

# 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 do 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

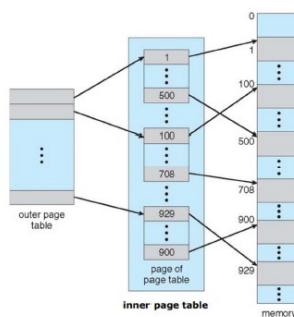
In the Linux kernel, the page table is broken into multiple levels. Address of a system with a 4-level page table is as follows:



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:



Our 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.

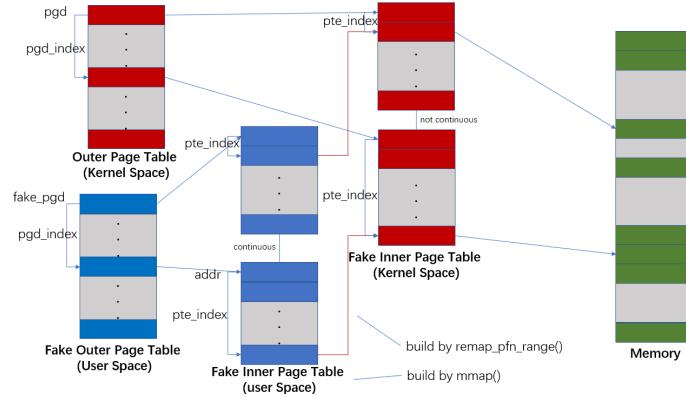
**Related files:** syscall.c(wrote by my self), VATranslate.c(wrote by myself), pagewalk.c (kernel file)

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: VATTranslate.c

The parameter reading is very easy:

```

49 int main(int argc,char **argv)
50 {
51     //unsigned long *table_addr;
52     //unsigned long *fake_pgd_addr;
53     //unsigned long pgd_ind,phy_addr;
54     //unsigned long *phy_base;
55     printf("-----\n");
56     printf("VATTranslate\n\n");
57     if(argc!=3)
58     {
59         printf("argument unmatched!\n");
60         return -1;
61     }
62
63     pid_t pid=atoi(argv[1]);
64     begin_vaddr=strtoul(argv[2],NULL,16);
65

```

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:

```

22 struct pagetable_layout_info{
23     uint32_t pgdir_shift;
24     uint32_t pmd_shift;
25     uint32_t page_shift;
26 };

```

```

65
66 //first system call: get_pagetable_layout
67 if(!syscall(356,&layout_info,sizeof(struct pagetable_layout_info)))
68 {
69     printf("pgdir_shift: %d\t pmd_shift: %d\t page_shift: %d\n",
70           layout_info.pgdir_shift, layout_info.pmd_shift, layout_info.page_shift);
71 }
72 else
73 {
74     printf("System call to get pagetable layout infomation failed!\n ");
75 }
76

```

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:

```
82 //allocate memory for fake pgd  
83 fake_pgd=malloc(sizeof(unsigned long)*page_size);  
84
```

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”:

```
85 //this allocates virtual memory  
86 page_table_addr=mmap(NULL,  
87 1<<22,  
88 PROT_READ | PROT_WRITE,  
89 MAP_SHARED | MAP_ANONYMOUS,  
90 -1,  
91 0);  
92  
93 if(!page_table_addr||!fake_pgd)  
94 {  
95     printf("allocate memory failed!\n");  
96     return -1;  
97 }
```

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:

```
99 int err=syscall(357,pid,fake_pgd,0,page_table_addr,begin_vaddr,begin_vaddr+1);  
100 //error handler.  
101 switch (err)  
102 {  
103     case 1:  
104         printf("expose_page_table: address boundary error!\n");  
105         return 1;  
106     case 2:  
107         printf("expose_page_table: failed to find pid!\n");  
108         return 2;  
109     case 3:  
110         printf("expose_page_table: failed to copy fake pgd!\n");  
111         return 3;  
112     case 4:  
113         printf("expose_page_table: kmalloc error!\n");  
114         return 4;  
115     case 5:  
116         printf("target process has no vm area!\n");  
117         return 5;  
118     case 6:  
119         printf("walk_page_range failed!\n");  
120     default:  
121         break;  
122 }  
123  
124 }
```

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;

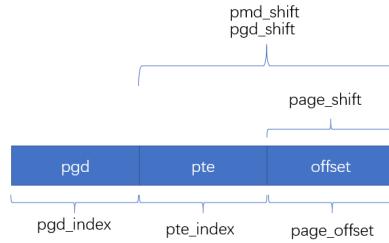
So we need to calculate pgd\_index and pte\_index at first:

```
35 /  
36 #define pgd_index(va,info) ((va)>>info.pgdir_shift)  
37 #define pte_index(va,info) (((va)>>info.page_shift)&((1<<(info.pmd_shift-info.page_shift))-1))  
38 #define page_offset(va,info) ((va)& ( (1<<info.page_shift)-1 ) )  
39 |
```

```
77 const unsigned long page_size=1<<(layout_info.page_shift);  
78 const unsigned long pte_size = 1<<(layout_info.pmd_shift-layout_info.page_shift);  
79 const unsigned long pgd_size = 1<<(32-layout_info.pgdir_shift);  
80 |
```

```
127 unsigned long pgd_index=pgd_index(begin_vaddr,layout_info);  
128 unsigned long pte_index=pte_index(begin_vaddr,layout_info);  
129 unsigned long page_offset = begin_vaddr & 0x0fff;  
130 |
```

