CS356: Operating System Project

Report for Project 2

Android Memory Management

Name: Ziteng Yang

This report contains how I implemented required program and functions, and the result of running and testing. I also add some note when studying the Linux source code here, regarding it as part of "detail", and if its not required, please just skip it.

1. **Problem 1: Compile the Kernel**

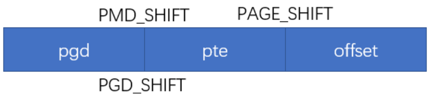
   This problem has nothing to with technological knowledge, since I just need to follow the instruction step by step to configure environment, and enter "make -j4" at the terminal in kernel file's location. In fact, this is just a preparation for the following 3 problem.

2. **Problem 2: Map a Target Process's Page Table**

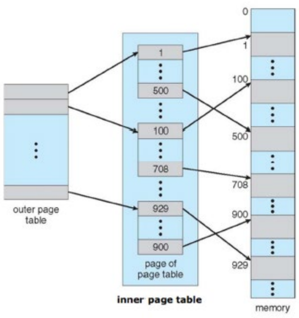
   2.1. Description

   In the Linux kernel, the page table is broken into multiple levels. Address of a system with a 4-level page table is as follows:

   The system of my 32-bit android virtual devices has a 2-level page table, which means pud=pmd=0 (found in implementation):



   So the page table just has the following structure:



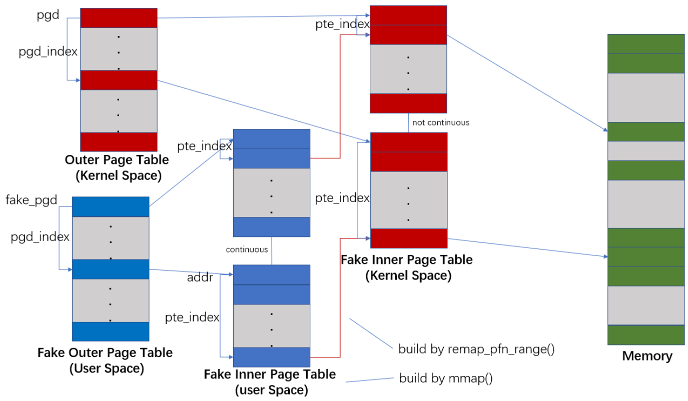
   Out goal is to map this structure from kernel space into user space with our own system call. In another word, given a pid of some process A and some virtual address, I need to use my own process "VATranslate" to translate the virtual address into physical address. To complete the mission, I need to use my system call to build my own outer page table and inner page table in my process of user space, while accessing the inner table just gives us the physical address of target process.

   Related files: syscall.c(wrote by my self), VATranslate.c(wrote by myself), pagewalk.c (kernel file changed a little), mm,h (kernel file changed a little)

2.2. Implementing Details

2.2.1. In general

Since I need to translate the virtual address right, I cannot just apply for a large number of memory and copy the table from kernel space to user space, but need to access the inner page table (pte) directly in read-only mode, like the following figure:



The relation built by remap_pfn_range will be discussed later.

2.2.2. Implementation of user program: VATranslate.c

The parameter reading is very easy:

```c
int main(int argc,char **argv)
{
    //unsigned long *table_addr;
    //unsigned long *fake_pgd_addr;
    //unsigned long pgd_ind,phy_addr;
    //unsigned long *phy_base;
    printf("------------------------------------------------\n");
    printf("VATranslate\n\n");
    if(argc!=3)
    {
        printf("argument unmathed!\n");
        return -1;
    }

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

Firtly we need to investigate the page table layout as required, so I invoke a system call "get_pagetable_layout" (implemented in syscall.c), and just pass the struct to it and get the answer:

```c
struct pagetable_layout_info{
    uint32_t pgdir_shift;
    uint32_t pmd_shift;
    uint32_t page_shift;
};
```

```c
    //first system call: get_pagetable_layout
    if(!syscall(356,&layout_info,sizeof(struct pagetable_layout_info)))
    {
        printf("pgdir_shift: %d\tpmd_shift: %d\tpage_shift: %d\n",
        layout_info.pgdir_shift, layout_info.pmd_shift, layout_info.page_shift);
    }
    else
    {
        printf("System call to get pagetable layout infomation failed!\n ");
    }
```

Now I can build my fake page table. Firtly I need to allocate memory to fake outer pagetable. Since this need a smaller scale of array, I can just use malloc:

```
    //allocate memory for fake pgd
    fake_pgd=malloc(sizeof(unsigned long)*page_size);

```

Next is to allocate memory for fake inner page table, as the instruction says, "it's a bad idea to use malloc to prepare a memory section for mapping page tables, because malloc cannot allocate memory more than MMAP_THRESHOLD (128kb in default). Instead you should consider to use the mmap system call":

```
    //this allocates virtual memory
    page_table_addr=mmap(NULL,
                         1<<22,
                         PROT_READ | PROT_WRITE,
                         MAP_SHARED | MAP_ANONYMOUS,
                         -1,
                         0);

    if(!page_table_addr||!fake_pgd)
    {
        printf("allocate memory failed!\n");
        return -1;
    }
```

This system call builds a large memory area, but its virtual memory. Use the function remap_pfn_rang() in system call "expose_page_table" can build a mapping relation of fake inner page table and the "real" inner page table:

```
    int err=syscall(357,pid,fake_pgd,0,page_table_addr,begin_vaddr,begin_vaddr+1);
    //error handler.
    switch (err)
    {
        case 1:
            printf("expose_page_table: address boundary error!\n");
            return 1;

        case 2:
            printf("expose_page_table: failed to find pid!\n");
            return 2;

        case 3:
            printf("expose_page_table: failed to copy fake pgd!\n");
            return 3;
        case 4:
            printf("expose_page_table: kmalloc error!\n");
            return 4;
        case 5:
            printf("target process has no vm area!\n");
            return 5;
        case 6:
            printf("walk_page_range failed!\n");
    default:
        break;
    }
```

As we only translate one address, we only need a small interval between begin address and end address, so assigned end_addr=begin_addr+1;

After the system call, if it works (assume it does), we should be able to visit the real inner page table (PTE entry) through fake_pgd;

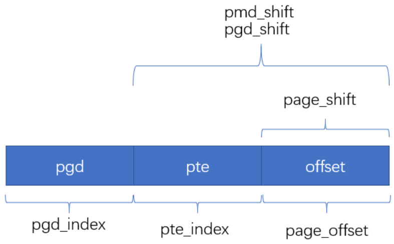
So we need to calculate pgd_index and pte_index at first:

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

```

```
    const unsigned long page_size=1<<(layout_info.page_shift);
    const unsigned long pte_size = 1<<(layout_info.pmd_shift-layout_info.page_shift);
    const unsigned long pgd_size = 1<<(32-layout_info.pgdir_shift);

```

```
    unsigned long pgd_index=pgd_index(begin_vaddr,layout_info);
    unsigned long pte_index=pte_index(begin_vaddr,layout_info);
    unsigned long page_offset = begin_vaddr & 0x0fff;
```

This calculation formula is according to the following structure:



Finally we can access the page table and translate the virtual address:

```
    //get the entry in the table.
    unsigned long frame=p[pte_index];

    frame &= ~(page_size-1);      //mask higher bit
    if(!frame)
    {
        printf("target virtual address is not in memory!\n");
        return -4;
    }

    //add the offset.
    unsigned long physical_address = frame + page_offset;
    printf("Virtual address: 0x%08lx\t", begin_vaddr);
    printf("Physical address: 0x%08lx\n", physical_address);

```

Before returning we need to free the memory allocated, otherwise it would cause memory leak.

2.2.3. Implementation of system call: syscall.c

This file in fact implemented two system call: get_pagetable_layout() and expose_page_table().

The first system call is for investigating the page table layout. It just pass the 3 desired parameter from kernel to user but nothing else:

```
static int get_pagetable_layout(struct pagetable_layout_info __user *pgtbl_info,
                                int size)
{
    printk("________________________________________\n");
    printk("start get_pagetable_layout() system call...\n");

    if(sizeof(struct pagetable_layout_info)!=size)
    {
        printk("struct size unmathed!\n");
        return -1;
    }

    struct pagetable_layout_info tmp;
    tmp.pgdir_shift=PGDIR_SHIFT;
    tmp.pmd_shift=PMD_SHIFT;
    tmp.page_shift=PAGE_SHIFT;

    if(copy_to_user(pgtbl_info,&tmp,sizeof(struct pagetable_layout_info)))
    {
        printk("Error copying page table layout to user!\n");
        return -2;
    }
    printk("end get_pagetable_layout() system call...\n");
    printk("________________________________________\n");
    return 0;
}

```

The second system call is the core and most difficult part. I need to build the mapping relationship here. It receive parameters including pid, begin and end address, the address of fake outer page table and fake inner page table.

