isec Engenharia
Coimbra Higher Institute of Engineering Department of Computer and Systems Engineering

Operating Systems 25/26

Practical Work - Programming in C for UNIX

General rules

The practical work covers knowledge of the Unix system and must be carried out using the language, API, and mechanisms covered in class. 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 that are peripheral to the discipline.

With regard to the manipulation of system resources, priority should be given to the use of operating system calls[1] over the use of library functions[2] (for example, *read()* and *write()* should be used instead of *fread()* and *fwrite()*). An exception to this is the reading of secondary files, such as configuration or data backup files, which can be manipulated with library functions (but in the use of *named pipes*, system functions should be used).

The use of third-party libraries that are part of the work or that conceal the use of the resources and mechanisms studied in the course is not permitted. 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 properly explained, particularly during the defense.

An approach based on merely pasting excerpts from other programs or examples is not permitted. 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 simulating a management platform for a taxi service comprising a fleet of autonomous vehicles. The platform consists of three applications: **customer** – interaction between the user and the transport service scheduling platform; **controller** – responsible for managing the car fleet, receiving transport requests, allocating vehicles, and accounting for the total kilometers traveled; and **vehicle** – simulates the operation of an autonomous vehicle, interacting with the **customer** whenever necessary. A user of this service can schedule a transport service through the **client** application**.** The platform (**controller** application**)** is responsible for allocating one of the autonomous vehicles in the taxi fleet at the time requested by the customer. T h e autonomous vehicle (**vehicle** application**)** travels to the location indicated by the customer and, before starting the service, asks the customer directly where they want to go.

The user interface is console-based (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.

Users begin by identifying themselves (i.e., "they begin by identifying themselves using the **client** application") to the platform with a *username*. Users do not need to register in advance; they simply need to enter a *username* that has not already been used by any of the other users who are currently connected. No *password* is required.

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

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

Users can schedule transportation services, check scheduled services, and cancel a specific service that has been scheduled but not yet performed (**client** application). The user can request one or more transportation services, indicating the *time*, *starting location*, and *distance of the route*. The specific location to which they wish to travel is indicated by the user to **the** autonomous **vehicle** when it arrives. You can assume that there is a maximum (limited) number of services and users. It is not necessary to perform *business rule* checks such as preventing the same user from requesting two different trips at the same time (this is not the focus of the work).

At the specified time, the **controller** sends an autonomous vehicle to perform the requested transport service. To do this, it must launch a **vehicle** process, passing it the information about the service to be performed via command line arguments. Please note that the **vehicle** will need to contact the **customer and**, for this reason, it needs to know the customer's "contact" details (i.e., the controller also sends this information, also via command line arguments). From the moment the **vehicle** is launched**,** the **controller** receives information about **the vehicle's** status **(**corresponding to the very popular concept of telemetry). The information is sent by **the vehicle** via its *stdout,* and the **controller** must capture this information. The controller will do this for all vehicles in operation, and there may be several at the same time. The relevant information is: the customer has entered the vehicle, the percentage of the distance already traveled (the vehicle reports every 10%), and the customer has left the vehicle (indicating whether they have completed the trip or decided to leave early).

The **vehicle** is an application that simulates the operation of an autonomous vehicle. For each scheduled service, a **vehicle** application will be executed**, launched by the controller program (as mentioned in the previous paragraph)**, which ends when the customer leaves the vehicle (either voluntarily mid-journey or upon arrival at their destination). Information about the service and the customer is specified through command line arguments. From the moment it is executed until it ends, the **vehicle** process must send the vehicle status information (telemetry) to the monitor (*stdout)*, as indicated above. The format of the information is open and must be decided by the implementer: it can be a single *string* in any format that encapsulates all the information to be indicated at the time in question, or it can be a transmission for each piece of information, indicating which piece of information and its respective value. The requirement is that the information must be passed from **the vehicle** to the **controller** via *stdout,* and must respect the periodicity (e.g., 10% of the distance). This information is collected by **the controller** to know the status of the entire vehicle fleet—which/how many **vehicles** are performing services and what their status is.

2. Programs involved

The platform is fully implemented through the three programs mentioned above: **customer**, **vehicle**, and **controller.**

Customer

The **client** program is used by the user to schedule, consult, and cancel services, in addition to actions such as entering and exiting (mid-journey) a vehicle. Sending commands (actions to be performed) and receiving information (arrival of the transport vehicle, etc.) can occur simultaneously (meaning that while the user is writing a command, the content of a message sent by **the controller** and/or **vehicle** may appear on the terminal**)**. The **client** program's functionality is essentially the user interface, with all platform management being done in the **controller** program**.** The user will only be able to use the platform's functionality after identifying themselves and having that identification accepted. Identification consists of *a username (*a single word), without any *password,* and is always accepted as long as there is no other user with that name. Each user will launch an instance of the **client** program**,** and there may be more than one at a given moment.

