HW #3 – Virtual address translation and COW in fork

Overview

In this homework, you need to implement a system call that returns the physical address corresponding to a given virtual address in a process's address space on Linux. If you know how to do this, you can proceed directly to the assignment section (Section III).

In the following, we will first give you some briefing on how to add a new system call in the Linux kernel (Section I). And, we will also show you how address translation takes place in x86_64 Linux kernel (Section II).

I. Adding a new system call in the Linux kernel

Modern operating systems such as Windows and Linux are structured into two spaces: user space and kernel space. Most of the operating system functions are implemented in the kernel. Programs in the user space have to use appropriate system calls to invoke the corresponding kernel functions. In this homework, we will take a closer look at the system call mechanism by tracing system calls made by a user process calls. We will then demonstrate how to implement a new system call on Fedora Linux.

A. Use 'strace' to trace the system calls made by the 'ls' command

1. Use 'strace'

```
$ strace ls 2>& strace.txt
```

2. Open/Cat the output file 'strace.txt' (e.g. Figure 1)

```
execve("/bin/ls", ["ls", "2"], [/* 51 vars */]) = 0
brk(0)                                  = 0x1f93000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5d351d6000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=58372, ...}) = 0
mmap(NULL, 58372, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f5d351c7000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/librt.so.1", O_RDONLY)       = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220!\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=31744, ...}) = 0
mmap(NULL, 2128848, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5d34db100
mprotect(0x7f5d34db8000, 2093056, PROT_NONE) = 0
mmap(0x7f5d34fb7000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/libselinux.so.1", O_RDONLY)  = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20Y\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=117592, ...}) = 0
mmap(NULL, 2217480, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5d34b9300
mprotect(0x7f5d34baf000, 2093056, PROT_NONE) = 0
mmap(0x7f5d34dae000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
mmap(0x7f5d34db0000, 1544, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
close(3)                                = 0
```

Figure 1. screenshot of strace command

3. You can see all the system calls made by the ls command in sequential order. For instance, in Figure 1, we can see that the ls command has invoked the execve, brk, access, and mmap system calls

B. Add a custom system call

1. Download the kernel source (same steps as in **Homework 2**)
2. Add a custom system call to the syscall table (see Figure 2)

```
$ vim [source code directory]/arch/x86/syscall/syscall_64.tbl
```

```
304 common  open_by_handle_at    sys_open_by_handle_at
305 common  clock_adjtime        sys_clock_adjtime
306 common  syncfs           sys_syncfs
307 64  sendmmsg         sys_sendmmsg
308 common  setns            sys_setns
309 common  getcpu           sys_getcpu
310 64  process_vm_readv     sys_process_vm_readv
311 64  process_vm_writev    sys_process_vm_writev

# simple system call
312 common  sayhello         sys_sayhello

#
# x32-specific system call numbers start at 512 to avoi
# for native 64-bit operation.
#
512 x32 rt_sigaction         sys32_rt_sigaction
513 x32 rt_sigreturn         stub_x32_rt_sigreturn
514 x32 ioctl            compat_sys_ioctl
```

Figure 2. add a system call 'sayhello' to syscall table

3. Add the system call definition to the syscall interface (see Figure 3)

```
$ vim [source code directory]/include/linux/syscalls.h
```

```
                                unsigned long
                                unsigned long

// simple system call
asmlinkage long sys_sayhello(void);

#endif
```

Figure 3. add the system call 'sayhello' definition to the syscall interface

4. Implement the custom system call (see Figure 4)

```
$ vim [source code directory]/kernel/sayhello.c
```

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

asmlinkage long sys_sayhello(void) {
    printk(KERN_DEBUG "Hello !\n");
    return 0;
}
```

Figure 4. the system call 'sayhello'

5. Modify the Makefile (e.g. Figure 5)

```
$ vim [source code directory]/kernel/Makefile
```

```
obj-y     = fork.o exec_domain.o panic.o printk.o \
        cpu.o exit.o itimer.o time.o softirq.o resource.o \
        sysctl.o sysctl_binary.o capability.o ptrace.o timer.o user.o \
        signal.o sys.o kmod.o workqueue.o pid.o \
        rcupdate.o extable.o params.o posix-timers.o \
        kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o mutex.o \
        hrtimer.o rwsem.o nsproxy.o srcu.o semaphore.o \
        notifier.o ksysfs.o cred.o \
        async.o range.o groups.o \
        sayhello.o
```

Figure 5. modify the Makefile

6. Make the new kernel

- For a multi-core PC, you can accelerate the kernel make process with the '-j [number of threads]' option.

