OPERATING SYSTEM
- PROJECT 1 REPORT

B07902143 陳正康

排程器設計

以下稱排程器 process 為 parent，要執行的任務 process 為 child

CPU 使用

parent 和 child 都用 setaffinity 綁到 CPU 0，比較好維護同步的問題

OS 排程策略

使用 sched_setscheduler() 調整 parent 和 child 的優先度，並將 policy 設為 SCHED_FIFO。根據 manpage，設為 FIFO 的 process 在 Linux 上的優先權高於其它 process，因此能避免其他 process 跑到 CPU 0 上執行。同時，FIFO 能避免跑到一半的 process 被 OS preempt 掉，讓所有 preemption 都由 parent 控制，穩定控制變因。

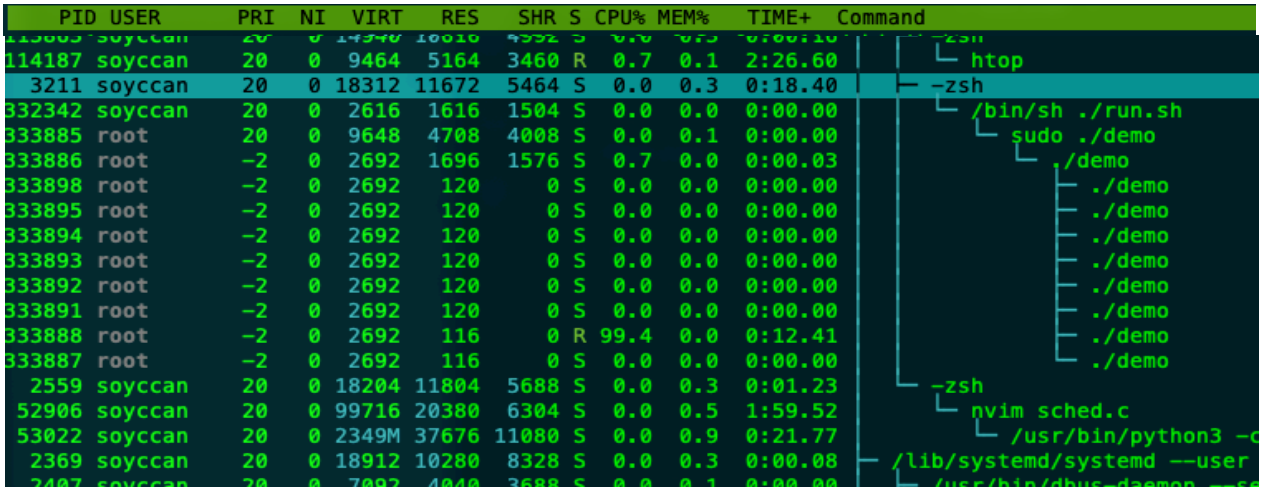
User Mode 排程方式

模擬 OS 的排程器運作，固定在每個 time unit 結束時把 CPU 交給 parent，由 parent 決定當下要跑哪一個 child (以 O(N) 時間掃過所有 child 的 array)，並送一個 SIGUSR1 給該 child，自己用 sigsuspend() 進入 blocked state。此時 OS 的排程器會尋找下一個目標進入 running state，因為只有一個 child 收到 signal，而且 FIFO 具有優先權，因此該 child 會被喚醒進入 running state。當 child 跑完一個 time unit 後，一樣送一個 SIGUSR1 給 parent，並讓自己等在 sigsuspend 上。如此每個 time unit 都要往返 parent 和 child 一次，並且 parent 或 child 在主動被 block 之前，不會被其它 process 打斷執行。

為了統一 parent 和 child 的執行時間，若 parent 沒有 child 可以執行時，會交給一個 dummy child 執行，而不是讓 parent 執行 time unit，以 time unit 是一致的。

觀察排程器運作

如圖的 htop 畫面，被設為 FIFO 的 process 的 pri 值是 -2，優先於其它為 20 的 process，故 CPU 0 會優先給 parent 和 child 執行，並且由於 parent 和所有 child 的 pri 值都相同，被 block 的 child 在有其它 child 執行時拿不到 CPU (TIME 欄位 為 0)