Any exchange of information between the **client** and the **vehicle**, or between the **client** and the **controller**, will be done via ***named pipes***.

Vehicle

The **vehicle** program simulates an autonomous vehicle. It is executed by **the controller** at the time specified for the requested service and receives, through command line arguments, the data for the service to be performed (including the contact details of the customer for whom the service is intended). It reports all service events to the monitor (*stdout*). Communication with the **customer** is done through *named pipes*. You cannot use *named pipes* to communicate with **the** controller—you must use command line arguments (to receive information from **the controller)** and send status information to the monitor (*stdout)* (to send information to **the controller)**. There may be several **vehicle** processes running, this value being limited by the size of the vehicle fleet (defined by an environment variable that is considered by **the controller)**.

Controller

The **controller** program manages the fleet of autonomous vehicles. It can interact directly with a user (**platform** administrator**,** not system administrator), who specifies commands in the command logic written in the console (and received by **the controller** in its *stdin)* that allow, for example, to shut down the platform, show active users, the list of scheduled services, the list and status of vehicles performing services, etc. The **controller** communicates: i) with the **client(**s) via *named pipes, ii)* with the **vehicle(**s) via command line arguments and redirection of the output (*stdout)* of **the vehicle** process**(**es). At any given time, there will be at most a single **controller** process running.

It works with a server and receives requests from various clients (ultimately, users) via *named pipes*. Requests can be for **scheduling**, **canceling**, and consulting transportation services (only yours). It is responsible for "sending" the vehicle at the specified time to the departure location (i.e., launching the vehicle process that corresponds to the service, at the time requested by the user, specifying the details of the service via the command line), and for receiving the status of a service from each of the vehicles. This application must keep a counter updated with the total number of kilometers traveled by all vehicles.

The size of the vehicle fleet is defined in the NVEICULOS environment variable. This means that there will be no more than NVEICULOS **vehicles** running at the same time. "Sending" a vehicle means launching a **vehicle** process.

The **controller** is responsible for keeping the fleet status information up to date. To this end, **vehicles** report the status of the service they are performing, providing the following information:

- the start of the service (when the customer entered the vehicle, which occurs *when* and *through* interaction with that user's **customer**);
- the distance already traveled (percentage of the total distance, indicated every 10%); and
- the end of the service (the customer left the vehicle halfway or arrived at their destination).

To cancel a service that is already being performed, the **controller** must send the SIGUSR1 signal to the **vehicle** process.

Predefined limits

In the implementation, you can assume that there are maximum values for the following entities

- Users: maximum 30
- Vehicles: maximum 10 (the actual value is specified using the NVEÍCULOS environment variable)

2. Use of the platform

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

- **User**. This is the person who uses the **client** and, after identifying themselves, uses the platform to request transportation services (schedule, consult, and cancel) and perform actions on a vehicle (enter and exit). There is no way to contact a **vehicle** directly; this can only be done after initial contact from **the vehicle** (at the start of the service). No

There is no client-client interaction. To handle a new user, a new terminal will simply be opened through which it interacts with the **client**.

- **Administrator**. Controls the **controller** and is responsible for launching its execution. It can interact with the platform through the **controller** program, following the logic of commands written in its *stdin* (these commands will be described in detail later). It is important to note that the "administrator" has no relation to the operating system administrator (*root*).

All platform users correspond to the same operating system user where the various **client** programs
are executed. Simulating the existence of another user corresponds to executing another **client** on another terminal.

Controller

From the point of view of the administrator user (**controller** program):

- The only user who interacts with the **controller** is the platform administrator, through written commands to perform control actions. There is no *login* procedure: the administrator simply runs the **controller** program and interacts with it. The administrator runs an instance of the **controller** program, and only one instance (process) of this program can be running at any given time. The validation of this aspect is the responsibility of the program and not the user. Example of execution:

  ```
  $ ./controller
  ```

- The actions (commands) available to the administrator are:

The **controller** has an interface, following the command line paradigm, which allows the administrator to perform the following commands:

- **list** – Shows information on all scheduled services.
- **use** – Shows the list of users currently connected (indicating waiting for car / traveling).
- **fleet** – Shows the percentage of the trip (% of the route already completed) for each vehicle.
- **cancel** <id> – Cancels a service using its *ID*, whether it is scheduled or currently being performed. If the *ID* is 0 (zero), it cancels all scheduled services, including those already being performed (in this case, you must cancel these services while they are being performed).
- **km** – Shows the total number of kilometers traveled by all vehicles.
- **time** – Shows the current simulated time (not the system time).
- **terminar** – Ends the execution of the entire system (you must cancel all services and notify all **customers**).

