1. 設計

```
.
├── Makefile
├── README.md
├── demo
├── kernel_files
├── main.c
├── output
├── process.c
├── process.h
├── report.docx
├── scheduler.c
├── scheduler.h
├── script
├── testcase
└── ~$report.docx
```

檔案架構如上，process.h / process.h 中定義控制 process 的函式，用來設定 CPU affinity 及控制哪個 process 佔用 CPU。scheduler.h / scheduler.c 中定義 scheduler 在不同 policy (FIFO, RR, SJF, PSJF)下的行為。

(1) 編譯及執行

- make – 編譯
- make clean – 刪除所有.o 檔
- make run < input > output – 執行主程式

(2) Main – 讀取測資，傳給 scheduling() 進行模擬

(3) scheduling

設計理念 - 因為不能 explicitly 控制哪一個 process 會被 schedule 到，因此利用調整 process 的 priority 的方式來間接控制，比如說，利用 proc_block() 將一個 process 設定為 background process(最低 priority)後，該 process 就很難被 schedule 到。利用 proc_wakeup()，將一個 process 設定為較高 priority，使其佔用 CPU。另外，為了讓 scheduler 可以一直保持運行，scheduler 和期 child process 分別佔用一個 core (*note: 虛擬機要設定雙核以上)。

設定 scheduler core, 調整至高 priority

```
/* Set single core prevent from preemption */
proc_assign_cpu(getpid(), PARENT_CPU);
/* Set high priority to scheduler */
proc_wakeup(getpid());
```

設定好初始值以後進入 scheduling 的 while loop，首先 wait 已經結束的 process 並輸出 name 和 pid

```
if (running != -1 && proc[running].t_exec == 0) {
        waitpid(proc[running].pid, NULL, 0);
        printf("%s %d\n", proc[running].name, proc[running].pid);
        running = -1;
        finish_cnt++;
```

```
            /* All process finish */
            if (finish_cnt == nproc)
                break;
}
```

確認是否有 process 已經 ready，若有則執行並且 block 住。

```
for (int i = 0; i < nproc; i++) {
            if (proc[i].t_ready == t_cur && proc[i].pid == -1) {
                proc[i].pid = proc_exec(proc[i]);
                proc_block(proc[i].pid);
            }
}
```

利用 next_process()來 schedule 下一個 process，如果 next == -1 代表目前沒有 ready 對 process，因此不用進行 context switch。若 next != 0 且 next 並非目前的 running process 則進行 context switch。

```
int next = next_process(proc, nproc, policy);
 if (next != -1) {
     if (running != next) {
         proc_wakeup(proc[next].pid);
         proc_block(proc[running].pid);
         running = next;
         t_last = t_cur;
     }
 }
```

我們再深入一點看 next_process()在不同 policy 下的行為。
如果 policy 為 non-preemptive 且目前有 process 在 run，則直接回傳現在的 process index。

```
if (running != -1 && (policy == SJF || policy == FIFO))
        return running;
```

SJF 系列選擇剩餘執行時間最少來執行

```
if (policy == PSJF || policy ==   SJF) {
        for (int i = 0; i < nproc; i++) {
            /* process not ready for done */
            if (proc[i].pid == -1 || proc[i].t_exec == 0)
                continue;
```

```
            if (ret == -1 || proc[i].t_exec < proc[ret].t_exec)
                ret = i;
        }
    }
```

FIFO 選最早 ready 的來執行

```
else if (policy == FIFO) {
        for(int i = 0; i < nproc; i++) {
            if(proc[i].pid == -1 || proc[i].t_exec == 0)
                continue;
            if(ret == -1 || proc[i].t_ready < proc[ret].t_ready)
                ret = i;
        }
     }
```

RR 則分為多種情況。且設計較特殊。在 RR 下，t_ready 並非其最初的 ready time，而是隨時間不斷更新。每當一個 process 重新進入 ready queue 的後端時，它的 t_ready 會被設為當下的時間。因此從所有已經 ready 的 process 中選 t_ready 最小者等同從 ready queue 前端取出第一個 process。

