Appointment Manager (AMR)

Generated by Doxygen 1.8.8

Wed Apr 8 2015 21:34:51

Contents

Chapter 1

Main Page

An appointment management software that have the calendar and scheduling function.

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 "‘

Documentation

Project doucmentationi: doc/AMRReport.pdf

API documentation: doc/API.pdf

File structure

-bin/ ..... executable file -src/ ..... source and header files -obj/ ..... object files during make -doc/ ..... documentations -test/in .. testing input -test/out . testing output

How to test

Use input redirection ./bin/amr alice bob charlie < test/in/fcfs.in ./bin/amr alice bob charlie < test/in/prio.in ./bin/amr alice bob charlie < test/in/opti.in

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 Appointment Struct Reference

Store a appointment record.

`#include <appointment_list.h>`

Data Fields

- enum **AppointmentType type**
- int **id**
- int **caller_id**
- int **callee_id** [10]
- time_t **start**
- time_t **end**
- int **is_accepted**
- int **rescheduled**
- char **reason** [50]
- struct **Appointment** ∗ **prev**
- struct **Appointment** ∗ **next**

4.1.1 Detailed Description

Store a appointment record.

Definition at line **48** of file **appointment_list.h**.

The documentation for this struct was generated from the following file:

- **appointment_list.h**

4.2 AppointmentList Struct Reference

A double-linked list for appointment record.

`#include <appointment_list.h>`

Data Fields

- int **count**
- struct **Appointment** ∗ **head**
- struct **Appointment** ∗ **tail**

4.2.1 Detailed Description

A double-linked list for appointment record.

Definition at line **67** of file **appointment_list.h**.

The documentation for this struct was generated from the following file:

- **appointment_list.h**

4.3 AppointmentType Struct Reference

Store all the appointment type.

```
#include <appointment_list.h>
```

4.3.1 Detailed Description

Store all the appointment type.

The documentation for this struct was generated from the following file:

- **appointment_list.h**

4.4 Summary Struct Reference

Data Fields

- int **total_accepted**
- int **total_rejected**
- int **accepted** [**USER_NUMBER**]
- int **rejected** [**USER_NUMBER**]
- int **empty_timeslot** [**USER_NUMBER**]
- time_t **start**
- time_t **end**

4.4.1 Detailed Description

Definition at line **28** of file **scheduler.h**.

The documentation for this struct was generated from the following file:

- **scheduler.h**

4.5 User Struct Reference

Store the basic information of the user and the appointments.

```
#include <user.h>
```

Data Fields

- char **username** [**MAX_USERNAME**]
- struct **AppointmentList** ∗ **accepted**
- struct **AppointmentList** ∗ **rejected**

4.5.1 Detailed Description

Store the basic information of the user and the appointments.

Definition at line **40** of file **user.h**.

The documentation for this struct was generated from the following file:

- **user.h**

Chapter 5

File Documentation

5.1 appointment_list.c File Reference

Handling the appointments and appointment list.

```
#include "appointment_list.h"
#include "user.h"
```

Functions

- struct **AppointmentList** ∗ **CreateAppointmentList** ()

  Create a appointment list and init the value.
- struct **Appointment** ∗ **CreateAppointment** ()

  Create a appointment and init the value.
- void **AddAppointment** (struct **AppointmentList** ∗list, const struct **Appointment** ∗newItem)

  Insert a copy of the appointment into the end of the appointment list.
- void **AddAppointmentOrdered** (struct **AppointmentList** ∗list, const struct **Appointment** ∗newItem)

  Insert a copy of the appointment into the sorted appointment list. Using the start time then end time as the sorting condition.
- void **AddAppointmentFromList** (struct **AppointmentList** ∗dst_list, const struct **AppointmentList** ∗src_list)

  Insert a copy of the appointment into the end of the appointment list.
- void **AddAppointmentOrderedFromList** (struct **AppointmentList** ∗dst_list, const struct **AppointmentList** ∗src_list)

  Insert a copy of the appointment into the sorted appointment list. Using the start time then end time as the sorting condition.
- int **CompareAppointment** (const struct **Appointment** ∗a, const struct **Appointment** ∗b)

  Compare the start time and then end time of the appointment. Used to keep the ordered appointment list.
- int **CompareAppointmentPriority** (const struct **Appointment** ∗a, const struct **Appointment** ∗b)

  Compare the appointment by it's priority.
- void **RemoveItemFromList** (struct **AppointmentList** ∗list, const struct **Appointment** ∗item)

  Remove an item from the list. Items should be unique inside the list. Delete if the two item have the same id.
- void **RemoveListFromList** (struct **AppointmentList** ∗ori_list, const struct **AppointmentList** ∗del_list)

  Remove a list of items from the list. Items should be unique inside the list. Delete if the two item have the same id.
- void **PrintAppointment** (const struct **Appointment** ∗item)

  Print out the appointment.
- void **PrintAppointmentList** (const struct **AppointmentList** ∗list)

  Print out the appointment list.

- int **IsConflict** (const struct **Appointment** ∗a, const struct **Appointment** ∗b)

    Check whether if two appointments have time conflict.
- int **IsConflictInList** (const struct **AppointmentList** ∗list, const struct **Appointment** ∗item)

    Check whether if the appointment item have conflict with the list.
- struct **AppointmentList** ∗ **ConflictInList** (const struct **AppointmentList** ∗list, const struct **Appointment** ∗item)

    Check whether the the new appointment is conflict with the existing appointments that are already in the list.
- struct **Appointment** ∗ **GetAppointmentById** (const struct **AppointmentList** ∗list, int id)

    Return the appointment that match the id in the list.
- void **SetReasonForList** (struct **AppointmentList** ∗list, const char ∗reason)

    Set the reject reason.

Variables

- const char ∗ **AppointmentTypeStr** [ ]

5.1.1 Detailed Description

Handling the appointments and appointment list.

Author

oneonestar `oneonestar@gmail.com`

Version

1.0

Copyright

2015

5.1.2 LICENSE

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see `http://www.gnu.org/licenses/`.

Definition in file **appointment_list.c**.

5.1.3 Function Documentation

5.1.3.1 void AddAppointment (struct AppointmentList ∗ *list,* const struct Appointment ∗ *newItem*)

Insert a copy of the appointment into the end of the appointment list.

Parameters

out	*list*	The destination appointment list.
in	*newItem*	The item that needs to add into the list.

Definition at line **65** of file **appointment_list.c**.

5.1.3.2 void AddAppointmentFromList (struct AppointmentList ∗ *dst_list*, const struct AppointmentList ∗ *src_list*)

Insert a copy of the appointment into the end of the appointment list.

Parameters

out	*list*	The destination appointment list.
in	*newItem*	The item that needs to add into the list.

Definition at line **125** of file **appointment_list.c**.

5.1.3.3 void AddAppointmentOrdered (struct AppointmentList ∗ *list*, const struct Appointment ∗ *newItem*)

Insert a copy of the appointment into the sorted appointment list. Using the start time then end time as the sorting condition.

Parameters

out	*list*	The destination appointment list.
in	*newItem*	The item that needs to add into the list.

Definition at line **84** of file **appointment_list.c**.

