Schedule This()

Paul D. Camarata

12 August 2018

Abstract

Intelligence is an accident of evolution, and not necessarily an advantage. -Isaac Asimov.

1 Introduction

The purpose of this programming project was to gain a better understanding of how an OS can leverage various scheduling algorithms. Using C, I have provided five different examples. All of the work performed on this project is available on github, https://github.com/paulcamarata/CYBR-570.

2 Part I: Before Coding

Several c and header files have been provided for us. Also, the instructor has given us an additional function, add(). Understanding these is essential to being able to complete the assignment.

2.1 traverse()

The first function I examined was traverse(). Located in list.c, this function was vital for teaching me how to access the different sub components of the list. Understanding that the list begins at head and ends at NULL is the first observation.

```
void traverse(struct node *head) {
    struct node *temp;
    temp = head;

    while (temp != NULL) {
        printf("[%s] [%d] [%d]\n",temp->task->name, temp->task->priority, temp->task->burst
        temp = temp->next;
    }
}
```

2.2 run()

The run() function ends up being very basic. The sole purpose of this function is to show that the correct task has been selected to run. At the end of the day it is just a print statement since we are running as a simulation versus actually executing a task.

```
void run(Task *task, int slice) {
    printf("Running task = [%s] [%d] [%d] for %d units.\n",task->name, task->priority, task-
}
```

2.3 insert()

Insert ends up being a very important function for creating the list. This function is the nuts and bolts for how you create the linked list. It grabs the current location of the list, inserts the new task at the front of it, and shifts the head of the list.

```
void insert(struct node **head, Task *newTask) {
    // add the new task to the list
    struct node *newNode = malloc(sizeof(struct node));

    newNode->task = newTask;
    newNode->next = *head;
    *head = newNode;
}
```

2.4 delete()

The delete function removes a task from the list. It does this by tweaking the linking in the list. If the item is at the start of the list, then it just changes the location of head. If it is anywhere in the middle, then it finds the previous task and sets its 'next' location to be the item after the deleted task. This function is intended to be run when a task has fully completed its burst.

```
void delete(struct node **head, Task *task) {
    struct node *temp;
    struct node *prev;

    temp = *head;
    // special case - beginning of list
    if (strcmp(task->name,temp->task->name) == 0) {
        *head = (*head)->next;
    }
    else {
        // interior or last element in the list
        prev = *head;
        temp = temp->next;
        while (strcmp(task->name,temp->task->name) != 0) {
            prev = temp;
            temp = temp->next;
        }

        prev->next = temp->next;
    }
}
```

2.5 add()

The last function provided to us is add(). This function is called from main() and helps setup the task structure that will be required for the insert() function to work properly.

```
void add(char *name, int priority, int burst) {
    // first create the new task
    Task *newTask = (Task *) malloc(sizeof(Task));

    newTask->name = name;
    newTask->priority = priority;
    newTask->burst = burst;

    // insert the new task into the list of tasks
    insert(&head, newTask);
}
```

3 Part II: My Code

All code written by me is included in the appendices. This section is here to walk through my thoughts on how to write the code and reference some of the things actually done. There were two placeholders provided to me in the sample file, schedule() and pickNextTask().

3.1 schedule()

I interpreted this function to be the ”How” to run a task. There are several ways to set this function up, but I opted to utilize recursion. After talking to another classmate, he explained to me why, in this particular instance, recursion might work, but is not the most optimal way to set this up. I am able to show that it is capable of working however.

There are two different paths with my schedule function. One is used for all programs that run to completion once a task has been chosen. The other method is for hanlding round robin based scheduling. The latter ends up being more complicated.

The goal for all schedule functions is to take the selected task, run it for the duration required, and then either update it and place it back in the queue or delete it from the queue once it is done.

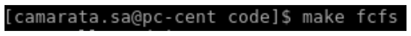
3.2 pickNextTask()

This function, in my opinion, was more of the ”What” to run. Every time schedule() runs, it makes a call to pickNextTask(). This function then provides a task to be run. The goal with this function was to step through the list using a pointer, or two, to make a decision on which task is appropriate depending on the scheduling algorithm desired.

