COMP2432 Group Project Design Documentation

Star Poon

April 4, 2015

Contents

1 Foreword **2**
1.1 Introduction . . . 2
1.2 Scope . . . 2

2 Requirement Analysis **3**
2.1 Input commands . . . 3
2.1.1 Execute the program . . . 3
2.1.2 Add study . . . 3
2.1.3 Add assignment . . . 3
2.1.4 Add project . . . 3
2.1.5 Add gathering . . . 4
2.1.6 Add batch . . . 4
2.1.7 Add schedule . . . 4
2.1.8 End the program . . . 4
2.2 Special requirements . . . 4
2.2.1 Priority scheduler . . . 4
2.2.2 Modulation . . . 4
2.2.3 Timeslot . . . 5

3 Design and Implementation **6**
3.1 Data Structure . . . 6
3.1.1 struct Appointment . . . 6
3.1.2 struct AppointmentList . . . 6
3.1.3 struct User . . . 6
3.2 Algorithm . . . 7
3.2.1 Check appointment conflict . . . 7
3.2.2 First come first served . . . 7
3.2.3 Priority . . . 7
3.2.4 Optimal . . . 7

Chapter 1

Foreword

1.1 Introduction

1.2 Scope

Chapter 2

Requirement Analysis

2.1 Input commands

Input formats follow the examples and make no difference and we simply assume all the input formats are correct. Format checking is not required.

2.1.1 Execute the program

```
./amr [users]
./amr alice bob charlie ...
```

Username are passed from the command line arguments. Number of users can be determined by the number of arguments.

Requirement:

1. Convert the names to the standard format (i.e. first letter capitalized regardless of the input string).
2. Error checking: Number of users are between the range of 3 and 10.
3. Error checking: Duplicated names of users.

2.1.2 Add study

```
addStudy -[caller] [datetime] [duration]
addStudy -adam 2015-04-02 18:30 2.5
```

[caller]	[xxx]	The username of the caller, should be one of the names from command line arguments.
[datetime]	[YYYY-MM-DD hh:mm]	Date and time of the event, YYYY:Year (4 digits), MM:Month (2 digits), DD:Day (2 digits), hh:Hour (2digits) and mm:Minute (2 digits).
[duration]	[n.n]	Duration of the appointment in hours (fixed point of one decimal place).

Requirement:

1. Error checking: Caller's name should be one of the names from command line arguments.

2.1.3 Add assignment

```
addAssignment -[caller] [datetime] [duration]
addAssignment -adam 2015-04-02 18:30 2.5
```

[caller]	[xxx]	The username of the caller, should be one of the names from command line arguments.
[datetime]	[YYYY-MM-DD hh:mm]	Date and time of the event, YYYY:Year (4 digits), MM:Month (2 digits), DD:Day (2 digits), hh:Hour (2digits) and mm:Minute (2 digits).
[duration]	[n.n]	Duration of the appointment in hours (fixed point of one decimal place).

Requirement:

1. Error checking: Caller's name should be one of the names from command line arguments.

2.1.4 Add project

```
addProject -[caller] [datetime] [duration] [callees]
addProject -adam 2015-04-02 18:30 2.5 alice bob
```

[caller]	[xxx]	The username of the caller, should be one of the names from command line arguments.
[datetime]	[YYYY-MM-DD hh:mm]	Date and time of the event, YYYY:Year (4 digits), MM:Month (2 digits), DD:Day (2 digits), hh:Hour (2digits) and mm:Minute (2 digits).
[duration]	[n.n]	Duration of the appointment in hours (fixed point of one decimal place).
[callees]	[xxx yyy]	The username of the callee(s), should be one of the names from command line arguments.

Requirement:

1. Error checking: Caller's name should be one of the names from command line arguments.
2. Error checking: Callees' name should be one of the names from command line arguments.

2.1.5 Add gathering

```
addGathering -[caller] [datetime] [duration] [callees]
addGathering -adam 2015-04-02 18:30 2.5 alice bob
```

[caller]	[xxx]	The username of the caller, should be one of the names from command line arguments.
[datetime]	[YYYY-MM-DD hh:mm]	Date and time of the event, YYYY:Year (4 digits), MM:Month (2 digits), DD:Day (2 digits), hh:Hour (2digits) and mm:Minute (2 digits).
[duration]	[n.n]	Duration of the appointment in hours (fixed point of one decimal place).
[callees]	[xxx yyy]	The username of the callee(s), should be one of the names from command line arguments.

Requirement:

1. Error checking: Caller's name should be one of the names from command line arguments.
2. Error checking: Callees' name should be one of the names from command line arguments.

2.1.6 Add batch

```
addBatch -[filename]
addBatch -batch001.dat
```

[filename]	[path/filename]	Specify a text file that records one or more appointment requests.

Requirement:

1. Error checking: The file is exist and can be read by the program.

2.1.7 Add schedule

```
printSchd -[secheduler]
printSchd -fcfs
```

[secheduler]	[fcfs]	First come first served
	[prio]	Priority
	[opti]	Optimized

Requirement:

1. Error checking: Caller's name should be one of the names from command line arguments.
2. Error checking: Callee's name should be one of the names from command line arguments.

2.1.8 End the program

```
endProgram
```

Requirement:

1. Exit the parent and all the child processes.

2.2 Special requirements

2.2.1 Priority scheduler

For the priority algorithm, the priority of the appointments are as follow:

1. Assignment
2. Project
3. Study
4. Gathering

A higher priority appointment will displace the already scheduled appointments. The already accepted appointment will be moved to the rejected list once a higher priority appointment displace the original appointment.

