第 9 組

黃允誠 107502520
王伯綸 110521095
黃柏儒 110522136

※這次比上次加了更多的超連結，如果助教們有想要對照 Linux kernel code 的內容確認可以留意一下。
※淺藍色的都是。

基本觀念

Context Switch

schedule() → __schedule() → context_switch() → switch_to

switch_to(prev, next, last)，表示做 context switch 的切換順序，prev 為目前要被中斷的 process，next 為下一個要執行 process，而 last 為做 context switch 切回 prev 繼續執行的前一個 process。

但是，以 5.11 版 x86 為例，現在改成這樣：

```
#define switch_to(prev, next, last)                     \
do {                                                    \
    ((last) = __switch_to_asm((prev), (next)));         \
} while (0)
```

看起來跟教授上課講的概念還是一樣，但我們甚至根本看不到__switch_to_asm()……應該是像教授上課提過的那樣動態產生的檔案，總之這邊已經超出我們能力範圍了……

nvcsw 和 nivcsw

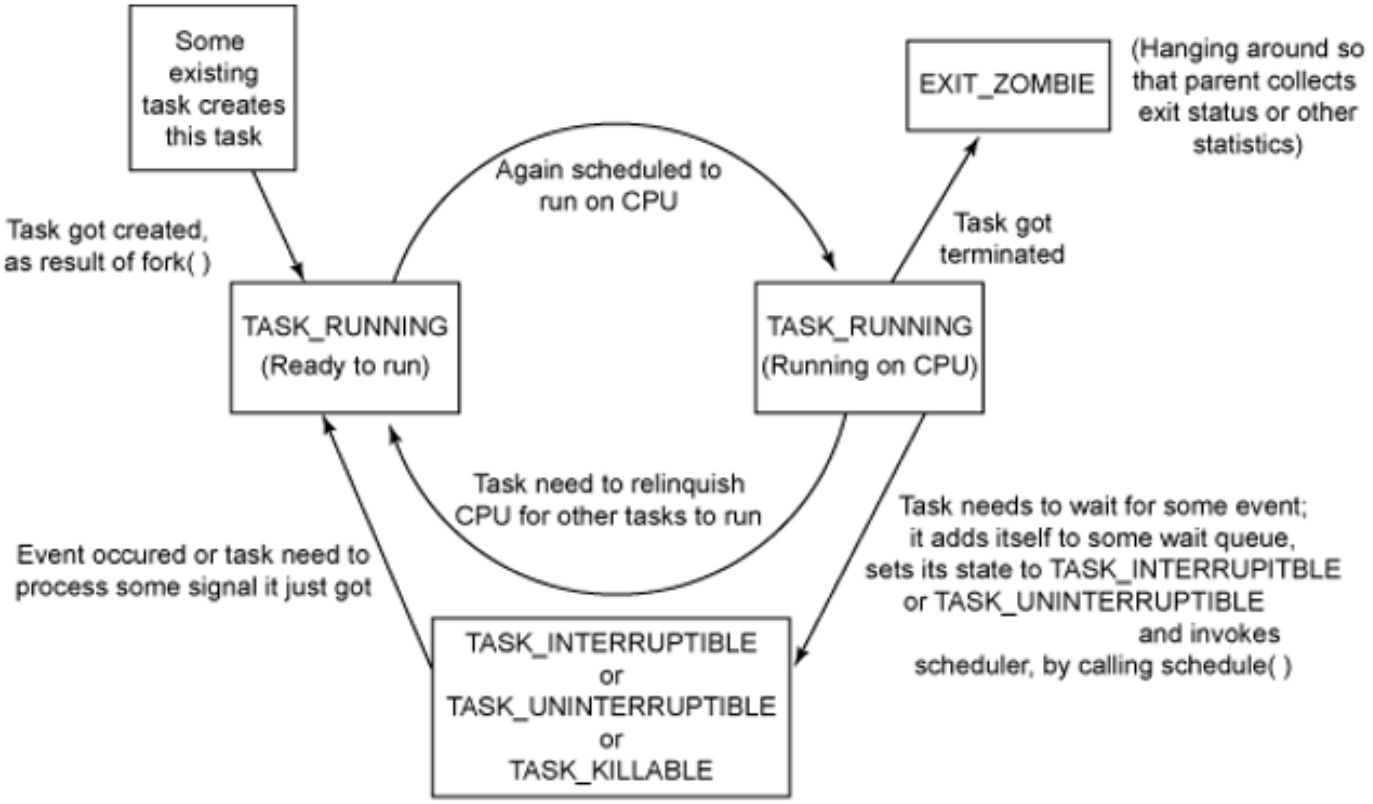
在 task_struct 中，nvcsw 是 Number of Voluntary Context SWitches，nivcsw 是 Number of InVoluntary Context SWitches，兩個都是 unsigned long 計數器，相加就是總共的 context switch 次數。

