COMP2432 Group Project Design Documentation

Star Poon

April 4, 2015

Chapter 1

Foreword

1.1 Introduction

1.2 Scope

1.3 Requirements

Chapter 2

Design and Implementation

2.1 Data Structure

2.1.1 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 0.2.1.

2.1.2 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.

2.1.3 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;
```

2.2 Common Algorithm

2.2.1 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 2.

Algorithm 1 Check conflict between two appointment item

```
if $Item_A.Start \leq Item_B.Start$ and $Item_A.End \leq Item_B.Start$ then
    $Item_A$ happened before $Item_B$
else if $Item_A.Start \geq Item_B.End$ and $Item_A.End \geq Item_B.End$ then
    $Item_A$ happened after $Item_B$
else
    Conflict happen
```

2.3 Scheduler

2.3.1 First come first served (FCFS)

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

Algorithm 2 First come first served (FCFS)

```
if $Item_A.Start \leq Item_B.Start$ and $Item_A.End \leq Item_B.Start$ then
```

2.3.2 Priority

2.3.3 Optimal