```
$ make -j 4
```

C. Invoke system call by the system all number (see Figure 6)

1. Include the following header files

```
#include <unistd.h>
#include <sys/syscall.h>
```

2. Use function 'syscall' to invoke system call

```
Usage: syscall(int [syscall number], [parameters to syscall])
```

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

int main() {
    int ret = syscall(312);
    printf("ret: %d\n", ret);
    return 0;
}
```

Figure 6. invoke a system call in a program

- For detailed information of syscall, please check Linux man pages

```
$ man syscall
```

3. After running the code, you can use 'dmesg' to see the messages output from printk (e.g. Figure 7)

```
$ dmesg
```

```
oshw4 [/home/ychsu] -ychsu- % dmesg | tail -n 1
[  724.729489] Hello !
```

Figure 7. the 'printk' messages from 'sayhello' system call

※ You can download the full source code of the examples in the section B and section C here.

II. x86_64 Page Table Structure and Address Translation

When using virtual memory, every process will have its own memory space. For example in Figure 8, the address 0x400254 in process A is pointed to physical address 0x100000 but in process B address 0x400254 may be pointed to physical address 0x300000.

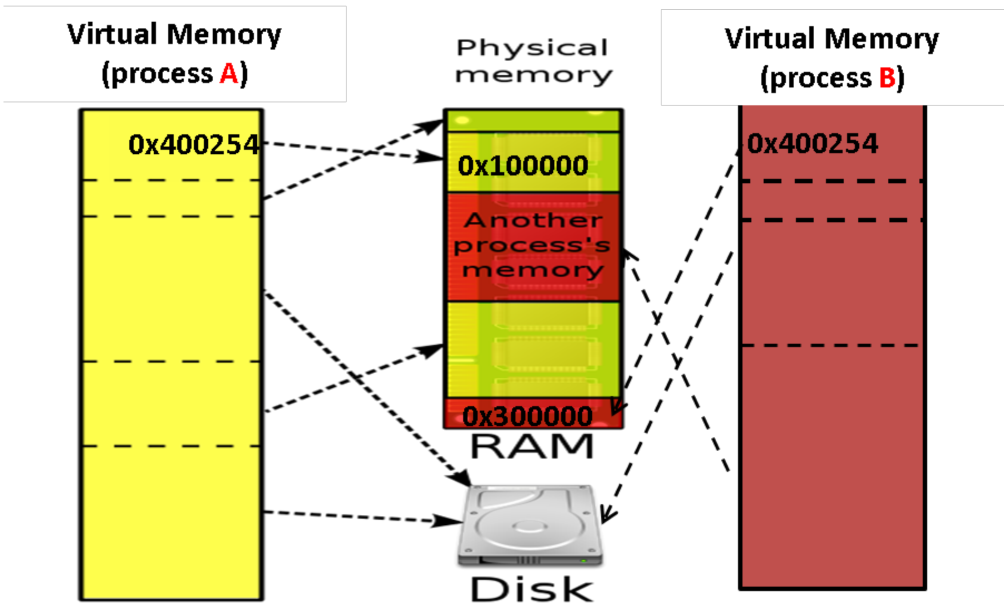

Figure 8. Virtual Memory(Modified from Wikipedia)

A. x86_64 Page Table Structure

We will demonstrate how a virtual address is translated into a physical address on x86_64 architecture with 4KB pages.