5.1.3.4 void AddAppointmentOrderedFromList (struct AppointmentList ∗ *dst_list*, const struct AppointmentList ∗ *src_list*)

Insert a copy of the appointment into the sorted appointment list. Using the start time then end time as the sorting condition.

Parameters

out	*list*	The destination appointment list.
in	*newItem*	The item that needs to add into the list.

Definition at line **135** of file **appointment_list.c**.

5.1.3.5 int CompareAppointment (const struct Appointment ∗ *a*, const struct Appointment ∗ *b*)

Compare the start time and then end time of the appointment. Used to keep the ordered appointment list.

Parameters

in	*a*	**Appointment** (p. 7) to be compared.
in	*b*	**Appointment** (p. 7) to be compared.

Return values

<0	a is before b
0	a is equal to b

>0	a is after b

Definition at line **145** of file **appointment_list.c**.

5.1.3.6 int CompareAppointmentPriority (const struct **Appointment** * *a*, const struct **Appointment** * *b*)

Compare the appointment by it's priority.

Parameters

`in`	*a*	**Appointment** (p. 7) to be compared.
`in`	*b*	**Appointment** (p. 7) to be compared.

Return values

<0	a is before b
0	a is equal to b
>0	a is after b

Definition at line **155** of file **appointment_list.c**.

5.1.3.7 struct **AppointmentList*** ConflictInList (const struct **AppointmentList** * *list*, const struct **Appointment** * *item*)

Check whether the the new appointment is conflict with the existing appointments that are already in the list.

Parameters

`in`	*list*	The destination appointment list.
`in`	*newItem*	The item that needs to add into the list.

Definition at line **254** of file **appointment_list.c**.

5.1.3.8 void PrintAppointment (const struct **Appointment** * *item*)

Print out the appointment.

Parameters

item	**Appointment** (p. 7) to be printed.

Definition at line **192** of file **appointment_list.c**.

5.1.3.9 void PrintAppointmentList (const struct **AppointmentList** * *list*)

Print out the appointment list.

Parameters

list	**Appointment** (p. 7) list to be printed.

Definition at line **224** of file **appointment_list.c**.

5.1.4 Variable Documentation

5.1.4.1 const char* AppointmentTypeStr[ ]

Initial value:

```
= {[STUDY] = "Study", [ASSIGNMENT] = "Assignment",
    [PROJECT] = "Project", [GATHERING] = "Gathering"}
```

For printing

Definition at line **26** of file **appointment_list.c**.

5.2 appointment_list.c

```
00001
00022 #include "appointment_list.h"
00023 #include "user.h"
00024
00026 const char *AppointmentTypeStr[] = {[STUDY] = "Study", [ASSIGNMENT] = "Assignment",
00027     [PROJECT] = "Project", [GATHERING] = "Gathering"};
00028
00029 /****************************************************
00030  * Implementation
00031  ****************************************************/
00032 struct AppointmentList* CreateAppointmentList()
00033 {
00034     struct AppointmentList *list = (struct AppointmentList*)malloc(sizeof(struct
      AppointmentList));
00035     if(!list)
00036     {
00037         fprintf(stderr, "Failed to allocate memory.\n");
00038         exit(EXIT_FAILURE);
00039     }
00040     list->count = 0;
00041     list->head = NULL;
00042     list->tail = NULL;
00043     return list;
00044 }
00045
00046 struct Appointment* CreateAppointment()
00047 {
00048     struct Appointment *item = (struct Appointment*)malloc(sizeof(struct
      Appointment));
00049     if(!item)
00050     {
00051         fprintf(stderr, "Failed to allocate memory.\n");
00052         exit(EXIT_FAILURE);
00053     }
00054     for(int i=0; i<USER_NUMBER; i++)
00055         item->callee_id[i] = -1;
00056     item->is_accepted = 0;
00057     item->id = -1;
00058     item->rescheduled = 0;
00059     item->prev = NULL;
00060     item->next = NULL;
00061     strcpy(item->reason, "");
00062     return item;
00063 }
00064
00065 void AddAppointment(struct AppointmentList *list, const struct Appointment *newItem)
00066 {
00067     struct Appointment *item = CreateAppointment();
00068     *item = *newItem;
00069     item->next = item->prev = 0;
00070     if(!list->head) //if the list is empty
00071     {
00072         list->head = item;
00073         list->tail = item;
00074     }
00075     else
00076     {
00077         list->tail->next = item;
00078     }
00079     item->prev = list->tail;
00080     list->tail = item;
00081     list->count++;
00082 }
00083
00084 void AddAppointmentOrdered(struct AppointmentList *list, const struct
      Appointment *newItem)
00085 {
00086     struct Appointment *item = CreateAppointment();
00087     *item = *newItem;
00088     item->next = item->prev = 0;
00089     struct Appointment *ptr = list->head;
00090     //if the list is empty
00091     if(!ptr)
```

```
    {
        list->head = item;
        list->tail = item;
    }
    else if(CompareAppointment(item, ptr)<0)    //if insert at the head
    {
        if(!list->head)
            list->head->prev = item;
        item->next = list->head;
        list->head = item;
    }
    else    //insert at middle or at the tail
    {
        while(ptr->next)    //find the insertion position
        {
            if(difftime(item->start, ptr->next->start)<0)
                break;
            ptr = ptr->next;
        }
        if(!ptr)    //insert at the tail
        {
            ptr = list->tail;
            list->tail = item;
        }
        item->prev = ptr;   //insert after ptr
        item->next = ptr->next;
        if(item->next)
            item->next->prev = item;
        ptr->next = item;
    }
    list->count++;
}

void AddAppointmentFromList(struct AppointmentList *dst_list, const struct AppointmentList *src_list)
{
    struct Appointment *newItem = src_list->head;
    while(newItem)
    {
        AddAppointment(dst_list, newItem);
        newItem = newItem->next;
    }
}

void AddAppointmentOrderedFromList(struct AppointmentList *dst_list, const struct AppointmentList *src_list)
{
    struct Appointment *newItem = src_list->head;
    while(newItem)
    {
        AddAppointmentOrdered(dst_list, newItem);
        newItem = newItem->next;
    }
}

int CompareAppointment(const struct Appointment *a, const struct Appointment *b)
{
    if(difftime(a->start, b->start)<0)
        return -1;  //a before b
    else if(difftime(a->start, b->start)==0)
        return difftime(a->end, b->end);
    else
        return 1;
}

int CompareAppointmentPriority(const struct Appointment *a, const struct Appointment *b)
{
    return a->type - b->type;
}

void RemoveItemFromList(struct AppointmentList *list, const struct Appointment *item)
{
    struct Appointment *delItem = list->head;
    while(delItem)
    {
        if(delItem->id == item->id)
        {
            if(!delItem->prev)  //if prev is null, first item in list
                list->head = delItem->next;
            else
                delItem->prev->next = delItem->next;
            if(!delItem->next)  //if next is null, last item in list
                list->tail = delItem->prev;
            else
                delItem->next->prev = delItem->prev;
            list->count--;
```

```c
            return;
        }
        delItem = delItem->next;
    }
}

void RemoveListFromList(struct AppointmentList *ori_list, const struct AppointmentList *del_list)
{
    struct Appointment *delItem = del_list->head;
    while(delItem)
    {
        RemoveItemFromList(ori_list, delItem);
        delItem = delItem->next;
    }
}

void PrintAppointment(const struct Appointment *item)
{
    struct tm tm_start, tm_end;
    memcpy(&tm_start, localtime (&item->start), sizeof(struct tm));
    memcpy(&tm_end, localtime (&item->end), sizeof(struct tm));
    printf("%2d ", item->id);
    printf("%4d-%02d-%02d   %02d:%02d   %02d:%02d   %-12s ", tm_start.tm_year+1900, tm_start.tm_mon+1, tm_start.tm_mday, tm_start.tm_hour,
     tm_start.tm_min, tm_end.tm_hour, tm_end.tm_min, AppointmentTypeStr[item->type]);
    if(item->rescheduled)
        printf("%-9c ",  'Y');
    else
        printf("%-9c ", 'N');
    if(strcmp(item->reason, ""))
    {
        printf(" %s", item->reason);
    }
    else
    {
        if(item->callee_id[0] == -1)
            printf("-");
        else
            printf("%s ", user[item->caller_id].username);
        for(int i=0; i<USER_NUMBER; i++)
        {
            if(item->callee_id[i]==-1)
                break;
            printf("%s ", user[item->callee_id[i]].username);
        }
    }
    printf("\n");
}

void PrintAppointmentList(const struct AppointmentList *list)
{
    struct Appointment *ptr = list->head;
    while(ptr!=NULL)
    {
        PrintAppointment(ptr);
        ptr = ptr->next;
    }
}

int IsConflict(const struct Appointment *a, const struct Appointment *b)
{
    return !(difftime(a->end, b->start)<=0 ||   //a before b
        difftime(a->start, b->end)>=0);     //a after b
}

int IsConflictInList(const struct AppointmentList *list, const struct Appointment *item)
{
    if(!list || !item)
        return 0;
    struct Appointment *ptr = list->head;
    while(ptr)
    {
        if(IsConflict(ptr, item))
            return 1;
        ptr = ptr->next;
    }
    return 0;
}

struct AppointmentList* ConflictInList(const struct AppointmentList *list, const struct Appointment *item)
{
    if(!list || !item)
        return NULL;
    struct AppointmentList *conflict_list = CreateAppointmentList();
    struct Appointment *ptr = list->head;
    while(ptr)
    {
        if(IsConflict(ptr, item))
            AddAppointment(conflict_list, ptr);
        ptr = ptr->next;
    }
    return conflict_list;
}

struct Appointment* GetAppointmentById(const struct AppointmentList *list, int id)
{
    if(!list)
        return NULL;
    struct Appointment *ptr = list->head;
    while(ptr)
    {
        if(ptr->id == id)
            return ptr;
        ptr = ptr->next;
    }
    return NULL;
}


void SetReasonForList(struct AppointmentList *list, const char *reason)
{
    if(!list)
        return;
    struct Appointment *ptr = list->head;
    while(ptr)
    {
        strcpy(ptr->reason, reason);
        ptr = ptr->next;
    }
}
```