This calculation formula is according to the following structure:



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

```

138
139 //get the entry in the table.
140 unsigned long frame=p[pte_index];
141
142 frame &= ~(page_size-1);      //mask higher bit
143 if(!frame)
144 {
145     printf("target virtual address is not in memory!\n");
146     return -4;
147 }
148
149 //add the offset.
150 unsigned long physical_address = frame + page_offset;
151 printf("Virtual address: 0x%08lx\t", begin_vaddr);
152 printf("Physical address: 0x%08lx\n", physical_address);
153
154
  
```

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:

```

44 static int get_pagetable_layout(struct pagetable_layout_info __user *pgtbl_info,
45                                int size)
46 {
47     printk("start get_pagetable_layout() system call...\n");
48
49     if(sizeof(struct pagetable_layout_info)!=size)
50     {
51         printk("struct size unmatched!\n");
52         return -1;
53     }
54
55     struct pagetable_layout_info tmp;
56     tmp.pgdir_shift=PGDIR_SHIFT;
57     tmp.pmd_shift=PMD_SHIFT;
58     tmp.page_shift=PAGE_SHIFT;
59
60     if(copy_to_user(pgtbl_info,&tmp,sizeof(struct pagetable_layout_info)))
61     {
62         printk("Error copying page table layout to user!\n");
63         return -2;
64     }
65     printk("end get_pagetable_layout() system call...\n");
66     printk("\n");
67
68     return 0;
69 }
70
  
```

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:

```

129 |     struct pid* current_pid=find_get_pid(pid);
130 |     if(!current_pid)
131 |     {
132 |         printk("failed to find pid!\n");
133 |         return 2;
134 |     }
135 |     struct task_struct *current_task=get_pid_task(current_pid,PIDTYPE PID);
136 |     if(!current_task)
137 |     {
138 |         printk("error finding task struct!\n");
139 |         return 2;
140 |     }
141 |     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.

```

72 //instruct to carry when walking the page
73 struct myPrivate
74 {
75     unsigned long *fake_pgd_base;
76     unsigned long pte_base;
77 };

```

```

163     my_private.fake_pgd_base=kalloc(PAGE_SIZE,sizeof(unsigned long),GFP_KERNEL);
164     if(!my_private.fake_pgd_base)
165     {
166         printk("kalloc error!\n");
167         return 4;
168     }
169     my_private.pte_base=page_table_addr;
170

```

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.

```

168 */
169 int walk_page_range(unsigned long addr, unsigned long end,
170                      struct mm_walk *walk)
171 {
172 }
173
174 EXPORT_SYMBOL(walk_page_range);

```

The struct mm\_walk is defined in linux/mm.h:

```

917 /**
918 * mm_walk - callbacks for walk_page_range
919 * @pgd_entry: if set, called for each non-empty PGD (top-level) entry
920 * @pud_entry: if set, called for each non-empty PUD (2nd-level) entry
921 * @pmd_entry: if set, called for each non-empty PMD (3rd-level) entry
922 *           this handler is required to be able to handle
923 *           @pmd_trans_huge() pmds. They may simply choose to
924 *           split_huge_page() instead of handling it explicitly.
925 * @pte_entry: if set, called for each non-empty PTE (4th-level) entry
926 * @pte_hole: if set, called for each hole at all levels
927 * @hugetlb_entry: if set, called for each hugetlb entry
928 *           *Caution*: The caller must hold mmap_sem() if @hugetlb_entry
929 *           is used.
930 *
931 * (see walk_page_range for more details)
932 */
933 struct mm_walk {
934     int (*pgd_entry)(pgd_t *, unsigned long, unsigned long, struct mm_walk *);
935     int (*pud_entry)(pud_t *, unsigned long, unsigned long, struct mm_walk *);
936     int (*pmd_entry)(pmd_t *, unsigned long, unsigned long, struct mm_walk *);
937     int (*pte_entry)(pte_t *, unsigned long, unsigned long, struct mm_walk *);
938     int (*pte_hole)(unsigned long, unsigned long, struct mm_walk *);
939     int (*hugetlb_entry)(pte_t *, unsigned long,
940                          unsigned long, unsigned long, struct mm_walk *);
941     struct mm_struct *mm;
942     void *private;
943 };

```

It has several function pointer, which would 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 to user 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:

```

169     my_private.pte_base=page_table_addr;
170
171     walk.pgd_entry=&my_pgd_entry;//my_pgd_entry;
172     walk.pud_entry=NULL;//my_pud_entry;
173     walk.pmd_entry=NULL;//my_pmd_entry;    //my pmd entry function;
174     walk.pte_entry=NULL;//my_pte_entry;
175     walk.pte_hole=NULL;
176     walk.hugetbl_entry=NULL;
177     walk.mm=current_task->mm;
178     walk.private=&my_private;
179

```

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

