Operating System Project 1

b06902039 資工三 賈本晧

設計理念

main.c

```
char scheduler_policy[16];
assert(scanf("%s", scheduler_policy) == 1);

int process_num;
assert(scanf("%d", &process_num) == 1);

struct process *proc;
proc = (struct process *)malloc(process_num * sizeof(struct process));

for (int n = 0; n < process_num; n++) {
    scanf("%s%d%d", proc[n].name, &proc[n].ready_time, &proc[n].exec_time);
    proc[n].pid = -1;
}

int policy;
switch(scheduler_policy[0]) {
    case 'F':
        policy = FIFO;
        break;
    case 'R':
        policy = RR;
        break;
    case 'S':
        policy = SJF;
        break;
    case 'P':
        policy = PSJF;
        break;
}
scheduling(proc, process_num, policy);
exit(0);
```

主程式讀進input，並將input傳給scheduling()。

tool.c

(1) assign_CPU()，負責將process assign到不同CPU上，parent在CPU 0上，child process在CPU 1上。

(2) proc_setscheduler()，負責調整各process的priority，如果變數是SCHED_OTHER代表priority上升，如果是SCHED_IDLE則是priority下降。

(3) proc_exec()，負責fork出新的process，並賦予struct process中的pid新process的pid，順便handle 計時。

scheduler.c

(1) scheduling()，首先先就start time進行排序。變數n_time模擬時間的推進，根據不同的policy還有時間點呼叫proc_exec()和proc_setscheduler()，且呼叫next_process()決定下一個process是誰。執行完的process會將pid設回-1。

(2) next_process()，主要根據四個policy選擇process。current代表現在正在執行的pid，沒有process在執行時current=-1。

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

如果是nonpreemptive的policy且current不是-1，就讓process繼續做。

```
else if( policy == FIFO ) {
    for(int i = 0 ; i < nproc ; i++)
        if( proc[i].pid != -1 )
            return i;

    return -1;
}
```

如果是FIFO，則依據proc的順序尋找下一個start_time已經到的process。(因為proc有先排序過，所以只需要從0到nproc去找下一個有pid的process就可以了)

```
else if( policy == RR ) {
    if( current == -1 ) {
        for(int i = 0; i < nproc; i++)
            if( proc[i].pid != -1 && proc[i].exec_time > 0)
                return i;
    }
    else if ( (ntime - t_last) % 500 == 0 ) {
        for(int i = current + 1 ; i < nproc+current ; i++) {
            if( proc[(i%nproc)].pid != -1 )
                return (i%nproc);
        }
    }
    return current;
}
```

接著處理RR的情況。如果current=-1，代表沒有process佔著資源，那麼搜尋下一個正在等待的process。如果current不等於-1，就檢查他有沒有超過500個unit time，使用 i%nproc是因為可能有編號前面的process還沒完整跑完，所以需要回頭檢查前面的process是不是都完成了。

```
else{
    int min_idx = -1;
    for( int i = 0 ; i < nproc ; i++){
        if( proc[i].pid == -1 || proc[i].exec_time == 0 )
            continue;

        if( min_idx == -1 || proc[i].exec_time < proc[min_idx].exec_time )
            min_idx = i;
    }

    if( min_idx == -1 ) return -1;

    if( current != -1 && proc[min_idx].exec_time == proc[current].exec_time ) return current;
    else return min_idx;
}
```

剩下就是SJF和PSJF，因為前面處理過SJF在current = -1時的情況，所以這邊可以一起計算。上面的迴圈是在尋找還沒執行完且exec_time最少的process，如果min_idx是 -1 代表目前沒有process在排隊，回傳 -1，如果現在有process在跑，且找到的min_idx的exec_time剛好跟現在在跑的process剩下的exec_time一樣大，選擇讓現在正在跑的process繼續跑完，若沒有上述情況，就回傳min_idx。

核心版本

```
chia@chia-VirtualBox:~$ uname -a
Linux chia-VirtualBox 4.14.25 #3 SMP Tue Apr 28 23:16:53 CST 2020 x86_64 x86_64
x86_64 GNU/Linux
chia@chia-VirtualBox:~$
```

比較實際結果與理論結果，並解釋造成差異的原因

透過各檔案的dmesg.txt和TIME_MEASUREMENT.txt的關係算出各process在執行中用了幾個unit_time。我的結果是以每個程式從start_time到結束的時間。四種方法各用一個來呈現結果。

