COMP304 Group Project
Appointment Organizer System
Semester 2 Mid 2013

Chien, Wei Der, Steven
Chu, Hoi Man, Veronica
Chan, Wai Yan, Flora
Fung, Yuen Ki, Yuki
Ng, Yan, Elena

1)Introduction

The objective of the software is to provide easy scheduling for events involving multiple persons. Users of the software can input their occupied time and time of events which involves multiple persons. The system will report if users involved are all available and if the appointment is accepted or rejected.

2)Scope

The project involves concept in scheduling and processes.

Scheduling in the program refers to the handling request of events. The well know scheduling algorithm includes but not limited to first come first serve, priority, priority with preemption, shortest job first, shortest remaining time, round robin. In the context of scheduling for human being, first come first serve, priority and priority with preemption makes the most sense. First come first serve will be the choice of algorithm in this project.

Multiple processes are involved in this project where each process act as a particular user and will be able to handle scheduling for the particular user and communicate with the main program.

3)Concept

First come first serve algorithm counts importance of jobs according to arrival time. The importance of job are assigned according to their arrival time. In the context of this software, importance will be classified to 1 and 0 where 1 has a higher priority than 0. Consider any particular time period, a time period can only hold one event since multitasking of different events does not make sense in the real world.
Unlike computers, jobs cannot be accumulated to be processed in a first in first out queue since all events are specified with starting and ending time. In this context, the particular time slot will be first come first served, meaning that the slot will entertain events recorded earlier and assigned importance 1 and subsequent events will be regarded as 0 and be rejected.

Besides, this program adopts multi-process mechanism. For each user joining the system, a child process will be created for each and every user. With the use of *fork()* and *pipe()*, multiple processes are created. Child processes can communicate with parent process through reading from and writing to pipes. Parent will give instructions to child processes. Child processes only proceed with actions when they are instructed by the parent. Multiple processes are run simultainously.

4)Structure of the program

4.1)Data structures

The software employs various types of data structure in the implementation. There are mainly 3 types of data which need to be stored. Data or user, details of events and sets of pipes.

With regard to users' data, each user's data will be stored in a structure called 'User'. Elements of User includes name of user: char name[10], where the first letter is capitalized and the maximum letters allowed is 20; process ID: pid_t process_id for the respective user; pointer to beginning and ending of the appointment list: event *start, *end; pointer to the beginning and ending of the rejection list: event* rej_start, *rej_end(will be explained in greater detail later); number of accepted appointments and rejected appointments; finally the id of child process: int id, starting from 0.

With regard to event's data, each event will be stored in a structure called 'Event'. Elements of Event includes the nature of the event: priority, denoted with 1, 2, 3, referring to gathering, meeting and class respectively; starting and ending time of the event: time_t start_time, end_time; name of users involved if of nature 1 or 2: char usr[10][10], with maximum 10 users who have names within 10 character; parti_no stores the number of participants for a particular event; pointer to the next event: event *next. Finally pipes will be declared as an array to facilitate development.

4.2)Explanation of data structure used

Name of users are important data even after submission which users will be represented as IDs, because they will be involved and recognized from commands given by the user at a later stage. Process ID is needed to in order to differentiate child processes with main program. Event *start and *end is the head and tail of a link list of events for that particular user. Event *rej_start, *rej_end follows the same principle. Number of appointments and rejected events need to be stored to keep track of the size of the two link lists and facilitate printing. Int id is used to store the representation of a particular process/user in the program. The id correspond to the particular pipe to avoid confusion.

For events, int priority is used to keep track of nature of event mainly to facilitate printing of result. time_t start_time, end_time are used to store the timing of event in the form of seconds since epoch. Duration of events need not be stored since it can be represented as the difference between end_time and start_time. Char usr[10][10] is used to store participants if events are of nature 1, 2. int parti_no is used to facilitate printing of report and event *next is the pointer pointing to next event, which will be pointing to NULL if it is already the last on the list.

time_t is an integral value holding the number of seconds since Epoch, 00:00, Jan 1 1970 UTC

Pipes used to communicate between processes are declared as an array for easy processing and identification. The index of a pipe corresponds to the id of a particular user/process.

5)Mechanism of the program

The software operates as a number of separated processes where all processes have their own copy of variables. The main program will be responsible to instruct child processes to do certain operations. When the program starts, it will reproduce a number of process according to the number of persons requested by the user.

5.1)Initialization

The program will store the persons' basic information for the child process' use as an array structure. The program also record down its own PID to differentiate itself with other processes. Next, the main program will create a number of pipes before ready to create child processes. Two sets of pipes will be created namely P2C and C2P. P2C will handle communication initiated by parent whereas C2P will handle communication initiated by child to avoid confusion and facilitate software development. The

pipes will be in a set of array indexed by their respective process as explained earlier. When all the pipes are properly created, the main program will start running *fork()* system call in a loop to create processed requested by the users. Within the loop, process ID and id are given and stored in their own structure also indexed by 'id'.
After running fork(), the program will check whether the system call returns 0, which represents that the process checking it is a child process. If so, the program will execute *child(user*, int)* function which defines the behavior of a sub process. The main program will keep running *fork()* until the loop terminates. The parent program will then cross check its PID with the PID recorded earlier to determine if it is a parent. If so, it will run *parent(usr*)* function which defines the behavior of parent.

5.2)Parent's behavior

The parent will define a set of variables and enter an infinite loop. The parent will then invite user inputs. User inputs will be analyzed token by token with string tokenizer provided by string.h. When strtok() is invoked, the program will analyze the first token to determine whether it is to add an event, print a report or end the program. When the instruction is known, the parent can proceed to its respective case.

5.3)Child's behavior

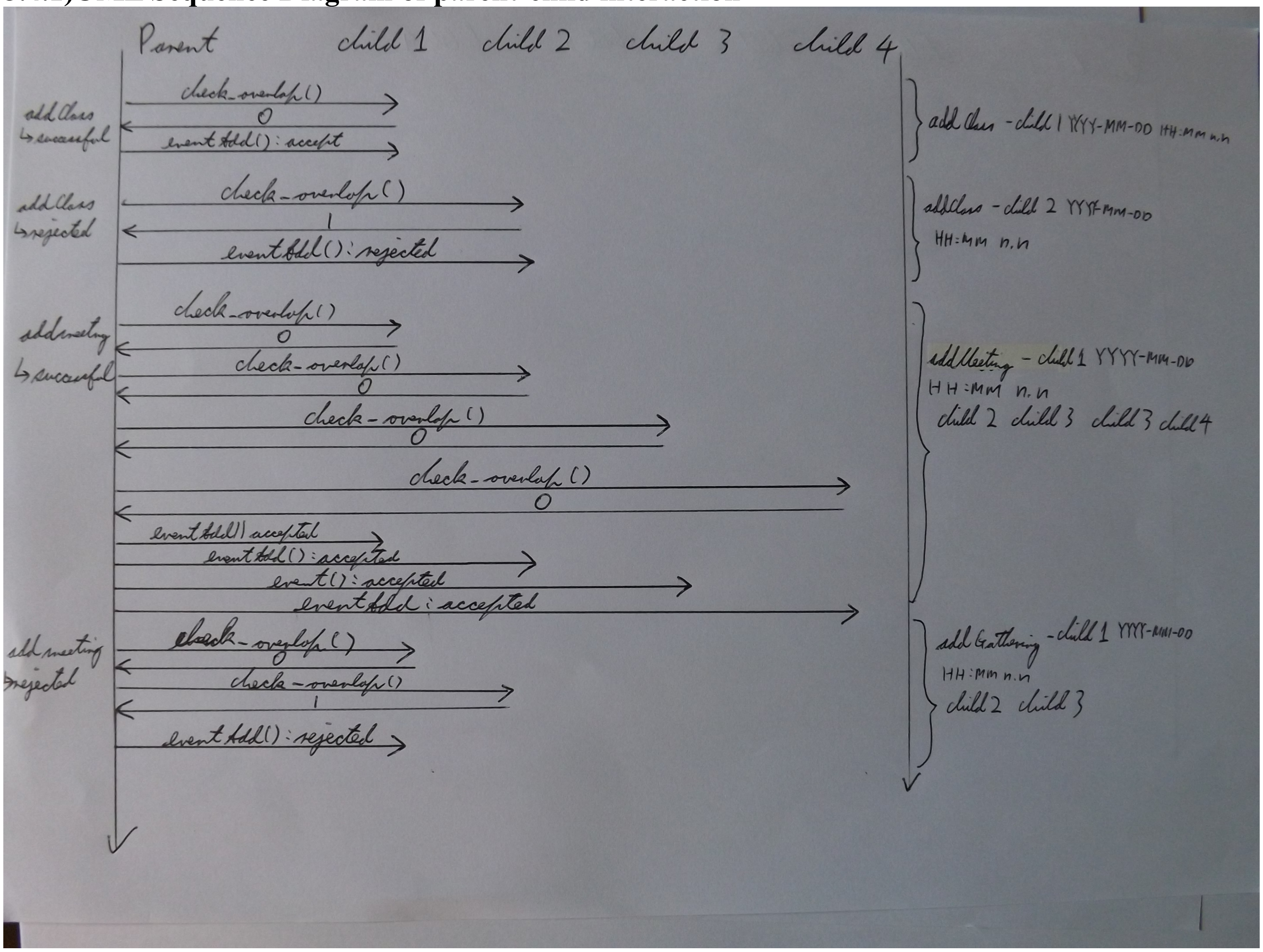
Like the parent, a child will define a set of variables. Additionally it will close all pipes that does not belongs to it. This is done through closing all pipes except the one that matches its process ID. Finally two pointers for P2C and C2P, named as this_P2C and this_C2P will be used for easier programming.