5.3 appointment_list.h File Reference

Handling the appointments and appointment list.

```
#include <ctype.h>
#include <math.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "user.h"
```

Data Structures

- struct **Appointment**

  Store a appointment record.
- struct **AppointmentList**

  A double-linked list for appointment record.

Enumerations

- enum **AppointmentType** { **ASSIGNMENT** = 0, **PROJECT**, **STUDY**, **GATHERING** }

Functions

- struct **Appointment** ∗ **CreateAppointment** ()

  Create a appointment and init the value.

- struct **AppointmentList** ∗ **CreateAppointmentList** ()

  Create a appointment list and init the value.
- void **AddAppointment** (struct **AppointmentList** ∗list, const struct **Appointment** ∗newItem)

  Insert a copy of the appointment into the end of the appointment list.
- void **AddAppointmentOrdered** (struct **AppointmentList** ∗list, const struct **Appointment** ∗newItem)

  Insert a copy of the appointment into the sorted appointment list. Using the start time then end time as the sorting condition.
- void **AddAppointmentFromList** (struct **AppointmentList** ∗dst_list, const struct **AppointmentList** ∗src_list)

  Insert a copy of the appointment into the end of the appointment list.
- void **AddAppointmentOrderedFromList** (struct **AppointmentList** ∗dst_list, const struct **AppointmentList** ∗src_list)

  Insert a copy of the appointment into the sorted appointment list. Using the start time then end time as the sorting condition.
- struct **AppointmentList** ∗ **ConflictInList** (const struct **AppointmentList** ∗list, const struct **Appointment** ∗item)

  Check whether the the new appointment is conflict with the existing appointments that are already in the list.
- int **IsConflict** (const struct **Appointment** ∗a, const struct **Appointment** ∗b)

  Check whether if two appointments have time conflict.
- int **IsConflictInList** (const struct **AppointmentList** ∗list, const struct **Appointment** ∗item)

  Check whether if the appointment item have conflict with the list.
- void **RemoveItemFromList** (struct **AppointmentList** ∗list, const struct **Appointment** ∗item)

  Remove an item from the list. Items should be unique inside the list. Delete if the two item have the same id.
- void **RemoveListFromList** (struct **AppointmentList** ∗ori_list, const struct **AppointmentList** ∗del_list)

  Remove a list of items from the list. Items should be unique inside the list. Delete if the two item have the same id.
- int **CompareAppointment** (const struct **Appointment** ∗a, const struct **Appointment** ∗b)

  Compare the start time and then end time of the appointment. Used to keep the ordered appointment list.
- int **CompareAppointmentPriority** (const struct **Appointment** ∗a, const struct **Appointment** ∗b)

  Compare the appointment by it's priority.
- void **PrintAppointment** (const struct **Appointment** ∗item)

  Print out the appointment.
- void **PrintAppointmentList** (const struct **AppointmentList** ∗list)

  Print out the appointment list.
- struct **Appointment** ∗ **GetAppointmentById** (const struct **AppointmentList** ∗list, int id)

  Return the appointment that match the id in the list.
- void **SetReasonForList** (struct **AppointmentList** ∗list, const char ∗reason)

  Set the reject reason.

5.3.1 Detailed Description

Handling the appointments and appointment list.

Author

oneonestar `oneonestar@gmail.com`

Version

1.0

Copyright

2015

5.3.2 LICENSE

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see `http://www.gnu.org/licenses/`.

Definition in file **appointment_list.h**.

5.3.3 Function Documentation

5.3.3.1 void AddAppointment (struct **AppointmentList** ∗ *list,* const struct **Appointment** ∗ *newItem*)

Insert a copy of the appointment into the end of the appointment list.

Parameters

`out`	*list*	The destination appointment list.
`in`	*newItem*	The item that needs to add into the list.

Definition at line **65** of file **appointment_list.c**.

5.3.3.2 void AddAppointmentFromList (struct **AppointmentList** ∗ *dst_list,* const struct **AppointmentList** ∗ *src_list*)

Insert a copy of the appointment into the end of the appointment list.

Parameters

`out`	*list*	The destination appointment list.
`in`	*newItem*	The item that needs to add into the list.

Definition at line **125** of file **appointment_list.c**.

