2020 OS Project 1 – Report

B06202018 數學三 張芷榕

1. 設計

- **kernel files**

```
asmlinkage int sys_get_time(unsigned long *sec, unsigned long *nsec);
```

用 `getnstimeofday` 來獲得時間。

```
asmlinkage int sys_show_info(char *output);
```

用 `printk` 來 output information。

- **process.c**

自定義結構型態 process 來存取 process 的 information：

```
1.  typedef struct {
2.      pid_t pid; // pid of the process
3.      char name[32]; // name of the process
4.      int exec_t; // the amount of time the process still need to be executed
5.      int ready_t; // the time the process is ready
6.  } process;
```

```
void proc_assign_cpu(pid_t pid, int cpu);
```

利用 `sched_setaffinity` 將所有 process assign 到同一 CPU 來模擬排程。

```
pid_t proc_exec(process *p);
```

根據 process 的 information 來執行此 process。

在 fork 出的 child process 用 system call 獲得開始執行的時間，然後依照要求 run 數個 unit time，結束後同樣地用 system call 獲得結束時間，並 output 到 dmesg。

parent process 則會回傳 child process 的 pid。

```
void proc_blck(pid_t pid);
```

利用 `sched_setscheduler(pid, SCHED_IDLE, &param)`的方式來 block 指定的 process。

```
void proc_wake(pid_t pid);
```

利用 `sched_setscheduler(pid, SCHED_OTHER, &param)`來提高指定

process 的 priority，使得此 process 能被執行。

- **schedule.c**

用來記錄排程 information 的一些變數：

```
1.  // initialization
2.  int cur_t = 0; // current time
3.  int num_fin = 0; // number of finish process
4.  int last_p = -1; // index of last executed process, -1 implies none
5.  int last_t; // the last time context switch happens, used for RR policy
```

```
int sched_next(process *P, int num_p, int policy);
```

根據 scheduling policy 找出下一秒要執行的 process，return 此 process 在 process list 中的 index。

屬於 non-preemptive 的 SJF 和 FIFO，在目前執行的 process 尚未結束的情況下，不能選擇別的 process，故 return `last_p`。

剩下為 preemptive policy，或是 non-preemptive 但目前沒有 process 在執行的情況。

1. SJF, PSJF
   在 process list 中挑出待執行時間 `exec_t` 最短的 process，即為下一秒要執行的 process。
2. FIFO
   因為 process list 在一開始已經依照 ready 的時間從早排序，所以 list 中第一個已經 ready 並且還沒結束執行的 process 即為下一秒要執行的 process。
3. RR
   若距離上次 context switch 已經過了 500 個 unit time，或是目前沒有 process 在執行，便從 list 中尋找下一個可執行（已經 ready 並且還沒結束）的 process 來執行，不然仍舊選擇現在執行的 process 來 return。

```
void scheduler(process *P, int num_p, int policy);
```

將 process list 中的 process 依照 policy 模擬排程。

首先用 `proc_assign_cpu(getpid(), SCHED_CPU)`及 `proc_wake(getpid())`，使得 scheduler 能獨自使用一顆 CPU，不與排程的 process 混在一起執行。

將 process list 依照每個 process 的 ready time 進行 sorting，以便後續的操作。

初始時先將每個 process 的 pid 設成-1，表示這個 process 還沒有 ready。

每一秒檢查是否有 process 在這一秒 ready（`P[i].ready_t == cur_t`），若是 ready 了就呼叫 `proc_exec(&P[i])`來執行並取得此 process 的 pid，隨後使用 `proc_blck(P[i].pid)` 先將 process block 起來，等待排程來決定執行的時間。

再來呼叫 `sched_next(P, num_p, policy)`來決定要執行的 process，並用 `proc_blck(P[last_p].pid)`和 `proc_wake(P[cur_p].pid)`進行 context switch。

執行一個 unit time 後，將目前執行的 process 的待執行時間（`P[cur_p].exec_t`）減一，若時間剩下為零表示 process 執行結束。依照要求將需要的 information output 到 stdout，並將記錄完成 process 數的 `num_fin` 加一。

等到所有 process 結束執行（`num_fin == num_p`），便結束排程。

- **main.c**

  根據 stdin 的 input 創造一個 process list P，再呼叫 `scheduler(P, num_p, policy)`進行排程。

2. 核心版本

4.14.25

3. 理論與實際結果的比較

比較實際結果與理論結果，process 完成的順序是正確的。因為將 sheduler 與 process 分開在不同 CPU 執行，所以能有較為準確的結束時間計算，但每個 process 的結束時間仍然多少會和理論有所出入，這可能是因為在執行 context switch 時的 overhead 所導致。