Coimbra Institute of Engineering
Department of Computer and Systems Engineering

Operating Systems 24/25

Practical Work - Programming in C for UNIX

This is the English version of the published assignment statement in Portuguese. It is aimed at Erasmus students (who can use the Portuguese version or this one).

General rules

The practical work covers knowledge of the Unix system, and must be done in C language, for the Unix platform (Linux), using the mechanisms of this operating system covered in the theoretical and practical classes. The work focuses on the correct use of the system's mechanisms and resources and no particular preference is given to choices in the implementation of aspects peripheral to the subject (questions such as "should I use a linked list or a dynamic *array*?" are not important questions. Similarly, secondary aspects that are more associated with the theme than with the subject matter are not relevant.

When it comes to manipulating system resources, priority should be given to using calls to the operating system[1] rather than library functions[2] (for example, *read()* and *write()* should be used instead of *fread()* and *fwrite()*). The exception here is the reading of files of a secondary nature, such as configuration files or data backups, which can be read with the usual *fread* and *fscanf* (when using *named pipes,* the system functions should be used).

It is not allowed to use third-party libraries that are part of the work, or that hide the use of the resources and mechanisms studied in the course. The use of Unix system APIs that have not been covered in class is permitted as long as they are within the same category of resources studied - that is, minor variations of the functions studied, and must be duly explained, particularly during the defense.

An approach based on merely pasting excerpts from other programs or examples is not allowed. All code presented must be understood and explained by the person presenting it, otherwise it will not be counted.

1. General description, main concepts and elements involved

The work consists of a platform for sending and receiving short messages, organized by topic. A user can send messages to a particular topic and subscribe to one or more topics. Whenever a message is sent to a particular topic, users who have subscribed to it receive that message. There will be a "***manager***" program that manages the reception and distribution of messages, and there will be another "***feed***" program (send and receive) that is used by users to interact with the messaging platform. Any action described as "*the user does*..." should be understood as "*the user does... using the **feed** program*".

The user interface is in a console environment (text). There will be several users, each using a different terminal, but always on the same machine. *Platform user* and operating *system user* are different concepts.

[1] Documented in section 2 of the *manpages*

[2] Documented in section 3 of the *manpages*

Users first identify themselves (i.e. "start by identifying themselves, using the ***feed***") to the platform using a *username*. Users don't need to register beforehand, they just need to enter a *username* that hasn't already been used by any of the other users who are already logged in. No password is required.

Any user can post to a specific topic. If such a topic does not exist, and the server's maximum topic limit has not yet been reached, then the topic will automatically be created and users can subsequently subwrite it. Topics are identified by their name (e.g. "soccer", "IT", "movies", etc.), with the topic name assumed to be just one word.

Whenever a user sends a message (using their ***feed***) to a topic, this message will be delivered and displayed immediately to all users who are online and have subscribed to that topic.

There are two types of message: ***non-persistent*** and ***persistent***. ***Non-persistent*** messages are not stored: as soon as the manager receives them, it distributes them to the users interested in their topic and then discards them. ***Persistent*** messages are also distributed to users interested in their topic, but they are also stored in the ***manager*** for a certain period (the "lifetime" of the message). During this time, the messages are delivered to all users who connect to the platform in the meantime and subscribe to the topic of those messages (or, if they are already online, have not yet subscribed to the topic and have done so in the meantime). Each persistent message has its own lifespan, which does not have to be the same for all of them. The user sending the message always has to specify the duration of the message (in seconds) when sending it. If the specified lifetime is 0, the message will be treated as non-persistent. After the specified lifetime, the ***manager*** discards the message.

When started, the ***manager*** will retrieve from a text file the persistent messages that were still within their lifetime (implicitly creating the respective topics). For these messages, the time count continues (i.e. their lifespan does not go back to the "beginning"). Further details are provided below.

Programs involved - additional details

The platform consists of two programs: the ***feed*** program and the ***manager*** program**.**

- The ***feed*** program is used by the user to send and receive messages, and these actions can take place simultaneously (this means that while the user is writing a message, the content of a message that has appeared in a topic they have subscribed to can appear on the terminal). The functionality of the ***feed*** program is essentially the user interface, and all the platform management is done in the ***manager*** program. Users will only be able to use the platform's functionality once they have identified themselves and their identification has been accepted. The identification consists of a *username* (just one word), with no password, and is always accepted as long as there isn't already another user with that name. Each user will launch an instance of the ***feed*** program, and there may be more than one simultaneously at any given time.
- The ***manager*** program manages the platform's topics, messages and users, and interacts with the various ***feeds***. It can interact directly with a user (platform administrator), who specifies commands (received by the ***manager*** on its *stdin*) that allow it, for example, to shut down the platform, show active users, expel a user, etc. At any one time there will be at most one ***manager*** process running.