在沒有 running process 存在時，選擇 queue 的第一個。
否則確認是否已經經過完整的 time quantum，若是的話更新 running process 的 t_ready，並選擇 queue 的第一個。
若兩種情形皆不符合代表不用進行 context switch，因此選擇 running process

```
else if (policy == RR) {
        if (running == -1){
            for(int i = 0; i < nproc; i++) {
                if(proc[i].pid == -1 || proc[i].t_exec == 0)
                    continue;
                if(ret == -1 || proc[i].t_ready < proc[ret].t_ready)
                    ret = i;
            }
        }
        else if((t_cur - t_last) % 500 == 0){
            proc[running].t_ready = t_cur; /* enter ready queue again */
            /* if a process go back to ready queue when another new process is ready,
            the old process will enter queue before the new one*/
            for(int i = 0; i < nproc; i++) {
```

```
                if(proc[i].pid == -1 || proc[i].t_exec == 0)
                    continue;
                if(ret == -1 || proc[i].t_ready < proc[ret].t_ready)
                    ret = i;
            }
        }
        else{
            return running;
        }
    }
```

(4) process

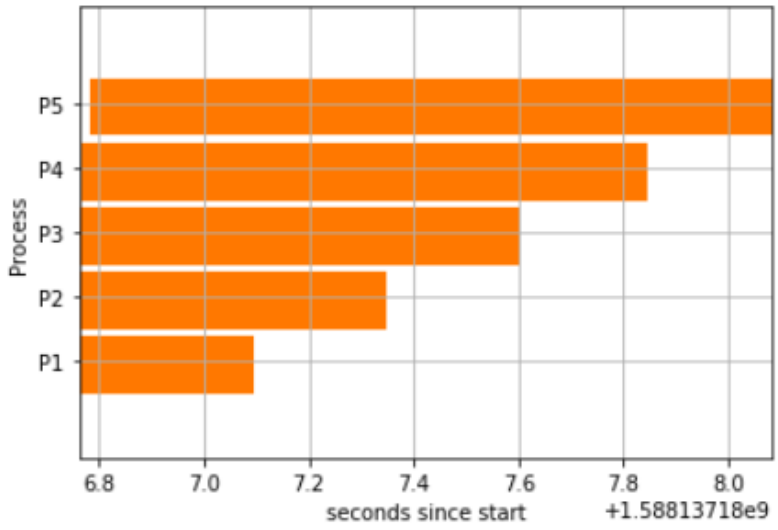
proc_block()和 proc_wakeup()前面提過了就跳過。proc_exec()負責 fork 出 child process，並且讓 child process 在執行指定個 UNIT_T()前後紀錄時間並寫入 system log。裡面使用到自行定義的 system call 333 和 334。一個用來取得當下時間，一個用來寫 system log。

```
int proc_exec(struct process proc)
{
    int pid = fork();

    if (pid < 0) {
        perror("fork");
        exit(-1);
    }

    if (pid == 0) {
        struct timespec start_time, end_time;
        char dmesg[256];
        syscall(GET_TIME, &start_time);
        // proc_assign_cpu(0, CHILD_CPU);
        // proc_block(0);
        for (int i = 0; i < proc.t_exec; i++) {
            UNIT_T();
#ifdef DEBUG
            if (i % 100 == 0)
                fprintf(stderr, "%s: %d/%d\n", proc.name, i, proc.t_exec);
#endif
        }
```

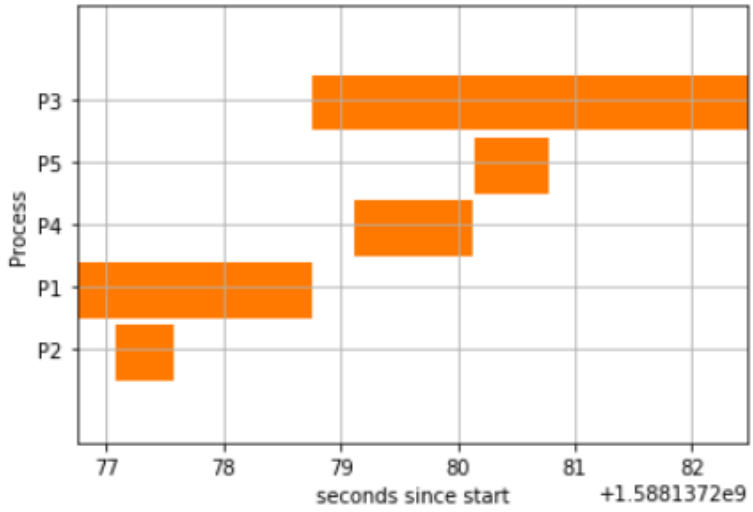
```
        syscall(GET_TIME, &end_time);
        sprintf(dmesg, "[OS project1] %d %lu.%09lu %lu.%09lu\n", getpid(),\
                start_time.tv_sec, start_time.tv_nsec, end_time.tv_sec, end_time.tv_nsec);
        syscall(PRINTK, dmesg);
        exit(0);
    }
    else{ /* Parent assign child to another core */
        proc_assign_cpu(pid, CHILD_CPU);
    }
    return pid;
}
```