Customer

It acts as an interface between users and the platform**.** It sends requests via *named pipes* using the command line paradigm. From the point of view of a platform user (**client** program):

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

  ```
  $ ./client pedro
  ```

  The **client** only accepts execution if the **controller** is running and should not remain running if the **controller** terminates.

- The **client** initially sends the user name to **the controller**. If the validation is successful (there is no other user with the same name and the maximum number of users has not been reached), the **client** program waits for commands from the user. Otherwise, the **controller** informs the user of what has happened through the **client**.

- The user interacts with the **client** through text commands that allow, among other things, to manage their transport services, interact with the autonomous vehicle, and receive information from the platform.
- Communication between the **client** and the **vehicle** is done directly between the two, without involving the **controller** (as an intermediary). However, there may be situations where the **client** needs to talk to both **the vehicle** and **the controller**.

The **client** program must simultaneously handle the information coming from the **controller** and **vehicle** applications (via *named pipes*) and the commands entered by the user. Depending on the command entered, it will be sent to **the controller** or the **vehicle.** Without prejudice to the usability of the user interface, it is not mandatory to use the **ncurses** library**;** it is sufficient to use an interface based on the console paradigm (*scroll-up* text) as long as it is clear and logical.

The user can enter (on the client) the following commands (which will be directed to **the controller** or the **vehicle**, as appropriate and necessary to fulfill the user's order):

- **schedule** <time> <location> <distance>

  Schedule a service for the specified *time*, from *the* departure *location* to a destination at a specified *distance*. The time is specified in seconds (counted from the start of **the controller**), the departure location is for information purposes only (it can take any value – *string*), and the distance is indicated in kilometers.

- **cancel** <id>

  Cancels a previously scheduled service that has not yet been performed. If the *id* is 0 (zero), it cancels all services scheduled by the user (the **controller** manages the schedule).

- **consult**

  Displays information about the services you have scheduled (this information is stored in **the controller**).

- **enter** <destination>

  Enter the **vehicle** after its "arrival" (**the vehicle** contacts **the customer** to inform **them** that it has arrived at the departure location). It can only be used at the start of a service and must indicate the destination (the distance to be traveled is known in advance by **the vehicle**). The destination can be specified as a *string (*e.g., Coimbra, Porto, Lisbon, etc.), the content of which has no other use than informational. This command is directed to **the vehicle.**

- **exit**

  Allows you to indicate to **the vehicle** that you want to exit before reaching your destination. In this case, the **vehicle** terminates the service and the controller is notified. The distance actually traveled can be assumed to be the last update of the distance traveled by the vehicle—multiples of 10%, such as the moment the customer exits the vehicle. This command is directed to **the vehicle**.

- **end**

  Allows you to exit the **client** application, provided that you do not have any services running (you can only exit when the service is completed and your payment has been issued by **the controller**). If you have scheduled services, these must be canceled by **the controller** (the user must inform them of their intention).

All commands must have *feedback* on the terminal indicating the success/failure of the operation and the relevant data, as per the written command.

Messages received by the client must have visible effects on the user interface, indicating where they come from (**controller** or **vehicle**).

Vehicle

Used to simulate an autonomous vehicle. To this end, it sends all the status information of the service being performed to the monitor (stdout): customer entry, distance already traveled, service canceled midway at the customer's request (if applicable), and service completed.

It receives the *location*, *distance* (number of kilometers to travel), and how to reach the client (for example, the *named pipe* used by the **client** process) via the command line.

This application is launched by **the controller** at the time specified for the service. As soon as it is executed (assuming it arrives instantly at the service's starting point), it must contact the **client** (direct contact via *named pipes*) to start the service, reporting this fact. Remember that "time" in the context of the statement is a simple counter that starts at 1 when the **controller** starts and increments by 1.

During the execution of the service, the **vehicle** must report the status of the service every 10% of the total distance traveled. For this purpose, all vehicles have the same speed, which is 1 km per unit of time (the simulated time). If you receive the SIGUSR1 signal, you must cancel the service you are performing, informing the **client** of this fact, and terminating the process immediately.

At the end, you must report the completion of the service (**customer** departure) and terminate its execution (the process ends, with the
controller has one more vehicle available for the next services).

This application does not receive any commands directly from the user (unlike the other applications – **client** and **controller**), interacting only with the **client** and **controller** applications (using the communication mechanisms already indicated). However, the **vehicle** can be indirectly affected by the user: for example, if the user decides to leave in the middle of the trip (this intention is communicated first to **the client,** who then handles the situation).

