設計:

FIFO 原理:

先到的(先準備好可以跑的)process 先跑，剩下的 process 根據準備好的時間照順序一個一個跑。

FIFO 程式概念:

在 running == -1(當前沒有運行的 process)時，找出還沒運作完(t_exec!=0)且擁有最小 t_ready 的 pid，return 它成為下一個運行的 process。

```
    if (running != -1 && (strcmp(stringpolicy,"SJF")==0 || strcmp(stringpol
icy,"FIFO")==0))
        return running;
```

```
else if (strcmp(stringpolicy,"FIFO")==0)
{
    for(int i = 0; i < numofproc; 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 原理:

所有 process 每隔一段時間換人運作，且只有已經準備好的可以進去排隊。

RR 程式概念:此處參考郭瑋喆同學的概念，沒有 process 運作時一樣選 t_ready 最小的，過了指定時間(500 unit time)後，選出當前 t_ready 自己以外最小的，同時把自己的 t_ready 設成 ntime(當前時間)，也就是插在目前排隊的底端，但又不會排到還沒到 ready time 的 process 後面。

```
else if (strcmp(stringpolicy,"RR")==0)
{
    if (running == -1)
    {
        ret = -1;
        for(int i=0;i<numofproc;i++)
        {
            if(proc[i].pid!=-1&&proc[i].t_exec>0)
            {
                if(ret == -1)
                {
                    ret = i;
                    continue;
                }

                if(proc[i].t_ready < proc[ret].t_ready)
                    ret = i;
                else if(proc[i].t_ready == proc[ret].t_ready)
                    if(i<ret)
                        ret = i;
            }
        }
    }
```

```
else if ((ntime - t_last) % 500 == 0)
{
    proc[running].t_ready = ntime;
    ret = -1;
    for(int i=0;i<numofproc;i++)
    {
        if(proc[i].pid == -1||proc[i].t_exec == 0)
            continue;
        if(ret == -1)
        {
            ret = i;
            continue;
        }
        if(proc[i].t_ready < proc[ret].t_ready)
            ret = i;
        else if(proc[i].t_ready == proc[ret].t_ready)
        {
            if(i<ret)
                ret = i;
        }
    }
}
else
    ret = running;
```

SJF 原理:選擇已準備好且執行時間最短的 process 來運作，但 process 若跑到一半不能插隊。

SJF 程式概念:在 running == -1(沒有 process 在跑時)，選擇還沒做完(t_exec!=0)且擁有最小 t_exec 的 pid 並 return 它成下一個 process

PSJF 原理:基本概念如 SJF，但可以在 process 跑到一半時，若有一個執行時間夠短的 process 準備好了則可以插隊先做。

PSJF 程式概念:基本如同 SJF，但不考慮當前是否有 process 在運作(不考慮 running 是否等於-1)

```
    if (running != -1 && (strcmp(stringpolicy,"SJF")==0 || strcmp(stringpol
icy,"FIFO")==0))
        return running;

    int ret = -1;

    if (strcmp(stringpolicy,"PSJF")==0 || strcmp(stringpolicy,"SJF")==0)
    {
        for (int i = 0; i < numofproc; i++) {
            if (proc[i].pid == -1 || proc[i].t_exec == 0)
                continue;
            if (ret == -1 || proc[i].t_exec < proc[ret].t_exec)
                ret = i;
        }
    }
```

核心版本: 4.14.25

比較實際結果與理論結果

首先計算出我們的 unit time 長度

```
b06902057@linux2 [~/OS/output] cat TIME_MEASUREMENT_dmesg.txt
[ 8148.063725] [Project1] 3080 1588254793.538294725 1588254794.466654056
[ 8150.085940] [Project1] 3081 1588254795.500675974 1588254796.488898472
[ 8152.086619] [Project1] 3082 1588254797.519376340 1588254798.489607480
[ 8153.947415] [Project1] 3083 1588254799.444732809 1588254800.350429543
[ 8155.916167] [Project1] 3084 1588254801.363758085 1588254802.319210756
[ 8157.850814] [Project1] 3085 1588254803.286738689 1588254804.253887745
[ 8159.780672] [Project1] 3086 1588254805.223705263 1588254806.183773769
[ 8161.717430] [Project1] 3087 1588254807.159962949 1588254808.120561199
[ 8163.667625] [Project1] 3088 1588254809.103687634 1588254810.070777273
[ 8165.602145] [Project1] 3089 1588254811.041909806 1588254812.005333197
```

總和 (5000 unit time) = 9.566291216

平均 (500 unit time) = 0.9566291216

Unit time = 0.0019132582

以下每個方法均取範例測資 2 為例:

FIFO2

```
b06902057@linux2 [~/myproj1/OS_PJ1_Test] cat FIFO_2.txt
FIFO
4
P1 0 80000
P2 100 5000
P3 200 1000
P4 300 1000b06902057@linux2 [~/myproj1/OS_PJ1_Test]
```

