# Linux OS Project 3 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


## 專案指定內容

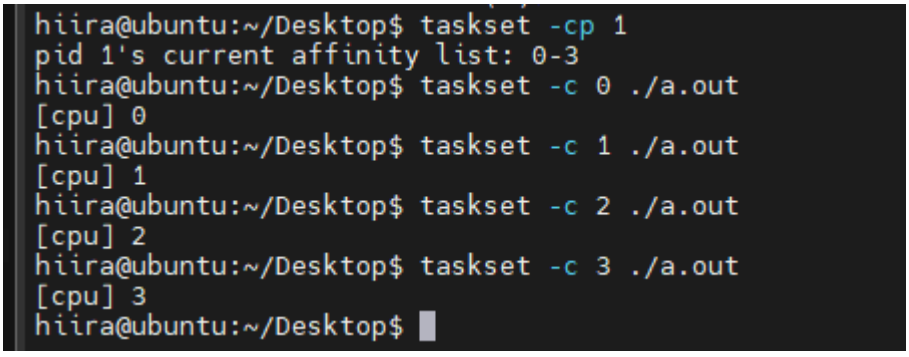
### 第一部分：取得當前處理程序的 CPU 編號

本部分的要求為取得當前處理程序的 CPU 編號。

我們撰寫了一個 system call (`sys_get_cpu_id()`) 以達成本部分要求，在這個 system call 中，我們透過 `current_thread_info()` 取得當前程序的執行資訊，其中便包含本部分要求的 CPU 編號。

除此之外，我們使用 `taskset` 命令指定執行程序的 CPU 編號，透過檢查執行前指定的 CPU 編號與執行後我們撰寫之 system call 顯示之結果，驗證我們取得的 CPU 編號是否正確。

實驗結果如圖一所示，能發現我們依序指定 CPU 0 ~ 4 執行程序，而我們撰寫之 system call 都能正確地輸出當前執行程序的 CPU 編號。




> 圖一，取得當前處理程序的 CPU 編號之執行結果。

## Source Codes

### User Space

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


#define __NR_GET_CPU 351


unsigned int call_cpu_id() {
    return syscall(__NR_GET_CPU);
}


int main() {
    unsigned int id = syscall(351);
    printf("[cpu] %u\n", id);

    return 0;
}
```

### Kernel Space

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


asmlinkage unsigned int sys_get_cpu_id(void) {
    struct thread_info *ti;
    ti = current_thread_info();

    unsigned int cpu_id;
    cpu_id = ti->cpu;

    return cpu_id;
}
```

## 第二部分：計算程序執行中 CPU 轉換的次數

本部分的則要求為取得 CPU-bound 與 I/O-bound 兩支程式執行時的 CPU 轉換次數。

我們首先在 `task_struct` 中新增了 `process_switch_counter` 變數，用於統計程序執行時的 CPU 轉換次數。

接著，我們撰寫了兩個 system call (`sys_start_count_number()`、`sys_end_count_number()`)。在 `sys_start_count_number()` 中，我們將 `current->process_switch_counter` 的值設為零；在 `sys_end_count_number()` 中，我們則取得 `current->process_switch_counter` 的值取出並回傳。

最後，我們依序在 CPU-bound 和 I/O-bound 兩個 user space 的程式上測試，結果如圖二、圖三所示。兩個程式的結構大致相同，都是使用一個迴圈，使兩個程式都能在指定時間內測量 CPU 轉換的次數，兩者的差別在於：CPU-bound 在迴圈內不斷執行加法，目的為實驗 CPU 負擔較大時的 CPU 的轉換情況；I/O-bound 則是在迴圈內不斷印出字串，目的為實驗 I/O 負擔較大時的 CPU 的轉換情況。

實驗結果顯示，I/O-bound 會比 CPU-bound 發生多很多的 content switch，我們推測這是由於 I/O-bound 會讓當下執行程序的 CPU 產生閒置，因此當系統需要資源執行其他任務時，便會為了使用當前 CPU 之資源而將 I/O-bound 程序移至其他 CPU，從而導致 I/O-bound 有非常多的轉換次數。

```
hiira@ubuntu:~/Desktop$ vim cpu_bound.c
hiira@ubuntu:~/Desktop$ gcc cpu_bound.c
hiira@ubuntu:~/Desktop$ ./a.out

During the pass time the process makes 208 times process switches.
hiira@ubuntu:~/Desktop$
```