Unlike the parent, children will only passively operate. It will only start working AFTER instruction is given by parent. If not instruction is given by parent, it will stay idle.

The program will enter an infinite loop and wait for instruction from parent. The child will go to different cases according to instruction received from parent.

5.4)Analysis of instructions

5.4.1)UML Sequence Diagram of parent-child interaction



5.4.2)event adding

In order to add an event, user will enter command in the form of “add(event) -(caller) YYYY-MM-DD HH:MM n.n (callees)”. (event) refers to Class, Meeting or Gathering. Meeting and Gathering involves multiple participants. YYYYYY-MM-DD refers to the date of event, HH:MM refers to starting time of the event in 24 hour format. n.n refers to duration of event in decimal.

When user enters a command to add a class, parent will engage in case 0. The parent will read the next token to obtain the name of the caller in order to obtain his id by comparing the name to data stored in the structure. The parent will then ask if the caller if he is available. This is done by sending instruction 0 followed by the command given by user. The child receives instruction 0 from parent and proceed to case 0 which is checking if time slot requested is available. It will create a temporary event to store the starting and ending time. It calls *parse_time(char*, time_t, time_t)* to analysis the starting and ending time from the command. Afterward, it will *check_overlap(user*, time_t, time_t)* with the start and end time obtained earlier to see if there is a collision.

parse_time(char, time_t, time_t)* returns start and end time in time_t by analyzing the command by extracting the time data one by one and store them in a structure tm. When everything settles, the function will call *mktime(struct tm*)* to convert the structure to time_t format.

check_overlap(user, time_t, time_t)* operates in that it check
1. if the list is empty;
2. if the time slot needed is in the middle of the list;
3. if the slot is beyond the list.
4. 1 is returned if there are collision, 0 if there are collision.

check_overlap(user, time_t, time_t)* constitutes the scheduling module.

Importance of event is classified as 1 and 0 where events of class 1 will be entertained. The child will send result 1 or 0 to parent through the pipe where 0 means collision not true; 1 means collision true. When parent receives the result, it will determine if the event is to be entertained.

If the adding of class is to be entertained, it send instruction 1 to the child, and again the command from user. The child after receiving instruction 1, will proceed to add a class. *eventAdd(user*, char*, int)* will be called. *eventAdd(user*, char*, int)* will create a new event, get the starting and ending time and link it up with the list. It searches the right slot to add in, like *check_overlap(user*, time_t, time_t)*, it checks if the event is to be added to the top of the list, middle or end.

eventAdd(user, char*, int)* constitutes the "Input Module".

If event is not to be entertained, it will be added to the rejected list. The parent will send instruction 2 to the child, which it represents rejecting a class. When child receives the instruction, it will call *eventAdd(user*, char*, int)*, with flag indicating that the event is to be rejected. The function will analysis the command just like when the event is added to the event list but this time it will not care about the sequence and just add it to the end of the list.

When "addMeeting" or "addGathering" is called, the parent will enter **case 1** where events are added and rejected the same way as in "addClass". But this time the parent will read names of participants and get their ID to check if they are available. Parent will send instruction of time checking to children involved and wait for returns. If all child replies that they are available, the event will be added just like "addClass" as *eventAdd(user*, char*, int)* will know if the event involves multiple persons when reading the command. If not all the participants are available, it will be added to the rejected list of the caller. The parent will send rejection instruction 3 and record the first unavailable callee and send its name to the child as one unavailable callee is already enough to make the event a rejected one. Finally the parent sends the command to the caller. If all callees are available, the add instruction 1 and same set of data will be sent to all children.

5.4.3)Report printing

report(usr, char*, int)* will open a file stream with write privilege. The formatting data will be written to the file with *fprintf()*. The function will then start reading the link list depending the user requests appointment list or rejected list or both. The program will convert start_time and end_time to two time structures with *localtime()* which time data can be printed as strings in specified format with *strftime()*.

The program will read from link list and write to file stream date, time, nature of event and participants if any.

5.4.4)Termination

When user sends command "endProgram", parent will enter case 5. Instruction 5 will be sent to all active children. When child receives instruction 5, it will invoke *exit(0)* to terminate the process. When instructions are sent to all children, the parent will wait for a period of time to avoid creating zombie processes. Finally the parent will issue exit itself to terminate itself.

In case of unexpected termination of parent due to wrong user input for example, children will attempt to read from the broken pipe in an infinite loop, thus burning the CPU. A safety mechanism is implemented in the child that it will check if parent is still alive before reading from the pipe.

5.5)Description and behavior of functions

*void parse_time(char *command, time_t end_time, time_t start_time)*

Parameter:
1. *command*: character pointer to the command given by user
2. *end_time*: ending time in time_t;
3. *start_time*: starting time in time_t

Description:
It extracts time data from the command and fill the two time_t variables provided.
Warning! Any data stored in *start_time* and *end_time* will be destroyed and replaced

Return value: *NULL*

int check_overlap(user member, time_t end_time, time_t start_time)*

Parameter:
1. *member*: pointer to structure User
2. *end_time*: ending time in time_t
3. *start_time*: start time in time_t

Description:
Scheduling Module, examines if given time can be fit into the requested time slot

Return value: integer value *1* if there is collision, *0* if there is no collision

*void eventAdd(user *member, char *command, int flag)*

Parameter:
1. *member*: pointer to structure User
2. *command*: command given by user
3. *flag*: integer 0 representing rejected or 1 representing accepted event.

Description:
Input Module, add event to link list of accepted or rejected event in ascending order.

Return value: NULL

*void report(user *usr, char *output, int flag)*

Parameter:
1. *usr*: pointer to structure User
2. *output*: output file name in character array
3. *flag*: integer option 1 to print appointment list; 2 to print rejected list; 3 to print full report

Description:
Output Module, print report to files
Warning! File will the same file name in the directory where the program is executed will be overwritten without warning

Return value: NULL

Structure ***User***#:
1. char *name*[20];
2. pid_t *process_id*;
3. int *event_count*;
4. int *rejection_count*;
5. int *id*;
6. event **start*, **end*;
7. event **rej_start*, **rej_end*;

Structure ***Event*** #:
1. int *priority*;
2. time_t *start_time*;
3. time_t *end_time*;
4. char *usr*[20][20];
5. int *parti_no*;
6. struct Event **next*;

Structures are declared as definition with *typedef*.

Pipes:
P2C[10][2];
C2P[10][2];

C2P for child to parent communication;
P2C for parent to child communication.

Instruction list of child:
0: check if time slot is free
1: add class or meeting or gathering
2: reject a class
3: reject a meeting or gathering
4: print report
5: self termination

6)Testing

6.1)Initialization of the Software

[steven@localhost project]$ ./aos_debug victoria steven alfred star
DEBUG!: struct created
DEBUG!: main(): usr id 0, name Victoria
DEBUG!: main(): usr id 1, name Steven
DEBUG!: main(): usr id 2, name Alfred
DEBUG!: main(): usr id 3, name Star
DEBUG!: names copied
(add data initialized)

DEBUG!: main(): forking 0 child
DEBUG!: main(): forking 1 child
DEBUG!: main(): forking 2 child
DEBUG!: child: my ID is 0
DEBUG!: child: my ID is 1
DEBUG!: main(): forking 3 child
DEBUG!: child: my ID is 2
DEBUG!: main(): I am Parent
~Welcome to AOS~
please enter appointment:
DEBUG!: child: my ID is 3
(children created)

The debug codes revealed that names of users are correctly written and stored in the structure with first letter of names converted to capital letter. User id are also correctly assigned. This is reflected form the child's response telling their ID.

