Project 4: 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.

The implementation of this project may be completed in either C or Java, and the program files supporting both of these languages are provided in the source code download for the textbook. These supporting files read in the schedule of tasks, insert the tasks into a list, and invoke the scheduler.

The schedule of tasks has the form ***[task name] [priority] [CPU burst]***.

There are a few different strategies for organizing the list of tasks. One approach is to place all tasks in a single unordered list, where the strategy for task selection depends on the scheduling algorithm. You are likely to find the functionality of a general list to be more suitable when completing this project.

Further Challenges: two additional challenges are presented for this project:

- Each task provided to the scheduler is assigned a unique task (`tid`). If a scheduler is running in a SMP environment where each CPU is separately running its own scheduler, there is a possible race condition on the variable that is used to assign task identifiers. Fix this race condition using an atomic integer.

  On Linux and macOS systems, the **___sync_fetch_and_add()** function can be used to atomically increment an integer value.

- Calculate the average turnaround time, waiting time and response time for each of the scheduling algorithms.

0.1 FCFS Scheduling Algorithm

Design: My design for this task is shown as follows:

- A new task will be insert in the head of the process list, so the next task to run is in the tail of the process list.
- I use **___sync_fetch_and_add()** function (atomic operation add) to solve the racing conditions. (**Further Challenges**)

- I modified the `task.h` to calculate the average turnaround time, waiting time and response time for each of the scheduling algorithms. (**Further Challenges**)

The code of `task.h` is shown as follows.

```c
/**
 * Representation of a task in the system.
 */

#ifndef TASK_H
#define TASK_H

extern int tid_value;

typedef struct state {
    int arrival;
    int waiting;
    int response;
    int turnaround;
    int last_execute;
} State;

// representation of a task
typedef struct task {
    char *name;
    int tid;
    int priority;
    int burst;
    State state;
} Task;



#endif
```

The main code of this task is shown as follows.

```c
#include "schedulers.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "list.h"

#include "cpu.h"
#include "task.h"

struct node *head = NULL;

int time = 0;

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

```
17
18          t -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
19          strcpy(t -> name, name);
20          t -> tid = __sync_fetch_and_add(&tid_value, 1);
21          t -> priority = priority;
22          t -> burst = burst;
23
24          State *state = &(t -> state);
25          state -> arrival = time;
26          state -> last_execute = time;
27          state -> waiting = 0;
28          state -> response = 0;
29          state -> turnaround = 0;
30
31          insert(&head, t);
32  }
33
34  void schedule() {
35          int task_num = 0;
36          int turnaround_total = 0;
37          int waiting_total = 0;
38          int response_total = 0;
39
40          while(head != NULL) {
41                  struct node *h = head;
42                  while (h -> next != NULL) {
43                          h = h -> next;
44                  }
45
46                  Task *task = h -> task;
47                  run(task, task -> burst);
48                  delete(&head, task);
49
50                  State *state = &(task -> state);
51                  state -> waiting += time - state -> last_execute;
52                  if (state -> last_execute == state -> arrival) {
53                          state -> response = time - state -> last_execute;
54                  }
55                  state -> last_execute = time + task -> burst;
56                  state -> turnaround = time + task -> burst - state -> arrival;
57
58                  task_num ++;
59                  turnaround_total += state -> turnaround;
60                  waiting_total += state -> waiting;
61                  response_total += state -> response;
62
63                  time += task -> burst;
64
65                  free(task -> name);
66                  free(task);
67          }
```

```
        printf("-----------------RESULT---------------------\n");
        printf("Total Time: %d , Total Task: %d \n", time, task_num);
        printf("Waiting Time: %d , Average: %.2lf \n", waiting_total, (double)
            waiting_total / task_num);
        printf("Turnaround Time: %d , Average: %.2lf \n", turnaround_total, (double)
            turnaround_total / task_num);
        printf("Response Time: %d , Average: %.2lf \n", response_total, (double)
            response_total / task_num);
        printf("-----------------RESULT---------------------\n");
}
```

The FCFS scheduler can be executed with the following instructions.

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

The execution result of the program is shown as follow.

```
misaka@ubuntu:~/Documents/project4$ make fcfs
gcc -Wall -c task.c
gcc -Wall -c driver.c
gcc -Wall -c list.c
gcc -Wall -c CPU.c
gcc -Wall -c schedule_fcfs.c
gcc -Wall -o fcfs task.o driver.o schedule_fcfs.o list.o CPU.o
misaka@ubuntu:~/Documents/project4$ ./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.
-----------------RESULT---------------------
Total Time: 170 , Total Task: 8
Waiting Time: 585 , Average: 73.12
Turnaround Time: 755 , Average: 94.38
Response Time: 585 , Average: 73.12
-----------------RESULT---------------------
```

图 1: FCFS Scheduler

0.2 SJF Scheduling Algorithm

Design: My design for this task is shown as follows:

- The program need to find the process with the shortest burst time in the process list. I traverse the process list to find out the target process and execute it.
- **Further Challenges** are solved using the similar method as the first task.

The main code of this task is shown as follows.