```

81 int my_pgd_entry(pmd_t *pgd,unsigned long addr,unsigned long next,struct mm_walk *walk)
82 {
83
84     unsigned long pgd_index=pgd_index(addr);
85
86     //get the physical frame number.
87     unsigned long pfn = page_to_pfn(pmd_page((unsigned long)*pgd));
88     if(pgd_none(*pgd)||pgd_bad(*pgd)||!pfn_valid(pfn))
89     {
90         printk("failed to find pfn!\n");
91         return 0;
92     }
93     printk(KERN_INFO"pfn:%08X\n",pfn);
94
95     struct myPrivate *base=walk->private;
96
97     //struct vm_area_struct *vma=current->mm->mmap;
98     struct vm_area_struct* vma = find_vma(current->mm, base->pte_base);
99     if(!vma)
100     {
101         printk("find_vma error!\n");
102         return 0;
103     }
104
105
106     down_write(&current->mm->mmap_sem);
107     int err=remap_pfn_range(vma,base->pte_base,pfn,
108                             PTE_SIZE*sizeof(unsigned long),
109                             vma->vm_page_prot);
110     up_write(&current->mm->mmap_sem);
111
112     //remap: can find pte_base according to pgd_base and pmd_index
113     //fake_pgd_base + (pgd_index * sizeof(each_entry))
114     //you will either get a null for non-exist pmd or the address of a fake pmd
115     //pgd_index and pmd_index are the same here since only 2-level in this 32-bit OS
116     base->fake_pgd_base[pgd_index] = base->pte_base;
117     //you can get the remapped address of a Page Table
118     //by reading the content at the address
119     //fake_pmd_base + (pmd_index * sizeof(each_entry))
120     base->pte_base += PTE_SIZE;
121
122 }

```

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 mm.h      C pagewalk.c x
1 #include <linux/mm.h>
2 #include <linux/highmem.h>
3 #include <linux/sched.h>
4 #include <linux/hugetlb.h>
5 #include <linux/export.h>
6

c mm.h      C pagewalk.c x
168 */
169 int walk_page_range(unsigned long addr, unsigned long end,
170                      struct mm_walk *walk)
171 {
172 }
173
174 EXPORT_SYMBOL(walk_page_range);

c mm.h      x  C pagewalk.c
929 *
930 *
931 * (see walk_page_range for more details)
932 */
933 struct mm_walk {
934     int (*pgd_entry)(pgd_t *, unsigned long, unsigned long, struct mm_walk *);
935     int (*pud_entry)(pud_t *, unsigned long, unsigned long, struct mm_walk *);
936     int (*pmd_entry)(pmd_t *, unsigned long, unsigned long, struct mm_walk *);
937     int (*pte_entry)(pte_t *, unsigned long, unsigned long, struct mm_walk *);
938     int (*pte_hole)(unsigned long, unsigned long, struct mm_walk *);
939     int (*hugetbl_entry)(pte_t *, unsigned long,
940                          unsigned long, unsigned long, struct mm_walk *);
941     struct mm_struct *mm;
942     void *private;
943 };
944
945 extern int walk_page_range(unsigned long addr, unsigned long end,
946                           struct mm_walk *walk);
947 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)

```
and traveling page table...
_end_expose_page_table() system call...
_____
type=1400 audit(1506014294.431:629): avc: denied { module_request } for pid=8199 comm="iptunnel" dev="pt TPMSS" scontext=u:r:netd:s0 tcontext=u:r:kernel:s0 tclass=system permissive=0
headdth: battery l5: v0 = 0 t=2 sec=2 chg=a
type=1400 audit(1506014305.551:630): avc: denied { module_request } for pid=7986 comm="netd" knode="netd-dummy" scontext=u:r:netd:s0 tcontext=u:r:kernel:s0 tclass=system permissive=0
type=1400 audit(1506014305.551:631): avc: denied { sys_module } for pid=7986 comm="netd" capability=16 scontext=u:r:netd:s0 tcontext=u:r:netd:s0 tclass=capability permissive=0
_____
int: Service 'zygote' (pid 7988) killed by signal 9
int: Service 'zygote' (pid 7988) killing any children in process group
int: Service 'file' Unable to open '/sys/android_power/request_state': No such file or directory
int: write_file: Unable to write to '/sys/power/state': Invalid argument
int: Service 'media' is being killed...
int: Service 'media' is being killed...
int: Service 'media' (pid 7988) killed by signal 9
int: Service 'media' (pid 7988) killing any children in process group
int: Service 'media' (pid 7987) killed by signal 9
int: Service 'media' (pid 7987) killing any children in process group
int: Starting service 'media'...
int: Starting service 'media'...
int: Starting service 'zygote'...

