

# Linux OS Project 1 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

### Execution Results

```
1 Segment Name: text, Virutal Address: 0x80485ec, Physicall Address: 0x28005ec
2 Segment Name: data, Virutal Address: 0x804a030, Physicall Address: 0x30
3 Segment Name: bss, Virutal Address: 0x804a03c, Physicall Address: 0x3c
4 Segment Name: heap, Virutal Address: 0x9c2d008, Physicall Address: 0x8
5 Segment Name: libraries, Virutal Address: 0x8048470, Physicall Address: 0x2800470
6 Segment Name: text, Virutal Address: 0x80485ec, Physicall Address: 0x28005ec
7 Segment Name: data, Virutal Address: 0x804a030, Physicall Address: 0x30
8 Segment Name: bss, Virutal Address: 0x804a03c, Physicall Address: 0x3c
9 Segment Name: heap, Virutal Address: 0x9c2d008, Physicall Address: 0x8
10 Segment Name: libraries, Virutal Address: 0x8048470, Physicall Address: 0x2800470
11 Segment Name: stack, Virutal Address: 0xb6de7320, Physicall Address: 0xf7417320
12 Segment Name: stack, Virutal Address: 0xb75e8320, Physicall Address: 0x2800320
```

圖一，兩個 thread 分別尋物理位址的結果。第 1 - 5、11 行為 thread 1 的輸出；第 6 - 10、12 行則為 thread 2 的輸出。

## Results Describing and Analyzing of Experiments

表一為實驗結果的整理。由表一可見，兩個不同的 thread 分別共用的 segment 有 text 段、data 段、bss 段、heap 段、libraries 段（這些 segment 的變數的 physical address 在不同 thread 之間是相同的），而 stack 段則是不一樣的。代表說，這裡面所宣告的是不一樣的變數，這裡就把它視為不同的segment。

Segments	Thread 1	Thread 2
text	0x28005ec	0x28005ec
data	0x30	0x30
bss	0x3c	0x3c
heap	0x8	0x8
libraries	0x2800470	0x2800470
stack	==0xf7417320==	==0x2800320==

表一，實驗結果整理。

Source Codes

Multi-Threads Application

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <syscall.h>
#include <sys/types.h>
#include <time.h>
#define arr_size 6

typedef struct {
    char *segment_name;
    void *address;
} data_format;

int for_data_segment = 48763;
int for_bss_segment = 0;
int *for_heap_segment;
void *for_libraries_segment = &printf;
void for_text_segment() {}

void* child_thread(void) {
    int for_stack_segment = 357;
    data_format items[] = {
        {"text", &for_text_segment}
        , {"data", &for_data_segment}
        , {"bss", &for_bss_segment}
        , {"heap", for_heap_segment}
        , {"libraries", for_libraries_segment}
```

```

        , {"stack", &for_stack_segment}
    };

    for(int i = 0; i < (sizeof(items)/sizeof(items[0])); i++) {
        unsigned long virtual = items[i].address;
        unsigned long physical = syscall(351, virtual);

        printf("Segment Name: %s, Virutal Address: %p, Physical Address: %p\n"
               , items[i].segment_name
               , items[i].address
               , (void *)physical
            );
    }

    pthread_exit(NULL);
}

int main(void) {
    for_heap_segment = malloc(10);
    for_libraries_segment = &printf;

    pthread_t pt_1 = 0;
    pthread_t pt_2 = 0;

    pthread_create(&pt_1, NULL, (void*)child_thread, "pt_1");
    sleep(1);
    pthread_create(&pt_2, NULL, (void*)child_thread, "pt_2");

    pthread_join(pt_1, NULL);
    pthread_join(pt_2, NULL);

    return 0;
}

```

## Kernel

```

#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_virt_to_phy(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));

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

    printf("pgd_val = 0x%lx\n", pgd_val(*pgd));
    printf("pgd_index = %lu\n", pgd_index(v_addr[i]));