4 Part III: Results

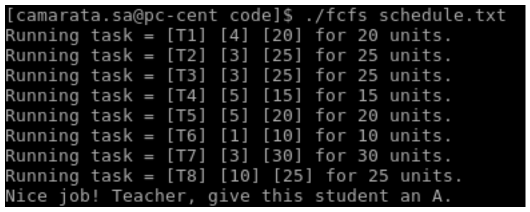
This section is very straight forward. First we compile the correct code, per the make file, and then we execute the desired schedule. All code executes correctly and the programs gracefully exit.

4.1 FCFS



```
[camarata.sa@pc-cent code]$ make fcfs
```

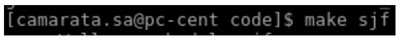
Figure 1: make fcfs



```
[camarata.sa@pc-cent code]$ ./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.
Nice job! Teacher, give this student an A.
```

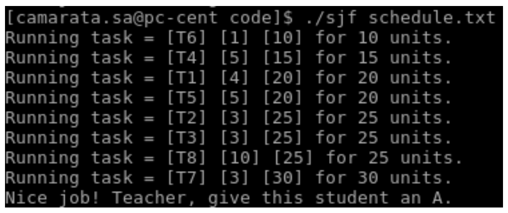
Figure 2: run fcfs

4.2 SJF



```
[camarata.sa@pc-cent code]$ make sjf
```

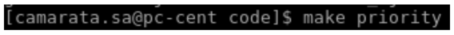
Figure 3: make sjf



```
[camarata.sa@pc-cent code]$ ./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.
Nice job! Teacher, give this student an A.
```

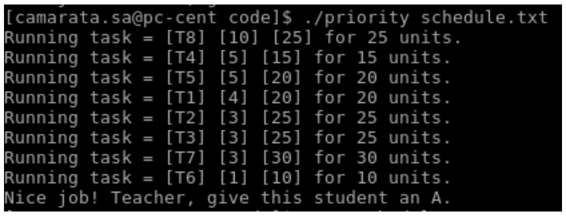
Figure 4: run sjf

4.3 Priority



```
[camarata.sa@pc-cent code]$ make priority
```

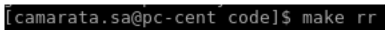
Figure 5: make priority



```
[camarata.sa@pc-cent code]$ ./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.
Nice job! Teacher, give this student an A.
```

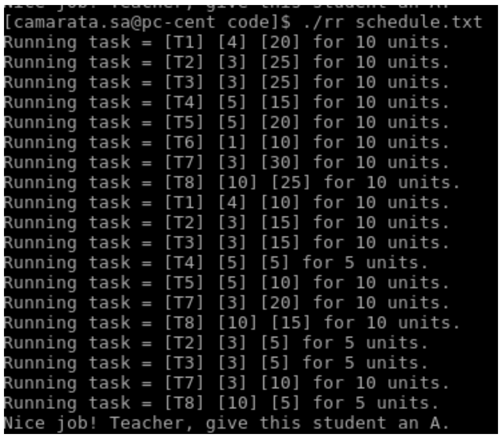
Figure 6: run priority

4.4 Round Robin



```
[camarata.sa@pc-cent code]$ make rr
```

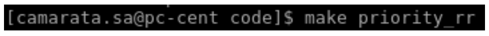
Figure 7: make rr



```
[camarata.sa@pc-cent code]$ ./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.
Nice job! Teacher, give this student an A.
```

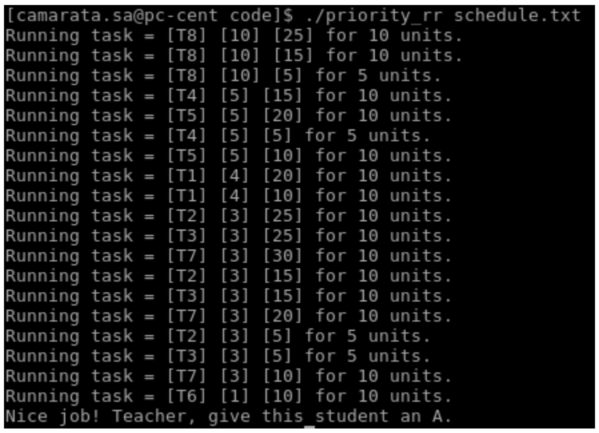
Figure 8: run rr

4.5 Priority Round Robin



```
[camarata.sa@pc-cent code]$ make priority_rr
```

Figure 9: make priority_rr



```
[camarata.sa@pc-cent code]$ ./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.
Nice job! Teacher, give this student an A.
```

Figure 10: run priority_rr

5 Conclusion

In conclusion, I think this was an interesting exploitation project. I would definitely be interested in a project that dives a bit deeper. I felt that with the examples presented in class it was impossible to not be able to complete the steps needed to exploit these applications. The extra credit did require a bit more knowledge about how to manipulate a linux system to get information about how the application is running, but I think the class should be capable of more complex disassembly.

