# COMP2432 Operating System

COMP2432_20222_A

Instructors : Dr LEONG Hong Va

# APO Report

## *~ APPointment Organizer ~*

*Schedule your Time*

**Group 1**



| Student Name | Student ID |
|---|---|
| HU Wenqing | 21094549D |
| LIU Chenxi | 21096794D |
| ZHANG Tianyi | 21099833D |
| ZHANG Wanyu | 21094724D |

# CONTENTS


**Part I. Introduction** **3**

**Part II. Scope** **4**

**Part III. Algorithm Concept** **5**

*FCFS* *5*

*Priority* *6*

*PrintSchd ALL* *7*

**Part IV. New algorithm** **8**

**Part V. Reschedule** **8**

**Part VI. Software Structure** **10**

*OS Service* *10*

*OS Structure* *16*

**Part VII. Testing Cases** **26**

**Part VIII. Performance** **28**

**Part IIII. Program Set up and Execution** **30**

**Appendix** **38**

**Part X. Contribution Form** **46**

# Part I.Introduction

**An appointment organizer (APO)** is a tool designed to help individuals and organizations manage their time effectively by keeping track of important events, meetings, and tasks. A general APO allows users to schedule and organize appointments, set reminders, and manage to-do lists. By avoiding scheduling conflicts, preventing missed meetings or events, and improving time management, an APO can increase productivity. Similar to calendar products like Apple Calendar and Google Calendar, an APO provides a comprehensive tool for managing schedules and time commitments.

In the context of this Operating System course, the purpose of writing an APO could serve as raising our awareness in following Operating System concepts:

- **Process management**: This APO requires multiple processes to simulate real-life users. And we need to pass information across the different processes. This requires the techniques of creating processes using system calls fork(), as well as system call pipe() for interprocess communications.
- **Scheduling**: An APO is in essence all about time management, which involves scheduling of different tasks. We can implement several well-known scheduling algorithms learned in course such as First-Come-First-Serve(FCFS), Priority.
- **File and input management**: An APO typically involves storing and retrieving data from files. We could practice the use of relevant system calls to manipulate files in C and Linux/Unix. Also the methods to input both from files and keyboard.
- **Process control**: The appointment organizer involves managing multiple processes, such as scheduling appointments, sending reminders, and updating the user interface. The concept which requires to be learnt is about creating, managing, and synchronizing processes to ensure that the organizer functions correctly.
- **Mutual exclusion**: The appointment organizer may require access to shared resources, such as appointment data, which can lead to issues such as race conditions and data inconsistency. To prevent these problems, mutual exclusion techniques can be implemented, such as locks or semaphores, to control access to shared resources.
- **Deadlock**: The appointment organizer may involve multiple processes competing for shared resources, which can result in deadlock. We need to understand the causes of deadlock and how to prevent it by using techniques such as resource ordering and deadlock detection.

- **Synchronization**: The appointment organizer requires synchronization to ensure that different processes are coordinated and communicate effectively. Synchronization techniques such as monitors, semaphores, and message passing can be used to achieve efficient and reliable communication between processes.

# Part II. Scope

1. **Process Management**:
   APO in this project need to manage multiple processes, to achieve functions, such as updating timetables and sending meeting notifications.

2. **Interprocess Communication and programming**
   Parent as the manager needs to communicate with its children, which serve as different users. In this program, a STAR communication topology is deployed, in which every information sent by each child to another child is transmitted by the parent.

3. **Scheduling**
   APO needs to schedule events and tasks based on various criteria, such as date, time, availability and priority. We implement **2 classical scheduling algorithms**: FCFS and Priority, plus **another algorithm invented by ourselves**: **Long job first and auto reschedule short jobs,** named **New algorithm**, to achieve better time utilization.

4. **Memory Management**
   Each day has 5 valid time slots for users to arrange their time, which is similar to **MVT** algorithm in Memory Management. When a request comes, we check if relevant time slots are available to use. If not, we also implement an automatical reschduling algorithm, which resembles the **First-Fit** algorithm in memory allocation.This algorithm finds the first time slot that is big enough to hold the event and reschedules the event there.

5. **File Management**
   APO needs to both read batch file containing the request and also keyboard input. When reading files, we need to use relevant file Library Calls introduced in Lab5 such

as fopen(). Besides reading, we also need to write our scheduled timetable, rejected requests and performance analysis into a new file.

# Part III Algorithm Concept

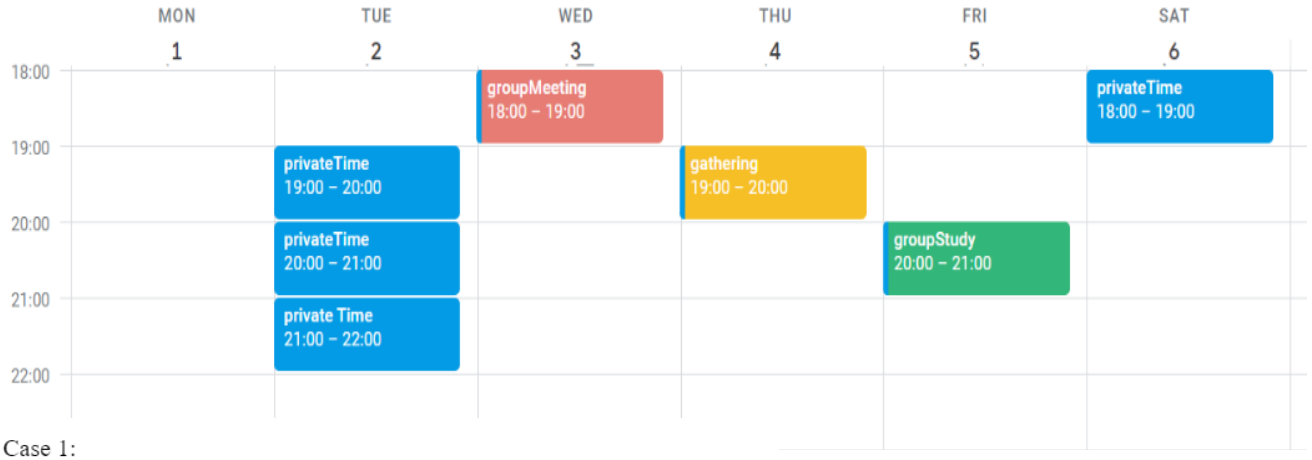
## 1. First-Come-First-Serve(FCFS)

First-Come-First-Serve (FCFS) CPU Scheduling is one of the simplest scheduling algorithms used in operating systems. In this algorithm, the processes are executed in the order they arrive in the ready queue. The process that arrives first in the ready queue is executed first by the CPU, and the next process in the queue is executed after the completion of the current process.

The FCFS algorithm is a non-preemptive scheduling algorithm, which means that once the CPU starts executing a process, it will not interrupt it until it completes its execution. This can lead to a problem called "convoy effect," where a long process may hold up the CPU, causing all the other processes to wait even if they have a shorter execution time.

The FCFS algorithm is easy to implement. However, it may not be suitable for systems with short processes and long processes, as the long processes may monopolize the CPU and cause the short processes to wait for a long time.

In our implementation of th\xe FCFS algorithm. We basically follow the style of it in CPU Scheduling, instead in our program, the “CPU” is actually a calendar time slot.




Case 1:
1.New Request: groupMeeting lucy 20230501 1900 2.0 john mary
2.checking timetable:
if available → schedule new events
if not → reject this event
3.No in this case: then reject

Case 2: New requests:
gathering john 20230502 2100 1.0 mary lucy
checking timeable
yes in this case, then accept

```
for (i = startTime; i < startTime + duration; ++i)
{
     if (timeTable[date][i] !=-1 )
      {
            rejectFlag = 1;
            break;
      }
}
```

## 2. Priority

Priority scheduling is a common algorithm used in operating systems to schedule processes for execution. It assigns a priority to each process, and the process with the highest priority is executed first.

There are two types of priority scheduling algorithms:

**Non-preemptive Priority Scheduling:**

In this algorithm, once a process is assigned the CPU, it continues to run until it completes its task or voluntarily releases the CPU. The scheduler selects the next process to run based on the priority of the waiting processes. The representation of priority is different in Unix/Linux and Windows system

**Preemptive Priority Scheduling:**

In this algorithm, a running process can be interrupted and moved out of the CPU if a higher-priority process becomes available. This allows higher-priority processes to execute quickly and ensures that the CPU is always running the most important process.

In both cases, each process is assigned a priority value. The representation of priority is different in Unix/Linux and Windows system

Priority scheduling can be effective in ensuring that important processes are executed quickly and efficiently. However, it can also lead to starvation, where low-priority processes never get a chance to execute if there are always high-priority processes waiting.

In our implementation of the priority algorithm, the execution order follows the prioritized list of privateTime, projectMeeting, groupStudy, and gathering, based on their level of importance.

All user inputs are stored in a char array called "appointment", where each appointment requirement is represented by appointment[i].

Upon recognizing user input for printSchd Priority (indicated by the presence of "P" in the 11th place of the input string), the program creates a new two-dimensional char array called "Pappointments". The program then splits the string in appointment[i] by spaces, checks the first word to identify the appointment type, memorizes the amounts and positions, and re-sorts the "appointment" array. The results are then stored in the "Pappointments" array, codes shown as below:

```
int ptrPrivate = 0;
int ptrProjectMeeting = numPrivate;
int ptrGroupStudy = ptrProjectMeeting + numProjectMeeting;
int ptrGathering = ptrGroupStudy + numGroupstudy;
for (i = 0; i < numAppointments; ++i)
{
    if (appointments[i][3] == 'v')
    {
        strcpy(Pappointments[ptrPrivate], appointments[i]);
        ptrPrivate++;
    }
    else if (appointments[i][3] == 'j')
    {
        strcpy(Pappointments[ptrProjectMeeting], appointments[i]);
        ptrProjectMeeting++;
    }
    else if (appointments[i][3] == 'u')
    {
        strcpy(Pappointments[ptrGroupStudy], appointments[i]);
        ptrGroupStudy++;
    }
    else if (appointments[i][3] == 'h')
    {
        strcpy(Pappointments[ptrGathering], appointments[i]);
        ptrGathering++;
    }
}
```

### 3. PrintSchd ALL

When printSchd All, the user can choose to print out the schedule results corresponding to all the algorithms, including the recommendation results. and user can make a selection from them,

# Part IV Our new scheduling algorithm

## Long Job First + Autoreschedul Short Jobs

**Definition:** Sorting the appointments based on their duration value, then automatically rescheduling the rest appointments.