2. 核心版本 linux-4.14.25
3. 理論與實際結果比較

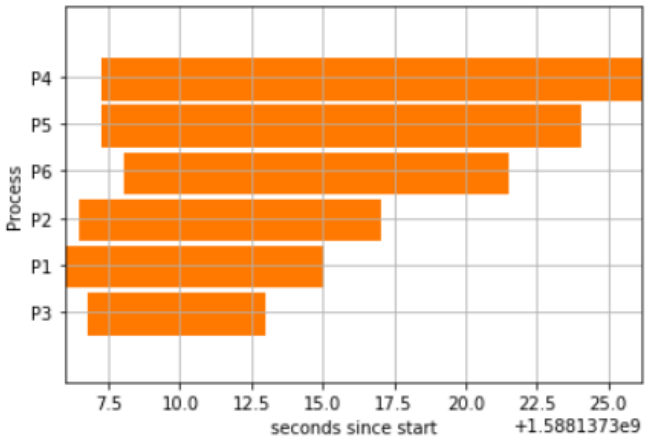
FIFO_1.txt



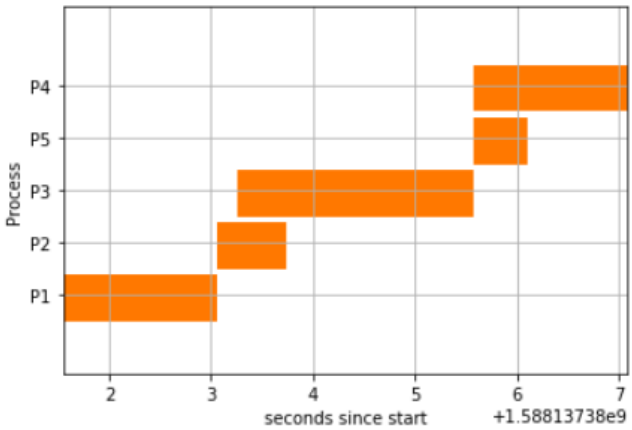
PSJF_2.txt



RR_3.txt



PSJF_4.txt



開始、結束時間順序都是正確的，但是開始時間的大小有點扭曲。原因記錄時間是由 child process 來進行，但 child process 的 priority 由 parent process 控制，如果 child process 在被 block 之前就先記錄時間則時間偏早，如果 child 在被 block 之後才記錄時間則時間會偏晚。

比如 PSJF_4 的 P4, P5 一個開始的時間分別是 5000, 7000，但兩者幾乎重疊。原因是 P4 在被創造的時候 P3 佔用 CPU 大部分時間，因此 P4 沒有馬上被 schedule 到，紀錄的開始時間就比較晚。

另一個理論跟實驗的不同是，兩個 core 運行 UNIT_T()的時間可能不同。如果 child 所在的 CPU 遠快於 parent 所在的 CPU，parent 就不能妥善控制 child 是否使用 CPU。因為 parent 在認為 child process 結束之前，如果 running child 已經結束，child 就會自己換另一個來跑，此時這個 child 是不受控制的，可能造成意外的結果。所幸在實驗中兩 CPU 速度差距不大，沒造成順序異常。

最後一個不同是，在 linux-4.14.25 中，內核使用 CFS 來進行排程，因此即便我們把 process 的 priority 設到最低，該 process 還是會被 schedule 到，只是佔用 CPU 的時間很短暫而已。但 UNIT_T()的時間足夠長，所以即便某個被 block 住的 process 被 schedule 到，也不太可能直接跑完並且記錄時間，所以順序正常。