CS - F372 OPERATING SYSTEMS

SECOND SEMESTER 2024-25

ASSIGNMENT 1: CHAT MANAGEMENT AND MODERATION SYSTEM
MAX. MARKS = 30

DEADLINE: 2:00 AM, 17/02/2025

Note: This document contains the problem statement, assignment constraints, hints, submission guidelines, late submission policy, plagiarism policy, and demo guidelines. Please go through the entire document thoroughly.

Problem Statement

The nation of Paradise has recently fallen under the rule of dictator Zachary, whose singular goal is to keep his citizens walled in under tight surveillance and strict control. Determined to silence any dissent and maintain his absolute authority, Zachary has ordered your team to build a real-time chat management and moderation system.

This tool must monitor every conversation among the people and flag anyone Zachary considers a potential threat to his regime. Given his peculiar approach to dealing with dissidents, straying from the provided specifications is not an option—any deviation will be met with dire consequences.

System Overview

The system simulates a chat management and moderation framework where different entities, such as moderator, and groups, are represented as processes. Within this system, each group process corresponds to a chat group (similar to a WhatsApp group) consisting of multiple users (also processes) who can interact with each other.

To simulate this interaction, each group process will spawn one or more child processes, each representing an individual user within the group. Users will send messages to their respective group processes, which will monitor the content for restricted words—terms Zachary deems unacceptable. Every time a message is sent to a group, the moderator will check the message and check if the message violates the terms. The moderator process will track the number of violations committed by each user. If a user exceeds a predefined threshold, the moderator will instruct the respective group process to remove/ban that user from the group.

The group process must handle the user messages and forward them to the validation process via a message queue in the correct order based on their unique timestamps. The validation process (validation.out) will verify that the messages are correctly ordered in ascending order of their timestamps and ensure that messages from banned users (users who have been removed because of violations) are properly blocked. Failure to meet these requirements will result in a failed test case, with corresponding mark deductions.

If a user is removed/banned, the group must continue sending messages to the validation process in the correct order, ensuring that only messages from users who are currently in the group are sent. This implies that even if a banned user ends up sending a message to the group process, that message is ignored and is not forwarded by the group process.

If the number of users in a group becomes less than 2, the group must be terminated.

Please note that the executable file for the validation process has been provided (validation.out). You may have to execute the command chmod 777 validation.out before running the file.

You are only required to develop solutions for app.c (the overall framework), groups.c (representing each group), and moderator.c (moderator process), as well as the IPC connections between them. You are not permitted to create any additional files. Note that multiple source files should not be created for different instances of group processes. The executable of the same groups.c should be used to deploy the different group processes. Also, app.c and groups.c should be two separate source files. The logic for groups.c should not be included inside app.c. Conversely, the logic for the user processes should be included in the groups.c file and not in a separate source file.

You should deploy the different processes in the following manner:

1. **First execute validation.out on one terminal. Use the command** `./validation.out X`**, where X is the test case number.**
2. **Then, execute moderator.out (executable for moderator.c) on a separate terminal. Use command** `./moderator.out X`**, where X is the test case number.**
3. **Finally, execute app.out (executable for app.out) on a separate terminal. You should not execute groups.out separately. Use command** `./app.out X`**, where X is the test case number.**

For each execution of the application, a single test case will be used.

Constraints for each test case:

