

Linux 核心與驅動程式 第二次程式作業

本次作業，承續上次的 mini-example，在這個小平台上，將核心的基礎部份打造出來。這會是一個有排程，有搶先能力的核心，並提供簡單的行程同步機制。在介面與架構上，盡量配合 Linux kernel，讓大家在實作時，同時達到學習作業系統概念與瞭解 Linux kernel 的實作方式。

請先下載上次作業的答案與作業二的開發環境更新

<http://www.cs.ccu.edu.tw/~lhr89/linux-kernel/embed-example-for-2.6.tar.gz>

<http://www.cs.ccu.edu.tw/~lhr89/linux-kernel/mk.tar.gz>

解開後 embed-example 後，在其目錄下，將 mk 解開，並輸入

```
patch -p1 < mk/patch-mini-kernel
```

mk 目錄下的 preboot 及 Image，是已完成的可執行檔，給大家參考。小提醒，開發過程中，可直接執行根目錄的 vmkernel 來驗證你的實作，不需要透過 preboot。GDB 也可以直接對 vmkernel 作 debugging。

作業程式中，已提供以下資料結構與輸出函式：

```
int printk(const char *fmt, ...);
```

```
int snprintf(char * buf, size_t size, const char *fmt, ...);
```

```
struct list_head;                    操作此list的函式也有，如: list_add()
```

```
struct itimerval;                    與 setitimer()
```

所有 signal(訊號)處理的資料結構與函式如: **sigset_t**, **struct sigaction**, **sigaction()**, **sigprocmask()**等，請自行參與作業程式中的 **signal.h**，用法可參考 Linux man pages 或 Advanced Programming in the Unix Environment 一書。

以下簡要說明，已在此核心的資料結構與全域變數：

```
struct task_struct; {                    代表task的資料結構
    volatile long state;                task目前的狀態：執行中/結束中/暫停等
    unsigned int time_slice;            每次timer來都減1，當值為0，強制排程
    int preempt_count;                此值大於0時，不可切換掉此執行中task
    struct list_head tasks;            所有的task都串在這上
    struct thread_struct thread;        task CPU狀態儲存，只存esp與eip
};
struct task_struct *current;            指向目前執行中的 task
volatile unsigned long jiffies;        目前的核心時間，每次 timer 來加 1
```

```

unsigned int need_resched;          當此值為真, timer會作task排程
struct task_struct *init_task;     代表(一開始)start_kernel()的 task
union task_union {                 task與task使用的stack一起配置
    struct task_struct task;
    unsigned long stack[4096/sizeof(long)];
};
#define HZ 1                          timer 中斷的頻率, 每秒 1 次

```

核心的運作從 start_kernel()開始，它呼叫

- 1) sched_init()完成 scheduler 相關的初始化動作
- 2) time_init()完成 timer 的中斷頻率設定與中斷函式的掛載
- 3) app_start(), 你的應用程式進入點，呼叫 kernel_thread()產生 user thread。
- 4) cpu_idle(), 進入無限迴圈。只要有其他 task 可被排程，會主動讓出 CPU。init_task 為代表執行 cpu_idle 的 task_struct。

一、(15分)核心要產生一新的 thread，必須將 task_struct、stack 空間及其內容準備好。請研究以下幾個函式的設計：kernel_thread(), kernel_thread_helper(), do_exit()。說明各函式的**目的**、產生的 thread 的**第一個進入點**在那？**stack 內放了些什麼**，使用者提供的 fn()是怎樣被傳入及呼叫的？kernel_thread_helper() 加上 **asm linkage** 的目的是什麼？

二、task 切換(切換 CPU registers)的函式：context_switch()已經完成。請先瞭解 context_switch()做了那些事。

- 1) (5分)**已被排程過**的 task 與一個**剛出生**、尚未被排程的 task，在每次被 scheduler 選中切換時，**出發點**有何差異？
- 2) (25分)開始動手作吧！你要**完成 schedule()**這個函式：一個簡單的 round-robin scheduler，task 輪流被排程來使用 CPU 時間。schedule()每次被呼叫，便挑選下個 task 來出來，使用 context_switch()作切換。