Further details about these programs are given below.

Users - There are two types of user in this system:

- **Client**. This is the person who, after identifying themselves, participates in the platform. You will not be able to interact directly with other users. The functionality of sending messages and subscribing to topics is done through written commands. All the information provided is text only. Adding a new user simply opens a new terminal through which they interact with the ***feed***.
- **Administrator**. Controls the ***manager*** and is responsible for executinb it. It can interact with the platform through the ***manager*** program, following the logic of commands written on its *stdin* (these commands will be described in detail later). It's important to note that the "administrator" is not related to the operating system administrator (*root*).

All users of the platform correspond to the same user of the operating system where the various ***feed*** programs are run. Simulating the existence of another user corresponds to running another ***feed*** on another terminal.

Message content, management and delivery

Predefined limits: In the implementation you can assume that there are maximums for the following entities

- Users: maximum 10
- Topics: maximum 20
- Persistent messages (per topic): maximum 5

Message information: Messages consist of the following information

- Topic name: up to 20 characters, a single word (i.e. no spaces).
- Message body: text up to 300 characters, with the possibility of spaces (several words).
- Additional information needed to meet the requirements.
- This list refers to information and not necessarily to a specific message or message structure, which is up to those who implement the programs.

Storing messages in memory

- If messages need to be stored in memory, the maximum values indicated above can be assumed.
- The storage will include the information necessary to meet the functional requirements.
- In the ***feed*** program, you don't need to store your messages: just display them as soon as you receive them.

Backup and recovery of persistent messages in a file.

- Persistent messages that still have time to live are saved in a file when the ***manager*** is closed and then retrieved when the ***manager is*** restarted.
- The file in question is a text file with the following mandatory format:
    - One message per line.
    - Each line has *<thread name> <author name> <lifetime left> <message body>*
        - As usual in these cases, the characters < and > delimit the name of the fields in this description and should not actually exist in the file, nor should they be entered by the user.
        - Topic, username and remaining lifetime will each be "a single word". However, the body of the message may have spaces, and should be understood as everything between the lifetime and the end of the line.

    Example

```
soccer sr_silva 110 that league game was really weird
mario 99 snacks does anyone know where to buy cheap bifanas?
```

- The name of the file is in the **MSG_FICH environment variable**, and its use is mandatory.

Sending messages

- The body of the message can be up to 300 characters long, although it can be significantly shorter. A solution that sends only the part (the characters) actually used by the message, avoiding sending unused characters, will have a higher rating than solutions that simply send messages of the maximum length.

Time management by the program manager.

Persistent messages have a maximum lifetime, specified by the sender, as described above. This time is specified in number of seconds. The program ***manager*** should remove messages as soon as they expire. This time management is not directly related to system time.

Data files involved

The only file needed is the one that stores the persistent messages that have not yet expired when the ***manager*** closes. The topics of these messages don't need to be stored separately: they are deducted and created as the persistent messages are retrieved (in a similar way to when a message is received from a topic that didn't exist yet and comes into existence at that moment).

We suggest using C library functions (fread, fgets, etc.) rather than system functions (read, write) for the issue of saving and retrieving persistent messages as they simplify these tasks. This suggestion does not extend to other features of the platform that may involve the use of write/read operations, where the preference for system functions will prevail.

2. Use of the platform (functionality seen by users)

From the point of view of the *client* user (feed program)

- The client user executes the **feed** program by entering their *username* via the command line. Example:

```
$ ./feed pedro
```

- The ***feed*** only accepts execution if the ***manager*** is running and must not remain running if the ***manager*** terminates.
- The ***feed*** initially sends the ***manager*** the user's name. If validation is successful (there is no other user with the same name and the maximum user limit has not been reached), the ***feed*** program waits for commands from the user. Otherwise, the ***manager*** informs the user of what happened via the ***feed***.
- The user interacts with the ***feed*** using text commands that allow them, among other things, to send text messages and subscribe to topics, and receive the messages associated with them.
- Message communication between sender clients and subscribers is always mediated by the ***manager***. The message is sent by the sender to the ***manager*** and the ***manager*** distributes the message to all the connected ***feeds*** that have subscribed to the thread where it was published.