5.3.3.3 void AddAppointmentOrdered (struct **AppointmentList** ∗ *list,* const struct **Appointment** ∗ *newItem*)

Insert a copy of the appointment into the sorted appointment list. Using the start time then end time as the sorting condition.

Parameters

`out`	*list*	The destination appointment list.
`in`	*newItem*	The item that needs to add into the list.

Definition at line **84** of file **appointment_list.c**.

5.3.3.4 void AddAppointmentOrderedFromList (struct **AppointmentList** ∗ *dst_list,* const struct **AppointmentList** ∗ *src_list*)

Insert a copy of the appointment into the sorted appointment list. Using the start time then end time as the sorting condition.

Parameters

`out`	*list*	The destination appointment list.

`in`	*newItem*	The item that needs to add into the list.

Definition at line **135** of file **appointment_list.c**.

5.3.3.5 int CompareAppointment (const struct **Appointment** ∗ *a,* const struct **Appointment** ∗ *b*)

Compare the start time and then end time of the appointment. Used to keep the ordered appointment list.

Parameters

`in`	*a*	**Appointment** (p. 7) to be compared.
`in`	*b*	**Appointment** (p. 7) to be compared.

Return values

<0	a is before b
0	a is equal to b
>0	a is after b

Definition at line **145** of file **appointment_list.c**.

5.3.3.6 int CompareAppointmentPriority (const struct **Appointment** ∗ *a,* const struct **Appointment** ∗ *b*)

Compare the appointment by it's priority.

Parameters

`in`	*a*	**Appointment** (p. 7) to be compared.
`in`	*b*	**Appointment** (p. 7) to be compared.

Return values

<0	a is before b
0	a is equal to b
>0	a is after b

Definition at line **155** of file **appointment_list.c**.

5.3.3.7 struct **AppointmentList**∗ ConflictInList (const struct **AppointmentList** ∗ *list,* const struct **Appointment** ∗ *item*)

Check whether the the new appointment is conflict with the existing appointments that are already in the list.

Parameters

`in`	*list*	The destination appointment list.
`in`	*newItem*	The item that needs to add into the list.

Definition at line **254** of file **appointment_list.c**.

5.3.3.8 void PrintAppointment (const struct **Appointment** ∗ *item*)

Print out the appointment.

Parameters

item	**Appointment** (p. 7) to be printed.

Definition at line **192** of file **appointment_list.c**.

5.3.3.9 void PrintAppointmentList (const struct **AppointmentList** ∗ *list*)

Print out the appointment list.

Parameters

list	**Appointment** (p. 7) list to be printed.

Definition at line **224** of file **appointment_list.c**.

5.4 appointment_list.h

```
#ifndef APPOINTMENT_LIST
#define APPOINTMENT_LIST

#include <ctype.h>
#include <math.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "user.h"

enum AppointmentType
{
    ASSIGNMENT = 0, PROJECT, STUDY, GATHERING
};

struct Appointment
{
    enum AppointmentType type;
    int id;
    int caller_id;
    int callee_id[10];
    time_t start;
    time_t end;
    int is_accepted;
    int rescheduled;
    char reason[50];
    struct Appointment *prev;
    struct Appointment *next;
};

struct AppointmentList
{
    int count;
    struct Appointment *head;
    struct Appointment *tail;
};


struct Appointment* CreateAppointment();

struct AppointmentList* CreateAppointmentList();


void AddAppointment(struct AppointmentList *list, const struct Appointment *newItem);

void AddAppointmentOrdered(struct AppointmentList *list, const struct
Appointment *newItem);

void AddAppointmentFromList(struct AppointmentList *dst_list, const struct
AppointmentList *src_list);

void AddAppointmentOrderedFromList(struct AppointmentList *dst_list, const struct
AppointmentList *src_list);

struct AppointmentList* ConflictInList(const struct AppointmentList *list, const struct
Appointment *item);

int IsConflict(const struct Appointment *a, const struct Appointment *b);
int IsConflictInList(const struct AppointmentList *list, const struct
    Appointment *item);

void RemoveItemFromList(struct AppointmentList *list, const struct Appointment *item);

void RemoveListFromList(struct AppointmentList *ori_list, const struct
    AppointmentList *del_list);

int CompareAppointment(const struct Appointment *a, const struct Appointment *b);

int CompareAppointmentPriority(const struct Appointment *a, const struct
    Appointment *b);

void PrintAppointment(const struct Appointment *item);

void PrintAppointmentList(const struct AppointmentList *list);

struct Appointment* GetAppointmentById(const struct AppointmentList *list, int id);

void SetReasonForList(struct AppointmentList *list, const char *reason);

#endif
```

5.5 main.c File Reference

Appointment (p. 7) Manager (AMR) main program. Also the input handling.

```
#include <ctype.h>
#include <math.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include "appointment_list.h"
#include "scheduler.h"
#include "user.h"
```

Functions

- void **HandleInput** (const char ∗line)
- void **HandleSchedule** (const char ∗algorithm)
- void **inputLoop** (FILE ∗stream)
- int **main** (int argc, char ∗argv[])

Variables

- int **NumOfUser**
- struct **User user** [**USER_NUMBER**]
- struct **AppointmentList** ∗ **inputList**

5.5.1 Detailed Description

Appointment (p. 7) Manager (AMR) main program. Also the input handling.

Author

oneonestar oneonestar@gmail.com

Version

1.0

Copyright

2015

5.5.2 LICENSE

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.

Definition in file **main.c**.

5.5.3 Variable Documentation

5.5.3.1 int NumOfUser

Global variable storing current number of user.

Definition at line **55** of file **user.h**.

5.5.3.2 struct User user[USER_NUMBER]

Global variable storing the user data.

Definition at line **50** of file **user.h**.

