OS Project1 Report

1. 設計：

   - System calls: 實作了兩個函式分別是findtime跟printk。

     findtime: findtime使用了getnstimeofday這個函式來獲得時間資訊，並且把tv_sec的值乘上10的9次方加上tv_nsec的值return回去。這樣做的好處是不用傳參數進去findtime函式裡面，來自user space的data，有的時候需要用copy_from/to_user之類的函式來處理，顯得有點麻煩。

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

     asmlinkage unsigned long long int sys_findtime(void){
             struct timespec time;
             getnstimeofday(&time);
             return time.tv_sec*1000000000+time.tv_nsec;
     }
     ```

     printk: printk我實作的很簡單，因為我在user space就先把要印的message整理好放在一個字串裡面當參數傳進來printk，所以只是單純的把message印出來而已。

     ```
     #include<linux/kernel.h>
     #include<linux/linkage.h>


     asmlinkage void sys_printk(char *msg){
             printk("[Project1] %s",msg);
     }
     ```

   - Main: 把data讀進來並且把排程演算法的字串對應到一個define在scheduler.h裡面的數字，這樣之後寫起來會方便許多不用一直strcmp兩個演算法。最後印出所有child process的pid跟名稱。

     ```
     int map_policy(char *policy){
         if(strcmp(policy,"FIFO") == 0)
             return FIFO;
         else if(strcmp(policy,"RR") == 0)
             return RR;
         else if(strcmp(policy,"SJF") == 0)
             return SJF;
         else if(strcmp(policy,"PSJF") == 0)
             return PSJF;
         else{
             fprintf(stderr,"Policy name can't be found.\n");
             exit(0);
         }
     }

     ```

   - Scheduler: 控制child process的執行順序，依照演算法選出下個執行的人，利用sched_scheduler調整process的cpu priority，以此來模擬context switch。

     在進入實際排程之前，我先把所有process依照他們的ready time sort過。這樣一來，我在做FIFO演算法時，只要從頭開始找並且一遇到已經ready且execution time不是零的process就可以把它return回去了。然後還有在做RR演算法時，我只需要從前一個執行的process的index往後找到第一個已經ready且execution time不為零的process即可把它return。這樣便可以模擬ready queue的運作。

     還有，我利用函式sched_setaffinity將scheduler自己放在一顆cpu上；其他child們一起放在另外一顆cpu上。如此一來scheduler的執行時間就不會影響到排程的表現。

     每一輪，我先檢查上一輪執行的process是不是已經跑完他的execution time了。如果是，就利用waitpid使得scheduler 被block住直到那個process printk完並且exit。

     ```
             if(running != -1){
                 if(proc[running].remain == 0){
                     proc_priorup(proc[running].pid);
                     //fprintf(stderr,"%d %d\n",time,running);
                     exit_proc += 1;
                     // suspend until the child terminate
                     if(waitpid(proc[running].pid,NULL,0) < 0)
                         fprintf(stderr,"Wait Error.\n");
                     running = -1;
                 }
             }
     ```

     再來，我檢查有沒有process在這一輪要進入ready queue等待執行。有的話，就跑函式proc_exec去fork出一個child來代表它。並且我將這個process的index放進我RR排程演算法專用的ready queue裡面。

```
        for(int i = 0; i < procnum; i++)
            if(proc[i].ready == time){
                proc[i].pid = proc_exec(proc[i]);
                inqueue(i);
            }
```

接著，就進入選下一個要執行的process的部分。根據不同的排程演算法我寫了一共四個函式。

- FIFO:很單純地，如果沒有process在執行，就從頭找一遍直到第一個ready且remaining execution time不為零的process；如果有process在執行，則讓它繼續執行，因為是non-preemtive的演算法。

```
int sched_next_FIFO(Process *proc, int procnum){
    int nextid = -1;
    if(running == -1){
        for(int i = 0; i < procnum; i++)
            if(proc[i].pid != -1 && proc[i].remain != 0){
                nextid = i;
                break;
            }
    }
    else
        nextid = running;
    return nextid;
}
```

- RR:我用queue的概念去實作process等待的ready queue。有process還沒執行完他的execution time就被context switch掉的話，就把它放進queue裡面等待(放進去的部分寫在sched_scheduling那邊)；且如果遇到需要找process執行的狀況(沒人執行或是執行的人跑完一次quantum of time)，就從queue的頭拿出一個element當作下一個執行的process，如果queue裡面沒有element了就回傳-1或是現在正在跑的這個process(取決於現在跑的這個process結束它的execution time了沒)。

```
int sched_next_RR(Process *proc, int procnum){
    int nextid = -1;
    if((time-last_cs)%500==0 || running == -1){
        if(head == tail)
            nextid = running;
        else
            nextid = dequeue();
    }
    else
        nextid = running;
    return nextid;
}
```

