P3: Scheduling Algorithms

Note: the OSC book says your solution for this project can be written in C or Java. For our class however, it must be in C.

This project is at the end of Chapter-5, under "Programming Projects" in our textbook.

Scheduling Algorithms

This project involves implementing several different process scheduling algorithms. The scheduler will be assigned a predefined set of tasks and will schedule the tasks based on the selected scheduling algorithm. Each task is assigned a priority and CPU burst. The following scheduling algorithms will be implemented:

- First-come, first-served (FCFS), which schedules tasks in the order in which they request the CPU.
- Shortest-job-first (SJF), which schedules tasks in order of the length of the tasks' next CPU burst.
- Priority scheduling, which schedules tasks based on priority.
- Round-robin (RR) scheduling, where each task is run for a time quantum (or for the remainder of its CPU burst).
- Priority with round-robin, which schedules tasks in order of priority and uses round-robin scheduling for tasks with equal priority.

Priorities range from 1 to 10, where a higher numeric value indicates a higher relative priority. For round-robin scheduling, the length of a time quantum is 10 milliseconds.

I. Implementation

The implementation of this project should be completed in C. Program files supporting C are provided in the source code download for the text. (See http://www.os-book.com

Links to an external site.

) These supporting files read in the schedule of tasks, insert the tasks into a list, and invoke the scheduler. You can find the source code on GitHub at https://github.com/greggagne/osc10e/tree/master/ch5/project/posix

Links to an external site.

as well.

The schedule of tasks has the form [***task name***] [***priority***] [***CPU burst***], with the following example format:

```
T1, 4, 20
T2, 2, 25
T3, 3, 25
T4, 3, 15
T5, 10, 10
```

Thus, task T1 has priority 4 and a CPU burst of 20 milliseconds, and so forth. It is assumed that **all tasks arrive at the same time**, so your scheduler algorithms do not have to support higher-priority processes preempting processes with lower priorities. In addition, tasks do not have to be placed into a queue or list in any particular order.

There are a few different strategies for organizing the list of tasks, as first presented in Section 5.1.2. One approach is to place all tasks in a single unordered list, where the strategy for task selection depends on the scheduling algorithm. For example, SJF scheduling would search the list to find the task with the shortest next CPU burst. Alternatively, a list could be ordered according to scheduling criteria (that is, by priority). One other strategy involves having a separate queue for each unique priority, as shown in Figure 5.7.

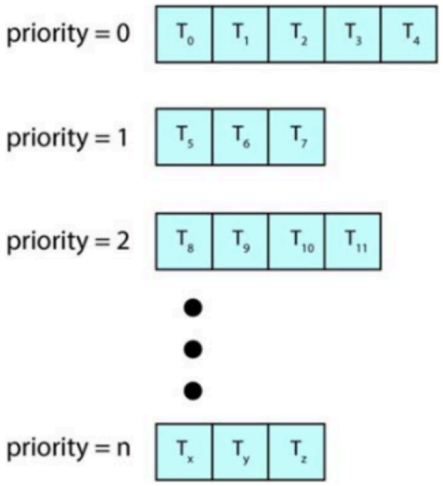

Figure 5.7 Separate queues for each priority.

These approaches are briefly discussed in Section 5.3.6. It is also worth highlighting that we are using the terms ***list*** and ***queue*** somewhat interchangeably. However, a queue has very specific FIFO functionality, whereas a list does not have such strict insertion and deletion requirements. You are likely to find the functionality of a general list to be more suitable when completing this project.

II. C Implementation Details

The file *driver.c* reads in the schedule of tasks, inserts each task into a linked list, and invokes the process scheduler by calling the *schedule()* function. The schedule() function executes each task according to the specified scheduling algorithm. Tasks selected for execution on the CPU are determined by the *pickNextTask()* function and are executed by invoking the *run()* function defined in the *CPU.c* file. A *Makefile* is used to determine the specific scheduling algorithm that will be invoked by driver. For example, to build the FCFS scheduler, we would enter

```
make fcfs
```

and would execute the scheduler (using the schedule of tasks *schedule.txt*) as follows:

```
./fcfs schedule.txt
```

Refer to the *README* file in the source code download for further details. Before proceeding, be sure to familiarize yourself with the source code provided as well as the *Makefile*.