Time management by the controller program

It is not necessary to use the operating system time to specify the "time" at which a **customer** wishes to schedule a service. You can assume that "time" is specified as an integer value, which corresponds to the number of seconds that have elapsed since the **controller** application was launched.

Data files involved

There are no configuration or data files.

3. Requirements and restrictions

Implementation

- **You cannot** use the *select* mechanism in the **controller** program.
- Regular files are repositories of information—**they are not** communication mechanisms.
- The communication mechanism between the **client** and other applications (**controller** and **vehicle**) is the *named pipe*. The number of *named pipes* involved, who creates them, and how they are used must follow the principles exemplified in the lessons. Using more *named pipes* than necessary may result in penalties.
- The communication mechanisms between the **controller** and the **vehicle**(s) are command line arguments (**controller** to **vehicle**) and *anonymous pipes* – redirecting output from **the vehicle**(s) (**vehicle** to **controller**).
- Only the communication mechanisms covered in class may be used.
- The system API must be the one that was studied. Any variation must be within the studied API. A function that was not covered in detail, but is related to those studied, is acceptable, as long as it remains within the context of what was covered (example: *dup2* instead of *dup* is acceptable – as long as it is explained in the defense, but the 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 addressed and handled appropriately.
- Situations that require programs to deal with actions that may occur simultaneously must be resolved with solutions that do not delay or prevent this simultaneity. These situations, if they occur, have appropriate solutions that have been studied in class.
- The use of code excerpts from examples (books, stackoverflow, github, bots, etc.) cannot be extensive or address the central issues of the subject, and must be explained in the defense.
- All 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 will not be valued. The following sentence and its variations are not an explanation: "It's how I saw it in class and I do it the same way, but I don't know why or what it means." Neither is this: "I was having problems and some colleagues told me to do it this way."

End of programs

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

4. General work rules

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

- The work can and should be carried out in groups of two students.
- There is a mandatory defense, which is individual and in person. There may be additional requirements to be defined and announced through the usual channels at the time it is relevant (for example, whether or not it is necessary to bring a computer).
- The practical work is submitted via Nônio (inforestudante) by submitting a single ***zip*** **file** whose **name** follows the following standard[4]:

  (*name* and *number* are the names and student numbers of the group members)
- Failure to adhere to the specified compression format (.zip) or file name standard will be penalized and *may result in the work not even being viewed*.
- The zip file must contain:
  - All **source code** developed;
  - ***Makefile***(s) with compilation *targets* "all" (compilation of all programs), "controller" (compilation of ***the controller*** program), "client" (compilation of the ***client*** program), "vehicle" (compilation of the ***vehicle*** program), and "clean" (deletion of all temporary files and executables – *.o);
  - A **report** (in pdf) with relevant content, which must be written exclusively by the group members.
- Each group submits the work once, regardless of which of the two students does so.
- The student who submits the work must **also associate the other student in the group with the submission in Nônio**.

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

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

- **It is advisable that both students be enrolled in practical classes (even if in different classes). Failure to enroll will prevent the work from being registered on the platform and may therefore lead to loss of grade.**

Due date: Saturday, December 13, 2025.

5. Assessment of the assignment

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

- **System architecture** – There are aspects related to the interaction of the various processes that must be planned in order to present an elegant, lightweight, and simple solution. The architecture must be well explained in the report.
- **Implementation** – It must be rational and not waste system resources. The solutions found must be clear and well documented in the report. The programming style must follow best practices. The code must have relevant comments. The system's resources and API must be used according to their nature.
- **Report** – The report should describe the work well. Superficial or generic descriptions are not acceptable. The report should describe and justify the strategy and models followed, the implementation structure, and the choices made. It should include a table with the required functionalities and, for each one, an indication of whether it has been fulfilled, partially fulfilled, or not fulfilled (and why). The report is delivered in PDF format within the submitted file.
- **Defense** – The assignments are subject to individual defense, during which authorship and knowledge will be verified. There may be more than one defense if doubts remain. The final grade for the assignment is directly proportional to the quality of the defense. Members of the same group may receive different grades depending on their individual performance and level of participation in the defense.
  Failure to attend the defense automatically results in the loss of the entire grade for the work.
- Plagiarism and work done by third parties: the school regulations describe what happens in cases of fraud.
- Assignments that do not work will be heavily penalized regardless of the quality of the source code or architecture presented. Assignments that do not even compile will receive an extremely low grade.
- 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 carried out by both members of the group. Strict divisions whereby one member does one part and only that part, knowing nothing about the rest, are not accepted. If there is unequal participation within a group, the teacher supervising the defense must be informed of this fact at the beginning of the defense. If they are not informed and this inequality is detected, both students will be penalized instead of just the one who did less work.