COMP2432 Group Project Design Documentation

CHEUNG Wai Fai 13003551D
POON Yat Sing 13040015D
LIU Wang Ho 13068152D
LAM Pui Yan 12139709D
KWAN Leung Yu 13027272D

April 8, 2015

Contents

Chapter 1

Foreword

1.1 Introduction

1.1.1 Problem Identification

Being a student is not easy in today's school life. A student may take up to 7 subjects while each subject may have their own individual assignments and group projects with deadline very close to each other. In addition, students may have nonacademic activities and meetings. It is a fact that solving this problem by changing the academic system is impossible. Therefore good time management is the only solution and becomes a serious issue for today's students. Nevertheless there are some obstacles for students to have good time scheduling.

Students have hard time to arrange time for group discussions. As each group member may study different programme which have different class timetable, it is difficult to come up with a time slot which all the members are available. The problem lies on the difficulty of comparing each other's schedule manually.

Student have difficulty to rearrange their schedule for some important events. It is a fact that students' schedule is always in a packed situation. However, there are still many cases which some emergency appointment is requested and there is no choice but to change the original schedule. Nevertheless since the schedule already packed, it is hard for students to manually search for alternative time for the original work.

1.1.2 Objective

To address these issues, we would like to design an Appointment Manager (AMR) in order to deal with the issue of appointment scheduling and schedules rearrangement. Our system have the following objectives.

- Allow multiple users create their own schedule
- Allow easy arrangement for group events such as group project discussion
- Provide alternative time slot in case of schedules rearrangement

1.2 Scope

1.2.1 Process creation

Process creation means a process is created when a program is running. The process that creating is the parent process. The created process is the child process. Fork() system call is used in Unix/Linux for process creation. Parent process and child process execute concurrently. It will enhance the efficiency. Parent process need to wait until child process completed. After parent process collected a child process, it will resume its process.

There are 3 relationships between parent and child processes. In terms of resource sharing, as parent and child are two completely independent process, they share no resources. The child process continue to execute right at the position the process is fork(). The child process have the same variable as parent process right before it is fork(). In terms of execution, parent and child process can run concurrently. In normal case, parent process should wait it's child terminated before terminate itself.

1.2.2 Interprocess communication

Interprocess communication allows processes to communicate and to synchronise. Shared memory mechanism is used to create shared memory segments within kernel memory space. Message passing mechanism allows processes to communicate via pipes and sockets.

Pipe is a mechanism in the OS which allows multiple processes communicate through channels inside kernel. This prevent unauthorised change done outside program itself. There are mainly 2 types of pipe which are unnamed and named pipe.

For the unnamed pipe, it is created with system call pipe(). Pipe allows parent and child communication and it's created before fork(). A pipe itself allow two way communication. However, as OS cannot control which process read the output, it is danger to use one pipe to handle both read and write. We will close() the unused pipe to prevent accidental use.

For the named pipe, it is actually a special file created by mkfifo() what multiple process can read and write at the same time. Those processes does not necessary in a parent and child relationship. The pipe name is assigned by the program. Several processes connect together with the same name. The disadvantage of using named pipe is it may create a security issue as anyone can read and write into the file as long as they have the name of the pipe.

1.3 Process Scheduling

For the appointment scheduling in our system, we basically follow the concept of CPU scheduling. We focus on First Come First Serve (FCFS) and Priority Scheduling.

FCFS

For the FCFS, it basically accept and entertain the process that come earlier in time. In terms of CPU scheduling, the CPU will process the whole earlier coming process before entertaining the next one. The waiting process will queue according to their arrive time. Note that each process will be assigned a process ID. Normally process come earlier will have a smaller PID in UNIX/Linux. Which means a process with smaller PID will be processed first if 2 processes come at around the same time.

In terms of our system, FCFS means appointment comes earlier will be scheduled first as long as the timeslots are available.

Priority Scheduling

For the priority, the idea is to accept the process according to its priority. Each coming process will also be manually assigned with a priority. The process with higher priority will run first. If a higher priority process arrive while a lower priority process is running, the current running process will be suspended and let the higher priority process run first. The lower priority process will be resumed after the higher priority process terminated.