Max. number of groups per test case = 30

Max. number of users per group = 50

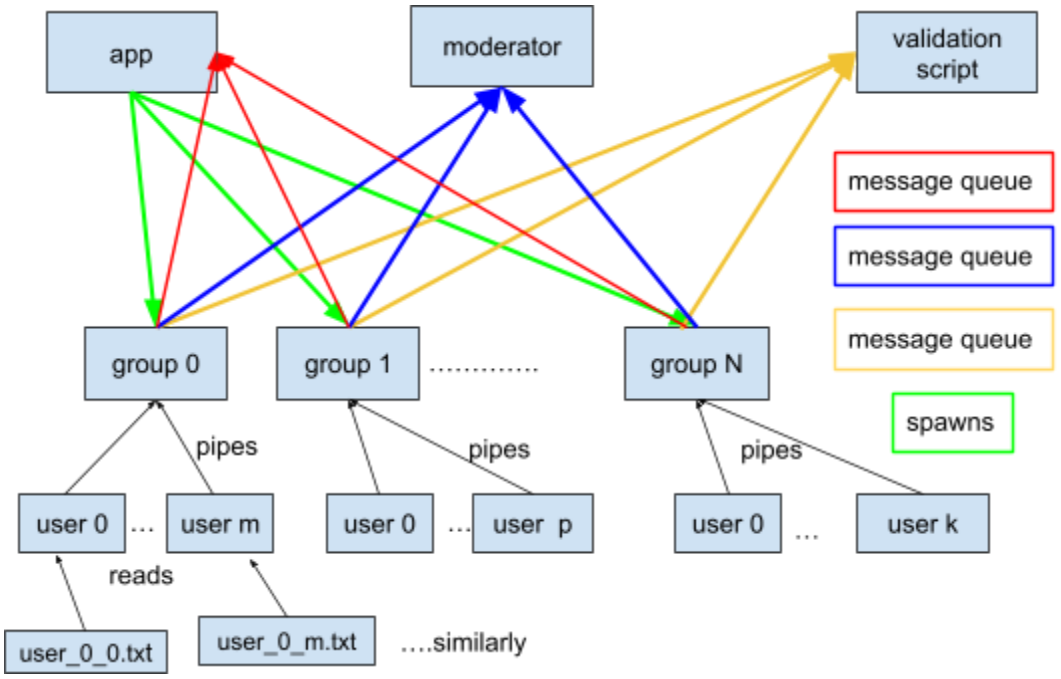
Max. number of filtered words = 50

Max. length of a filtered word = 20

Max. length of a line in users_X_Y.txt file (timestamp + space + text message including the '\n') = 256 characters

Max. message timestamp = 2147000000

The overall block diagram of the application is represented diagrammatically as follows:

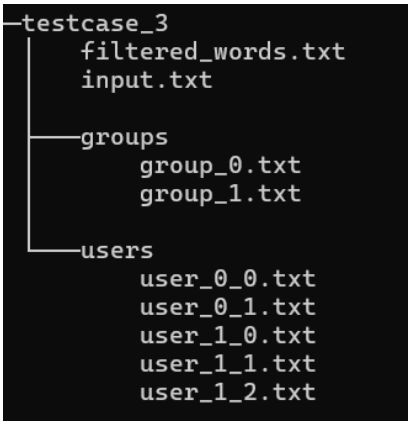


Note:

- **Red arrows** indicate a **message queue** between the **group processes** and the **app process**. All group processes communicate with the app process via a single message queue.
- **Green arrows** represent that the **app process spawns the group processes**.
- **Blue arrows** depict communication between **group processes** and the **moderator process** via a single **message queue**. All group processes communicate with the moderator process via a single message queue.
- **Yellow arrows** indicate that **group processes send messages** to the **validation script** using a single **message queue**. All group processes communicate with the validation process via a single message queue.
- **Black arrows** represent **communication between user processes and their respective group processes** through **unnamed/ordinary pipes**. Each user process communicates with the corresponding group process via a separate pipe. You should not use one single pipe for the entire group.
- Overall, the entire application uses three message queues.

Input Files and Directory Structure: Specific details of each file are listed below. These files will be provided to you. Of course, the ones attached with the assignment will not be used in the demos.

Testcase X will be given in the form of a folder containing multiple files. The directory structure of each test case is as follows:



Note: The number of group and user files will vary based on the test case. **Ensure that your C programs are present in the same parent directory as that of testcase_X** (where X varies depending on the test case number) **to maintain consistent relative file paths during the demo.** The groups and users directories may contain more files than you are required to process as per the test case. You don't have to worry about these additional files.

A description of each file in the input is given below.

`input.txt`: The **`input.txt`** file contains the following information:

1. An integer representing the initial number of groups (n).
2. A single number representing the key to connect to the message queue associated with the validation.out file.
3. Two keys for the message queues, one between app.c and groups.c, and the other between moderator.c and groups.c.
4. An integer representing the violations threshold.
5. n relative file paths, each pointing to the respective group process input files.