_start_get_pagedable_layout() system call...
struct size unmatched

_start_get_pagedable_layout() system call...
_end_get_pagedable_layout() system call...
_____
start_expose_page_table() system call...
pid: 2 name: kthreadd
Virtual memory area:
target process has no vm area!
```

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:

S	USER	PID	PPID	RSS	WCHAN	PC	NAME
S	root	1	0	2208	752	sys_epoll	0x666dbc
S	root	2	0	0	0	kthread	0x1
S	root	3	2	0	0	run_ksofti	0x00000000
S	root	6	2	0	0	rescuer_th	0x00000000
S	root	7	2	0	0	bdt_forker	0x00000000
S	root	8	2	0	0	bdt_forker	0x00000000
S	root	9	2	0	0	rescuer_th	0x00000000
S	root	10	2	0	0	rescuer_th	0x00000000
S	root	12	2	0	0	kswapd	0x00000000
S	root	13	2	0	0	fsmryptd	0x00000000
S	root	14	2	0	0	rescuer_th	0x00000000
S	root	25	2	0	0	worker_thr	0x00000000
S	root	30	2	0	0	mtd_blktra	0x00000000
S	root	35	2	0	0	mtd_blktra	0x00000000
S	root	40	2	0	0	mtd_blktra	0x00000000
S	root	41	2	0	0	rescuer_th	0x00000000
S	root	42	2	0	0	rescuer_th	0x00000000
S	root	43	2	0	0	worker_thr	0x00000000
S	root	44	2	0	0	mmc_queue	0x00000000
S	root	45	2	1948	652	mtd_bktrched	0x00000000
S	root	47	2	0	0	rescuer_th	0x00000000
S	root	48	2	0	0	rescuer_th	0x00000000
S	root	51	2	0	0	bdi_wlrbite	0x00000000
S	root	53	2	0	0	kjournalard	0x00000000
S	root	54	2	0	0	rescuer_th	0x00000000
S	root	59	2	0	0	kjournalard	0x00000000
S	root	60	2	0	0	rescuer_th	0x00000000
S	logd	61	1	12456	4156	sys_rt_sigt	0x23ca1e18
S	root	62	1	8812	384	htmwtime	0x00000000
S	root	68	2	0	0	tauditd	0x00000000
S	root	70	2	2432	2432	epoll	0x00000000
S	root	71	3	3852	1128	sys_epoll	0x00000000
S	root	76	2	0	0	bdt_157478	0x00000000

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

### 3. Problem 3: Investigate Android Process Address Spaces

### 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.

```
131     printf("\npage - frame \n");
132     unsigned long i;
133     for(i=begin_vaddr>>layout_info.page_shift;i< end_vaddr>>layout_info.page_shift;++i)
134     {
135         unsigned long pgd_index=pgd_index(i<<layout_info.page_shift,layout_info);
136         unsigned long pte_index=pte_index(i<<layout_info.page_shift,layout_info);
137         unsigned long* p=fake_pgd[pgd_index];
138         if(p)
139         {
140             unsigned long frame=p[pte_index];
141             if(frame>>layout_info.page_shift)
142             {
143                 printf("0x%lx - 0x%lx\n",i,frame>>layout_info.page_shift);
144             }
145         }
146     }
```

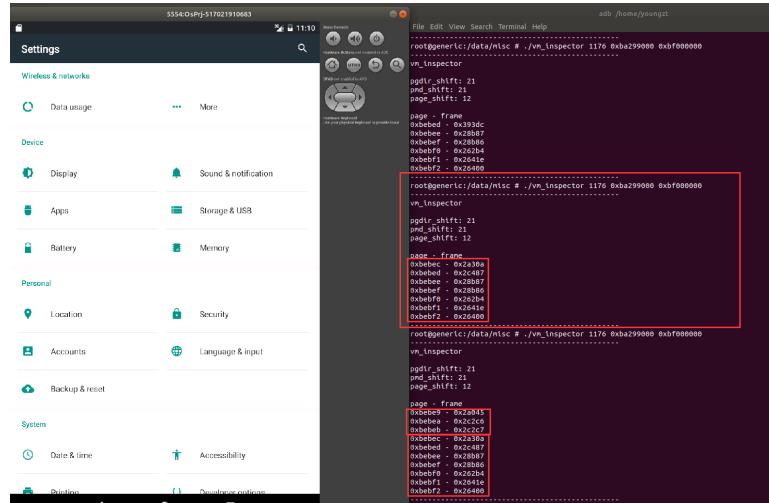
### 3.3. Result

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:



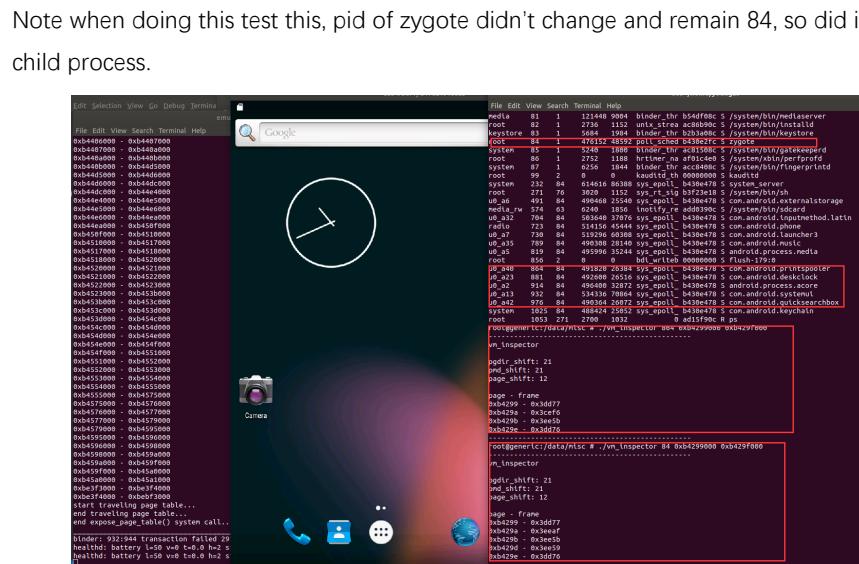
This implies that when running an app, its virtual memory area are always changing.

### 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 might work, so I changed it and it didn't 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.