**Long Job First Implementation**:

This code block sorts the appointments based on their duration (stored in value) and saves them in Lappointments in descending order. It also calculates the number of appointments for each duration (numONE, numTWO, etc.), and uses these counts to determine the appropriate starting slot for each duration category. The appointments are then copied into Lappointments starting from the appropriate slot for their duration category.

```
void newsort(){}
```

**Autoreschedule Short Jobs:**

After scheduling the events that take longer time first, the system will automatically schedule the events that take shorter time and use the **First-Fit** method to insert the corresponding time to ensure that most of the events will have gaps to be executed.

Based on the information provided, it appears that the New algorithm has a unique characteristic in which the user can only choose from integer time lengths ranging from one to five. When using the New algorithm for rescheduling, each time length is rescheduled separately, implying that the New algorithm may handle each task or job separately, rather than treating them as a batch or group. This could potentially lead to more flexibility in scheduling and allow for better optimization of resources, but may also result in a more complex scheduling process

# Part V Reschedule

The program has set up a function to handle the automatic reschedule function.

It seems that the system has three algorithms: FCFS, Priority, and New algorithms. When the user requests to output the scheduling results for these algorithms, the default order is FCFS. All instructions are stored and read in Lappointment. To use this function, different algorithms are identified to import different sequence results. The system then determines whether the keyword is New0, Priority, or FCFS and performs analysis and transfer accordingly.

```
void reschedule(char app[][255],char keywords[]){
```

Iterate through all appointment records:

```
for(k=0;k<numAppointments;k++)
```

If the system finds that the location is not empty, it means that the appointment is rejected by the system.

```
if (rejectTable[k]!=0)
```

Extract each element of this reservation string and put it into data:

```
memset(data, 0, sizeof(data));
            setData(app[k], strlen(app[k]), data);
```

Total 31 days per month:

```
for(i=1;i<=31;i++)
```

The time slots which can be scheduled is from 18:00 to 23:00.

```
for(j=18;j<23;j++)
{
if(count == data[4])
{
          found = 1;
          break;
       }
       if ((ac_people[i][j]&data[5]) == 0)
           count++;
       else
       }
            count = 0;
      }
      if (found == 1)
           break;
```

If the entire schedule is traversed and there is still no suitable date the loop is automatically terminated. So use if to determine if it is because the right time has been found.

```
if (i<=31)
```

Empty the original reject table. Post successful appointments advice to the system and send them to users. Since the new algorithm has been reordered, it is not necessary to return the result to the system through a new communication, but in the case of the other two algorithms the suggested result needs to be returned, so there is a judgment on the keyword at the end.

```
if (strcmp(keywords,"NEW0")!=0){
```

# Part VI Software structure of the system

## Operating System Services

**User command interface:**

Our program runs in command-line interface of Linux system, user could input request and acquire relevant services through command line information input.

**Program execution**

Our program relies on operating system to load and run. When a user launches an application or executes a program, the operating system is responsible for managing the various resources required by the program and ensuring its proper execution. OS also takes care of different program execution stage, including loading, linking, initialization, execution and termination.

**I/O operations**

I/O (input/output) operations in computer systems refer to the process of transferring data between a computer and an external device or storage medium. I/O operations can involve reading data from a device or storage medium, writing data to a device or storage medium, or both.

The program reads the file when doing the batch input,  write each request into an All request file, when printing, the APO will check whether the users' appointment is acceptable, the appointment will be written in the different scheduling file, and also output the reject and performance file.

**File-system manipulation**

A process may need to read and write files and directories, create and delete them, search them, get file information, access permission management.

In the program, `fopen()` is used to open each file. `flose()` is used to stop communication with a file. `fgets()` is to get the content of the file. `fprintf()` is used to write contents in a file.

**Error detection**

**There are seven kinds of error need to be handled:**

process control error and file manipulation error.

1. when the file cannot be correctly open, the system will exit.

```
 if (rejectFile == NULL)
{
    printf("Open file failed\n");
    exit(1);
}
```

```
 if (performancefile == NULL)
{
    printf("Open file failed\n");
    exit(1);
}printf("-> [Exported file: %s]\n", fileName);
```

2. when creating pipe, means the pipe is not successfully created, the system will exit.

```
  if (n <= 0)
{
    printf("pipe error\n");
    exit(1);
}
```

4. when using the pipe to communicate, there may develop error. The System will exit.

```
for (i = 0; i < numUser; ++i)
   {
       if (pipe(cmd[i]) < 0 || pipe(rsp[i]) < 0)
       {
           printf("User %d Pipe Failed\n", i + 1);
           exit(1);
       }
   }
```

5. fork error.

```
if (fPID < 0)
{
    printf("User %d Fork Failed\n", i + 1);
    exit(1);
}
```

6. Another kind of implementation when the file cannot be opened.

```
if (!file)
{
    perror(path);
    return EXIT_FAILURE;
}
```

7. preventing confusion in reading if there is a deadlock issue.

   Children can only read **accept/reject signal** from parent after they report back their accept/reject information. We added some conditional statements to prevent the child process from reading incorrect information.

**Protection and security**

OS separates the memory space of each process, preventing users interfere and writing into other's memory space, to protect the data safety of each user process.

**The application has eleven functions.**

init() is used for initialization.

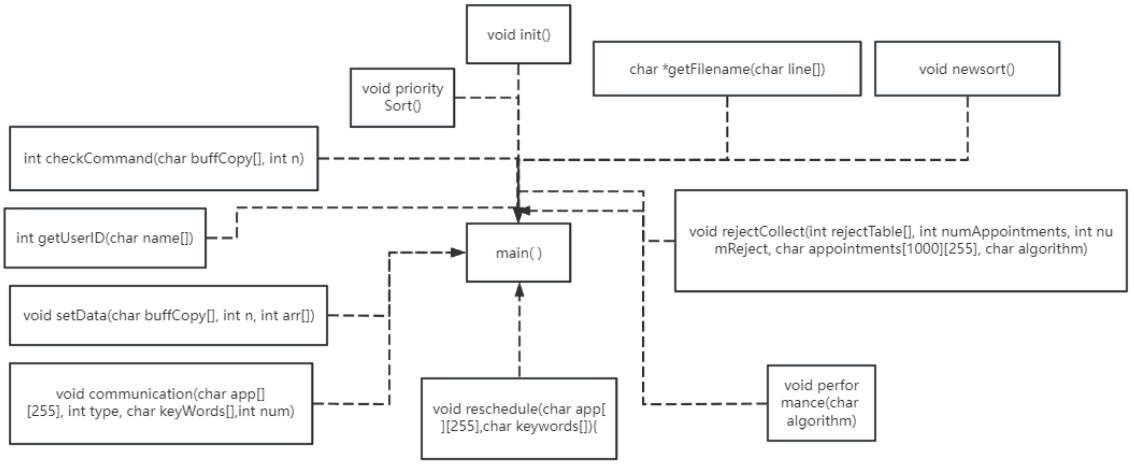
checkCommand is used to check the users' input.

setData is used to get specific contents of each appointment.

newport(), prioritySort(), and reschedule() are used to complete the algorithms.

Performance is used to test the result of each algorithm.

getUserID and getFilename are called when the application needs to obtain users' information and the content of a batch input file.

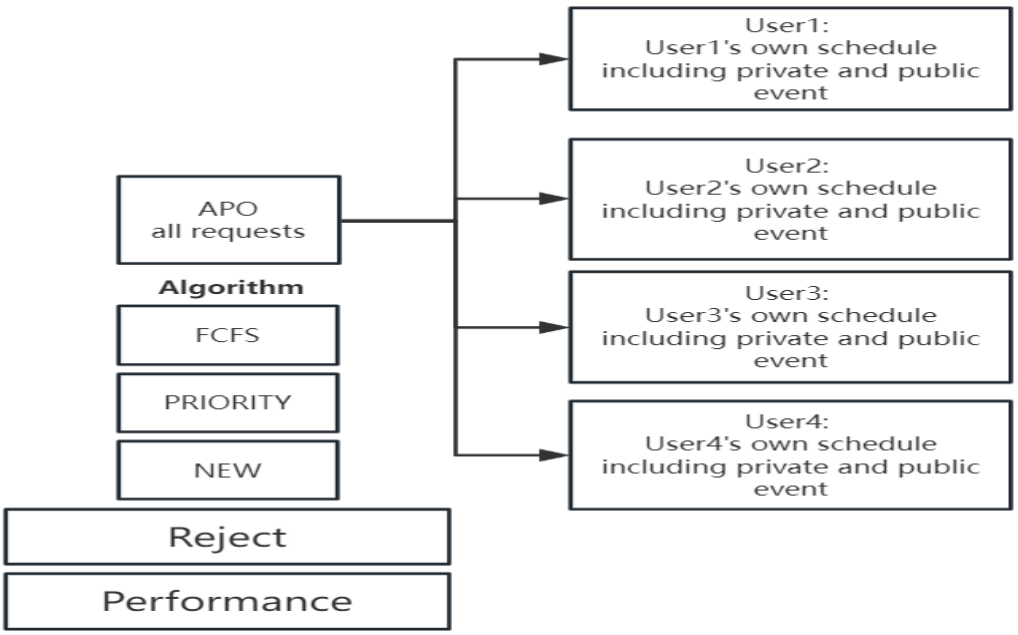



**The Parent and Children Structure**

After receiving the name of users, the parent process will generate different children process by using fork() to process each users' schedule.

Though, APO mainly handles data from different algorithms, gets the time the user is available through pipe(), and then returns to the user whether the activity can be scheduled successfully or not. What will be stored in children process is the schedule information for each user.