那麼 Linux kernel 是如何操作這兩個計數器的呢？很簡單，首先回到 Project2 的 copy_mm()，在當中會進行初始化。接著在__schedule()內有個指標 unsigned long *switch_count 用來指向 nvcsw 或 nivcsw 二選一，最後會由++*switch_count;計數。

而二選一是怎麼個選法？首先會先指向非自願的一邊，然後 if (!preempt && prev_state)就改指向自願的一邊。preempt 直翻是「占用」，是__schedule()的唯一輸入參數布林值，我們發現 schedule()呼叫__schedule()的時候都是傳入 False，其他很多會呼叫__schedule()的 function 有些是 True 有些是 False，反正就是代表是否是占用資源太久的情況。至於 prev_state 就是要交出控制權的 process 的狀態(/* -1 unrunnable, 0 runnable, >0 stopped: */)，也是從 prev task_struct 裡面直接拿，我們搭配 Linux tasks 的各種狀態看，就發現當成布林值的話，TASK_RUNNING 是 0 也就是 False，其他大部分都是大於 0 也就是 True。這邊是 preempt 要 False 且 prev_state 要 True 才改指到自願(nvcsw)，而 state 轉成布林要是 True 就代表「不是 TASK_RUNNING」，所以這 if 條件邏輯就是「你不是占用且也不是跑到一半才是自願 switch」。例如說等待外部硬體輸入 call schedule()，並不是占用(schedule 呼叫__schedule 會傳入 False)，而且狀態會變成 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 不是 TASK_RUNNING (> 0, True)。

當 thread 進行特定 system call 時，就會發生 Voluntary Context Switches；而當 thread 執行時間過長(通常大約 10 毫秒以上)並且有其他 thread 在等待 CPU 時，就會發生 Involuntary Context Switches。另外 kernel 可以根據這兩個值來判斷 cpu 的執行效率。

Process 狀態流程



process state 的流程:

一開始先 fork 產生一個新的 child process 進入 TASK_RUNNING，當成為 priority 最高的 process 就拿到 cpu 但還是在 TASK_RUNNING，接下來有 3 種 case:

case1: 執行完畢，直接終止。

case2: 當 process 持有 cpu 太久且沒有在等待任何 event，timer_interrupt 會丟它回去 run queue 裡面且優先權變低，cpu 給其他人用，但 state 仍舊是 TASK_RUNNING。

csae3: 在執行的時候需要執行 diskIO、開 device 或者 timer 時間到，state 會變成 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 或 TASK_KILLABLE，進入 3 種狀態都會把 process 丟進去 wait queue，等到目標事件出現(例如需要的資料讀進來後)，硬體再產生 interrupt 去叫等待的 process 起來，放到 run queue，state 變回 TASK_RUNNING。

當 decvice driver 在 probe 的時候，送 signal 給他會讓他以為 probe 結束，可能導致系統 crash，因此需要 TASK_UNINTERRUPTIBLE 狀態來預防這種情況：當 process 進入 TASK_UNINTERRUPTIBLE 這個 state，即使傳給它 signal 也無法 wakeup process，直到它做完再告訴系統執行完畢。然而把 process 設為此狀態，可能導致 wakeup 因為某些原因一直無法發生(例如 IO 其實壞掉了)，然後此 process 永遠無法被終止又死不做事，也就是所謂的 hung task，如果 kernel 無法解決只有重新開機。

因此加入 TASK_KILLABLE，在此狀態下 process 收到 fatal signal 的話可以提前 wake up 它，相當於 TASK_WAKEKILL 加上 TASK_UNINTERRUPTIBLE。fatal signal 就是會讓 process 被 kernel 殺掉、或是如果 process 不主動處理就會被 kernel 殺掉的 signal。

set_current_state 有 memory barrier，用於對正在執行的 process 更改 state，因為 task_struct.state 是 volatile，就是隨時都可能高頻率變動的意思(主要是執行中的時候)，所以需要 memory barrier 才安全。對於在 wait queue 中的 process 用沒有 barrier 的__set_current_state 比較快速。

