Operating Systems 19/20

Practical Assignment - UNIX C programming

The practical assignment of operating systems is to implement a message management and redistribution system called MSGDIST. Messages are simple text messages, described in more detail below. The system takes care of accepting messages, storing them and redistributing them to anyone interested in them. Details are given below.

The practical assignment must be programmed in C language for Unix (Linux), using the system mechanisms discussed in the theoretical and practical classes. Priority should be given to the use of operating system calls (documented in section 2 of the *manpages*) over the use of library functions (documented in section 3 of the *manpages*). The work focuses on the correct use of system mechanisms and resources and no preference is given to the choices in implementing aspects peripheral to the course (e.g. the question "should I use a linked list or a dynamic array?" will be answered with a shrug). The use of libraries that do, or hide, part of the work is not allowed. The use of system mechanisms that have not been addressed in class is allowed but will have to be properly explained. An approach based on simply pasting excerpts from other programs or examples is not allowed. All code presented must be understood and explained by the submitter.

This document starts with a functional description, followed by operation rules of the editing system.

1. General description

The MSGDIST system comprises the following elements:

- People: Administrator and Users
- Processes and Programs: Manager, Verifier and Clients

The following concepts are involved:

- Message and topic.
    - A message is a text organized in fields: topic, title, body, duration. The message body can have up to 1000 characters. Set the maximum message size (in total, with all of its fields) as you see fit.
    - A topic is a broad theme (subject) into which messages can be framed. A message sent by a client may mention a new topic, i.e. nonexistent in the manager. In this case, the manager will have one more topic in his list of topics available for subscription.
    - Duration is the maximum time that the message will be stored in the system (in the manager process).
    - All information referenced herein is textual except the length of the message which is an integer that represents the duration of the message, in number of seconds.

The goal of the MSGDIST system is to mediate the reception and delivery of messages between clients (for the benefit of their users). Clients send messages to the manager, which will then be delivered to clients according to their choices and queries. The functionality is quite straightforward, and no more elaborate mechanisms are required than described.

Client usage

All users use the same client program to interact with the manager. Each user will execute one instance (i.e., process) of the client program. Through the client program, the user can:

- Write a new message specifying all its fields and send it to the manager. The interface for the input of the message should look like a text editing control (N rows x M columns, with N and M chosen to allow the input of a 1000 characters), in which the user can move the cursor and type wherever he wants similar to a notepad. This can be accomplished with the *ncurses* library.
    - The message, once sent and accepted by the manager, is available to other users. If a user is using a client program and has subscribed the topic of the sent message, he should be alerted immediately by his client as a result of the notification received from the manager.
- List the existing topics (whether or not they have messages).
- List the message titles under a specific topic.
- Display the contents of a message from a specific topic.
- Subscribe / unsubscribe one topic.

The client is notified whenever a new message, from a topic he subscribes, becomes available in the manager. In this situation the message “new message (message title) of topic (topic name) is available during (duration)” is displayed. This warning appears on the screen at the precise instant that manager sends the notification, regardless of what the client is doing at the moment, and in a way that does not interfere with what that clients' user is currently doing (for example writing another message). Therefore, care should be taken when designing the user interface (suggestion: use the *ncurses* library).

The user specifies his username to the client program as a command line argument. The first step of the client program is to identify the user to the manager. This corresponds to sending the specified username. If another user with the same username is currently using the MSGDIST system, the manager automatically adds a number to the username (e.g. "manuel" becomes "manuel3" because "manuel", "manuel1" and "manuel2" already exist). The client is always informed about the username that was given by the manager. When there is a difference between the username specified by the user at the command line and the one that the server assigned to him, the client must inform his user.

Automatic interaction between client and server:

- The client will refuse to continue its execution if it realizes that the manager is not running (Note: there are several ways to accomplish this).
- When the user terminates his client process, the client should inform the manager of this fact, so that his username is released, and the manager stops sending notifications of new messages on topics subscribed by that user.
- When the manager finishes, it should inform the connected and currently running clients of this situation. Clients should also terminate and inform the user of this fact.
- The manager should automatically detect that a client has ceased to exist within a time frame of 10 seconds (for example, if the client abruptly terminates due to improper memory usage, the manager will know this no later than 10 seconds and treat this as a client shutdown).
- The client should detect that the manager is no longer running within a time frame of 10 seconds. This and the previous requirement are easy to accomplish using *heartbeat* / *keep alive* mechanisms (search for these keywords to investigate the underlying concept and develop it using the mechanisms covered in the course - there are several ways to implement this mechanism). This requirement is related to the previous one: the mechanisms used by the manager and the client are designed to work together.

The exchange of information between the client and the server should be done using *named pipes*. Other mechanisms are involved in client-server interaction, but for information exchanged, *named pipes* should be used. There is no direct interaction between two clients.