5.6 main.c

```

#include <ctype.h>
#include <math.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>

#include "appointment_list.h"
#include "scheduler.h"
#include "user.h"

extern int NumOfUser;
extern struct User user[USER_NUMBER];

struct AppointmentList *inputList;

void HandleInput(const char *line);
void HandleSchedule(const char *algorithm);
void inputLoop(FILE *stream);

void HandleSchedule(const char *algorithm)
{
    if(inputList->count == 0)
    {
        printf("Empty timetable.\n");
        return;
    }

#ifdef NO_FORK
    struct Summary *summary = NULL;
    //TODO: remove items from list
    for (int i=0; i<NumOfUser; i++)
    {
        user[i].accepted = CreateAppointmentList();
        user[i].rejected = CreateAppointmentList();
    }
    if(!strcmp(algorithm, "-fcfs"))
        summary = Schedual_FCFS(inputList);
    else if(!strcmp(algorithm, "-prio"))
        summary = Schedual_PRIO(inputList);
    else if(!strcmp(algorithm, "-opti"))
        summary = Schedual_OPTI(inputList);
    else
    {
        printf("Unknown scheduler.\n");
        return;
    }
    PrintAllUser();
    PrintSummary(summary);
#else
    struct Summary *summary;
    int fd[2];
    if (pipe(fd) < 0) {
        printf("Pipe creation error\n");
        exit(EXIT_FAILURE);
    }

    int ret=fork();
    if (ret < 0)
    {
        printf("error in fork!");
        exit(EXIT_FAILURE);
    }
    else if (ret == 0) {    //Child
        //TODO: remove items from list
        for (int i=0; i<NumOfUser; i++)
        {
            user[i].accepted = CreateAppointmentList();
            user[i].rejected = CreateAppointmentList();
        }
        if(!strcmp(algorithm, "-fcfs"))
            summary = Schedual_FCFS(inputList);
        else if(!strcmp(algorithm, "-prio"))
            summary = Schedual_PRIO(inputList);
        else if(!strcmp(algorithm, "-opti"))
            summary = Schedual_OPTI(inputList);
        else
        {
            printf("Unknown scheduler.\n");
            return;
        }
        PrintAllUser();
        if(write(fd[1], summary, sizeof(struct Summary)) < 0)
            printf("Oh dear, something went wrong with write()! %s\n", strerror(errno));
        _exit(EXIT_SUCCESS);
    }

    summary = (struct Summary *)malloc(sizeof(struct Summary));
    if(read(fd[0], summary, sizeof(struct Summary)) < 0)
            printf("Oh dear, something went wrong with read()! %s\n", strerror(errno));
    wait(NULL);
    PrintSummary(summary);
#endif
}

void HandleInput(const char *line)
{
    char command[25];
    char caller[MAX_USERNAME];
    int year, month, day;
    int hour, minutes;
    float duration;
    int callee_count = 0;
    char *pch;

    struct Appointment *item = CreateAppointment();
    char myLine[255];
    strcpy(myLine, line);

    //parse the command
    pch = strtok(myLine, " \n");
    if(!pch)
        goto UNKNOWN;   //eg. strtok("\n", " \n") will return null from strtok, goto unknown command
    strcpy(command, pch);
    if(!strcmp(command, "addStudy"))
        item->type = STUDY;
    else if(!strcmp(command, "addAssignment"))
        item->type = ASSIGNMENT;
    else if(!strcmp(command, "addProject"))
        item->type = PROJECT;
    else if(!strcmp(command, "addGathering"))
        item->type = GATHERING;
    else if(!strcmp(command, "addBatch"))
    {
        pch = strtok(NULL, " \n");
        char filename[255];
        strcpy(filename, pch);
        FILE *f = fopen(filename+1, "r");   //offset +1 to remove the '-'
        if(!f)
        {
            fprintf(stderr, "Failed to open file %s.\n", filename);
            return;
            // exit(EXIT_FAILURE);
        }
        inputLoop(f);
        return;
    }
    else if(!strcmp(command, "printSchd"))
    {
        pch = strtok(NULL, " \n");
        char algorithmStr[30];
        strcpy(algorithmStr, pch);
        HandleSchedule(algorithmStr);
        return;
    }
    else if(!strcmp(command, "endProgram"))
    {
        printf("Received end program command.\n");
        exit(EXIT_SUCCESS);
    }
    else
    {
        UNKNOWN:
        printf("Unknown command: %s\n", line);
        return;
    }

    pch = strtok(NULL, " \n");
    strcpy(caller, pch+1);

    pch = strtok(NULL, " \n");
    sscanf(pch, "%d-%d-%d", &year, &month, &day);

    pch = strtok(NULL, " \n");
    sscanf(pch, "%d:%d", &hour, &minutes);

    pch = strtok(NULL, " \n"); duration = atof(pch);

    while(1)
    {
        pch = strtok(NULL, " \n");
        if(!pch)
            break;
        int id = GetUserID(pch);
        if(id==-1)
        {
            printf("->[Rejected: Unknown callee %s]\n", pch);
            return;
        }
        item->callee_id[callee_count++] = id;
    }

    item->caller_id = GetUserID(caller);
    if(item->caller_id==-1)
    {
        printf("->[Rejected: Unknown caller %s]\n", caller);
        return;
    }
    //time
    struct tm timeinfo, timeinfo_tmp;
    memset(&timeinfo, 0, sizeof(timeinfo));
    timeinfo.tm_isdst = -1;
    timeinfo.tm_year = year - 1900;
    timeinfo.tm_mon = month - 1;
```

```c
    timeinfo.tm_mday = day;
    //start time
    //convert ot half hour base
    timeinfo.tm_hour = hour;
    if(minutes>=0 && minutes <= 30)
        timeinfo.tm_min = 0;
    else
        timeinfo.tm_min = 30;

    timeinfo_tmp = timeinfo;    //because mktime could modify the value
    item->start = mktime(&timeinfo_tmp);
    //convert duration to end time
    double _;
    double fractional = modf(duration, &_);
    minutes += fractional*60;
    hour = hour+(int)duration;
    if(minutes>=60)
        hour++;
    minutes %= 60;

    timeinfo.tm_hour = hour;
    if(minutes>0 && minutes <= 30)
        timeinfo.tm_min = 0;
    else
        timeinfo.tm_min = 30;
    item->end = mktime(&timeinfo);

    item->id = inputList->count;
    AddAppointment(inputList, item);
    printf("-> [Pending]\n");
}


void inputLoop(FILE *stream)
{
    const int MAX_CHAR = 255;
    char line[MAX_CHAR];
    char *return_val;
    while(1)
    {
        printf("Please enter appointment:\n");
        return_val = fgets(line, MAX_CHAR, stream);
        if(!return_val)
        {
            if(feof(stream))
            {
                printf("Received EOF.\n");
                return;
            }
            else
            {
                fprintf(stderr, "IO error, existing program.\n");
                return;
                // exit(EXIT_FAILURE);
            }
        }
        HandleInput(line);
    }
}

int main(int argc, char* argv[])
{
    if (argc < 4 || argc > 11)
    {
        fprintf(stderr, "Error: The number of users should between 3 and 10.\n");
        return EXIT_FAILURE;
    }
    NumOfUser = argc - 1;
    //Initialize each user in struct user[];
    for (int i=0; i<NumOfUser; i++)
    {
        if(GetUserID(argv[i+1]) != -1)
        {
            printf("Duplicate names of users!\n");
            exit(EXIT_FAILURE);
        }
        strcpy(user[i].username, argv[i+1]);
        user[i].username[0] = toupper(user[i].username[0]);
        user[i].accepted = CreateAppointmentList();
        user[i].rejected = CreateAppointmentList();
    }
    inputList = CreateAppointmentList();

    printf("~~WELCOME TO AMR~~\n");
    inputLoop(stdin);
    return EXIT_SUCCESS;
}
```

5.7 scheduler.c File Reference

Secheduling algorithms.

```
#include "appointment_list.h"
#include "user.h"
#include <unistd.h>
#include "scheduler.h"
```

Functions

- struct **Summary** ∗ **Schedual_FCFS** (struct **AppointmentList** ∗inputList)

  First come first served. The order is folloing the input order. The result will be putted into the each user's appointment lists (accept / reject).
- struct **Summary** ∗ **Schedual_PRIO** (struct **AppointmentList** ∗inputList)

  Priority. The order is folloing the pre-defined priority. The result will be putted into the each user's appointment lists (accept / reject).
- struct **Summary** ∗ **Schedual_OPTI** (struct **AppointmentList** ∗inputList)

  Optimized. Bonus part, reschedule those rejected appointments. The result will be putted into the each user's appointment lists (accept / reject).
- void **PrintSummary** (struct **Summary** ∗summary)

  Print out the summary about the scheduling.