A sample input.txt file:

```
3 // no. of groups
3430 // key for message queue between groups.c and validation.out
4928 // key for message queue between groups.c and app.c
9131 // key for message queue between groups.c and moderator.c
5 // violations threshold for each user
groups/group_0.txt
groups/group_3.txt
groups/group_7.txt //note that these .txt files may not follow a contiguous numbering (as in 0, 1, 2, 3,.... may not all be present)
```

`filtered_words.txt:` This file contains a list of all the filtered (restricted) words, with one word on each line. A sample **`filtered_words.txt`** file is:

```
aBorm
ErSTowQULV
VIxsMCicp
aFTu
GnmRp
xdoqLcm
```

`group_X.txt` (where X is the group number)

This file contains:

1. A number indicating the maximum number of users in that group.
2. File paths to the user files associated with each group member. The number after the first underscore is the group number, and the number after the second underscore indicates the user number/ID.

```
3
users/user_1_0.txt // Group 1, User 0
users/user_1_5.txt // Group 1, User 5
users/user_1_7.txt // Group 1, User 7
```

`user_X_Y.txt:` X and Y represent the group number and user number, respectively. Each line in this file contains two components: a number (int) representing the timestamp, followed by the message (as a single string), with a whitespace separating the timestamp and the message. The messages are sorted according to their timestamps (ascending order). Each message is a single word string and contains only alphabets (either uppercase or lowercase).

```
1 IaGnmRpSamxdoqLcm
4 LetsoaBormrowtheGnmRprnment
5 RevoluVIxsMCicpthewayForward
6 dxdoqLcmiththedVIxsMCicptor
11 Meetmeafterthepizzaparty
```

What You Have to Do:

Note: The term ‘runner’ is used for the person executing the program, and ‘user’ is used to indicate a user process belonging to a group.

1. Write a POSIX-compliant C program named `app.c` that implements the following:

A. It takes input from input.txt, reading N, the number of initial groups to be created. **No new groups will be added dynamically.**

B. It creates a new process for each of the N (N being the no. of groups to be created) groups by spawning the group processes (described below) from within the app.c. Note that each group process is started by executing the .out file corresponding to groups.c and there should be only 1 instance of the file groups.c in the entire application. Also, app.c (corresponding to the overall app process) and groups.c (corresponding to the group processes) are two distinct separate source files. (**Hint:** Find out how Linux creates and runs new processes.)

C. Tracking of Active Groups

I. After app.c creates the initial N groups, it also tracks the number of active groups.
II. Once all the groups have terminated and there are no more active groups, app.c must also terminate.
III. To facilitate communication between app.c and groups.c you have to use a message queue with the key given in input.txt.

D. Output from app.out: The app process must display the following message whenever a group is terminated. This message should be printed for every group termination.

```
All users terminated. Exiting group process X.
```

2. Write a POSIX-compliant C program named `groups.c` which has the following functionalities:

A. This file is responsible for creating a group, adding users to the group, receiving messages from the users, and communicating with app.c, moderator.c, and validation.out.

B. Only one single source file for group processes should be used. The logic for group processes should not be included inside app.c.

Understanding how users relate to the group:

Users are child processes forked from the parent group process. Each group process, spawned by app.c, creates M user processes through forking. After forking these user processes, the original process continues functioning as the parent group process, responsible for managing all incoming messages from its child users.

Each user process reads its messages from dedicated text files and communicates with the parent group process using unnamed/ordinary pipes. Additional details are available under Subheading C.

C. To communicate with the validation process, all groups must use the same message queue. The key for this message queue is given in the input.txt file. The structure of the message to be used for the message queue:

```
typedef struct {
    long mtype;
    int timestamp;
    int user;
    char mtext[256];
    int modifyingGroup;
} Message;
```

Message types (`mtype`) recognised by validation.out:

- `1 - tells validation that a new group is created`
- `2 - tells validation that a new user has been created`
- `3 - tells validation that a particular group is to be terminated`
- `MAX_NUMBER_OF_GROUPS + GROUP_NUMBER - when a particular group sends a message. For example, MAX_NUMBER_OF_GROUPS = 30 and GROUP_NUMBER = 5, then when group number 5 sends messages of mtype = 35.`

D. Initialization

I. When a group is created, it must send a message to the validation process to indicate the creation of the group. This message must have an **mtype of 1** and the **field modifyingGroup should be set to the group number**. All the other fields of the message will be ignored. (**Note:** This message must be sent every time a group is created.)