Linux tasks 的各種狀態

Wait Queue: Process wait and wake up

___wait_event → init_wait_entry → autoremove_wake_function → default_wake_function → try_to_wake_up → ttwu_queue → ttwu_do_activate → ttwu_do_wakeup

從上面的過程 trace 到的 code 可以看出，遇到需要 wait 的情況觸發事件之後，會先由 init_wait_entry 產生一個 wait queue 中的元素(wq_entry)，並把要 wait 的 task_struct(用 current 抓)放進去，同時喚醒的方式(func)指定為 autoremove_wake_function。之後要 wake up 的時候執行這個 function 會呼叫 Project3 提示給的 defult_wake_function，而根據 wake 成功與否，如果成功 wake 就會執行 list_del_init 把對應的 wq_entry 又從 wait queue 中刪除。

所以雖然粗略理解是「Process 進入 wait queue 等待，等待的條件滿足後從 wait queue 中移出繼續執行」，但實際上 kernel code 的具體實作中，Process 等待或執行中只是記錄狀態在 task_struct 上，而真正進出 wait queue 的是包著 Process (task_struct)的 wq_entry。

實作細節

問題答案

Q1 只要將 nvcsw 跟 nivcsw 相加就是答案，Q2 則沒有內建可以用，所以要自己加一個計數器，然後處理「初始化」跟「計數」兩個問題，最後就跟 Q1 一樣吐出去就好了。

system call 的改編

首先為了節省編譯時間，我們選擇將兩個問題整合成同一個 system call，畢竟只要開兩個輸入參數，然後把兩題的答案分別 copy_to_user 就可以了。不過既然都已經有 nvcsw 跟 nivcsw，那為甚麼還要相加？區分開來不是更好嗎！

所以我們的 system call 由 user 端傳入四個指標，分別取得 Q1 答案、nvcsw、nivcsw、Q2 答案。結果最後還是手癢，試試看用 Q2 的方式做 Q1，所以「Q1 答案」是用我們額外加入的計數器。

加入計數器

在 task_struct 中加入自己的計數器，除了之前說的要加最後之外，kernel 原始程式中這邊很貼心的有一些註解提醒我們也不能放太後，最後選好位子放上。

計數器初始化

有提示在 copy_thread()中，但那已經是 machine dependent，甚至有的組員還找不到對應的程式碼，最後決定往上一層，在 copy_process()裡面加在 copy_thread()底下。

計數器更新計數

Q1 的部分沒什麼看頭，就直接黏在原本 kernel 自己在計數的位子旁邊就沒問題了。

default_wake_function → try_to_wake_up → ttwu_queue → ttwu_do_activate → ttwu_do_wakeup

Q2 就比較有趣，從上面的 trace 中，我們本來認為就在 ttwu_do_wakeup 裡面 state 設定成 TASK_RUNNING 的地方把計數器+1 就好。但是後來仔細看 try_to_wake_up 才發現，本來以為它一定會呼叫 ttwu_queue，但實際上裡面用了一些 goto，例如如果是要 wake up current，它就不會走這條路，又或者有些特定 config 之下也不一定會走這邊。除此之外，會有個叫"success"的 flag 紀錄是否成功 wake，也就是說即使進入這個 function 也不一定就 wake up，就像上面 wait queue 那邊提到的不一定會移出 wait queue (雖然看起來會執行到 ttwu_queue 的話好像都會先把 success 設成 1)。

於是想說應該把+1 的地方放在 try_to_wake_up 最後 return 之前有個 if (success)裡面。雖然後來我們又加另一個計數器，用本來的想法(在 ttwu_do_wakeup +1)，然後實驗看起來兩個結果都一樣，但還是覺得後面這種做法比較合理。

Code

system call

```
#include …
SYSCALL_DEFINE4(project3, unsigned long* __user, csw, unsigned long* __user, vcsw, unsigned long* __user, ivcsw, unsigned long* __user, wait){
	unsigned long csw_k;
	csw_k = current->project3_switch_count;
	copy_to_user(csw, &csw_k, sizeof(unsigned long));
	unsigned long vcsw_k;
	vcsw_k = current->nvcsw;
	copy_to_user(vcsw, &vcsw_k, sizeof(unsigned long));
	unsigned long ivcsw_k;
	ivcsw_k = current->nivcsw;
	copy_to_user(ivcsw, &ivcsw_k, sizeof(unsigned long));
	unsigned long wait_k;
	wait_k = current->project3_wait_count;
	copy_to_user(wait, &wait_k, sizeof(unsigned long));
	return 0;
}
```

