Reference: B07902058

1. scheduler design:

   Since I design the scheduler for single core, to make sure my scheduler would work, I implemented two functions:

   1. set_priority(pid_t pid, int priority):
      Assign the priority of the selected process.
   2. create_share_variable(char *filename):
      Since the modification of some variables in the child process can't be recognized by the parent process, I create some share variables in the physical interface to help my parent processes control the process.

   For every case, I first read the files and sort the cases by their ready time. I also created some data structures to help support my method, including queue and circular link list, which are used to simulate the ready queue.
   A share variable timer records the time, and a share variable finish determines which processes has finished.

   (1) FIFO:

   For FIFO it's simple. The scheduler will run an idle process if there's no task in the ready queue and no task has arrived. When the timer has reached a task's ready time, put the task into a queue, which is the ready queue, and fork it. Upon fork, record the process id and set the process' priority to a low priority to return the CPU back to scheduler. After creating every child process, the scheduler determines which task arrived first in the ready queue, which is the first process in the queue. Scheduler then assign a high priority to the process, in order to allow the task into CPU. The task would run the same amount units of time as its execute time. When the executing is finished, the child process would set the share variable finish to its task id in order to tell the parent process that it has terminated. Then , the scheduler will eliminate the process from the ready queue.

   (2) SJF:

   Most part of the SJF is the same as FIFO. The only difference is that upon searching, the scheduler will do a linear search in the ready queue to find the shortest task, and give it the priority. The rest is the same as FIFO.

   (3) RR:

   For round-robin, instead of using queue for process queue, I use circular link instead. The adding and creating process part is the same as FIFO. The difference is that now the child process will not only return the CPU

when terminated, but also return it when 500 units of time has passed. Therefore, when timer reaches 500, the process will set its priority to low and return the CPU back to scheduler, which will then decide the next process to execute. When a new process is ready, it will be inserted in front of the current running process in the circular link list, which represent the current tail of the ready queue.

(4) PSJF:

PSJF is almost the same as SJF. However, the running child process needs to return the CPU whenever a new task is ready. Therefore, I created a shared variable next for the child process to notice when the next process in the task list will be ready, and a counter to see how many units of time has the process run. The task will be sent back to the ready queue when a new task is ready, and its execute time will be subtracted by the counter. Then, the scheduler will add the next tasks into the ready queue, and do a linear search to find the shortest job like SJF.

(5) system call:

I implement two system calls, get_process_time() and print_process_time(). get_process_time() will return the system time in long long format, which is $\text{timespec.tv\_sec} \times 10^9 + \text{timespec.tv\_nsec}$. Timespec is the result of getnstimeofday. print_process_time() simply restores the process time back, and print them out by printk.

2. Core version:

```
b07902062@b07902062-VirtualBox:~$ uname -a
Linux b07902062-VirtualBox 4.14.25 #5 SMP Sun Apr 26 21:27:04 CST 2020 x86_64 x8
6_64 x86_64 GNU/Linux
```

3. I noticed that the time result in printk is much larger than the executing time. The reason may be the two of the below: the start time of the process is set by the time the child process is created, and the context switch overhead and the time cost by scheduler may be huge. For example, doing linear search in SJF may cause the time for scheduler to be large as number of task grows. Also, context switching between scheduler and process is also a waste of CPU resource. For round-robin, the difference is especially huge.