6.2)Adding of event to an empty list

addClass -victoria 2013-05-04 18:00 2.5
DEBUG!: parent: length of command is 39
DEBUG!: parent: command: addClass -victoria 2013-05-04 18:00 2.5
(command received)
DEBUG!: parent: option is 0
DEBUG!: parent: case 0
(command identified)
DEBUG!: parent: name of caller is Victoria
DEBUG!: parent: caller's id is 0
(caller identified)

It is shown from program execution that the parent process correctly resolved the command as a addClass command and proceeded into case 0.

DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 0
DEBUG!: child 0: read command: addClass -victoria 2013-05-04 18:00 2.5
DEBUG!: parent: instruction 0 sent to child
(instruction sent by parent and received by right child)

The above code reflected that the correct signal has been transmitted and received by the correct child, where Victoria's ID is 0.

DEBUG!: child 0: start time: 1367661600; end time: 1367670600
DEBUG!: child 0: start checking time slot...
DEBUG!: child 0: check_overlap(): given start 1367661600, end 1367670600
DEBUG!: child 0: check_overlap(): duration of event is 2.500000
DEBUG!: child 0: check_overlap(): list is empty or requesting to add to start, no overlap
(check_overlap() called by child and examined the link list)

The above code reflects the child's ability to resolve the time and date of event. The return result is correct in that the link list is empty and there is no conflict.

DEBUG!: child 0: overlap = 0
DEBUG!: child 0: operation completed...
DEBUG!: parent: read from child, t = 0
(result of whether there's overlap obtained)
->[accepted]
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 1
DEBUG!: child 0: eventAdd(): caller's name is Victoria
DEBUG!: child 0: event start 1367661600 end 1367670600 priority 3
(eventAdd() called by child and added event to the link list)

Acceptance of the event is initiated by child's respond that there is no overlapping and parent's transmission of instruction 1 to the child. Finally the event is successfully added as responded from the eventAdd() command.

6.3)Adding event to top of the list

addClass -victoria 2013-05-04 15:00 1.0
DEBUG!: parent: length of command is 39
DEBUG!: parent: command: addClass -victoria 2013-05-04 15:00 1.0
DEBUG!: parent: option is 0
DEBUG!: parent: case 0
DEBUG!: parent: name of caller is Victoria
DEBUG!: parent: caller's id is 0
DEBUG!: parent: instruction 0 sent to child
(instruction sent by parent and received by right child)

Again parent resolved the command correctly and passed the correct instruction to child.

DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 0
DEBUG!: child 0: read command: addClass -victoria 2013-05-04 15:00 1.0
DEBUG!: child 0: start time: 1367650800; end time: 1367654400
DEBUG!: child 0: start checking time slot...
DEBUG!: child 0: check_overlap(): given start 1367650800, end 1367654400
DEBUG!: child 0: check_overlap(): duration of event is 1.000000
DEBUG!: child 0: check_overlap(): list is empty or requesting to add to start, no overlap
DEBUG!: child 0: overlap = 0
DEBUG!: child 0: operation completed...
(check_overlap() called and discover that the required time slot is before head of list)

The check_overlap() function correctly resolved the time and duration of the event, and recognized that the time slot requested is at the start of the link list.

DEBUG!: parent: read from child, t = 0
->[accepted]
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 1
DEBUG!: child 0: eventAdd(): caller's name is Victoria
DEBUG!: child 0: event start 1367650800 end 1367654400 priority 3
(instruction sent by parent and received by right child)

6.4)Adding to tail of list

addClass -Victoria 2013-05-04 21:00 1.0
DEBUG!: parent: length of command is 39
DEBUG!: parent: command: addClass -Victoria 2013-05-04 21:00 1.0
DEBUG!: parent: option is 0
DEBUG!: parent: case 0
DEBUG!: parent: name of caller is Victoria
DEBUG!: parent: caller's id is 0
DEBUG!: parent: instruction 0 sent to child
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 0
DEBUG!: child 0: read command: addClass -Victoria 2013-05-04 21:00 1.0
DEBUG!: child 0: start time: 1367672400; end time: 1367676000
DEBUG!: child 0: start checking time slot...
DEBUG!: child 0: check_overlap(): given start 1367672400, end 1367676000
DEBUG!: child 0: check_overlap(): duration of event is 1.000000
DEBUG!: child 0: check_overlap(): checking
DEBUG!: child 0: check_overlap(): check if continue searching
DEBUG!: child 0: check_overlap(): last position, no overlap
DEBUG!: child 0: overlap = 0
DEBUG!: child 0: operation completed...
(searched until the last item and no overlapping is found)

This showcased that check_overlap() correctly located the event to be at the bottom of the list after reading the structures one by one.

DEBUG!: parent: read from child, t = 0
->[accepted]
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 1
DEBUG!: child 0: eventAdd(): caller's name is Victoria
DEBUG!: child 0: event start 1367650800 end 1367654400 priority 3
(eventAdd() called and added)

Again after the child added the event it replied parent that the event is successfully added.

6.5)Adding or rejecting meeting or gathering

addMeeting -steven 2013-05-04 18:00 1.0 alfred victoria
DEBUG!: parent: length of command is 55
DEBUG!: parent: command: addMeeting -steven 2013-05-04 18:00 1.0 alfred victoria
DEBUG!: parent: option is 1
DEBUG!: parent: case 1
DEBUG!: parent: name of caller is Steven
DEBUG!: parent: ready to read names
DEBUG!: parent: participant 0 is Alfred, id 2
DEBUG!: parent: participant 1 is Victoria, id 0
DEBUG!: parent: participant 2 is Steven, id 1
(name of people involved correctly resolved)
DEBUG!: child 2: n = 0
DEBUG!: child 2: received instruction: 0
DEBUG!: child 2: read command: addMeeting -steven 2013-05-04 18:00 1.0 alfred victoria
DEBUG!: child 2: start time: 1367661600; end time: 1367665200
D**EBUG!: child 2: start checking time slot...**
DEBUG!: child 2: check_overlap(): given start 1367661600, end 1367665200
DEBUG!: child 2: check_overlap(): duration of event is 1.000000
DEBUG!: child 2: check_overlap(): list is empty or requesting to add to start, no overlap
DEBUG!: child 2: overlap = 0
DEBUG!: child 2: operation completed...
DEBUG!: parent: child sent feedback 0
(instruction given to Alfred to check time and get reply)
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 0
DEBUG!: child 0: read command: addMeeting -steven 2013-05-04 18:00 1.0 alfred victoria
DEBUG!: child 0: start time: 1367661600; end time: 1367665200
DEBUG!: child 0: start checking time slot...
DEBUG!: child 0: check_overlap(): given start 1367661600, end 1367665200
DEBUG!: child 0: check_overlap(): duration of event is 1.000000
DEBUG!: child 0: check_overlap(): checking
DEBUG!: child 0: check_overlap(): check if continue searching
DEBUG!: child 0: check_overlap(): checking
DEBUG!: child 0: check_overlap(): Sat May 4 17:35:00 2013

DEBUG!: child 0: check_overlap(): Sat May 4 20:30:00 2013

DEBUG!: child 0: check_overlap(): slot start Sat May 4 18:00:00 2013

DEBUG!: child 0: check_overlap(): slot end Sat May 4 18:00:00 2013

DEBUG!: check_overlap(): time slot allowed is 0.000000
DEBUG!: child 0: overlap = 1
DEBUG!: child 0: operation completed...
DEBUG!: parent: child sent feedback 1

DEBUG!: parent: child 0 unavailable!
(instruction given to 'Victoria' to check if she is available, and she replies that she is not)
DEBUG!: parent: instruction 3 sent
->[rejected] - Victoria is unavailable
DEBUG!: child 1: n = 0
DEBUG!: child 1: received instruction: 3
DEBUG!: child 1: eventAdd(): caller's name is Steven
DEBUG!: child 1: eventAdd(): adding to reject list
(parent instructs Steven to reject the event)

The above is an example of how the program process events which involves multiple persons. The program will instruct involving children one by one to check if their time slot is free. When encountered not available participant, parent will cease checking with the rest of the participants because one absentee is enough to dismiss a gathering. This saves resources. The name of the unavailable participant will be transmitted to the caller of the event.

