Linux OS Project 2 Write Up

[TOC]

資訊

隊伍

- 隊名：第 48763 組

成員

- 張桓融
  - 資訊工程學系四年級
  - 108502515
- 洪裕翔
  - 資訊工程學系四年級
  - 108502520
- 李宗奇
  - 資訊工程學系四年級
  - 108502578

老師指定內容

推測結果

執行 Syscall 前，我們試著印出各 Segment 的變數之虛擬位址，結果如下:

```
Child Process
Virtual Address:
0x5060a010        // Data Seg
0x5060a028        // BSS Seg
0x5171c2a0        // Heap Seg
0xd636770         // Lib Seg
0x506071e9        // Text Seg
0x73ceeac0        // Stack Seg before fork
0x73ceeac4        // Stack Seg after fork
Parent Process
Virtual Address:
0x5060a010        // Data Seg
0x5060a028        // BSS Seg
0x5171c2a0        // Heap Seg
0xd636770         //Lib Seg
0x506071e9        //Text Seg
0x73ceeac0        // Stack Seg before fork
0x73ceeac4        // Stack Seg after fork
```

我們發現在父子 process 中的變數虛擬位址皆相同，推測出可能有兩種情況：

1. 父子 Process 中的變數的實體位址皆是相同的。
    - 不太可能，由上課內容可知，理論上不同的 Process 會有獨立記憶體區段，不會共用同一塊 Memory Space。
2. 父子 Process 中的變數只是虛擬位址相同，經轉換後會對應到不同的實體位址。

我們認為第二種情況是最有可能的，也就是說不同 Process 各自擁有 memory descripter 管理自己的獨立記憶體區段，因此各 VMA 對應到的實體位址便不一樣。

實際結果

```
Child Process
Virtual Address:
0x804a034        // Data Seg
0x804a040        // BSS Seg
0x9586008        // Heap Seg
0x80483d0        // Lib Seg
0x804855b        // Text Seg
0xbf86a714       // Stack Seg before
0xbf86a718       // Stack Seg after
Physical Address:
0xfcb06034        // Data Seg
0xfcb06040        // BSS Seg
0xfb1ff008        // Heap Seg
0xfb1f83d0        // Lib Seg
0xfb1f855b        // Text Seg
0xfc846714        // Stack Seg before fork
0xfc846718        // Stack Seg after fork
Parent Process
Virtual Address:
0x804a034         // Data Seg
0x804a040         // BSS Seg
0x9586008         // Heap Seg
0x80483d0         // Lib Seg
0x804855b         // Text Seg
0xbf86a714        // Stack Seg before
0xbf86a718        // Stack Seg after
Physical Address:
0xfc84b034        // Data Seg
0xfc84b040        // BSS Seg
0xfc3cc008        // Heap Seg
0xfb1f83d0        // Lib Seg
0xfb1f855b        // Text Seg
0xfc847714        // Stack Seg before
0xfc847718        // Stack Seg after
```

由上面的結果可知，fork() 後的兩個 Processes 除了 Library Segment 和 Text Segment 之外，其餘 VMA 的 Physical Memory Address 皆會相同。換言之，不同 Process 之間只有 Library Segment 和 Text Segment 會共用 Physical Memory，與我們的推測結果不完全相同。

Source Codes

User Code

```
#include<stdio.h>
#include<unistd.h>


#define ADDR_SIZE 7


int data_seg = 123;
int bss_seg = 0;
int *heap_seg;
void *lib_seg = &printf;
void text_seg() {}

int main() {
    heap_seg = malloc(10);
    int stack_seg_bf = 10;

    pid_t pid = fork();

    int stack_seg_af = 11;

    if(pid == 0) {
        printf("child process\n");
    }
    else {
        sleep(2);
        printf("parent process\n");
    }

    int in_addr[ADDR_SIZE] = {
        &data_seg
        , &bss_seg
        , heap_seg
        , lib_seg
        , &text_seg
        , &stack_seg_bf
        , &stack_seg_af
    };
    int out_addr[ADDR_SIZE];

    printf("Virtual Address:\n");
    for(int i = 0; i < ADDR_SIZE; i++) {
        printf("%p\n", in_addr[i]);
    }

    syscall(352, in_addr, ADDR_SIZE, out_addr);

    printf("Physical Address:\n");
```

```
    for(int i = 0; i < ADDR_SIZE; i++) {
        printf("%p\n", out_addr[i]);
    }

    return 0;
}
```

