OS project 1 Report

B07902132 陳威翰

設計

system call: my_add.c

在編譯 kernel 時測試用的 system call（跟作業一的 system call 是一樣的）。system call 編號是 333。

system call: osproject1_gettime.c

傳入一個 timespec 的指標，藉由 getnstimeofday 這個函數，得到 1970/1/1 0:00:00 到現在的秒數，以及這個秒數的奈秒。system call 是 334。

system call: osproject1_showinfo.c

傳入五個參數：process id、相對應 process 開始的秒 & 奈秒、process 結束的秒 & 奈秒。並且把這些資訊輸出到 kernal info 上。 system call 是 335 。

scheduler.c / scheduler.h

這個是 scheduler 的本體，他會從 stdin 讀入測試資料、判斷現在是哪一種 schedule algorithm、送去相對應的函數執行、最後把每個 process 對應到的 pid 輸出到 stdout 上。

ds.c / ds.h

裡面存著這次 project 需要的兩個資料結構： queue 跟 treap 。

queue 就是一般的佇列，實做上我是使用 doubly linked list 來維護。我會在 queue 的頭尾加上 head, tail 這兩個 node ，來方便我實做 pop、push、front 這幾個功能。這個資料結構我使用在 FIFO, SJF, RR 這三個演算法中。

treap 是一種平衡二元樹，我這邊採用的是 merge-split treap with randomized priority 。利用 priority 是 random 的性質，因此樹的期望深度是 $O(\log N)$（N 是現在 treap 的大小）。主要的操作是 merge （ `merge` 函數）跟 split （ `split_by_key` 、 `split_by_sz` ）。merge 的功能是把兩棵 treap 花 $O(\log N)$ 的時間合併起來，而 split 則是依照 key ，或者是 size ，把樹堆分成兩個部份！

而藉由此資料結構，我可以 $O(\log N)$ 把一個 Process 的 id, execution time 插入到 treap 裡面（ `insert_new` 函數），$O(\log N)$ 把剩下 execution time 最小的 process 拿出來執行（ `do_begin` 函數），$O(\log N)$ 把最小 execution time 的 process 從 treap 中刪除（ `pop_begin` 函數）。treap 這個資料結構我使用在 SJF、PSJF 這兩個演算法中。

util.c / util.h

這裡面實做了跟系統相關的函數： `cmp_process` 、 `set_cpu` 、 `set_pri` 、 `real_start_process` 、 `start_process` ，以及定義了 `Process` 這個 struct 。

`Process` 這個 struct 裡面有存 Process 的名子、ready time、execution time、sort 前後的編號，以及有沒有被 fork 過（ `forked` 這個變數）。

`cmp_process` 這個 function 主要是排序 Process 用的。比較基準是先比較 ready time ，如果 ready time 一樣，再比較 index 。

`set_cpu` 則是使用了 `sched_setaffinity` 這個函數去指定 process 要跑在哪一個 CPU 上。我的設計是讓 scheduler 跑在 `cpu_id = 0` 上，其他 fork 出來的 process 都跑在 `cpu_id = 1` 上面。

`set_pri` 這個 function 的功用是設定 process 的 priority 。priority 有分成兩類：high priority（我使用 `sched_get_priority_max(SCHED_FIFO)` ）以及 low priority（我使用 `sched_get_priority_min(SCHED_FIFO)` ）。設定完 priority 後，我使用 `sched_setscheduler` 來設定 priority。

我在這次 project 有一個特殊的設定：**當 process 真正要被 scheduler 執行時，我才會 fork 這個 process，確保 stdout 的時間是第一次進入 CPU 的時間** 。因此，在設定 priority 之前，我會先看這個 process 之前有沒有被 fork 過 （ `forked` 這個變數），如果要設成 high priority 而還沒被 fork ，那麼 scheduler 才會去 fork。如果要想把這個特殊的設定拿掉，可以把 `util.c` 中的第 103 行拿掉。

`start_process`，`real_start_process` 這兩個函數主要功能就是 fork process 。 fork 完之後，會在 child process 執行相對應 process 中的 `exec_time` 次一單位的程式碼（ `{ volatile unsigned long i; for(i=0;i<1000000UL;i++); }` ）。並且在執行前、後，去呼叫我寫的 system call 334 得到時間，最後在藉由 system call 335 把相對應的訊息弄到 KERN_INFO 。parent process 則是輸出 stdout 要求的東西，並且把 child process 設定到 `cpu_id = 1` 上執行。

FIFO.c / FIFO.h

這份檔案是實做 FIFO 演算法。我實做的方式是開一個 queue 去模擬 ready queue 的狀況（ `struct Queue *waiting_queue = calloc(1, sizeof(struct Queue));` ）。每當有某一些 process 的 ready time 抵達時，就把那些 process 丟到 ready queue 裡面，而每次去執行 ready queue 最前面（front）的 Process（實做上是使用 `running_process` 這個變數儲存）。當 front 執行完後，就 pop 掉。