Firstly, we need to find the task_struct type of target process, according to pid, using some existing function:

```c
struct pid* current_pid=find_get_pid(pid);
if(!current_pid)
{
    printk("failed to find pid!\n");
    return 2;
}
struct task_struct *current_task=get_pid_task(current_pid,PIDTYPE_PID);
if(!current_task)
{
    printk("error finding task_struct!\n");
    return 2;
}
printk(KERN_INFO" pid: %d\tname: %s",current_task->pid,current_task->comm);
```

Then we need to apply for a memory of kernel space to store the outer page table, since we already assume the outer page table won't change for simplicity.

```c
//istruct to carry when walking the page
struct myPrivate
{
    unsigned long *fake_pgd_base;
    unsigned long pte_base;
};
```

```c
my_private.fake_pgd_base=kcalloc(PAGE_SIZE,sizeof(unsigned long),GFP_KERNEL);
if(!my_private.fake_pgd_base)
{
    printk("kcalloc error!\n");
    return 4;
}
my_private.pte_base=page_table_addr;
```

Later it would be copied to user.

The newly defined struct myPrivate was used to be carried into the calling of walk_page_range function. This function we defined in mm/pagewalk.c . It can recursively walk the page table for the memory area in a VMA, calling supplied callbacks. Callbacks are called in-order (first PGD, first PUD, first PMD, first PTE, second PTE... second PMD, etc.). If lower-level callbacks are omitted, walking depth is reduced. If any callback returns a non-zero value, the walk is aborted and the return value is propagated back to the caller. Otherwise 0 is returned. walk->mm->mmap_sem must be held for at least read if walk->hugetlb_entry is not NULL.

```c
 */
int walk_page_range(unsigned long addr, unsigned long end,
            struct mm_walk *walk)
{⋯
}

EXPORT_SYMBOL(walk_page_range);
```

The struct mm_walk is defined in linux/mm.h:

```c
/**
 * mm_walk - callbacks for walk_page_range
 * @pgd_entry: if set, called for each non-empty PGD (top-level) entry
 * @pud_entry: if set, called for each non-empty PUD (2nd-level) entry
 * @pmd_entry: if set, called for each non-empty PMD (3rd-level) entry
 *         this handler is required to be able to handle
 *         pmd_trans_huge() pmds.  They may simply choose to
 *         split_huge_page() instead of handling it explicitly.
 * @pte_entry: if set, called for each non-empty PTE (4th-level) entry
 * @pte_hole: if set, called for each hole at all levels
 * @hugetlb_entry: if set, called for each hugetlb entry
 *         *Caution*: The caller must hold mmap_sem() if @hugetlb_entry
 *               is used.
 *
 * (see walk_page_range for more details)
 */
struct mm_walk {
    int (*pgd_entry)(pgd_t *, unsigned long, unsigned long, struct mm_walk *);
    int (*pud_entry)(pud_t *, unsigned long, unsigned long, struct mm_walk *);
    int (*pmd_entry)(pmd_t *, unsigned long, unsigned long, struct mm_walk *);
    int (*pte_entry)(pte_t *, unsigned long, unsigned long, struct mm_walk *);
    int (*pte_hole)(unsigned long, unsigned long, struct mm_walk *);
    int (*hugetlb_entry)(pte_t *, unsigned long,
                unsigned long, unsigned long, struct mm_walk *);
    struct mm_struct *mm;
    void *private;
};
```

It has several function pointer, which woud be called every time the walk_page_range() enter a pgd entry (calling walk->pgd()), a pud entry (calling walk->pud()), ···

Since our purpose is to remap pte table to user space, and the system has a 2-level page table, we need to accomplish the procedure every time we get into pgd_entry, and remap the responding pte table touser program.

This structure also contains a point to the self-defined struct, so that we can carry some useful variable to finish the walk.

Before walking the page, we need to initialize the mm_walk variable walk:

```
    my_private.pte_base=page_table_addr;

    walk.pgd_entry=&my_pgd_entry;//my_pgd_entry;
    walk.pud_entry=NULL;//my_pud_entry;
    walk.pmd_entry=NULL;//my_pmd_entry;    //my pmd entry function;
    walk.pte_entry=NULL;//my_pte_entry;
    walk.pte_hole=NULL;
    walk.hugetlb_entry=NULL;
    walk.mm=current_task->mm;
    walk.private=&my_private;

```

As we won't need other function, only carry one so that it can finish the remap procedure. The function was defined as following:

```
int my_pgd_entry(pmd_t *pgd,unsigned long addr,unsigned long next,struct mm_walk *walk)
{

    unsigned long pgd_index=pgd_index(addr);

    //get the physical frame number.
    unsigned long pfn = page_to_pfn(pmd_page((unsigned long)*pgd));
    if(pgd_none(*pgd)||pgd_bad(*pgd)||!pfn_valid(pfn))
    {
        printk("failed to find pfn!\n");
        return 0;
    }
    printk(KERN_INFO"pfn:%08X\n",pfn);

    struct myPrivate *base=walk->private;

    //struct vm_area_struct *vma=current->mm->mmap;
    struct vm_area_struct* vma = find_vma(current->mm, base->pte_base);
    if(!vma)
    {
        printk("find_vma error!\n");
        return 0;
    }


    down_write(&current->mm->mmap_sem);
        int err=remap_pfn_range(vma,base->pte_base,pfn,
                                PTE_SIZE*sizeof(unsigned long),
                                vma->vm_page_prot);
    up_write(&current->mm->mmap_sem);

    //remap: can find pte_base according to pgd_base and pmd_index
    //fake_pgd_base + (pgd_index * sizeof(each_entry))
    //you will either get a null for non-exist pmd or the address of a fake pmd
    //pgd_index and pmd_index are the same here since only 2-level in this 32-bit OS
    base->fake_pgd_base[pgd_index] = base->pte_base;
    //you can get the remapped address of a Page Table
    //by reading the content at the address
    //fake_pmd_base + (pmd_index * sizeof(each_entry))
    base->pte_base += PTE_SIZE;
    return 0;
}
```

In this function, we find current process's virtual memory area's vm_area_struct variable, and target page table's page frame number (PTE) and use the function remap_pfn_range() to remap the PTE to user space, each time an area of PTE_SIZE*sizeof(unsigned long) .

Since this function was called each time walk_page_range enter a pgd, the whole page table of target interval was remapped into user space.

Note: To successfully use function walk_page_range(), we need to add two sentences at mm/pagewalk.c , and set it as external at linux/mm.h:

```c
#include <linux/mm.h>
#include <linux/highmem.h>
#include <linux/sched.h>
#include <linux/hugetlb.h>
#include <linux/export.h>
```

```c
 */
int walk_page_range(unsigned long addr, unsigned long end,
		struct mm_walk *walk)
{…
}

EXPORT_SYMBOL(walk_page_range);
```

