CS39002 Operating Systems Laboratory
Spring 2026

Lab Assignment: 4
Date posted: 06-Feb-2026

Inter-process Communication Using Shared Memory

This assignment has to do with a multi-process implementation of the game of Snake Ludo (Snakes and Ladders). The processes involved in the application are the following.

- A ***coordinator process*** *CP* to handle user inputs. This runs in the window where the application is launched.
- A process for each of the *n* ***players*** (call them *A*, *B*, *C*, …). These processes print information about their moves in a single window (separate from the window used by *CP*). These processes are launched by a parent process called the ***player-parent process*** *PP*.
- A ***board process*** *BP* that keeps on printing the current ludo board in a third window.
- Two other processes associated with the windows of the players (and their parent) and of the board. Let us call these processes ***xterm processes*** *XPP* and *XBP*. These processes are created by *CP*.

The process hierarchy is described in the following figure. Each arrow → indicates a `fork()`.

CP → *XBP* → *BP*
→ *XPP* → *PP* → *A*
→ *B*
→ *C*
→ …

The process tree looks as follows (use the shell command `ps af`, when all the processes are still running).

```
16319 pts/1    Ss     0:00 -csh
20189 pts/1    S+     0:00  \_ make run
20205 pts/1    S+     0:00      \_ ./ludo 4
20206 pts/1    S+     0:00          \_ xterm -T Board -fs 15 -geometry 150x24+50+100 -bg #003300 -e ./board 4 5
20207 pts/5    Ss+    0:00          |   \_ ./board 4 5
20208 pts/1    S+     0:00          \_ xterm -T Players -fs 15 -geometry 100x24+1000+100 -bg #000033 -e ./players 4 5 20207
20209 pts/6    Ss+    0:00              \_ ./players 4 5 20207
20210 pts/6    S+     0:00                  \_ ./players 4 5 20207
20211 pts/6    S+     0:00                  \_ ./players 4 5 20207
20212 pts/6    S+     0:00                  \_ ./players 4 5 20207
20213 pts/6    S+     0:00                  \_ ./players 4 5 20207
```

The rules for our version of Snake Ludo

We use the standard 10 × 10 board with the cells marked 1, 2, … , 100 in a zigzag fashion. A sample board (Source: Wikipedia) is shown to the right. The text file *ludo.txt* storing this board is given below.

```
L 3 21                  Ladder from cell 3 to cell 21
L 4 36
L 15 48
L 24 58
S 29 7                  Snake from cell 29 to 7
L 30 75
L 31 70
S 38 20
S 44 14
L 49 90
S 55 11
L 60 79
S 62 40
L 63 99
L 72 91
S 73 52
L 77 97
S 82 60
S 93 43
S 96 17
S 98 48
E                       End of board
```

The initial position of each player is an (invisible) cell 0 called home. Moves alternate among the players in the sequence A, B, C, … Let Π be the player to make the next move. Suppose that *pos* is the position of Π just before the move of Π. At the beginning *pos* = 0. If *pos* = 100, Π has already reached the destination, and is out of the game. For $0 \leqslant pos \leqslant 99$, Π throws a dice. With equal probability (that is, 1/6), one of the faces 1 – 6 turns up. If that is 6, Π throws the dice again. A face 1 – 6 again turns up (each with equal probability). If that second face is also 6, a third throw is made by Π. Again a face 1 – 6 appears (with equal probability). If it is 6 again, the three throws are canceled, and Π starts throwing the dice all over again. Let *m* be the sum of the outcomes of the throws (one, two, or three) of a valid (not canceled) throw sequence. The next position of Π is first calculated as *nextpos* = *pos* + *m*. If a ladder (lower end) or a snake (upper end) is at that cell, *nextpos* is updated accordingly. Following ladders and/or snakes continues until *nextpos* reaches a cell with no climb/descend possibility. If *nextpos* > 100, the movement of Π is not allowed. If the cell at *nextpos* is occupied by another player Π′, then also the movement of Π is not allowed (we enforce that each cell can be occupied by at most one player). If the movement of Π is not allowed, Π continues to stay at the old position *pos*. Otherwise its position is updated to *nextpos*.