Kernel Code

```
#include <linux/init_task.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/string.h>
#include <linux/uaccess.h>
#include <linux/mm.h>
#include <linux/linkage.h>
#include <linux/highmem.h>
#include <linux/gfp.h>


asmlinkage unsigned long sys_vtop(
        unsigned long *initial
        , int len_vir
        , unsigned long *result) {
    pgd_t *pgd;
    pud_t *pud;
    pmd_t *pmd;
    pte_t *pte;

    unsigned long *v_addr = kmalloc(len_vir*sizeof(unsigned long), GFP_KERNEL);
    unsigned long *p_addr = kmalloc(len_vir*sizeof(unsigned long), GFP_KERNEL);

    copy_from_user(v_addr, initial, len_vir*sizeof(unsigned long));

    printk("cpaddr[0]:%p\n", v_addr[0]);
    printk("cpaddr[1]:%p\n", v_addr[1]);
    printk("cpaddr[2]:%p\n", v_addr[2]);
    printk("cpaddr[3]:%p\n", v_addr[3]);
    printk("cpaddr[4]:%p\n", v_addr[4]);
    printk("cpaddr[5]:%p\n", v_addr[5]);

    struct page *page;

    for(int i = 0; i < len_vir; i++) {
        pgd = pgd_offset(current->mm, v_addr[i]);
        printk("pgd_val = 0x%lx\n", pgd_val(*pgd));
        printk("pgd_index = %lu\n", pgd_index(v_addr[i]));

        if (pgd_none(*pgd)) {
            printk("Not mapped in pgd!\n");
            return -1;
```

```
        }

        pud = pud_offset(pgd, v_addr[i]);
        printk("pud_val = 0x%lx\n", pud_val(*pud));
        printk("pud_index = %lu\n", pud_index(v_addr[i]));

        if (pud_none(*pud)) {
            printk("Not mapped in pud!\n");
            return -1;
        }

        pmd = pmd_offset(pud, v_addr[i]);
        printk("pmd_val = 0x%lx\n", pmd_val(*pmd));
        printk("pmd_index = %lu\n", pmd_index(v_addr[i]));

        if (pmd_none(*pmd)) {
            printk("Not mapped in pmd!\n");
            return -1;
        }

        pte = pte_offset_map(pmd, v_addr[i]);
        printk("pte_val = 0x%lx\n", pte_val(*pte));
        printk("pte_index = %lu\n", pte_index(v_addr[i]));

        if (pte_none(*pte)) {
            printk("Not mapped in pte!\n");
            return -1;
        }

        page = pte_page(*pte);
        pte_unmap(pte);
        p_addr[i] = (page_to_phys(page) & PAGE_MASK) | (v_addr[i] & ~PAGE_MASK);
        printk("p_addr: %p\n", p_addr[i]);
    }

    copy_to_user(result, p_addr, len_vir*sizeof(unsigned long));

    return 0;
}
```