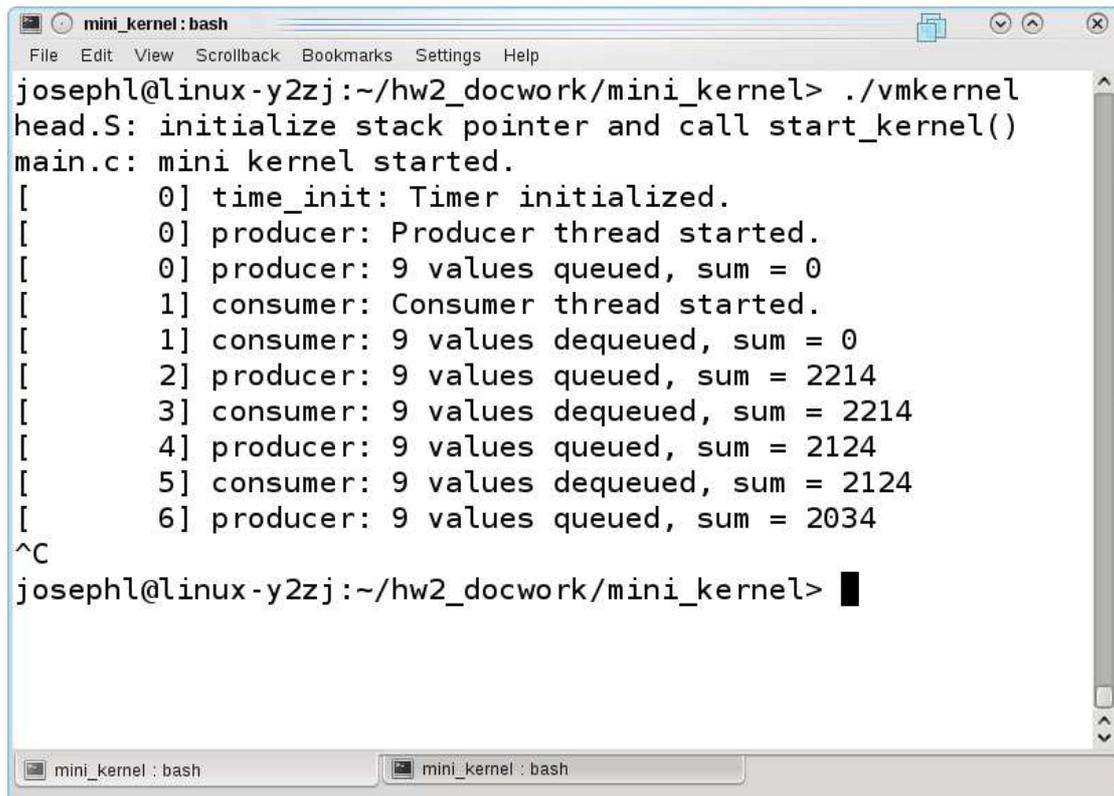
核心內已提供 2 個簡單的 task (在 apps.c 裡)，一個叫 producer，另一個叫 consumer。Producer 每次寫滿 queue，便會呼叫 schedule()，把 CPU 時間讓出來。Consumer 每次讀光 queue，也會呼叫 schedule()。這種由 task 主動呼叫 schedule()，去驅動排程器運作的方式，稱為 cooperative scheduling (合作式排程)。如果你完成第二題，執行核心驗證一下，是否 producer 及 consumer 輪流使用 CPU，如下圖。

```
mini_kernel: bash
File Edit View Scrollback Bookmarks Settings Help
josephl@linux-y2zj:~/hw2_docwork/mini_kernel> ./vmkernel >& output
^C
josephl@linux-y2zj:~/hw2_docwork/mini_kernel> head output
head.S: initialize stack pointer and call start_kernel()
main.c: mini kernel started.
[      0] producer: Producer thread started.
[      0] producer: 9 values queued, sum = 0
[      0] consumer: Consumer thread started.
[      0] consumer: 9 values dequeued, sum = 0
[      0] producer: 9 values queued, sum = 0
[      0] consumer: 9 values dequeued, sum = 0
[      0] producer: 9 values queued, sum = 0
[      0] consumer: 9 values dequeued, sum = 0
josephl@linux-y2zj:~/hw2_docwork/mini_kernel>
```