再如下圖的 perf 輸出，紅色是 parent (pid=454798)，pid 454800 - 454805 是 child，可以看到排程器正確的運作，在 parent 和 child 間來回切換 (context-switch)，中間沒有其他不相干的 process 搶占 CPU。每個 child 的 runtime 約為一個 time unit 的時間 (在測試機上約為 1.6 - 1.8 ms)，每次 parent 的執行時間約為 5 - 7 µs，效能仍遠輸給系統排程器的 1 µs。指令為：

$ perf sched record <command>

$ perf sched timehist -C 0

time	cpu	task name [tid/pid]	wait time (msec)	sch delay (msec)	run time (msec)
25478.304658	[0000]	demo[454798]	0.000	0.011	1.143
25478.304701	[0000]	demo[454800]	0.000	0.444	0.043
25478.306072	[0000]	demo[454801]	0.000	0.444	1.370
25478.306121	[0000]	demo[454802]	0.000	1.717	0.049
25478.306160	[0000]	demo[454803]	0.000	1.695	0.038
25478.306199	[0000]	demo[454804]	0.000	1.664	0.038
25478.306232	[0000]	demo[454805]	0.000	1.603	0.033
25478.306239	[0000]	demo[454798]	1.573	0.162	0.007
25478.307803	[0000]	demo[454801]	0.167	0.001	1.563
25478.307809	[0000]	demo[454798]	1.563	0.001	0.005
25478.309533	[0000]	demo[454801]	0.005	0.001	1.723
25478.309541	[0000]	demo[454798]	1.723	0.003	0.007
25478.310890	[0000]	demo[454801]	0.007	0.001	1.349
25478.310896	[0000]	demo[454798]	1.349	0.001	0.005
25478.312797	[0000]	demo[454801]	0.005	0.001	1.900
25478.312803	[0000]	demo[454798]	1.900	0.002	0.006
25478.314156	[0000]	demo[454801]	0.006	0.001	1.353
25478.314163	[0000]	demo[454798]	1.353	0.001	0.006
25478.315476	[0000]	demo[454801]	0.006	0.001	1.313
25478.315481	[0000]	demo[454798]	1.313	0.001	0.005
25478.317282	[0000]	demo[454801]	0.005	0.001	1.800

核心編譯

版本

Linux Kernel 5.6.5

增加 system call

- **439 void printk(const char *message, size_t num)**

  kernel 的 printk() 函式的 system call wrapper

  使用 copy_from_user() 將字串從 user space 複製到 kernel space

  再輸出到 kernel message 中，使 dmesg 中可見

- **440 void gettime(time64_t *tv_sec, long *tv_nsec)**

  kernel 的 ktime_get_real_ts64() 函式的 system call wrapper

  使用 put_user() 將數值從 kernel space 複製到 user space

- 兩個函式的 source “printk.c, gettime.c” 和 Makefile 都放在 beau/ 這個目錄下，讓 kernel 的 Makefile include 他們
- 比較重要的差別是，在 v5.0 之後好像要用 SYSCALL_DEFINEn() 這個 function wrapper 才能編譯成功，因此 HW1 裡的作法要修改一下。
- 參考：
  - 新增 system call：https://www.kernel.org/doc/html/v5.6/process/adding-syscalls.html
  - 取得 kernel 時間：https://www.kernel.org/doc/html/v5.6/core-api/timekeeping.html

編譯核心

只編譯當前開機有載入的模組，大輻減少編譯時間
(建議助教可以加在明年的 HW1 簡報裡，救人救世)

$ lsmod > /tmp/lsmod

$ make LSMOD=/tmp/lsmod localmodconfig

$ make install

$ make -C tools/perf DESTDIR=/usr/local

$ make -C tools/perf DESTDIR=/usr/local install

參考：https://www.kernel.org/doc/html/v5.6/admin-guide/README.html#configuring-the-kernel