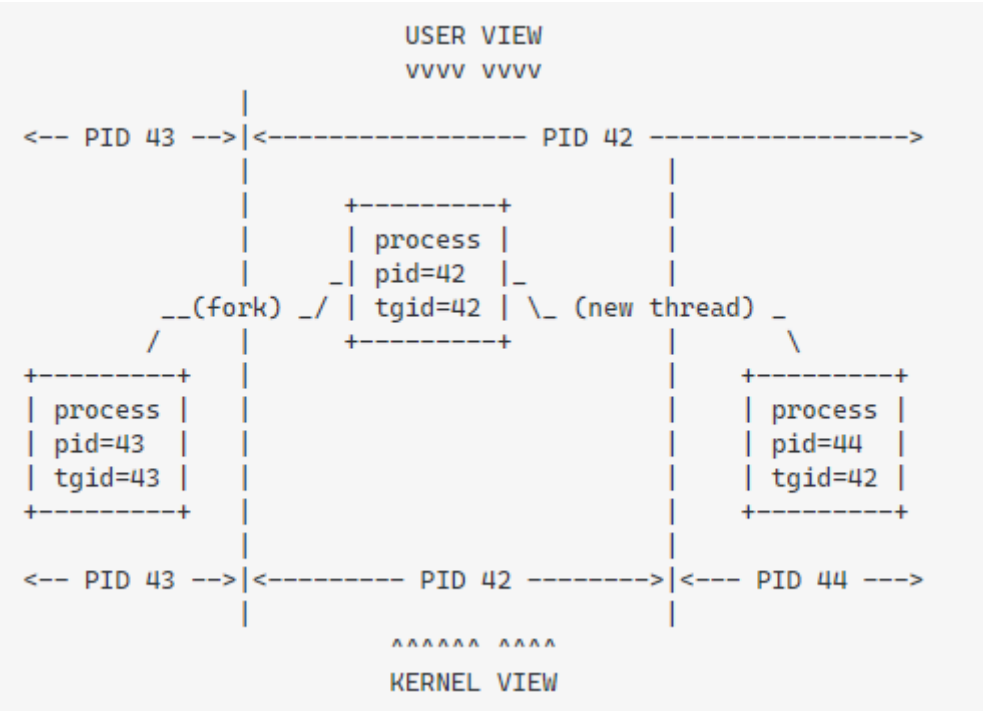
Kernel 與 OS 版本

- Kernel: 3.10.108
- OS: Ubuntu 16.04.7 LTS

問題們

Q1. Process 和 Thread 的差異？

- PID 與 TPID
    - 一個 Process 會有相同的 PID 和 TGID，而 `fork()` 後得到的兩個 process PID 不同。

- Thread 會有和原 process 相同的 TGID，但有不同的 PID。
- 

> References:
>
> - multithreading - If threads share the same PID, how can they be identified? - Stack Overflow

- 共享的記憶體區段
  - Process 只有 Library Segment 和 Text Segment 共享，其餘 VMA 皆獨立。
  - Thread 共享除了 Stack Segment 以外的所有 VMA。
- System Call
  - Process 最後呼叫到 `do_fork()` 函式時，傳入參數 (clone_flag) 並沒有預設值，因此 Kernel 會另外分配新的記憶體空間給新創建的任務使用。
  - Thread 最後呼叫到 `do_fork()` 函式時，傳入參數 (clone_flag) 則有預設值，因此新創建的任務會繼承父 Process 的 VMA，造成了第二點的共享記憶體區段差異。

簡單總結，Process 和 Thread 的管理方式類似，但 Process 比較像是 Thread 的頂級域，也就是 1 個 Process 中可以有 N 個 Thread，N 為正整數，反之則不成立。

Q2. 為什麼 fork() 後的兩個 Process 能共享 Library Segment 和 Text Segment？

我們參考了上面這個問題的解答，嘗試推測為何不同 Process 之間能共享 Library Segment 和 Text Segment 的原因（可能不正確）。 主要的原因是 Linux 為了優化記憶體的分配，便以 Copy on Write 為原則進行設計。 Copy on Write 簡單來說就是當被分配的記憶體區塊有寫入的動作時，才將記憶體區塊複製，形成獨立區塊，反之則保持共享。 這在解釋為何不同 Process 之間能共享 Library Segment 和 Text Segment 是合理的，因為一般的程式在執行時，基本上不太會對存放 Library 的 Library Segment 和存放 Code 的 Text Segment 進行寫入，而其他記憶體區塊則會，因此 Copy on Write 可以解釋兩記憶體區段被共享的現象。 然而，這裡有部分我們尚不知如何解釋的問題。 如 Copy on Write 的定義所言，OS 只有在偵測到記憶體有被寫入的動作時，才會將該區段複製以形成獨立區塊。 但在我們撰寫的 User Space Code 中，代表 BSS Segment 和 Heap Segment 的變數都是沒有經過 Initialize 的，這意味的此二變數只有 Declaration，沒有 Initialization，而沒有