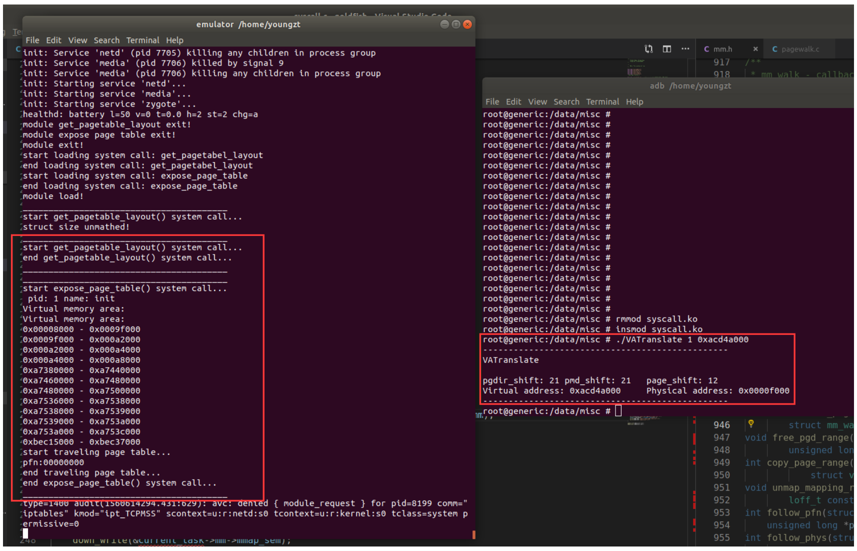
```c
 *		is used.
 *
 * (see walk_page_range for more details)
 */
struct mm_walk {
	int (*pgd_entry)(pgd_t *, unsigned long, unsigned long, struct mm_walk *);
	int (*pud_entry)(pud_t *, unsigned long, unsigned long, struct mm_walk *);
	int (*pmd_entry)(pmd_t *, unsigned long, unsigned long, struct mm_walk *);
	int (*pte_entry)(pte_t *, unsigned long, unsigned long, struct mm_walk *);
	int (*pte_hole)(unsigned long, unsigned long, struct mm_walk *);
	int (*hugetlb_entry)(pte_t *, unsigned long,
			     unsigned long, unsigned long, struct mm_walk *);
	struct mm_struct *mm;
	void *private;
};

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

(Thanks for the discussion in our course's Wechat group between Ruizhen Chen and Jinwei Xi.)

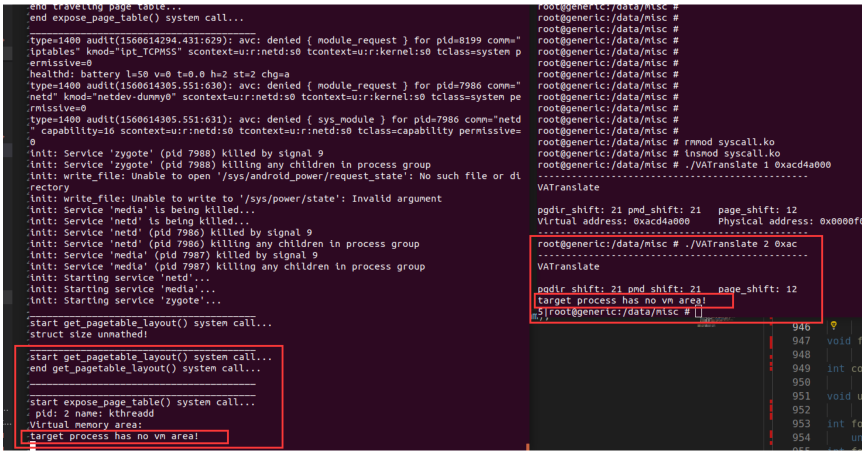
2.3. Result

Following was some screen captures of running program VATranslate:

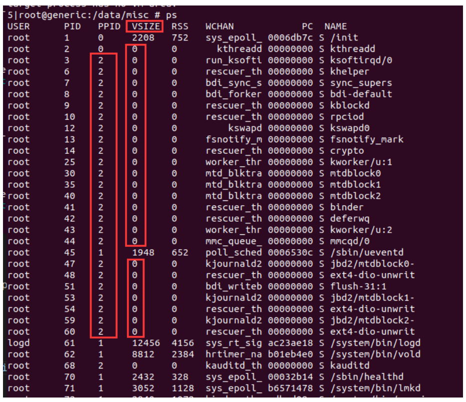
Test the “init” process (pid = 1):



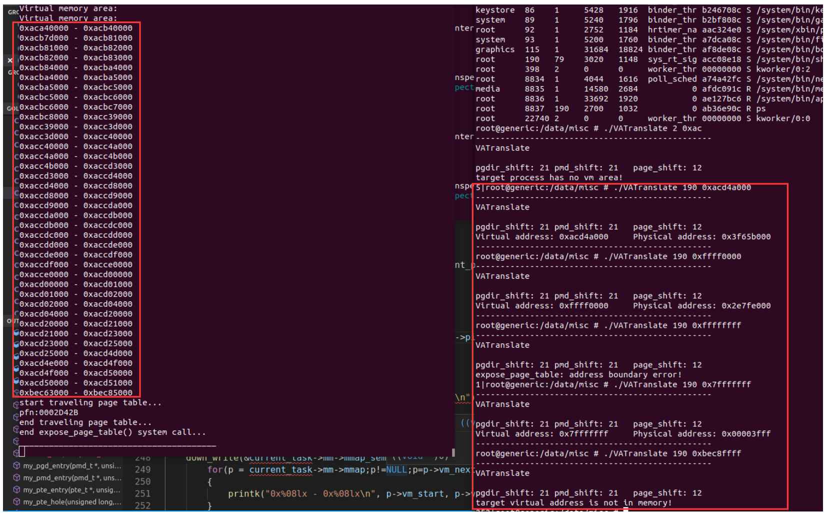
Test the “kthreadd” process (pid =2).



Note: we can see from ps command that it and all child process of it has no virtual memory area, therefore our system call must check this, otherwise “kernel panic” might occur if it tries to access a empty pointer:



Test “sh” process (pid=190 when tested):



3. Problem 3: Investigate Android Process Address Space

3.1. Description

This problem was almost the same as the VATranslate program, but we need to translate a range of virtual address.

We also need to discover some details about zygote

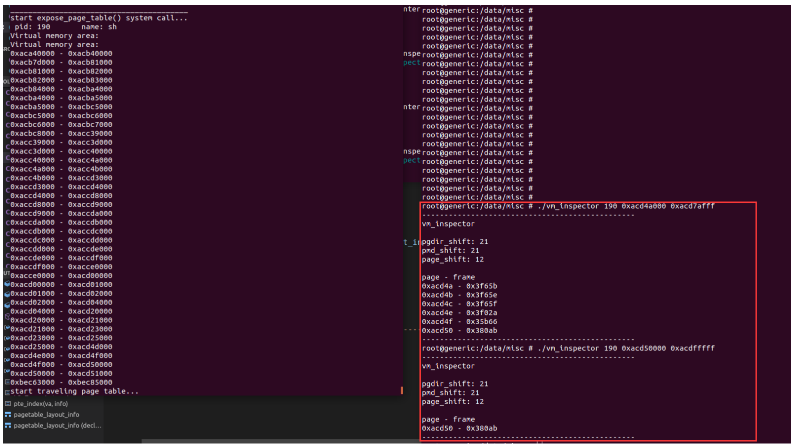
3.2. Implementing Details

Most part of it was the same as VATranslate.c, the main difference was that we need a for-loop to check and translate valid virtual address.

```
    printf("\npage - frame \n");
    unsigned long i;
    for(i=begin_vaddr>>layout_info.page_shift;i< end_vaddr>>layout_info.page_shift;++i)
    {
        unsigned long pgd_index=pgd_index(i<<layout_info.page_shift,layout_info);
        unsigned long pte_index=pte_index(i<<layout_info.page_shift,layout_info);
        unsigned long* p=fake_pgd[pgd_index];
        if(p)
        {
            unsigned long frame=p[pte_index];
            if(frame>>layout_info.page_shift)
            {
                printf("0x%lx - 0x%lx\n",i,frame>>layout_info.page_shift);
            }
        }
    }

```

3.3. Result

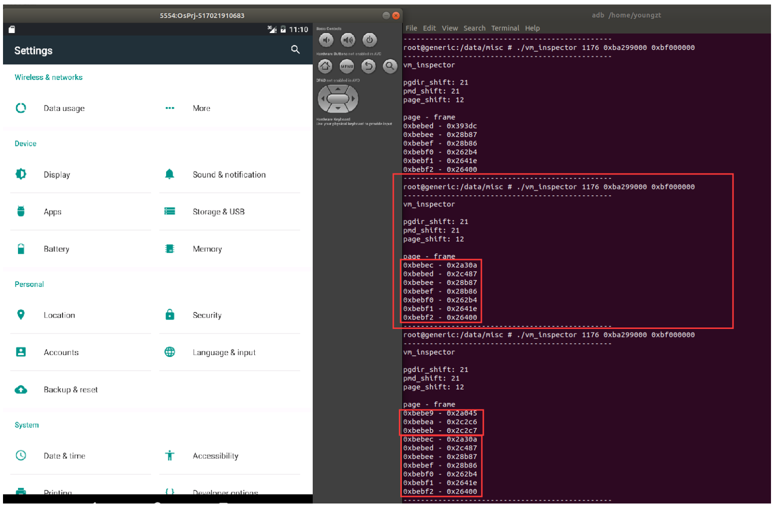
Following was a simple test of vm_inspector. The detail was discussed at 3.4.



3.4. Discovery

3.4.1. Dump page table twice while playing an app

I tried with "com.android.settings". When I open it and dump its VMA again, I found in the same interval, there is 3 more virtual page occurred:



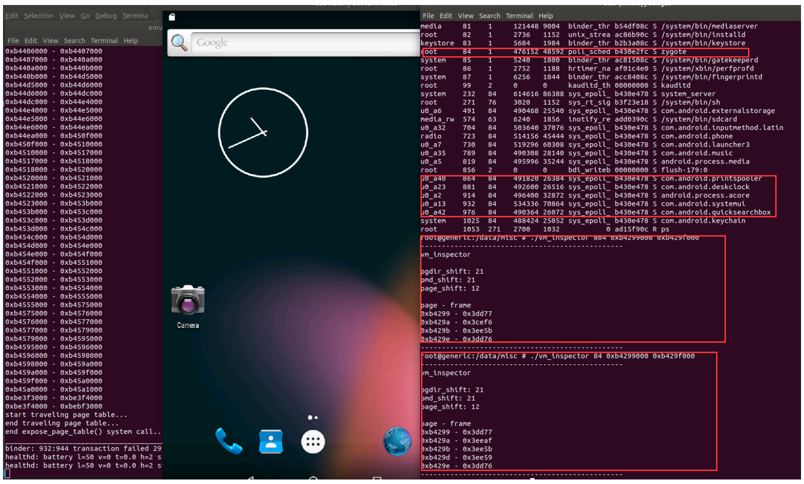
This implies that when running an app, its virtual memory area are always changing.

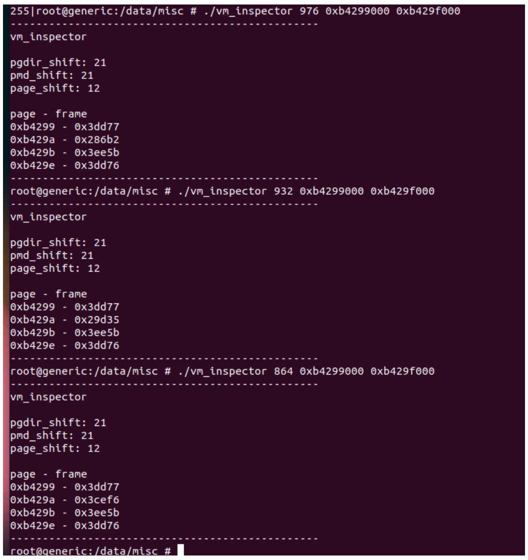
3.4.2. About Zygote and other Android app

One thing need to be notice is that, when load new system call, we need to choose carefully which old system call to be replaced. Initially I choose 356 and 357 just like project 1, but when I came to problem 3, I found that zygote was killed again and again, and the test is hard to continue. I spend a long time to discover that its initialization need original system call 356 and 357, so I need to consider others. I also tried 233 and 666, and other mysterious things also happened… Finally someone told me 380 and 381 migth work, so I changed it and it didn's go wrong.

We can easily search some material about the zygote process. It's child process of init. All process named "com.*" was forked from it.

The following figures shows some relationship with zygote process and other process. Note when doing this test this, pid of zygote didn't change and remain 84, so did its child process.

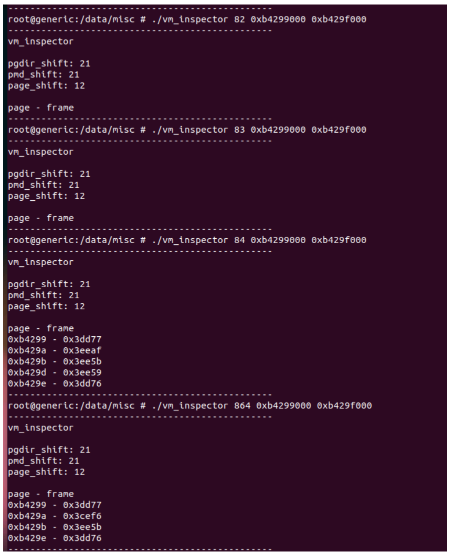