Eventually, a player Π_1 reaches the destination (Cell 100). Π_1 is removed from the board (with rank 1), and will no longer take part in the round robin sequence of moves. Then, a second player Π_2 reaches the destination, and is removed from the board (with rank 2). This continues until all of the *n* players reach the destination. At this point, the game ends.

What is shared by the processes

Two shared-memory segments are used in this application. One segment M_B is used to store the board. The other segment M_P is used to store the positions of the players.

The segment M_B is prepared by the coordinator process *CP* from the input file *ludo.txt*. This is a segment capable of storing 101 integers. The 0-th cell is left unused. For $1 \leqslant c \leqslant 100$, $M_B[c]$ stores an integer. If a ladder starts at Cell c, then $M_B[c]$ stores the top end (cell number) of the ladder minus the bottom end (c) of the ladder, so $M_B[c]$ is a positive integer in this case. If there is a snake's mouth at Cell c, then $M_B[c]$ stores the cell number of the tail of the snake minus the cell number (c) of the mouth of the snake, so $M_B[c]$ is a negative integer in this case. If it is neither of the two cases, we have $M_B[c] = 0$. We assume that there is no ambiguity in the board (that is, two different ladders/snakes or one ladder and one snake do not start at the same cell). However, it is allowed to have the mouth of a snake and the top of a ladder at the same cell, and to have similar situations that do not lead to ambiguity.

After *CP* populates M_B, this memory segment is used as a read-only segment for the rest of the game.

The second segment M_P consists of $n + 1$ integer-valued cells. The players are called A, B, C, … , but they may be indexed as 0, 1, 2, … The cell $M_P[p]$ stores the current position of the p-th player, so $0 \leqslant M_P[p] \leqslant 100$. The cell $M_P[n]$ stores the number of players that are still in the game. This count can be computed from the first n cells (those that do not store 100), but is recommended to quickly check the end-of-game condition.

CP initializes M_P to (0, 0, … , 0, n). Subsequently, each player p updates its position $M_P[p]$. These writes are at mutually exclusive locations of M_P, but all the players need to read the positions of the other players. A player p reaching the destination decrements the count $M_P[n]$, that is, multiple leaving players may try to update this cell simultaneously. As explained later, synchronization is achieved by sequentializing the moves (using blocking waits).

The source files

Write the following three files in C/C++.

ludo.c(*pp*)	This implements the coordinator process *CP*.
board.c(*pp*)	This process keeps on printing the board after every move (the process *BP*).
players.c(*pp*)	This implements the player-parent process *PP*, and the players *A*, *B*, *C*, …

Beginning of the game

You open a shell, and run *ludo* (the executable from *ludo.c*). This launches the coordinator process *CP*. It creates the two shared-memory segments M_B and M_P in the exclusive mode. These segments are initialized by *CP* as explained above. *CP* also creates a pipe ℘ for receiving the PID's of *BP* and *PP*, and also acknowledgments from *BP*.

CP then forks two child processes *XBP* and *XPP*. Each of these processes opens an xterm for displaying relevant information. *XBP* runs *board* (the executable from *board.c*) leading to the conception of the process *BP*. On the other hand, *XPP* runs *players* (the executable from *players.c*), so the process *PP* is born. Notice that xterm forks a child process that in turn runs an executable (a shell by default; *board* or *players* in our case), so the processes *XPP* and *XBP* running the xterms are different from the processes *PP* and *BP* forked by these xterms.

At the beginning, *BP* sends its PID to *CP* via the pipe ℘ (created by *CP*), and sleeps for a second (so that *CP* can complete the reading of the PID's of both *BP* and *PP*). Notice that the forking by *CP* generates the process *XBP*, so *CP* knows its PID. *BP* is the child of *XBP*, so *CP* is unaware of the PID of *BP* (a grandchild of *CP*). A communication from *BP* to *CP* is therefore necessary. After the sleep, *BP* prints the initial board, and sends an acknowledgment message to *CP* via the pipe ℘. When *CP* reads it, the initialization of the game is complete (by this time, *PP* and its children (the player processes) should be ready; they have got about one second anyway).