II. Next, the group must read its corresponding groups/group_X.txt (X is the group number) file, and add the corresponding users to the group. The first line of the group_X.txt file contains M, the number of users to be added initially for the group. The following lines M contain the paths of the M user files (see group_X.txt example above).

III. To create a user, the group must fork a child process for each user of the group. Communication between a child and the group will be done using unnamed/ordinary pipes.

IV. The group must also send a message to the validation process to indicate the addition of a user to the group.

V. This message must have an **mtype of 2,** the **field modifyingGroup must be set to the group number, and the field user should be set to the user number.** All the other fields of the message will be ignored.

VI. Each child (user) process must then read the corresponding user file (users/user_X_Y.txt, where X is the group number and Y is the user number), and send these text messages to the group process using a pipe.

E. Sending chats to the validation process

I. The group must now send these text messages to the validation process, **ordered by timestamp (in ascending order). It must ensure that the text messages sent are from users who are current members of the group.** When sending a user message to the validation process, the **mtype of the Message struct must be set to MAX_NUMBER_OF_GROUPS + GROUP_NUMBER.** The timestamp field of the Message struct must be set to the timestamp of the text message, the user field should be set to the user number of the corresponding text message and the **text message should be put in the mtext field** (see Message struct above). Note that the mtext field should only contain the text part of the message. mtext should not contain the timestamp and characters like '\n' at the end.

F. Communicating with the moderator process

I. The group must also send each message it receives to the moderator process using a single message queue. The key for the message queue is given in input.txt.
II. The moderator process will send a message back to the group process to indicate if the user sending the message has exceeded the threshold number of violations and should be removed from the group (the working of the moderator process is described later).

G. Removing Users

I. If the moderator process indicates that a user should be removed from a group, **OR** if a user sends the group all of its messages, **THEN**, that user must be removed.

H. Terminating a group

I. **IMPORTANT:** If at any point, the number of users in a group becomes less than 2, that group must be terminated. **Note** that the number of users drops if a user is removed from the group, **OR** if the user has sent all its text messages (in either of these cases, the user process terminates).
II. When a group is terminated, the group process must communicate this with both the validation and other relevant processes. You may use the message queues provided in input.txt for this.
III. To communicate termination of a group with the validation process, it must send a message of **mtype = 3** to the validation process using the message queue. The **modifyingGroup field of the Message struct must be set to the group number of the group being deleted and the user field of the Message struct must be set to the total number of users who were removed from the group due to excess violations (does not include those users who sent all their messages without exceeding the threshold number of violations).** If the termination of the

group is valid, the validation process will not return any message. **If it is invalid, the validation process will terminate and the test case will fail.**

3. Write a POSIX-compliant C program `moderator.c` which has the following functionalities:

A. This file is responsible for receiving messages from the groups, searching the messages for filtered words (substring matching), and keeping track of the number of violations committed by each user of each group.

B. The number of violations for a given message is defined as the number of filtered words (from filtered_word.txt) present in the message.

C. Rules and examples are provided below:

I. A word from the `filtered_words.txt` file is considered a **violation** if it appears as a **case-insensitive substring** in the given message.

II. **Each unique word** from the filtered list counts as **one violation**, regardless of how many times it appears in the message.

III. **If a filtered word is a substring of another filtered word**, both are counted as separate violations.

Example 1:

Filtered Words: `pink, ink, apple`
Message: `Thisappleispinkerthantheotherapple`
Violations: `apple` → 1 violation, `pink` → 1 violation, `ink` (substring of `pink`) → 1 violation
Total Violations: 3

Example 2:

Filtered Words: `cheat, hack, ban`
Message: `Playerscaughtusingcheatswillbepermanentlybanned.`
Violations: `cheat` → 1 violation, `ban` → 1 violation
Total Violations: 2

Example 3:

Filtered Words: `Net, woRk, netWOrk`
Message: `ThecomputerNeTwoRkisnotworkingproperlyonthenet.`
Violations: `NeT` →1 violation, `woRk` → 1 violation, `NeTwoRk` → 1 violation
Total Violations: 3

Example 4:

Filtered Words: `Net, woRk, netWOrk`
Message: `ThecomputerNeTwoRkisdown.`
Violations: `NeT` →1 violation, `woRk` → 1 violation, `NeTwoRk` → 1 violation
Total Violations: 3

D. Once the moderator updates the number of violations committed by that particular user, it must tell the corresponding group process if the user needs to be removed or not. If the number of violations committed by the user is **greater than or equal** to the threshold number of violations (defined in input.txt), then the user must be deleted.