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

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

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

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

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

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

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

```

```
    }

    page=pte_page(*pte);
    pte_unmap(pte);
    p_addr[i]=page_to_phys(page);

    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

## Kernel 編譯過程

1. 使用以下指令安裝套件：

```
sudo apt-get install libncurses5-dev
```

2. 創建`config`，這邊我們使用預設的 `config`：

```
sudo make oldconfig
```

3. 無情開編，記得加上`-j`參數，多線程編譯會快很多：

```
sudo make -j 4
```

4. 編譯完成後，使用以下指令安裝 `kernel`：

```
sudo make modules_install install
```

5. 若為第一次編譯完成，將`/etc/default/grub`內的以下兩行註解掉，才有開機選單可以用：

```
GRUB_HIDDEN_TIMEOUT=0
GRUB_HIDDEN_TIMEOUT_QUIET=true
```

## 6. 更新grub設定：

```
sudo update-grub
```

## 7. 重開機，選擇新編譯的 kernel 版本即可。

# 新增 system call 過程

1. 創建一個裝 system call 的資料夾，並編寫一個 system call。我們將資料夾名取為modohiyaku，檔名則取為kirito.c。kirito.c內容如下：

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

asmlinkage long sutabasuto_sutorimu(void) {
    printk("Give me ten seconds!\n");
    return 48763;
}
```

2. 在modohuyaku資料夾內新增一個Makefile，內容如下：

```
obj-y := kirito.o
```

3. 返回上一層目錄，編輯Makefile，修改部分內容，將下方第一行修改成第二行（即增加我們自己新建的 system call 資料夾）。

```
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ modohiyaku/
```

4. 編輯arch/x86/syscalls/syscall\_32.tbl，將我們自定義的 system call 寫入 system call table。在文件的最後加上下面內容：

```
{PID}    i386  modohiyaku  sutabasuto_sutorimu
```

5. 編輯include/linux/syscalls.h，讓我們寫的程式可以呼叫我們自定義的 system call。在#endif前加上下面內容：

```
asmlinkage int modohiyaku(void);
```

## 問題們

## 1. pgd\_offset 實作細節？

- 根據目前進程的 mm\_struct 和 virtual addr (with pgd\_index function) · 計算出當前的 pgd\_offset。
- [Solution Source](#)

## 2. p4d 層是否真的存在？

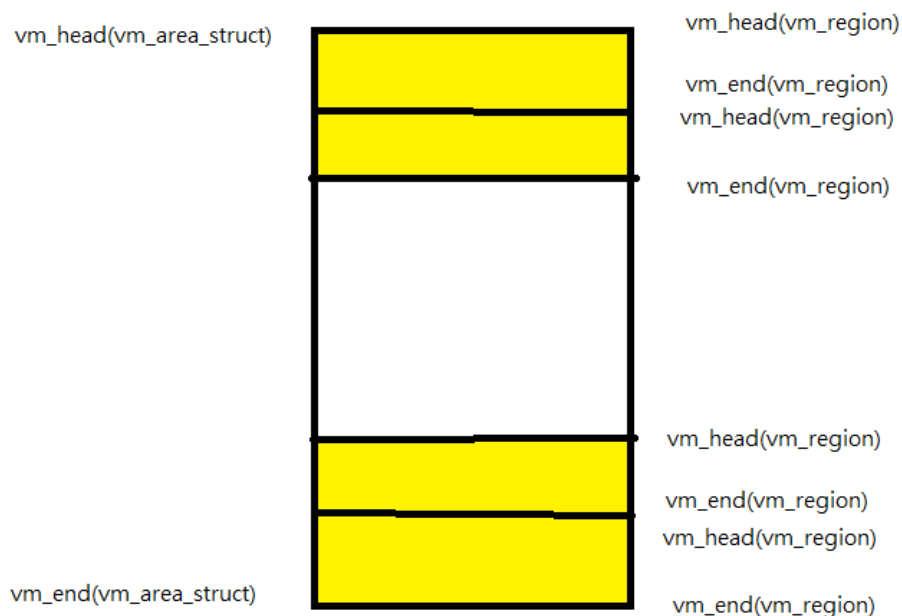
- 對於使用 5 級的系統 (kernel 4.14 up), p4d 存在。
- 對於使用 4 級的系統 · p4d\_offset 可以直接將先前傳進來的第一級 (pgd\_offset) 傳給下一層 · 造成 p4d 物理上不存在的樣子。

## 3. current 為什麼可以不宣告直接用

- 在 `asm/current.h` 進行宣告 · 為一個 macro · 實際上為一個 `get_current` 的 function。
- 詳情請看第5點。

## 4. mm\_struct 中的 mmap

- 是什麼？
  - 為 vm\_area\_struct (VMA)
  - 而這個結構本身為雙向的 linked list (有 vm\_next & vm\_prev)
- VMA 中的 start end 分別代表什麼？
  - 如果說是 vm\_region 內的 vm\_start 以及 vm\_end 話 · 代表的是這一個 segment 可以用的頭跟尾 · 然後由一個 vm\_top 來管控目前 allocate 的位置
  - 如果說是 vm\_area\_struct 內的 vm\_start 跟 vm\_end 的話 · 則代表的是整個 memory descriptor (mm\_struct) 的最前面的位置和最後的位置
  - 猜測一個 process 的 memory 與 vm\_start、vm\_end 的關係圖如下



## 5. current 是什麼？指向哪？

- 在 `x86/include/asm/current.h` 宣告 `#define current get_current()` · 因此我們可以知道這個為一個 function (`get_current()`) 的 macro)。

