# PROJECT Nº2 – Table for Playing Cards

The main goal is to create an infrastructure for the playing of card games in text terminals.

## BRIEF

A *Table for Playing Cards* program is to be built, so that a number of people is able to play simple games of cards through text-terminals.

From the user point of view, each player opens a terminal window, invokes the *Table for Playing Cards* with appropriate parameters and waits for other players; when the necessary number of players is reached, the first of them can begin the game, shuffling the deck and dealing (giving cards); the cards of each player (his hand) are presented in his terminal, as well as other information, such as the cards that are up on the table; each player is able to see if it is her turn and, that being the case, is able to play by putting cards on the table, etc. All player's actions are triggered by choosing a option from a menu.

From a programming point of view, the *Table for Playing Cards*, is initiated on each terminal of each player with command line parameters that specify the main options of the game. Based on them, each running process enables the means for the interprocess communication: a local FIFO pipe and, if not yet set up, a shared memory area (populated with global variables and means for synchronized accesses). Then, a couple of threads are created, each of them controlling a facet of the game: the reading of the player's keyboard and the participation on the games' course, by interacting with the other player's processes.

The delivered *Table for Playing Cards* application should be demonstrated by allowing the play of some very simple card games. For each game, a file log of all the events and hands of each player should be produced, so that the whole sequence of the game could be verified, later on.

## ACADEMIC GOAL

To familiarize the students with the system programming in Unix / Linux, involving:

- processes and threads
- mechanisms for communication and synchronization between processes / threads.

## REQUIREMENTS

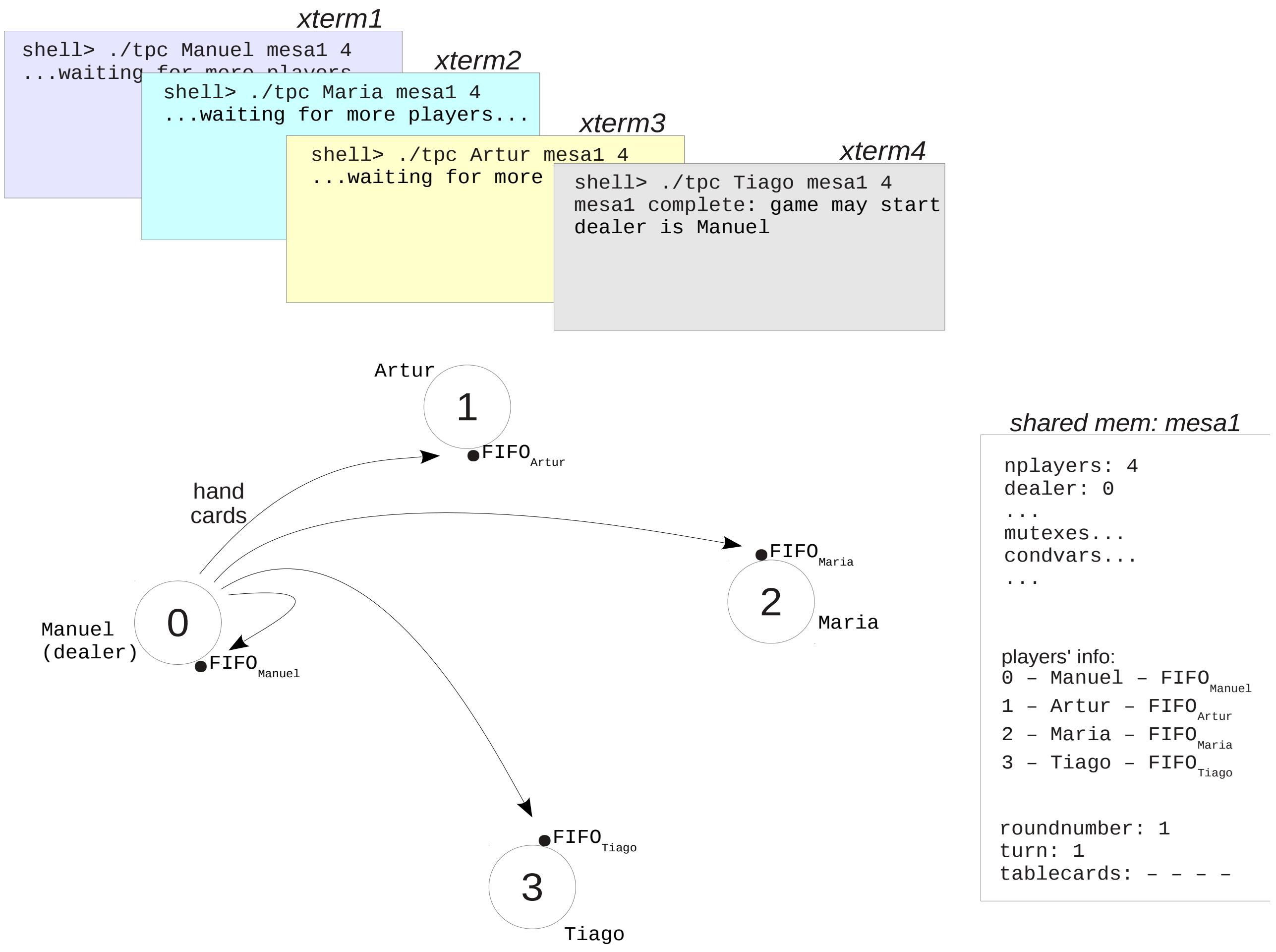
The *Table for Playing Cards*' application should be built in a way that allows the demonstration of the Unix system programming skills mentioned. So, its architecture and the operation and synchronization of the interacting processes should follow a number of rules. (As a side note, the main architectural aspect of the application follows a peer to peer model.)
On the other hand, its usage should try to emulate to a reasonable degree, the playing of a real card game.