The ***feed*** program must deal simultaneously with the information coming from the ***manager*** and the commands entered by the user. Without prejudice to the usability of the user interface, it is not mandatory to use the ***ncurses*** library, and it is sufficient to use an interface based on the console paradigm (text *scroll-up*) as long as it is clear and logical.

The ***client*** user can do this:

- Get a list of all topics: **topics**

  It shows the name of the existing topics, the number of persistent messages in each one and their status (locked/unlocked - a subject explained later).

- Send a message to a specific topic: **msg** <topic> <duration> <message>

  The duration is always indicated. If it is not zero, the message is automatically understood to be persistent. Messages will be delivered immediately to currently connected users who are interested in the topic, or, if the message is persistent and has not yet expired, as soon as they subscribe to the topic, including those who connect in the meantime.

- Subscribe to a topic: **subcribe** <topic>

  If the topic already exists, the user immediately receives all persistent messages from that topic. If it doesn't exist, and the ***manager***'s maximum topic limit has not yet been reached, it will be created. In either case, the user will receive messages sent to that topic.

- **Unsubscribe** from a particular topic: **unsubcribe** <topic>

  The user will no longer receive any messages sent to that topic. If there are no persistent messages in that topic and no user is subscribing to the topic, then it disappears from the ***manager*** altogether.

- **Exit**, terminating the ***feed*** process: **exit** command

  The ***manager*** should be aware that the user has left and react accordingly.

All commands must have *feedback* on the terminal indicating the success/unsuccess of the operation and the relevant data, according to the command written.

All messages received (relating to subscribed topics) must indicate the topic in which it was submitted, the user who sent it and its content.

Other functionalities associated with visualization

- The user receives a notification/message each time a topic they are subscribed to is locked/unlocked by the administrator (explained below). When a topic is locked, the customer will not be able to send new messages to that topic until it is unlocked (**msg** commands from the ***feed*** program will be ignored by the ***manager***). It is possible to subscribe to a locked topic and immediately receive all its existing messages.
- The user receives a notification if they are deleted from the platform by the administrator. In this situation, the ***feed*** program should terminate in an orderly fashion, returning to the command line.

From the point of view of the user ***administrator*** (program manager)

The administrator runs an instance of the ***manager*** program, and only one instance (process) of this program can be running at any given time. This is validated by the program and not by the user. Example of execution:

```
$ ./manager
```

The only user who interacts with the ***manager*** is the platform **administrator**, by writing commands to perform control actions. There is no *login* procedure: the **administrator** simply runs the ***manager*** program and interacts with it. The actions (commands) available to the **administrator** are:

- List **users** currently using the platform: **users**
- Delete a user: **remove** *<username>*
  It has the same effect as the **exit** command in the **feed** program. The user in question is informed, and their ***feed*** program should end automatically. In addition, all other ***feeds*** should be informed that the user has been deleted from the platform.
- List the topics on the platform: **topics**

  The name of the topics and the number of persistent messages are displayed.
- List the messages in a given topic: **show <topic>**

  All persistent messages from a given topic are displayed.
- Lock a topic: **lock** <topic>

  Blocks new messages from being sent to the indicated topic. Persistent messages continue to exist until their duration expires, and it is still possible for a user to subscribe to the topic.
- **Unlock** a topic: **unlock** <topic>
- **Close** the platform: **close**

  Close the platform and terminate the ***manager***. All processes running the ***feed*** are notified and must also terminate. The system resources in use are released.