Manager functionality

The manager should implement all the functionalities expected from the clients, as described above:

- Receives messages sent by clients, stores them for the time indicated in the message duration field, and automatically deletes them as the duration runs out. The maximum number of messages to store is fixed during manager execution and is specified in MAXMSG environment variable, which is looked for during manager start up.

- Notifies clients / users about new messages received on topics that they are subscribing.
- Maintains information about which clients and users it is currently interacting with. Although related, in the ratio of 1 client - 1 user, client and user are distinct entities. The client is a process and the user is the person who instantiated the client program and is identified in the system by a specific username at a given time.
- Keeps track of which topics are subscribed by which clients / users.
- It is launched by the administrator and accepts written commands entered according to the "command line" paradigm (e.g. menus should not be used).
- Detects which clients have stopped running (because their users have closed them, or because they don't show anylife signs for more than 10 seconds).
- Notifies known clients that it will be terminated, and lets the clients know that it is still running (depending on how this functionality is implemented on the client side).
- Immediately reports to *stderr* all actions that change the list of messages or topics (some examples: message received, deletion of messages as soon as the duration time elapses, creation of new topics and deletion of existing topics, etc.), and actions that change the list of clients / users (some examples: user joined, client no longer exists, etc.). The report consists of text messages indicating what happened. These messages are sent to *stderr*.

Each message received by the manager is analyzed with the aid of the independent program "verifier", which receives the body of the message and identifies the number of forbidden words in it. If the number of forbidden words exceeds the value indicated in the MAXNOT environment variable, then the message is rejected, and the referring client is notified. The forbidden words are in a text file whose name is specified in the WORDSNOT environment variable. Pay attention to how the "verifier" gets the name of that file.

The "verifier" program code is provided in the appendix. You should compile this program so that the executable is available to the manager. You should analyze the source code provided to understand how the "verifier" gets the words to check and how it indicates the resulting count in order to understand how the manager should communicate with him. You should also try running it at the command prompt. You cannot change anything in the source code of the " verifier" program. The manager should strive to optimize the use of the verifier (for example, keep it running instead of starting a new instance for each message received).

Data persistence

There is no persistence of information from one manager execution to another, nor from one client execution to another by the same user:

- There is no permanent file record about subscribed topics by which client or user. As soon as the client closes, the manager forgets its existence.
- There is no permanent user account (not even passwords).
- Messages in the manager exist only in memory. When the manager shuts down, all messages, topics, and user information is lost.
- The user needs to re-subscribe to the topics of interest in between client executions.

If you wish, you can make this information persistent (in file). But it is not part of the requirements.

Automatic manager behavior

The manager should perform the message length checking, new messages notification to the clients, verification on the existence of the username and whatever is necessary for clients to realize that the manager is still running. The list presented here is not exhaustive and you should deduce the rest of the automatic behavior from the remaining text.

Manager administration

The administrator is a user who interacts with the manager through written commands. The logic is similar to a *shell* command line. However, this is an analogy: you should not use *bash* – you simply should not use a menu-driven interface. The administrator's intent is expressed through a multi-word line of text where the first one is the command and the following are the options or values of that command. The line is read at once: one line – one action that the administrator wants to perform.
The actions that the administrator can take are:

- Turn on / off forbidden word filtering: **filter on / filter off**
- List users: **users**
- List topics: **topics**
- List messages: **msg**
- List posts under a topic: **topic topic-in-question**
- Delete message (you have to propose and adopt a way to identify them): **del message-in-question**
- Delete a user (who will be notified and whose client will close): **kick username-in-question**
- Exit manager: **shutdown**
- Delete topics that currently have no messages (existing subscriptions to these topics are unsubscribed and their users are notified): **prune**

You can implement other options as you please (tips: listing messages for a particular user, listing messages expiring in the next x seconds, etc.). These are not extras, but may compensate other flaws and/or improve the debugging process.

The following mechanisms must be able to work independently / parallel to each other so that the delay in one does not directly jeopardize the execution of the others.

- Verification of words through verifier
- Processing of messages sent by clients
- Answering administrator commands
- Detection of closed or unreachable clients
- Verification of message age and deletion of expired messages.

A few notes

The manager must be able to handle multiple tasks simultaneously. The client too. The concurrent task list in the manager is not the same as in the client.

