Lab Report of Project 4
Scheduling Algorithm

Name: Li Dongyue Student Number: 516030910502
Computer System Engineering (Undergraduate), Fall 2018

November 18, 2018

1 Project Introduction

This project involves implementing several different process scheduling algorithms. The scheduler will be assigned a predefned 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)**: schedules tasks in the order in which they request the CPU
- **Shortest-job-first (SJF)**: schedules tasks in order of the length of the tasks' next CPU burst
- **Priority Scheduling**: schedules tasks based on priority
- **Round-robin (RR) scheduling**: each task is run for a time quantum (or for the remainder of its CPU burst)
- **Priority with round-robin**: schedules tasks in order of priority and uses round-robin scheduling for tasks with equal priority

To note, 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.

2 Scheduling Algorithm Implementation

To implement each scheduling algorithm, we basically write the following functions in the given source code: `add()`, `schedule()`. In `Schedule()`, we need to implement the algorithm in the `pickNextTask()` and `run()`.

2.1 The `add()` function

Given the link list data structure in the source code, we use the given functions in the link list to add tasks into the list: `insert()`, `delete()`, and `traverse()`, shown below:

```
// add a task to the list
void add(char *name, int priority, int burst)
{
    struct task* newTask = malloc(sizeof(struct task));
    newTask->name = name;
    newTask->priority = priority;
    newTask->burst = burst;
    insert(&head, newTask);
    countTask+=1;
}
```

schedule.c

2.2 The `schedule()` function

We implement the scheduling algorithms as followed:

- **First-come, first-served (FCFS)**: Since all the tasks are coming in at the same time, the FCFS algorithm simply picks one out and run it.

```c
// invoke the scheduler
void schedule()
{
    while(countTask>0)
    {
        run(head->task, head->task->burst);
        delete(&head, head->task);
        countTask--;
    }
}
```

schedule_fcfs.c

- **Shortest-job-first (SJF)**: we find out the shortest job every time by traversing though the list and runs it.

```c
// invoke the scheduler
void schedule()
{
    while(countTask>0)
    {
        Task *ShortestTask = head->task;

        struct node *temp;
        temp = head;

        while (temp != NULL) {
            if (temp->task->burst < ShortestTask->burst)
                ShortestTask = temp->task;
            temp = temp->next;
        }

        run(ShortestTask, ShortestTask->burst);
        delete(&head, ShortestTask);
        countTask--;
    }
}
```

schedule_sjf.c

- **Priority Scheduling**: Similarly implemented as SJF algorithm, we find out the job with highest priority every time by traversing though the list and runs it.

```c
// invoke the scheduler
void schedule()
{
    while(countTask>0)
    {
        Task *nextTask = head->task;

        struct node *temp;
        temp = head;

        while (temp != NULL) {
            if (temp->task->priority > nextTask->priority)
                nextTask = temp->task;
            temp = temp->next;
        }

        run(nextTask, nextTask->burst);
        delete(&head, nextTask);
        countTask--;
    }
}
```

schedule_priority.c

- **Round-robin (RR) scheduling**: We remember the last one we pick while iterating through all the tasks. We choose one at a time and run it for 10 ms unless its remaining time is less than 10 ms. Otherwise we run it for its remaining time.

```c
// invoke the scheduler
void schedule()
{
```

```
    struct node* tmp = head;
    while(countTask>0)
    {
        Task *nextTask = tmp->task;

        run(nextTask, 10);
        nextTask->burst -= min(10, nextTask->burst);
        if (nextTask->burst == 0)
        {
            delete(&head, nextTask);
            countTask--;
        }

        if (tmp->next == NULL)
            tmp = head;
        else tmp = tmp->next;
    }
}
```

schedule_rr.c

- **Priority with round-robin**: Similarly implemented as round-robin algorithm, we choose one different task at one time. However, we have multiple waiting queue with different priority. We first choose task from the high priority queue. When there is no task left in the priority queue, we move to the next priority:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "schedulers.h"
#include "list.h"
#include "task.h"
#include "cpu.h"

#define max(a,b)    (((a) > (b)) ? (a) : (b))
#define min(a,b)    (((a) < (b)) ? (a) : (b))

#define SIZE    100
struct node* headList[11] = {NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL};
int countTask = 0;

// add a task to the list
void add(char *name, int priority, int burst)
{
    struct task* newTask = malloc(sizeof(struct task));
    newTask->name = name;
    newTask->priority = priority;
    newTask->burst = burst;
    insert(&headList[priority], newTask);
    countTask+=1;
}

// invoke the scheduler
void schedule()
{
    int curPriority = 10;
    struct node* tmp = headList[curPriority];
    while(countTask > 0)
    {
        while (tmp == NULL)
        {
            tmp = headList[--curPriority];
        }
        Task *nextTask = tmp->task;


        run(nextTask, 10);
        nextTask->burst -= min(10, nextTask->burst);
        if (nextTask->burst == 0)
        {
            delete(&headList[curPriority], nextTask);
            countTask--;
        }

        if (tmp->next == NULL)
            tmp = headList[curPriority];
```

```
51            else tmp = tmp->next;
52        }
53    }
```

schedule_priority_rr.c

To compile the scheduling task files, we can enter: `make scheduler_name`.

To run it, we can enter: `./scheduler_name schedule.txt`, where `scheduler_name` can be `fcfs, sjf, priority, rr` and `prioritt_rr`.