task_struct: /include/linux/sched.h

```
…
	unsigned long project3_switch_count;
	unsigned long project3_wait_count;
	/*
	 * New fields for task_struct should be added above here, so that
	 * they are included in the randomized portion of task_struct.
	 */
	randomized_struct_fields_end
	/* CPU-specific state of this task: */
	struct thread_struct		thread;
	/*
	 * WARNING: on x86, 'thread_struct' contains a variable-sized
	 * structure.	It *MUST* be at the end of 'task_struct'.
	 *
	 * Do not put anything below here!
	 */
…
```

copy_process(): /kernel/fork.c

```
…
	retval = copy_thread(clone_flags, args->stack, args->stack_size, p, args->tls);
	p->project3_switch_count=0;
	p->project3_wait_count=0;
	if (retval)
		goto bad_fork_cleanup_io;
…
```

try_to_wake_up(): /kernel/sched/core.c

```
…
	ttwu_queue(p, cpu, wake_flags);
unlock:
	raw_spin_unlock_irqrestore(&p->pi_lock, flags);
out:
	if (success) {
		ttwu_stat(p, task_cpu(p), wake_flags);
		p->project3_wait_count++;
	}
	preempt_enable();
	return success;
}
```

userspace test

```
#include …
#define NUMBER_OF_IO_ITERATIONS	6
#define NUMBER_OF_ITERATIONS	99999999
#define input	for(i=0; i<NUMBER_OF_IO_ITERATIONS; i++){	\
			v=1;	\
			c = getchar();	\
		}
#define output	for(i=0; i<NUMBER_OF_IO_ITERATIONS; i++){	\
			v=1;	\
			printf("I love my home.\n");	\
		}
#define loop	for(i=0; i<NUMBER_OF_ITERATIONS; i++)	\
			v=(++t)*(u++);
#define testS	"This process encounters %lu times context switches, \
%lu voluntary, %lu involuntary.\nThis process enters a wait queue %lu times.\n"
#define test	if(syscall(444, &csw, &vcsw, &ivcsw, &wait)!=0)	\
			printf("Error !\n");	\
		else	\
			printf(testS, csw, vcsw, ivcsw, wait);
int main (){
	char c;
	int i,t=2,u=3,v;
	unsigned long csw,vcsw,ivcsw,wait;
	input test loop test input test output test loop test
}
```

執行結果

測試程式 IO

12345
This process encounters 1 times context switches, 1 voluntary, 0 involuntary.
This process enters a wait queue 1 times.
This process encounters 9 times context switches, 1 voluntary, 8 involuntary.
This process enters a wait queue 1 times.
1
3
5
This process encounters 12 times context switches, 4 voluntary, 8 involuntary.
This process enters a wait queue 4 times.
I love my home.
I love my home.
I love my home.
I love my home.
I love my home.
I love my home.
This process encounters 12 times context switches, 4 voluntary, 8 involuntary.
This process enters a wait queue 4 times.
This process encounters 22 times context switches, 4 voluntary, 18 involuntary.
This process enters a wait queue 4 times.

執行結果解讀

1. 測試動作是輸入→迴圈→輸入→輸出→迴圈，每兩個動作之間都 system call 輸出情況。
2. 每次 input 換行因為要等，所以會進 wait queue；但是沒有換行的部分，因為是用 buffer 一次輸入，很快所以不進 wait queue，也沒有自願 csw。雖然輸入是 6 個字元但換行的\n 也算。
3. 本來以為 IO 都要進 wait queue，但 output 一樣因為很快不進 wait queue。
4. loop 過程中因為瘋狂 run 很久，所以會 timer_interrupt 出現很多強制 csw。
5. 無論何時都有可能 timer_interrupt 強制 csw，2 跟 3 不進 wait queue 只是不會有自願 csw、比較少且通常沒有強制 csw，不代表不可能有。多次實驗就有遇過幾次期間出現強制 csw。
6. 改成分開測試之後，第一次輸出情況(test)之前只有輸入，多半都還 0 次強制 csw，但同 5 非絕對。

額外發現

schedule()不負全責

先從比較小的事情開始，Project3 提示說「Inside Linux kernel, kernel function schedule() is in charge of the context switch operation.」，但在上面解釋 nvcsw 跟 nivcsw 那邊在 trace code 時就知道，schedule 只是__schedule 的其中一個 caller，只是呼叫__schedule 的眾多來源的其中一種，而且這眾多來源對__schedule 的呼叫也分成兩類(preempt true or false)。所以這個描述不是很精確，真正負責 context switch 的人應該至少從__schedule 這一層開始算。