E. If the moderator determines that user X has to be deleted from group Y after M violations, it must print it on the terminal in the following format:

```
User X from group Y has been removed due to M violations.
```

Sample Output:

A sample output on successfully completing a test case from the validation script will be as follows:

```
================================================================
Number of groups created: 9
Number of users created: 96
Number of messages received: 1365
Number of users deleted: 25
Number of groups deleted: 9
```

If the test case fails, the validation script provides a description of why the test case failed. For example:

```
Testcase failed: Due to XYZ reason
```

Note: After termination, validation.out removes the message queues resulting in an error in receiving or sending messages to the message queue. Handle these errors appropriately.

For example:

```
msgrcv failed: Identifier removed
```

Implementation Guidelines and Constraints:

- **Using validation.out:**
  - If you are an Intel/AMD/Older Mac (x64) user, use this validation.out file.
  - If you are a Mac M1 (AArch64) user, use this validation.out file.
- **Error Handling:**
  - Perform proper error handling for all system calls.
  - Failure to handle errors appropriately will result in mark deductions.
- **File Specifications:**

    - All `.txt` files mentioned (e.g., `group_X.txt`, `user_X_Y.txt`) are ASCII (text-only) files.

- **Inter-Process Communication (IPC):**
    - Only the specified IPC mechanisms—**pipes** and **message queues**—are to be used.
    - The use of any other IPC mechanisms will result in mark deductions.

- **Input Restrictions:**
    - No additional inputs are allowed beyond those specified in the problem statement.
    - Any deviation from this will lead to mark deductions.

- **Synchronization and Multithreading:**
    - **Do not use** synchronization constructs such as **mutexes** or **semaphores**.
    - **Multithreading** is strictly prohibited in the implementation.

- **Pipe Management:**
    - Close all unused pipe ends wherever applicable to prevent resource leaks.

- **Adherence to Constraints:**
    - Any violation of the constraints mentioned above will lead to deductions in marks.

- **Use of unstructured programming constructs:**
    - You are not allowed to use unstructured programming constructs like goto in any part of your code. Marks will be deducted if you use any such constructs.

Hints:

1. Ensure that the fields in the various structures are consistently ordered across both the sending and receiving processes to ensure smooth and error-free code execution.

2. Note that a message queue writer can also read the messages it writes, leading to undefined behavior. Ensure that mtype(s) and structs are designed properly to prevent this.

3. Ensure that the `msgsz` parameter in the functions `msgsnd()` and `msgrcv()` are written in this way : `sizeof(struct) - sizeof(struct.mtype)` .

   For example:

```
msgsnd(APP_MSGQ_ID, &appMessage, sizeof(appMessage)-sizeof(appMessage.mtype), 0)
```

4. When writing to pipes, it is crucial to **pad the text** in the buffer to **exactly `MAX_TEXT_SIZE(256)`** bytes before writing. Failing to do so may result in multiple messages being **concatenated together** when reading, as the pipe does not inherently preserve message boundaries.

5. Since this assignment involves forking a process multiple times, please pay close attention to how you **close the read and write ends of each pipe.** Failure to do so will result in a **block on read()** that causes your program to hang indefinitely.

Submission Guidelines:

● All programs should be POSIX-compliant C programs. No other programming language should be used.

● All codes should run on Ubuntu 22.04 or 24.04 systems. You can be asked to demo the application on either system variant.

● Submissions are to be done through the Google Form:
https://docs.google.com/forms/d/1DU3TAmZYn3eOagfm-uQzXzD6QeoIF74WOIE3remhSEo/preview

● There should be only one submission per group.

● Each group should submit a zipped file containing all the relevant C programs and a text file containing the correct names and IDs of all the group members. The names and IDs should be written in uppercase letters only. The zipped file should be named GroupX_A1 (X stands for the group number). You will be notified of your group number after a few days.

● If there are multiple submissions from a single group, any one of the submissions will be considered randomly.