三、(30分)核心如有週期性的中斷訊號進來，便能主動驅動排程器運作，而不需要 task 自己去呼叫 `schedule()`。這樣的排程方式，稱為 preemptive scheduling (搶先式排程)。Linux 核心提供每個 user process 三個週期性的 timer，每次 timer 設時間到了，核心就會送一個 UNIX signal 給 process，中斷當時的工作，改執行 signal handler。Signal handler return 後會回復 user process 原來的工作。應用 UNIX signal，我們可以實作出類似硬體的時間中斷，驅動排程器的運作。請完成下面二個函式，實作出搶先式排程。

- 1) `time_init()`：請使用 `ITIMER_REAL(SIGALRM)` 這個計時器。呼叫 `sigaction()` 安裝 signal handler。注意加上 `SA_NODEFER` 這個 flag，否則未以 `sigreturn()` 結束的 signal handler，該 signal 會被永久的 mask 掉。使用 `setitimer()` 設定計時時間間隔，你得將 HZ 轉換為 `setitimer()` 接受的時間單位。
- 2) `timer_interrupt()`：`sigaction()` 應安裝此函式來處理中斷。此函式必須更新系統時間 jiffies 與 current process 的 `time_slice`，當 process 的 `timeslice` 用完時，回復它的值，並呼叫 `schedule()` 做行程切換。

當你完成此題，便可以把 `apps.c` 裡，producer 與 consumer 呼叫 `schedule()` 的程式碼移除。觀察 producer 及 consumer 是否一樣會輪流執行。



```
mini_kernel: bash
File Edit View Scrollback Bookmarks Settings Help
josephl@linux-y2zj:~/hw2_docwork/mini_kernel> ./vmkernel
head.S: initialize stack pointer and call start_kernel()
main.c: mini kernel started.
[ 0] time_init: Timer initialized.
[ 0] producer: Producer thread started.
[ 0] producer: 9 values queued, sum = 0
[ 1] consumer: Consumer thread started.
[ 1] consumer: 9 values dequeued, sum = 0
[ 2] producer: 9 values queued, sum = 2214
[ 3] consumer: 9 values dequeued, sum = 2214
[ 4] producer: 9 values queued, sum = 2124
[ 5] consumer: 9 values dequeued, sum = 2124
[ 6] producer: 9 values queued, sum = 2034
^C
josephl@linux-y2zj:~/hw2_docwork/mini_kernel>
```

當搶先式多工開啟後，核心以及 apps.c 裡的 producer 與 consumer 都有不少 race condition 的問題。必須適時關閉中斷發生或關閉呼叫 schedule() 的能力，來避免 race condition。舉個例子：某 task A 主動呼叫 schedule() 放棄執行權，schedule() 選擇 task B 為 next 來執行，當程式執行到 context_switch(A, B) 的 current=next 此行後，時間中斷進來了，時間中斷函式也呼叫 schedule()。此時的 current 指向 B，而 CPU register 屬於 A；第二次的 schedule() 會因著 current 指向 B，誤以為目前的 CPU register 是 B 的，於是把屬於 A 的 CPU register 儲存在 B 的 task_struct 裡。因此造成 task B 的執行狀態被覆寫。所以，在 schedule() 中得避免 schedule() 再次被呼叫，才不會出現這種情況。定義在 task_struct 裡的 preempt_count 是 schedule() 的開關，每次呼叫 schedule() 前，應檢查此值，如果大於 0，不可呼叫 schedule()。

另外，如果 task 與 signal(interrupt) handler 有共有資料而產生 race condition 的情況，必須暫時關閉 signal 的發生。

- 四、1) (5 分) 請觀察 apps.c 裡的 producer 及 consumer，如我上面舉的例子一樣，請你想一個情況，是會讓 producer 或 consumer 發生錯誤的。詳述每個步驟，直到以致錯誤發生。(限: producer 誤寫了尚未被 consumer 讀取的資料或 consumer 誤讀了尚未被 producer 寫入的資料)
- 2) (20 分) sched.h 裡有二類同步函式，一類以 preempt 開頭，控制是否

schedule()可被呼叫。另一類以 local_irq 開頭，控制 signal(interrupt) 是否可以進來。請將這些函式內容補齊，並修改 apps.c 裡的 producer 及 consumer，使用以上的同步函式，使 race condition 不再發生。

五、(10 分加分題)請閱讀 Linux 核心 Documentation 目錄下的 volatile-considered-harmful.txt 文件，瞭解什麼時候該用，什麼時候不該用 volatile 這個 C keyword。簡述一下你的心得。