Example:

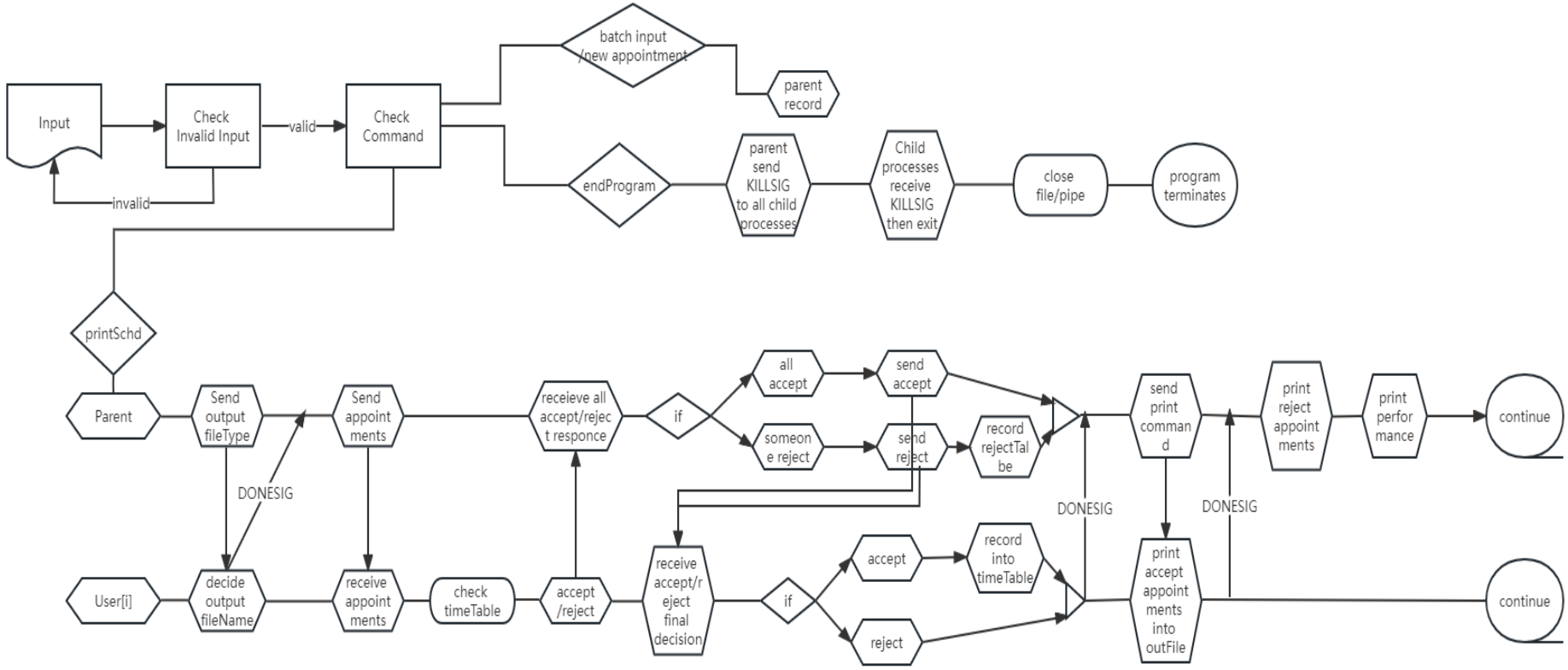



**Variables**

- The variables globalStartTime and globalEndTime are likely used to be unchanged at the beginning of the program initialization.
- the process can only accept **1000** appointments. Because we use char[1000][255]. each string will save each command.
- The variables `requestFile, FCFSfile,` and `priorityFile` are file pointers used to access input and output files.
- The arrays `fileNames, fileTypes, commands, KILLSIG, ACCEPTSIG, REJECTSIG, PRINTACCEPTSIG, DONESIG, userName, appointments, RESCHEDULE, and INITSIG` contain string values used as input and output file names, command strings, and signal strings.
- The variables person_ac_num, numAppointments, numUser, cmd, rsp, fileName, data, rejectTable, i, j, k, numFile, reject_num, accept_num, timeu, and ac_people are used to store various data points and information.
- The arrays `Pappointments` and `Lappointments` contain appointment data processed by different algorithms.

```
int globalStartTime, globalEndTime;
FILE *requestFile = NULL, *FCFSfile = NULL, *priorityFile = NULL;
char buff[260], buffRsp[260], fileNames[5][30], fileTypes[4][30],
commands[9][20], KILLSIG[10], ACCEPTSIG[6], REJECTSIG[6], PRINTACCEPTSIG[12],
DONESIG[4], userName[10][20],
appointments[1000][255],RESCHEDULE[20],INITSIG[20];
int person_ac_num[10];
int numAppointments;
int numUser;
int cmd[10][2], rsp[10][2];
char fileName[30];
int data[6];
int rejectTable[1000];
int i, j, k;
int numFile = 0;
double reject_num,accept_num;
double timeu[10];
int ac_people[40][25];
char Pappointments[1000][255];
int numONE = 0, numTWO = 0, numTHREE = 0, numFOUR = 0, numFIVE = 0;
char Lappointments[1000][255];
```

General Program Structure:



# Operating System Structure

# System call

### Process control

We create 3-10 child processes using `fork()` system call, according to the number of users. When a child process encounter some problems, it exit through `exit()` with an non-zero return value. We use a set of signals to achieve the control for parent over children. For terminate process, after receiving a signal called "`KILLSIG`", child will report back with a "`DONESIG`", denoting it receives the message, then it will terminate with a zero return status. In execution, parent uses a series of signals including `PRINTACCEPTSIG,INITSIG,REJECTSIG` and `DONESIG` to control and manipulate the behavior of children.

### File management

We uses `fopen()` together with other opening flags such as "`w+`", "`a`" and "`r`" to control the read and write status of files. We use `fprintf()` library call to write into the outputted files and uses `fgets()` to get the line from the input files. Besides, when closing file, we uses `fclose()`.

### Device management

The program interacts with the device mainly by manipulating local files.
**Request**: In a command line interface (CLI), device management requests can be made using commands that allow the user to interact with the operating system and access devices.

### Information maintenance

Information maintenance refers to the process of managing information related to system resources, such as files, processes, and devices. The operating system uses various data structures to maintain information about these resources and to manage their usage. For example, read the local file to do the batch input, using pipe to satisfy the communication between children and parent.

**Communication**

We create two sets of pipe `cmd[10][2]` and `rsp[10][2]` to set up the bi-dirctional connection with children. System call `pipe()`create the pipe. We adopt a STAR communication for parent and child, in which all messages from child can only be sent to parent, not to other child. Both appointment and control signal are transmitted through these pipes. A signal, `DONESIG` is used by child to prevent parent overwhelming child, and also prevent the overflow of pipe buffer. `read()`and `write()`system calls are used to transmit information.

**Mutual Exclusion:**

Mutual exclusion refers to a technique used in operating systems to prevent multiple processes from accessing shared resources simultaneously. In the appointment organizer, we can apply mutual exclusion techniques to various shared resources, including:

Shared appointment resource:

1. To prevent multiple child (user) processes from modifying the same appointment at the same time, we can assign appointment modifying tasks to the parent process. Child processes will not have the right to modify the appointment.

Shared file resource:

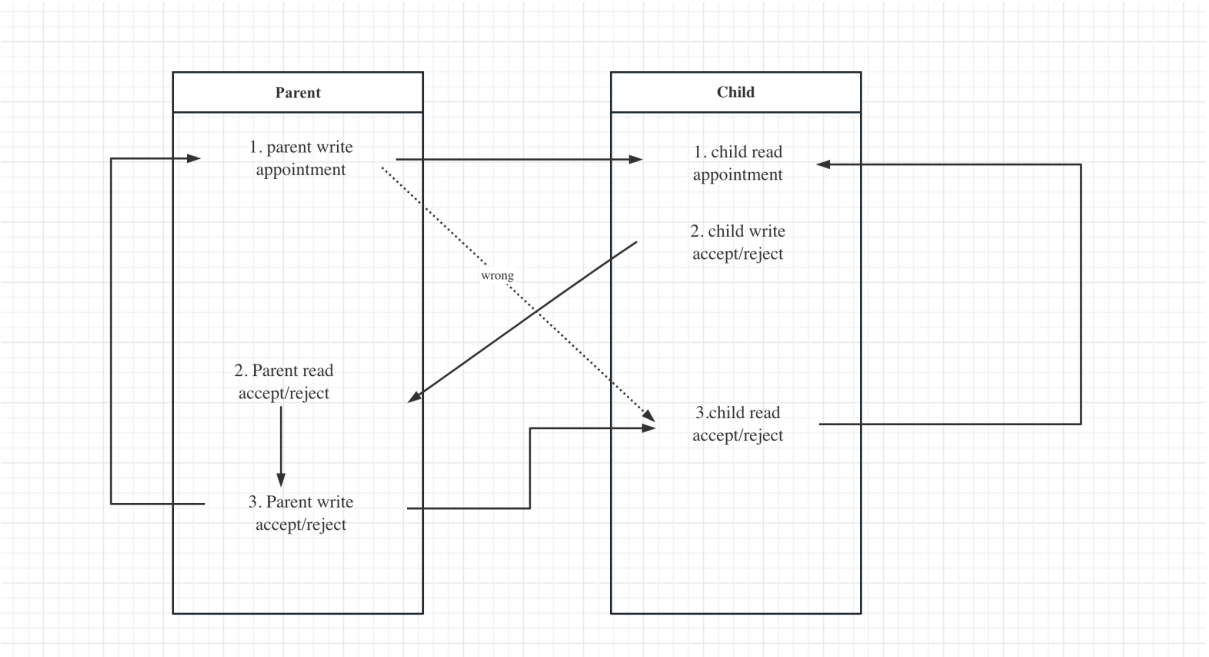
2. To avoid confusion caused by multiple child processes outputting to the same file simultaneously, we can set `PRINTACCEPT` and `DONESIG` (print signal and done signal). These signals will help the parent process control the working order of child processes. Each child process needs to send a `DONESIG` through the pipe after finishing printing. The parent will only allow the next process to print (send `PRINTACCEPT` signal to the particular child process through pipe) when it receives the `DONESIG` from the previous child (user) process.

Shared pipe resource:

3. When a process is reading from a pipe, new data being written to the pipe can lead to confusing readings. To avoid this, we can make full use of `DONESIG`. The parent process will not move to the next appointment until it has received all `DONESIG`s from all related child processes.

**Deadlock:**

We deal with deadlock problem with `DONESIG`. The diagram below illustrates the consequence without a `DONESIG`.




From the above diagram, we can see if there is no `DONESIG`, parent will send the next appointment before child process has finished the previous one. Then, when parent process write the next appointment to the pipe, the other side of pipe may be reading the `REJECTSIG/ACCEPTSIG` from the pipe. It leads the confusion reading. Since the next appointment is already read and the pipe is empty, the child process can not read the appointment when it should do. Then, child process is waiting for parent's writing, and parent process is waiting for the feedback of child process. Deadlock appears in this case.

**Synchronization:**

Semaphores are implemented internally by pipe itself, because the default setting of Unix/Linux's interprocess communication is blocking reading and non-blocking writing. Child and parent process and only read only if there are messages inside pipe.

In our implementation, parent process is itself a monitor, which uses a series of signals to control its children.

Messaging passing technique is widely used in our program and serves as core function in our programs running.

**Initialization:**

after the system begins, using the initialization function which will be described in details below.