```
#include "schedulers.h"

#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>
#include "list.h"

#include "cpu.h"
#include "task.h"

struct node *head = NULL;

int time = 0;

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

        t -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
        strcpy(t -> name, name);
        t -> tid = __sync_fetch_and_add(&tid_value, 1);
        t -> priority = priority;
        t -> burst = burst;

        State *state = &(t -> state);
        state -> arrival = time;
        state -> last_execute = time;
        state -> waiting = 0;
        state -> response = 0;
        state -> turnaround = 0;

        insert(&head, t);
}

void schedule() {
        int task_num = 0;
        int turnaround_total = 0;
        int waiting_total = 0;
        int response_total = 0;

        while(head != NULL) {
                struct node *h = head;
        struct node *i = head;
                while (i != NULL) {
                        if (i -> task -> burst < h -> task -> burst) {
                h = i;
            }
            i = i -> next;
                }

                Task *task = h -> task;
                run(task, task -> burst);
                delete(&head, task);

                State *state = &(task -> state);
```

```
                state -> waiting += time - state -> last_execute;
                if (state -> last_execute == state -> arrival) {
                        state -> response = time - state -> last_execute;
                }
                state -> last_execute = time + task -> burst;
                state -> turnaround = time + task -> burst - state -> arrival;

                task_num ++;
                turnaround_total += state -> turnaround;
                waiting_total += state -> waiting;
                response_total += state -> response;

                time += task -> burst;

                free(task -> name);
                free(task);
        }
        printf("-----------------RESULT--------------------\n");
        printf("Total Time: %d , Total Task: %d \n", time, task_num);
        printf("Waiting Time: %d , Average: %.2lf \n", waiting_total, (double)
            waiting_total / task_num);
        printf("Turnaround Time: %d , Average: %.2lf \n", turnaround_total, (double)
            turnaround_total / task_num);
        printf("Response Time: %d , Average: %.2lf \n", response_total, (double)
            response_total / task_num);
        printf("-----------------RESULT--------------------\n");
}
```

The SJF scheduler can be executed with the following instructions.

```
make sjf
./sjf schedule.txt
```

The execution result of the program is shown as follow.

```
misaka@ubuntu:~/Documents/project4$ make sjf
gcc -Wall -c schedule_sjf.c
gcc -Wall -o sjf task.o driver.o schedule_sjf.o list.o CPU.o
misaka@ubuntu:~/Documents/project4$ ./sjf schedule.txt
Running task = [T6] [1] [10] for 10 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 = [T8] [10] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.
----------------RESULT--------------------
Total Time: 170 , Total Task: 8
Waiting Time: 490 , Average: 61.25
Turnaround Time: 660 , Average: 82.50
Response Time: 490 , Average: 61.25
----------------RESULT--------------------
```

图 2: SJF Scheduler

0.3 Priority Scheduling Algorithm

Design: My design for this task is shown as follows:

- The program need to find the process with the highest priority in the process list. I traverse the process list to find out the target process and execute it.
- **Further Challenges** are solved using the similar method as the first task.

The main code of this task is shown as follows.

```c
#include "schedulers.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "list.h"

#include "cpu.h"
#include "task.h"

struct node *head = NULL;

int time = 0;

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

        t -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
        strcpy(t -> name, name);
        t -> tid = __sync_fetch_and_add(&tid_value, 1);
        t -> priority = priority;
        t -> burst = burst;
```

```
23
24          State *state = &(t -> state);
25          state -> arrival = time;
26          state -> last_execute = time;
27          state -> waiting = 0;
28          state -> response = 0;
29          state -> turnaround = 0;
30
31          insert(&head, t);
32  }
33
34  void schedule() {
35          int task_num = 0;
36          int turnaround_total = 0;
37          int waiting_total = 0;
38          int response_total = 0;
39
40          while(head != NULL) {
41                  struct node *h = head;
42          struct node *i = head;
43                  while (i != NULL) {
44                          if (i -> task -> priority >= h -> task -> priority) {
45                  h = i;
46              }
47              i = i -> next;
48                  }
49
50                  Task *task = h -> task;
51                  run(task, task -> burst);
52                  delete(&head, task);
53
54                  State *state = &(task -> state);
55                  state -> waiting += time - state -> last_execute;
56                  if (state -> last_execute == state -> arrival) {
57                          state -> response = time - state -> last_execute;
58                  }
59                  state -> last_execute = time + task -> burst;
60                  state -> turnaround = time + task -> burst - state -> arrival;
61
62                  task_num ++;
63                  turnaround_total += state -> turnaround;
64                  waiting_total += state -> waiting;
65                  response_total += state -> response;
66
67                  time += task -> burst;
68
69                  free(task -> name);
70                  free(task);
71          }
72          printf("-----------------RESULT--------------------\n");
73          printf("Total Time: %d , Total Task: %d \n", time, task_num);
```