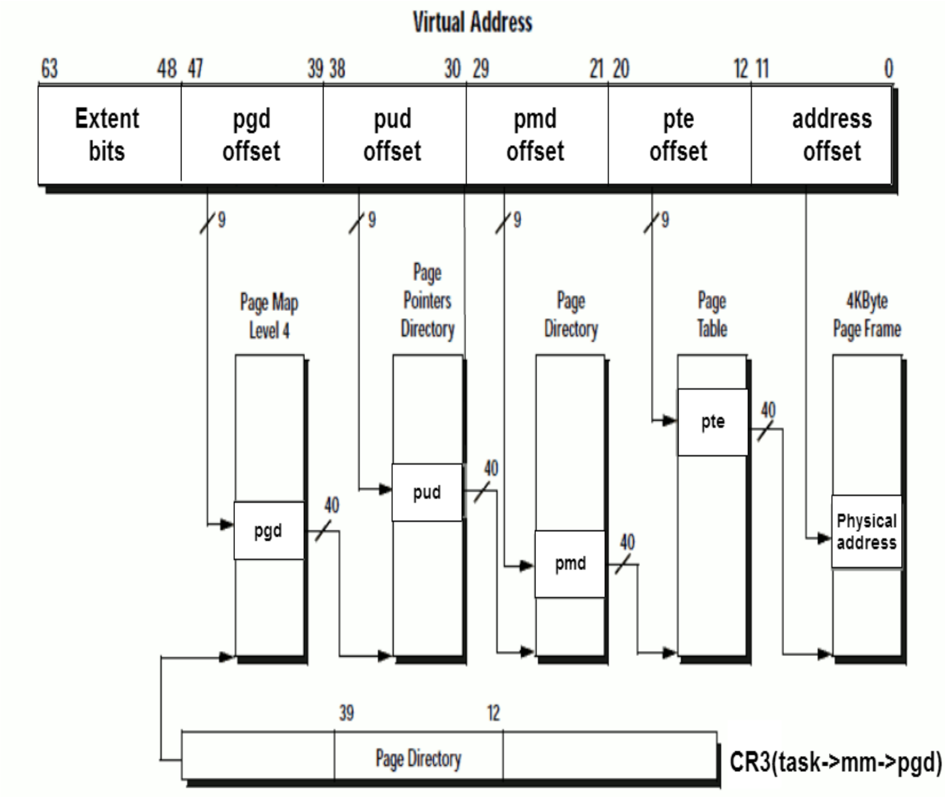

Figure 9. 64 bits page table overview

Figure 9 is the page table structure on x86_64. You can see that there are 4 levels of address translation. Figure 10 shows how a virtual address gets converted to the physical address.

(Note. You can observe that there are 9 bits for each offset(except address offset). This means that there are 512(2^9) entries in one page table (Because each page is 4K bytes, that means each page table entry is 64 bits).