Initialization 就代表著此變數並沒有被寫入過，依照 Copy on Write 的定義，此二記憶體區段應該也要被 Process 共享，但實驗結果顯示事實並非如此。 這是我們目前尚不知道如何解釋的問題。

> References:
>
> - linux - Which segments are affected by a copy-on-write? - Stack Overflow

Q3. sleep() 執行時會把 process 的狀態改成什麼？

在kernel 3.10.108 中 task 的 state 會通常有三種情況：

1. TASK_RUNNING
2. TASK_INTERRUPTIBLE
3. TASK_UNINTERRUPTIBLE `sleep()` 會去呼叫 `nanosleep()`，在 `nanosleep()` 中，會把 task 的 state 改成 TASK_INTERRUPTIBLE，也就是可以透過 signal 等相關的 interrupt 的 function，去中止這個 `sleep()`。

> References:
>
> - sleep(3) — Linux manual page
> - Linux中的休眠函数
> - unistd.h source code [include/unistd.h] - Codebrowser

Q4. sleep() 是什麼時候真正做到 do_sleep() 這個動作的？

根據 sleep(3) - Linux manual page，我們可以得到一張call sleep的關係表：

而 `hrtimer_nanosleep()` 是在設定完 `hrtimer` 這個 timer 的資料結構之後才真正去 call `do_nanosleep()`。

Q5. sleep() 的 timer 在哪裡設定？

`hrtimer_nanosleep()` 和 `do_nanosleep()` 都是去維護 `hrtimer_sleeper` 的資料結構：

```
struct hrtimer_sleeper {
    struct hrtimer timer;
    struct task_struct *task;
};
```

而 `sleep()` 的timer就是去設定 `hrtimer_sleeper` 內的 `hrtimer`。

Q6. 當程序在 sleep 時，要如何將其回復（使用什麼 signal 實現）？

`sleep()` 會有計時器用於計算 process 暫停的時間，當計時器遞減至零時，會發出 `SIGALRM` signal，而 `sleep()` 正是以 `SIGALRM` signal 作為當前是否要回復 process 的判斷依據。

> References:
>
> - 【Linux】定时器发出的SIGALRM信号与sleep、usleep、select、poll等函数冲突的解决办法

Q7. signal 主要的用途為何？

signal 用於通知 process 發生非同步事件，作用的雙方可以是 process 之間，也可以為 kernel 與 process。

Q8. fork() 會呼叫什麼 system call？

在我們使用的 Linux kernel version (3.10.108) 中，`fork()` 最後會使用的是 `do_fork()`，process 和 thread 的差別只在於傳入 `do_fork()` 的參數 (clone_flags) 不同。

Q9. mmap 這個 system call 在幹嘛？

這個系統呼叫用於將實體資料映射到當前進程的虛擬內存空間 (VMA) 內。 在 `mmap` 的 `flags` 參數欄位部分有多種映射方式能選擇，以下只列較基本的兩個：

1. MAP_PRIVATE: 此映射方式會在寫入時產生一個映射資料的複製，也就是 `copy on write`，複製後的區塊與原映射資料是獨立的。
2. MAP_SHARED: 進程的寫入真的會寫入映射資料空間，且允許不同進程共享這塊空間。

> References:
>
> - linux .so mmap,linux中mmap系统调用原理分析与实现_老班长-宫晓的博客-CSDN博客
> - mmap(2) - Linux manual page

Q10. 有哪些原因會造成 page fault？

原因主要可以分為三類：

1. page 被放入記憶體但 MMU 還沒轉址成功，導致 CPU 需要使用資料時卻沒有對應的 page（也就是位址非法），需要先中斷 process 去處理，便會發生 page fault（例如 malloc 記憶體但還沒分配實際記憶體），可能發生於兩個（或多個）process 共享記憶體，但只有其一有轉址。
2. process 存取非法的 memory segment 時，因為沒有存取權限（read only）而觸發 page fault。
3. page 沒有被載入到 memory，需要去觸發 IO 把 page 傳到 memory。

Q11. 原本的 page table 在 fork() 後，經過 copy on write 會多幾份 page table？

產生 child proces 時，需要為其建立一個 task_struct，並且會複製 mm_struct 及 page table，因此根據 child process 數量決定 page table 的數量。