```

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
0xb429c - 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
0xb429c - 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 - 0x3ceef6
0xb429b - 0x3ee5b
0xb429c - 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 - 0x3ceef6
0xb429b - 0x3ee5b
0xb429c - 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 0xbff000000
-----
vm_inspector
pgdir_shift: 21
pmd_shift: 21
page_shift: 12

page - frame
0xbebed - 0x393dc
0xbebee - 0x3bf7f
0xbebef - 0x3b352
0xbebef - 0x3c3e2
0xbebef - 0x2627d
0xebef2 - 0x2643d
-----
root@generic:/data/misc # ./vm_inspector 83 0xba299000 0xbff000000
-----
vm_inspector
pgdir_shift: 21
pmd_shift: 21
page_shift: 12

page - frame
0xbebef12 - 0x3e76d
0xbebef13 - 0x3e78d
0xbebef14 - 0x3e78f
0xbebef15 - 0x3f381
0xbebef16 - 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 exists 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:

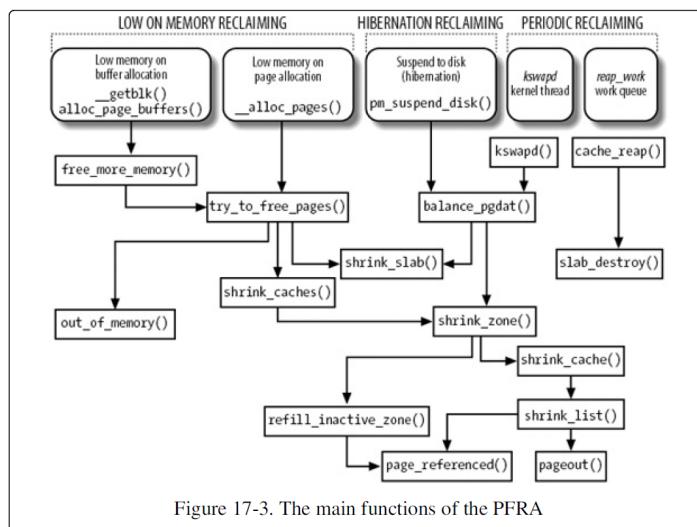
```
The active list and the inactive list of pages are the core data structures of the
page frame reclaiming algorithm. The heads of these two doubly-linked lists
are stored, respectively, in the active_list and inactive_list fields of
each zone descriptor (see the section "Memory Zones" in Chapter 3). 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),
    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:



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.

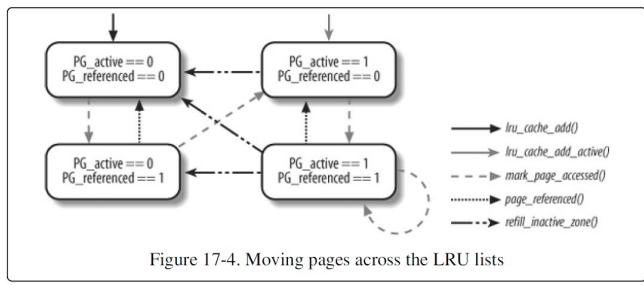


Figure 17-4. Moving pages across the LRU lists

?

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_accessed()`:

#### 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 find 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`:

```

1723 static void shrink_active_list(unsigned long nr_to_scan,
1724         struct mem_cgroup_zone *mz,
1725         struct scan_control *sc,
1726         int priority, int file)
1727 {
1728     #ifdef CONFIG_SWAP
1729     static int inactive_anon_is_low_global(struct zone *zone)
1730     {
1731         unsigned long active, inactive;
1732         active = zone_page_state(zone, NR_ACTIVE_ANON);
1733         inactive = zone_page_state(zone, NR_INACTIVE_ANON);
1734         if (inactive * zone->inactive_ratio < active)
1735             return 1;
1736         return 0;
1737     }
1738     /**
1739      * inactive_anon_is_low - check if anonymous pages need to be deactivated
1740      */
1741     /* deactivate pages if there are too many inactive ones */
1742 }

```

### 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