5.7.1 Detailed Description

Secheduling algorithms.

Author

oneonestar `oneonestar@gmail.com`

Version

1.0

Copyright

2015

5.7.2 LICENSE

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see `http://www.gnu.org/licenses/`.

Definition in file **scheduler.c**.

5.8 scheduler.c

```

#include "appointment_list.h"
#include "user.h"

#include <unistd.h>

#include "scheduler.h"

static int AllAvailable(const struct Appointment *item)
{
    if(!item)
        return 0;
    struct AppointmentList *temp_list;
    //check caller timetable
    temp_list = ConflictInList(user[item->caller_id].accepted, item);
    if(temp_list->count)
        return item->caller_id;
    //check callees timetable
    for(int i=0; i<USER_NUMBER; i++)
    {
        if(item->callee_id[i]==-1)
            break;
        temp_list = ConflictInList(user[item->callee_id[i]].accepted, item);
        if(temp_list->count)
            return item->callee_id[i];
    }
    return -1;
}

static int AllAvailablePriority(const struct Appointment *item)
{
    struct AppointmentList *temp_list;
    struct Appointment *ptr;
    //check caller timetable
    temp_list = ConflictInList(user[item->caller_id].accepted, item);
    ptr = temp_list->head;
    while(ptr)
    {
        if(CompareAppointmentPriority(item, ptr)>=0)     //if item has equal or lower priority
            return item->caller_id; //not available
        ptr = ptr->next;
    }
    //check callees timetable
    for(int i=0; i<USER_NUMBER; i++)
    {
        if(item->callee_id[i]==-1)
            break;
        //TODO: may need refactoring
        temp_list = ConflictInList(user[item->callee_id[i]].accepted, item);
        ptr = temp_list->head;
        while(ptr)
        {
            if(CompareAppointmentPriority(item, ptr)>=0)     //if item has equal or lower priority
                return item->callee_id[i];  //not available
            ptr = ptr->next;
        }
    }
    return -1;  //available
}

static void AddToAllAccept(struct Appointment *item)
{
    strcpy(item->reason, "");
    AddAppointmentOrdered(user[item->caller_id].accepted, item);
    for(int i=0; i<USER_NUMBER; i++)
    {
        if(item->callee_id[i]==-1)
            break;
        AddAppointmentOrdered(user[item->callee_id[i]].accepted, item);
    }
}

static void AddToAllAcceptForced(struct Appointment *item)
{
    struct AppointmentList *temp_list;
    strcpy(item->reason, "");
    //caller
    //delete old appointments from accepted list
    temp_list = ConflictInList(user[item->caller_id].accepted, item);
    RemoveListFromList(user[item->caller_id].accepted, temp_list);
    //add to accept and reject lsit
    AddAppointmentOrdered(user[item->caller_id].accepted, item);
    SetReasonForList(temp_list, "Higher priority item being added.");
    AddAppointmentOrderedFromList(user[item->caller_id].rejected, temp_list);
```

```c
    for(int i=0; i<USER_NUMBER; i++)
    {
        if(item->callee_id[i]==-1)
            break;
        //delete old appointments from accepted list
        temp_list = ConflictInList(user[item->callee_id[i]].accepted, item);
        RemoveListFromList(user[item->callee_id[i]].accepted, temp_list);
        //add to accept and reject lsit
        SetReasonForList(temp_list, "Higher priority item being added.");
        AddAppointmentOrdered(user[item->callee_id[i]].accepted, item);
        AddAppointmentOrderedFromList(user[item->callee_id[i]].rejected, temp_list);
    }
}

static void AddToAllReject(const struct Appointment *item)
{
    AddAppointmentOrdered(user[item->caller_id].rejected, item);
    for(int i=0; i<USER_NUMBER; i++)
    {
        if(item->callee_id[i]==-1)
            break;
        AddAppointmentOrdered(user[item->callee_id[i]].rejected, item);
    }
}

static void SetAppointmentAccepted(struct AppointmentList *inputList)
{
    for(int i=0; i<NumOfUser; i++)
    {
        struct Appointment *ptr = user[i].accepted->head;
        while(ptr)
        {
            GetAppointmentById(inputList, ptr->id)->is_accepted = 1;
            ptr = ptr->next;
        }
    }
}

static struct AppointmentList* GetEmptyTimeSlotInDay(struct AppointmentList *userList, time_t date)
{
    struct AppointmentList *empty_list = CreateAppointmentList();
    //18:00-22:00
    struct tm timeinfo, timeinfo_tmp;
    timeinfo = *localtime (&date);

    //foreach 18:00-22:00 half hour timeslot
    timeinfo.tm_hour = 18;
    while(timeinfo.tm_hour<22)
    {
        struct Appointment *item = CreateAppointment();
        //first half hour
        timeinfo_tmp = timeinfo;    //because mktime could modify the value
        item->start = mktime(&timeinfo_tmp);
        timeinfo.tm_min = 30;
        timeinfo_tmp = timeinfo;    //because mktime could modify the value
        item->end = mktime(&timeinfo_tmp);
        if(!IsConflictInList(userList, item))
            AddAppointmentOrdered(empty_list, item);


        //second half hour
        timeinfo_tmp = timeinfo;    //because mktime could modify the value
        item->start = mktime(&timeinfo_tmp);
        timeinfo.tm_hour++;
        timeinfo.tm_min = 0;
        timeinfo_tmp = timeinfo;    //because mktime could modify the value
        item->end = mktime(&timeinfo_tmp);
        if(!IsConflictInList(userList, item))
            AddAppointmentOrdered(empty_list, item);
    }
    return empty_list;
}

static struct AppointmentList* GetEmptyTimeSlotInRange(struct AppointmentList *userList, time_t start_date, time_t end_date)
{
    struct AppointmentList *empty_list = CreateAppointmentList();
    struct tm timeinfo, timeinfo_tmp;

    //set the time start<end so that we can use difftime() to compare the date.
    timeinfo = *localtime (&start_date);
    timeinfo.tm_hour = 1;
    timeinfo_tmp = timeinfo;
    start_date = mktime(&timeinfo_tmp);

    timeinfo = *localtime (&end_date);
    timeinfo.tm_hour = 2;
    timeinfo_tmp = timeinfo;
    end_date = mktime(&timeinfo_tmp);

    timeinfo = *localtime (&start_date);
    while(difftime(start_date, end_date)<0)
    {
        AddAppointmentFromList(empty_list, GetEmptyTimeSlotInDay(userList, start_date));

        timeinfo.tm_mday++;
        timeinfo_tmp = timeinfo;
        start_date = mktime(&timeinfo_tmp);
    }
    return empty_list;
}

static time_t GetEarliestStartTime(struct AppointmentList *list)
{
    struct Appointment *item = list->head;
    time_t earliest = item->start;
    while(item)
    {
        if(difftime(item->start, earliest)<0)
            earliest = item->start;
        item = item->next;
    }
    return earliest;
}

static time_t GetLatestEndTime(struct AppointmentList *list)
{
    struct Appointment *item = list->head;
    time_t latest = item->end;
    while(item)
    {
        if(difftime(latest, item->end)<0)
            latest = item->end;
        item = item->next;
    }
    return latest;
}

static struct AppointmentList* GetContinueTimeslotFromList(const struct
AppointmentList *list, time_t duration)
{
    struct Appointment *ptr;
    struct Appointment *item = list->head;