數據分析

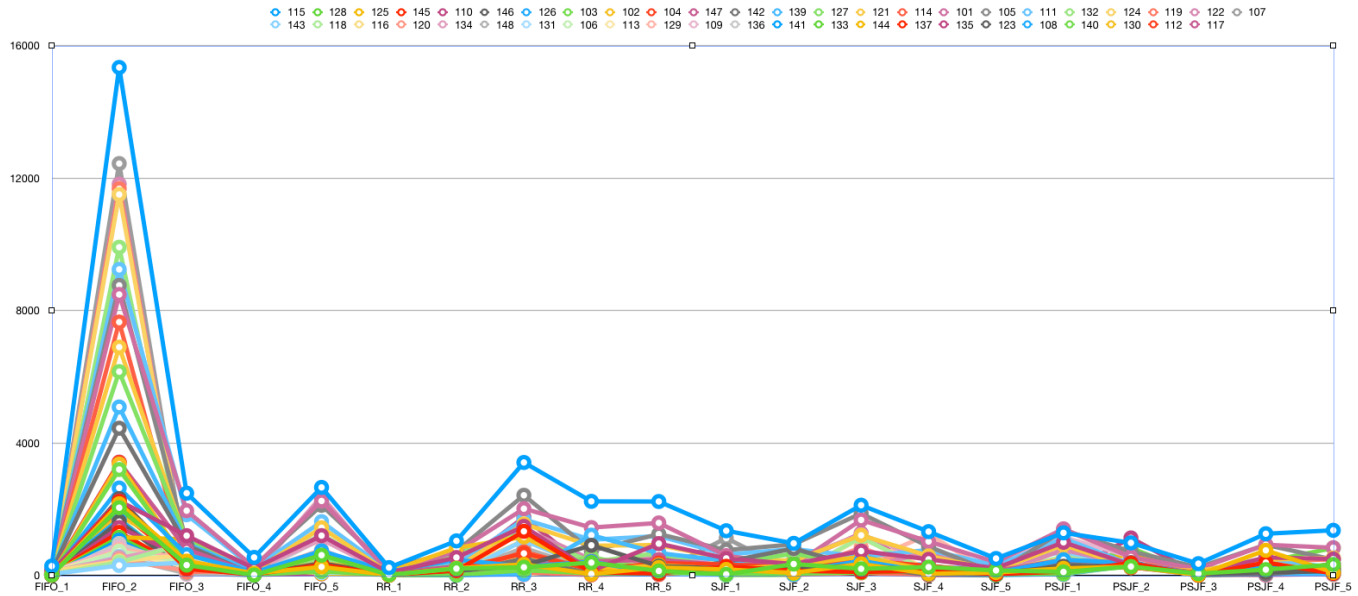
詳細的數據在 **err.csv** 檔案裡有 147 回合的實驗輸出，每回合都先用 TIME_MEASURE.txt 估計 time unit，再分別跑 20 個輸入。數據的浮動相當巨大，猜測是有變因沒有控制好。每個欄位裡的是每個開始與結束時間的實際與理論值的 SSE (sum of squared error) / 資料數再開根號。

以第 103 回的 FIFO_1 為例：

```
❯ cat test/103/err/FIFO_1.txt
----------------
103/FIFO_1_stdout.txt stdev=5.60357029044876
correct output:
  0.0 500.0
  500.0 1000.0
  1000.0 1500.0
  1500.0 2000.0
  2000.0 2500.0
program output:
  P1 0.0 503.0
  P2 503.0 1004.0
  P3 1004.0 1507.0
  P4 1507.0 2009.0
  P5 2009.0 2498.0
```

共有 10 個時間點，因此就是：sqrt{[(0-0)^2 + (500-503)^2 + (500-503)^2 + (1000-1004)^2) +...] / 10}

將誤差作圖 (原圖在 **err.png**) ，縱軸為前述誤差值，橫軸為測資，每回合都是一條折線，可以看到 FIFO_2 明顯大出許多，還有 FIFO_5, RR_3 等，都是總執行時間較久的。由此推論，誤差隨著時間累積。