「在 fork 之後」到底是在甚麼之後

雖然在執行結果的解釋上我們是參考了同學 github 上的答案(並作為依據改測試流程強化驗證，有點倒過來……)，但是對於 task_struct 欄位的初始化，本來並沒有遇到甚麼問題。組員是忘記初始化，我就很單純的因為 copy_thread 很難找所以往上放到 copy_process 裡 copy_thread return 回來的位置，想說試試看這樣應該是一樣的。但是後來看到那組同學忽然更新說他們初始化遇到問題，又說甚麼以為要在 fork 初始化，然後我就開始很好奇到底為甚麼在前面先初始化會不對。

明明東西是我們自己另外加的，kernel 原本的 code 不可能動到啊？是要到「多前面」會不對？中間發生了甚麼事？我在 copy_proccess 晃來晃去，但一直不得要領。後來我忽然想到思路不對：不是在前面初始化之後被動到，而是根本就沒有成功初始化到。這兩種情況都會導致這個結果，但很明顯前面那種不可能。轉變思路之後很快就找到了。

在 copy_proccess 裡面這個 dup_task_struct 其實我在 Project2 的時候就有注意到，但那個不是當時的重點，demo 時也沒機會跟助教講到，結果忘了。p = dup_task_struct 才是真正拿了要 fork 的 child process 的 task_struct，雖然在這之前就有這個「p」沒錯，但那時候還是 copy_proccess 中一個無意義的 local variable，只是先宣告出來的空的「容器」而已。

雖然後來那組同學有說他們是在更高層的地方，根本找不到 task_struct 來初始化，但反正結論就是在這一行 p =還有後面檢查確認 p 存在之後，理論上就可以放心的初始化了。

有趣的進階測試

根據執行結果解讀，我們還可以進行一些小測試：既然換行會把前面所有東西用 buffer 的方式一次輸入，不用每個字元 wait，我們也知道，輸入 buffer 這東西你可以先把你知道程式要的輸入全部一次打完按 Enter。那麼程式要兩次輸入，我們一次全給，是不是後面直接 buffer 不用 wait？是：

```
123456789ab
This process encounters 1 times context switches, 1 voluntary, 0 involuntary.
This process enters a wait queue 1 times.
This process encounters 12 times context switches, 1 voluntary, 11 involuntary.
This process enters a wait queue 1 times.
This process encounters 12 times context switches, 1 voluntary, 11 involuntary.
This process enters a wait queue 1 times.
I love my home.
I love my home.
I love my home.
I love my home.
I love my home.
I love my home.
This process encounters 12 times context switches, 1 voluntary, 11 involuntary.
This process enters a wait queue 1 times.
This process encounters 26 times context switches, 1 voluntary, 25 involuntary.
This process enters a wait queue 1 times.
```

還不只這樣，我們知道輸入 buffer 可以用指令改成以 txt 檔案當成輸入 buffer，直接程式一執行立刻輸入到完，例如「./project3test < in.txt」。那我是不是可以 0 次 wait 了？很可惜，好像不是。但是我不死心，一直用各種的 in.txt 測試，最後歸納一些表面現象的結論：

0. 文字檔長度不到的話，剩下所有要的輸入都給 EOF，直接讓程式把讀到的東西輸出會是一個怪怪的符號，用%d 輸出當數字會是-1。
1. 文字檔如果全空沒有任何字元，那麼直接執行可以 0 次 wait。
2. 文字檔如果有任何字元，那麼存檔後直接執行還是要 1 次 wait，但之後文字檔不編輯，重複執行會 0 次 wait。
3. 文字檔有任何字元，只要再存檔就算內容完全沒變也還是要執行一次 1 次 wait，之後再重複執行才會又 0 次 wait。

這些現象以我的程度很難找原因，但有些推論：

首先或許即使檔案輸入對程式來說還是太慢，所以還是要 wait 一下。而至於沒編輯重複執行會 0 次，可能是有類似 catch 這種概念的機制在。還有就算沒改內容進行存檔，下一次又會 1 次 wait，可能是 catch 的機制是用檔案的編輯時間來判斷要不要真的去讀。最後，關於文字檔全空 0 bytes 直接推一堆 EOF 進去為何可以第一次執行就 0 次 wait，這我猜可能真的就沒有實際進行輸入，但很不確定。