6.6)Collision detection

6.6.1)collision at head of the list

addClass -Victoria 2013-05-04 15:50 1.5
DEBUG!: parent: length of command is 39
DEBUG!: parent: command: addClass -Victoria 2013-05-04 15:50 1.5
DEBUG!: parent: option is 0
DEBUG!: parent: case 0
DEBUG!: parent: name of caller is Victoria
DEBUG!: parent: caller's id is 0
DEBUG!: parent: instruction 0 sent to child
(correct initialization)
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 0
DEBUG!: child 0: read command: addClass -Victoria 2013-05-04 15:50 1.5
DEBUG!: child 0: start time: 1367653800; end time: 1367659200
DEBUG!: child 0: start checking time slot...
DEBUG!: child 0: check_overlap(): given start 1367653800, end 1367659200
DEBUG!: child 0: check_overlap(): duration of event is 1.500000
DEBUG!: child 0: check_overlap(): last position, overlap
DEBUG!: child 0: overlap = 1
DEBUG!: child 0: operation completed...
(overlapping detected)
DEBUG!: parent: read from child, t = 1
->[rejected] - Victoria is unavailable
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 2
DEBUG!: child 0: eventAdd(): caller's name is Victoria
DEBUG!: child 0: eventAdd(): adding to reject list

The function successfully rejected the adding of event

6.6.2)Collision at the tail of the list

addClass -victoria 2013-05-04 21:30 1.5
DEBUG!: parent: length of command is 39
DEBUG!: parent: command: addClass -victoria 2013-05-04 21:30 1.5
DEBUG!: parent: option is 0
DEBUG!: parent: case 0
DEBUG!: parent: name of caller is Victoria
DEBUG!: parent: caller's id is 0
DEBUG!: parent: instruction 0 sent to child
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 0
DEBUG!: child 0: read command: addClass -victoria 2013-05-04 21:30 1.5
DEBUG!: child 0: start time: 1367674200; end time: 1367679600
DEBUG!: child 0: start checking time slot...
DEBUG!: child 0: check_overlap(): given start 1367674200, end 1367679600
DEBUG!: child 0: check_overlap(): duration of event is 1.500000
DEBUG!: child 0: check_overlap(): checking
DEBUG!: child 0: check_overlap(): check if continue searching
DEBUG!: child 0: check_overlap(): checking
DEBUG!: child 0: check_overlap(): Sat May 4 20:30:00 2013

DEBUG!: child 0: check_overlap(): Sat May 4 22:00:00 2013

DEBUG!: child 0: check_overlap(): slot start Sat May 4 21:30:00 2013

DEBUG!: child 0: check_overlap(): slot end Sat May 4 21:00:00 2013

DEBUG!: check_overlap(): time slot allowed is -0.500000
DEBUG!: child 0: overlap = 1
DEBUG!: child 0: operation completed...
(successfully detected that the the start time of intended event overlapped the last event by 0.5 hour)

DEBUG!: parent: read from child, t = 1
->[rejected] - Victoria is unavailable
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 2
DEBUG!: child 0: eventAdd(): caller's name is Victoria
DEBUG!: child 0: eventAdd(): adding to reject list

This showcased that the function successfully located the correct time slot t and engaged in calculation of the size of the slot. The time needed is 1.5 hour but the time from the start of event to the start of next event is only 1.5, thus the event is rejected.

6.6.3)Collision in the middle

addClass -victoria 2013-05-04 20:00 1.5
addClass -victoria 2013-05-04 20:00 1.5
DEBUG!: parent: length of command is 39
DEBUG!: parent: command: addClass -victoria 2013-05-04 20:00 1.5
DEBUG!: parent: option is 0
DEBUG!: parent: case 0
DEBUG!: parent: name of caller is Victoria
DEBUG!: parent: caller's id is 0
DEBUG!: parent: instruction 0 sent to child
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 0
DEBUG!: child 0: read command: addClass -victoria 2013-05-04 20:00 1.5
DEBUG!: child 0: start time: 1367668800; end time: 1367674200
DEBUG!: child 0: start checking time slot...
DEBUG!: child 0: check_overlap(): given start 1367668800, end 1367674200
DEBUG!: child 0: check_overlap(): duration of event is 1.500000
DEBUG!: child 0: check_overlap(): checking
DEBUG!: child 0: check_overlap(): Sat May 4 16:00:00 2013

DEBUG!: child 0: check_overlap(): Sat May 4 20:30:00 2013

DEBUG!: child 0: check_overlap(): slot start Sat May 4 20:00:00 2013

DEBUG!: child 0: check_overlap(): slot end Sat May 4 18:00:00 2013

DEBUG!: check_overlap(): time slot allowed is -2.000000
DEBUG!: child 0: overlap = 1
DEBUG!: child 0: operation completed...
(Collison detected correctly)

DEBUG!: parent: read from child, t = 1
->[rejected] - Victoria is unavailable
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 2
DEBUG!: child 0: eventAdd(): caller's name is Victoria
DEBUG!: child 0: eventAdd(): adding to reject list

Admitting events to the middle-successful case

addClass -victoria 2013-05-04 16:05 1.5
DEBUG!: parent: length of command is 39
DEBUG!: parent: command: addClass -victoria 2013-05-04 16:05 1.5
DEBUG!: parent: option is 0
DEBUG!: parent: case 0
DEBUG!: parent: name of caller is Victoria

DEBUG!: parent: caller's id is 0
DEBUG!: parent: instruction 0 sent to child
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 0
DEBUG!: child 0: read command: addClass -victoria 2013-05-04 16:05 1.5
DEBUG!: child 0: start time: 1367654700; end time: 1367660100
DEBUG!: child 0: start checking time slot...
DEBUG!: child 0: check_overlap(): given start 1367654700, end 1367660100
DEBUG!: child 0: check_overlap(): duration of event is 1.500000
DEBUG!: child 0: check_overlap(): checking
DEBUG!: child 0: check_overlap(): Sat May 4 16:00:00 2013

DEBUG!: child 0: check_overlap(): Sat May 4 20:30:00 2013

DEBUG!: child 0: check_overlap(): slot start Sat May 4 16:05:00 2013

DEBUG!: child 0: check_overlap(): slot end Sat May 4 18:00:00 2013

DEBUG!: check_overlap(): time slot allowed is 1.916667
(size of available slot calculated)
DEBUG!: child 0: overlap = 0
DEBUG!: child 0: operation completed...
DEBUG!: parent: read from child, t = 0
->[accepted]
DEBUG!: child 0: n = 0
DEBUG!: child 0: received instruction: 1
DEBUG!: child 0: eventAdd(): caller's name is Victoria
DEBUG!: child 0: event start 1367650800 end 1367654400 priority 3

The above showcased that the program successfully calculated the requested time slot, and allowed time slot. The program calculated the time slot required will start at 16:05 and end at 18:00, or 1.9 hour and the program calculate that the duration of event 1.5.

6.6.4)Accuracy of events recorded

Appointment Schedule
Victoria, you have 4 appointments

Date	start	end	type	people
2013-05-04	15:00	16:00	Class	
2013-05-04	16:05	17:35	Class	
2013-05-04	18:00	20:30	Class	
2013-05-04	21:00	22:00	Class	

-the end-

Rejected List

Victoria, you have 3 rejected appointment

Date	start	end	type	reasons
2013-05-04	15:50	17:20	Class	not available
2013-05-04	21:30	23:00	Class	not available
2013-05-04	20:00	21:30	Class	not available

-the end-

Events for Victoria are correctly recorded in order

Appointment Schedule
Steven, you have 0 appointments

Date	start	end	type	people

-the end-

Rejected List
Steven, you have 1 rejected appointment

Date	start	end	type	reasons
2013-05-04	18:00	19:00	Meeting	Victoria not available

-the end-

Event for Steven is also correctly recorded with name of the absentee in the reject list

7)Performance analysis

The algorithm adopted in the program is first come first serve. Events entered earlier will have priority over the later ones. If time slot is available, the adding of event will be accepted. However if the time slot is occupied or there is collision with other earlier entered event, the event at hand will be rejected.

Another algorithm that makes sense in this program's context is priority. In priority scheduling, not only will the program check if the time slot is free, it will further examine the priority of the colliding event and the event at hand. The event at hand will be discarded if it has a lower priority. On the contrary, if the event at hand has a higher priority, colliding event or events will be rejected and discarded. The event at hand will be admitted to the just freed time slot.

