Project 1 Report

我的核心版本是 4.14.25 版，在我的 demo 一開始有展示。

```
ed@ed-VirtualBox:~/Desktop/Project1$ uname -r
4.14.25
ed@ed-VirtualBox:~/Desktop/Project1$ cat /proc/cpuinfo | grep
processor | wc -l
4
ed@ed-VirtualBox:~/Desktop/Project1$
```

以下開始程式設計的介紹：

```c
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <sys/time.h>
```

首先上方是我的標頭檔。

```c
#define time_unit  { volatile unsigned long i; for(i=0;i<1000000UL;i++); }
#define swap(x,y)   {int tmp=x;x=y;y=tmp;}
#define timequa 500
```

上方 define 分別是 time_unit、交換和 RR 的 time quantum，因為很常用，為求整潔寫成 define。

```c
struct process{
    char N[33];
    short done;
    unsigned int R;
    unsigned int T;
    pid_t PID;
    struct timespec start;
    struct timespec end;
};
```

以上是我用存取 process 資料與狀態的資料結構，目的是希望變數清楚明瞭。N 是 process 名稱，done 是已被 fork()，R 是 ready time，T 是 execution time，PID 是該 process 取得之 pid，timespec start、end 則是儲存第一次進 CPU 和最後結束時間。

```c
void FIFO(struct process *p,int n);
void RR(struct process *p,int n);
void SJF(struct process *p,int n);
void PSJF(struct process *p,int n);
```

我的各個排程方式都寫在個別的子程式，主程式只負責讀寫資料。

主程式介紹：

```
int main(int argc,char *argv[]){ //1,2
    //open file
    printf("input file : %s\n",argv[1]);
    printf("output file : %s\n",argv[2]);
    FILE *fin;
    fin=fopen(argv[1],"r");
    char S[5];
    int n;
    fscanf(fin,"%s",S);
    fscanf(fin,"%d",&n);
```

首先我的讀、寫檔是透過 argc 及 argv 完成，所以輸入時不加< >，舉例如 ./a.out FIFO_1.txt FIFO_1_stdout.txt。主程式開頭就是讀取 argv 檔名，並從檔案中讀取指令。

```
struct process *p = (struct process *)malloc(n*sizeof(struct process));
// initialize done
for (int i = 0; i < n; ++i){
    p[i].done=0;
    fscanf(fin,"%s %u %u",p[i].N,&p[i].R,&p[i].T);
}
```

Process *p 建造 n 個儲存各 process 狀態的資料結構。並初始化其中的 done 值。

```
if (strncmp(S,"FIFO",5)==0){
    FIFO(p,n);
}
else if (strncmp(S,"RR",5)==0){
    RR(p,n);
}
else if (strncmp(S,"SJF",5)==0){
    SJF(p,n);
}
else if (strncmp(S,"PSJF",5)==0){
    PSJF(p,n);
}
```

讀取指令要的排程方式，進入不同子程式執行。

```
//output
FILE *fout;
fout=fopen(argv[2],"w+b");
for (int i = 0; i < n; ++i){
    fprintf(fout,"%s %d\n",p[i].N,p[i].PID);
    printf("%s %d\n",p[i].N,p[i].PID);
}
return 0;
```

子程式完畢後，打印出結果，同時也輸出成檔案。

FIFO 介紹：

```
void FIFO(struct process *p,int n){
    cpu_set_t set;
    CPU_ZERO(&set);
    int time=0;
```

首先，先開好需要的變數，set 為之後設定 cpu_affinity 要用到的變數，而 time 則是記錄目前總共已經跑了多久。

```
for (int i = 0; i < n; ++i){
    //select process
    int j;
    for (j = 0; j < n ; ++j){
        if(p[j].done==1) continue;
        if(p[j].R <= time) break;
    }
    // if no processes ready
    if (j == n){ time_unit;  i--;  time++;  continue;}
```