```

27 /**
28  * Each physical page in the system has a struct page associated with
29  * it to keep track of whatever it is we are using the page for at the
30  * moment. Note that we have no way to track which tasks are using
31  * a page, though if it is a pagecache page, rmap structures can tell us
32  * who is mapping it.
33  *
34  * The objects in struct page are organized in double word blocks in
35  * order to allow us to use atomic double word operations on portions
36  * of struct page. That is currently only used by slab but the arrangement
37  * allows the use of atomic double word operations on the flags/mapping
38  * and lru list pointers also.
39 */
40 struct page { ...
41 #if defined(CONFIG_HAVE_CMPXCHG_DOUBLE) && \
42 #else ...
43 #endif ...
44 #endif ...
45 #if defined(CONFIG_64BIT...
46 #else ...
47 #endif ...
48 #endif ...
49 #if USE_SPLIT_PTLOCKS...
50 #endif ...
51 #if defined(WANT_PAGE_VIRTUAL) ...
52 #endif /* WANT_PAGE_VIRTUAL */
53 #if defined(CONFIG_WANT_PAGE_DEBUG_FLAGS) ...
54 #endif ...
55 #ifdef CONFIG_KMEMCHECK...
56 #endif ...
57 /**
58  * @page - a pointer to the struct page for a given virtual address
59  * @pte - a pointer to the page table entry for a given virtual address
60  * @lru - a pointer to the lru list pointer for a given virtual address
61  * @lru_index - the index of the lru list for a given virtual address
62  * @lru_lock - the lock protecting the lru list
63  * @lru_lock_index - the index of the lru lock for a given virtual address
64  * @pg_referenced - the referenced bit for a given virtual address
65  * @pg_lru - the lru list pointer for a given virtual address
66  */
67 /**
68  * This struct page can be forced to be double word aligned so that atomic ops
69  */

```

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:

```

356 /*
357  * Mark a page as having seen activity.
358  *
359  * inactive,unreferenced    -> inactive,referenced
360  * inactive,referenced      -> active,unreferenced
361  * active,unreferenced     -> active,referenced
362  */
363 void mark_page_accessed(struct page *page)
364 {
365     /* not original */
366     if (!PageActive(page) && !PageUnevictable(page) &&
367         | PageReferenced(page) && PageLRU(page)) {
368         activate_page(page);
369         ClearPageReferenced(page);
370     } else if (!PageReferenced(page)) {
371         SetPageReferenced(page);
372     }
373     */
374     //deleted by Z.T.Yang
375
376     /* not original */
377     #define INCREASE_VALUE 1<<20    //2^K in instruction
378     //added by Z.T. Yang
379
380     /* not original */
381     if (!PageActive(page) && !PageUnevictable(page) &&
382         | PageLRU(page))
383         activate_page(page);
384
385     page->PG_referenced = page->PG_referenced >> 1;
386     page->PG_referenced += INCREASE_VALUE;
387     //printf("mark_page_accessed(): page-%ld\n", page->index);
388
389     //added by Z.T.Yang
390 }
391 EXPORT_SYMBOL(mark_page_accessed);
392

```

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).

```

445 /*
446  * If the page can not be invalidated, it is moved to the
447  * inactive list to speed up its reclaim. It is moved to the
448  * head of the list, rather than the tail, to give the flusher
449  * threads some time to write it out, as this is much more
450  * effective than the single-page writeout from reclaim.
451  *
452  * If the page isn't page_mapped and dirty/writeback, the page
453  * could reclaim asap using PG_reclaim.
454  *
455  * 1. active, mapped page -> none
456  * 2. active, dirty/writeback page -> inactive, head, PG_reclaim
457  * 3. inactive, mapped page -> none
458  * 4. inactive, dirty/writeback page -> inactive, head, PG_reclaim
459  * 5. inactive, clean -> inactive, tail
460  * 6. Others -> none
461  *
462  *
463  * In 4, why it moves inactive's head, the VM expects the page would
464  * be write it out by flusher threads as this is much more effective
465  * than the single-page writeout from reclaim.
466  */
467 static void lru_deactivate_fn(struct page *page, void *arg)
468 {
469     int lru, file;
470     bool active;
471     struct zone *zone = page_zone(page);
472
473     if (!PageLRU(page))-
474     if (PageUnevictable(page))-*
475     /* Some processes are using the page */
476     if (page_mapped(page))-
477     active = PageActive(page);
478
479     file = page_is_file_cache(page);
480     lru = page_lru_base_type(page);
481     del_page_from_lru_list(zone, page, lru + active);
482     ClearPageActive(page);
483     */
484     //not original
485     //ClearPageReferenced(page);
486     //deleted by Z.T.Yang
487     add_page_to_lru_list(zone, page, lru);
488

```

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:

```

1738     del_page_from_lru_list(zone, page, lru);
1739     if (unlikely(PageCompound(page))) {
1740         spin_unlock_irq(zone->lru.lock);
1741         (*get_compound_page_dtor(page))(page);
1742         spin_lock_irq(zone->lru.lock);
1743     } else
1744         list_add(&page->lru, pages_to_free);
1745 }
1746 }
1747 mod_zone_page_state(zone, NR_LRU_BASE + lru, pgmoved);
1748 if (!is_active_lru(lru))
1749     __count_vm_events(PGDEACTIVATE, pgmoved);
1750 }
1751 }