Advantage of first come first serve algorithm

There are a number of advantages to user first come fist serve algorithm. Firstly, the algorithm only requires an empty time slot and no further action is necessary. Implementation is very easy. Secondly it involves low overhead, which means that steps involving discovering the colliding event is not needed. This saves resources and time.

Disadvantage of first come first serve algorithm

First come first serve scheduling may create so called convey effect, meaning that there exists a chance where important events cannot be accepted because all time slots are occupied. This method is also not fair for more important events

Advantage of Priority of nature of event algorithm

By considering the priority with the nature of event is more fair that more important events can supersede less important events regardless of the time of recording.

Disadvantage of Priority of nature of event algorithm

Disadvantage includes higher overhead. Meaning that extra steps have to be taken to to detect the colliding event for a free time slot.

Collision of events can occur because of collision of staring time, end time and middle. It implies that the admission of one event can be rejected because of three other events in the worst case even for minor collision. This implies that the cost of accepting a higher priority event can be very high.

After detecting events which cause collision, steps has to be taken to discard them by unlinking and freeing them. Further, it is necessary to link up the new event with the new 'previous' and 'next' event.

7.1)Evidence that priority according to event nature is more complicated

```
int check_overlap(user *member, time_t _start, time_t _end, int _priority) {

 double duration = difftime(_end, _start);

 //create new node to process event
 event *p = (event*)malloc(sizeof(event));
 p = member->start;
 event *r = (event*)malloc(sizeof(event));

 if(p==NULL ||(_end)<=(p->start_time)) {
    //check if list is empty, return 0 if so
    return 0;
 } else {
    //look for right insertion position if list is not empty
    for(p=member->start, r=p->next; p!=NULL; p=p->next, r=p->next) {
      //find position, p =  one item before the positon
        if(p->next == NULL) {
           //case for last node in the list is reached
           if((p->end_time)<=(_start)) {
              return 0;
           } else if((p->end_time)>(_start)) {
              if(_priority > p->priority) return 1;
              else printf("New event overwrites event starting at %s", ctime(&(p->start_time)));
                 return 2;
           }

        } else if((_start)>=(p->end_time)) {
           //case for last node hasn't reached
           if((_start)>=(r->start_time)) {
                //if not reaching the right slot, continue
                 continue;
           }
         //right slot found
           break;
        } else {
            if(_priority > p->priority) return 1;
            else {
               printf("New event overwrites event starting at %s", ctime(&(p->start_time)));
               if(_end > r->start_time) printf("New event overwrites event starting at %s",
ctime(&(r->start_time)));
                 return 2;
           }
        }
     }
   }
```

```
  //examine width of slot
time_t starting = p->start;
time_t ending = r->start_time;

//compute availability of slot in the list
double slot = difftime(ending, starting);

//determine if slot is wide enough
 if(slot >= duration) {
   //slot is enough to add a event
   return 0;
 } else {
   if(_start < starting) {
     if(_priority > p->priority) return 1;
       printf("New event overwrites event starting at %s", ctime(&(p->start_time)));
       return 2;
   }
  //slot is not wide enough
  if(_priority > r->priority) return 1;
    printf("New event overwrites event starting at %s", ctime(&(r->start_time)));
    return 2;
 }
}
```

It can bee seen that extra conditions have to be implemented in order to determine if an event can be discarded because of the event at hand.

8)Program setup and execution

The program is written in C version 99. It is designed to be compiled with GCC on any Linux platform. The platform for testing of the software is Fedora 18 x86_64 and the program can be successfully compiled on the 'apollo' server with SUSE Enterprise Linux 12.

The program is compiled with the following command

gcc -o aos aos.c

The program is executed with ./aos (user names)

9)Future expansion and possible improvements

9.1)Better error detection

The currently program is implemented without error detection ability. This means that if user made wrong input, the program would crash. When the parent crashed, the child will be left unattended in an infinite loop thus burning the CPU. A mechanism being implemented currently is that the child will check every time when the loop returns if the parent is still running. If not the child will prompt error and exit. However, the long term solution on top of the inter process checking is to have better error checking so that the parent will not crash easily.

9.2)Confirmation signal

Currently when parent instruct the child to add an event, the signal is one direction where the parent assumes the event will be added successfully. If not, it will also be assumed so. This creates hidden error if there is bug in the eventAdd() function that the problem will not be discovered instantly and create a false impression to user that the event has been added successfully. In the future successful confirmation should be implemented to avoid the child getting lost. A set of error codes should be created for the child to tell the parent what is wrong and let the parent display corresponding error message to promote the user.

9.3)Graphic user interface

The current implementation of the software is with command line interface which is unfriendly to common users. Graphic user interface can be developed for the parent process. QT is cross platform but the code will not be portable since fork() is native to Linux anyway.

9.4)Easier editing of database

Currently, once the user has entered the command there is no way to reverse since there exists no mechanism for user to modify given data. This create inconveniences as if user inputted anything wrong information he or she will have to start again.