    struct AppointmentList *ret_list = CreateAppointmentList();
    int timeslot = duration / 60 / 30 - 1;  //how many half hour
    while(item)
    {
        ptr = item;
        for(int i=0; i<timeslot; i++)
            ptr = ptr->next;
        if(!ptr)
            return ret_list;
        if(difftime(ptr->end, item->start)==duration)
            AddAppointmentOrdered(ret_list, item);
        item = item->next;
    }
    return ret_list;
}

struct Summary* Schedual_FCFS(struct AppointmentList *inputList)
{
    struct Appointment *ptr = inputList->head;
    while(ptr)
    {
        int ret = AllAvailable(ptr);
        if(ret<0)
        {
            AddToAllAccept(ptr);
        }
        else
        {
            char reason[50];
            strcpy(reason, user[ret].username);
            strcat(reason, " is unavailable.");
            strcpy(ptr->reason, reason);
            AddToAllReject(ptr);
        }
        ptr = ptr->next;
    }

    //Summary
    struct Summary *summary = (struct Summary *)malloc(sizeof(struct Summary));
    summary->start = GetEarliestStartTime(inputList);
    summary->end = GetLatestEndTime(inputList);

    SetAppointmentAccepted(inputList);
    ptr = inputList->head;
    while(ptr)
    {
        if(ptr->is_accepted)
            summary->total_accepted++;
        else
            summary->total_rejected++;
        ptr = ptr->next;
    }
    for(int i=0; i<NumOfUser; i++)
    {
        summary->accepted[i] = user[i].accepted->count;
        summary->rejected[i] = user[i].rejected->count;
        summary->empty_timeslot[i] = GetEmptyTimeSlotInRange(user[i].accepted, summary->start, summary->end)->count;

        // PrintAppointmentList(GetContinueTimeslotFromList(GetEmptyTimeSlotInRange(user[i].accepted, summary->start, summary->end), 2*60*30));
        // PrintAppointmentList(GetEmptyTimeSlotInRange(user[i].accepted, summary->start, summary->end));
    }
    return summary;
}

struct Summary* Schedual_PRIO(struct AppointmentList *inputList)
{
    struct Appointment *ptr = inputList->head;
    while(ptr)
    {
        int ret = AllAvailablePriority(ptr);
        if(ret<0)
        {
            AddToAllAcceptForced(ptr);
        }
        else
        {
            char reason[50];
            strcpy(reason, user[ret].username);
            strcat(reason, " is unavailable.");
            strcpy(ptr->reason, reason);
            AddToAllReject(ptr);
        }
        ptr = ptr->next;
    }

    //Summary
    struct Summary *summary = (struct Summary *)malloc(sizeof(struct Summary));
    summary->start = GetEarliestStartTime(inputList);
    summary->end = GetLatestEndTime(inputList);

    SetAppointmentAccepted(inputList);
    ptr = inputList->head;
    while(ptr)
    {
        if(ptr->is_accepted)
            summary->total_accepted++;
        else
            summary->total_rejected++;
        ptr = ptr->next;
    }
    for(int i=0; i<NumOfUser; i++)
    {
        summary->accepted[i] = user[i].accepted->count;
        summary->rejected[i] = user[i].rejected->count;
        summary->empty_timeslot[i] = GetEmptyTimeSlotInRange(user[i].accepted, summary->start, summary->end)->count;
        // PrintAppointmentList(GetEmptyTimeSlotInRange(user[i].accepted, summary->start, summary->end));
    }
    return summary;
}

struct Summary* Schedual_OPTI(struct AppointmentList *inputList)
{
    struct Summary *summary = Schedual_PRIO(inputList);
    time_t day = 60*60*24;

    //For each user, try to reschedule their rejected jobs
    for(int i=0; i<NumOfUser; i++)
    {
        struct Appointment *item = user[i].rejected->head;
        NEXT_ITEM:
        while(item)
        {
            time_t ori_start = item->start;
            time_t ori_end = item->end;
            time_t duration = difftime(item->end, item->start);
```

```c
            struct AppointmentList *list = GetEmptyTimeSlotInRange(user[i].accepted, item->start, item->start+3*day);
            struct AppointmentList *c_list = GetContinueTimeslotFromList(list, duration);

            struct Appointment *timeslot = c_list->head;
            while(timeslot)
            {
                item->start = timeslot->start;
                item->end = item->start + duration;

                int ret = AllAvailable(item);
                if(ret<0)
                {
                    item->rescheduled = 1;
                    AddToAllAccept(item);
                    struct Appointment *temp = item;
                    item = item->next;
                    for(int j=0; j<NumOfUser; j++)
                        RemoveItemFromList(user[j].rejected, temp);
                    goto NEXT_ITEM;
                }
                strcpy(item->reason, "No available timeslot for the reschedule.");
                item->start = ori_start;
                item->end = ori_end;
                timeslot = timeslot->next;
            }
            item = item->next;
        }
    }

    //Re-calculate the summary
    memset(summary, 0, sizeof(struct Summary));
    summary->start = GetEarliestStartTime(inputList);
    summary->end = GetLatestEndTime(inputList);
    SetAppointmentAccepted(inputList);
    struct Appointment *ptr = inputList->head;
    while(ptr)
    {
        if(ptr->is_accepted)
            summary->total_accepted++;
        else
            summary->total_rejected++;
        ptr = ptr->next;
    }
    for(int i=0; i<NumOfUser; i++)
    {
        summary->accepted[i] = user[i].accepted->count;
        summary->rejected[i] = user[i].rejected->count;
        summary->empty_timeslot[i] = GetEmptyTimeSlotInRange(user[i].accepted, summary->start, summary->end)->count;
        // PrintAppointmentList(GetEmptyTimeSlotInRange(user[i].accepted, summary->start, summary->end));
    }
    return summary;
}

static int rdn(int y, int m, int d) {
    if (m < 3)
        y--, m += 12;
    return 365*y + y/4 - y/100 + y/400 + (153*m - 457)/5 + d - 306;
}

void PrintSummary(struct Summary *summary)
{
    float total = summary->total_accepted+summary->total_rejected;
    struct tm timeinfo, timeinfo2;
    int days;

    printf("Performance:\n");
    timeinfo = *localtime(&summary->start);
    printf("Date start: %4d-%02d-%02d\n", timeinfo.tm_year+1900, timeinfo.tm_mon+1, timeinfo.tm_mday);
    timeinfo2 = *localtime(&summary->end);
    printf("Date end: %4d-%02d-%02d\n\n", timeinfo2.tm_year+1900, timeinfo2.tm_mon+1, timeinfo2.tm_mday);

    printf("Total Number of Appointment Assigned: %d (%.1f%%)\n", summary->total_accepted, summary->total_accepted/total*100);
    printf("Total Number of Appointment Rejected: %d (%.1f%%)\n", summary->total_rejected, summary->total_rejected/total*100);

    days = rdn(timeinfo2.tm_year, timeinfo2.tm_mon+1, timeinfo2.tm_mday) - rdn(timeinfo.tm_year, timeinfo.tm_mon+1, timeinfo.tm_mday) + 1;
    printf("Utilization of Time Slot: (%d days)\n", days+1);
    for(int i=0; i<NumOfUser; i++)
    {
        printf("    %-10s - %.1f%%\n", user[i].username, (days*2*4-(float)summary->empty_timeslot[i])/(days*2*4)*100);  //each day have 2*8 timeslots
    }