接著進入 for 迴圈，for 迴圈的條件是，每跑完一個 process 則 i+1，直到所有 process 都跑完了。而迴圈一開始要先選出 ready time 比 time(總時間)小的，代表已 ready，若有一樣的 ready time 則排序靠前的先。若沒有 ready 的 process 則主程式自己跑一個 time_unit，並把時間+1，而以下的程式就都先跳掉。

```
int status;
CPU_SET(1, &set);
sched_setaffinity(0 , sizeof(cpu_set_t) , &set);
p[j].PID = fork();
switch(p[j].PID){
    // PID == -1 代表 fork 出錯
    case -1:
        perror("fork()");
        exit(-1);

    // PID == 0 代表是子程序
    case 0:
        printf("i'm process %d.\n",j);
        syscall(333,&p[j].start);
        for (int k = 0; k < p[j].T; ++k)  time_unit;
        syscall(333,&p[j].end);
        syscall(334,getpid(),&p[j].start,&p[j].end);
        printf("[Project1] %d %lu.%lu %lu.%lu\n",getpid(),p[j].start.tv_sec,p
        exit(-1);
```

若有找到 ready 的 process，則先設定到 1 號 core 然後 fork，使之後所有子程式都在 1 號 core 跑。其中 pid = case 0 為 child，一旦要開始跑 time_unit 先 syscall(333, …)記錄當下時間，跑完 timespec 時也立即儲存當下時間。並 syscall(334, …)，打印在 kernel 打印資訊，最後直接 exit(-1)避免子程式繼續跑。

syscall(333, …)內容如下

```c
#include <linux/linkage.h>
#include <linux/kernel.h>
#include <linux/ktime.h>
#include <linux/timekeeping.h>

asmlinkage void sys_my_get(struct timespec *t){
    printk("my_get is invoked!\n");
    getnstimeofday(t);
}
```

syscall(334, …)內容如下

```c
#include <linux/linkage.h>
#include <linux/kernel.h>
#include <linux/ktime.h>
#include <linux/timekeeping.h>

asmlinkage void sys_my_print(int pid,struct timespec *s,struct timespec *e){
    printk("[Project1] %d %lu.%lu %lu.%lu\n",pid,s->tv_sec,s->tv_nsec,e->tv_sec,e->tv
}
```

以下為父程式

```c
        // PID > 0 代表是父程序
        default:
            CPU_SET(0, &set);
            sched_setaffinity(0 , sizeof(cpu_set_t) , &set);
            time+=p[j].T;
            p[j].done=1;
            wait(&status);
    }
}
```

而父程式則是控制在 0 號 core 上執行。先將總時間加上目前在跑的子程式跑完時間，並更改該子程式為已 fork()，最後使用 wait，block 直到 child 送出執行完畢的 signal。

誤差討論：

TIME_MEASUREMENT

24504	1588137687	1588137688	1.00817	500
24506	1588137689	1588137690	1.0638	500
24516	1588137691	1588137692	1.0318	500
24517	1588137693	1588137694	1.04364	500
24518	1588137695	1588137696	1.02914	500
24519	1588137698	1588137699	0.98964	500
24520	1588137700	1588137701	1.01234	500
24521	1588137702	1588137702	0.94608	500
24522	1588137703	1588137704	1.00178	500
24523	1588137705	1588137706	1.01418	500

每個 process 平均花 1.014057 秒，每個 time_unit 平均花 0.002028114 秒。

以 FIFO_1 分析

24547	1588137734	1588137735	1.055949926
24548	1588137735	1588137736	1.009250164
24549	1588137736	1588137737	1.017670155
24558	1588137737	1588137738	1.045490026
24559	1588137738	1588137739	0.98890996

每個 process 平均花 1.023454046，每個 time_unit 平均花 0.002046908 秒

與 TIME_MEASUREMENT 比較之誤差為 0.00926673747，接近 1%，這個誤差其實不算大，但在程式中感覺已經蠻大的了，照理來說 FIFO 應是誤差最小的，因為程式一旦開跑就記錄開始時間，一旦結束就記錄結束時間，中間並沒有任何多餘動作，其誤差可能來自 cpu 剛好多跑一點或少跑一點，而且畢竟我們是在虛擬機裡，虛擬機還需要跟其他程式搶 cpu。所以每次結果依據電腦的狀態不同都有一點點微小的差異。

