ECE 220 Honor Project Final Report
Yizhen Lu

INTRODUCTION

This program aims at writing a program to generate optimal schedule based on input information given. Such program can be useful for faculty to assign office hours to TAs while not conflicting with their class schedules, also for registered student organization(RSO) team captains to arrange work shift based on each member's availability, etc.

PROJECT DESCRIPTION

The program would work in the following way:
First, we load the time slots that needs to be filled from a pre-written file. Then, we want to have input from members for their names and their availability (i.e. free time). The scheduler will then match all members to the slots provided and then generate one possible solution. If no possible schedule exists, which means there would always be some time slots that could not match with any worker's availability, the scheduler will return a statement saying no schedule is possible to be generated based on the current algorithm.

DATA STRUCTURES

```
typedef struct timeperiod period;

typedef struct timeperiod{
  int day;
  int start_time;
  int end_time;
  period * next;
}period;

typedef struct worker{
  char name[10];
  period * time;
  int availability;
  int index;
}mem;

typedef struct timeslot{
  period * time;
  int num_member;
  int filled;
  int fit_index[50];
}slot;
```

First, I introduce the linked-list into my array of structs for members and slots. This would allow each member to have unlimited/flexible numbers of available time slots. Struct "*period*" is used to store all values of a time period, namely which day and starting & ending time. It is also used to hold the linked list.
In struct "*mem*", *char name[10]* is used to hold the name of the worker, *period * time* holds the head of the linked list of the worker's available time slots, *int availability* holds the number of shifts the worker is to be assigned, *int index* holds the index value of the worker in this array of mem structs.
In struct "*slot*", *int num_member* holds the value of how many members this time slot wants, *int filled* holds the index of the worker assigned to this time slot, and the *fit_index[50]* holds index value for all workers who would be available over this time period.

```
typedef struct timetable
{
  char day[20];
  char* time[28];
  /* from 8am to 10pm, each array
   * represent the person for this
   * half-hour period
   */
}agenda;
```

Another struct, “timetable” is defined to store all time slots that have been filled in with some workers. The output table is just printing all information in an array of agenda structs. The time array is used to store the name of the member to be assigned in the time period indicated by the array index.

FUNCTIONALITY

In main.c

```
agenda * CreateTable();
```

This function allocates an array of seven structs and initializes all values.

```
void destroy_table(agenda * day);
```

This function frees all allocated space for the array of *agenda* struct;

```
int main();
```

This is the main function, which organizes the flow of the entire program.

In io.c

```
mem * member_input();
slot * timeslot_input();
```

These two functions use basic I/O to load members’ available time period and time slots info. And the information are stored in two arrays, and two pointers are returned.

```
mem * file_member_input(char* file);
slot * file_timeslot_input(char* file);
```

These two functions read in members’ available time period and time slots info using file I/O. And the information are stored in two arrays, and two pointers are returned.

Below are the standard format of files that these two functions would be able to read and load correctly.(left side is the format for members and right side for slots)

```
(number of total members)
N (name)
T (day) (start_time) (end_time)
N (name)
T (day) (start_time) (end_time)
```

```
(number of total slots)
(day) (start_time) (end_time) (num_member)
(day) (start_time) (end_time) (num_member)
(day) (start_time) (end_time) (num_member)
```

For members' availability, the number on the first line indicates the total number of workers we are free to select from. Then for each line, we start with a type identifier, "N" for name and "T" for time.
Similarly, for time slots requirement, the first number indicates the total number of slots to be assigned. And the number after three parameters to describe the time period is the number of workers required for the slot.

```
void insert(mem * mem, period * time);
```

This function is called in mem * file_member_input(char* file), and is used to sort all available time slots in order.

```
void print_mem_list(mem * mem);
void print_slot_list(slot * slot);
```

These two functions print out the data stored in the two arrays of struct and in all the linked list.

```
void destroy_mem_list(mem * ptr);
void destroy_slot_list(slot * ptr);
```

These two functions free all allocated space for the two arrays of struct to prevent memory leaking.

```
void find_match_member(slot * slot, mem * list);
```

This function finds all matched members for every time slot and stores in fit_index with (index number +1) so that we can avoid ambiguity between 0 index and no fit.

```
int check_possible_schedule(slot * slot);
```

This function checks whether it is possible to generate a schedule by checking if each time slot has some matched members. Return 0 is impossible to generate a schedule, 1 otherwise

```
int GenerateSchedule(slot * slot, int * nth, mem * member);
```

This function uses back-tracking recursion to match each time slot with a member based on the fit_index we found earlier, namely it works based on time slot restrictions.

```
void assign_table(agenda * day, slot * slot, mem * list);
```

This function puts the names of workers assigned for each slot into corresponding blocks in the day[7] array, which stores info for our output table.

```
void file_print_readfile(slot * time, mem * list);
```

This function prints out the generated schedule in a format that can be read by the program, i.e. the same as the format required by function *mem * file_member_input(char* file)*.