10)License of the software

10)Source Code

```
/***********************************************************************/
/* Copyright 2013 Steven Chien                                          */
/* 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/> .*/
/***********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <errno.h>

int P2C[10][2];
int C2P[10][2];
//development purpose
int child_no = 3;
int priority = 2;
char absentee[20];
pid_t parent_pid;

//create type event
typedef struct Event {
        //define struct event
        int priority;                //identify event type
        time_t start_time;           //start time of event
        time_t end_time;             //end time of event
        char usr[20][20];            //name of participant
        int parti_no;                //number of participants
        struct Event *next;          //pointer to next event
} event;

//create type user
typedef struct User {
        //define struct user
```

```
	char name[20];				//name of user
	pid_t process_id;			//process ID of user
	int event_count;			//number of events engaged in
	int rejection_count;		//number of events being rejected
	int id;						//(array) id of user
	event *start, *end;			//head and tail of link list of events
	event *rej_start, *rej_end;	//head and tail of link list of rejected events
} user;

//int availibility[10] = { 1 };

//function to extract time from command
void parse_time(char *command, time_t *start_time, time_t *end_time) {
	//define time struction for start and end
	struct tm start;
	struct tm end;
	int i;
	char *token;
	//tokenize string to the part where time data is descripted
	token = strtok(command, " :-.");
	token = strtok(NULL, " :-.");

	//extract year
	token = strtok(NULL, " :-.");
	int year = atoi(token);
	////printf("DEBUG!: eventAdd(): year %d\n", year);
	start.tm_year = year - 1900;

	//extract month
	token = strtok(NULL, " :-.");
	int month = atoi(token);
	////printf("DEBUG!: eventAdd(): month %d\n", month);
	start.tm_mon = month - 1;

	//extract day
	token = strtok(NULL, " :-.");
	int day = atoi(token);
	////printf("DEBUG!: eventAdd(): day %d\n", day);
	start.tm_mday = day;

	//extract hour
	token = strtok(NULL, " :-.");
	int hour = atoi(token);
	////printf("DEBUG!: eventAdd(): hour %d\n", hour);
	start.tm_hour = hour;

	//extract minute
```

```
	token = strtok(NULL, " :-.");
	int minute = atoi(token);
	////printf("DEBUG!: eventAdd(): minute %d\n", minute);
	start.tm_min = minute;

	//second
	start.tm_sec = 0;

	//create end_time as a duplicate of start
	memcpy(&end, &start, sizeof(struct tm));

	//adjust end time
	//adjust ending hour
	token = strtok(NULL, " :-.");
	int duration_hour = atoi(token);
	////printf("DEBUG!: parse_time(): duration hour = %d\n", duration_hour);
	end.tm_hour = end.tm_hour + duration_hour;

	//adjust ending min
	token = strtok(NULL, " :-.");
	int duration_min = atoi(token);
	//printf("parse_time(): duration min = %d\n", duration_min);
	end.tm_min = end.tm_min + ((float)duration_min/10*60);

	//finialize times to time_t format
	*start_time = mktime(&start);
	*end_time = mktime(&end);
	//printf("her end time is %s\n", ctime(end_time));

}

//check if requested event colid with events in list; return 0 = no overlap, 1 = overlap
int check_overlap(user *member, time_t _start, time_t _end) {
	//debug!
	//printf("DEBUG!: child %d: check_overlap(): given start %d, end %d\n", member->id, _start,
_end);

	//compute duration of requested event
	double duration = difftime(_end, _start);
	//printf("DEBUG!: child %d: check_overlap(): duration of event is %f\n", member->id,
duration/60.0/60.0);

	//create new node to process event
	event *p = (event*)malloc(sizeof(event));
	p = member->start;
	event *r = (event*)malloc(sizeof(event));
```

```
	if(p==NULL ||(_end)<=(p->start_time)) {
		//check if list is empty, return 0 if so
		//printf("DEBUG!: child %d: check_overlap(): list is empty, no overlap\n", member-
>id);
		return 0;
	} else {
		//look for right insertion position if list is not empty
		for(p=member->start, r=p->next; p!=NULL; p=p->next, r=p->next) {		//find
position, p = one item before the positon

			if(p->next == NULL) {
				//case for last node in the list is reached
				if((p->end_time)<=(_start)) {
					//printf("DEBUG!: child %d: check_overlap(): last position,
overlap\n", member->id);
					return 0;				//debug
				} else if((p->end_time)>(_start)) {
					//printf("DEBUG!: child %d: check_overlap(): last position, no
overlap\n", member->id);
					return 1;
				}
				break;

			} else  if((_start)>=(p->end_time)) {
				//case for last node hasn't reached
				//printf("DEBUG!: child %d: check_overlap(): checking\n", member-
>id);
				if((_start)>=(r->end_time)) {				//check if the slot is
the right slot
					//if not reaching the right slot, continue
					//printf("DEBUG!: child %d: check_overlap(): check if continue
searching\n", member->id);
					continue;
				}
				//right slot found
				//printf("DEBUG!: child %d: check_overlap(): %s\n", member->id,
ctime(&(p->end_time)));		//debug!
				//printf("DEBUG!: child %d: check_overlap(): %s\n", member->id,
ctime(&(r->end_time)));
				break;
			}
		}
	}

	//examine width of slot
	time_t starting = _start;
	time_t ending = r->start_time;
```

```
	//debug!
	//printf("DEBUG!: child %d: check_overlap(): slot start %s\n", member->id, ctime(&starting));
	//printf("DEBUG!: child %d: check_overlap(): slot end %s\n", member->id, ctime(&ending));

	//compute availability of slot in the list
	double slot = difftime(ending, starting);
	//debug!
	//printf("DEBUG!: check_overlap(): time slot allowed is %f\n", slot/60.0/60.0);

	//determine if slot is wide enough
	if(slot>=duration)
		//slot is enough to add a event
		return 0;
	else
		//slot is not wide enough
		return 1;
}

//function to add event to link list, member=struct User, command=instruction from user; flag: 1 = add
to appointment, 0  to reject
void eventAdd(user *member, char *command, int flag) {

	//create new event and allot space
	event *q = (event*)malloc(sizeof(event));
	char *token;
	char *tok;
	int i = 0, j = 0;
	int id;
	char caller[20];

	//set flag if one of the participant not available
	/*for(i=0; i<10; i++) {
		if(availability[i]==0) {
			flag = 0;
		}
	}*/

	//dublicate command to keep safe
	char command_cc[80];
	strcpy(command_cc, command);

	//determine class of event
	token = strtok(command_cc, " :-.");
	if(strcmp(token, "addClass")==0)
		priority = 3;
	else if(strcmp(token, "addMeeting")==0)
```

```
		priority = 2;
	else if(strcmp(token, "addGathering")==0)
		priority = 1;

	//determine caller's name and store to char caller[]
	token = strtok(NULL, " .:-");
	strcpy(caller, token);
	caller[0] = toupper(caller[0]);
	//printf("DEBUG!: child %d: eventAdd(): caller's name is %s\n", member->id, caller);

	//extract time and priority level from command
	parse_time(command, &(q->start_time), &(q->end_time));
	q->priority = priority;
	i = 0;

	//read user name if evnt involves multiple users
	if(flag==1&&(priority==2 || priority==1)) {
		//case for event is added
		//store caller's name as first participant
		strcpy(q->usr[i], caller);
		i++;
		//store subsequent participants
		while((token=strtok(NULL, " :-."))!=NULL) {
			strcpy(q->usr[i], token);
			q->usr[i][0] = toupper(q->usr[i][0]);
			i++;
		}
		//store no. of participant
		q->parti_no = i;
	} else {
		//case for event is rejected
		//add name of the absentee
		strcpy(q->usr[0], absentee);
		//no. of name stored in q->usr[] is 1 for this case
		q->parti_no = 1;
	}


	event *p, *r;

	//link up the lis
	if(flag==0) {
		//case for rejected event, add to rejection list
		//printf("DEBUG!: child %d: eventAdd(): adding to reject list\n", member->id, member-
>name);
		if((member->rej_start)==NULL) {
			member->rej_start = q;
```

```
                member->rej_end = member->rej_start;
            } else {
                member->rej_end->next = q;
                member->rej_end = q;
            }
            member->rejection_count++;                                  //add rejected
event
        } else if(flag==1) {
            //case for accepted event, add to appointment list
            p = member->start;

            if(p==NULL ||((q->end_time)<=(p->start_time))) {
                //case where list is empty or adding to the top of the list
                q->next = member->start;
                member->start = q;
                member->event_count++;
                //printf("DEBUG!: child %d: empty list or add to 1st position\n", member->id);
                //debug!
                return;
            }
            //not to be added to first position, initiate searching for the right slot
            for(p=member->start, r=p->next; p!=NULL; p=p->next, r=p->next) {          //find
position, p = one item before the positon
                if(p->next == NULL) {
                    //case where event is added to last of the list
                    //printf("DEBUG!: child %d: add to last\n", member->id);
            //debug
                    break;
                }

                if((q->start_time)>=(p->end_time)) {
                    //compare and find slot
                    if((q->start_time)>=(r->end_time)) {                        //check
if the slot is the right slot
                        continue;
                    }
                    //printf("DEBUG!: child %d: finding position...\n", member->id);
            //debug!
                    //printf("DEBUG!: child %d: p %d, r %d\n", member->id, p->end_time,
r->end_time);       //debug!
                    //printf("DEBUG!: child %d: add to middle\n", member->id);
                    break;
                }
            }
            //case for adding to 2nd or later position
            q->next = p->next;
            p->next = q;
```

```
			member->event_count++;
		}
}

//report generation function, usage: usr=struct User, output=output file name, flag: 1=appointment;
2=rejection; 3=app+rej
void report(user *usr, char output[10], int flag) {

	int i = 0;
	event *p;
	struct tm *time_info;
	char string[20];
	FILE *file;

	//open or create file, name specified by user
	file = fopen(output, "w");

	if(flag==1 || flag==3) {

		fprintf(file, "***Appointment Schedule***\n\n");

		fprintf(file, "%s, you have %d appointments\n\n", usr->name, usr->event_count);

		fprintf(file, "Date\t\tstart\tend\ttype\t\tpeople\t\t\n");
		fprintf(file,
"==================================================================================
===================\n");

		for(p=usr->start; p!=NULL; p=p->next) {
			//printing time data
			time_info = localtime(&(p->start_time));
			strftime(string, 15, "%Y-%m-%d", time_info);
			fprintf(file, "%s\t", string);
			strftime(string, 15, "%H:%M", time_info);
			fprintf(file, "%s\t", string);
			time_info = localtime(&(p->end_time));
			strftime(string, 15, "%H:%M", time_info);
			fprintf(file, "%s\t", string);

			//print event type
			if((p->priority)==3)
				fprintf(file, "Class\t\t");
			else if((p->priority)==2)
				fprintf(file, "Meeting\t\t");
			else if((p->priority)==1)
				fprintf(file, "Gathering\t");
```

```
				//print participant
				for(i=0; i<(p->parti_no); i++) {
					fprintf(file, "%s ", p->usr[i]);
				}

				//return
				fprintf(file, "\n");
			}
			fprintf(file, "\n\t\t\t\t\t-the end-\n\n");

			printf("->[appointment list generated]\n");
	}

	if(flag==2 || flag==3) {

			fprintf(file, "***Rejected List***\n\n");

			fprintf(file, "%s, you have %d rejected appointment\n\n", usr->name, usr-
>rejection_count);

			fprintf(file, "Date\t\tstart\tend\ttype\t\treasons\t\t\n");
			fprintf(file,
"=========================================================================
==================\n");

			for(p=usr->rej_start; p!=NULL; p=p->next) {
				//printing time data
				time_info = localtime(&(p->start_time));
				strftime(string, 15, "%Y-%m-%d", time_info);
				fprintf(file, "%s\t", string);
				strftime(string, 15, "%H:%M", time_info);
				fprintf(file, "%s\t", string);
				time_info = localtime(&(p->end_time));
				strftime(string, 15, "%H:%M", time_info);
				fprintf(file, "%s\t", string);

				//print event type
				if((p->priority)==3)
					fprintf(file, "Class\t\t");
				else if((p->priority)==2)