PSJF 介紹：

```
void PSJF(struct process *p,int n){
    int time=0;int i=0;
    cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(0, &set);
    sched_setaffinity(0 , sizeof(cpu_set_t) , &set);
    //create pipe
    int pp[n][2];
    for (int k = 0; k <= n; ++k){
        if (pipe(pp[k])<0){
            perror("fork()");
            exit(1);
        }
    }
```

首先一樣先令好變數，設定好 core，但 PSJF 因為各個 fork 不能一次做完整個 execution time，因此需要溝通什麼時候停下，什麼時候繼續跑。我的方法是最多能有 n 個子程式就建 n+1 個 pipe，前 0 ~ n-1 個提供父程式告知對應的第 0 ~ n-1 個子程式要跑多少個 time_unit，最後第 n 個 pipe 則用在當子程式跑完該跑的量之後回報給父程式，讓父程式知道可以叫下一位跑了。

```
int childnum = 0;
while (i < n){
    //select and fork ready process
    for (int j = 0; j < n ; ++j){
        if(p[j].done==1 || p[j].R > time) continue;
        childnum++;
```

接著進入 while 圈，i 為完成的 process 個數。for 迴圈則是找有沒有還不是已經 fork 而已經到了 ready time 的 process。有則 childmun++代表目前有正在運行子程式個數。

```
    //fork the process
    p[j].done = 1;
    p[j].PID = fork();
    switch(p[j].PID){
        // PID == -1 代表 fork 出錯
        case -1:
            perror("fork()");
            exit(-1);

        // PID == 0 代表是子程序
        case 0:;
            //initialize
            int isfirst = 1;
            char inbuf[11];  int runtime;
            int k;
            char buf[11]; buf[01]='1';
            int exec=p[j].T;
            while(exec>0){
                CPU_SET(2, &set);
                sched_setaffinity(0 , sizeof(cpu_set_t) , &set);
                read(pp[j][0], inbuf, 10);
                sscanf(inbuf,"%d",&runtime);
                printf("I'm process %d.\n",j+1);
                CPU_SET(1, &set);
                sched_setaffinity(0 , sizeof(cpu_set_t) , &set);
                if(isfirst) {syscall(333,&p[j].start);  isfirst=0;}
                for (k = 0; k < runtime; ++k)  time_unit;
                exec -= runtime;
                if (exec <= 0){
                    syscall(333,&p[j].end);
                    syscall(334,getpid(),&p[j].start,&p[j].end);
                    printf("[Project1] %d %lu.%lu %lu.%lu\n",getpid(),p[j].s
                }
                sprintf(buf, "%d", k);
                write(pp[n][1], buf, 10);
            }
            exit(-1);
```

接著直接 fork 剛剛抓到 ready 的 process。子程式的部分最一開始就是令變數，isfirst 是記錄這個程式是不是第一次跑 time_unit 以記錄開始時間。而 inbuf、buf 則是供 pipe 使用的讀取、寫入 buf。Exec 代表執行時間，歸零則表程式結束。子程式剛進入時都會先到 2 號 core 做 read 的 block，以免影響真的在跑的程式。而 read 會 block 直到父程式 pipe 傳送要求該子程式跑多少個 time_unit，一旦接收到執行命令，才會換回 1 號 core 跑父程式所要求的量。若 exec==0 該程式完全執行完畢，當然就是立刻讀取當下時間。接著才回傳訊號，最後才離開。

```
        // PID > 0 代表是父程序
        default:;
            //printf("child was born.\n");
    }
}
// process control
if (childnum > 0){
    //choose the ready and shortest process
    int running = -1;
    for (int j = 0; j < n; ++j){
        //printf("p[%d].T  %d\n",j,p[j].T);
        if (p[j].R <= time && p[j].T!=0 && (running == -1 || p[j].T < p[runni
    }
    //calculate runtime
    int runtime = -1;   int tmp;
    for (int j = 0; j < n; ++j){
        if(j == running || p[j].T==0)  continue;
        tmp = p[j].R-time;
        if (tmp < 0) tmp=0;
        //test preemptive :ture for no preemptive
        if (p[j].T >= p[running].T - tmp)  tmp = p[running].T;
        //preemptive and the shortest
        if(runtime > tmp || runtime == -1) runtime = tmp;
    }
    if(runtime == -1) runtime = p[running].T;
    //printf("running %d runtime %d\n",running,runtime);
    char buf[11];
    sprintf(buf, "%d", runtime);
    write(pp[running][1], buf, 10);
    //printf("unlock process %d.\n",running+1);
    read(pp[n][0],buf,10);
    //convert str to int
    int timeplus;
    sscanf(buf,"%d",&timeplus);
    time += timeplus;
    //adjust T
    p[running].T -= runtime;
    if (p[running].T == 0){
        childnum--;
        i++;
    }
    continue;
}
```

