EI331 Course Project

By: Wang Haoxuan

Instructor: Wu ChenTao

November 12, 2018

I. PROJECT 4: SCHEDULING ALGORITHMS

This Project requires us to implement the scheduling algorithms we have learned in class. The outframe of the algorithms are given and details(an *add*() function and a *schedule*() function) need to be implemented by us.

A. First-come, first-served

FCFS is a simple and basic scheduling method. It deals with tasks in the order they arrive. The earlier tasks arrive, the earilier they are dealt with. The code framework guaranteed a data structure of a linked list, with a struct as each node. The task's name, priority and burst time is recorded in the node. All tasks arrive at the same time, thus we do not have to consider the preemptive conditions. Code is shown in $schedule\_fcfs.c$, and the result is as follow:

```
jerrywang@ubuntu:~/Documents/EI338/ch5/project/posix$ make fcfs
gcc -Wall -c schedule_fcfs.c
gcc -Wall -o fcfs driver.o schedule_fcfs.o list.o CPU.o
jerrywang@ubuntu:~/Documents/EI338/ch5/project/posix$ ./fcfs schedule.txt
Running task = [T1] [4] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T7] [3] [30] for 30 units.
Running task = [T8] [10] [25] for 25 units.
```

FIG. 1: The result for FCFS

Additionally, the actual order in the linked list is that task 8 is at the head. To make the process more accurate, I reversed the list and made task 1 at the head.

B. Shortest-job-first

SJF requires the task with the shortest bursting time to be done first. The immediate idea is to sort the linked list by its burst time. But this process is too complicated and it might be even more time-consuming. Thus, I am encouraged by the simplest idea of going through the linked list again and again and find the shortest burst time in each iteration. Since we can delete a node after each iteation, the whole process is not as slow as it may seem, and its code implementation is quite simple. The program is shown in $schedule\_sjf.c$, and the result is as follows:

```
jerrywang@ubuntu:~/Documents/EI338/ch5/project/posix$ make sjf
gcc -Wall -c schedule_sjf.c
gcc -Wall -o sjf driver.o schedule_sjf.o list.o CPU.o
jerrywang@ubuntu:~/Documents/EI338/ch5/project/posix$ ./sjf schedule.txt
Running task = [T6] [1] [10] for 10 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T8] [10] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.
```

FIG. 2: The result for SJF

C. Priority scheduling

Priority scheduling takes another attribute of the tasks into consideration, which is priority. Each task is assigned with a priority and the larger it is, the higher the priority. The idea of this algorithm is quite the same as SJF, except the parameter needing to be sorted changes from burst time to priority, and we consider the largest number first. Code is similar and we show it in *schedule_priority.c*. Result is as follows:

```
jerrywang@ubuntu:~/Documents/EI338/ch5/project/posix$ make priority
gcc -Wall -c schedule_priority.c
gcc -Wall -o priority driver.o schedule_priority.o list.o CPU.o
jerrywang@ubuntu:~/Documents/EI338/ch5/project/posix$ ./priority schedule.txt
Running task = [T8] [10] [25] for 25 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.
Running task = [T6] [1] [10] for 10 units.
```

FIG. 3: The result for priority scheduling

D. Round-Robin scheduling

RR scheduling runs the tasks in order like the FCFS, but each task is only assigned a limited quantum of time to run and the tasks are ran over and over again until they have all finished. For tasks that can't run a quantum time, only the remainder of their CPU burst time is cost. Two *while* iterations are used in my code to make the process go on till the end. Similar to FCFS, a *reverse*() function is called to make the process more natural. Code is shown in *schedule_rr.c* and result is as follows:

```
jerrywang@ubuntu:~/Documents/EI338/ch5/project/posix$ make rr
gcc -Wall -c schedule_rr.c
gcc -Wall -o rr driver.o schedule_rr.o list.o CPU.o
jerrywang@ubuntu:~/Documents/EI338/ch5/project/posix$ ./rr schedule.txt
Running task = [T1] [4] [10] for 10 units.
Running task = [T2] [3] [15] for 10 units.
Running task = [T3] [3] [15] for 10 units.
Running task = [T4] [5] [5] for 10 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T7] [3] [20] for 10 units.
Running task = [T8] [10] [15] for 10 units.
Running task = [T1] [4] [10] for 10 units.
Running task = [T2] [3] [5] for 10 units.
Running task = [T3] [3] [5] for 10 units.
Running task = [T4] [5] [5] for 5 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T7] [3] [10] for 10 units.
Running task = [T8] [10] [5] for 10 units.
Running task = [T2] [3] [5] for 5 units.
Running task = [T3] [3] [5] for 5 units.
Running task = [T7] [3] [10] for 10 units.
Running task = [T8] [10] [5] for 5 units.
```