					fprintf(file, "Meeting\t\t");
				else if((p->priority)==1)
					fprintf(file, "Gathering\t");

				//print participant
				for(i=0; i<(p->parti_no); i++) {
					fprintf(file, "%s ", p->usr[i]);
```

```
				}
				fprintf(file, " not available");

				//return
				fprintf(file, "\n");
			}

			fprintf(file, "\n\t\t\t\t\t-the end-\n\n");

			printf("->[rejection list generated]\n");
		}

		fclose(file);

}

void parent(user *usr) {

		char string[80];
		char command[80];
		int caller, participant_id[9], feedback, absentee_id;
		char *token;
		int option = 10, t, i, j, flag;
		int availability[10];
		event *q = (event*)malloc(sizeof(event));
		printf("~Welcome to AOS~\n");
		printf("please enter appointment:\n");
		/*for(i=0; i<10; i++) {
			close(P2C[i][0]);
			close(C2P[i][1]);
		}*/
		while(1) {
			//read command
			strcpy(command, " ");
			strcpy(string, " ");
			fgets(command, 79, stdin);
			command[strlen(command)-1] = 0;
			//printf("DEBUG!: parent: length of command is %d\n", strlen(command));

			//analysis instruction
			strcpy(string, command);
			token = strtok(string, " -:.");
			//printf("DEBUG!: parent: command: %s\n", command);

			if(strcmp(token, "addClass")==0)
				option = 0;
			else if(strcmp(token, "addMeeting")==0)
```

```
        option = 1;
else if(strcmp(token, "addGathering")==0)
        option = 1;
else if(strcmp(token, "printSchd")==0)
        option = 2;
else if(strcmp(token, "endProgram")==0)
        option = 3;
//printf("DEBUG!: parent: option is %d\n", option);

//engage in operation with child
switch(option) {
        case 0:
                //add  class
                //printf("DEBUG!: parent: case 0\n");

                //get caller ID
                token = strtok(NULL, " -:.");
                token[0] = toupper(token[0]);
                //printf("DEBUG!: parent: name of caller is %s\n", token);

                //get user id
                for(i=0; i<child_no; i++) {
                        if(strcmp(token, usr[i].name)==0)
                                break;
                }

                caller = i;
                //printf("DEBUG!: parent: caller's id is %d\n", caller);
                int writer = 0;

                //creating node
                t = 0;

                //sending command to child
                close(P2C[caller][0]);
                write(P2C[caller][1], &t, sizeof(int));
                write(P2C[caller][1], &command, strlen(command));
                //close(P2C[caller][1]);
                //printf("DEBUG!: parent: instruction %d sent to child\n", t);

                t = 10;
                //reading feedback from child
                close(C2P[caller][1]);
                read(C2P[caller][0], &t, sizeof(int));
                //close(C2P[caller][0]);

                //give instruction to child according to feedback
```

```
		//printf("DEBUG!: parent: read from child, t = %d\n", t);

		if(t==0) {
			t = 1;
			close(P2C[caller][0]);
			write(P2C[caller][1], &t, sizeof(int));
			write(P2C[caller][1], &command, 80);
			//close(P2C[caller][1]);
			printf("->[accepted]\n");
		} else {
			t = 2;
			close(P2C[caller][0]);
			write(P2C[caller][1], &t, sizeof(int));
			write(P2C[caller][1], &command, 80);
			//close(P2C[caller][1]);
			printf("->[rejected] - %s is unavailable\n", usr[caller].name);
		}
		break;
	case 1:
		//add meeting or gathering
		//printf("DEBUG!: parent: case 1\n");

		//get caller ID
		token = strtok(NULL, " -:.");
		token[0] = toupper(token[0]);
		//printf("DEBUG!: parent: name of caller is %s\n", token);

		//get user id
		for(i=0; i<child_no; i++) {
			if(strcmp(token, usr[i].name)==0)
				break;
		}
		//DEBUG!
		caller = i;
		////printf("DEBUG!: parent: caller's id is %d\n", caller);
		writer = 0;

		//get participants
		for(i=0; i<8; i++) {
			token = strtok(NULL, " :.-");
		}
		//printf("DEBUG!: parent: ready to read names\n");
		j = 0;

		while(token!=NULL) {
			token[0] = toupper(token[0]);
			for(i=0; i<child_no; i++) {
```

```
					if(strcmp(token, usr[i].name)==0) {
						participant_id[j] = i;
						//printf("DEBUG!: parent: participant %d is %s, id
%d\n", j, usr[i].name, participant_id[j]);
						j++;
					}
				}
				token = strtok(NULL, " :-.");
			}
			participant_id[j] = caller;
			//printf("DEBUG!: parent: participant %d is %s, id %d\n", j,
usr[caller].name, caller);

			//creating node
			t = 0;
			feedback = 0;

			//sending command to children
			for(i=0; i<j+1; i++) { //need to be fixed!
				close(P2C[participant_id[i]][0]);
				write(P2C[participant_id[i]][1], &t, sizeof(int));
				write(P2C[participant_id[i]][1], &command, strlen(command));

				close(C2P[participant_id[i]][1]);
				read(C2P[participant_id[i]][0], &feedback, sizeof(int));
				//printf("DEBUG!: parent: child sent feedback %d\n", feedback);

				if(feedback==1) {
					absentee_id = participant_id[i];
					break;
				}
			}

			if(feedback==1) {
				//printf("DEBUG!: parent: child %d unavailable!\n",
participant_id[i]);
				t = 3;
				close(P2C[caller][0]);
				write(P2C[caller][1], &t, sizeof(int));
				//printf("DEBUG!: parent: instruction %d sent\n", t);

				write(P2C[caller][1], &participant_id[i], sizeof(int));
				write(P2C[caller][1], &command, 80);

				write(P2C[caller][1], &(usr[absentee_id].name),
sizeof(usr[absentee_id].name));
				printf("->[rejected] - %s is unavailable\n",
```

```
usr[absentee_id].name);

				} else if(feedback==0) {
					for(i=0; i<j;i++) {
						t = 1;
						close(P2C[caller][0]);
						write(P2C[caller][1], &t, sizeof(int));
						write(P2C[caller][1], &command, 80);

						for(i=0; i<j; i++) {
							close(P2C[participant_id[i]][0]);
							write(P2C[participant_id[i]][1], &t, sizeof(int));
							write(P2C[participant_id[i]][1], &command, 80);
							//printf("DEBUG!: parent: instruction %d sent to
child %d\n", t, participant_id[i]);
						}
					}
					printf("->[accepted]\n");

				}

				break;

			case 2:
				//print report
				//printf("DEBUG!: parent: case 2\n");
				token = strtok(NULL, " .-:");
				token[0] = toupper(token[0]);
				for(i=0; i<child_no; i++) {
					if((strcmp(token, usr[i].name))==0)
						break;
				}
				//printf("DEBUG!: parent: caller is %s, id %d\n", token, i);
				caller = i;

				t = 4;
				close(P2C[caller][0]);
				write(P2C[caller][1], &t, sizeof(t));
				write(P2C[caller][1], &command, sizeof(command));
				//printf("DEBUG!: parent: instruction %d sent to child %d\n", t, caller);

				break;

			case 3:
				t = 5;
				for(i=0; i<child_no; i++) {
					//sent terminating message
```

```
                    close(P2C[i][0]);
                    write(P2C[i][1], &t, sizeof(t));
                }
                /*
                while(1) {
                    int status;
                    pid_t done = wait( &status );
                    if ( done == -1 ) {
                        if ( errno == ECHILD ) break; // no more child processes
                    } else {
                        if ( !WIFEXITED( status ) || WEXITSTATUS( status ) != 0
) {
                            printf( "PID %d failed\n", done );
                            exit(1);
                        }
                    }
                }*/
                wait(NULL);
                printf("->Bye!\n");
                exit(0);
                break;

        }
    }
}

void child(user *usr, int id) {

    char string[80];
    int cmd, i, n, t=999;
    int *this_P2C = P2C[id];
    int *this_C2P = C2P[id];
    int temp, flag;
    char *token;
    //close pipes of other child
    for(i=0; i<child_no; i++) {
        close(P2C[i][1]);
        close(C2P[i][0]);
        if(i!=id) {
            close(P2C[i][0]);
            close(C2P[i][1]);
        }
    }

    //printf("DEBUG!: child: my ID is %d\n", id);
    while(1) {
        //check if parent's alive
```

```
		if(getppid()==1) {
			printf("error!: child %d: parent process terminated, exiting...\n", usr->id);
			exit(1);
		}

		strcpy(string, " ");
		cmd = 0;
		while((read(this_P2C[0], &cmd, sizeof(int)))>0) {

			close(this_P2C[1]);
			//printf("DEBUG!: child %d: n = %d\n", usr->id, n);
			//printf("DEBUG!: child %d: received instruction: %d\n", usr->id, cmd);
			switch(cmd) {
				case 0:
					//check time
					read(this_P2C[0], string, sizeof(string));
	//send command
					//printf("DEBUG!: child %d: read command: %s\n", usr->id,
string);

					//close(this_P2C[0]);
					event *p = (event*)malloc(sizeof(event));

					//parse time from command
					parse_time(string, &(p->start_time), &(p->end_time));
	//get time
					//printf("DEBUG!: child %d: start time: %d; end time: %d\n", usr-
>id, p->start_time, p->end_time);
					//printf("DEBUG!: child %d: start checking time slot...\n", usr-
>id);

					//check overlap with child
					t = check_overlap(usr, p->start_time, p->end_time);
					//printf("DEBUG!: child %d: overlap = %d\n", usr->id, t);
					close(this_C2P[0]);
					write(this_C2P[1], &t, sizeof(int));
					//close(this_C2P[1]);
					//printf("DEBUG!: child %d: operation completed...\n", usr->id);
					break;
				case 1:
					//add event
					close(this_P2C[1]);
					read(this_P2C[0], string, sizeof(string));
					//close(this_P2C[0]);
					eventAdd(usr, string, 1);
					//printf("DEBUG!: child %d: event start %d end %d priority
%d\n", usr->id, usr->start->start_time, usr->start->end_time, usr->start->priority);
```

```
			break;
		case 2:
			//reject class
			close(this_P2C[1]);
			read(this_P2C[0], string, sizeof(string));
			//close(this_P2C[0]);
			eventAdd(usr, string, 0);
			break;
		case 3:
			//reject meeting and gathering
			close(this_P2C[1]);
			read(this_P2C[0], &temp, sizeof(int));
			read(this_P2C[0], string, sizeof(string));
			read(this_P2C[0], absentee, sizeof(absentee));

			eventAdd(usr, string, 0);

			break;

		case 4:
			//print report
			close(this_P2C[1]);
			read(this_P2C[0], string, sizeof(string));
			//printf("DEBUG!: child %d: received command %s\n", usr->id,
string);

			token = strtok(string, " :.-");
			token = strtok(NULL, " :.-");
			token = strtok(NULL, " :.-");
			if(strcmp(token, "t")==0)
				flag = 1;
			else if(strcmp(token, "r")==0)
				flag = 2;
			else if(strcmp(token, "f")==0)
				flag = 3;
			else {
				printf("error!: invalid option\n");
				break;
			}
			//printf("DEBUG!: child %d: flag is %d\n", usr->id, flag);

			token = strtok(NULL, " :-");

			report(usr, token, flag);

			break;

		case 5:
```

```
                        //end program
                        //printf("DEBUG!: child %d: ending...\n", usr->id);
                        exit(0);
                        break;
                    default:
                        return;
                }
            }

            //close(this_P2C[1]);
            //close(this_C2P[0]);
            //wait(NULL);
        }
        ////printf("DEBUG!: child %d: exit loop, ending...\n", usr->id);

}

int main(int argc, char **argv) {

        user *usr = (user*)malloc((argc-1)*sizeof(user));
        int i = 0;
        child_no = argc - 1;
        if(child_no<3 || child_no>10) {
                printf("error: number of user incorrect!\n");
                return -1;
        }
        parent_pid = getpid();
        //printf("DEBUG!: struct created\n");

        //initializing all users
        for(i=0; i<(argc-1); i++) {
                strcpy(usr[i].name, argv[i+1]);
                usr[i].name[0] = toupper(usr[i].name[0]);
                usr[i].id = i;
                usr[i].start = NULL;
                usr[i].end = NULL;
                usr[i].rej_start = NULL;
                usr[i].rej_end = NULL;
                usr[i].rejection_count = 0;
                usr[i].event_count = 0;
                //printf("DEBUG!: main(): usr id %d, name %s\n", usr[i].id, usr[i].name);
        }
        //printf("DEBUG!: names copied\n");

        //forking child processes
        for(i=0; i<(argc-1); i++) {
                pipe(P2C[i]);
```

```
		pipe(C2P[i]);
		//printf("DEBUG!: main(): forking %d child\n", i);
		usr[i].process_id = fork();
		//determine
		if(usr[i].process_id==0) {
			child(&usr[i], i);
			break;
		} else if(usr[i].process_id>0) {
		}
	}

	//parent process
	if(getpid()==parent_pid) {
		//printf("DEBUG!: main(): I am Parent\n");
		parent(usr);
	}

	return 0;
}
```