FIFO: FIFO_2.txt

```
FIFO
4
P1 0 80000
P2 100 5000
P3 200 1000
P4 300 1000
```

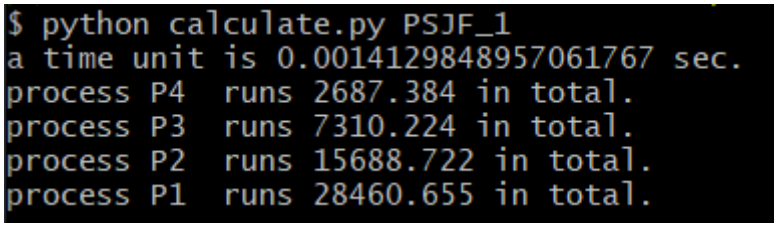
```
$ python calculate.py FIFO_2
a time unit is 0.0014129848957061767 sec.
process P1  runs 105201.256 in total.
process P2  runs 109537.141 in total.
process P3  runs 110465.278 in total.
process P4  runs 111141.886 in total.
```

理論值 P1: 80000 P2: 84900 P3: 85800 P4: 86700

可以看出 P1花了比理論值還要多出30%的時間，但是後面三個process之間的差距與各自的exec_time差不多，推測應為在執行P1時遇到比較多次需要執行排成以外的事情，或是電腦執行了其他程序，80000 unit_time更增加了這些機會，造成時間大幅增加。

PSJF: PSJF_2.txt

```
PSJF
4
P1 0 10000
P2 1000 7000
P3 2000 5000
P4 3000 3000
```

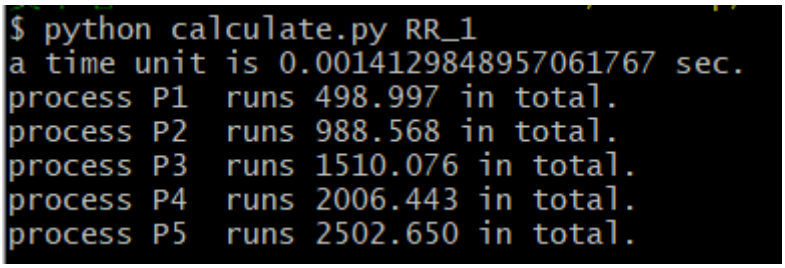


理論值 P4: 3000 P3: 8000 P2: 15000 P1: 25000

發現P4和P3都比理論值早，有可能是TIME_MEASUREMENT的標準比較大。

RR: RR_1.txt

```
RR
5
P1 0 500
P2 0 500
P3 0 500
P4 0 500
P5 0 500
```

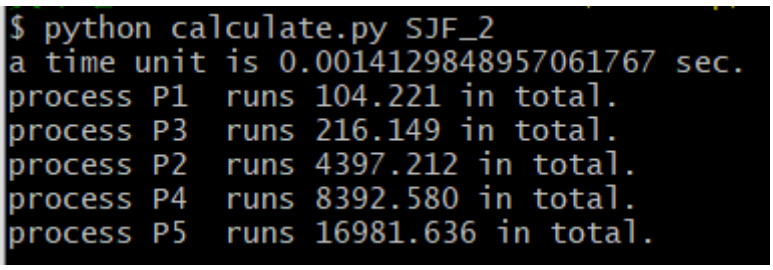


理論值 P1: 500 P2: 1000 P3: 1500 P4: 2000 P5: 2500

與結果差不多。

SJF: SJF_2.txt

```
SJF
5
P1 100 100
P2 100 4000
P3 200 200
P4 200 4000
P5 200 7000
```



理論值 P1: 100 P3: 200 P2: 4200 P4: 8100 P5: 15100

exec_time比較小的process與理論值較相近，推測與前面相同，執行時間少的遇到其他會搶資源的次數期望值也較小。

造成差異的主要原因：scheduler除了排程以外還有其他如調整priority的指令要執行，又或是程序外部有其他程序要執行，這些都會平白增加計時的時間。再者，也有可能是TIME_MEASUREMENT的參考時間不一定準確。

Represent

```
make
sudo ./main.out < <filepath> > <outputpath>
```