```
74          printf("Waiting Time: %d , Average: %.2lf \n", waiting_total, (double)
                waiting_total / task_num);
75          printf("Turnaround Time: %d , Average: %.2lf \n", turnaround_total, (double)
                turnaround_total / task_num);
76          printf("Response Time: %d , Average: %.2lf \n", response_total, (double)
                response_total / task_num);
77          printf("-----------------RESULT--------------------\n");
78  }
```

The priority scheduler can be executed with the following instructions.

```
1  make Priority
2  ./priority schedule.txt
```

The execution result of the program is shown as follow.

```
misaka@ubuntu:~/Documents/project4$ make priority
gcc -Wall -c schedule_priority.c
gcc -Wall -o priority task.o driver.o schedule_priority.o list.o CPU.o
misaka@ubuntu:~/Documents/project4$ ./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.
-----------------RESULT--------------------
Total Time: 170 , Total Task: 8
Waiting Time: 600 , Average: 75.00
Turnaround Time: 770 , Average: 96.25
Response Time: 600 , Average: 75.00
-----------------RESULT--------------------
```

图 3: Priority Scheduler

0.4 RR Scheduling Algorithm

Design: My design for this task is shown as follows:

- The program need to execute the tasks in the task list one by one with a given execution time for each task.
- The current task is executed for a time slice, and if it is not finished, it will be put into the process list waiting for the next execution.
- **Further Challenges** are solved using the similar method as the first task.

The main code of this task is shown as follows.

```
1  #include "schedulers.h"
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include "list.h"
7
```

```c
#include "cpu.h"
#include "task.h"

struct node *head = NULL;

int time = 0;

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

        t -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
        strcpy(t -> name, name);
        t -> tid = __sync_fetch_and_add(&tid_value, 1);
        t -> priority = priority;
        t -> burst = burst;

        State *state = &(t -> state);
        state -> arrival = time;
        state -> last_execute = time;
        state -> waiting = 0;
        state -> response = 0;
        state -> turnaround = 0;

        insert(&head, t);
}

void schedule() {
        int task_num = 0;
        int turnaround_total = 0;
        int waiting_total = 0;
        int response_total = 0;

        while(head != NULL) {
                struct node *h = head;
                while (h -> next != NULL) {
                        h = h -> next;
                }

                Task *task = h -> task;

                if (task -> burst <= QUANTUM) {
                        run(task, task -> burst);
                        delete(&head, task);

                        State *state = &(task -> state);
                        state -> waiting += time - state -> last_execute;
                        if (state -> last_execute == state -> arrival) {
                                state -> response = time - state -> last_execute;
                        }
                        state -> last_execute = time + task -> burst;
                        state -> turnaround = time + task -> burst - state -> arrival;
```

```
                        task_num ++;
                        turnaround_total += state -> turnaround;
                        waiting_total += state -> waiting;
                        response_total += state -> response;

                        time += task -> burst;

                        free(task -> name);
                        free(task);
                } else {
                        run(task, QUANTUM);
                        delete(&head, task);
                        task -> burst = task -> burst - QUANTUM;

                        State *state = &(task -> state);
                        state -> waiting += time - state -> last_execute;
                        if (state -> last_execute == state -> arrival) {
                                state -> response = time - state -> last_execute;
                        }
                        state -> last_execute = time + QUANTUM;

                        time += QUANTUM;
                        insert(&head, task);
                }


        }
        printf("-----------------RESULT--------------------\n");
        printf("Total Time: %d , Total Task: %d \n", time, task_num);
        printf("Waiting Time: %d , Average: %.2lf \n", waiting_total, (double)
            waiting_total / task_num);
        printf("Turnaround Time: %d , Average: %.2lf \n", turnaround_total, (double)
            turnaround_total / task_num);
        printf("Response Time: %d , Average: %.2lf \n", response_total, (double)
            response_total / task_num);
        printf("-----------------RESULT--------------------\n");
}
```

The RR scheduler can be executed with the following instructions.

```
make rr
./rr schedule.txt
```

The execution result of the program is shown as follow.