1752 static void shrink_active_list(unsigned long nr_to_scan,
1753                               struct mem_cgroup_zone *mz,
1754                               struct scan_control *sc,
1755                               int priority, int file)
1756 {
1757     unsigned long nr_taken;
1758     unsigned long nr_scanned;
1759     unsigned long vm_flags;
1760     LIST_HEAD(l_hold); /* The pages which were snipped off */
1761     LIST_HEAD(l_inactive);
1762     struct page *page;
1763     struct zone *zone = m_z->zone;
1764     unsigned long nr_rotated = 0;
1765     isolate_mode_t isolate_mode = ISOLATE_ACTIVE;
1766     struct zone *zone = mz->zone;
1767     lru.add_drain();
1768     reset_reclaim_mode(sc);
1769     if (sc->may_unmap)
1770     if (sc->may_writepage)
1771         spin_lock_irq(&zone->lru.lock);
1772 }

1773 while (!list_empty(&l_hold)) {
1774     cond_resched();
1775     page = lru_to_page(&l_hold);
1776     list_del(&page->lru);
1777     if (unlikely(!page_evictable(page, NULL))) {
1778         continue;
1779     }
1780     if (unlikely(buffer_heads_over_limit)) {
1781         continue;
1782     }
1783     // If (page referenced == 0, mz->mem_cgroup, &vm_flags) {
1784     //     nr_rotated += hpage->pages->page;
1785     //     /*
1786     //      * Identify referenced, file-backed active pages and
1787     //      * give those one more trip around the active list. So
1788     //      * that executable code get better chances to stay in
1789     //      * memory under moderate memory pressure. Anon pages
1790     //      * are not likely to be evicted by user-space streaming
1791     //      * IO, so we can route them to anon VM backed pages,
1792     //      * so we ignore them here.
1793     //      */
1794     //     if ((vm_flags & VM_EXEC) && page_is_file_cache(page)) {
1795         //     list_add(&page->lru, &l_active);
1796         //     continue;
1797     }
1798     // }
1799     // deleted by Z.T.Yang
1800     //not original
1801     //check active == inactive ?
1802     if (pg->PG_referenced && pg->referenced >= 1)
1803     if (pg->PG_referenced < SHRESHOLD)
1804     {
1805         printk(KERN_INFO "active list: page-%lx was scanned but stays\n", page->index);
1806         list_add(&page->lru, &l_active);
1807         continue;
1808     }
1809     printk(KERN_INFO "active list: page-%lx was scanned and removed to inactive list\n");
1810 // added by Z.T.Yang
1811 }
1812
1813

```

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:

```

744     PAGEREF_ACTIVATE,
745 };
746
747 static enum page_references page_check_references(struct page *page,
748                                                   struct mem_cgroup_zone *mz,
749                                                   struct scan_control *sc)
750 {
751     int referenced_ptes, referenced_page;
752     unsigned long vm_flags;
753
754     referenced_ptes = page_referenced(page, 1, mz->mem_cgroup, &vm_flags);
755     //not original
756     //reference page = TestClearPageReferenced(page);
757     //deleted by Z.T.Yang
758
759     //not original
760     unsigned long tmp = page->PG_referenced;
761     page->PG_referenced = page->PG_referenced >= 1;
762     referenced_page = page->PG_referenced;
763     printk(KERN_INFO "page-%ld was referenced!\n", page->index);
764 // added by Z.T.Yang
765
766     /* Lumpy reclaim - ignore references */
767     if (sc->reclaim_mode & RECLAIM_MODE_LUMPYRECLAIM)
768         return PAGEREF_RECLAIM;
769
770     /*
771     * Block lost the isolation race with us. Let try_to_unmap()
772     * move the page to the unevictable list.
773     */
774     if (vm_flags & VM_LOCKED)
775         return PAGEREF_RECLAIM;
776
777     /*
778     * if (referenced_ptes) {
779     #define SHRESHOLD 1<<5
780     if (referenced_page > SHRESHOLD || referenced_ptes > 1)
781         return PAGEREF_ACTIVATE;
782     //added by Z.T.Yang
783
784     /*
785     * Activate file-backed executable pages after first usage.
786     */
787

```

#### 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:

```
525[root@generic:/data/misc # cat /proc/meminfo
MemTotal:           1017948 kB
MemFree:            577016 kB
Buffers:             8048 kB
Cached:              271268 kB
SwapCached:          0 kB
Active:              184648 kB
Inactive:            227412 kB
Active(anon):        132752 kB
Inactive(anon):      26912 kB
Active(file):         51896 kB
Inactive(file):      200500 kB
Unevictable:          0 kB
Mlocked:              0 kB
HighTotal:            270336 kB
HighFree:              3636 kB
LowTotal:             747612 kB
LowFree:              573380 kB
SwapTotal:              0 kB
SwapFree:              0 kB
Dirty:                  0 kB
Writeback:              0 kB
AnonPages:            132776 kB
Mapped:               137900 kB
Shmem:                 26940 kB
Slab:                  14476 kB
SReclaimable:          5576 kB
SUnreclaim:             8900 kB
KernelStack:            3272 kB
PageTables:             7504 kB
NFS_Unstable:            0 kB
Bounce:                  0 kB
WritebackTmp:            0 kB
Commitlimit:            508972 kB
Committed_AS:          3873988 kB
VmallocTotal:            245760 kB
VmallocUsed:             36860 kB
VmallocChunk:            180228 kB
root@generic:/data/misc #
```

#### 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  $\text{THRESHOLD} = 2^5$ .

The screenshot shows a terminal window with two panes. The left pane displays a continuous stream of log messages from the kernel, specifically from the `mark_page_accessed()` function, indicating page access events. The right pane shows the output of the `cat /proc/meminfo` command, which provides detailed memory statistics.

Memory Type	Value
VmAllocUsed	40940 kB
VmAllocChunk	182276 kB
rootgenerics/	# cat /proc/meminfo
MemTotal:	1016924 kB
MemFree:	602680 kB
Buffers:	576728 kB
Cached:	261228 kB
SwapCached:	0 kB
Active:	176544 kB
Inactive:	208692 kB
Active(anon):	111720 kB
Inactive(anon):	29892 kB
Active(file):	59292 kB
Inactive(file):	178800 kB
Unevictable:	0 kB
HighWater:	0 kB
HighTotal:	270336 kB
HighFree:	3036 kB
LowTotal:	746588 kB
LowFree:	59044 kB
SwapTotal:	0 kB
SwapFree:	0 kB
Dirty:	52 kB
Writeback:	0 kB
AnonPages:	117232 kB
Mapped:	13264 kB
Shmem:	29916 kB
Slab:	13348 kB
SReclaimable:	5156 kB
SUnreclaim:	8192 kB
KernelStack:	3468 kB
PageTables:	8490 kB
NFS_Unstable:	0 kB
Bounce:	0 kB
WritebackTmp:	0 kB
CommitLimit:	506400 kB
CommittedAS:	4088048 kB
VmallocTotal:	245760 kB
VmallocUsed:	40940 kB
VmallocChunk:	182276 kB
rootgenerics/	# cat /proc/meminfo
MemTotal:	1016924 kB
MemFree:	587512 kB
Buffers:	57152 kB
Cached:	270872 kB
SwapCached:	0 kB
Active:	181592 kB
Inactive:	218524 kB
Active(anon):	121880 kB
Inactive(anon):	37892 kB
Active(file):	59492 kB
Inactive(file):	180636 kB

It seems K=10 and SHREAHOLD=2<sup>5</sup> 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,

```
File Edit View Search Terminal Help

dmesg | grep shrink_active_list
shrink_active_list(): page-2ff was referenced!
shrink_active_list(): page-381 was referenced!
shrink_active_list(): page-382 was referenced!
shrink_active_list(): page-383 was referenced!
shrink_active_list(): page-385 was referenced!
shrink_active_list(): page-386 was referenced!
shrink_active_list(): page-387 was referenced!
shrink_active_list(): page-388 was referenced!

active list: page-84d was scanned but stays!
active list: page-9b2 was scanned but stays!
active list: page-9c1 was scanned but stays!
active list: page-9e6 was scanned but stays!

shrink_active_list(): page-3ee was referenced!
mark_page_accessed(): page-0
```

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<sup>5</sup> 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
808948 158332 30700
741124 226052 30964
673120 293504 31008
6011324 3611552 34668
520468 430012 46924
447680 498664 50692
377976 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  $K=12$ ,  $\text{THRESHOLD}=2^5$  and get:

```
youngzt@ubuntu -> adb shell
root@generic:/ # cd /data/misc
root@generic:/data/misc # ./test
Start test.
active list      inactive list    free memory
697676  133124  169204
626732  202652  170516
551260  275140  173352
483572  342656  173308
414652  410972  173504
344832  479448  174628
276236  547500  174820
201588  617300  179208
133900  684700  179200
65808   752428  179296
12264   832736  150760
12068   918100  64956
Killed
137|root@generic:/data/misc #
```

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

```
youngzt@ubuntu ~> adb shell
root@generic:/ # cd /data/misc
root@generic:/data/misc # ./test
Start test.
active list      inactive list    free memory
770036  148928  80808
701692  216804  80888
622856  287532  88000
548060  356408  93360
477728  424728  95256
406440  493604  96868
328584  567320  99640
260260  635140  99796
192184  702876  99812
124356  770456  99812
56528   838044  99812
12428   910992  70596
Killed
```

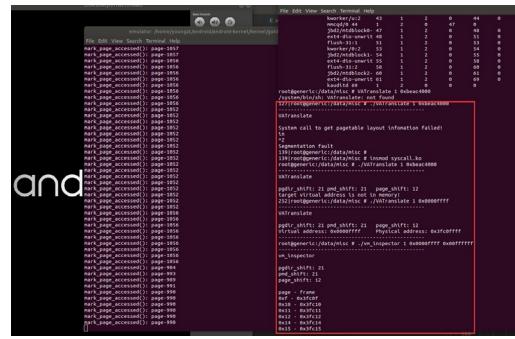
The first thing to find is that when the process gets killed, the free memory is larger if  $K/\log(\text{THRESHOLD})$  is larger. So probably the first one is a better choice.

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;

VATTranslate 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 system . Discussion with classmates is also very 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://bricktoyou.cn/mm/pagewalk\\_walk\\_page\\_range\\_en.html](http://bricktoyou.cn/mm/pagewalk_walk_page_range_en.html)

[http://www.cs.columbia.edu/~kri/os/lectures/I\\_17-LinuxPaging.pdf](http://www.cs.columbia.edu/~kri/os/lectures/I_17-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://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>