

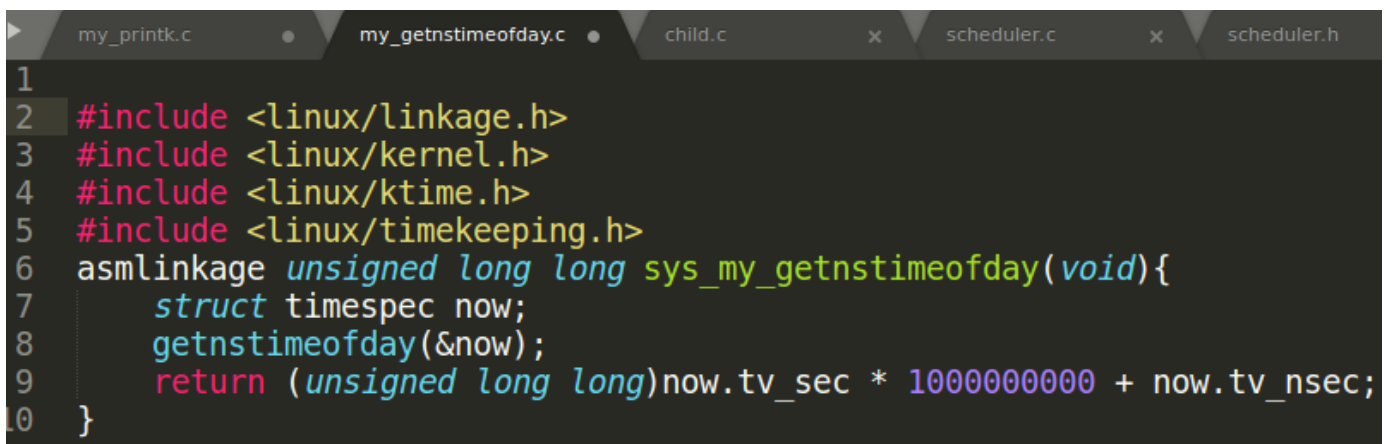
OS Report

B07902048 資工二 李宥霆

1. 設計

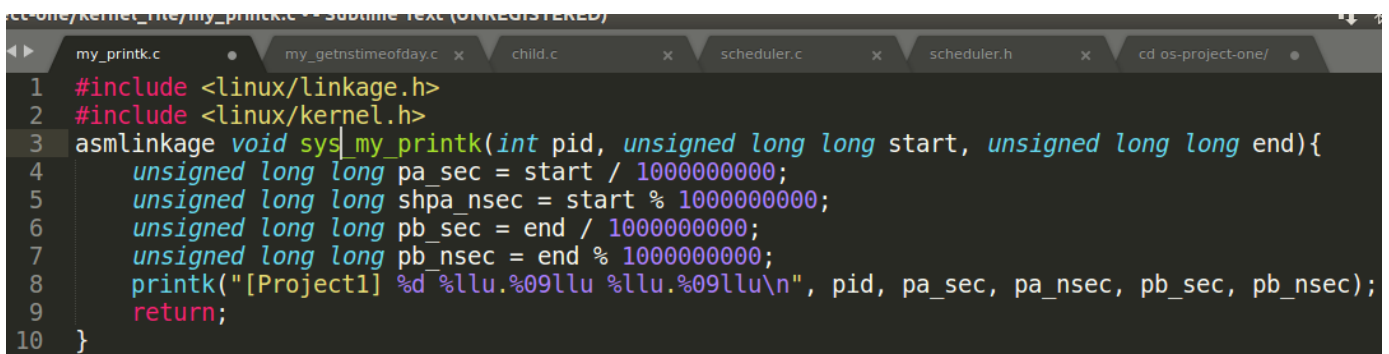
System call部份：

(1) 此system call以回傳unsigned long long的方式，用於獲取child process的開始與結束時間



```
1
2 #include <linux/linkage.h>
3 #include <linux/kernel.h>
4 #include <linux/ktime.h>
5 #include <linux/timekeeping.h>
6 asmlinkage unsigned long long sys_my_getnstimeofday(void){
7     struct timespec now;
8     getnstimeofday(&now);
9     return (unsigned long long)now.tv_sec * 1000000000 + now.tv_nsec;
10 }
```

(2) 此system call把傳入該child process的pid與兩個時間值，經過處理後把時間print在kernel裡面



```
1 #include <linux/linkage.h>
2 #include <linux/kernel.h>
3 asmlinkage void sys_my_printk(int pid, unsigned long long start, unsigned long long end){
4     unsigned long long pa_sec = start / 1000000000;
5     unsigned long long shpa_nsec = start % 1000000000;
6     unsigned long long pb_sec = end / 1000000000;
7     unsigned long long pb_nsec = end % 1000000000;
8     printk("[Project1] %d %llu.%09llu %llu.%09llu\n", pid, pa_sec, pa_nsec, pb_sec, pb_nsec);
9     return;
10 }
```

Scheduler部份：

(1) 首先在main function裡面把CPU set成第0顆，然後用sched_FIFO的方式把scheduler的priority設為50，之後讀進input並將其依照ready time進行排序，再以變數ready_num代表當下時間進入到ready queue的task數量，最後初始化share memory好就開始跑scheduler的程式。使用share memory是因為設定scheduler的參數sched_FIFO，在process遇到I/O的部份還是會被踢出CPU，無法滿足預期的行為，因此改用memory mapping的方式來確保scheduler可以傳資料給child process。

```

int main(){
    Set_cpu();
    Set_priority(getpid(), 50, -1);
    Read_input();
    Init_shm();
    Scheduler();
    for(int i = 0; i < N; i++){
        wait(NULL);
    }
}

