學號： R08922167 姓名：曾民君

1. 設計:

A. Makefile:

將所有檔案 (main.c, sheduler.c, process.c)包成 demo 執行檔

```
CC = gcc
.PHONY: clean
objs:= main.o scheduler.o process.o
demo: $(objs)
    $(CC) -o demo $(objs)
main.o: main.c process.h scheduler.h
    $(CC) main.c -c
scheduler.o: scheduler.c scheduler.h
    $(CC) scheduler.c -c
process.o: process.c process.h
    $(CC) process.c -c
clean:
    rm -f $(objs)
run:
    sudo ./demo
```

B. Main:

這邊讀取 input 所有資訊，其中 process 為自定義的 structure，定義在 process.h 檔案中，然後因為 input 的每行資訊都是空白做間隔，所以直接使用 scanf 將 process name, start time, end time 分別讀入到 process 中

```
scanf("%s",policy);
scanf("%d",&N);

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

for( int i = 0; i < N; i++ ){
    scanf("%s%d%d",proc[i].name,&proc[i].R,&proc[i].T);
}
```

然後再依據事先讀入到的 policy 名稱，分別去呼叫相對應的 cpu 排程

```
if( strcmp(policy,"FIFO") == 0 )
    scheduling(proc,N,FIFO);
else if( strcmp(policy,"RR") == 0 )
    scheduling(proc,N,RR);
else if( strcmp(policy,"SJF") == 0 )
    scheduling(proc,N,SJF);
else if( strcmp(policy,"PSJF") == 0 )
    scheduling(proc,N,PSJF);
```

c. **Scheduler:**

PSJF 或 SJF 的執行優先順序，是比較還沒結束的 process 還剩下的 execution time 的長短為依據，剩餘時間越少則優先權越高

```
int now = -1;
if (policy == PSJF || policy ==  SJF) {
    for (int i = 0; i < number; i++) {
        if (proc[i].pid == -1 || proc[i].T == 0)
            continue;
        if (now == -1 || proc[i].T < proc[now].T)
            now = i;
    }
}
```

FIFO 的執行優先順序，是比較 ready time 的先後順序，至於題目中多個同時到達的話，就看哪個 process 搶先取得初始時間

```
else if(policy == FIFO){
    for(int i=0; i<number; i++){
        if(proc[i].pid == -1 || proc[i].T ==0)
            continue;
        if(now == -1 || proc[i].R < proc[now].R)
            now = i;
    }
}
```

RR 的執行順序，是固定每個 process 每次能夠執行的時間，並且只要在限定時間內沒有完成所有工作內容，就必須暫停執行並交由下一個排程的 process 執行，並且重新排隊等待執行機會

```
else if (policy == RR){
    if(idofrp == -1){
        for(int i=0; i<number; i++){
            if(proc[i].pid != -1 && proc[i].T > 0){
                now = i;
                break;
            }
        }
    }
    else if((current_unit - lastcs) % 500 == 0){
        now = (idofrp + 1) % number;
        while(proc[now].pid == -1 || proc[now].T == 0)
            now = (now + 1) % number;
    }
    else
        now = idofrp;
}
```

最後如果所有的Process都結束了就跳出迴圈

```
if (idofrp != -1 && proc[idofrp].T == 0) {
    waitpid(proc[idofrp].pid, NULL, 0);

    setvbuf(stdout, NULL, _IONBF, 0);
    printf("%s %d\n", proc[idofrp].name, proc[idofrp].pid);
    idofrp = -1;
    finish++;

    /* 所有process都完成了結束while*/
    if (finish == number)
        break;
```