```
255|root@generic:/data/misc # ./vm_inspector 976 0xb4299000 0xb429f000
-----------------------------------------------
vm_inspector

pgdir_shift: 21
pmd_shift: 21
page_shift: 12

page - frame
0xb4299 - 0x3dd77
0xb429a - 0x286b2
0xb429b - 0x3ee5b
0xb429e - 0x3dd76
-----------------------------------------------
root@generic:/data/misc # ./vm_inspector 932 0xb4299000 0xb429f000
-----------------------------------------------
vm_inspector

pgdir_shift: 21
pmd_shift: 21
page_shift: 12

page - frame
0xb4299 - 0x3dd77
0xb429a - 0x29d35
0xb429b - 0x3ee5b
0xb429e - 0x3dd76
-----------------------------------------------
root@generic:/data/misc # ./vm_inspector 864 0xb4299000 0xb429f000
-----------------------------------------------
vm_inspector

pgdir_shift: 21
pmd_shift: 21
page_shift: 12

page - frame
0xb4299 - 0x3dd77
0xb429a - 0x3cef6
0xb429b - 0x3ee5b
0xb429e - 0x3dd76
-----------------------------------------------
root@generic:/data/misc #
```

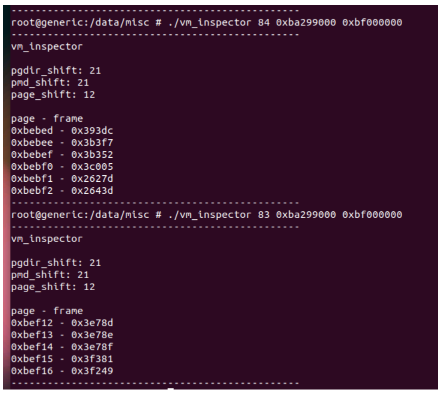
We can see that all process who's parent was zygote same to share the same page table, i.e. using the same virtual address and physical address.

What about other app? We can see that in the following figure, only zygote and its child process are mapped in this area above process tested.



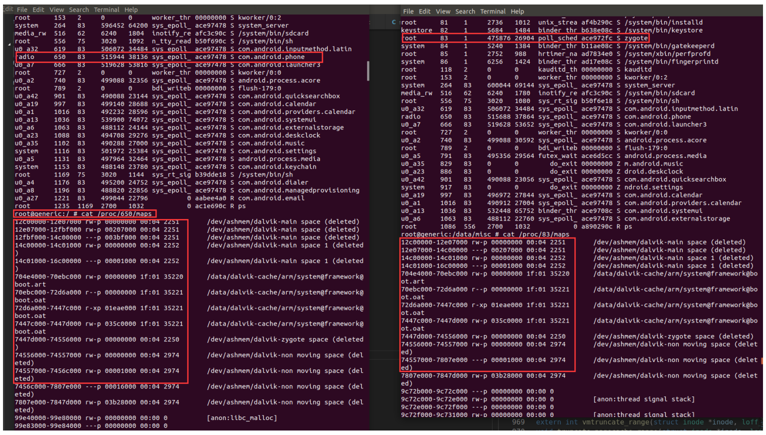
```
-----------------------------------------------
root@generic:/data/misc # ./vm_inspector 82 0xb4299000 0xb429f000
-----------------------------------------------
vm_inspector

pgdir_shift: 21
pmd_shift: 21
page_shift: 12

page - frame
-----------------------------------------------
root@generic:/data/misc # ./vm_inspector 83 0xb4299000 0xb429f000
-----------------------------------------------
vm_inspector

pgdir_shift: 21
pmd_shift: 21
page_shift: 12

page - frame
-----------------------------------------------
root@generic:/data/misc # ./vm_inspector 84 0xb4299000 0xb429f000
-----------------------------------------------
vm_inspector

pgdir_shift: 21
pmd_shift: 21
page_shift: 12

page - frame
0xb4299 - 0x3dd77
0xb429a - 0x3eeaf
0xb429b - 0x3ee5b
0xb429d - 0x3ee59
0xb429e - 0x3dd76
-----------------------------------------------
root@generic:/data/misc # ./vm_inspector 864 0xb4299000 0xb429f000
-----------------------------------------------
vm_inspector

pgdir_shift: 21
pmd_shift: 21
page_shift: 12

page - frame
0xb4299 - 0x3dd77
0xb429a - 0x3cef6
0xb429b - 0x3ee5b
0xb429e - 0x3dd76
-----------------------------------------------
```

We can also see from the following figure that zygote and /system/bin/keystore are differently mapped in this area:



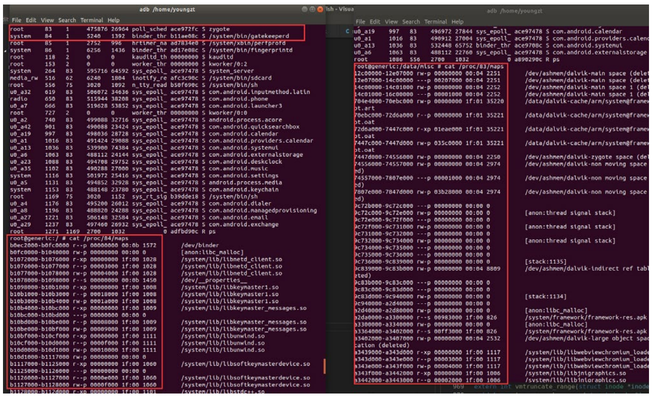
```
-----------------------------------------------
root@generic:/data/misc # ./vm_inspector 84 0xba299000 0xbf000000
-----------------------------------------------
vm_inspector

pgdir_shift: 21
pmd_shift: 21
page_shift: 12

page - frame
0xbebed - 0x393dc
0xbebee - 0x3b3f7
0xbebef - 0x3b352
0xbebf0 - 0x3c005
0xbebf1 - 0x2627d
0xbebf2 - 0x2643d
-----------------------------------------------
root@generic:/data/misc # ./vm_inspector 83 0xba299000 0xbf000000
-----------------------------------------------
vm_inspector