The application is constituted by a single program (executable), an instance of which is to be started in several text terminals, one for each player, running on a single machine. The set of running processes, by allowing the playing of some simple card games, should prove their correct synchronization and good overall infrastructure's operation.

## User's requirements

The application's program, should:

- Be invoked through a *shell* with (see Fig. 1):
  ```
  tpc <player's name> <shm name> <n. players>
  ```
  (e.g. `./tpc Manuel mesa1 4` )
- Let the player see/know more or less what she is supposed to see/know in a real cards' game; at least:
  - her hand of cards (see Fig. 2)
  - the cards up on the table
  - who is to play next (turn)
- Let the player do more or less what he is supposed to do in a real cards' game, at least:
  - put a card on the table
- In order to simplify the development of the application, the possible games are restricted to the ones where all the cards are distributed at the beginning, each player must play one and just one card on his turn, and players are not allowed to abandon the game during its course. Also, the program should not impose any type of game and, consequently, assume nothing about game rules.



*xterm1*
```
shell> ./tpc Manuel mesa1 4
...waiting for more players
```

*xterm2*
```
shell> ./tpc Maria mesa1 4
...waiting for more players...
```

*xterm3*
```
shell> ./tpc Artur mesa1 4
...waiting for more
```

*xterm4*
```
shell> ./tpc Tiago mesa1 4
mesa1 complete: game may start
dealer is Manuel
```

*shared mem: mesa1*
```
nplayers: 4
dealer: 0
...
mutexes...
condvars...
...

players' info:
0 – Manuel – FIFO_Manuel
1 – Artur – FIFO_Artur
2 – Maria – FIFO_Maria
3 – Tiago – FIFO_Tiago

roundnumber: 1
turn: 1
tablecards: – – – –
```

*Fig.1. Pictorial representation of the beginning of a game using Table for Playing Cards, showing aspects of user interface, conceptual view and internal computer memory structure.*

Table of ranks:

| rank | A<br>Ace | 2<br>Deuce | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | J<br>Jack | Q<br>Queen | K<br>King |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Table of suits:

| suit | c<br>clubs | d<br>diamonds | h<br>hearts | s<br>spades |
|---|---|---|---|---|

Examples:

| card representation: | 5c<br>(five of clubs) | Ah<br>(ace of hearts) | Kd<br>(king of diamonds) |
|---|---|---|---|
| hand presentation: | `Ac-5c-8c-Jc / Ah-2h / 6d-10d-Kd / 3s-5s-4s-7s` | | |