```
misaka@ubuntu:~/Documents/project4$ make rr
gcc -Wall -c schedule_rr.c
gcc -Wall -o rr task.o driver.o schedule_rr.o list.o CPU.o
misaka@ubuntu:~/Documents/project4$ ./rr schedule.txt
Running task = [T1] [4] [20] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T8] [10] [25] for 10 units.
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 5 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T7] [3] [20] for 10 units.
Running task = [T8] [10] [15] 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.
-----------------RESULT---------------------
Total Time: 170 , Total Task: 8
Waiting Time: 860 , Average: 107.50
Turnaround Time: 1030 , Average: 128.75
Response Time: 280 , Average: 35.00
-----------------RESULT---------------------
```

图 4: RR Scheduler

0.5 Priority-RR Scheduling Algorithm

Design: My design for this task is shown as follows:

- The program need to execute the tasks in the task list in order of priority with a given execution time for each task.
- First, find the process(es) with the highest priority, and then use RR algorithm to execute them.
- The current process is executed for a time slice, and if it is not finished, it will be put into the process list waiting for the next execution.
- **Further Challenges** are solved using the similar method as the first task.

The main code of this task is shown as follows.

```c
#include "schedulers.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "list.h"

#include "cpu.h"
#include "task.h"
```

```
10
11  struct node *head = NULL;
12
13  int time = 0;
14
15  void add(char *name, int priority, int burst) {
16      Task *t = (Task *) malloc (sizeof(Task));
17
18          t -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
19          strcpy(t -> name, name);
20          t -> tid = __sync_fetch_and_add(&tid_value, 1);
21          t -> priority = priority;
22          t -> burst = burst;
23
24          State *state = &(t -> state);
25          state -> arrival = time;
26          state -> last_execute = time;
27          state -> waiting = 0;
28          state -> response = 0;
29          state -> turnaround = 0;
30
31          insert(&head, t);
32  }
33
34  void schedule() {
35          int task_num = 0;
36          int turnaround_total = 0;
37          int waiting_total = 0;
38          int response_total = 0;
39
40          while(head != NULL) {
41                  struct node *h = head;
42          struct node *i = head;
43                  while (i != NULL) {
44                          if (i -> task -> priority >= h -> task -> priority) {
45                  h = i;
46              }
47              i = i -> next;
48                  }
49
50                  Task *task = h -> task;
51
52                  if (task -> burst <= QUANTUM) {
53                          run(task, task -> burst);
54                          delete(&head, task);
55
56                          State *state = &(task -> state);
57                          state -> waiting += time - state -> last_execute;
58                          if (state -> last_execute == state -> arrival) {
59                                  state -> response = time - state -> last_execute;
60                          }
```

```
                        state -> last_execute = time + task -> burst;
                        state -> turnaround = time + task -> burst - state -> arrival;

                        task_num ++;
                        turnaround_total += state -> turnaround;
                        waiting_total += state -> waiting;
                        response_total += state -> response;

                        time += task -> burst;

                        free(task -> name);
                        free(task);
                } else {
                        run(task, QUANTUM);
                        delete(&head, task);
                        task -> burst = task -> burst - QUANTUM;

                        State *state = &(task -> state);
                        state -> waiting += time - state -> last_execute;
                        if (state -> last_execute == state -> arrival) {
                                state -> response = time - state -> last_execute;
                        }
                        state -> last_execute = time + QUANTUM;

                        time += QUANTUM;
                        insert(&head, task);
                }


        }
        printf("-----------------RESULT--------------------\n");
        printf("Total Time: %d , Total Task: %d \n", time, task_num);
        printf("Waiting Time: %d , Average: %.2lf \n", waiting_total, (double)
            waiting_total / task_num);
        printf("Turnaround Time: %d , Average: %.2lf \n", turnaround_total, (double)
            turnaround_total / task_num);
        printf("Response Time: %d , Average: %.2lf \n", response_total, (double)
            response_total / task_num);
        printf("-----------------RESULT--------------------\n");
}
```

The Priority-RR scheduler can be executed with the following instructions.

```
make priority_rr
./priority_rr schedule.txt
```

The execution result of the program is shown as follow.

```
misaka@ubuntu:~/Documents/project4$ make priority_rr
gcc -Wall   -c -o schedule_priority_rr.o schedule_priority_rr.c
gcc -Wall -o priority_rr task.o driver.o schedule_priority_rr.o list.o CPU.o
misaka@ubuntu:~/Documents/project4$ ./priority_rr schedule.txt
\Running task = [T8] [10] [25] for 10 units.
Running task = [T8] [10] [15] for 10 units.
Running task = [T8] [10] [5] for 5 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [5] for 5 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T1] [4] [20] for 10 units.
Running task = [T1] [4] [10] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T2] [3] [15] for 10 units.
Running task = [T3] [3] [15] for 10 units.
Running task = [T7] [3] [20] 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 = [T6] [1] [10] for 10 units.
----------------RESULT--------------------
Total Time: 170 , Total Task: 8
Waiting Time: 670 , Average: 83.75
Turnaround Time: 840 , Average: 105.00
Response Time: 550 , Average: 68.75
----------------RESULT--------------------
```

图 5: Priority-RR Scheduler