PP first sends its PID to *CP* via the pipe ℘. *PP* then forks *n* player processes *A*, *B*, *C*, … After being born, each player process jumps to a function which never returns.

Continuation of the game

After the initial work, the processes in the application are engaged in the following (non-busy) waits.

- *CP* waits for the user to enter the next command.
- *PP* waits to receive the signal SIGUSR1 or SIGUSR2 from *CP*.
- Each player Π waits for the signal SIGUSR1 or SIGINT from *PP*.
- *BP* waits for the signal SIGUSR1 (from a player process) or SIGUSR2 from *CP*.

Now, suppose that the user enters the command *next* to *CP*. This kickstarts the next move that proceeds as follows.

- *CP* sends SIGUSR1 to *PP*, and waits until it gets an acknowledgment from *BP* (via the pipe ℘).
- *PP* decides who will be the next player Π to make the move.
- *PP* sends SIGUSR1 to Π, and itself again pauses.
- Π throws dice(s), and changes its position (or stays put in case of a move that is not permitted).
- If the move of Π leads it to the destination (Cell 100), it decrements $M_P[n]$, sends SIGUSR1 to *BP*, and exits.
- Otherwise, Π sends SIGUSR1 to *BP*, and goes to a pause again.
- *BP*, upon receiving SIGUSR1 from Π, prints the updated board.
- *BP* then sends an acknowledgment of completion to *CP* (via the pipe ℘), and goes to pause.
- *CP*, upon receiving the acknowledgment, becomes ready for the next move.

If the user is tired of manually initiating the next move, (s)he can set a *delay* in ms (the default is 1000ms, that is, one second), and continues the game in *autoplay* mode. In the autoplay mode, *CP* no longer waits for the user to enter the next command, but sleeps for the specified delay, and initiates the next move itself. This delay is for the user to see an animation of the progress of the game. The user may however specify a delay of 0, and the game should continue as expected. As explained above, each step of the synchronization $CP \rightarrow PP \rightarrow \Pi \rightarrow BP \rightarrow CP$ involves a blocking wait (for a signal or for a communication via the pipe ℘), so no real delays are needed for the synchronization. A delay is meant only for the human observer.

End of the game

The game ends when one of the following two things happens. These two situations are handled in the same way.

- The user supplies the command *quit* in the interactive mode (before the game is over).
- All players have reached the destination, that is, we have $M_P[n] = 0$ (in interactive or autoplay mode).

CP waits for the user to hit the return key. When the user is satisfied with the proceedings of the game (actually the correctness of the implementation), (s)he does as *CP* wants. After this, we have the following sequence of events.

- *CP* sends SIGUSR2 to *PP*.
- *PP* sends SIGUSR2 (or SIGINT or SIGKILL) to each player process that is still in the game, and waits for the termination of all the player processes. Note that a player process may terminate normally after reaching the destination cell, or after receiving the signal from *PP*. It stays in the system as a zombie process until *PP* eventually waits on it.
- *PP* then exits, letting *XPP* terminate too.
- *CP* catches the termination of *XPP* (waitpid). (*CP* cannot wait for its grandchild *PP*. *XPP* should wait for *PP*, but that is in the code of xterm, not your headache.)
- *CP* then sends SIGUSR2 (or SIGINT or SIGKILL) to *BP*.
- *BP* has no handler for this signal, and so it terminates. This causes *XBP* to terminate too.
- *CP* catches the termination of *XBP* (waitpid).
- *CP* removes the shared memory segments M_P and B_P, and exits. Notice that this removal succeeds only when there is no process still attached to the segments. Make sure that every process that uses the shared-memory segments detaches itself before exiting.

Signal handlers

From the above description, the processes need to catch and handle the following signals.

CP	No signal handling is required. Synchronization is via the pipe ℘.
PP	SIGUSR1 (from *CP* for initiating the next move) and SIGUSR2 (from *CP* to end the game).
A, *B*, *C*, …	SIGUSR1 (from *PP* for making the next move). SIGUSR2 from *PP* at the end should terminate a player, so this signal is not to be handled.
BP	SIGUSR1 (from Π as a printing request). SIGUSR2 from *CP* at the end should terminate *BP* (so no handler for this).
XPP, *XBP*	These are the xterm processes (precompiled). You cannot write signal handlers for them.