First, check whether the user is included in the range ( between 3 and 10).

```
int main(int argc, char *argv[])
{
    if (argc < 6 || argc > 13)
    {
        printf("The number of users is out of range (not between 3 and 10)\n");
        return 0;
    }
    // initialize some data
    init();
    // get start-time, end-time and user-information from the arguments
    globalStartTime = atoi(argv[1]);
    globalEndTime = atoi(argv[2]);
    int i, j;
    for (i = 3; i < argc; ++i)
    {
        argv[i][0] = argv[i][0] - 32;
        strcpy(userName[numUser], argv[i]);
        numUser++;
    }
```

Since the input follows the format:

```
21094724d-apollo:home:/21094724d$ gcc G01_APO.c -o apo
./apo 20230501 20230531 john mary paul lucy lisa mike michael lily james robert
```

YYYYMMDD YYYYMMDD User1 User2 User3…

Input: Read the command at the beginning. Save the global start time, the global end time as the integers. Save the user in the 2 dimensional char array, and capitalizing the first letter of the user name.

The system begins by receiving the start and end dates of the schedule, as well as the total number of users and each user's name. Since the user input names are not in formal case like PaUL or john, the system converts them to uppercase. Using `fork()`, the system generates different child processes based on the number of users, ranging from 3 to 10.

Each child process stores the information for each user. The parent process acts as the central processor for the program. When a user makes an appointment request, after log the request, parent sends the request to child to check if the user is available at this time. If the requested time slot is available, the appointment is accept by user, then the user reports back with "`ACCEPTSIG`". If not, it reports back with "`REJECTSIG`".

## Function 1: checkCommand

This function takes in a command string and its length as input. It checks the command against a list of predefined commands, and returns the index of the command if it matches with one of the predefined commands. If it does not match any of the commands, it returns -1. The function does this by first extracting the first word from the command string and then comparing it with the pre-defined commands using the strcmp() function.

```
int checkCommand(char buffCopy[], int n)
{
    char tmp1[255];
    memset(tmp1, 0, sizeof(tmp1));
    int i, j;
    for (i = 0; i <= n; ++i)
    {
        if (buffCopy[i] == ' ' || i == n)
        {
            if (i == n)
                strncpy(tmp1, buffCopy, i + 1);
            else
                strncpy(tmp1, buffCopy, i);
            for (j = 1; j < 9; ++j)
            {
                if (strcmp(tmp1, commands[j]) == 0)
                    return j;
            }
            return -1;
        }
    }
}
```

## Function 2: `init`

Initialization

This function initializes the global variables used in the program. It clears the appointments array and sets the counters for the number of appointments and number of users to zero. It also initializes the command strings and file types and file names used in the program. Lastly, it sets the signal strings used for communication between the processes in the program.

```
void init()
{

    memset(appointments, 0, sizeof(appointments));
    numAppointments = 0;
    numUser = 0;
    strcpy(commands[1], "endProgram");
    strcpy(commands[2], "printSchd");
    strcpy(commands[3], "priority");
    strcpy(commands[4], "privateTime");
    strcpy(commands[5], "projectMeeting");
    strcpy(commands[6], "groupStudy");
    strcpy(commands[7], "gathering");
    strcpy(commands[8], "file");

    strcpy(fileTypes[0], "FCFS");
    strcpy(fileTypes[1], "Priority");
    strcpy(fileTypes[2], "NEW");
    strcpy(fileTypes[3], "ALL");
    strcpy(fileTypes[4], "NEW0");

    strcpy(fileNames[0], "G01_XX_FCFS.txt");
    strcpy(fileNames[1], "G01_XX_PRIORITY.txt");
    strcpy(fileNames[2], "G01_XX_NEW.txt");
    strcpy(fileNames[3], "G01_XX_ALL.txt");
    strcpy(fileNames[4], "G01_XX_NEW0.txt");

    strcpy(PRINTACCEPTSIG, "printAccept");
    strcpy(INITSIG, "initialization");
    strcpy(REJECTSIG, "reject");
    strcpy(ACCEPTSIG, "accept");
    strcpy(DONESIG, "done");
    strcpy(KILLSIG, "kill");
}
```

### Function 3: `getUserID`

The system will give each user an id from initialization, and according to the order they are entered, each time the user's information is called, it is based on this id. the function is to get the corresponding id when a name is received.

```
int getUserID(char name[])
{
    int i;
    if (name[0]<='z'&&name[0]>='a') name[0]-=32;
    for(j=1;j<strlen(name);j++){
        if (name[j]<='Z'&&name[j]>='A') name[j]+=32;
    }
    for (i = 0; i < numUser; ++i)
    {
        if (strcmp(name, userName[i]) == 0)
            return i;
    }
}
```

This is because during initialization, we changed the name format of each user to the correct format, but each time the user enters a new command, the username used will be saved and called by us directly in string format, in order to prevent not being able to find the user by the wrong formatted name, so the formatting should be done before querying the corresponding userID. However, it does not change the contents of the original command stored.

### Function 4: setData

This function, `setData()`, is used to extract and process appointment data from a character array. The function takes in three parameters: `buffCopy`, an array of characters that stores the input command, n, the length of `buffCopy`, and arr, an integer array that stores the processed appointment data.

The function iterates through the input command, and whenever it encounters a space character, it extracts the data between the previous space character and the current one. The extracted data is then stored in a temporary array, tmp, and processed based on its position in the command.

If the position is 0, the function calls `checkCommand()` to determine the type of appointment. If the position is 1, the caller's user ID is extracted and stored in `arr[1]`, and the bit for the caller's user ID is set to 1 in `arr[5]`. If the position is 2, the appointment date is extracted and stored in `arr[2]`. If the position is 3, the appointment start time is extracted and stored in `arr[3]`. If the position is 4, the appointment duration is extracted and stored in

`arr[4]`. If the position is greater than or equal to 5, the callee's user ID is extracted and used to set the appropriate bit in `arr[5]`.

After processing all the appointment data, arr contains the appointment type, caller's user ID, appointment date, start time, duration, and the callees' user IDs.

The codes is as follow:

```
void setData(char buffCopy[], int n, int arr[])
{
    int pre = -1; // the index of the previous space
    int pos = 0;  // the position of the current data in the command line (eg.
pos=0 means current information is for 'command type')
    char tmp[255];
    int i;
    for (i = 0; i <= n; ++i)
    {
        if (buffCopy[i] == ' ' || i == n)
        {
            memset(tmp, 0, sizeof(tmp));
            if (i == n)
                strncpy(tmp, buffCopy + pre + 1, i - pre);
            else
                strncpy(tmp, buffCopy + pre + 1, i - pre - 1);
            if (pos == 0)
                arr[0] = checkCommand(buffCopy, n); // appointment type
            else if (pos == 1)
            {
                arr[1] = getUserID(tmp); // caller userID
                arr[5] = 1 << arr[1];
            }
            else if (pos == 2)
                arr[2] = atoi(tmp) % 100; // appointment date
            else if (pos == 3)
                arr[3] = atoi(tmp) / 100; // appointment start time
            else if (pos == 4)
                arr[4] = (int)(atof(tmp)); // appointment duration
            else if (pos >= 5)
            {
                arr[5] = arr[5] + (1 << (getUserID(tmp))); // a binary number to
store the callees (eg. 5=101 means callees are user0 and user2)
            }
            pos++;
            pre = i;
        }
    }
}
```

## Function 5: `rejectCollect`

This function is used to generate a rejected list of appointments based on the rejection table. The rejection table is an integer array that stores the rejection status of each appointment, with 0 indicating acceptance and 1 indicating rejection. The function takes as input the rejection table, the total number of appointments, the number of rejected appointments, the appointment list, and the algorithm used for scheduling (FCFS, Priority, or All).

The function starts by initializing variables and creating a new file for the rejected list, with a filename based on the algorithm used for scheduling and the current file number. It then iterates through the appointment list and checks if each appointment has been rejected, if so, it writes the appointment information to the reject file. Finally, it writes a summary of the rejected appointments to the file and closes the file. The rejected list includes the appointment number, appointment type, appointment date, caller ID, start time, duration, and a binary number to indicate the callees.

## Function 6: communication

This function communication is responsible for coordinating communication between the parent process and child processes in order to schedule appointments. It takes in an array of appointments, the appointment type, key words, and the number of appointments. It initializes arrays for storing accepted appointment information and clears arrays for storing rejected appointment information. It then iterates through each user and writes the appointment type to their respective pipe, reads the response from the pipe, and checks whether the appointment is rejected. If an appointment is rejected, it increments the reject counter, sets the corresponding value in the reject table to 1, and sends a rejection signal to each user's pipe. If an appointment is accepted, it records the accepted appointment information and sends an acceptance signal to each user's pipe. Finally, it exports the appointment schedule to a file, collects rejected appointments into a file, and calculates performance metrics. If the key words are "NEW", it initializes the appointment scheduling system.

# Part VII. Testing cases

**Defining Testing Scope:**

1. Functional Testing:
    a. functionality of each APO features including creating and scheduling event, recording request and rejecting requests
    b. making sure the events are correctly displayed and updated on the outputed files and are associated with correct dates and times.
    c. verify the events can be rescheduled to a proper timeslot when rejected
    d. verify that user can input calendar events in both keyboard and batch file input
    e. verify the calendar can reject request on Sunday and public holiday automatically
2. Performance Testing:
    a. verify that the APO can respond correctly to user actions, even when user has a large number of events and calendars.
    b. verify that the APO can handle the expected number of concurrent users(i.e. 3 to 10 users) without crashing
    c. verify that the APO can successfully reschedule all requests if there are a large number of rejected files
    d. verify that the program can perform well if there are three or more same incoming requests.
3. Compatibility Testing
    a. verify that the APO works correctly on Unix/Linux environment(i.e. apollo and apollo2 in this case)
4. Usability Testing
    a. verify that the APO's output prompt and output files are clear and easy to understand
    b. verify that the APO's input prompt are clear and has no confusion for users

## Testing Cases Development and Testing Procedures:

a. Our testing data aims to cover all the testing scopes specified above to conduct a thorough testing for the project.
b. We assume there are no user input errors, so we also exclude the possible errors in testing data, e.g. the date is out of specified range, the user is not inside the list of defined users, time is out of range, or wrong event name.
c. We develop a range of testing data with different numbers of users, different numbers of requests.
d. Users ranges from **3 to 10. The user input** is assumed to have **no errors. Weekend and holiday** should **not** be included in the schedule**.** The date is assumed from 1st May to May 31th.

**File**

- No file with the same name should present in the working directory
- If this txt file is created on a system running windows, it should first use the dos2unix command to change the file format from windows to unix format.