FIG. 4: The result for RR

E. Priority scheduling with round-robin

This method schedules tasks in the order of their priority and uses round-robin to deal with tasks with the same priority. Though simple in explaining, its implementation needs some tricks. I introduced a variable

single to indicate whether there are tasks with the same priority in the current state. If there is not, we deal it in the simple form of priority scheduling: Get the task, output its information, and delete it. If there are tasks with the same priority, I use a new linked list to store these tasks and implement it in the RR method. During the RR implementation, information is printed and tasks are deleted from the original linked list sychronously. The code is shown in *schedule_priority_rr.c*, and result is as follows:

```
jerrywang@ubuntu:~/Documents/EI338/ch5/project/posix$ make priority_rr
gcc -Wall   -c -o schedule_priority_rr.o schedule_priority_rr.c
gcc -Wall -o priority_rr driver.o schedule_priority_rr.o list.o CPU.o
jerrywang@ubuntu:~/Documents/EI338/ch5/project/posix$ ./priority_rr schedule.txt
Running task = [T8] [10] [25] for 25 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T4] [5] [5] for 5 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T7] [3] [20] for 10 units.
Running task = [T3] [3] [15] for 10 units.
Running task = [T2] [3] [15] for 10 units.
Running task = [T7] [3] [10] for 10 units.
Running task = [T3] [3] [5] for 5 units.
Running task = [T2] [3] [5] for 5 units.
Running task = [T6] [1] [10] for 10 units.
```

FIG. 5: The result for priority scheduling with round-robin

F. Further Challenges: AtomicInteger

This class in Java is simple to use, we only have to declare a variable of type AtomicInteger and use it in the program. It would not cause any type of race condition.

G. Further Challenges: Calculation

The scheduling task is shown in the table below:

Task name	Priority	Burst Time
T1	4	20
T2	3	25
T3	3	25
T4	5	15
T5	5	20
T6	1	10
T7	3	30
T8	10	25

TABLE I: The scheduling task table

All the calcualtion below has a unit of millisecond.

1. FCFS

Average turnaround time = $\frac{20+45+70+85+105+115+145+170}{8} = 94.375$.

Average waiting time = $\frac{0+20+45+70+85+105+115+145}{8} = 73.125$.

Average response time = $\frac{0+20+45+70+85+105+115+145}{8} = 73.125$.

2. SJF

Average turnaround time = $\frac{10+25+45+65+90+115+140+170}{8} = 82.5$.

Average waiting time = $\frac{0+10+25+45+65+90+115+140}{8} = 61.25$.

Average response time = $\frac{0+10+25+45+65+90+115+140}{8} = 61.25$.

3. Priority Scheduling

Average turnaround time = $\frac{25+40+60+80+105+130+160+170}{8} = 96.25$.

Average waiting time = $\frac{0+25+40+60+80+105+130+160}{8} = 75$.

Average response time = $\frac{0+25+40+60+80+105+130+160}{8} = 75$.

4. RR

Average turnaround time = $\frac{90+150+155+115+125+60+165+170}{8} = 128.75$.

Average waiting time = $\frac{(0+70)+(10+70+45)+(20+70+40)+(30+70)+(40+65)+(50)+(60+55+20)+(70+55+20)}{8} = 107.5$.

Average response time = = $\frac{0+10+20+30+40+50+60+70}{8} = 35$.

5. Priority scheduling + RR

Average turnaround time = $\frac{80+145+150+50+60+170+160+25}{8} = 105$.

Average waiting time = $\frac{(60)+(80+20+20)+(90+20+15)+(25+10)+(35+5)+(160)+(100+20+10)+(0)}{8} = 83.75$.

Average response time = $\frac{60+80+90+25+35+160+100+0}{8} = 68.75$.