pgdir_shift: 21
pmd_shift: 21
page_shift: 12

page - frame
0xbef12 - 0x3e78d
0xbef13 - 0x3e78e
0xbef14 - 0x3e78f
0xbef15 - 0x3f381
0xbef16 - 0x3f249
-----------------------------------------------
```

On the other hand, check the /proc/pid/maps file, I also get the result:



It again shows Zygote's child process shares some memory area of it.

Check another process which is not Zygote's child, I can easily they use totally different memory area of Zygote.



When I was reading <<Under Standing the Linux Kernal, Third Edition>>, I also learned that there did exits lots of page frame that are shared by multiple process (at Chapter 17.2). It also suggests that "*Anonymous pages are often shared among several processes. The most common case occurs when forking a new process…all page frames owned by the parent—including the anonymous pages—are assigned also to the child*". Results above also confirmed the conception.

4. Problem 4: Change Linux Page Replacement Algorithm

4.1. Description

In this problem, we need to change the page replacement algorithm of our android virtual devices.

The original algorithm and the target algorithm was briefly introduced in our instruction: add a new referenced variable to reflect the importance of a page. If a page is referenced by process, it should be shifted 1 bit to the right and added by 2^K which is defined by myself. Otherwise, the referenced value shifts 1 bit to the right for every period. I should check these two lists periodically, moving the pages whose reference value is greater than or equal to a threshold that defined by yourself to active list, and move the pages whose referenced value is smaller than it to inactive list.

But its too high overview. So first we need to find useful material to read and learn how the original algorithm was implemented, so that we can modify it and change to new one. The most important material was the Chapter 17.3. (Implementing the PFRA) of the book << Understanding the Linux Kernel, Third Edition >> (ULK book)

4.2. Knowledge Learning

All pages belonging to the User Mode address space of processes or to the page cache are grouped into two lists called the active list and the inactive list; The former list tends to include the pages that have been accessed recently, while the latter tends to include the pages that have not been accessed for some time. Clearly, pages should be stolen from the inactive list.

The following messages was from 17.3.1. of ULK book:

> The active list and the inactive list of pages are the core data structures of the page frame reclaiming algorithm. The heads of these two doubly linked lists are stored, respectively, in the `active_list` and `inactive_list` fields of each `zone` descriptor (see the section "Memory Zones" in Chapter 8). The `nr_active` and `nr_inactive` fields in the same descriptor store the number of pages in the two lists. Finally, the `lru_lock` field is a spin lock that protects the two lists against concurrent accesses in SMP systems.
>
> If a page belongs to an LRU list, its `PG_lru` flag in the page descriptor is set. Moreover, if the page belongs to the active list, the `PG_active` flag is set, while if it belongs to the inactive list, the `PG_active` flag is cleared. The `lru` field of the page descriptor stores the pointers to the next and previous elements in the LRU list.
>
> Several auxiliary functions are available to handle the LRU lists:
>
> `add_page_to_active_list( )`
> Adds the page to the head of the zone's active list and increases the `nr_active` field of the zone descriptor.
>
> `add_page_to_inactive_list( )`
> Adds the page to the head of the zone's inactive list and increases the `nr_inactive` field of the zone descriptor.
>
> `del_page_from_active_list( )`
> Removes the page from the zone's active list and decreases the `nr_active` field of the zone descriptor.
>
> `del_page_from_inactive_list( )`
> Removes the page from the zone's inactive list and decreases the `nr_inactive` field of the zone descriptor.
>
> `del_page_from_lru( )`
> Checks the `PG_active` flag of a page; according to the result, removes the page from the active or inactive list, decreases the `nr_active` or `nr_inactive` field of the zone descriptor, and clears, if necessary, the `PG_active` flag.
>
> `activate_page( )`
> Checks the `PG_active` flag; if it is clear (the page is in the inactive list), it moves the page into the active list: invokes `del_page_from_inactive_list( )`, then invokes `add_page_to_active_list( )`, and finally sets the `PG_active` flag. The zone's `lru_lock` spin lock is acquired before moving the page.
>
> `lru_cache_add( )`
> If the page is not included in an LRU list, it sets the `PG_lru` flag, acquires the zone's `lru_lock` spin lock, and invokes `add_page_to_inactive_list( )` to insert the page in the zone's inactive list.
>
> `lru_cache_add_active( )`
> If the page is not included in an LRU list, it sets the `PG_lru` and `PG_active` flags, acquires the zone's `lru_lock` spin lock, and invokes `add_page_to_active_list( )` to insert the page in the zone's active list.

The following figure captured from chapter 17.3.1. of ULK book, (PFRA=Page Frame Reclaim Algorithm) shows a high level overview of how PFRA works (how functions are invoked) in Linux operating system:

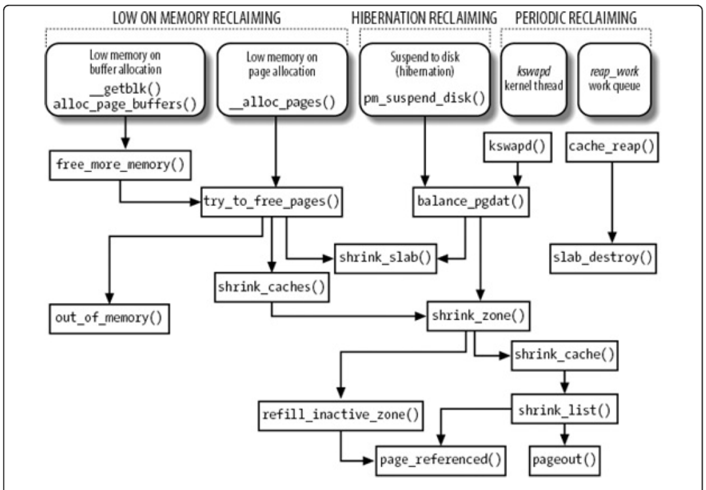


Figure 17-3. The main functions of the PFRA

We can see from the figure how those function are invoked.

The next figure which is also from 17.3.1. of that book shows how a page frame's state are changed through pre-defined functions.

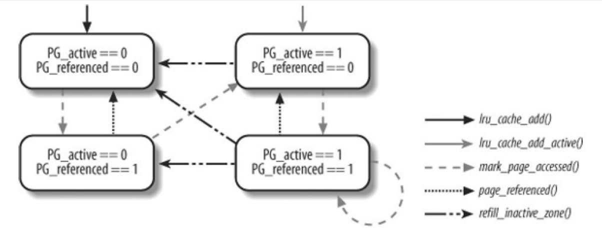


Figure 17-4. Moving pages across the LRU lists

?

These should be the core function we need to understand, though its purpose seems easy for us. The book explains some detail about mark_page_access():

The mark_page_accessed( ) function

Whenever the kernel must mark a page as accessed, it invokes the `mark_page_accessed( )` function. This happens every time the kernel determines that a page is being referenced by a User Mode process, a filesystem layer, or a device driver. For instance, `mark_page_accessed( )` is invoked in the following cases:

- When loading on demand an anonymous page of a process (performed by the `do_anonymous_page( )` function; see the section "Demand Paging" in Chapter 9).
- When loading on demand a page of a memory mapped file (performed by the `filemap_nopage( )` function; see the section "Demand Paging for Memory Mapping" in Chapter 16).
- When loading on demand a page of an IPC shared memory region (performed by the `shmem_nopage( )` function; see the section "IPC Shared Memory" in Chapter 19).
- When reading a page of data from a file (performed by the `do_generic_file_read( )` function; see the section "Reading from a File" in Chapter 16).
- When swapping in a page (performed by the `do_swap_page( )` function; see the section "Swapping in Pages" later in this chapter).
- When looking up a buffer page in the page cache (see the `__find_get_block( )` function in the section "Searching Blocks in the Page Cache" in Chapter 15).

The `mark_page_accessed( )` function executes the following code fragment:

```
if (!PageActive(page) && PageReferenced(page) && PageLRU(page)) {
    activate_page(page);
    ClearPageReferenced(page);
} else if (!PageReferenced(page))
    SetPageReferenced(page);
```

As shown in Figure 17-4, the function moves the page from the inactive list to the active list only if the `PG_referenced` flag is set before the invocation.

We can see from above how the state of a page are changed to be "more active" through the function. Following is what's necessary to know about the page_referenced() function：

The page_referenced( ) function

The `page_referenced( )` function, which is invoked once for every page scanned by the PFRA, returns 1 if either the `PG_referenced` flag or some of the `Accessed` bits in the Page Table entries was set; it returns 0 otherwise. This function first checks the `PG_referenced` flag of the page descriptor; if the flag is set, it clears it. Next, it makes use of the object-based reverse mapping mechanism to check and clear the `Accessed` bits in all User Mode Page Table entries that refer to the page frame. To do this, the function makes use of three ancillary functions; `page_referenced_anon( )`, `page_referenced_file( )`, and `page_referenced_one( )`, which are analogous to the `try_to_unmap_xxx( )` functions described in the section "Reverse Mapping" earlier in this chapter. The `page_referenced( )` function also honors the swap token; see the section "The Swap Token" later in this chapter.

The `page_referenced( )` function never moves a page from the active list to the inactive list; this job is done by `refill_inactive_zone( )`. In practice, this function does a lot more than move pages from the active to the inactive list, so we are going to describe it in greater detail.

When I tried to find a variable named PG_referenced in struct page, I failed unfortunately. But I can find another enum type named pageflags at linux/page-flags.h, who contains it. However, I didn't found this struct in struct page. Therefore, I assume that this is not a concrete variable in code, but a concept for understanding, while we can use its interface function like page_referenced() to change a page's state. There's a lot of other functions to be discover, I need to find what's useful for my implementation.

At 17.3.2.6. of the ULK book shows the core part of PFRA, i.e. the shrink_list() function:

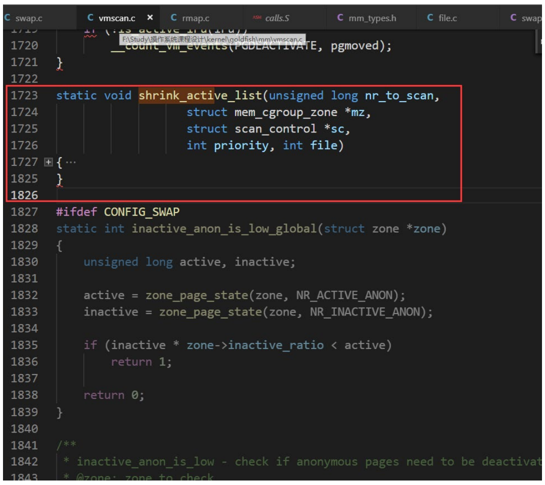
The shrink_list( ) function

We have now reached the heart of page frame reclaiming. While the purpose of the functions illustrated so far, from `try_to_free_pages( )` to `shrink_cache( )`, was to select the proper set of pages candidates for reclaiming, the `shrink_list( )` function effectively tries to reclaim the pages passed as a parameter in the `page_list` list. The second parameter, namely `sc`, is the usual pointer to a `scan_control` descriptor. When `shrink_list( )` returns, `page_list` contains the pages that couldn't be freed.

The function performs the following actions:

1. If the `need_resched` field of the current process is set, it invokes `schedule( )`.
2. Starts a cycle on every page descriptor included in the `page_list` list. For each list item, it removes the page descriptor from the list and tries to reclaim the page frame; if for some reason the page frame could not be freed, it inserts the page descriptor in a local list.
3. Now the `page_list` list is empty: the function moves back the page descriptors from the local list to the `page_list` list.
4. Increases the `sc->nr_reclaimed` field by the number of page frames reclaimed in step 2, and returns that number.

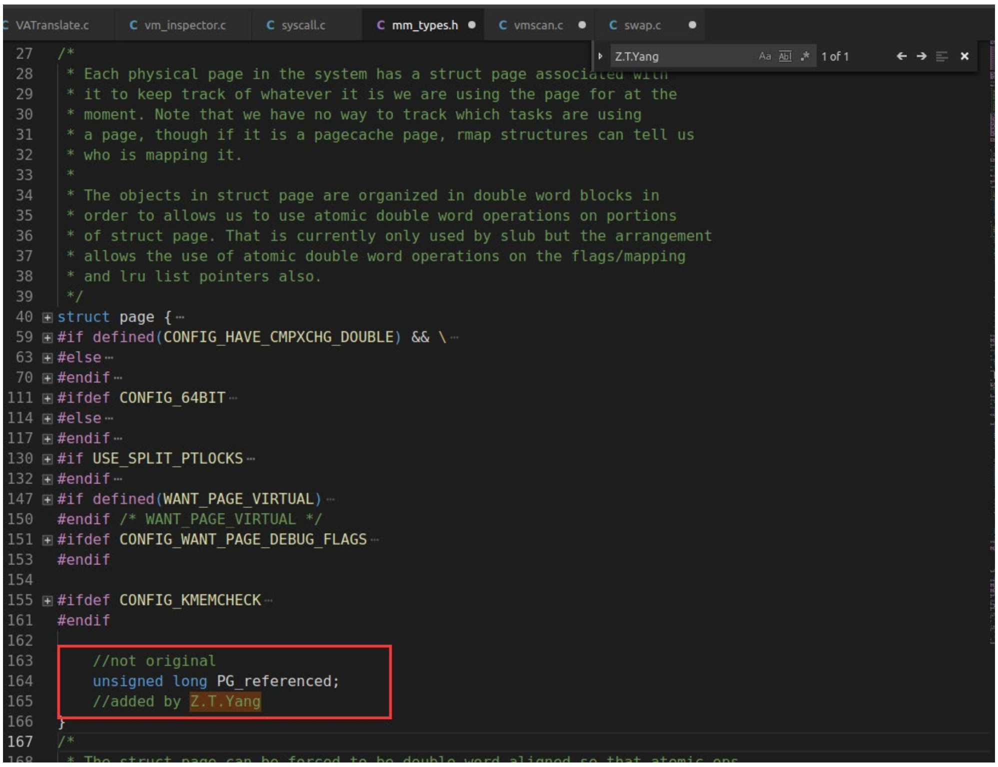
However, in our source code of android virtual device, the function was implemented as shrink_active_list() at mm/vmscan.c:



4.3. Implementing Details

As analyzed, the original referenced bit was changed only through the interface function, while our new algorithm just defines and modify a new variable, the main work was to change anything related with the interface function, i.e. TestClearPageReferenced(),ClearPageReferenced() , SetPageReferenced(), and anything related with them.

Firstly we define the new variable, PG_referenced (this name won't result in conflict, so it might confirmed my assumption before) at linux/mm_types.h



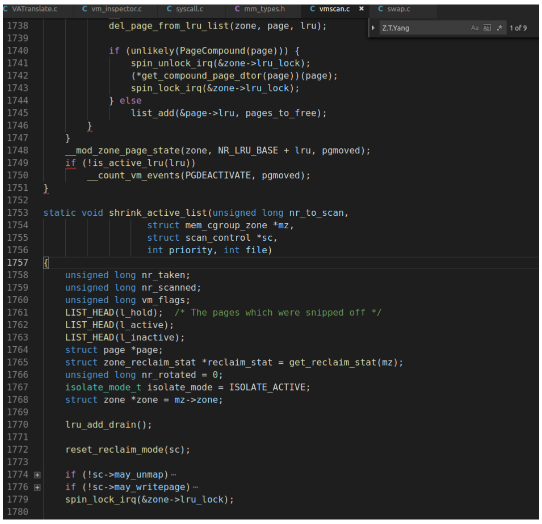
Since originally mark_page_referenced() at mm/swap.c was implemented according to 0 or 1 of "PG_referenced", we change this to shifting right this variable:

```c
/*
 * Mark a page as having seen activity.
 *
 * inactive,unreferenced    ->  inactive,referenced
 * inactive,referenced      ->  active,unreferenced
 * active,unreferenced      ->  active,referenced
 */
void mark_page_accessed(struct page *page)
{
    //not original
    /*
    if (!PageActive(page) && !PageUnevictable(page) &&
            PageReferenced(page) && PageLRU(page)) {
        activate_page(page);
        ClearPageReferenced(page);
    } else if (!PageReferenced(page)) {
        SetPageReferenced(page);
    }
    */
    //deleted by Z.T.Yang

//not original
#define INCREASE_VALUE 1<<20    //2^K in instruction
//added by Z.T. Yang

    //not original
    if (!PageActive(page) && !PageUnevictable(page) &&
            PageLRU(page))
        activate_page(page);

    page->PG_referenced = page->PG_referenced >> 1;
    page->PG_referenced += INCREASE_VALUE;
    //printk("mark_page_accessed(): page-%ld\n", page->index);

    //added by Z.T.Yang
}
EXPORT_SYMBOL(mark_page_accessed);
```

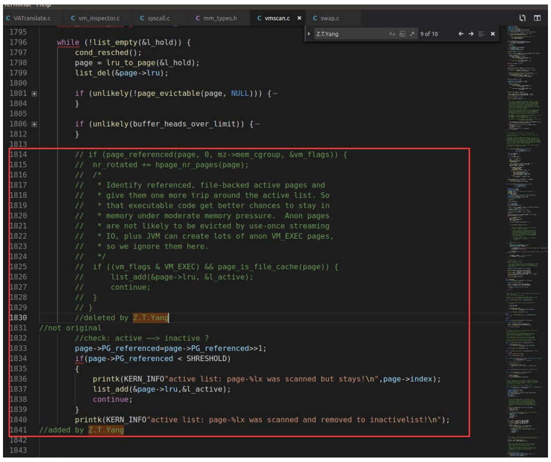
Here is also a function use ClearPageReferenced(), we just deleted it in case something wrong (Though I think it won't be anything wrong if it's not deleted).