- SJF:如果沒有process正在執行，就找所有ready且remaining execution time不為零的process中execution time最短的那個去執行；要不然就讓現在正在跑的process繼續跑(因為這個演算法是non-preemtive)。

```
int sched_next_SJF(Process *proc, int procnum){
    int nextid = -1;
    if(running == -1){
        int minid = -1, min = MAXINT;
        for(int i = 0; i < procnum; i++){
            if(proc[i].pid == -1 || proc[i].remain == 0)
                continue;
            if(proc[i].remain < min){
                minid = i;
                min = proc[i].remain;
            }
        }
        nextid = minid;
    }
    else
        nextid = running;
    return nextid;
}
```

- PSJF:找所有ready且remaining execution time不為零的process中remaining execution time最短的那個去執行。

```
int sched_next_PSJF(Process *proc, int procnum){
    int nextid = -1;
    int minid = -1, min = MAXINT;
    for(int i = 0; i < procnum; i++){
        if(proc[i].pid == -1 || proc[i].remain == 0)
            continue;
        if(proc[i].remain < min){
            minid = i;
            min = proc[i].remain;
        }
    }
    nextid = minid;
    return nextid;
}
```

選完下個要執行的process之後，我利用sched_scheduler這個函式來把現在正在跑的process的cpu priority降低，把接下來要跑的process的cpu priority提高。判斷有沒有發生context switch，如果有的話要維護last_cs跟RRqueue這兩個為RR演算法設計的東西。

並且讓scheduler跟著被選中的child一起跑一次Unit of time的時間。

```
144         if(nextidx != -1 && nextidx != running){
145             last_cs = time;
146             proc_priorup(proc[nextidx].pid);
147             if(running != -1){
148                 proc_priordown(proc[running].pid);
149                 prerunning = running;
150             }
151         }
152         if(running != -1 && nextidx != running && proc[running].remain != 0)
153             inqueue(running);
154         running = nextidx;
155         Unittime();
156         if(running != -1)
157             proc[running].remain -= 1;
158         time += 1;
```

- Child: fork出來之後先記錄下自己的start time(跟它的ready time一樣)，然後會先暫時被踢出cpu等待scheduler把它喚醒。被喚醒的話就會進去for裡面跑一次Unit of time(被喚醒N次理論上會跑大約N圈for)。跑完execution time次數的for迴圈之後，紀錄自己結束的時間並且printk然後exit。

```
189 pid_t proc_exec(Process P){
190     int childpid = fork();
191     if(childpid == 0){
192         unsigned long long int start, end;
193         start = syscall(334);
194         for(int i = 0; i < P.remain; i++)
195             Unittime();
196         end = syscall(334);
197         char mesg[100];
198         sprintf(mesg,"%d %llu.%09llu %llu.%09llu\n", getpid(), start/1000000000, start%1000000000, end/1000000000, end%1000000000);
199         //fprintf(stderr,"%d %d\n",time,getpid());
200         syscall(335,mesg);
201         exit(0);
202     }
203     proc_cpuassign(childpid,1);
204     proc_priordown(childpid);
205     return childpid;
206 }
```

2. 核心版本：linux-4.14.25

3. 實際結果跟理論結果的誤差：基本上99%情況下跑出來的結束順序都會跟實際上的結果一樣。但很偶爾會在SJF排程演算法的第三筆test data時出現類似race condition的狀況(有嘗試去解決但未找出確切的原因)。我在執行printk之前，有的時候某幾個特定的process會被scheduler context switch掉。不過我自己trace code了一整天還是不覺得哪裡奇怪，因為照理來說我的scheduler會在waitpid()那邊等到child執行完printk並且exit之後才會繼續往下找下一個執行的process，因此感覺不會context switch。如果助教發現我的bug，希望能寄信跟我說。感謝您！另外其他跟實際狀況不一樣的地方，大概是因為我選下一個執行process的演算法有待改進，執行效率跟實際的狀況比起來會比較差。理論上如果用priority queue來實作PSJF演算法的話會快一些。

```
110         if(running != -1){
111             if(proc[running].remain == 0){
112                 proc_priorup(proc[running].pid);
113                 //fprintf(stderr,"%d %d\n",time,running);
114                 exit_proc += 1;
115                 // suspend until the child terminate
116                 if(waitpid(proc[running].pid,NULL,0) < 0)
117                     fprintf(stderr,"Wait Error.\n");
118                 running = -1;
119             }
120         }
```