6 Appendix A: FCFS Code

```
void schedule() {

    Task *currentTask = pickNextTask();

    if(!currentTask) {
        exit(0);
    }

    run(currentTask, currentTask->burst);
    delete(&head, currentTask);
    schedule();
}

Task *pickNextTask() {

    struct node *temp;
    temp = head;

    if (!temp){
        printf("Nice job! Teacher, give this student an A.\n");
        exit(0);
    }

    while (temp) {
        if(!temp->next) {
            return temp->task;
        }

        temp = temp->next;
    }

    return 0;
}
```

7 Appendix B: SJF Code

```
void schedule() {

    Task *currentTask = pickNextTask();

    if(!currentTask) {
        exit(0);
    }

    run(currentTask, currentTask->burst);
    delete(&head, currentTask);
    schedule();
}

Task *pickNextTask() {

    struct node *temp;
    temp = head;

    if (!temp){
        printf("Nice job! Teacher, give this student an A.\n");
        exit(0);
    }

    int sjf = temp->task->burst;
    Task *chosenTask = temp->task;

    while (temp) {

        if(temp->task->burst <= sjf) {
            sjf = temp->task->burst;
            chosenTask = temp->task;
        }

        if(!temp->next) {
            return chosenTask;
        }

        temp = temp->next;
    }

    return 0;
}
```

8 Appendix C: Priority Code

```
void schedule() {

    Task *currentTask = pickNextTask();

    if(currentTask == NULL) {
        exit(0);
    }

    run(currentTask, currentTask->burst);
    delete(&head, currentTask);
    schedule();
}

Task *pickNextTask() {

    struct node *temp;
    temp = head;

    if (!temp){
        printf("Nice job! Teacher, give this student an A.\n");
        exit(0);
    }

    int pri = temp->task->priority;
    Task *chosenTask = temp->task;

    while (temp) {
        if(temp->task->priority >= pri) {
            pri = temp->task->priority;
            chosenTask = temp->task;
        }
        if(!temp->next) {
            return chosenTask;
        }
        temp = temp->next;
    }
    return 0;
}
```

9 Appendix D: Round Robin Code

```
void schedule() {

    Task *currentTask = pickNextTask();

    if(!currentTask) {
        exit(0);
    }

    if(currentTask->burst > QUANTUM) {
        run(currentTask, QUANTUM);
        currentTask->burst = currentTask->burst - QUANTUM;
        delete(&head, currentTask);
        insert(&head, currentTask);
    } else {
        run(currentTask, currentTask->burst);
        delete(&head, currentTask);
    }
    schedule();
}

Task *pickNextTask() {

    struct node *temp;
    temp = head;

    if (!temp){
        printf("Nice job! Teacher, give this student an A.\n");
        exit(0);
    }

    while (temp) {
        if(!temp->next) {
            return temp->task;
        }
        temp = temp->next;
    }

    return 0;
}
```

10 Appendix E: Priority Round Robin Code

```
void schedule() {

    Task *currentTask = pickNextTask();

    if(!currentTask) {
        exit(0);
    }
    if(currentTask->burst > QUANTUM) {
        run(currentTask, QUANTUM);
        currentTask->burst = currentTask->burst - QUANTUM;
        delete(&head, currentTask);
        insert(&head, currentTask);
    } else {
        run(currentTask, currentTask->burst);
        delete(&head, currentTask);
    }
    schedule();
}

Task *pickNextTask() {

    struct node *temp;
    temp = head;

    if (!temp){
        printf("Nice job! Teacher, give this student an A.\n");
        exit(0);
    }

    int pri = temp->task->priority;
    Task *chosenTask = temp->task;

    while (temp) {
        if(temp->task->priority >= pri) {
            pri = temp->task->priority;
            chosenTask = temp->task;
        }
        if(!temp->next) {
            return chosenTask;
        }
        temp = temp->next;
    }
    return 0;
}
```

References

[1] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne *Operating System Concepts* John Wiley & Sons Inc. 2018