D. Process

Process.c的作用為提供剛剛兩個程式所需對cpu執行的函數 首先proc_assign_cpu在scheduling function的開頭有用到，用作將process指派到特定CPU

```
int proc_assign_cpu( int pid, int cpu )
{
    if( cpu > sizeof( cpu_set_t ) ){
        printf( "Error: Assign to wrong CPU." );
        return -1;
    }

    cpu_set_t cpu_assign;
    CPU_ZERO( &cpu_assign );
    CPU_SET( cpu, &cpu_assign );

    if( sched_setaffinity( pid, sizeof( cpu_assign ), &cpu_assign ) < 0 ){
        printf( "Error: Set process affinity error." );
        exit( 1 );
    }

    return 0;
}
```

proc_exec function 作用為執行process

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

    if( pid < 0 ){
        printf( "Error: Fork error." );
        return -1;
    }
    if( pid == 0 ){ //Child process.
        struct timespec ts_start, ts_end;
        syscall( GET_TIME, &ts_start);
        for( int i = 0; i < proc.T; i++ ){
            UNIT_T();
        }
        syscall( GET_TIME, &ts_end);
        syscall( PRINTK, getpid(), &ts_start, &ts_end );
        exit( 0 );
    }

    proc_assign_cpu( pid, CHILD_CORE );

    return pid;
}
```

proc_block function 跟 proc_wakeup function 這兩個函數目的為開始與暫停特定process，透過增加或是減少process之間相對的priority來達成這個作用，他們的差別在於一個使用SCHED_OTHER另一個則是SCHED_IDLE

proc_exec function作用為執行process

```c
int proc_block(int pid)
{
    struct sched_param param;

    /* SCHED_IDLE should set priority to 0 */
    param.sched_priority = 0;

    int ret = sched_setscheduler(pid, SCHED_IDLE, &param);

    if (ret < 0) {
        perror("sched_setscheduler");
        return -1;
    }

    return ret;
}

int proc_wakeup(int pid)
{
    struct sched_param param;

    /* SCHED_OTHER should set priority to 0 */
    param.sched_priority = 0;

    int ret = sched_setscheduler(pid, SCHED_OTHER, &param);

    if (ret < 0) {
        perror("sched_setscheduler");
        return -1;
    }

    return ret;
}
```

2. 核心版本： 4.14.25

環境：Virtual Box，Ubuntu 16.04

安裝過程遇到的問題：

第一次安裝記憶體切不夠，導致安裝到一半失敗

第二次安裝不小心安裝到 32 bits 版本，導致開機無法正常讀取安裝核心

第三次安裝遇到跟其他人需要重新 make install 才能正常重啟的問題

3. 比較實際結果與理論結果，並解釋造成差異的原因：

實際結果跟理論沒有所有 case 都相同。 以FIFO 舉例，輸出的測資執行時間、pid等資訊是正確的，但是有時輸出並不是按照順序 如下列為其中一次執行 FIFO 測資的輸出:

資料：

FIFO
5
P1 0 500
P2 0 500
P3 0 500
P4 0 500
P5 0 500

結果：

[1417.136230] [Project1] 3444 1588152486.435688175 1588152486.659623864
[1417.334374] [Project1] 3445 1588152486.467843999 1588152486.857768708
[1417.516532] [Project1] 3446 1588152486.432595226 1588152487.39926149
[1417.708617] [Project1] 3447 1588152486.459642898 1588152487.232010817
[1417.878175] [Project1] 3448 1588152486.463670394 1588152487.401568964

原因應該是我使用排班的函數是 qsort ， 使用當下沒有注意到他不是 stable 的排序，使得相同 start time 的 process 並不一定會依照 pid 順序執行。另外實際執行時間比理想的執行時間多，可能原因是自己寫的 for 迴圈可能效率不好，還有計時跟打印結果的時間也都會干擾實際執行時間。

另外有發現說在執行 FIFO 時，理論上是要等到前者完全執行完後，後者才可以執行，但是打印出所有執行過程，發現偶爾會發生其他正在等待的 process 偷偷執行一點點的工作，不知是否是原本 process 執行到一半被不相關的 process 強行插隊，或是備系統插隊，導致順序上偶爾會有些波動。