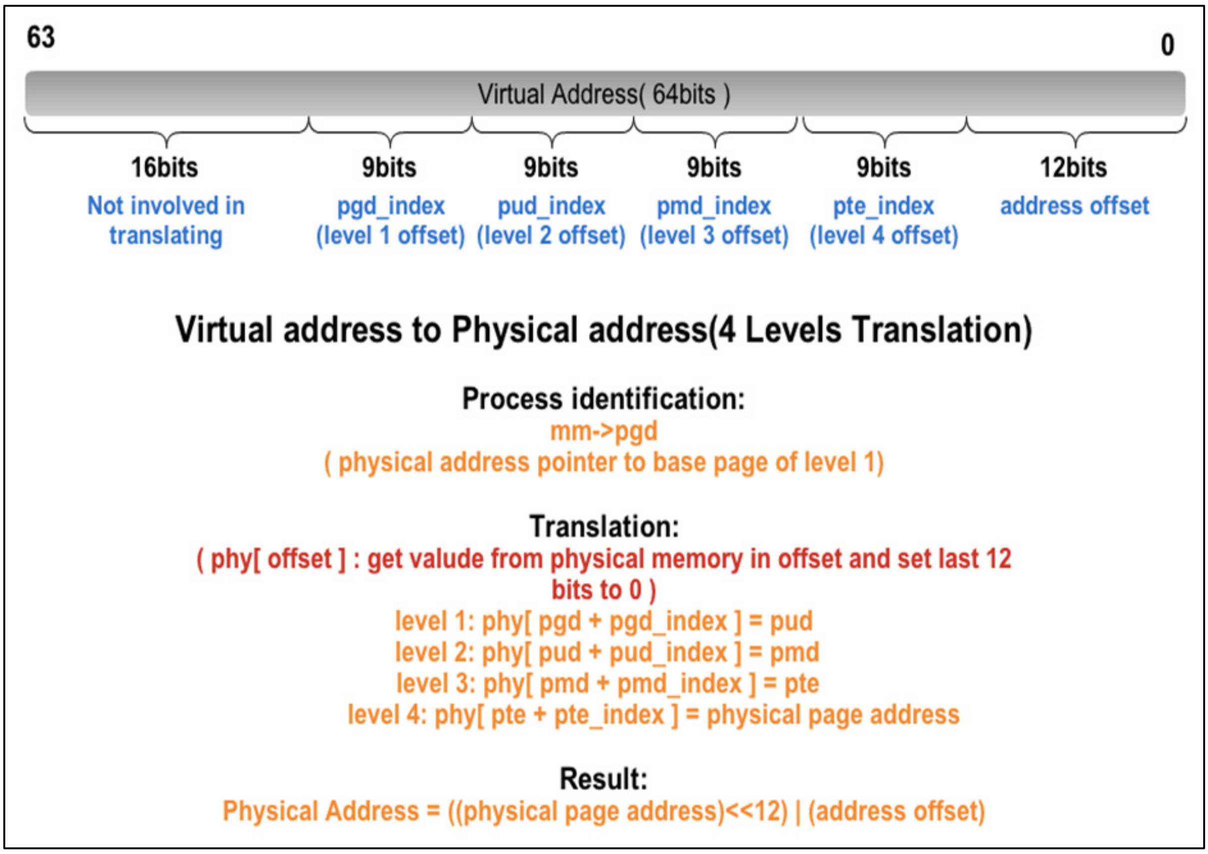

Figure 10. Virtual address to physical address

B. Super Page (2M sized pages)

Not all memory pages are 4K in size. **For instance, the system_call_table is placed on a 2M page,** and a 2M page is often referred to as a super page (as opposed to a 4KB small page). How can we locate a 2M page? It is almost the same as locating a 4k page except that we only need to walk 3 levels of page tables to locate a 2M page. There is no need for the 4th level page table in locating the physical address of a 2MB page, and we can say that the PMD is in fact the PTE for 2MB pages. Linux kernel uses the **int pmd_large(pmd_t *pmd)** function to determine if a PMD points to the 2M page. **If pmd_large() return 0, it means that the page is not a PTE for a 2M page so you will have to walk the**

forth level page table; otherwise, the PMD is the last level of page table of a 2MB page.

C. Address translation functions in Linux kernel

Linux kernel has some useful functions and structures (defined in `arch/x86/include/asm/pgtable.h`) to help translate virtual address to physical address.

```
PGD:
/*strcture of pgd*/
pgd_t *pgd;
/*get pgd from mm and virtual address*/
pgd_t* pgd_offset(struct mm_struct *mm, unsigned long virtual_address);
/*test if pgd is null, 1->null 0->non-null*/
int pgd_none(pgd_t pgd);
/*return pgd value*/
unsigned long pgd_val(pgd_t pgd);

PUD:
/*strcture of pud*/
pud_t *pud;
/*get pud from pgd and virtual address*/
pud_t* pgd_offset(pgd_t *pgd, unsigned long virtual_address);
/*test if pud is null, 1->null 0->non-null*/
int pud_none(pud_t pud);
/*return pud value*/
unsigned long pud_val(pud_t pud);

PMD:
/*strcture of pmd*/
pmd_t *pmd;
/*get pmd from pud and virtual address*/
pmd_t* pmd_offset(pud_t *pud, unsigned long virtual_address);
/*test if pmd is null, 1->null 0->non-null*/
int pmd_none(pmd_t pmd);
/*return pmd value*/
unsigned long pmd_val(pmd_t pmd);

PTE:
/*stucture of pte*/
pte_*t pte;
/*get pte from pmd and virtual address*/
pte_t *pte_offset_kernel(pmd_t *pmd, unsigned long virtual_address);
/*test if pte is null , 1->null 0->non-null*/
int pte_none(pte_t pte);
/*return pte value*/
unsigned long pte_val(pte_t pte);
```

Figure 11. Functions of address translation in Linux

```c
#define PID 1000
#define VADDR 0x400100
pgd_t *pgd;
pud_t *pud;
struct task_struct *p;
p = pid_task(find_vpid(PID), PIDTYPE_PID);
pgd = pgd_offset(p->mm, VADDR);
printk("pgd_val = 0x%lx\n", pgd_val(*pgd));
if (pgd_none(*pgd)) {
    printk("not mapped in pgd\n");
    return -1;
}
pud = pud_offset(pgd, VADDR);
```

Figure 12. Example of address translation

Figure 12 is an example of how to lookup the first level page table. The rest of translation is pretty much the same.