10.1)Test cases

addClass -peter 2013-05-05 12:00 1.5
addClass -peter 2013-05-05 14:00 1.5
addClass -peter 2013-05-05 15:00 2.5
addClass -paul 2013-05-05 13:00 0.5
addClass -paul 2013-05-05 12:00 1.5
addClass -lucy 2013-05-05 15:00 2.5
addClass -lucy 2013-05-05 16:00 1.5
addClass -mary 2013-05-05 18:00 2.0
addMeeting -peter 2013-05-05 08:30 1.5 mary
addMeeting -mary 2013-05-05 12:00 2.0 peter paul
addGathering -mary 2013-05-06 15:00 1.5 lucy

printSchd -peter -f peterSchd.dat
printSchd -paul -f paulSchd.dat
printSchd -lucy -f lucySchd.dat
printSchd -mary -f marySchd.dat

Appointment Schedule

Lucy, you have 2 appointments

Date	start	end	type	people
2013-05-05	15:00	17:30	Class	
2013-05-06	15:00	16:30	Gathering	Mary Lucy

-the end-

Rejected List

Lucy, you have 1 rejected appointment

Date	start	end	type	reasons
2013-05-05	16:00	17:30	Class	not available

-the end-

Appointment Schedule

Lucy, you have 2 appointments

Date	start	end	type	people
2013-05-05	10:30	11:30	Class	
2013-05-05	12:00	13:30	Meeting	Lucy Paul Peter

-the end-

Appointment Schedule

Mary, you have 3 appointments

Date	start	end	type	people
2013-05-05	08:30	10:00	Meeting	Peter Mary
2013-05-05	18:00	20:00	Class	
2013-05-06	15:00	16:30	Gathering	Mary Lucy

-the end-

Rejected List

Mary, you have 1 rejected appointment

Date	start	end	type	reasons
2013-05-05	12:00	14:00	Meeting	Peter not available

-the end-

Appointment Schedule

Paul, you have 1 appointments

Date	start	end	type	people
2013-05-05	13:00	13:30	Class	

-the end-

Rejected List

Paul, you have 1 rejected appointment

Date	start	end	type	reasons
2013-05-05	12:00	13:30	Class	not available

-the end-

Appointment Schedule
Peter, you have 3 appointments

Date	start	end	type	people
2013-05-05	08:30	10:00	Meeting	Peter Mary
2013-05-05	12:00	13:30	Class	
2013-05-05	14:00	15:30	Class	

-the end-

Rejected List

Peter, you have 1 rejected appointment

Date	start	end	type	reasons
2013-05-05	15:00	17:30	Class	not available

-the end-

Another test case

addClass -victoria 2013-05-04 18:00 2.5
addClass -victoria 2013-05-04 15:00 1.0
addClass -Victoria 2013-05-04 21:00 1.0
addClass -Victoria 2013-05-04 15:50 1.5
addClass -victoria 2013-05-04 21:30 1.5
addClass -victoria 2013-05-04 20:00 1.5
addClass -victoria 2013-05-04 16:05 1.5

Appointment Schedule

Victoria, you have 4 appointments

Date	start	end	type	people
2013-05-04	15:00	16:00	Class	
2013-05-04	16:05	17:35	Class	
2013-05-04	18:00	20:30	Class	
2013-05-04	21:00	22:00	Class	

-the end-

Rejected List

Victoria, you have 3 rejected appointment

Date	start	end	type	reasons
2013-05-04	15:50	17:20	Class	not available
2013-05-04	21:30	23:00	Class	not available
2013-05-04	20:00	21:30	Class	not available

-the end-