父程式則是當有子程式時，要挑選要跑哪個 process 以幾可以跑多久。不放在 default 是因為 default 包在 for 裡，若連續 fork()時會出錯。挑選的部分由 running 紀錄，挑選方式為已經 ready、非跑完且執行時間最短的程式。而 runtime 則是記錄剛程式可以跑多久，取決於將來會不會被 preempt，判斷方式為拿剛剛選中的 running process 分別和其他程式比較，當然自己和跑完的就不用比了，其他程式的 ready time 減當下時間，若非負則代表程式若被 preempt 可執行的時間，若不會被 preempt 則代表他能完整跑完他自己的 execution time。而會不會被 preempt 的判斷是目前這個 process 的 execution time 減到下個程式的 ready time 所剩的時間與另一個程式的 execution time 的比較，若較小則不會被 preempt。最後選出最小的 tmp 即為所要求知 runtime，若找完 runtime 仍為-1 則代表沒其他人與之比較，表示只

剩它了，那 runtime 就是 execution time。
最後挑好 runing 讓誰跑，runtime 跑多久，就可以用 running 的那個 pipe 告知 running 的子程式要跑多久了。

```
        //no process ready
        time_unit;  time++;
    }
}
```

最後則是沒有任何子程式在跑時，父程式自己運算時間。

誤差討論：

TIME_MEASUREMENT

每個 time_unit 平均花 0.00202811408 秒。

以 PSJF_1 分析

0 1 2 3 2 1 0

1000 1000 1000 3000 4000 6000 9000

P1 跑 25000 time_unit

P2 跑 15000 time_unit

P3 跑 8000 time_unit

P4 跑 3000 time_unit

24741	1588138135	1588138141	6.03082	3000	0.00201
24740	1588138133	1588138149	16.04291	8000	0.002005
24739	1588138131	1588138160	29.94122	15000	0.001996
24730	1588138128	1588138178	49.97077	25000	0.001999

每個 time_unit 平均花 0.00199972 秒

與 TIME_MEASUREMENT 比較之誤差為-0. 01400019919，1.4%，誤差比 FIFO 大，但理論上 PSJF 的時間應該要比 FIFO 還要長，因為每個子程式間若有被打斷都需要等待父程式控制，而父程式要計算出控制誰控制多久是需要時間的，但此結果卻比 TIME_MEASUREMENT 的值還小，可見執行那些運算指令造成的延遲仍微乎其微，影響主要還是來自當下的電腦狀態。

RR 介紹：

```
void RR(struct process *p,int n){
    int time=0;int i=0;
    cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(0, &set);
    sched_setaffinity(0 , sizeof(cpu_set_t) , &set);
    //create pipe
    int pp[n][2];
    for (int k = 0; k <= n; ++k){
        if (pipe(pp[k])<0){
            perror("fork()");
            exit(1);
        }
    }
```

此步驟和 PSJF 相同。一樣是適用 pipe 讓父與子溝通。

```
//initialize
int *readyqueue = (int *)malloc((n+1)*sizeof(int));
int front=0; int back=0; int tmp=0;
```