In print.c

```
void print_table(agenda *array);
```

This is the major function in the source file, which is responsible for displaying the generated time table into terminal using basic output (i.e. “printf”). In the content of the nested for loop, I use switch functions to determine which thing should be displayed.

For the first row and column, the program should print days and times, so to tell them apart from printing the content of time table, I use switch functions and create a case for row and column index equal to 0.

To make the table more clear and neat, I used space padding and printing with different colors. They are all in the content of loops within function *print_table()* like the codes shown below:

```
printf("\033[0;34m");//set printing color to be blue
printf("%10s", array[k-1].time[j-1]);
printf("\033[0m");//free setting
```

The first line sets the printing color into blue and the last line switch the color back to default. The second line prints the content with left-padding of ten characters’ space.

```
void file_print_table(agenda *array);
```

This function stores the generated table in file “output_table.txt”, with exactly the same format as the above function would print out.

```
int hour(int n);
int min(int n);
```

These two helper functions changed the index back to the time domain to indicate the time period each box represents. They contributes a lot on printing the time on the leftmost column of the time table. Index 0 represents time 8:00 am and index 27 represents 9:30pm. So the algorithm is developed as shown below.

They can be treated as the reverse operation of the *int start_time_to_index(int n)* and *int end_time_to_index(int n)* functions.

```
void line();
void fence();
```

These two helper functions builds up the basic structure of the table. The *void line()* function feed a new line of intersections between each row, while the *void fence()* function prints a horizontal stick to set apart columns.

INTERFACE DISPLAY

To better organize the program and make it easier and more clear to use, I set up some while loops in main function and have more instructions and options to make users more comfortable. Below are the screenshots of the interface display.

```
[yizhenl3@linux-a2 ECE220-Honor-Project]$ ./honor
You want to:
1: generate schedule; 2: find common meeting time
1
Please choose the way to input data :
1: stdin; 2: file input
2
WELCOME TO THE SCHEDULE GENERATOR!Please make sure the file is in correct format(similar to the example below)
2
N Alex
T 0 1300 1500
T 2 0800 0930
N Beta
T 3 1530 1640
3
0 1200 1400 1
2 0824 1322 3
5 0930 1030 1
Please enter the name of the file you want to load member data:
member.txt
Loading member.txt ...
Loading complete.
Please enter the name of the file you want to load time slot data
:timeslot.txt
Loading timeslot.txt ...
Loading complete.
```

At the beginning is the input part, it first asks the user to choose either basic I/O or file I/O, for simplicity here I choose to show the program behavior for file I/O options.

```
Enter the operation you want to conduct:
0: Halt the program;
1: Exhibit loaded input;
2: Generate Schedule(stored in "output_table.txt");
3: Print generated table;
```

After loading the input, the program asks for further instructions for next operation. One hidden principle is to perform operation #2 before #3, because the generated table is being produced during operation #2. But if you do choose #3 before #2, the program will give you feedback to conduct #2 first, as shown below.

```
Enter the operation you want to conduct:
0: Halt the program;
1: Exhibit loaded input;
2: Generate Schedule(stored in "output_table.txt");
3: Print generated table;
3
Please generate table first.
```

TESTING

To test whether the codes work as expected. I wrote two files of member's available time period and time slots required. Purposely I arrange the data in a way that only one schedule is possible. Then I run the program to see whether the output matches this only solution.

"member.txt" & "timeslot.txt":

```
5
N Aaron
T 0 1300 1500
T 3 1200 1730
T 5 1300 1500
N Beta
T 2 1000 1500
T 6 1400 1500
T 3 1200 1530
T 1 0800 1030
N Carlin
T 3 1400 1500
N Delta
T 1 0900 2000
T 2 1000 1500
N Elf
T 5 1100 1400
T 4 1100 1400
```

```
5
0 1300 1400 1
2 1000 1500 1
6 1400 1430 1
5 1300 1400 1
3 1400 1500 1
```

```
Summarize input info:
The following is the member list:
Aaron
Day0: 1300-1500
Day3: 1200-1730
Day5: 1300-1500
Beta
Day1: 800-1030
Day2: 1000-1500
Day3: 1200-1530
Day6: 1400-1500
Carlin
Day3: 1400-1500
Delta
Day1: 900-2000
Day2: 1000-1500
Elf
Day4: 1100-1400
Day5: 1100-1400
The following is the slot list:
Day0: 1300-1400
Day2: 1000-1500
Day6: 1400-1430
Day5: 1300-1400
Day3: 1400-1500
```