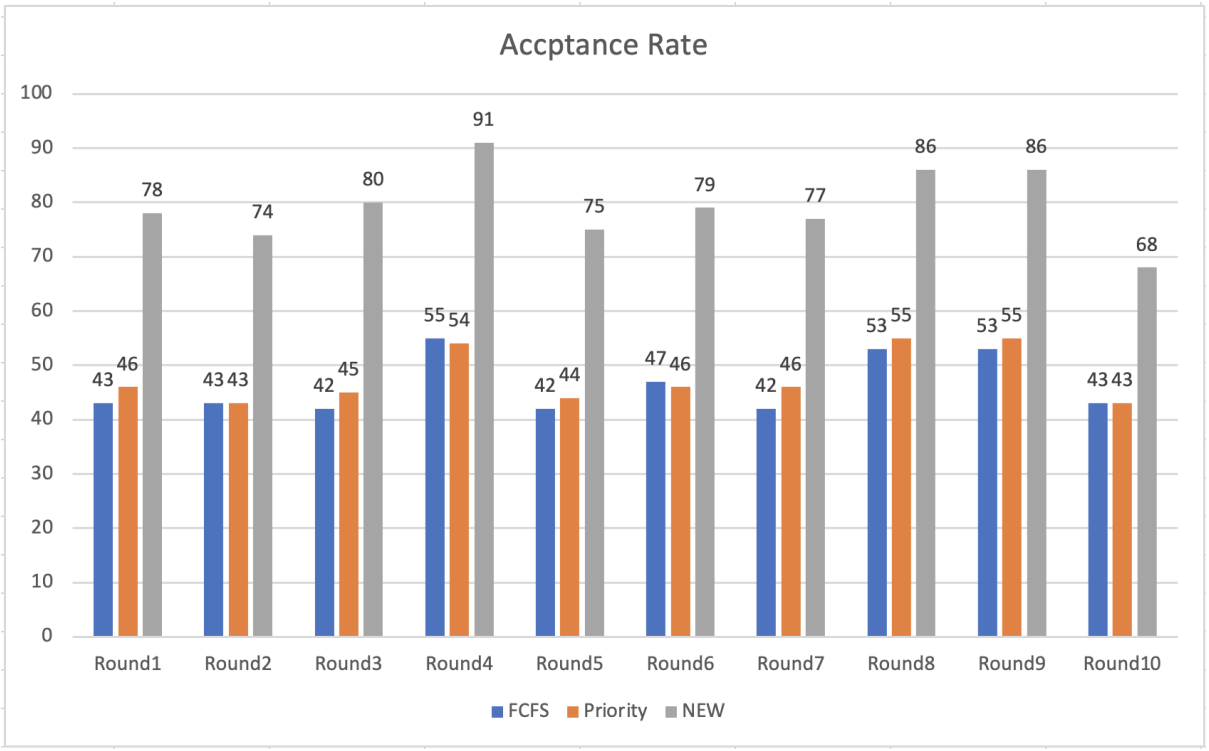
## Testing environment:

| | |
|---|---|
| Operating System: | Unix/Linux |
| Machine: | apollo/apollo2 |

Please see the details of our testing data in "**Appendix**".

# Part VIII. Performance analysis

In order to test the efficiency of different algorithms, we randomly generated 10 different datasets, each consisting of 100 requests with 10 users. We run our program with these 10 datasets, below is the result obtained:




From this chart, we can see clearly that our NEW Algorithm performs much better than both First-Come-First-Serve algorithm and Priority. In each testing round, the acceptance rate of FCFS and Priority only fluctuates around 45%, but mostly, our proposed algorithm can get to up to 91%.

There are several reasons why both FCFS and Priority cannot perform over our algorithm. First, both FCFS and Priority only consider the arrival time or the importance of the event, instead of duration of time. So if any appointments are very long and arrive early or have a higher priority, it may take over the whole day's timeslot. Thus, even though the availability of that day is enough to hold another appointment, it cannot accept that appointment because these time slots are separately distributed. The situation of allocating timeslot behaves similarly to memory allocation in MVT of memory. Besides, in the priority algorithm, if the incoming requests always have a higher priority, the low priority request can never be fulfilled, causing the situation of starvation.

But generally priority outperforms FCFS in our ten test cases, we think the one probable reason is priority algorithm guarantees highly-prioritized events would not clash with other events. In the only cases which priority accepts less appointments, there are possibly two many events with higher priority. In later development, we may consider adjusting the priority of a certain type of event if it occurs too much.

The situation of larger requests which take the middle time slot in a day is similar to internal fragmentation of memory allocation. Our algorithm has a similar concept with the “compaction” used in memory allocation to alleviate this problem. When we schedule the longer appointments, we usually start from the beginning timeslot in a day, which might leave enough time for other appointments to use. Therefore, we can significantly increase the acceptance rate.

# PART VIIII. Program set up and execution

## “How to compile and execute your project?”

**Brief Description**

The application features a schedule generation functionality that accommodates three to ten users. It enables the creation of both individual and team schedules. Users can add an unlimited number of events within their specified timeframes. If there are any time conflicts, the system offers timetable suggestions and reasonable rescheduling options using three different built-in algorithms. Each user can initiate events, and if there are no time conflicts, the event will appear on other users' schedules.

## Setup:

For executing the program apollo or apollo2 can be used.

The provided file G01_APO.c is the source code of the application.

the document of the testing date and source code should be saved in the same place.

Then execute the following codes in the command line:

```
21094724d-apollo:home:/21094724d$ gcc G01_APO.c -o apo
./apo 20230501 20230531 john mary paul lucy lisa mike michael lily james robert
```

It can be assumed that there are 10 users in total.

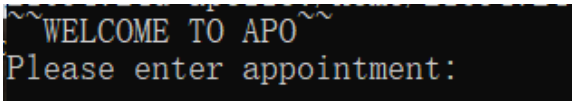
20230501 refers to the year 2023 on May day 1st.

20230531 refers to the year 2023 and May day 31th.

Which follows the format of “year/mouth/day”.

Then the system is initialized, and the users can begin to input their commands.

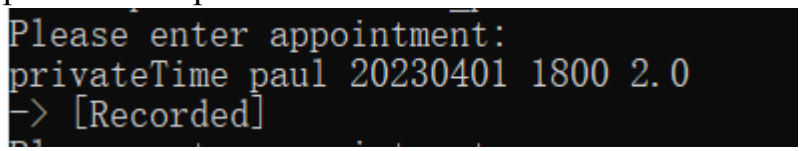
Beginning interface:

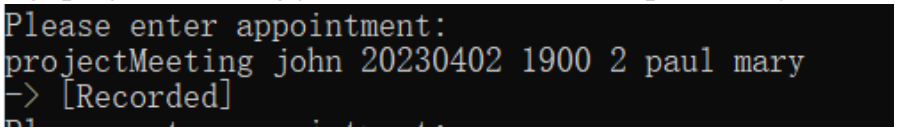



## Execution:

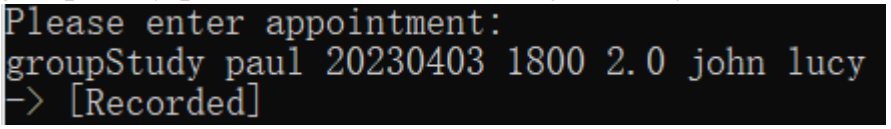
## Command Description

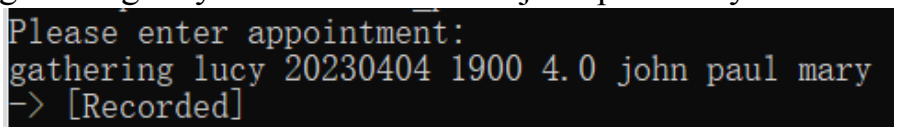
| Command | privateTime |
|---|---|

| | |
|---|---|
| **Format** | privateTime uuu YYYYMMDD hhmm n.n<br>e.g.:<br>privateTime paul 20230401 1800 2.0<br>  <br><br>[privateTime]:command type<br>[uuu] - the name of the caller<br>[YYYYMMDD] - the date of the event<br>[hhmm] the start time of the event<br>[n.n]Duration |
| **Using Method** | The private time is a kind of individual activity, for the user to arrange her or his own time slots.<br>For example, the above command means on April 1st, 2023, 18:00, paul has 2 hours private time. |

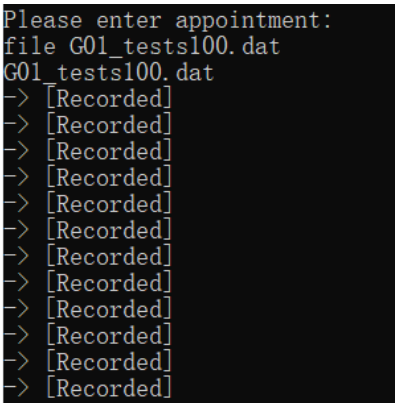
| **Command** | **projectMeeting** |
|---|---|
| **Format** | projectMeeting uuu YYYYMMDD hhmm n.n u1 u2…<br>e.g. projectMeeting john 20230402 1900 2 paul mary<br>  <br><br>projectMeeting paul 20230401 1800 2.0<br>[projectMeeting]:command type<br>[uuu] - the name of the caller<br>[YYYYMMDD] - the date of the event<br>[hhmm] the start time of the event<br>[n.n]Duration<br>[u] the name of users |
| **Using Method** | The project meeting is a kind of group activity for the user as a caller to arrange project meeting and call the other included callees.<br>For example, the above command means on April 2nd, 2023, 19:00, John call a project meeting, and then invites Paul and Mary. The event takes 2 hours in total. |

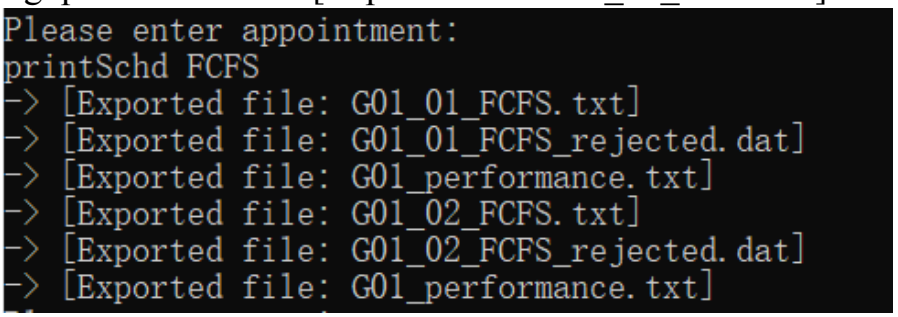
**Sample input in command line:**

| **Command** | **groupStudy** |
|---|---|
| **Format** | groupStudy uuu YYYYMMDD hhmm n.n u1 u2 … e.g. groupStudy paul 20230403 1800 2.0 john lucy<br>  |
| **Using Method** | The group study is a kind of group activity for the user as a caller to arrange group study and call the other included callees.<br>For example, the above command means on April 3rd, 2023, 18:00, Paul calls a project meeting, and then invites John and Lucy. The event takes 2 hours in total. |

| **Command** | **gathering** |
|---|---|
| **Format** | gathering uuu YYYYMMDD hhmm n.n u1 u2 … e.g. gathering lucy 20230404 1900 4.0 john paul mary<br>  |
| **Using Method** | The gathering is a kind of group activity for the user as a caller to arrange group study and call the other included callees.<br>For example, the above command means on April 4th, 2023, 19:00, Lucy calls a project meeting, and then invites John, Paul and Mary. The event takes 4 hours in total. |

| **Command** | **file** |
|---|---|
| **Format** | file xxx<br>e.g: file G01_tests100.dat |

| | |
|---|---|
| |  <br>[file] refers to the command type<br>[xxx] refers to the name of the document<br>In the document, each line should be recognized as one command, and also follows the other commands format. |
| **Using Method** | It is used for batch input. |