How to run an xterm

You may just type xterm from any shell. Then a terminal window will appear. By default, a shell (typically the user's log-in shell, like *bash*) is run in that window. However, you can run your own program in the xterm using the command-line option `-e`. Give the executable name after this. The remaining arguments in the call of xterm will go to the executable as its command-line parameters. Here is an example of `exec`()-ing xterm by *XPP*.

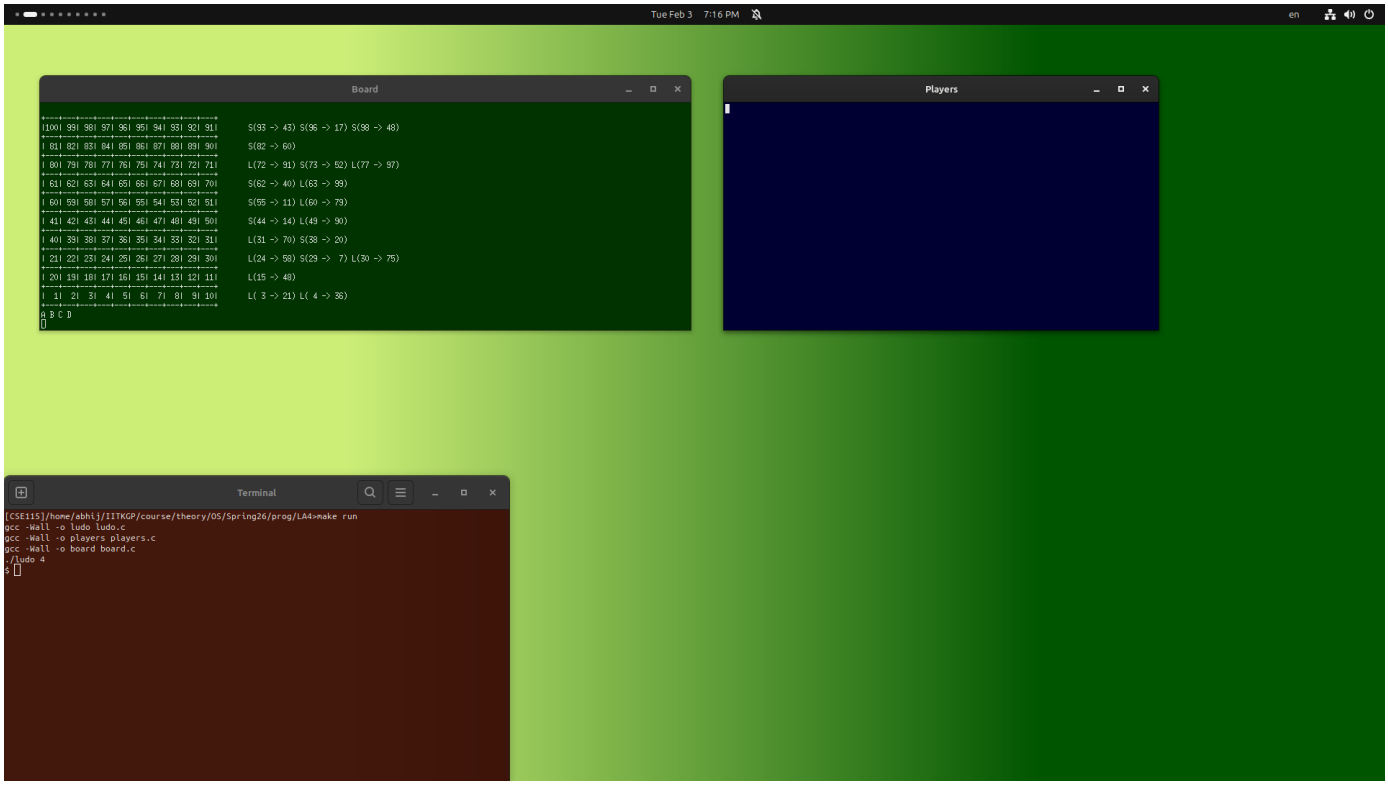
```
execlp("xterm",
       "xterm", "-T", "Players", "-fs", "15", "-geometry", "100x24+1000+100", "-bg", "#000033",
       "-e", "./players", narg, pfdarg, brdarg, NULL);
```

