OS 2020 Spring Project1 Report

系級:資工二 學號: b07902064 姓名:蔡銘軒

1. Design
   - Common design for all scheduling algorithms:
     - A process is defined by `struct Process`. The member variables are:
       - name: the name of the process
       - pid: the actual process id assigned by the kernel.
       - ready time/execution time: Information given in the input. Critical for scheduling decision.
       - remaining time: the remaining execution time of the process. Used in RR and PSJF.
       - index: the index of the process in the sorted array according to ready time. Used for quick access and making process management convenient.
     - After reading the input into an array of `struct Process`, I sort the processes by their ready time. This is needed regardless of the choice of the scheduling algorithms.
     - Run a for loop simulating the actual time flow. Using the variable `current_time` to indicate the time passed from the beginning.
     - When I set a process running, I set its priority to the highest possible value (in this case, 99), so the real scheduler in the kernel would execute the process.
     - When process A ***preempts*** process B, I set the priority of process A to the lowest possible value (in this case, 1), and set process B running (by setting its priority to the highest value). This way, the real scheduler in the kernel would execute process B instead of process A (hopefully).
   - FIFO:

     Since the array of processes is sorted, at every time unit, I simply check two things:

     1. If there are no process running at this point, check if the next process has arrived (`ready_time` $\leq$ `current_time`). If true, run the process.
     2. If there is a running process, check if it has finished its execution. If not, continue the loop; otherwise, remove the process so the next process can be scheduled.
   - RR:

     In the Round-Robin algorithm, I use a queue to maintain the flow of the processes. The queue is defined in the `queue.c` and `queue.h` files. In each time unit, I check the following things:

     1. Check if there are any processes ready at this moment. If yes, append all of them to the queue.
     2. If there is a running process, check if it has used up its time slice (quantum). If yes, check if it has finished its exeuction. If yes, simply remove the process; otherwise, preempt the current process with the next process in line. After setting the being preempted, move the process to the end of the queue to wait for the next execution.
     3. If there is no process running and the queue is not empty, run the process at the beginning of the queue and refresh the time slice (quantum) for the newly run process.
   - SJF:

In the Shortest Job First algorithm, I use a priority queue (min heap) to manage the processes. The heap is defined in the `heap.c` and `heap.h` files. In each time unit, I check the following things:

1. Check if there are any processes ready at this moment. If yes, insert all of them into the heap.
2. Iif there is process waiting in the heap, check if any process is currently running. If yes, continue (SJF is non-preemptive, we cannot preempt the running process); otherwise, run the process with the shortest execution time in the heap and remove it from the heap.
3. If a process is running, check if it has finished execution. If yes, remove the process.

- PSJF:

  In the Preemptive Shortest Job First algorithm, I use a priority queue identical to the one used in SJF. In each time unit, I check the following things:

  1. If there is a process running and it has finished execution, remove the process from execution and from the heap.
  2. If there are any processes ready at this moment, insert all of them to the heap.
  3. If the heap is non-empty, check if there is any process running. If not, run the process in the heap that has the shortest remaining time; otherwise, compare the remaining time of the current process and the one in the heap. If the current process has less remaining time, continue; otherwise preempt the current process with the one in the heap.

2. Kernel Version

   In this project, the kernel version is 4.14.25 x86_64. And the operating system is Ubuntu 16.04 server.

3. Performance Analysis

In the following chart, the numbers indicate how much longer a process takes to finish execution in this project compared to theoretical scheduling performance.

	Error
FIFO_1	1.68%
FIFO_2	3.6%
FIFO_3	0.53%
FIFO_4	-1.12%
FIFO_5	-0.57%
PSJF_1	0.18%
PSJF_2	1.62%
PSJF_3	2.16%
PSJF_4	0.29%
PSJF_5	6%
RR_1	2.92%

RR_2	10%
RR_3	0.38%
RR_4	0.75%
RR_5	1.23%
SJF_1	1.84%
SJF_2	2.29%
SJF_3	4.08%
SJF_4	7.26%
SJF_5	0.32%

In most of the cases, my scheduler takes longer than the theoretical expectation. I think this is reasonable since my scheduler is, after all, a user process that is scheduled by the actual scheduler in the kernel. The execution of child processes are also at the mercy of the real scheduler. Also, the scheduler in this project involves forking and managing child processes, which add more complexity to my scheduler than the theoretical implementation.

Another observation is that the performance is highly dependent on the status of the system and the kernel scheduler. I have repeated this experiment several times, and found that the error can be any number between -5% to 30%. A possible reason that it may take negative values is that, when many child processes are spawned, it is difficult to manage them all. Although the my scheduler sets the priority of the process that should be executing to a high value and set the priority of all other processes to a low value, it is still likely the kernel scheduler executes the process with low priority, which should not be executed. So the result is hard to reproduce, since I am not able to controll the kernel scheduler.