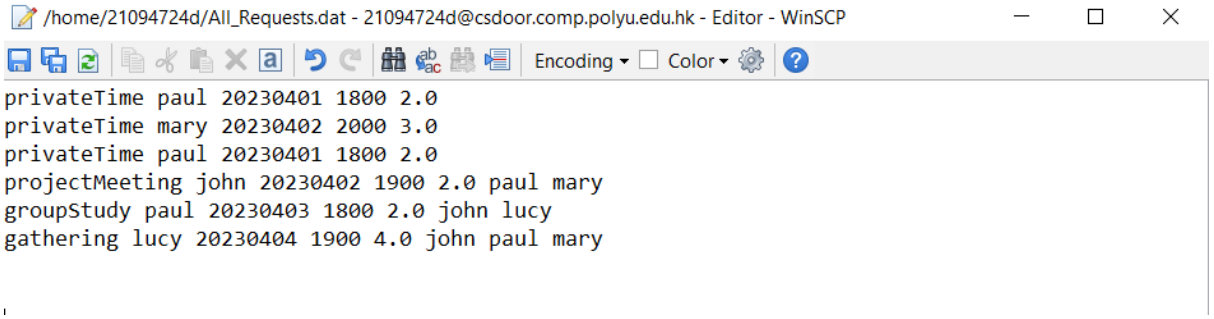
| **Command** | **printSchd** |
|---|---|
| **Format** | printSchd “sssss” / ALL<br>e.g. printSchd FCFS [Exported file: G01_01_FCFS.txt]<br>  |
| **Using Method** | It is used to print the time table and the rejected list after being processed by different algorithms.<br>“ssss” refers to the classification of the algorithm, which can be FCFS, PRIORITY, and NEW.<br>ALL means to print the result of all three algorithms. |

| **Command** | **endProgram** |
|---|---|
| **Format** | endProgram |
| **Using Method** | exit and leave the program. |

## Output Description

| File Type | All Request |
|---|---|
| **Format** | All_Requests.dat |
| **Details** | Log file of all input requests |

### Sample File Content:




| File Type | FCFS Algorithm |
|---|---|
| **Format** | G01_nn_FCFS.txt<br>nn: sequence number of report printed<br>e.g.: [G01_01_FCFS.txt] |
| **Details** | Exported file by command “printSchd FCFS”, which follows the first come first serve algorithm. |

### Sample File Content:

In G01_01_FCFS.txt, the initial commands are recorded for each user.

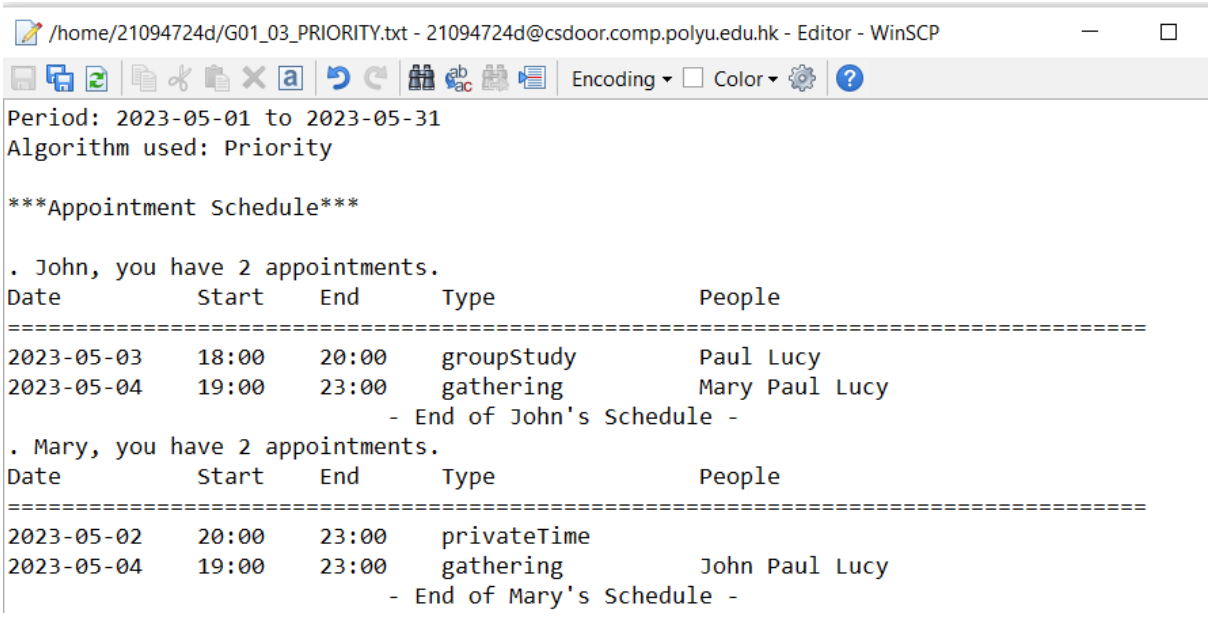
```
Period: 2023-05-01 to 2023-05-31
Algorithm used: FCFS

***Appointment Schedule***

. John, you have 10 appointments.
Date          Start    End      Type              People
================================================================================
2023-05-05    19:00    22:00    groupStudy        Lucy Mike
2023-05-08    19:00    20:00    projectMeeting    Mary Lucy Lisa Mike Michael Lily Robert
2023-05-09    18:00    19:00    projectMeeting    Mary Paul Lucy Mike Michael Lily
2023-05-12    20:00    22:00    privateTime
```

In G01_02_FCFS.txt, the processed results are recorded for each user, meaning that the overlap activity has been deleted and rescheduled. If still there is no free time slot to use, the event then will be rejected and saved in the reject file.

```
Period: 2023-05-01 to 2023-05-31
Algorithm used: FCFS

***Appointment Schedule***

. John, you have 22 appointments.
Date          Start    End      Type               People
================================================================================
2023-05-01    18:00    21:00    gathering          Paul Lisa
2023-05-01    22:00    23:00    groupStudy         Mary Paul Lisa Mike Michael Lily James Ro
2023-05-02    18:00    21:00    projectMeeting     James
2023-05-02    22:00    23:00    groupStudy         Paul Lucy Michael Robert
2023-05-03    18:00    20:00    projectMeeting     Lucy Lisa Mike Robert
2023-05-03    22:00    23:00    projectMeeting     Lucy Lisa Mike Robert
```

| **File Type** | **Priority Algorithm** |
|---|---|
| **Format** | G01_nn_PRIORITY.txt<br>nn: sequence number of report printed<br>e.g.: [G01_01_PRIORITY.txt] |
| **Details** | Exported file by command “printSchd PRIORITY”, which follows the priority algorithm. |

**Sample File:**



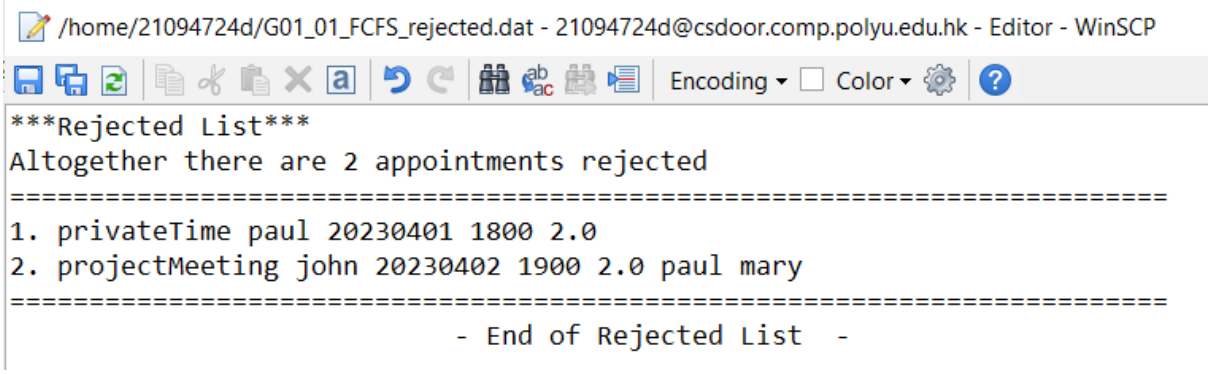
| File Type | **New Algorithm** |
|---|---|
| **Format** | G01_nn_NEW.txt<br>nn: sequence number of report printed<br>e.g.: [G01_01_NEW.txt] |
| **Details** | Exported file by command “printSchd NEW”, which follows the new algorithm. |

**Sample File:**

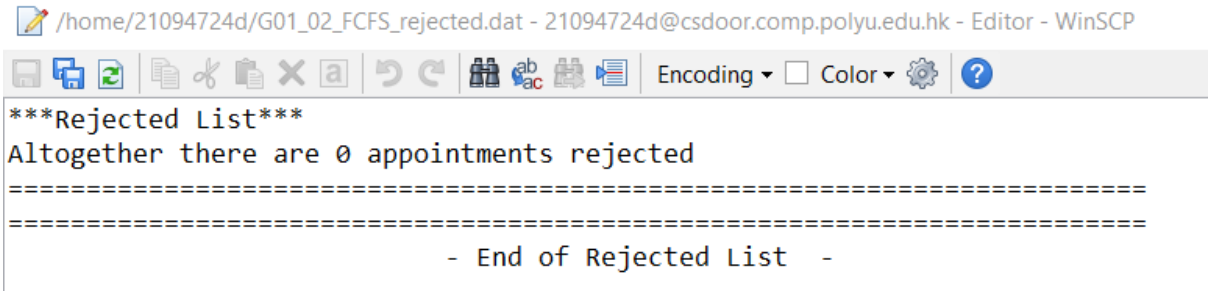
```
Period: 2023-05-01 to 2023-05-31
Algorithm used: NEW

***Appointment Schedule***

. John, you have 17 appointments.
Date          Start    End      Type              People
================================================================================
2023-05-01    18:00    21:00    gathering         Paul Lisa
2023-05-02    18:00    22:00    groupStudy        Mary Paul Lucy Lisa Michael Lily James Ro
2023-05-02    22:00    23:00    gathering         Mary Paul Lisa Michael Lily
2023-05-03    19:00    23:00    gathering         Mary Paul Lucy Lisa Mike Michael Lily Jam
2023-05-05    19:00    23:00    projectMeeting    Mary Lisa Mike Lily James Robert
2023-05-06    18:00    21:00    groupStudy        Lucy Mike
2023-05-06    21:00    23:00    projectMeeting    Paul Lucy Mike Lily Robert
```

| File Type | **Rejected** |
|---|---|
| **Format** | rejected.dat<br>e.g.: [G01_rejected.txt] |
| **Details** | |

In the first generated reject file, it includes the rejected appointments.