Here, `-T` sets the title for the xterm, `-fs` sets the font size, `-geometry cxr+i+j` configures the xterm to have *c* columns, *r* rows, and the top-left corner at the (*i*, *j*)-th pixel on the screen, and `-bg` sets the background color. The executable file `./players` takes three command-line arguments (all converted to strings): the number *n* of players, the descriptor for the write end of the pipe ℘ (for communication with *CP*), and the PID of *BP* (each player Π sends SIGUSR1 to *BP* after completing a move).

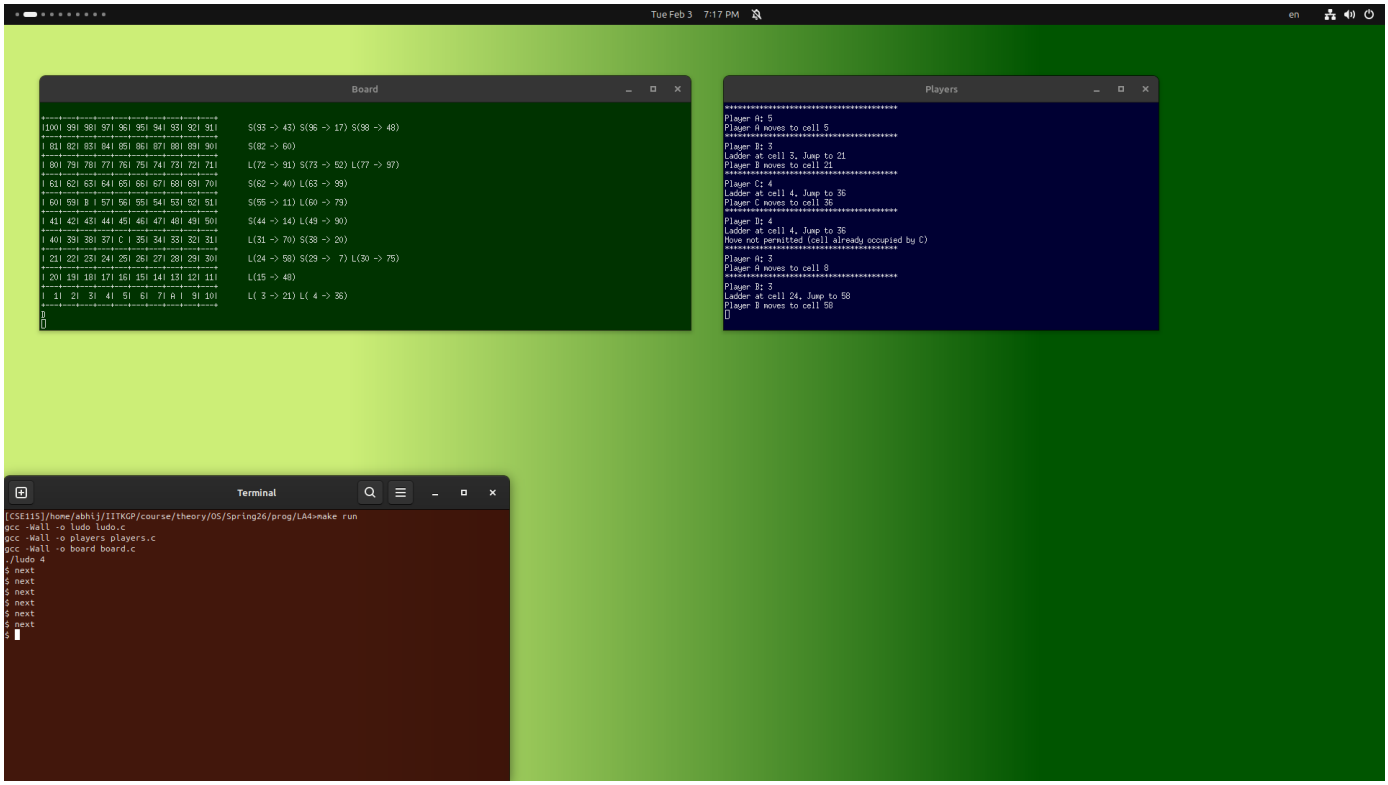
Submit a single zip/tar/tgz archive storing the three source files and a makefile with compile, run, and clean targets.