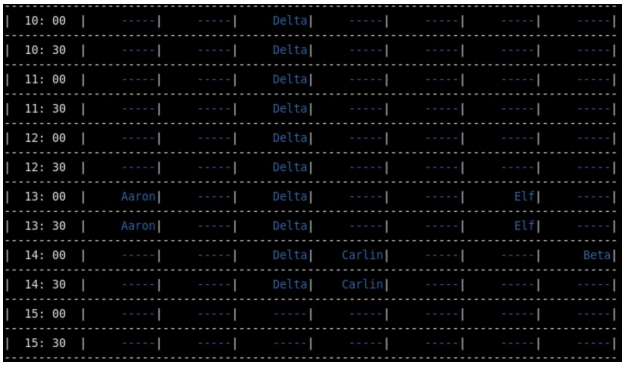
When we run the program with the input files above, these are the output of the print_xxx_list functions, which indicates that the file input functions and print functions are functioning properly. As we can compare the loaded input with the given input, we see that the order of the time slots are being modified so that they are now in order.

Then the find_match_member() functions also work nicely to match every member with the time slots they fit in.

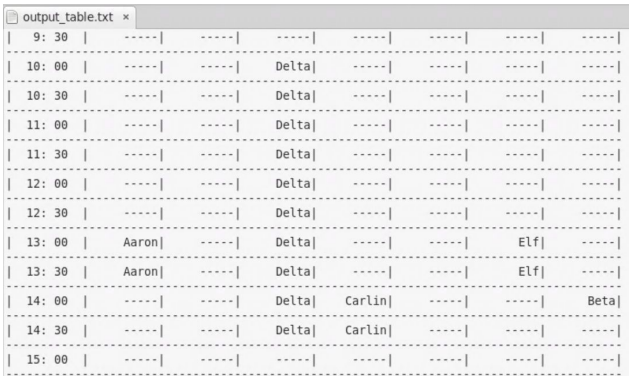
On the right side is what would be stored in the fit_index array of each struct within the array. This gives a clear view of the mapping between members and desired slots.

```
slot[0]: day 0, 1300-1400
matched member:
Aaron
slot[1]: day 2, 1000-1500
matched member:
Beta
Delta
slot[2]: day 6, 1400-1430
matched member:
Beta
slot[3]: day 5, 1300-1400
matched member:
Aaron
Elf
slot[4]: day 3, 1400-1500
matched member:
Aaron
Beta
Carlin
```

Finally, we would assign the members to each slot using back-tracking recursion. And then the generated table would be printed in the terminal and stored in a file, which is exactly the only possible arrangement for the given input.(Shown below) This means the codes are functioning properly, at least for this given testcase.



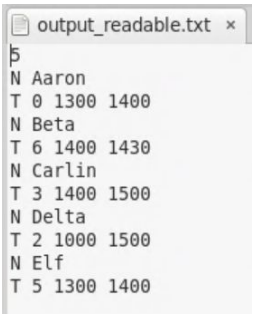
10: 00			Delta			
10: 30			Delta			
11: 00			Delta			
11: 30			Delta			
12: 00			Delta			
12: 30			Delta			
13: 00	Aaron		Delta		Elf	
13: 30	Aaron		Delta		Elf	
14: 00			Delta	Carlin		Beta
14: 30			Delta	Carlin		
15: 00						
15: 30						



output_table.txt

9: 30						
10: 00			Delta			
10: 30			Delta			
11: 00			Delta			
11: 30			Delta			
12: 00			Delta			
12: 30			Delta			
13: 00	Aaron		Delta		Elf	
13: 30	Aaron		Delta		Elf	
14: 00			Delta	Carlin		Beta
14: 30			Delta	Carlin		
15: 00						

More than that, as I add one more functionality to allow the program to store the output result in the format that can be read by the program. If I run the program again with “output_readable.txt” and “timeslot.txt”, I would get exactly the same result as the original input case.



output_readable.txt

```
5
N Aaron
T 0 1300 1400
N Beta
T 6 1400 1430
N Carlin
T 3 1400 1500
N Delta
T 2 1000 1500
N Elf
T 5 1300 1400
```

CONCLUSION

The project is now capable of generating reasonable schedule for office hours and deciding work shifts.
In this project, I think I accomplished what I expected, as written in the proposal, in the first place. I used C language, and applied a lot of approaches and structures taught during lectures, like dynamic memory allocation, array of structs, linked list, back-tracking recursion, etc. Also, I get to learn more about things that are equally important but not that heavily emphasized in class, like file I/O. Such experience provides me the chance to learn more useful functions in C library and understand how to apply them correctly. This would be really helpful for further programming studies.