The rejected event will be rescheduled, and saved in another reject file, if the new reschedule appointment still has some conflicts of the other events, the new reject list will be saved.
As below, all the reject event are correctly saved.




| File Type | Performance |
|---|---|
| Format | G01_Performance.txt |
| Details | |

```
***  Performance  ***

Total Number of Requests Received: 100
      Number of Requests Accepted: 67 (67.00%)
      Number of Requests Rejected: 33 (33.00%)

Number of Requests Accepted by Individual:
John            -17
Mary            -20
Paul            -26
Lucy            -22
Lisa            -25
Mike            -22
Michael         -20
Lily            -30
James           -21
Robert          -21
Utilization of Time Slot:
John            -32.90%
Mary            -32.90%
Paul            -43.87%
Lucy            -40.00%
Lisa            -42.58%
Mike            -501.29%
Michael         -417.42%
```

# Appendix:

Details of testing process:

**Testing set 1:**

1. two testing files with 4 users and each with 40, 400 requests.
2. date range: 2023.05.01-2023.05.31
3. time range: 18:00 - 23:00 each day

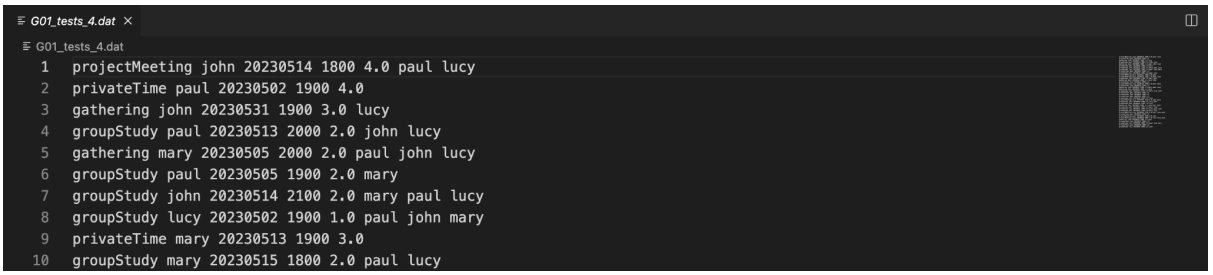
command sequences used for testing:

```
./apo 20230501 20230531 john mary lucy paul
file G01_tests_4.dat
printSchd FCFS
printSchd Priority
printSchd ALL
printSchd NEW

file G01_tests_4_400.dat
printSchd FCFS
printSchd Priority
printSchd ALL
printSchd NEW
```

Test File name:

| | |
|---|---|
| G01_tests_4.dat | 4 users and 40 requests |
| G01_tests_4_400.dat | 4 users and 400 requests |

Data Example:




Testing Result:

*Works well for both files*

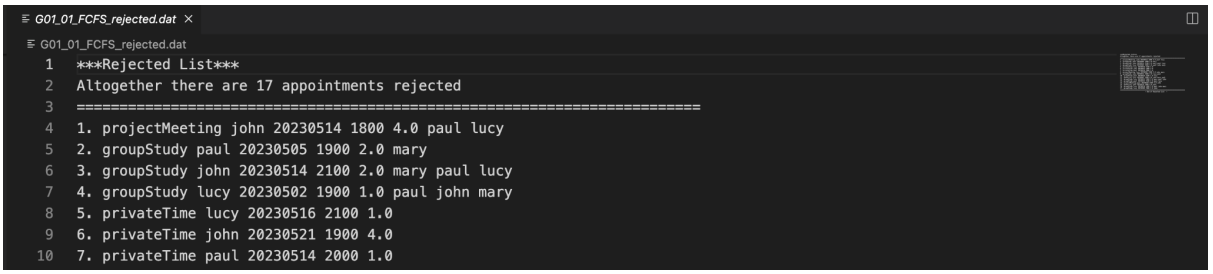
**Testing Result Screenshots:**

Original FCFS:


G01_01_FCFS.txt

```
Period: 2023-04-01 to 2023-04-30
Algorithm used: FCFS

***Appointment Schedule***

. John, you have 11 appointments.
Date          Start    End      Type              People
================================================================================
2023-04-03    21:00    23:00    projectMeeting    Mary Lucy Paul
2023-04-04    20:00    23:00    projectMeeting    Mary Lucy Paul
```

Original Rejected FCFS:


G01_01_FCFS_rejected.dat

```
***Rejected List***
Altogether there are 17 appointments rejected
========================================================================
1. projectMeeting john 20230514 1800 4.0 paul lucy
2. groupStudy paul 20230505 1900 2.0 mary
3. groupStudy john 20230514 2100 2.0 mary paul lucy
4. groupStudy lucy 20230502 1900 1.0 paul john mary
5. privateTime lucy 20230516 2100 1.0
6. privateTime john 20230521 1900 4.0
7. privateTime paul 20230514 2000 1.0
```

Rescheduled FCFS:


G01_02_FCFS.txt

```
Period: 2023-04-01 to 2023-04-30
Algorithm used: FCFS

***Appointment Schedule***

. John, you have 23 appointments.
Date          Start    End      Type              People
================================================================================
2023-04-02    18:00    19:00    groupStudy        Mary Lucy Paul
2023-04-02    19:00    23:00    privateTime
```

Rescheduled FCFS Rejected List:


G01_02_FCFS_rejected.dat

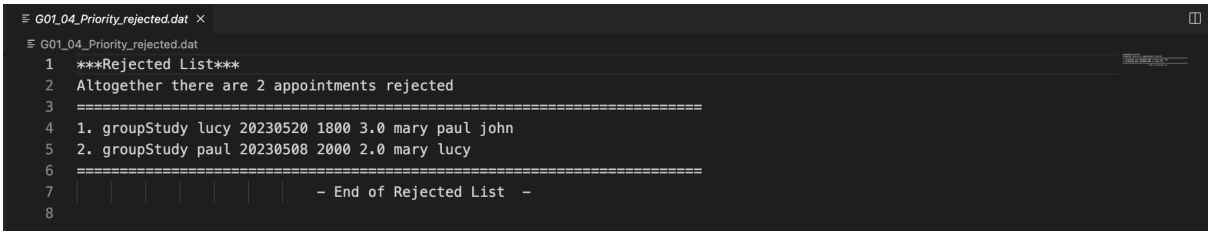
```
***Rejected List***
Altogether there are 0 appointments rejected
========================================================================
========================================================================
                    - End of Rejected List  -

```

Original Priority:

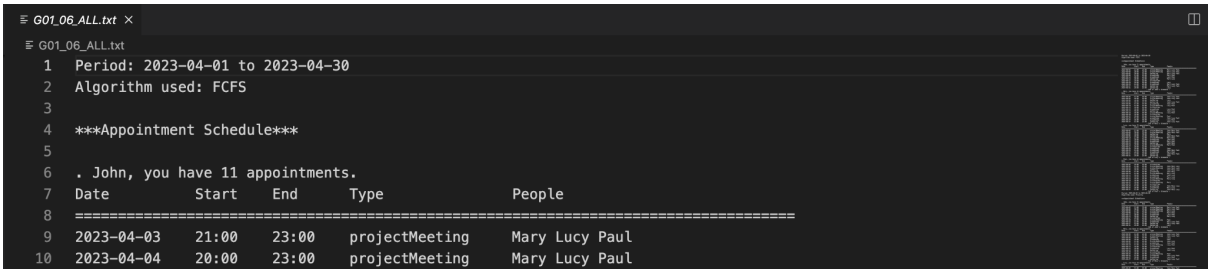

Original Priority Rejected File:


Rescheduled Priority File:

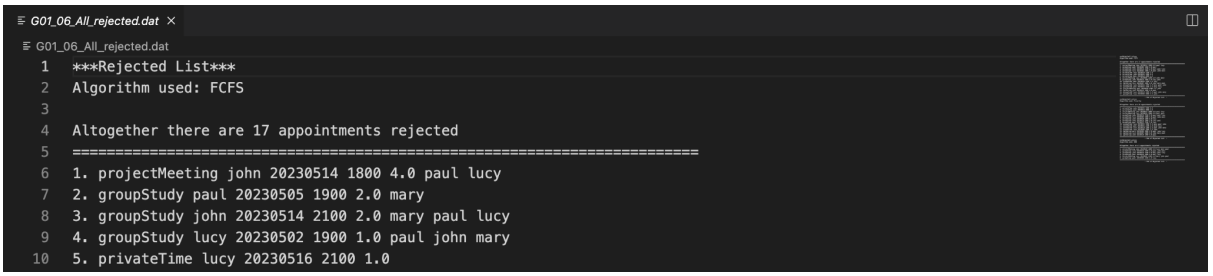

Rescheduled Priority Rejected File:



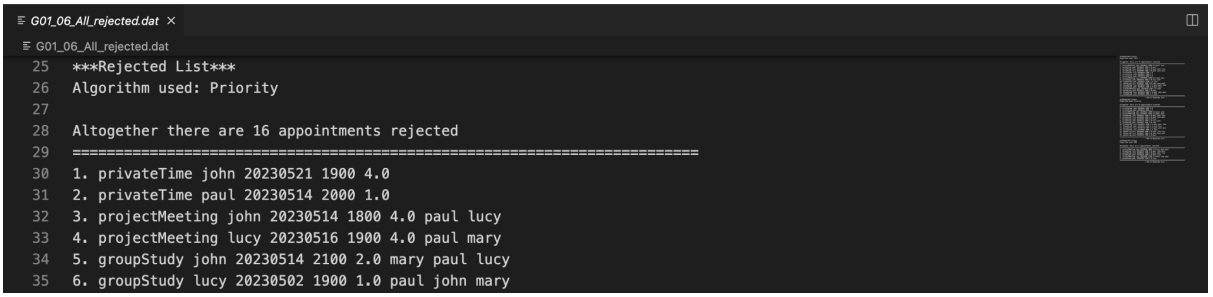

All file accepted request:



All rejected request:
FCFS:



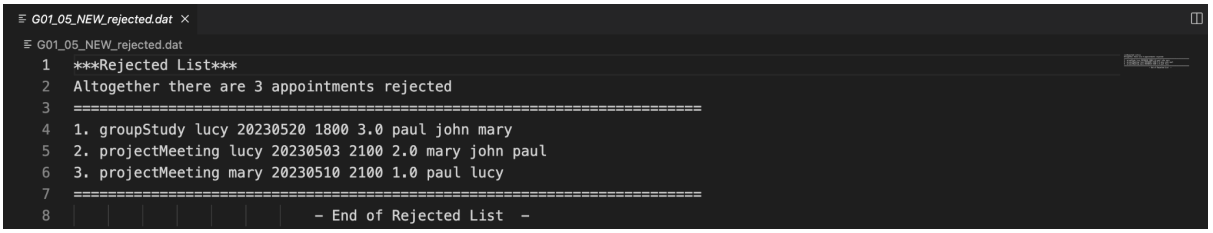

Priority:




NEW:


new algorithm:


new algorithm rejected:



**Testing Set 2:**

1. two testing files with 10 users and each with 100, 1000 requests.
2. date range: 2023.05.01-2023.05.31
3. time range: 18:00 - 23:00 each day

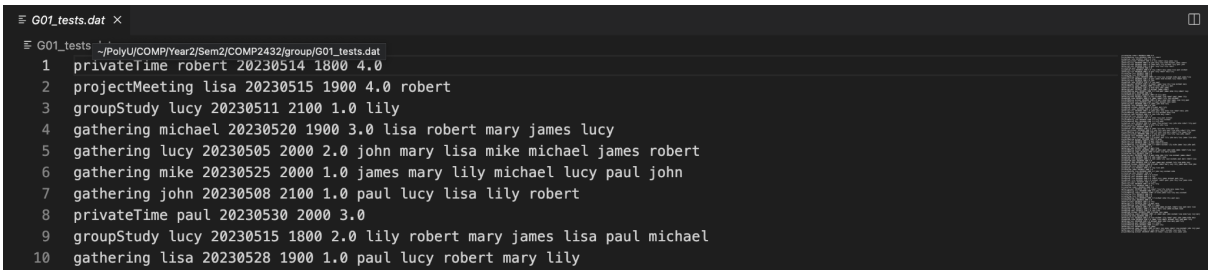
command sequences used for testing:

```
./apo 20230401 20230430 john mary paul lucy lisa mike michael
lily james robert
file G01_tests.dat
printSchd FCFS
printSchd Priority
printSchd ALL
printSchd NEW

file G01_tests_10_1000.dat
printSchd FCFS
printSchd Priority
printSchd ALL
printSchd NEW
```

Test File name:

| | |
|---|---|
| G01_tests.dat | 10 users and 100 requests |
| G01_tests_10_1000.dat | 10 users and 1000 requests |

Data Example:




Testing Result:

*Works well for both files*

Orignal FCFS:

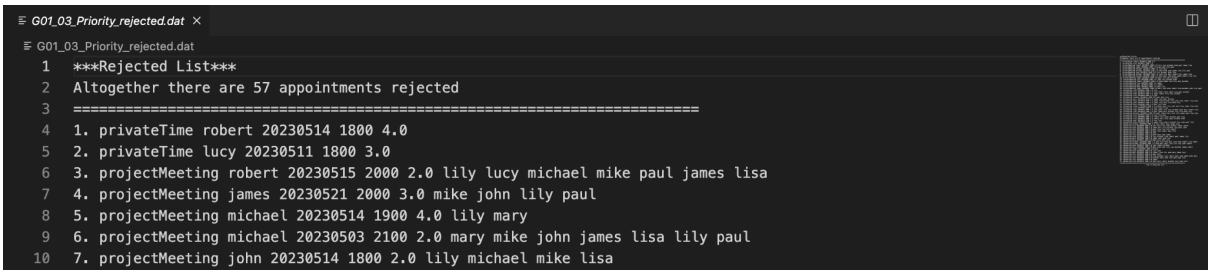

Original Rejected FCFS:


Rescheduled FCFS:

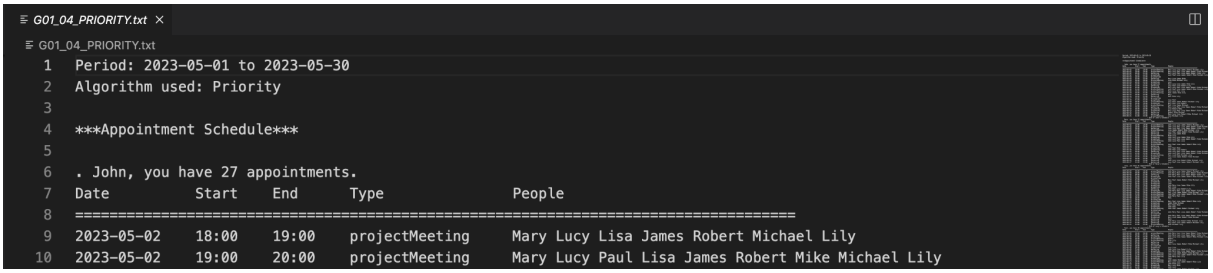

Rescheduled Rejected FCFS:

Original Priority:

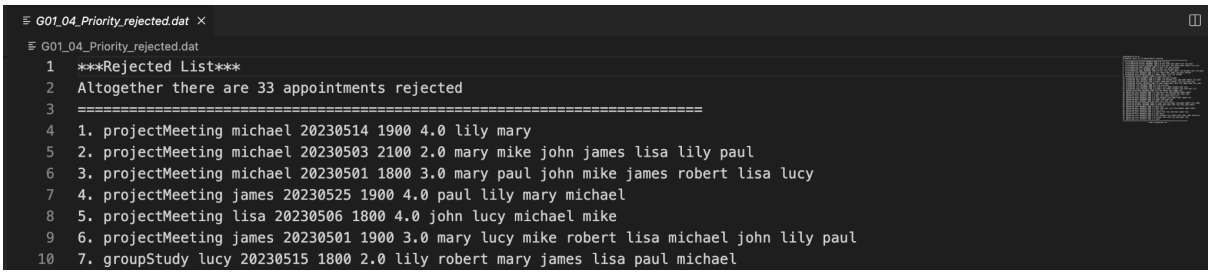

Original Rejected Priority:




Rescheduled Priority:



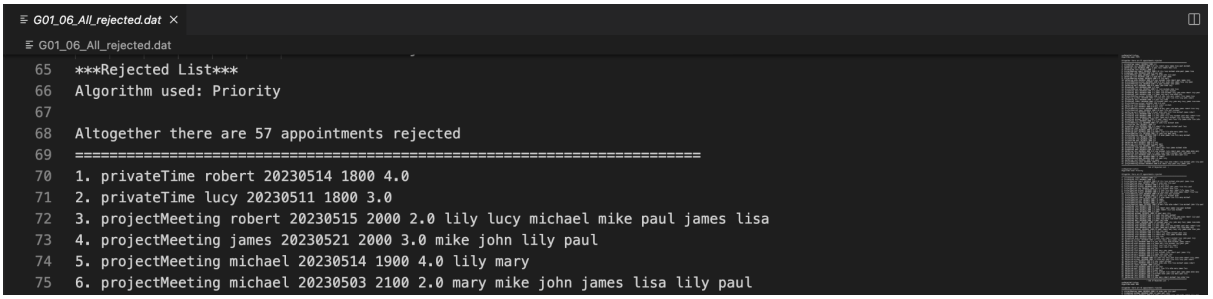

Rescheduled Rejected Priority:



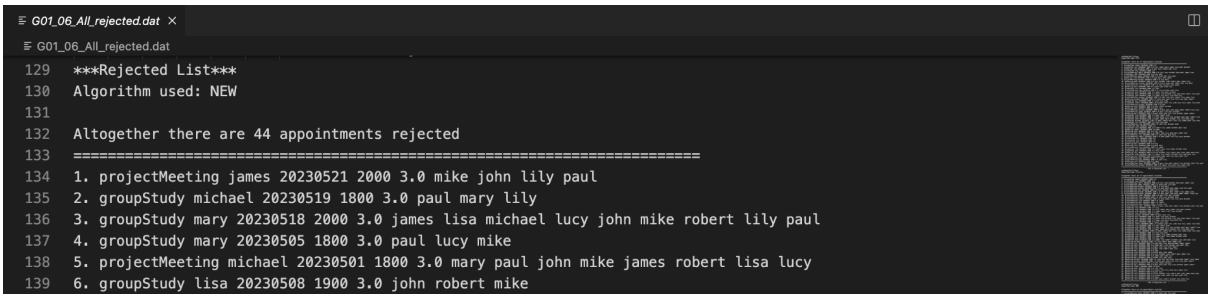

All file accepted request:

All file rejected requests:
FCFS:


Priority:




NEW:




new algorithm:


new algorithm rejected:

# Part X Contribution Form

| Item | Group: G01 | HU Wenqing | LIU Chenxi | ZHANG Tianyi | ZHANG Wanyu |
|---|---|---|---|---|---|
| 1 | Responsbility:<br>List the task(s) that each individual was(were) responsible for | 1. reject implementation<br>2. report writing | 1. auto reschedule implementation<br>2. new algorithm implementation<br>3. performance file output<br>4. print ALL | 1.Data processing(setData function)<br>2.fork/pipe establishment<br>3.communications through pipe between parent and children processes<br>4.FCFS algorithm<br>5.Output tasks of FCFS algorithm | 1. priority algorithm and batch input<br>2. report writng |
| 2 | Cooperation:<br>Able to work within team; willingly performed task(s) | A | A | A | A |
| 3 | Punctuality:<br>On time for team meetings | A | A | A | A |
| 4 | Reliability/Dependability:<br>Performed tasks within established times | A | A | A | A |

| | | | | | |
|---|---|---|---|---|---|
| 5 | Evaluative:<br>Offer constructive criticism and helpful evaluation of work | A | A | A | A |
| 6 | Creativity:<br>Provide meaning insight to project team. | A | A | A | A |
| 7 | Overall Effort:<br>Overall contribution to project | A | A | A | A |