```c
/*
 * If the page can not be invalidated, it is moved to the
 * inactive list to speed up its reclaim.  It is moved to the
 * head of the list, rather than the tail, to give the flusher
 * threads some time to write it out, as this is much more
 * effective than the single-page writeout from reclaim.
 *
 * If the page isn't page_mapped and dirty/writeback, the page
 * could reclaim asap using PG_reclaim.
 *
 * 1. active, mapped page -> none
 * 2. active, dirty/writeback page -> inactive, head, PG_reclaim
 * 3. inactive, mapped page -> none
 * 4. inactive, dirty/writeback page -> inactive, head, PG_reclaim
 * 5. inactive, clean -> inactive, tail
 * 6. Others -> none
 *
 * In 4, why it moves inactive's head, the VM expects the page would
 * be write it out by flusher threads as this is much more effective
 * than the single-page writeout from reclaim.
 */
static void lru_deactivate_fn(struct page *page, void *arg)
{
    int lru, file;
    bool active;
    struct zone *zone = page_zone(page);

    if (!PageLRU(page))…
    if (PageUnevictable(page))…
    /* Some processes are using the page */
    if (page_mapped(page))…
    active = PageActive(page);

    file = page_is_file_cache(page);
    lru = page_lru_base_type(page);
    del_page_from_lru_list(zone, page, lru + active);
    ClearPageActive(page);
    //not original
    //ClearPageReferenced(page);
    //deleted by Z.T.Yang
    add_page_to_lru_list(zone, page, lru);
```

The shrink_active_list() in mm/vmscan was called at a period of time, and move some pages from active_list to inactive_list. We need to change its moving condition:

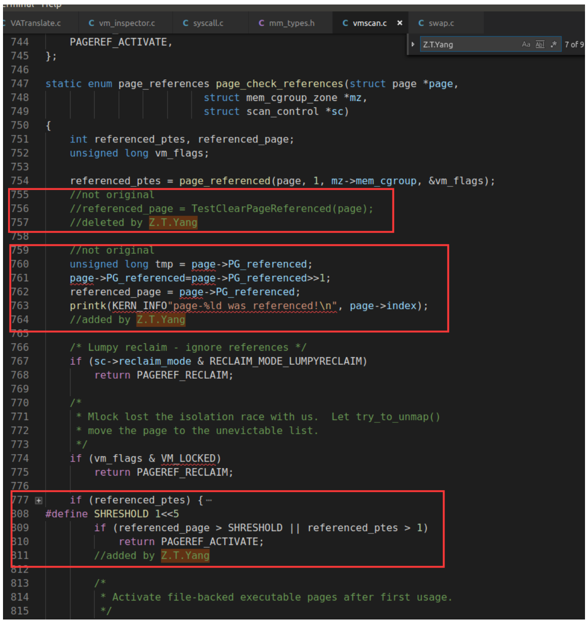


```
			del_page_from_lru_list(zone, page, lru);

			if (unlikely(PageCompound(page))) {
				spin_unlock_irq(&zone->lru_lock);
				(*get_compound_page_dtor(page))(page);
				spin_lock_irq(&zone->lru_lock);
			} else
				list_add(&page->lru, pages_to_free);
		}
	}
	__mod_zone_page_state(zone, NR_LRU_BASE + lru, pgmoved);
	if (!is_active_lru(lru))
		__count_vm_events(PGDEACTIVATE, pgmoved);
}

static void shrink_active_list(unsigned long nr_to_scan,
				struct mem_cgroup_zone *mz,
				struct scan_control *sc,
				int priority, int file)
{
	unsigned long nr_taken;
	unsigned long nr_scanned;
	unsigned long vm_flags;
	LIST_HEAD(l_hold);	/* The pages which were snipped off */
	LIST_HEAD(l_active);
	LIST_HEAD(l_inactive);
	struct page *page;
	struct zone_reclaim_stat *reclaim_stat = get_reclaim_stat(mz);
	unsigned long nr_rotated = 0;
	isolate_mode_t isolate_mode = ISOLATE_ACTIVE;
	struct zone *zone = mz->zone;

	lru_add_drain();

	reset_reclaim_mode(sc);

	if (!sc->may_unmap)…
	if (!sc->may_writepage)…
	spin_lock_irq(&zone->lru_lock);

```



```
	while (!list_empty(&l_hold)) {
		cond_resched();
		page = lru_to_page(&l_hold);
		list_del(&page->lru);

		if (unlikely(!page_evictable(page, NULL))) {…
		}

		if (unlikely(buffer_heads_over_limit)) {…
		}

		// if (page_referenced(page, 0, mz->mem_cgroup, &vm_flags)) {
		//  nr_rotated += hpage_nr_pages(page);
		//  /*
		//   * Identify referenced, file-backed active pages and
		//   * give them one more trip around the active list. So
		//   * that executable code get better chances to stay in
		//   * memory under moderate memory pressure.  Anon pages
		//   * are not likely to be evicted by use-once streaming
		//   * IO, plus JVM can create lots of anon VM_EXEC pages,
		//   * so we ignore them here.
		//   */
		//  if ((vm_flags & VM_EXEC) && page_is_file_cache(page)) {
		//      list_add(&page->lru, &l_active);
		//      continue;
		//  }
		// }
		//deleted by Z.T.Yang
//not original
		//check: active ~~> inactive ?
		page->PG_referenced=page->PG_referenced>>1;
		if(page->PG_referenced < SHRESHOLD)
		{
			printk(KERN_INFO"active list: page-%lx was scanned but stays!\n",page->index);
			list_add(&page->lru,&l_active);
			continue;
		}
		printk(KERN_INFO"active list: page-%lx was scanned and removed to inactivelist!\n");
//added by Z.T.Yang

```

The final function to modify was page_check_references(). In the original design, if a lot of process shares one page, multiple access was recorded as one, so we need to keep this property. The way is also to change the condition value to meet our own PG_referenced:

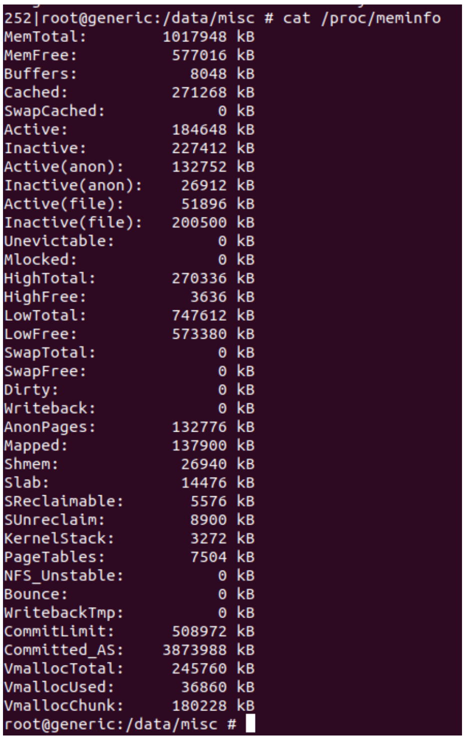


```
	PAGEREF_ACTIVATE,
};

static enum page_references page_check_references(struct page *page,
						  struct mem_cgroup_zone *mz,
						  struct scan_control *sc)
{
	int referenced_ptes, referenced_page;
	unsigned long vm_flags;

	referenced_ptes = page_referenced(page, 1, mz->mem_cgroup, &vm_flags);
	//not original
	//referenced_page = TestClearPageReferenced(page);
	//deleted by Z.T.Yang

	//not original
	unsigned long tmp = page->PG_referenced;
	page->PG_referenced=page->PG_referenced>>1;
	referenced_page = page->PG_referenced;
	printk(KERN_INFO"page-%ld was referenced!\n", page->index);
	//added by Z.T.Yang

	/* Lumpy reclaim - ignore references */
	if (sc->reclaim_mode & RECLAIM_MODE_LUMPYRECLAIM)
		return PAGEREF_RECLAIM;

	/*
	 * Mlock lost the isolation race with us.  Let try_to_unmap()
	 * move the page to the unevictable list.
	 */
	if (vm_flags & VM_LOCKED)
		return PAGEREF_RECLAIM;

	if (referenced_ptes) {…
#define SHRESHOLD 1<<5
		if (referenced_page > SHRESHOLD || referenced_ptes > 1)
			return PAGEREF_ACTIVATE;
		//added by Z.T.Yang