圖二，CPU-bound 之執行結果。

```
0 ][204481 ][204482 ][204483 ][204484 ][204485 ][204486 ][204487 ][204488 ][204489
 ][204490 ][204491 ][204492 ][204493 ][204494 ][204495 ][204496 ][204497 ][204498
][204499 ][204500 ][204501 ][204502 ][204503 ][204504 ][204505 ][204506 ][204507 ]
[204508 ][204509 ][204510 ][204511 ][204512 ][204513 ][204514 ][204515 ][204516 ][
204517 ][204518 ][204519 ][204520 ][204521 ][204522 ][204523 ][204524 ][204525 ][2
04526 ][204527 ][204528 ][204529 ][204530 ][204531 ][204532 ][204533 ][204534 ][20
4535 ][204536 ][204537 ][204538 ][204539 ][204540 ][204541 ][204542 ][204543 ][204
544 ][204545 ][204546 ][204547 ][204548 ][204549 ][204550 ][204551 ][204552 ][2045
53 ][204554 ][204555 ][204556 ][204557 ][204558 ][204559 ][204560 ][204561 ][20456
2 ][204563 ][204564 ][204565 ][204566 ][204567 ][204568 ][204569 ][204570 ][204571
 ][204572 ][204573 ]
During the pass time the process makes 206243 times process switches.
```

圖三，I/O-bound 之執行結果。

## Source Codes

### User Space

#### CPU-bound

```
#include<unistd.h>
#include<stdio.h>
#include<sys/time.h>

#define ON 1
#define OFF 0
#define WAIT_TIME 120 //sec
#define SEC_TO_USEC 1000000

void start_to_count_number_of_process_switches() {
    syscall(352);
}

unsigned int stop_to_count_number_of_process_switches() {
    unsigned int ret;
```

```
    ret = syscall(353);

    return ret;
}

int main() {
    unsigned int a;
    int loop_switch = ON;
    int b = 0;
    struct timeval start_tv;
    struct timeval now_tv;

    start_to_count_number_of_process_switches();
    gettimeofday(&start_tv, NULL);

    while(loop_switch) {
        b += 1;

        gettimeofday(&now_tv, NULL);

        int diff;
        diff = (now_tv.tv_sec - start_tv.tv_sec) * SEC_TO_USEC + (now_tv.tv_usec -
start_tv.tv_usec);

        if(diff >= WAIT_TIME * SEC_TO_USEC) loop_switch = OFF;
    }

    a = stop_to_count_number_of_process_switches();
    printf("\nDuring the pass time the process makes %u times process
switches.\n", a);

    return 0;
}
```

**I/O-bound**

```
#include<unistd.h>
#include<stdio.h>
#include<sys/time.h>


#define ON 1
#define OFF 0
#define WAIT_TIME 120 //sec
#define SEC_TO_USEC 1000000


void start_to_count_number_of_process_switches() {
    syscall(352);
}
```

```
unsigned int stop_to_count_number_of_process_switches() {
    unsigned int ret;
    ret = syscall(353);
    return ret;
}

int main() {
    unsigned int a;
    int loop_switch = ON;
    int b = 0;
    struct timeval start_tv;
    struct timeval now_tv;

    start_to_count_number_of_process_switches();
    gettimeofday(&start_tv, NULL);

    while(loop_switch) {
        usleep(10);
        printf("[%d ]", b++);

        gettimeofday(&now_tv, NULL);

        int diff;
        diff = (now_tv.tv_sec - start_tv.tv_sec) * SEC_TO_USEC + (now_tv.tv_usec -
start_tv.tv_usec);

        if(diff >= WAIT_TIME * SEC_TO_USEC) loop_switch = OFF;
    }

    a = stop_to_count_number_of_process_switches();
    printf("\nDuring the pass time the process makes %u times process
switches.\n", a);

    return 0;
}
```

## Kernel Space

#### System Calls

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

asmlinkage void sys_start_count_number(void) {
    unsigned int bf = current->process_switch_counter;
    current->process_switch_counter = 0;

    printk("[start count] %u to 0\n", bf);
}
```

```
asmlinkage unsigned int sys_end_count_number(void) {
    unsigned int ret = current->process_switch_counter;
    printk("[end count] %u\n", ret);

