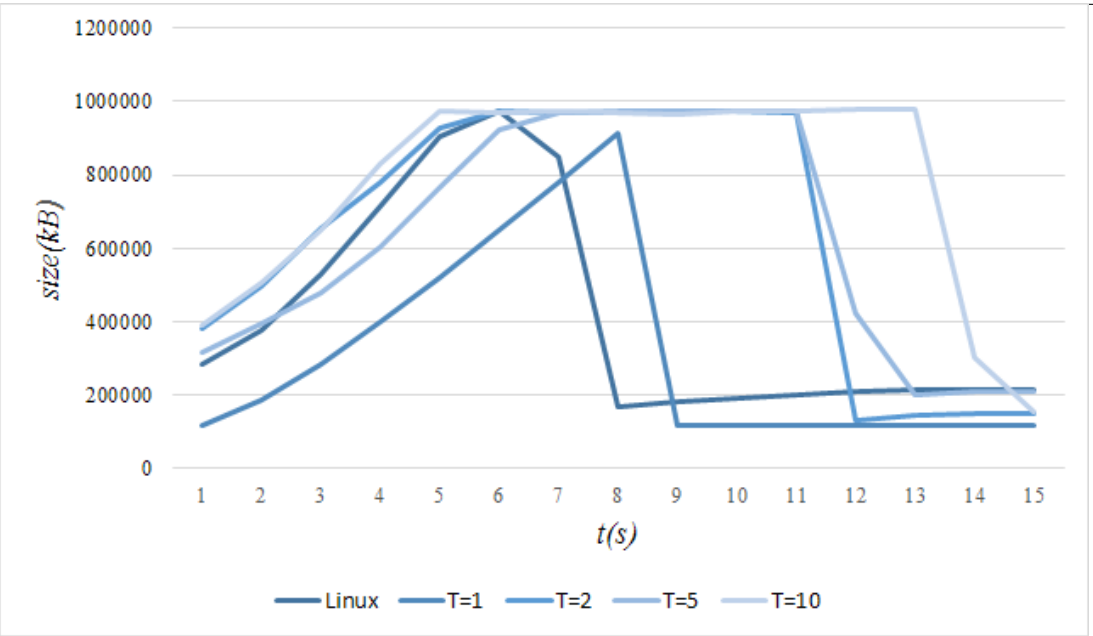


Figure 4.5: Total page list size-t graph

Personal Thinking: Since Linux uses an approximate combinational algorithm of LRU and second chance, the page faults should be comparably low.

From the patterns above, we are able to claim that, the Linux page replacement is the most stable one and efficient one.

In figure 4.3, it is shown that the original Linux page replacement algorithm manages the active page list in a most efficiently way since it's memory size of active list began to decrease first when lacking memory. Meanwhile, the larger threshold in my algorithm is, the more time system can tolerate huge memory occupation. There comes two explanation for this, one is changed algorithm runs less efficiently so it takes more time to reclaim pages when encounter huge memory occupation, the other is the changed algorithm increases the max size of active list. Since process being killed will take place on the new kernel, I prefer to choose the former one as the reason.

When coming to inactive list management in figure 4.4, there is not that much difference between my algorithm and the original one because both of them only give one chance for pages in inactive list to be activated instead of being reclaimed. One thing should be notice is the strange pattern of T=1 algorithm, I think it is not accurate nor there existed some hidden factors contributed to it.

Last but not least, in figure 4.5, it is clear that when T=1, our new algorithm performs in the highest similarity with the original one because T=1 is to some extend the same with second chances algorithm performed by real Linux kernel. However, I blocked the LRU together with second chance, resulting in the output difference in the end. In a word, I would choose T=1 to be the best threshold in my algorithm.