誤差分析

memory stall

每個 time unit 結果後，parent 需要重新選擇下一個要跑的 child，而這個過程可能需要取得 child 的 ready time, execution time 等資訊，取得記憶體的這段時間 (數個 CPU 週期，又稱 memory stall) 會依 ready queue 的狀態而不同，有可能要找的下一個 child 在 CPU 的 cache 中 (cache reference)，也有可能不在 (cache miss)，有可能造成誤差，但考慮到測試平台的 CPU (i5-8500) 有 32KB per core 的 L1 Cache，足以放下測資不超過 10 個 process 的資料，故 memory stall 對 parent 來說影響不大。

對 child 來說，memory stall 的影響會明顯很多，因為迴圈變數宣告成 volatile，使得必須頻繁寫入記憶體，而 memory stall 的時間和系統負載有關，是非常難控制的變因，有機會造成巨大的誤差。

觀察 perf 的輸出，看到 parent (紅色) 的執行時間約為 5 - 7 μs，差距約在 2 μs，而 child 的執行時間為 1.3 - 1.9 ms，差距可以到 500 μs 以上。推測每次跑程式的執行時間不同主要是受 child 的執行時間不同影響。(指令: perf sched timehist -C 0)

time	cpu	task name [tid/pid]	wait time (msec)	sch delay (msec)	run time (msec)
25478.304658	[0000]	demo[454798]	0.000	0.011	1.143
25478.304701	[0000]	demo[454800]	0.000	0.444	0.043
25478.306072	[0000]	demo[454801]	0.000	0.444	1.370
25478.306121	[0000]	demo[454802]	0.000	1.717	0.049
25478.306160	[0000]	demo[454803]	0.000	1.695	0.038
25478.306199	[0000]	demo[454804]	0.000	1.664	0.038
25478.306232	[0000]	demo[454805]	0.000	1.603	0.033
25478.306239	[0000]	demo[454798]	1.573	0.162	0.007
25478.307803	[0000]	demo[454801]	0.167	0.001	1.563
25478.307809	[0000]	demo[454798]	1.563	0.001	0.005
25478.309533	[0000]	demo[454801]	0.005	0.001	1.723
25478.309541	[0000]	demo[454798]	1.723	0.003	0.007
25478.310890	[0000]	demo[454801]	0.007	0.001	1.349
25478.310896	[0000]	demo[454798]	1.349	0.001	0.005
25478.312797	[0000]	demo[454801]	0.005	0.001	1.900
25478.312803	[0000]	demo[454798]	1.900	0.002	0.006
25478.314156	[0000]	demo[454801]	0.006	0.001	1.353
25478.314163	[0000]	demo[454798]	1.353	0.001	0.006
25478.315476	[0000]	demo[454801]	0.006	0.001	1.313
25478.315481	[0000]	demo[454798]	1.313	0.001	0.005
25478.317282	[0000]	demo[454801]	0.005	0.001	1.800

虛擬化環境

由於測試環境是 VMware Workstation 虛擬機，CPU 和記憶體的存取都要隔一層介面，而且 host 可能還有其它 service 在跑，同時在用記憶體，尤其對 memory stall 的影響想必非常巨大。

Kernel Mode 耗時

由於取得時間和 signal 處理都要進到 kernel mode，因此在 kernel mode 停留的時間不一會造成誤差。但由 perf 的輸出可見，對 parent 來說每次執行時間浮動都在 2 μs 以內，其中包含 user mode 和 kernel mode，故可推測 kernel mode 時間浮動小於 2 μs。

```
❯ sudo time ./demo < OS_PJ1_Test/TIME_MEASUREMENT.txt
[sudo] password for soyccan:
P0 1314298
P1 1314303
P2 1314312
P3 1314313
P4 1314318
P5 1314327
P6 1314438
P7 1314443
P8 1314452
P9 1314453
16.09user 0.07system 0:16.17elapsed 99%CPU (0avgtext+0avgdata 1792maxresident)k
0inputs+0outputs (0major+466minor)pagefaults 0swaps
```