```
b06902057@linux2 [~/OS/output] cat FIFO_2_stdout.txt
P1 2301
P2 2302
P3 2303
P4 2304
b06902057@linux2 [~/OS/output] cat FIFO_2_dmesg.txt
[ 5384.714483] [Project1] 2301 1588251879.676942168 1588252031.078496999
[ 5398.186241] [Project1] 2302 1588252034.714491502 1588252044.550436934
[ 5400.163752] [Project1] 2303 1588252044.552683284 1588252046.527974388
[ 5402.140707] [Project1] 2304 1588252046.529273221 1588252048.504955911
```

理論:

P1 = 80000 * unit time = 153.060656

P2 = 5000 * unit time = 9.566291

P3 = P4 = 1000 * unit time = 1.9132582

實際:

P1 = 151.401554831

P2 = 9.835945432

P3 = 1.975281104

P4 = 1.97568269

RR2

```
b06902057@linux2 [~/myproj1/OS_PJ1_Test] cat RR_2.txt
RR
2
P1 600 4000
P2 800 5000
```

```
b06902057@linux2 [~/OS/output] cat RR_2_dmesg.txt
[ 7410.002647] [Project1] 2881 1588254041.710061651 1588254056.394750717
[ 7412.731539] [Project1] 2882 1588254042.692910824 1588254059.123682736
```

理論:

P1 = 500 * ((4000/500)*2-1) * unit time = 14.3494365

P2 = 8500 * unit time = 16.2626947

實際:

P1 = 14.684689066

P2 = 16.430771912

SJF2

```
b06902057@linux2 [~/myproj1/OS_PJ1_Test] cat SJF_2.txt
SJF
5
P1 100 100
P2 100 4000
P3 200 200
P4 200 4000
P5 200 7000b06902057@linux2 [~/myproj1/OS_PJ1_Test]
```

```
b06902057@linux2 [~/OS/output] cat SJF_2_dmesg.txt
[ 7899.178494] [Project1] 3031 1588254545.381762035 1588254545.577755878
[ 7899.577160] [Project1] 3033 1588254545.579937601 1588254545.976427314
[ 7907.562784] [Project1] 3032 1588254545.978363075 1588254553.962167542
[ 7915.442452] [Project1] 3034 1588254553.963963038 1588254561.841953045
[ 7929.332582] [Project1] 3035 1588254561.843669180 1588254575.732287109
b06902057@linux2 [~/OS/output]
```

理論:

P1 = 100 * unit time = 0.19132582

P2 = 4000 * unit time = 7.6530328

P3 = 200 * unit time = 0.38265164

P4 = 4000 * unit time = 7.6530328

P5 = 7000 * unit time = 13.3928074

實際:

P1 = 0.195993843

P2 = 7.983804467

P3 = 0.396489713

P4 = 7.877990007

P5 = 13.888617929

PSJF2

```
b06902057@linux2 [~/myproj1/OS_PJ1_Test] cat PSJF_2.txt
PSJF
5
P1 0 3000
P2 1000 1000
P3 2000 4000
P4 5000 2000
P5 7000 1000b06902057@linux2 [~/myproj1/OS_PJ1_Test]
```

```
b06902057@linux2 [~/OS/output] cat PSJF_2_stdout.txt
P2 2415
P1 2414
P4 2417
P5 2418
P3 2416
b06902057@linux2 [~/OS/output]
```

```
b06902057@linux2 [~/OS/output] cat PSJF_2_dmesg.txt
[ 6291.410593] [Project1] 2415 1588252935.795287395 1588252937.787352287
[ 6295.192385] [Project1] 2414 1588252933.777056936 1588252941.569190201
[ 6301.145618] [Project1] 2417 1588252943.667874050 1588252947.522495690
[ 6303.236276] [Project1] 2418 1588252947.629188517 1588252949.613170603
[ 6309.455888] [Project1] 2416 1588252941.707431250 1588252955.832867316
b06902057@linux2 [~/OS/output]
```

理論:

P1 = 4000 * unit time = 7.6530328

P2 = 1000 * unit time = 1.9132582

P3 = 7000 * unit time = 13.3928074

P4 = 2000 * unit time = 3.8265164

P5 = 1000 * unit time = 1.9132582

實際:

P1 = 7.792133265

P2 = 1.992064892

P3 = 14.125436066

P4 = 3.85462164

P5 = 1.983982086

誤差原因:

它計算的時間不全是在跑 child process，排程跟執行新的 process 等都會算進

去，然後我在 fork 完為了避免 child 在我把它 assign 到一個 CPU 上跑之前就先偷跑，我有加一個 usleep(1000)，雖然不會造成自己的計時誤差，但會造成其他 process 的誤差，同時每次的 unit time 都會有浮動，或者說 CPU 本來就無法固定速度在一個很精確的值上，這也是理論數據跟實際數據不同的原因。

附註問題:
有時候執行./main < xx.txt > xx_stdout.txt 的時候，會有某些值重複印在 output 檔裡面，但直接印在螢幕上時不會出錯
dmesg 檔的[Project1]前面會印出數字，但不知道原因為何
kernel 編譯完每次開機都會出問題，用 recovery mode 進去才能正常開機