III. Assignment

You've learned in the class that the `fork` system call can be used to create a child process. In essence, the `fork` system call creates a separate address space for the child process. The child process has an exact copy of all the memory segments of the parent process. The copying is obviously a time consuming process. As a result, to reduce the overhead of memory copying, most `fork` implementation (including the one in Linux kernel) adopts the so-called copy-on-write strategy. The memory pages of the child process are initially mapped to the same physical frames of the parent process. Only when a child process memory page is about to be overwritten, will a new physical copy of that be created, so the modification on that page by one process will not be seen by the other process.

In this homework, you are asked to verify the copy-on-write behavior of `fork` system call. Specifically, you need to complete two tasks:

Task 1

Implement a custom system call to translate a virtual address to physical address.

You first need to implement a system call that translates a virtual address to the corresponding physical address. The inputs are `pid` (process id) and a virtual address. A template (named lookup_paddr.c) will help you complete the task. You just need to add the necessary code in it, integrate the template file into the kernel source, and rebuild the kernel. You can then test the effect of the system call following the same steps in Section I.

Task 2

Verify that "fork" uses copy-on-write in the creation of child process address space.

The fork system call creates a child process and duplicates the memory segments for the parent process for the child. To verify that fork uses copy-on-write, we can observe the mapping between the virtual addresses and the physical addresses for both the parent and the child.

```
Parent pid: 1486. [Var 'mem_alloc']vaddr: 0x1f21010, val: 1000
Child pid: 1487.  [Var 'mem_alloc']vaddr: 0x1f21010, val: 1000

*** Modify variable 'mem_alloc' from 1000 to 1 ***

Parent pid: 1486. [Var 'mem_alloc']vaddr: 0x1f21010, val: 1000
Child pid: 1487.  [Var 'mem_alloc']vaddr: 0x1f21010, val: 1
```

Figure 13. Expected output from running fork_ex.c

Figure 13 is an expected output from running fork_ex.c, if fork does use copy-on-write.

The virtual addresses of parent and child processes are initially the same. This is as expected. After the child modifies the value of the variable mem_alloc we can see that the memory pages of the parent and the child processes bear different values. However, their virtual addresses are still the same.

To verify the use of copy-on-write, we need to check if the two processes share the same physical frame before modifying the mem_alloc variable and use different memory frames for the variable mem_alloc after the variable value is overwritten.

```
Parent pid: 1539. [Var 'mem_alloc']vaddr: 0xa37010, paddr: 0x8000000104a63010, val: 1000
Child pid: 1540.  [Var 'mem_alloc']vaddr: 0xa37010, paddr: 0x8000000104a63010, val: 1000

*** Modify variable 'mem_alloc' from 1000 to 1 ***

Parent pid: 1539. [Var 'mem_alloc']vaddr: 0xa37010, paddr: 0x8000000104a63010, val: 1000
Child pid: 1540.  [Var 'mem_alloc']vaddr: 0xa37010, paddr: 0x8000000104abf010, val: 1
```

Figure 14. Expected physical address mapping change after overwriting `mem_alloc` variable

Figure 14 shows that the physical addresses to the memory page containing the `mem_alloc` variable are different after child process changes the value of `mem_alloc`. However, you can notice that the parent and the child processes did share the same physical frame at the beginning. It indicates the use of copy-on-write mechanism.

You have to complete the implementation of `fork_ex.c` to show that `fork` does use copy-on-write mechanism. The template source file is `fork_ex.c` provided for you fill-in the missing code.

Please pack

(1) The finished `lookup_paddr.c`
(2) The finished `fork_ex.c`
(3) A document in PDF(or DOC) to describe the implantations details

into a single RAR file. Submit the RAR file to E3.

If you have any questions, please direct your questions to

Ian Chen (陳義永)
E-mail: chen.yi-yung@livemail.tw
Office location: Microelectronics Building 7th floor Room E07

Alternative choice of operating system

You may do this homework with a different type of operating system such as Windows, OS X, Android, BSD, etc.

In this case, you will have to submit the complete source code and build instructions for us to verify its correctness.