		/*
		 * Activate file-backed executable pages after first usage.
		 */
```

4.4. Test and Result

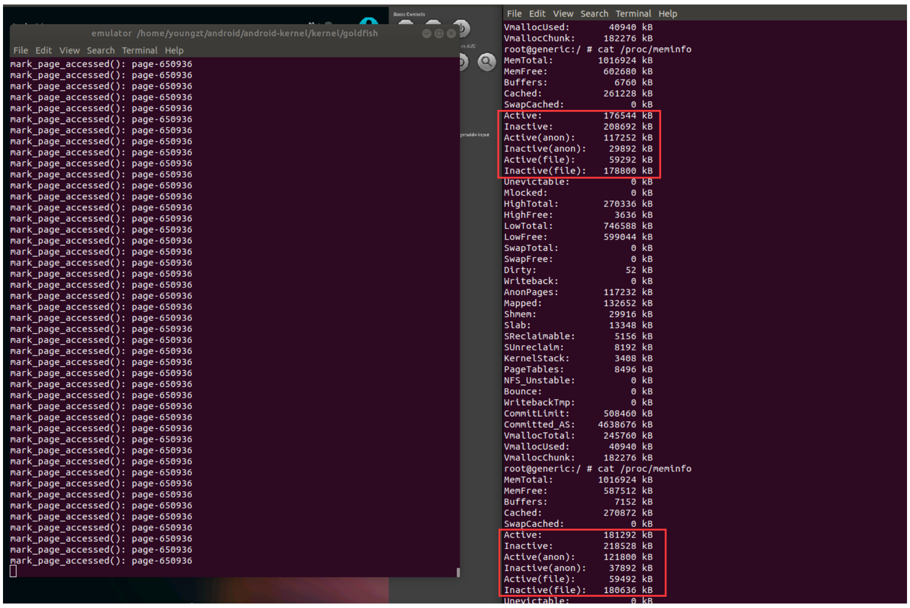
4.4.1. Memory information before the change of algorithm

The following screen capture was captured before compiling the kernel:



4.4.2. After recompiling the kernel

The following is the first time I started my AVD. The left screen shows some help information I use to know if I do the change successfully (Deleted later, while we can see that mark_page_referenced() was called very frequently but others were not). At this moment, K=10 and THRESHOLD $=2^5$.

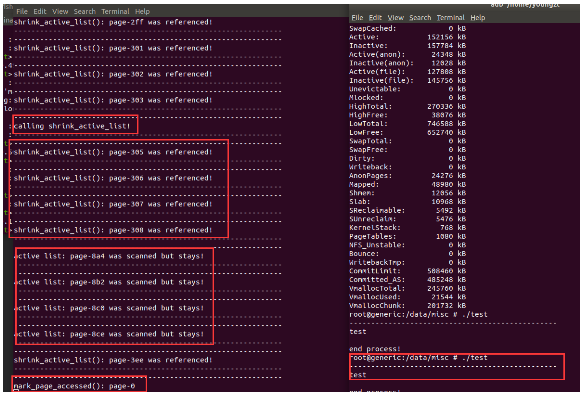


It seems K=10 and SHREAHOLD=2^5 is OK for now.

The kswapd() thread will be invoked only when memory was almost full. So to make sure my new algorithm works, I have to write another program to apply a large number of memory.

Here is the result of my rest program “test” (source code “test.c”, the later test was):

Running the test program,



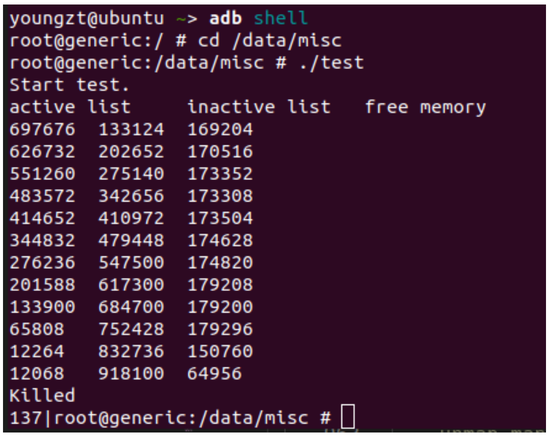
The correctness seems to be confirmed.

Since K and THRESHOLD should be picked by ourselves, I think I need to try some more value to discover some differences. Therefore I modified the test program to record the memory information.

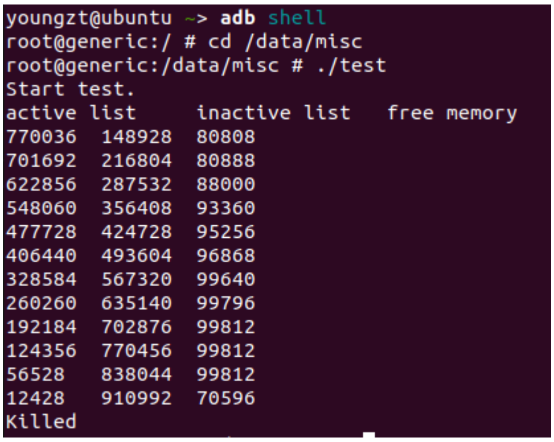
The following is the first time I try to occupy memnory, K=8, THRESHOLD=2^5 we can see from the information at first it increase the number of active list. But when the second time I start it, it get killed soon.

```
root@generic:/data/misc # ./test
Start test.
active list     inactive list   free memory
279224  147376
347096  147376
414816  147412
482668  147648
549352  147596
617076  147992
684952  149744
752960  152216
821088  153088
923508  61904
^Z[2] + Stopped              ./test
root@generic:/data/misc # ./test
Start test.
active list     inactive list   free memory
808924  158332  30700
741124  226052  30964
673120  293504  31008
601324  361552  34668
520468  430012  46924
447680  498664  50692
379976  565464  51376
312024  633124  51408
244072  700812  51420
176244  768276  51548
108788  835668  51556
40956   903612  51108
Killed
[2] + Killed               ./test
[1] - Killed               ./test
```

Then I set K=12, THRESHOLD=2^5 and get:



Then I set K=16, THRESHOLD=2^5 and get:

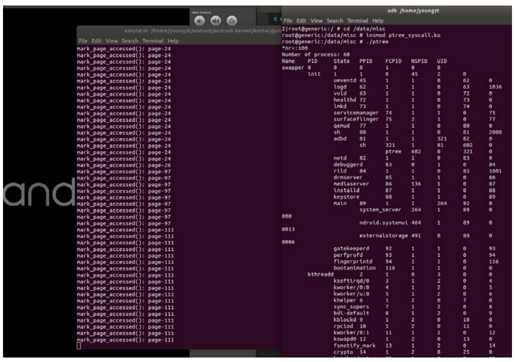


The firs thing to find is that when the process get killed, the free memory is larger if K/log(THRESHOLD) is larger. So probably the first one is a better choice.

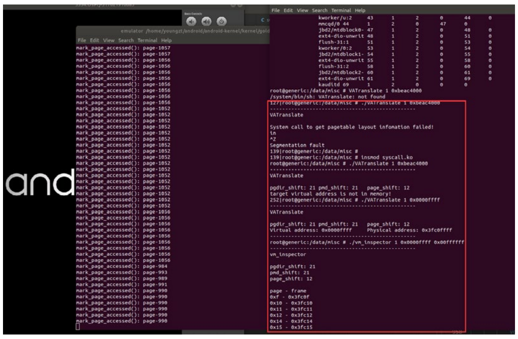
The data was also contained in my submitted files.

I also run the program of problem 2,3 and project 1 and it shows no problem of the system, i.e. no crash happens:

ptree is fine:



VATranslate and vm_inspector is also fine:



Discussion

This might be the most difficult project I've ever meet in my undergraduate study, but I do learned quite a lot, including finding articles and reading text book for useful information, and understand briefly about how linux kernel manage memory sysem . Discussion with classmates is also cery important (thanks for Y.X.Li and J.X.Li, who discussed a lot with me and enhanced my knowledge).

Appendix

1. Any *.c *.h *.mk file required was contained in submitted files;
2. Some test file of problem 4;
3. Figures to explain my undersdanding and design was attached with submitted files, named "figures for report.pptx". (If I misunderstanding the structure of the system please let me know in some way… If it was right, or even it would be helpful for future course of CS307 or CS356, I'd be very happy…)
4. A README file explaining the file structure was contained in submitted files.

Reference

mmap() system call:

https://www.zhihu.com/question/48161206 (user "*in nek*" 's answer)
https://www.cnblogs.com/huxiao-tee/p/4660352.html

walk_page_range() function:

https://elixir.bootlin.com/linux/v4.0/source/mm/pagewalk.c#L239
http://bricktou.cn/mm/pagewalk_walk_page_range_en.html
http://www.cs.columbia.edu/~krj/os/lectures/L17-LinuxPaging.pdf
http://www.vuln.cn/7036

remap_pfn_range() function:

http://blog.rootk.com/post/kernel-memory-mapping.html

http://www.vuln.cn/7036

Zygote process:

https://chromium.googlesource.com/chromium/src/+/HEAD/docs/linux_zygote.md
https://www.cnblogs.com/samchen2009/p/3294713.html
https://www.cnblogs.com/samchen2009/p/3294713.html

Linux Memory Management:

<<Under Sdanding the Linux Kernal, Third Edition >>

Chapter 8. Memory Management

8.1. Page Frame Management

8.2. Memory Area Management

The kswapd kernel threads

<<Under Sdanding the Linux Kernal, Third Edition >>

Chapter 17. The Page Frame Reclaiming

17.3. Implementing The PFRA

17.3.4. Periodic Reclaiming

Page Replacement Algorithm:

<<Under Sdanding the Linux Kernal, Third Edition >>

Chapter 17. The Page Frame Reclaiming

17.2. Reverse Mapping

17.3 Implementing The PFRA

http://www.cs.columbia.edu/~krj/os/lectures/L17-LinuxPaging.pdf
https://blog.csdn.net/zouxiaoting/article/details/8824896
https://linux-mm.org/PageReplacementDesign

Other Useful Information

Course website：

http://www.cs.sjtu.edu.cn/~fwu/teaching/cs307.html