3. Requirements and restrictions

Implementation

- Regular files are repositories of information - they are not communication mechanisms.
- The communication mechanism between ***feed*** and ***manager*** is the *named pipe*. The number of *named pipes* involved, who creates them and how they are used should follow the principles explained in class. Using more named pipes than necessary can be penalized.
- Only the communication mechanisms that have been covered in class can be used.
- The system's API must be the one studied. Any variation must be within the API studied. A function that hasn't been covered much, but is related to the ones studied, is accepted, as long as it stays within the context of what was covered (example: *dup2* instead of *dup* is accepted - as long as it is explained in the defense, but use of shared memory is not).
- The use of third-party libraries (except those provided by teachers) is not accepted.
- Any synchronization issues that may exist must be taken into account and dealt with appropriately.
- Situations that require programs to deal with actions that can occur simultaneously cannot be solved with solutions that delay or prevent this simultaneity. These situations, if they occur, have appropriate ways of being solved that have been studied in class and must be observed.
- The use of code excerpts from examples (books, stackoverflow, github, etc.) must not be extensive or address the central issues of the subject, and must be explained in the defense.
- All the code must be explained in the defense, whether or not it has been taken from examples - if it is not explained, it will be understood as "the knowledge is not there" and therefore the unexplained part is not valued.

Program termination

- Whether it's at the user's request, because the conditions for the program to run are not met, or because of a *runtime* error, programs must terminate in an orderly fashion, freeing up the resources used, giving as much notice as possible to the user and the programs they interact with.

4. General work rules

The following rules apply, as described in the first lesson and in the course outline (FUC):

- The work can and should be done in groups of two students. Groups of 3 are not allowed in any circumstance (failure to adhere will result in severe grade penalties).
- There is compulsory defense, which is individual and in person. There may be additional arrangements to be defined and announced through the usual channels (for example, whether or not it is necessary to bring a computer).
- The practical work is delivered via nónio (inforestudante) by submitting a single ***zip* file**[3] whose **name** respects the following pattern :[4]

  ***so_2425_tp_nome1_numero1_nome2_numero2*.zip**

  (*name* and *number* are the names and student numbers of the group members)
- Failure to adhere to the indicated compression format (.zip) or the file name standard will be penalized and *may result in the work not even being graded*.

[3] Read "***zip***" - not *arj*, *rar*, *tar*, or others. Using another format may be **penalized**. There are many UNIX command line utilities for dealing with these files (zip, gzip, etc.). Use one.

[4] Failure to comply with the name format causes delays in the management of the works received and will be **penalized**.

- The zip file should contain:
    - All **source code** developed;
    - ***Makefile***(s) with the compilation *targets* "all" (compilation of all programs), "feed" (compilation of the ***feed*** program), "broker" (compilation of the ***manager*** program) and "clean" (deletion of all temporary compilation support files and executables); and
    - A **report** (in pdf format) with the content that is relevant to justify the work carried out and which must be authored exclusively by the members of the group.
- Each group submits their work once, and it doesn't matter which of the two students does it.
- **It is compulsory** for the student making the submission **to link the submission to the other student in the group**.
- **Both students must be enrolled in practical classes (even if they are in different classes). Failure to register will prevent the work from being recorded on the platform, and could therefore lead to the loss of the grade.**

Delivery date: Sunday, December 15, 2024.

5. Work evaluation

The following elements will be taken into account when assessing the work:

- **System architecture** - There are aspects relating to the interaction of the various processes that must be planned in such a way as to present an elegant, light and simple solution. The architecture should be well explained in the report.
- **Implementation** - Must be rational and not waste system resources. The solutions found must be clear and well documented in the report. The programming style should follow good practices. The code should have relevant comments. System resources and APIs should be used according to their nature.
- **Report** - The report must describe the work properly. Merely superficial or generic descriptions are of little use. In general, the report describes and justifies the strategy and models followed, the structure of the implementation and the choices made. The report must be submitted together with the code in the file submitted.
- **Defense** - The work is subject to individual defense, during which authorship and knowledge will be verified, and there may be more than one defense if doubts remain. The final grade is directly proportional to the quality of the defense. Members of the same group may receive different marks depending on their individual performance and degree of participation in the defense.

  Failure to attend the defense will automatically result in the loss of the entire grade.
- Plagiarism and work done by third parties: the school regulations describe what happens in situations of fraud.
- Entries that don't work will be heavily penalized regardless of the quality of the source code or architecture submitted. Works that don't even compile will receive an extremely low score.
- The identification of group members must be clear and unambiguous (both in the zip file and in the report). Anonymous work will not be corrected.
- Any deviation from the format and form of submissions (e.g. file type) will result in penalties.

Important: The work must be done by both members of the group. Hermetic divisions in which one member does only one part and knows nothing about the rest are not acceptable. If there is unequal participation in a group, the teacher doing the defense must be informed of this at the beginning of the defense. If he or she is not informed and this inequality is detected, both students will be penalized rather than just the one who worked less hard.