1）設計：

使用單核心的 VM 跑
main.c:
主程式，讀入 policy，然後依據 policy 用 execlp 呼叫處理該 policy 的程式。
FIFO.c:
1.處理 FIFO 這個 policy。
2.先讀入 process 的數量，再讀入各 process 的 name,ready time 和 execution time。
3.先根據 ready Time 把所有 process sort 一遍（採用的是 qsort）.
4.while（1）裡，每次檢查是否有程式跑完，如果有就把那個程式的 name 和 PID 輸出到 stdout，finish_count++，如果 finish_count 等於 process 的數量，就是說所有程式跑完，跳出迴圈。
5.檢查完是否有程式跑完後，我們就檢查在當下的時間點是否有 process 已經 ready，如果有我們就呼叫，exec_proc 給予它 PID 並執行，之後再呼叫 block_proc 把那個 process 先 block 在 CPU 外等候。
6.檢查完是否有程式已經 ready 後，我們便要進入選擇 next process 的階段，呼叫 next_process 函式，next_process 函式為會先檢查是否有程式在跑，如果有便回傳該程式的編號（因為 FIFO 是 non-preemptive）否則就依序尋找下一個已 ready 的 process。

SJF.c:
1.處理 SJF 這個 policy。
2.先讀入 process 的數量，再讀入各 process 的 name,ready time 和 execution time。
3.先根據 ready Time 把所有 process sort 一遍（採用的是 qsort）.
4.while（1）裡，每次檢查是否有程式跑完，如果有就把那個程式的 name 和 PID 輸出到 stdout，finish_count++，如果 finish_count 等於 process 的數量，就是說所有程式跑完，跳出迴圈。
5.檢查完是否有程式跑完後，我們就檢查在當下的時間點是否有 process 已經 ready，如果有我們就呼叫，exec_proc 給予它 PID 並執行，之後再呼叫 block_proc 把那個 process 先 block 在 CPU 外等候。
6.檢查完是否有程式已經 ready 後，我們便要進入選擇 next process 的階段，呼叫 next_process 函式，next_process 函式為會先檢查是否有程式在跑，如果有便回傳該程式的編號（因為 SJF 是 non-preemptive）否則就尋找下一個已經 ready 的 process 且 execution time 是最短的。

PSJF.c:
1.處理 PSJF 這個 policy。
2.先讀入 process 的數量，再讀入各 process 的 name,ready time 和 execution time。
3.先根據 ready Time 把所有 process sort 一遍（採用的是 qsort）.
4.while（1）裡，每次檢查是否有程式跑完，如果有就把那個程式的 name 和 PID 輸出到 stdout，finish_count++，如果 finish_count 等於 process 的數量，就是說所有程式跑完，跳出迴圈。
5.檢查完是否有程式跑完後，我們就檢查在當下的時間點是否有 process 已經 ready，如果有我們就呼叫，exec_proc 給予它 PID 並執行，之後再呼叫 block_proc 把那個 process 先 block 在 CPU 外等候。

6.檢查完是否有程式已經 ready 後，我們便要進入選擇 next process 的階段，呼叫 next_process 函式，next_process 會為我們選擇當下已經 ready 且 execution time 是最短的 process。

RR.c:
1.處理 RR 這個 policy。
2.先讀入 process 的數量，再讀入各 process 的 name,ready time 和 execution time。
3.先根據 ready Time 把所有 process sort 一遍（採用的是 qsort）.
4.while（1）裡，每次檢查是否有程式跑完，如果有就把那個程式的 name 和 PID 輸出到 stdout，finish_count++，如果 finish_count 等於 process 的數量，就是說所有程式跑完，跳出迴圈。
5.檢查完是否有程式跑完後，我們就檢查在當下的時間點是否有 process 已經 ready，如果有我們就呼叫，exec_proc 給予它 PID 並執行，之後再呼叫 block_proc 把那個 process 先 block 在 CPU 外等候。
6.檢查完是否有程式已經 ready 後，我們便要進入選擇 next process 的階段，呼叫 next_process 函式，next_process 會檢查是否有程式在跑，如果沒有，便把當前 ready queue 的第一個 process 放上去跑。如果有，便檢查當前在跑的程式是否已經到了 quantum time，如果還沒，就讓該程式繼續跑，否則便把該程式拉下來換成 ready queue 的第一支程式上去跑。被拉下來的程式會把他的 ready time 設定為當前的 time，然後再把所有 process sort 一邊，以達到 ready queue 是 FIFO 的規則。

scheduling.c:
主要有四個函式，分別是 assign_cpu(),exec_proc(),block_proc(),wakeup_proc()。
assign_cpu()這個函式主要是 assign cpu 給某個 pid，當中要使用了 sched_setaffinity()來制定某支 process 對 CPU 的親和性。

```
int assign_cpu(int pid,int n){
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(n,&mask);
    if(sched_setaffinity(pid,sizeof(mask),&mask)<0){
        fprintf(stderr, "sched_setaffinity\n");
        exit(0);
    }
    return 0;
}
```

exec_proc():
是當某個 process ready 以後，便 fork 一個小孩去執行此 process。Child process 會用 syscall(334)取得開始時間，然後執行完後，再用 syscall(334)取得結束時間，之後再用 syscall(335)把 pid，開始時間和結束時間都寫到 kernel message。

```
int exec_proc(struct Process process){
    int pid = fork();
    if(pid < 0){
        fprintf(stderr, "exec_proc fork failed\n" );
        perror("exec_proc failed");
        return -1;
    }
    if (pid==0){
        //fprintf(stderr, "child created\n" );
        long start_sec,end_sec;
        char toprint[512];
        start_sec = syscall(334);
        for(int i = 0;i < process.execT;i++){
            volatile unsigned long j;
            for(j = 0;j < 1000000UL;j++);
        }
        end_sec = syscall(334);
        syscall(335,getpid(),start_sec,end_sec);
        exit(0);
    }
    assign_cpu(pid,1);
    return pid;
}
```

block_proc():
會使用 sched_setscheduler()把當前的 pid 的 priority 調成 SCHED_IDLE（也就是極低的 priority）以藉此把此 process 擋在 cpu 外。

```
47 int block_proc(int pid){
48
49     struct sched_param parameter;
50     parameter.sched_priority =0;
51     int ret = sched_setscheduler(pid,SCHED_IDLE,&parameter);
52     return ret;
53 }
54
```

wakeup_proc():
會使用 sched_setscheduler()把當前的 pid 的 priority 調成 SCHED_OTHER（高於 SCHED_IDLE 的 priority）以藉此把此 process 叫醒。

```
55 int wakeup_proc(int pid){
56     struct sched_param parameter;
57     parameter.sched_priority =0;
58     int ret = sched_setscheduler(pid,SCHED_OTHER,&parameter);
59
60     return ret;
61
62 }
```

2 ）核心版本：

```
zhiguan@zhiguan-VirtualBox:~/Desktop$ uname -a
Linux zhiguan-VirtualBox 4.14.25 #4 SMP Tue Apr 28 22:46:44 CST 2020 x86_64 x86
_64 x86_64 GNU/Linux
```

3）比較實際結果和理論結果的差異：

觀察：

經過比對，發現實際結果的執行時間會比理論結果的執行時間長。

原因：

1 . 電腦正在跑的程式非常多，我們的 VM 佔用的 CPU 可能發生 context switch，但時間會繼續不會因為 context switch 而停止

2 . 除了 process 跑迴圈的時間，scheduler 其實還有其他方面例如檢查是否有 process ready 之類，也會產生時間上的誤差。