2.2.2 Modulation

The system should be divided into different modules logically. System call `pipe()` and `fork()` shall be used in the system.

1. Input Module
2. Scheduling Module
3. Output Module

2.2.3 Timeslot

Every 30 minutes as one time slot. The events will be from 18:00 to 22:00 on weekdays. The system shall support at least two weeks' time. Once part of the time slot is occupied, the whole time slot would be treated as unavailable. The system shall support the overflow situation where appointments are more than the available timeslot.

Chapter 3

Design and Implementation

3.1 Data Structure

3.1.1 struct Appointment

The starting time of the appointment is stored in the type *time_t* which is defined in the standard C library <time.h>. This library provides some time manipulation and conversion functions that are useful in the system design.

The main issue is how should we store the duration of the appointment. The input of the appointment contains the duration of the appointment in the format of n.n hours. We have two choice of storing the ending time:

```
time_t start
float duration
```

Or we could use:

```
time_t start
time_t end
```

During the insertion, we need to check whether there are conflicts between the existing appointments and the new appointment. We need to determinate if the new appointment is in between the start time and the end time of the existing appointments. It is more convenient to store the ending time instead of the duration so that we can simply use *difftime()*. If we stored the duration instead of the time, we need to do more calculation each time we insert a new appointment. For more details about the conflict checking, please referee to subsection 3.2.1.

3.1.2 struct AppointmentList

We need to store all the appointments that are following the input order. Since the amount of input is unknown, we need to use the dynamic memory allocation to store the data. Considering most of the access are in linear order, *hashtable* is not useful in this situation. We have two choices here, either using the *array* or *linklist*.

For the `array`:

- Insert new item: **Difficult**
  When the input is too large, we need to allocate a bigger array and copy the original array into it.
- Insert item at the middle: **Difficult**
  We need to move all the items behind the insertion point.
- Accessing time: **Fast**

For the `linklist`:

- Insert new item: **Easy**
- Insert item at the middle: **Easy**
  Simply modify the two pointers.
- Accessing time: **Slow**
  We need to go through the pointers in order to jump to next item.

Since there are a lot of the insertions, linklist performs better in this dimension. We need to insert item at the middle when we doing the scheduling. Although the access of linklist is slower, the gain in insertion should be able to overcome the loss. During the scheduling, we may want to search forward or backward. So we choose a double linked-list to store the appointments.

3.1.3 struct User

We will store the username and a list of accepted and rejected appointments for each user. We will copy the appointment item from the original input list into each users individual list.

```
char username[MAX_USERNAME];
struct AppointmentList *accepted;
struct AppointmentList *rejected;
```

3.2 Algorithm

3.2.1 Check appointment conflict

Each time a new appointment is added, we need to check if there is a conflict between the existing appointments and the new appointment. Assuming we are now checking is there are any conflict between $Item_A$ and $Item_B$. There are two cases that the two items are *not* conflict 1.

Algorithm 1 Check conflict between two appointment item

```
if Item$_A$ happened before Item$_B$ or Item$_A$ happened after Item$_B$ then
    Conflict happen
end if
```

3.2.2 First come first served

This scheduling algorithm will allocate the timeslot to the first appointment. The late requests will be rejected.

Algorithm 2 First come first served (FCFS)

```
Let $Item_{new}$ be the new appointment
for each $List_i$ in accepted list of caller and callee do
    if Conflict between $List_i$ and $Item_{new}$ then
        Move $Item_{new}$ to rejected list          ▷ In FCFS, late requests will be rejected
        exit
    end if
end for
Move $Item_{new}$ to accepted list
```

3.2.3 Priority

This scheduling algorithm will allocate the timeslot to the appointment which has the highest priority. The priority should follow the requirement documented in subsection 2.2.1.

Algorithm 3 Priority

```
Let $Item_{new}$ be the new appointment

▷ Add all the conflict items into $List_{temp}$
for each $Item_i$ in accepted list of caller and callee do
    if Conflict between $Item_i$ and $Item_{new}$ then
        Add $Item_i$ into $List_{temp}$
    end if
end for

▷ Check if all the conflict items have lower priority than the new item
for each $Item_i$ in $List_{temp}$ do
    if Priority of $Item_i > Item_{new}$ then
        Move $Item_{new}$ to rejected list          ▷ The existing accepted appointment have higher priority. Reject the new item.
        exit
    end if
end for

▷ All the conflict items have lower priority than the new item. Move the original items to rejected list
for each $Item_i$ in $List_{temp}$ do
    Move $Item_i$ to rejected list
end for
Move $Item_{new}$ to accepted list
```

3.2.4 Optimal

Algorithm 4 Optimal

```
Call Priority Scheduler
▷ Try to reschedule each item in the rejected lists
for each $Item_i$ in the rejected list of each user do
    $List_{free}$ ← Free timeslots in the coming 3 day of $List_i$ of the callee
    for each $Timeslot_i$ in $List_{free}$ do
        Sub $Item_i$ into $Timeslot_i$
        if no conflict between the $Item_i$ and the accepted appointments then
            $Item_i$.Time = $Timeslot_i$
            $Item_i$.Rescheduled = True
            Move $Item_i$ to accepted list        ▷ Successfully rescheduled an appointment
            break
        end if
    end for
end for
```