In terms of our system. If a time slot is available, any appointment will be accepted. However if the time slot is filled, the system will compare the priority of the new and current appointments. If the new appointment have a higher priority, the new appointment will replace the original appointment.

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

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
```

Performance Analysis:

First come first served algorithm is easy to implement. Also the execution time is fast compare to the other two scheduler. It is because when a new appointment is added into the timetable, it only needs to compare is there any time conflict between the existing one and the new one. It doesn't need to take care priority of the appointments.

The drawback of this algorithm is that it it not intelligent. If some important appointment such as assignment has been added, the program may still allocate the time to the gathering. Also the timeslot usage will be poor because no re-scheduling has been done and the user needs to take care the time reallocation.

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
```

Performance Analysis:

Priority has a slower execution time compare to FCFS. It is because it needs to calculate the priority in order to decide whether the new appointment should be accepted or not. It is more intelligent because the timeslot will be allocated to the more important jobs. But still the algorithm doesn't have a re-schedule function. Since we need to move the already accepted item to reject list, the algorithm will be a lot more complicated to implement.

3.2.4 Optimal

The optimal algorithm first allocate the timeslot using the priority scheduler to ensure the timeslot will be kept for the high priority appointments. After that, the algorithm will go through the rejected list and try to re-schedule them as much as possible. In our design, the rejected job will be reallocated between the day of the appointment and 3 days after that. If there are no available timeslot between this 3 days, this appointment will be rejected.

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
    $List_{free} \leftarrow$ 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
```

Performance Analysis:

Compare to the above algorithms, this algorithm is most complicated and require the most computational time. Also, this is the most intelligent algorithm. Because it will first allow the high priority appointment to use the timeslot. Then it will go through the rejected appointments one-by-one and try to re-schedule the rejected appointments. Thus, this algorithm will have a higher timeslot usage rate. The user will be able to finish the important job first and then use the spare time for the lower priority job.

Chapter 4

Implementation

4.1 Software Structure

The program is divided into multiple modules. This can improve the readability and reusability of the code. Each module has a header file with comments to describe it's API. In this section, we will only describe the modules briefly. Please refer to the appendix for the detailed API documentation.

1. Appointment
2. Scheduler
3. User
4. Main Input Loop

4.1.1 Appointment Module

This modules contain the `Appointment` and `AppointmentList` structure. Also, it contains the functions that are useful for handling these two structure. The list is a double-linked list. All the create, insert, delete and sorting algorithms done on `AppointmentList` are included in this module. This module is the fundamental part of the program and other modules heavily depends on this module. Also, we use the `AppointmentList` to store all the input appointments and the scheduled accept appointments.

4.1.2 User Module

This modules contain the `user` structure that are responsible for storing the individual accepted and rejected appointments. Also it will provide function to validate the username is duplicated or not.

4.1.3 Scheduler Module

This modules contain the three scheduling algorithms. All the algorithm will return a summary of the performance, such as total number of accepted or rejected appointments, the utilization of timeslot of each user. This modules provide functions to check if the timeslot is available across different users.

4.1.4 Input Loop

When the program first startup, the program will check the command line argument. It will make sure the number of user is between 3 and 10. After that, the program will check the username and make sure there are not duplicated usernames.

The program will be looping and wait for the appointment inputs until the endProgam command is received. The program allow user to adding appointment individually and using batch file. It allow recursive addBatch command inside the batch file.

4.2 Fork and Pipe

This program using the `fork()` system call to create a child to handle the scheduling and printing of appointment. After the child process finished the scheduling, it will return a summary back to the parent using `pipe()`. After the scheduling, the child will terminate itself.

By using this approach, we fully utilize the property of `fork()`. The child will have the same copy of memory as the parent. This means when a scheduler is started, a child will be created and the child will own a copy of all the input appointments. After the scheduling finished, the child will print the result and terminate itself. This can prevent any memory leakage during the scheduling. The parent will continue to run after the child return the summary.

Chapter 5

Testing

Chapter 6

Deployment

6.1 Build

To compile the program:

```
make
```

The executable program is located in `bin`.

To clean up the object files:

```
make clean
```

To clean up the object files and the executable file:

```
make remove
```

To join the source files into one AMR.c file:

```
make onefile
```

Appendix A

Testing