當最後，當每個 process 都被 forked、executed的時候，就結束這個 scheduler。

SJF.c / SJF.o

這份檔案是實做 SJF 演算法。實做方式是開一個 treap ，維護現在可以被 schedule 的 process 們，另外再開一個 queue 去維護已經經過到達 ready time，但是還不能被 schedule 的 process 們（因為不能夠 preemptive 導致）。

因此，每到一個時間 unit 時，我會先把 ready time 抵達的放到 queue 裡面。如果 treap 是空的，或者是在上一個時間點中，有 process 被執行完畢，則把 queue 裡面的東西通通丟到 treap 裡面。之

後當 treap 不為空時，每次去執行 `do_begin` 去執行 execution time 最少的 process。

最後，如果 treap、queue 都是空的，而且每個 process 都被 forked、executed 了，就結束 scheduler 。

RR.c / RR.o

這份檔案是實做 RR 演算法。具體實做方式類似 FIFO，每次把 ready time 到達的 process 放到 queue 裡面，而每次去執行 front 的 process 。而在實做上，我紀錄了 `run_id` 這個變數，代表當前正在執行的 process 編號

而與 FIFO 不同的是，我開了一個 `left` 變數，去紀錄說 `p[run_id]` 這個 process 現在還可以進行多少個 TIME_UNIT ，初始值為 TIME_QUANTUM（根據題目的定義，TIME_QUANTUM 設為 500）。如果 `left == 0` ，就看當前的 process 還有沒有剩餘的 execution time 要執行，如果有的話，就把他丟到 queue 的尾端。

而到最後，如果queue 是空的、沒有 process 正在執行，並且每個 process 都被 fork 過了，就結束 scheduler 。

PSJF.c / PSJF.o

這部份的實做跟 SJF 極為類似，唯一不同點是當有 process 的 ready time 抵達時，我會直接把這個 process 丟到 treap 裡面（因為現在可以 preemptive 了），然後每次都執行 `do_begin()` 來執行當下 `exec_time` 最小的 process 。

實做上，我還開了一個 `last_id` 變數，去紀錄說上一個 TIME_UNIT 執行的 process 編號，如果這個編號跟 `do_begin()` 回傳的不一樣，代表上一個 process 被 interrupt 了，需要把他的 priority 設成最低!

到最後，如果 treap 是空的，而且每個 process 都有被 forked 過，就代表所有 process 都被執行過了，結束 scheduler 。

核心版本

linux-4.14.25 on Ubuntu 16.04 Server

比較實際結果與理論結果，並解釋造成差異的原因

想要得到比較的結果，可以參考 `readme.md` （在 repo 的根目錄執行 sh demo/go.sh ），結果會出現在 `result.txt` 裡面。

我實做 optimal solution 的 code 是 `check.cpp` ，使用方式為 `./check [TIME_MEASUREMENT_DMESG] [INPUT] [OUTPUT] [DMESG]` 。

而經過實測後，誤差範圍為 $\pm 8\%$ ，我認為是一個很合理的範圍。以下列出幾點會造成誤差的原因:

1. 運算誤差：我在實做 optimal solution 時，是使用 C++ 的 `long double` 變數型態，在計算的

過程中可能會造成誤差。

2. TIME_MEASUREMENT 的誤差：由於在進行一個 UNIT 的時間測量時，我們是使用 TIME_MEASUREMENT.txt 這筆測試資料。這筆測試資料有一個特性是：每個 process 在進入 ready queue 的時候，就可以馬上被執行。因此 TIME_MEASUREMENT.txt 這份程式的時間瓶頸會是 scheduler 。而在大多數的測試資料，幾乎每個時間點都會有 process 被 run ，因此時間的瓶頸會是 process 的執行。而 scheduler 會需要處理比較多的事情（EX: system call、演算法、...等等），而 process 只是簡單的 for 迴圈。所以有可能造成一個 time unit 被高估，因此造成實際執行的時間比 optimal time 還要低。
3. 演算法造成的誤差：由於 scheduler 需要執行一些 data structure 的操作，而這部份會造成測量上的誤差。例如：雖然 treap 的操作都可以在 $O(\log N)$ 內完成，但是他的常數偏大，在測試資料 N 都是偏小的，所以這部份可能會比 $O(N)$ 用 for 迴圈找到 `exec_time` 最小的還要慢。
4. 系統本身的誤差：由於系統還需要 run 一些其他的 process ，所以有可能會有 context switch 摻雜在其中，造成測量方面的誤差