Expected Output

./fcfs schedule.txt

For FCFS, you can simply choose first Task in the wait. Assuming *schedule.txt* as below:

```
T1, 4, 20
T2, 3, 25
T3, 3, 25
T4, 5, 15
T5, 5, 20
T6, 1, 10
T7, 3, 30
T8, 10, 25
```

The output will be

```
Running task = [T1] [4] [20] for 20 units.
        Time is now: 20
Running task = [T2] [3] [25] for 25 units.
        Time is now: 45
Running task = [T3] [3] [25] for 25 units.
        Time is now: 70
Running task = [T4] [5] [15] for 15 units.
        Time is now: 85
Running task = [T5] [5] [20] for 20 units.
        Time is now: 105
Running task = [T6] [1] [10] for 10 units.
        Time is now: 115
Running task = [T7] [3] [30] for 30 units.
        Time is now: 145
Running task = [T8] [10] [25] for 25 units.
        Time is now: 170
```

FCFS does not care about priorities AND all tasks arrive at the same time, so there are 8! possible combinations.

To simplify grading and to be able to easily compare outputs, when multiple tasks can be chosen, the scheduler should choose the task that comes first in dictionary (lexicographical order). You can use the following code or something similar:

```
bool comesBefore(char *a, char *b) { return strcmp(a, b) < 0; }

// based on traverse from list.c
// finds the task whose name comes first in dictionary
Task *pickNextTask() {
  // if list is empty, nothing to do
  if (!g_head)
    return NULL;

  struct node *temp;
  temp = g_head;
  Task *best_sofar = temp->task;

  while (temp != NULL) {
    if (comesBefore(temp->task->name, best_sofar->name))
      best_sofar = temp->task;
    temp = temp->next;
  }
  // delete the node from list, Task will get deleted later
  delete (&g_head, best_sofar);
  return best_sofar;
}
```

The above code would be sufficient for FCFS, but when using priority scheduling, you'd need to find the task that has name that is lexicographical first among all the tasks that have the highest priority. In the above example T8 will be chosen as it has the priority 10. Next, T4 will be chosen. Both T4 and T5 have priority 5, but T4 comes first in lexicographical order.

Bonus

2 points. For each of the scheduling algorithms, output the CPU utilization assuming that the dispatcher takes 1 units of time. CPU Utilization is the percentage time the CPU is idle. For example, for FCFS, if there is dispatcher time, the tasks will finish at 170. With dispatcher time, the tasks will finish at 177. CPU utilization is 170/177 = 96.04

```
CPU Utilization: 96.04%
```

3 points. For each of the scheduling algorithms, output a table of information like the one below. (**TAT**: Turn Around time, **WT**: Wait Time, **RT**: Response Time)

```
...| T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
TAT| 2  | 1  | 2  | 1  | 2  | 1  | 2  |  1 |
WT | 2  | 1  | 2  | 1  | 2  | 1  | 2  |  1 |
RT | 2  | 1  | 2  | 1  | 2  | 1  | 2  |  1 |
```

1. **You can only complete the bonus parts, if you have completed the base assignment**
2. The output for the Bonus part should come after the base assignment output. The base assignment output should not change in any way because you are completing the bonus.

What to Submit

Upload **schedule.zip** with all the project files including:

1. output.txt - showing your program compile and run for all different schedulers on schedule.txt file.
2. schedule_fcfs.c
3. schedule_sjf.c
4. schedule_priority.c
5. schedule_rr.c
6. schedule_priority_rr.c
7. self-assessment.pdf (Implementing each algorithm is worth 5 points. Indicate your expected grade and the breakdown)

In most cases, you should not need to make any changes to project files (such as driver.c, task.c, CPU.c, etc) but they should be included in **schedule.zip**, so we can easily unzip and run your project. If you do make any changes to project files, try to make minimal changes. For example, do not implement complex priority_queue structures. The goal of the project is understanding and emulating the scheduler. You do not have to make it super efficient.