建 ready queue，將 ready 之程式納入之列隊，已排隊獲取 time quantum 的執行時間，我的實作方式是用 circular array 做 queue，因此需要 n+1 個空間，使 front==end 時 array 必為空而非滿。

```
while (i < n){
    //fork ready process
    for (int k = 0; k < tmp; ++k){
        //get process number from queue
        int queuenum = (back-k-1)%(n+1);
        if (queuenum < 0) queuenum += (n+1);
        int j=readyqueue[queuenum];
        if (p[j].done==1){tmp++;continue;}
        p[j].done=1;
```

進入 while 圈，i 為執行完畢之 process 個數。For 圈找到 ready queue 中需要被 fork()的程式，其中 tmp 為新加入 queue 之子程式，但 tmp 的計算在程式末段。

```
//fork the process
p[j].PID = fork();
switch(p[j].PID){
    // PID == -1 代表 fork 出錯
    case -1:
        perror("fork()");
        exit(-1);

    // PID == 0 代表是子程序
    case 0:;
    //initialize
        char inbuf[4];
        int isfirst = 1;
        int l;
        char buf[4]; buf[01]='1';
        int exec=p[j].T;
        while(exec>0){
            CPU_SET(2, &set);
            sched_setaffinity(0 , sizeof(cpu_set_t) , &set);
            read(pp[j][0], inbuf, 1);
            printf("I'm process %d.\n",j+1);
            CPU_SET(1, &set);
            sched_setaffinity(0 , sizeof(cpu_set_t) , &set);
            if(isfirst) {syscall(333,&p[j].start);  isfirst=0;}
            for (l = 0; l < timequa && l < exec; ++l)  time_unit;
            exec -= 500;
            if (exec <= 0){
                syscall(333,&p[j].end);
                syscall(334,getpid(),&p[j].start,&p[j].end);
                printf("[Project1] %d %lu.%lu %lu.%lu\n",getpid(),p[j].st
            }
            sprintf(buf, "%d", l);
            write(pp[n][1], buf, 4);
        }
        exit(-1);
```

fork()剛剛從 ready queue 挑出的新加入 ready queue 的程式，子程式和 PSJF 大同小異，只差在 RR 不需要知道要跑多少個 time_unit，反正固定跑 time quantum 的量。

```
            // PID > 0 代表是父程序
            default:;
                //printf("child was born.\n");
        }
    }
    // process control
    if (front != back){
        char buf[4];
        int pop = readyqueue[front];
        write(pp[pop][1], "1",1);
        //printf("unlock process %d.\n",pop+1);
        read(pp[n][0],buf,4);
        //convert str to int
        int timeplus;
        sscanf(buf,"%d",&timeplus);
        time+=timeplus;
        //select ready process
        tmp=0;
        for (int k = 0; k < n ; ++k){
            if(p[k].done==1) continue;
            if(p[k].R <= time) {readyqueue[back]=k;  back++;  back=back%(n+1);
        }
        //adjust readyqueue
        p[pop].T -= 500;
        //i can't use p[readyqueue[front]].T to judge
        int strange = p[pop].T;
        if(strange > 0) {
            //printf("move %d to back %d\n",pop,back);
            readyqueue[back] = pop;
            back = (back+1)%(n+1);
            front = (front+1)%(n+1);
        }
        else{
            front = (front+1)%(n+1);
            i++;
        }
        continue;
    }
```

父程式的部分，從 ready queue 中取 front 出來，用 pipe 叫該子程式，並 read() block 住等子程式跑好，回傳跑了多少，以計算總時間。並將在這期間 ready 的程式放入 ready queue，新放入的 process 數量即 tmp。最後才將剛剛 front 的那個程式移至 back，back 和 front 都加一，若剛剛 front 的程式已經跑完了它的 execution time 則不在需要把它放到 back，移動 front+1 即可，並將 i++以幾路完成的 process 數量，全部完成時就會出大 while 圈了。

```
        //select ready process
        tmp=0;
        for (int k = 0; k < n ; ++k){
            if(p[k].done==1) continue;
            if(p[k].R <= time) {readyqueue[back]=k;  (back++)%(n+1);  tmp++;}
        }
        time_unit;  time++;
    }
}
```