Time Unit 的估計值

每次執行時的 time unit 不盡相同，以第一次執行 TIME_MEASURE.txt 所得到的 time unit 在後面的測資執行時不一定適用。在測試過程中，我試著統計 time unit 估計的精確度，也就是使用 TIME_MEASURE.txt，把 10 個分別執行 500 time unit 的 child 的時間，加上中間閒置在 parent 的 9 個空檔 (也是 500 time unit)，共 19 筆時間差資料，取標準差 (timeunit.py)，結果約在 5ms - 80ms 左右。發現如果標準差愈小，那麼以這個 time unit 做接下來的測試結果就愈精確。因此，若在估算 time unit 時，將離群值去除，將有助於提高精確度。

給合前面的 memory stall，我試著去掉 time unit 迴圈變數的 volatile 並開 -O2 優化，讓 time unit 的迴圈變數存在暫存器中，以證明 memory stall 的時間較不固定。結果如下圖，左為原本用記憶體的，右為改用暫存器。將這 19 筆資料的標準差除以平均值，用記憶體的約為 0.027，用暫存器的約為 0.0044，相差數倍，可見當迴圈變數存放在記憶體時，會使每個 time unit 的時間長短較不固定。

```
0.8802165209999657
0.8148415830000886
0.8603003109965357
0.8319277470000088
0.8374312490050215
0.828668314999959
0.8478050189951318
0.82061318000342
0.8504339009959949
0.8205450470049982
0.878227278997656
0.8111420720015303
0.8219469139949069
0.8292043149995152
0.8729640400051721
0.8152185219951207
0.8734072360020946
0.8395218340010615
0.8184193130000494
avg = 0.8396228630525385
stdev =  0.022714849590725367
time unit = 0.001679245726105077
```

```
0.13375956199888606
0.13416906699421816
0.1339403210004093
0.13494963000994176
0.13616255499073304
0.13381963400752284
0.13362738100113347
0.13416324499121401
0.13362698700802866
0.1338283609948121
0.13365274900570512
0.1340218149998691
0.13364451099187136
0.1338426319998689
0.1336893410043558
0.1340560450043995
0.13363735999155324
0.13401347100443672
0.13359680099529214
avg = 0.1340106035786448
stdev =  0.0005937831434608909
time unit = 0.0002680212071572896
```

(下依序為記憶體版和暫存器版迴圈的組合語言、以及暫存器版的 time unit code)

```
4019fe:       48 8b 44 24 40          mov    rax,QWORD PTR [rsp+0x40]
401a03:       48 3d 3f 42 0f 00       cmp    rax,0xf423f
401a09:       77 13                   ja     401a1e <proc_launch+0x8e>
401a0b:       48 83 44 24 40 01       add    QWORD PTR [rsp+0x40],0x1
401a11:       48 8b 44 24 40          mov    rax,QWORD PTR [rsp+0x40]
401a16:       48 3d 40 42 0f 00       cmp    rax,0xf4240
401a1c:       72 ed                   jb     401a0b <proc_launch+0x7b>
```

```
4019c5:       b8 40 42 0f 00          mov    eax,0xf4240
4019ca:       48 83 c0 ff             add    rax,0xffffffffffffffff
4019ce:       75 fa                   jne    4019ca <proc_launch+0x5a>
4019d0:       e8 7b f7 ff ff          call   401150 <getppid@plt>
4019d5:       89 c7                   mov    edi,eax
4019d7:       e8 f4 00 00 00          call   401ad0 <cpures_release>
4019dc:       31 c0                   xor    eax,eax
4019de:       e8 bd 00 00 00          call   401aa0 <cpures_acquire>
4019e3:       83 c5 01                add    ebp,0x1
4019e6:       39 dd                   cmp    ebp,ebx
4019e8:       75 db                   jne    4019c5 <proc_launch+0x55>
```

unsigned long i; for (i = 0; i < 1000000UL; i++) asm(“”);

結論

因為沒有裝實體系統，所以還沒有數據證明是虛擬機造成 memory stall 時間極不穩定，但實體系統應該會好很多。還能改進的地方是排程器的效能，排程器花的時間愈少，實驗結果就愈接近理論值。像 Linux 的 Complete Fair Scheduler 是用紅黑樹來維護 Shortest Job First 的 Proiroty Queue，當 ready queue 更大時就會有效能上的明顯差異。