*Fig. 2. Card representation in the application and example of hand: each card should be represented by two characters, the first, in uppercase if letter, for the rank and the second, in lowercase, for the suit; the cards in each suit are separated by a forward slash.*

## Developer's requirements

Once running, each of the application's interacting processes should:

- Set up the shared memory area, whose name was given as a command line parameter, avoiding the race conditions resulting from several processes trying to create the area at the “same” time. The first process that creates the area, will initialize all the global variables needed. This first process will also play the role of dealer (see Fig. 1).
- In the shared memory area, set some of the following global variables, as indicated:
    - array with players' info, holding for each entry (player) a structure with her number, nickname, FIFOname, etc.
      *Note: all users' processes will be able to read the array; each user's process should only write in its entry.*
    - number of players (`nplayers`)
    - turn to play (`turn`, corresponding to the player's number, 0..`nplayers-1`)
    - round number (`roundnumber`)
    - dealer's number identification (`dealer` is 0)
    - cards on the table, for each round (`tablecards`)
    - necessary inter-process synchronization variables (*mutexes*, condition variables) in order to avoid busy waiting.
- Create a local FIFO pipe that will be used for receiving the hands of cards from the dealer.
- If the process is the dealer:
    - shuffle the deck
    - distribute the initial hands for each player through their respective local FIFOs
    - specify the first player to play
- At the end of the game, remove from the system all traces of the permanent resources that where set up by each process, such as its FIFO and the shared memory (if it the owner).
- Create at least two threads, one for the asynchronous reading of the player's keyboard and another for the game actions synchronization with the others players' processes.
    - The keyboard's thread should wait and respond to the orders of the player, such as, show the cards on table, show the player's own hand, show previous round of cards, show the times of the game (e.g. the time elapsed since the beginning of a player's turn), etc.

- ◆ The game synchronization thread is responsible for the connection to the other players' processes, so that the game can be played. This thread should wait on a condition variable for changes on the state of the game, act accordingly and keep its player informed.
- ◆ If needed, to avoid busy waiting, these two threads should adequately synchronize each other when accessing global variables intra-process.

- Log all the hands played during the course of the game and the evolving hands of each player in the sequence of their happening, so that the playing of the game could be recreated later on. The logging is to be done concurrently to a *file* named `<shm name>.log` by appending lines with the format:

```
when                | who           | what          | result
2013-04-24 20:03:56 | Dealer-Manuel | deal          | -
2013-04-24 20:04:16 | Player1-Artur | receive_cards | Ac-5c-8c-Jc/Ah-2h/6d-10d-Kd/3s-5s-4s-7s
2013-04-24 20:04:56 | Player1-Artur | play          | 8c
2013-04-24 20:04:56 | Player1-Artur | hand          | Ac-5c-Jc/Ah-2h/6d-10d-Kd/3s-5s-4s-7s
...
2013-04-24 20:09:50 | Player0-Manuel| play          | Qc
...
```

## GRADING

The grading for this project will take into account several criteria, namely:

- Code readability in general, namely, identifier names, indentation, ... .
- Comments: function header comments and in-line comments.
- Adequate use of data structures.
- Code structure and modularity.
- Input validation.
- Operation of the applications according with the requirements.

## SUBMISSION

- Create a directory named **TxGyy** (where **x** and **yy** represent, respectively, the number of the class/*turma* and the team number (for example, T2G07, for team 7 of class 2).
- Create a folder, named **tpc**, and copy to it the source code of the program (**only the files with extensions .c** or **.h**) and an adequate **makefile**.
- Copy to it one or more examples of logfiles, illustrating some tests that have been done with the application.
- Aggregate the files into a file named **TxGyy.tar** and submit the file through the link available at the course page, at Moodle/FEUP.
- **Each working group must submit only one project.** The team may resubmit the project several times, but only the last submitted version will be graded and will be used to determine if the project is late.
- **Submission must be done before 2013/05/26, at 23:55.**