而最後方這些則是沒有子程式在 run 時，父程式自己跑時間以及檢查是否

又任何程式 ready 了，有則將 ready 的程式放入 ready queue，新放入的 process 數量即 tmp。

誤差討論：

TIME_MEASUREMENT

每個 time_unit 平均花 0.00202811408 秒。

以 RR_2 分析

~600 後

1 2 1 2 1 2 1 2 1 2 1 2 1 2
500 500 500 500 500 500 500 500 500 500 500 500 500 500
1 2 2 2
500 500 500 500

P1 跑 7500 time_unit

P2 跑 8500 time_unit

24939	1588138438	1588138453	15.26454997	7500	0.002035273
24940	1588138439	1588138456	17.23669004	8500	0.002027846

每個 time_unit 平均花 0.002031328 秒。

與 TIME_MEASUREMENT 比較之誤差為 0.00158468403，0.158%，這次的結果就相當接近 TIME_MEASUREMENT 的結果，但不確定是不是因為父程式對子程式的控制時間導致的，由前兩項的誤差來看，可能主要還是來自電腦當下的狀態。

SJF 介紹:

```
void SJF(struct process *p,int n){
    int time=0;
    cpu_set_t set;
    CPU_ZERO(&set);
    for (int i = 0; i < n; ++i){
        //select process
        int j=-1; int len=-1;
        for (int k = 0; k < n ; ++k){
            if(p[k].done==1) continue;
            if(p[k].R <= time && (p[k].T<len || len==-1)) {len = p[k].T;  j=k;}
        }
```

與 FIFO 作法相似，僅挑選方式不同，是在同樣的 ready 的 process 中挑選 execution time 最短的一個。

```
        // if no processes ready
        if (j == -1){ time_unit;  i--;  time++;  continue;}
        CPU_SET(1, &set);
        sched_setaffinity(0 , sizeof(cpu_set_t) , &set);
        int status;
        p[j].PID = fork();
        switch(p[j].PID){
            // PID == -1 代表 fork 出錯
            case -1:
                perror("fork()");
                exit(-1);

            // PID == 0 代表是子程序
            case 0:
                printf("i'm process %d.\n",j);
                syscall(333,&p[j].start);
                for (int k = 0; k < p[j].T; ++k)  time_unit;
                syscall(333,&p[j].end);
                syscall(334,getpid(),&p[j].start,&p[j].end);
                printf("[Project1] %d %lu.%lu %lu.%lu\n",getpid(),p[j].start.tv_sec,p
                exit(-1);

            // PID > 0 代表是父程序
            default:
                CPU_SET(0, &set);
                sched_setaffinity(0 , sizeof(cpu_set_t) , &set);
                time+=p[j].T;
                p[j].done=1;
                wait(&status);
        }
    }
}
```

而下方 fork()程式與 FIFO 完全相同。

誤差討論：

TIME_MEASUREMENT

每個 process 平均花 1.014057 秒，每個 time_unit 平均花 0.00202811408 秒。

以 RR_2 分析

2 3 4 1

2000 1000 4000 7000

P1 跑 7000 time_unit

P2 跑 2000 time_unit

P3 跑 1000 time_unit

P4 跑 4000 time_unit

24939	1588138438	1588138453	15.26454997	7500	0.002035273
24940	1588138439	1588138456	17.23669004	8500	0.002027846

每個 time_unit 平均花 0.002016475 秒。

與 TIME_MEASUREMENT 比較之誤差為-0.00573882927，0.57%，最後這個結果有差不多落在 1%左右，感覺取決於，CPU 當下的忙碌狀態、溫度、超頻、快

取、記憶體等等的數據影響較嚴重，各種雖然不是能非常精準的該跑多久就跑多久，但其結果的震盪似乎超過了方法帶來的影響，因此看不出什麼方法對於效能產生的差距。