```

(2) Scheduler決定要使用哪一種規則

```

void Scheduler(){
    if(strcmp(policy, "FIFO") == 0) FIFO();
    else if(strcmp(policy, "RR") == 0) RR();
    else if(strcmp(policy, "SJF") == 0) SJF();
    else if(strcmp(policy, "PSJF") == 0) PSJF();
}

```

(a) FIFO:

如果ready queue當下沒有任何process在等待，則叫scheduler自己跑距離下一個process開始的時間長度，跑完之後更新now_time，就可以fork出新的child process。

如果ready queue裡面有process在等待，則先進入ready queue的process先跑，除非中間有新的process要被create，否則直接讓該process跑到結束，直到所有process執行完畢則跳出迴圈。

```

78 void FIFO(){
79     int i = -1;
80     while(1){
81         //fprintf(stderr, "[scheduler] now_time = %d\n", now_time);
82         if(ready_num < N && !Task_is_in_ready_queue()){
83             Run_a_clock_time(task[ready_num].ready_time - now_time);
84             now_time = task[ready_num].ready_time;
85             Start_new_tasks();
86             continue;
87         }
88         Pick_next_job(&i);
89         while(Time_remain_create_task() < task[i].exec_time){
90             Assign_time_to_child(i, task[ready_num].ready_time - now_time);
91             Start_new_tasks();
92         }
93         Assign_time_to_child(i, task[i].exec_time);
94         Start_new_tasks();
95         if(Is_terminated()) break;
96     }
97 }

```

(b) RR:

如果ready queue當下沒有任何process在等待，則作法與FIFO相同。

如果ready queue裡面有process在等待，則每支process最多可以跑500個clock time，跑完後就要換下一支在ready queue裡的process跑。

```
98 void RR(){
99     int i = -1;
100     while(1){
101         if(ready_num < N && !Task_is_in_ready_queue()){
102             Run_a_clock_time(task[ready_num].ready_time - now_time);
103             now_time = task[ready_num].ready_time;
104             Start_new_tasks();
105             continue;
106         }
107         int round_remain = 500;
108         Pick_next_job(&i);
109         if(task[i].exec_time > round_remain){
110             while(Time_remain_create_task() < round_remain){
111                 round_remain -= task[ready_num].ready_time - now_time;
112                 Assign_time_to_child(i, task[ready_num].ready_time - now_time);
113                 Start_new_tasks();
114             }
115             Assign_time_to_child(i, round_remain);
116             Start_new_tasks();
117         }
118         else{
119             while(Time_remain_create_task() < task[i].exec_time){
120                 Assign_time_to_child(i, task[ready_num].ready_time - now_time);
121                 Start_new_tasks();
122             }
123             Assign_time_to_child(i, task[i].exec_time);
124             Start_new_tasks();
125         }
126         if(Is_terminated()) break;
127     }
128 }
```

(c) SJF:

如果ready queue當下沒有任何process在等待，則做法與FIFO相同。

如果ready queue裡面有process在等待，則每次執行時都挑裡面最短exec time的process進入CPU，直到該process結束後再挑選下一個最短的process。

```
139 void SJF(){
140     int i = -1;
141     while(1){
142         if(ready_num < N && !Task_is_in_ready_queue()){
143             Run_a_clock_time(task[ready_num].ready_time - now_time);
144             now_time = task[ready_num].ready_time;
145             Start_new_tasks();
146             continue;
147         }
148         Pick_shortest_job(&i);
149         while(Time_remain_create_task() < task[i].exec_time){
150             Assign_time_to_child(i, task[ready_num].ready_time - now_time);
151             Start_new_tasks();
152         }
153         Assign_time_to_child(i, task[i].exec_time);
154         Start_new_tasks();
155         if(Is_terminated()) break;
156     }
157 }
```

(d) PSJF:

如果ready queue當下沒有任何process在等待，則做法與FIFO相同。

如果ready queue裡面有process在等待，則挑選當下最短exec time的process，跟SJF的差別在於：當有新的process被加入ready queue，則會重新檢查ready queue裡所有process的剩餘exec time，且挑選最短的process進入CPU

```
158 void PSJF(){
159     int i = -1;
160     while(1){
161         if(ready_num < N && !Task_is_in_ready_queue()){
162             Run_a_clock_time(task[ready_num].ready_time - now_time);
163             now_time = task[ready_num].ready_time;
164             Start_new_tasks();
165             continue;
166         }
167         Pick_shortest_job(&i);
168         if(Time_remain_create_task() < task[i].exec_time){
169             Assign_time_to_child(i, task[ready_num].ready_time - now_time);
170             Start_new_tasks();
171         }
172         else{
173             Assign_time_to_child(i, task[i].exec_time);
174             Start_new_tasks();
175         }
176         if(Is_terminated()) break;
177     }
178 }
```

2. 核心版本

linux 4.14.25

3. 比較實際結果與理論結果，並解釋造成差異的原因

比較結果後，可以發現順序上跟理論值相同，但在執行時間上，會因為scheduler排程造成的overhead而使實際時間比理論時間久且process之間的context switch存在overhead。