Sample Output

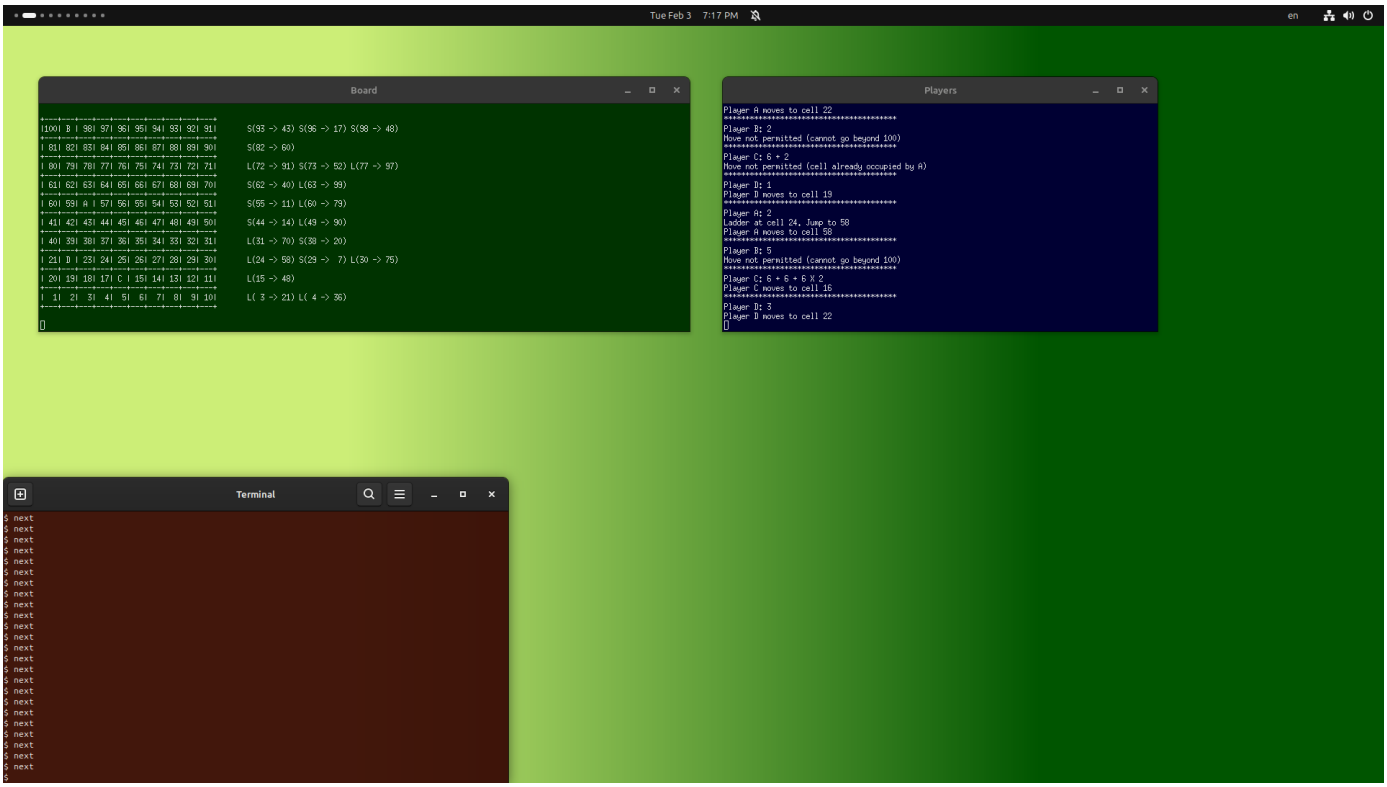
Screenshots are dumped below for demonstrating different stages of the game.