```

struct task_struct;

DECLARE_PER_CPU(struct task_struct *, current_task);

static __always_inline struct task_struct *get_current(void)
{
    return this_cpu_read_stable(current_task);
}

```

- 由上面的程式我們可以知道，`get_current()` 是把現在這個 `current_task` (由 `DECLARE_PER_CPU` 取得)這個 `per_cpu_variable` 轉成 `task_struct` 回傳。
- 而從 `task_struct` 我們可知 `task` 是一個樹狀結構，也就是老師上課所說的在 Linux 內，所有的 `process` 是會組合成一棵樹，也可以合理的推斷，這裡的 `task_struct` 就是紀錄該 `process` 的所有東西，包含所謂的樹狀結構。
- 小結論:
  - `current` 是一個 function，最終會回傳一個 `task_struct`，也就是 `process` 的資訊。
  - 而包括 `pid`, `mm_struct` (memory descriptor) 等跟 `process` 相關的東西都在這裡。
- References
  - [What is the "current" in Linux kernel source?](#)
  - [The implementation of Linux kernel current macro](#)
  - [What does value of 'current\\_task' mean in different version of linux kernel?](#)
  - [task\\_struct source](#)
  - [current on x86 source](#)

## 6. Kernel 5.x 怎麼增加 System Call?

- `arch/x86/entry/syscalls/syscall_64.tbl` 中新增 `syscall number`。
- 在 `include/linux/syscalls.h` 中新增 `function`。

## 7. pthread 有用到那些System call?

- 會用到了 `mmap`, `clone`, `mprotect` 等等 `system call`。
- [Solution Source](#)

## 8. 為什麼要用 copy-to-user 和 copy-from-user ?

- 兩者與 `memcpy` 相比，多了判斷 `addr` 是在 `user` 端或是 `kernel` 端。在 `MMU` 支援的平台上，傳進來的 `addr` 有可能是指向 `virtual addr`，如果 `virtual` 還沒有指向 `physical` 成功時，我們無法知道這件事，所以才會使用 `copy_(to|for)_user`，去確保訪問的 `addr` 是被合法 `allocate` 的 (在 `valid VMA` 中)。