EI338 Computer Systems Engineering

Project 4 Scheduling Algorithms

November 29, 2019

Zhihui Xie, 517030910356

The environment used in this project is **Deepin 15.11**, the latest version of an open source operating system based on Debian's stable branch. The kernel version is **Linux version 4.15.0**.

Exercise

In this exercise, we will implement 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 schedulestasksintheorderinwhich 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.

Basic Structure

To organize the list of tasks, our approach is to place all tasks in a single unordered list, where the strategy for task selection depends on the scheduling algorithm.

There are mainly two functions needed implemented:

- *add()*, which realizes how to add a task into the list. The implementation is basically the same among these algorithms.
- *schedule()*, which will invoke the scheduler and schedule the tasks in some order depending on the type of schedule algorithm.

Implementation

add()

For algorithms except for **Priority with round-robin**, we only need to call the funcion *insert()* to add the task into the link list.

```
static struct node *head = NULL;
static int t_num = 0;

void add(char *name, int priority, int burst) {
    Task *t = malloc(sizeof(Task));
```

```c
    t->name = malloc(strlen(name) + 1);

    strcpy(t->name, name);
    t->priority = priority;
    t->burst = burst;
    t->tid = t_num;
    ++t_num;

    insert(&head, t);
}
```

Listing 1: add()

But for **Priority with round-robin**, the implementation is slightly differnet. To separate tasks into different priority, we maintain a global array to assign tasks into different link lists.

```c
static int t_num = 0;
static int priority_max = INT_MIN;
static struct node *heads[SIZE];

void add(char *name, int priority, int burst) {
    Task *t = malloc(sizeof(Task));
    t->name = malloc(strlen(name) + 1);

    strcpy(t->name, name);
    t->priority = priority;
    t->burst = burst;
    t->tid = t_num;
    ++t_num;

    // update the maximal priority
    priority_max = max(priority, priority_max);

    insert(&heads[priority], t);
}
```

Listing 2: add() for priority with round-robin

schedule()

All 5 algorithms will be discussed here:

- **First-come, first-served (FCFS)**. Due to the fact that all tasks arrive at the same time, FCFS will simply do nothing but run tasks in orginal order.

```c
void schedule() {
    struct node *tmp;

    while (t_num > 0) {
        tmp = head;

        // find the first-order task
        while (tmp->next != NULL) {
            tmp = tmp->next;
        }

        run(tmp->task, tmp->task->burst);
        delete(&head, tmp->task);
        --t_num;
    }
}
```

Listing 3: schedule() for FCFS

- **Shortest-job-first (SJF)**. SJF will schedule the task with shortest CPU burst first. Hence, for every run we traverse the link list and find the shortest job.

```
void schedule() {
    struct node *tmp;
    Task *shortest_burst_t;
    int shortest_burst;

    while (t_num > 0) {
        tmp = head;
        shortest_burst = INT_MAX;

        // find the task with shortest burst
        while (tmp != NULL) {
            if (tmp->task->burst < shortest_burst) {
                shortest_burst = tmp->task->burst;
                shortest_burst_t = tmp->task;
            }

            tmp = tmp->next;
        }

        run(shortest_burst_t, shortest_burst);
        delete(&head, shortest_burst_t);
        --t_num;
    }
}
```

Listing 4: schedule() for SJF

- **Priority scheduling**. Priority scheduling is the general case of SJF, so the implementation is almost the same but for scheduling tasks with higher priority first.

```
void schedule() {
    struct node *tmp;
    Task *highest_t;
    int highest;

    while (t_num > 0) {
        tmp = head;
        highest = INT_MIN;

        // find the task with highest priority
        while (tmp != NULL) {
            if (tmp->task->priority > highest) {
                highest = tmp->task->priority;
                highest_t = tmp->task;
            }
            tmp = tmp->next;
        }

        run(highest_t, highest);
        delete(&head, highest_t);
        --t_num;
    }
}
```

Listing 5: schedule() for Priority scheduling

- **Round-robin (RR) scheduling**. Every time we let a task run for 10 ms (defined by QUANTUM) or a shorter period of time if its remaining CPU burst is less than 10ms. Once it runs, it will be put in the end of queue, which is realized by finding the next task adjacent to it.

```
void schedule() {
    struct node *tmp = head;
    struct node *nextTmp;
    int run_time;

    // go to front
    while (tmp->next != NULL) {
        tmp = tmp->next;
    }

    while (t_num > 0) {
        // run
        run_time = min(QUANTUM, tmp->task->burst);
        run(tmp->task, run_time);

        // find the next task to run
        nextTmp = head;

        if (tmp == head) {
            while (nextTmp->next != NULL) {
                nextTmp = nextTmp->next;
            }
        }
        else {
            while (nextTmp->next != tmp)
            {
                nextTmp = nextTmp->next;
            }
        }

        // check if the task needs to be deleted
        if (tmp->task->burst == run_time) {
            delete(&head, tmp->task);
            --t_num;
        }
        else {
            tmp->task->burst -= run_time;
        }

        // assign next task
        tmp = nextTmp;
    }
}
```

Listing 6: schedule() for RR

- **Priority with round-robin**. This time, we use several queue instead of one. Since tasks with higher priority will run before those with lower priority, we start from highest priority, run out all the tasks and then go to lower priority. For tasks in the same queue, the schedule method is exactly RR.

```
void schedule() {
    int current_head = priority_max;
    int run_time;
    struct node *tmp, *nextTmp;

    while (t_num > 0) {
        tmp = heads[current_head];

        // go to front
        while (tmp->next != NULL) {
            tmp = tmp->next;
        }

        while (tmp != NULL) {
            // run
            run_time = min(QUANTUM, tmp->task->burst);
            run(tmp->task, run_time);

            // find the next task to run
            nextTmp = heads[current_head];

            if (tmp == heads[current_head]) {
                while (nextTmp->next != NULL) {
                    nextTmp = nextTmp->next;
                }
            }
            else {
                while (nextTmp->next != tmp)
                {
                    nextTmp = nextTmp->next;
                }
            }

            // check if the task needs to be deleted
            if (tmp->task->burst == run_time) {
                delete(&heads[current_head], tmp->task);
                --t_num;
            }
            else {
                tmp->task->burst -= run_time;
            }

            // assign next task
            if (heads[current_head] == NULL) {
                tmp = NULL;
            }
            else {
                tmp = nextTmp;
            }
        }

        // go to lower priority
        while (tmp == NULL)
        {
            --current_head;
            tmp = heads[current_head];
        }
    }
}
```

Listing 7: schedule() for Priority with round-robin

Result

Data for testing is as following:

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

And experimental results for different algorithms look like:

Figure 1: Result 1

Figure 2: Result 2

Figure 3: Result 3