```

```
00442 }
```

5.9 scheduler.h File Reference

Secheduling algorithms.

```
#include "user.h"
#include <time.h>
```

Data Structures

- struct **Summary**

Functions

- struct **Summary** ∗ **Schedual_FCFS** (struct **AppointmentList** ∗inputList)

  First come first served. The order is folloing the input order. The result will be putted into the each user's appointment lists (accept / reject).
- struct **Summary** ∗ **Schedual_PRIO** (struct **AppointmentList** ∗inputList)

  Priority. The order is folloing the pre-defined priority. The result will be putted into the each user's appointment lists (accept / reject).
- struct **Summary** ∗ **Schedual_OPTI** (struct **AppointmentList** ∗inputList)

  Optimized. Bonus part, reschedule those rejected appointments. The result will be putted into the each user's appointment lists (accept / reject).
- void **PrintSummary** (struct **Summary** ∗summary)

  Print out the summary about the scheduling.

5.9.1 Detailed Description

Secheduling algorithms.

Author

oneonestar `oneonestar@gmail.com`

Version

1.0

Copyright

2015

5.9.2 LICENSE

Definition in file **scheduler.h**.

5.10 scheduler.h

```

#ifndef SCHEDULER
#define SCHEDULER

#include "user.h"
#include <time.h>

struct Summary
{
    int total_accepted;
    int total_rejected;
    int accepted[USER_NUMBER];
    int rejected[USER_NUMBER];
    int empty_timeslot[USER_NUMBER];
    time_t start;
    time_t end;
};
struct Summary* Schedual_FCFS(struct AppointmentList *inputList);

struct Summary* Schedual_PRIO(struct AppointmentList *inputList);

struct Summary* Schedual_OPTI(struct AppointmentList *inputList);

void PrintSummary(struct Summary *summary);

#endif
```

5.11 user.c File Reference

Handling each users.

```
#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include "user.h"
#include "appointment_list.h"
```

Functions

- int **strcicmp** (char const ∗a, char const ∗b)
- int **GetUserID** (const char ∗username)

  Return the user id that have the same username.
- void **PrintAccepted** (const struct **User** ∗**user**)

  Print the accepted list for user.
- void **PrintRejected** (const struct **User** ∗**user**)

  Print the rejected list for user.
- void **PrintAllUser** ()

  Print the accepted & reject list for all users.

5.11.1 Detailed Description

Handling each users.

Author

oneonestar `oneonestar@gmail.com`

Version

1.0

Copyright

2015

5.11.2 LICENSE

Definition in file **user.c**.

5.11.3 Function Documentation

5.11.3.1 int GetUserID (const char ∗ *username*)

Return the user id that have the same username.

Parameters

username	The username that are used to compare.

Returns

The user id that have the same username. Return -1 if user is not found.

Definition at line **38** of file **user.c**.

5.12 user.c

```
00001 
00022 #include <string.h>
00023 #include <ctype.h>
00024 #include <stdio.h>
00025 
00026 #include "user.h"
00027 #include "appointment_list.h"
00028 
00029 int strcicmp(char const *a, char const *b)
00030 {
00031     for (;; a++, b++) {
00032         int d = tolower(*a) - tolower(*b);
00033         if (d != 0 || !*a)
00034             return d;
00035     }
00036 }
00037 
00038 int GetUserID(const char *username)
00039 {
00040     for(int i=0; i<NumOfUser; i++)
00041         if (!strcicmp(user[i].username, username))
00042             return i;
00043     return -1;
00044 }
00045 
00046 
00047 
00048 void PrintAccepted(const struct User *user)
00049 {
    printf("%s, you have %d appointments.\n", user->username, user->accepted->count);
    PrintAppointmentList(user->accepted);
}



void PrintRejected(const struct User *user)
{
    printf("%s, you have %d appointments rejected.\n", user->username, user->rejected->count);
    PrintAppointmentList(user->rejected);
}


void PrintAllUser()
{
    printf("***Appointment Schedule - ACCEPTED ***\n\n");
    for(int i=0; i<NumOfUser; i++)
    {
        PrintAccepted(&user[i]);
    }
    printf("   -End-\n");
    printf("==========================================================================\n");
    printf("***Appointment Schedule - REJECTED ***\n\n");
    for(int i=0; i<NumOfUser; i++)
    { 00075  PrintRejected(&user[i]);
    }
    printf("   -End-\n");
    printf("==========================================================================\n");
}
```

5.13 user.h File Reference

Handling each users.

Data Structures

- struct **User**

  Store the basic information of the user and the appointments.

Macros

- #define **USER_NUMBER** 10
- #define **MAX_USERNAME** 31

Functions

- int **GetUserID** (const char *username)

  Return the user id that have the same username.
- void **PrintAccepted** (const struct **User** ***user**)

  Print the accepted list for user.
- void **PrintRejected** (const struct **User** ***user**)

  Print the rejected list for user.
- void **PrintAllUser** ()

  Print the accepted & reject list for all users.

Variables

- struct **User user** [**USER_NUMBER**]
- int **NumOfUser**

5.13.1 Detailed Description

Handling each users.

Author

oneonestar `oneonestar@gmail.com`

Version

1.0

Copyright

2015

5.13.2 LICENSE

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see `http://www.gnu.org/licenses/`.

Definition in file **user.h**.

5.13.3 Macro Definition Documentation

5.13.3.1 #define MAX_USERNAME 31

Maximum length of the username, 30 for the name and 1 for the null.

Definition at line **34** of file **user.h**.

5.13.3.2 #define USER_NUMBER 10

The number of users is from 3 to 10, so maximum is 10.

Definition at line **28** of file **user.h**.

5.13.4 Function Documentation

5.13.4.1 int GetUserID (const char * *username*)

Return the user id that have the same username.

Parameters

username	The username that are used to compare.

Returns

The user id that have the same username. Return -1 if user is not found.

Definition at line **38** of file **user.c**.

5.13.5 Variable Documentation

5.13.5.1 int NumOfUser

Global variable storing current number of user.

Definition at line **55** of file **user.h**.

5.13.5.2 struct User user[USER_NUMBER]

Global variable storing the user data.

Definition at line **50** of file **user.h**.

5.14 user.h

```
00001
00022 #ifndef USER
00023 #define USER
00024
00028 #define USER_NUMBER 10
00029
00034 #define MAX_USERNAME 31
00035
00040 struct User
00041 {
00042     char username[MAX_USERNAME];
00043     struct AppointmentList *accepted;
00044     struct AppointmentList *rejected;
00045 };
00046
00050 struct User user[USER_NUMBER];
00051
00055 int NumOfUser;
00056
00062 int GetUserID(const char *username);
00063
00067 void PrintAccepted(const struct User *user);
00068
00072 void PrintRejected(const struct User *user);
00073
00077 void PrintAllUser();
00078
00079 #endif
```

Index