Visual interface and message editing

The visual interface is most important in the client when editing the message text and displaying the manager's notifications. You may use the *ncurses* library. The *ncurses* library includes functions for character printing, keyboard reading, cursor positioning, and color definition. They are used in a similar way to any other library: "*there are these functions, they have these parameters, they are invoked this way*". Not much more to say about that. Eventually you could be given some information about *ncurses* in class. Among others, you can consult the website

http://tldp.org/HOWTO/NCURSES-Programming-HOWTO/

2. Additional Considerations

- Client and manager programs are launched via the command line. Each program should be launched from a different session / *shell* (locally or using *PuTTY* for remote users) but using the same user account of the underlying operating system.
- There should be only one manager process running. If a new instance of the manager is executed, it should detect the existence of the former and terminate immediately.
- If there are multiple tasks performed simultaneously involving the same data, care should be taken to synchronize access to the data using appropriate mechanisms to ensure the consistency of the manipulated information at all times.
- There may be several error situations and potential conflicts that are not explicitly mentioned in this document. These situations must be identified, and programs must deal with them in a controlled and stable manner. A program ending abruptly in one of these situations is not considered an appropriate way to handle the scenario.

  If inconsistencies are found in this document, they will be corrected and published. There may be additional documents in *moodle*, such as a FAQ with frequently asked questions and answers.

3. General Rules, GOALS and IMPORTANT DATES

The following rules, as described in the first class and in the course unit form (FUC), apply:

- The work can be done in groups of two students (groups of three are not admitted and any request in this regard will always be denied or ignored).
- There is a mandatory defense. The defense will be made in a manner to be defined and announced through the usual channels when relevant.
- There are two deadlines for the submission, as described in the FUC. Deadlines and their requirements are listed below. In each deadline the student is required to submit a **single zip**[1] , through *moodle*, whose **filename** follows the following template[2]:

  so_1920_pa_goal1_name1_number1_name2_number2.zip

  (of course, goal, name and number will be adapted to the goal, names and numbers of the group elements)

  Failure to adhere to the indicated file compression format or name pattern will be penalized and may result in the work not being accepted for evaluation.

Goals: Requirements and Delivery Dates (deadlines)

Goal 1:

Requirements:

- Plan and define the data structures responsible for managing the operation of the manager and the client. Define various *header* files with symbolic constants that record common and client- and server-specific default values as well as relevant data structures.
- Develop the logic for accessing the environment variables on the manager and the client, and update the data structures mentioned in the previous point. Tip: use the `getopt()`, `getsubopt()` and `getenv()` functions.
- Start implementing the reading and validation of the manager's administration commands and their parameters, and fully implementing the `shutdown` command.
- Prepare the manager-verifier connection to allow filter functionality to be tested with some words sent from the manager.
- Develop and deliver a *makefile* that has the compilation targets "all" (compilation of all programs), "client" (compilation of client program), "manager" (compilation of server program), "verifier" (compilation of verifier program) and clean (deletion of all temporary build support files and executables).

[1] Read "**zip**" - not *arj*, *rar*, *tar*, or others. Use of another format will be **penalized**. There are many UNIX command line utilities to handle these files (zip, gzip, etc.). Use one.
[2] Failure to comply with the filename format will be **penalized**.

Submission deadline: Sunday, November 17th. No possibility of late delivery.

Goal 2:

Requirements:

- All requirements stated in this document.

Submission deadline: Sunday, January 5, 2020. Subject to adjustments if changes occur in the academic calendar.

For all goals a report should be delivered. The report will include content that is relevant to justify the work done, it must be the sole responsibility of the group members, and include mandatory content that will be specified in *moodle* in a timely manner.

4. Evaluation

The following elements will be taken into account in the evaluation of the work:

- **System architecture** - There are aspects related to the interaction of the various processes that must be carefully planned in order to present an elegant, light and simple solution. The architecture should be well explained in the report so that there are no misunderstandings.
- **Implementation** - Must be rational and should not waste system resources. The solutions found for each work problem should be clear and well documented in the reports. The programming style should follow good practices. The code must have relevant comments. System resources must be used according to their nature.
- **Reporting** - Reports must conform to the information that will be provided in *moodle*. In general, the reports describe the strategy and models followed, the structure of the implementation and the decisions made. They will be sent along with the code in the file submitted via *moodle* for each goal.
- **Defense** - Works are subject to individual defense where the authenticity of their authorship will be tested. There may be more than one defense if doubts remain as to the authorship of the work. The final grade of the work is directly proportional to the quality of the defense. Members of the same group may have different grades depending on their individual defense performance.

  Although the defense is individual, both members of the group must attend at the same time. Failure to defend the work automatically implies the loss of the entire grade of the work.

  This year will be used software that automates the detection of fraud in the submitted works. The FUC describes what happens in situations of fraud (e.g. plagiarism).
- Works that do not function will be heavily penalized regardless of the quality of the source code or architecture presented. Works that do not even compile will have an extremely low grade.

- Identification of group elements should be clear and unambiguous (both in the archive and in the report). Anonymous submissions won't be evaluated.
- Any deviation in format and shape in submissions (e.g. file type) will result in penalties.