● Your group composition for Assignment 1 has been frozen now. You cannot add or drop any group member now.

● No extensions will be granted beyond the given deadline.

● Do not attempt any last-minute submissions. You need to be mindful of situations such as laptops not working at the last moment, Google Forms becoming unresponsive, etc.

Late Submission Policy:

● The Google form will accept submissions beyond the given deadline.

● SUBMITTING AFTER THE DEADLINE WILL INCUR A LATE PENALTY.

● For every 20 minutes of delay, there will be a late penalty of 0.25 marks. However, no distinction will be made within that 20-minute window. For eg., The deadline is 2:00 AM on 17/02/2025. If a group submits between 2:01 AM and 2:20 AM (inclusive), then that group will lose 0.25 marks. If the submission is made at 2:10 AM, then also the penalty will be 0.25 marks. However, if the submission is made at 2:21 AM, then 0.5 marks will be deducted.

● The submission will be summarily closed after 32 hours from the deadline. No submission will be allowed after 32 hours from the deadline.

● It is your discretion what you want to do – take a penalty and correct some last minute error or submit a partially correct implementation (which will lead to marks deduction).

● For calculating the late penalty, the date and timestamp of the submission via the Google form will be considered. No arguments will be entertained in this regard.

Plagiarism Policy:

● All submissions will be checked for plagiarism.

● Lifting code/code snippets from the Internet is plagiarism. Taking and submitting the code of another group(s) is also plagiarism. However, plagiarism does not imply discussions and exchange of thoughts and ideas (not code)

● All cases of plagiarism will result in awarding a hefty penalty. Groups found guilty of plagiarism may be awarded zero.

● All groups found involved in plagiarism, directly or indirectly will be penalized.

● The entire group will be penalized irrespective of the number of group members involved in code exchange and consequently plagiarism. So, each member should ensure proper group coordination.

● The course team will not be responsible for any kind of intellectual property theft. So, if anyone is lifting your code from your laptop, that is completely your responsibility. Please remember that it is not the duty of the course team to investigate cases of plagiarism and figure out who is guilty and who is innocent.

● **PLEASE BE CAREFUL ABOUT SHARING CODE AMONG YOUR GROUP MEMBERS VIA ANY ONLINE CODE REPOSITORIES. BE CAREFUL ABOUT THE PERMISSION LEVELS (LIKE PUBLIC OR PRIVATE). INTELLECTUAL PROPERTY THEFT MAY ALSO HAPPEN VIA PUBLICLY SHARED REPOSITORIES.**

● **SUBMITTING CODES CREATED BY PROMPTING LANGUAGE MODELS IS STRICTLY PROHIBITED. IF SUBMITTED CODES ARE FOUND TO BE GENERATED BY A LANGUAGE MODEL, THE ENTIRE GROUP WILL DEFINITELY BE PENALIZED AND MAY BE AWARDED ZERO.**

Demo Guidelines:

● The assignment also consists of a demo component to evaluate each student's effort and level of understanding of the implementation and the associated concepts.

● The demos will be conducted in either the I-block labs or D-block labs. Therefore, the codes submitted through the Google Form will be tested on the lab machines.

● No group will be allowed to give the demo on their laptop.

● The codes should run on Ubuntu 22.04 or Ubuntu 24.04.

● All group members should be present during the demo.

● Any absent group member will be awarded zero.

● The demos will be conducted in person. The demos will not be rescheduled.

● Though this is a group assignment, each group member should have full knowledge of the complete implementation. During the demo, questions may be asked from any aspect of the assignment.

● **Demos will be conducted on 22nd February and 23rd February 2025. Note that demos can be conducted on any one or both days. So, you need to be available on campus for both days. You need to book your demo slots as per the availability of your entire group. If any evaluation component of some other course gets scheduled on these dates, you will need to book your demo slot to avoid any clashes**.

● Demo slots will be made available in due time. You need to book your demo slots as per the availability of your entire group.

● The code submitted through Google Forms will be used for the demo. No other version of the codes will be considered.

● Each group member will be evaluated based on their overall understanding and effort. A group assignment does not imply that each and every member of a group will be awarded the same marks.

● Any form of bargaining for marks with the evaluators will not be tolerated during the demo.