    return ret;
}
```

**Task Struct**

```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;
.
.
.
.
.
.
#if defined(CONFIG_BCACHE) || defined(CONFIG_BCACHE_MODULE)
    unsigned int    sequential_io;
    unsigned int    sequential_io_avg;
#endif
    unsigned int    process_switch_counter; // 我們增加的參數
};
```

## Kernel 與 OS 版本

- Kernel: 3.10.108
- OS: Ubuntu 16.04.7 LTS

## taskset 原始碼追蹤筆記

`taskset` 是 `util-linux` 中實作的一個功能，用於對程序與 CPU 進行一些設定與操作，其中便有本專案第一部分所需的功能：指定執行程序的 CPU。

`taskset` 的主程式實作在 `/util-linux/schedutils/taskset.c` 內，其中最重要的是 `do_taskset` 函式中呼叫的 `sched_setaffinity()` 函式。

由於我們選擇的 Linux Kernel 版本為 3.10.108，因此以下皆以此版本進行說明 ( 不同版本實作有一點差別 ) 。

`sched_setaffinity()` 是 Linux 提供用於調度 CPU 的函式，位於 `/kernel/sched/core.c`，功能為將 CPU 和指定的程序綁定，使程序執行於特定 CPU 上。 其中最重要的是 `sched_setaffinity()` 會呼叫 `set_cpus_allowed_ptr()` 函式。

`set_cpus_allowed_ptr()` 中，主要會呼叫 `stop_one_cpu()` 來達成指定執行程序 CPU 的目標。 而其中 `stop_one_cpu()` 的引數包含了 `migration_cpu_stop()` 函式，其作用為設定程序的新 CPU，並將程序從源 CPU 搬移至新 CPU。

```
4874            dest_cpu = cpumask_any_and(cpu_active_mask, new_mask);
4875            if (p->on_rq) {
4876                    struct migration_arg arg = { p, dest_cpu };
4877                    /* Need help from migration thread: drop lock and wait. */
4878                    task_rq_unlock(rq, p, &flags);
4879                    stop_one_cpu(cpu_of(rq), migration_cpu_stop, &arg);
4880                    tlb_migrate_finish(p->mm);
4881                    return 0;
4882            }
```

`migration_cpu_stop()` 的主要功能在 `__migrate_task()` 內，這邊首先會取得源 CPU 和目標 CPU 的 rq, 接著透過 `dequeue_task()` 將程序從源 CPU 的工作序列中移除，使用 `set_task_cpu()` 設定更新好程序的新 CPU (目標 CPU) 資訊後，再以 `enqueue_task()` 將程序添加到目標 CPU 的工作序列中。

```
4925            if (p->on_rq) {
4926                    dequeue_task(rq_src, p, 0);
4927                    set_task_cpu(p, dest_cpu);
4928                    enqueue_task(rq_dest, p, 0);
4929                    check_preempt_curr(rq_dest, p, 0);
4930            }
```

> References:
>
> - sched_setaffinity如何在Linux內核中工作_cuma2369的博客-CSDN博客
> - Linux 进程管理之任务绑定-51CTO.COM
> - Linux source code (v3.10.108) - Bootlin

## gettimeofday 原始碼追蹤筆記

本專案所用之 `gettimeofday()` 為 C 語言提供之函數，使用前須在程式中以 `#include<sys/time.h>` 載入。雖然使用時並非直接呼叫 system call，但其在 Linux 下還是以 `sys_gettimeofday()` system call 實踐的。

在 user space 程式中使用 `gettimeofday()` 後，最後會執行到 `/kernel/time.c` 中 `gettimeofday()` 的 system call。而在這部分中，最重要的是呼叫了`do_gettimeofday()` 函式。

`do_gettimeofday()` 位於 `/kernel/time/timekeeping.c` 中，其主要利用 `getnstimeofday()` 取得儲存當前時間的結構 (timespec)，而 `getnstimeofday()` 則是在其中呼叫 `__getnstimeofday()` 進一步得到結果。

`getnstimeofday()` 與 `__getnstimeofday()` 都位於`/kernel/time/timekeeping.c`，在此二函式中，我們會取得儲存了時間資訊，結構名稱為 `timespec` 的資料 `ts` (在 `do_gettimeofday()` 稱為 `now`)。於 `do_gettimeofday()` 中，再將 `now` 中的時間資訊取出，放進結構名稱為 `timeval` 的資料 `tv`，其中包含了 `sec` 和 `usec` 兩種資料。

此外，`gettimeofday()` 的 system call 還會取得時區 `tz` 資料，它會直接將儲存了時區資訊，結構名稱為 `timezone` 的資料 `sys_tz` 透過 `copy_to_user()` 將資料複製到 user space 的程式 (取得時間資訊也有使用到 `copy_to_user()`，只是上面略過，於此處一起說明)。

> References:
>
> - 谈谈时间函数gettimeofday_donnyxia1128的博客-CSDN博客
> - Linux source code (v3.10.108) - Bootlin