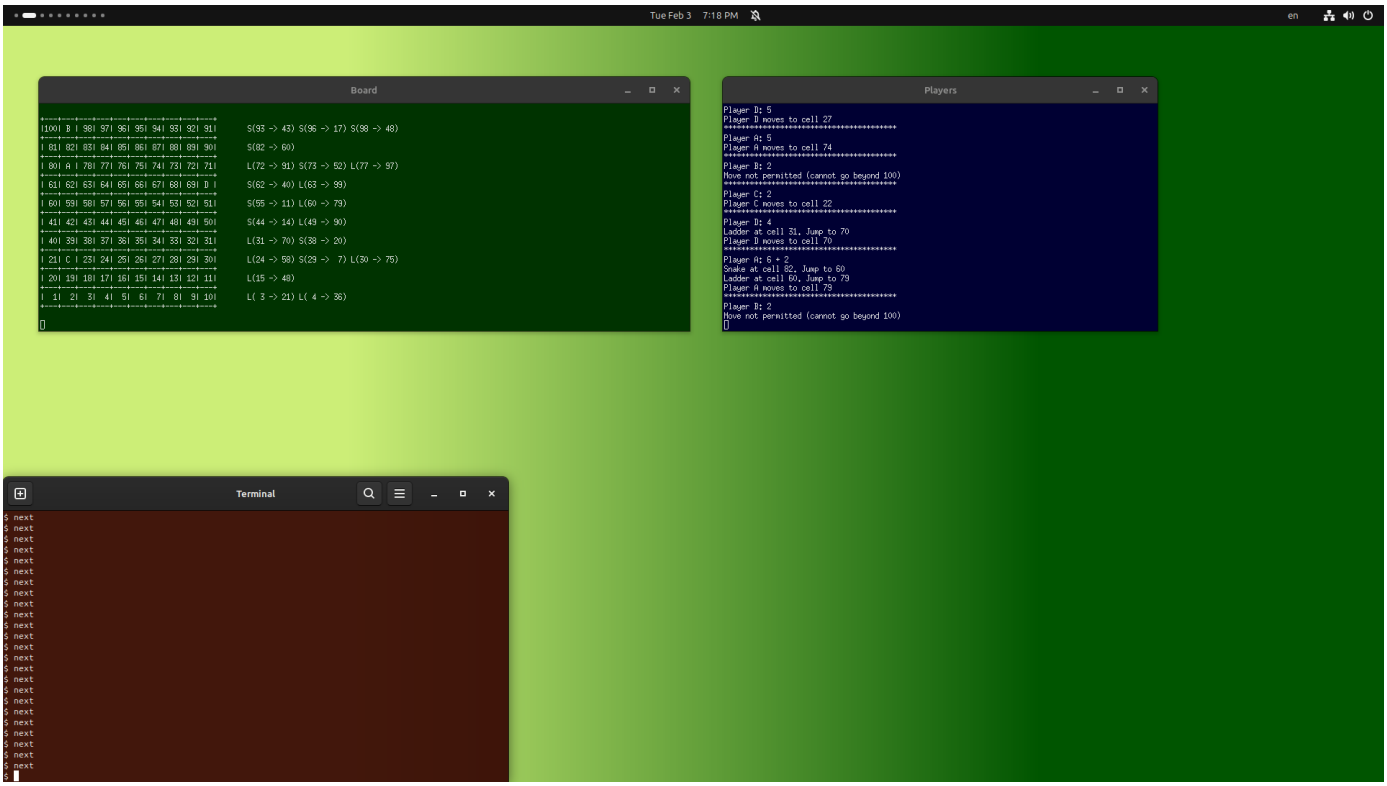
This is the beginning. The board is printed (see how the ladders and the snakes are shown). No player has made any move. The launching window is waiting for the first user command. All the players are home (line below the board).



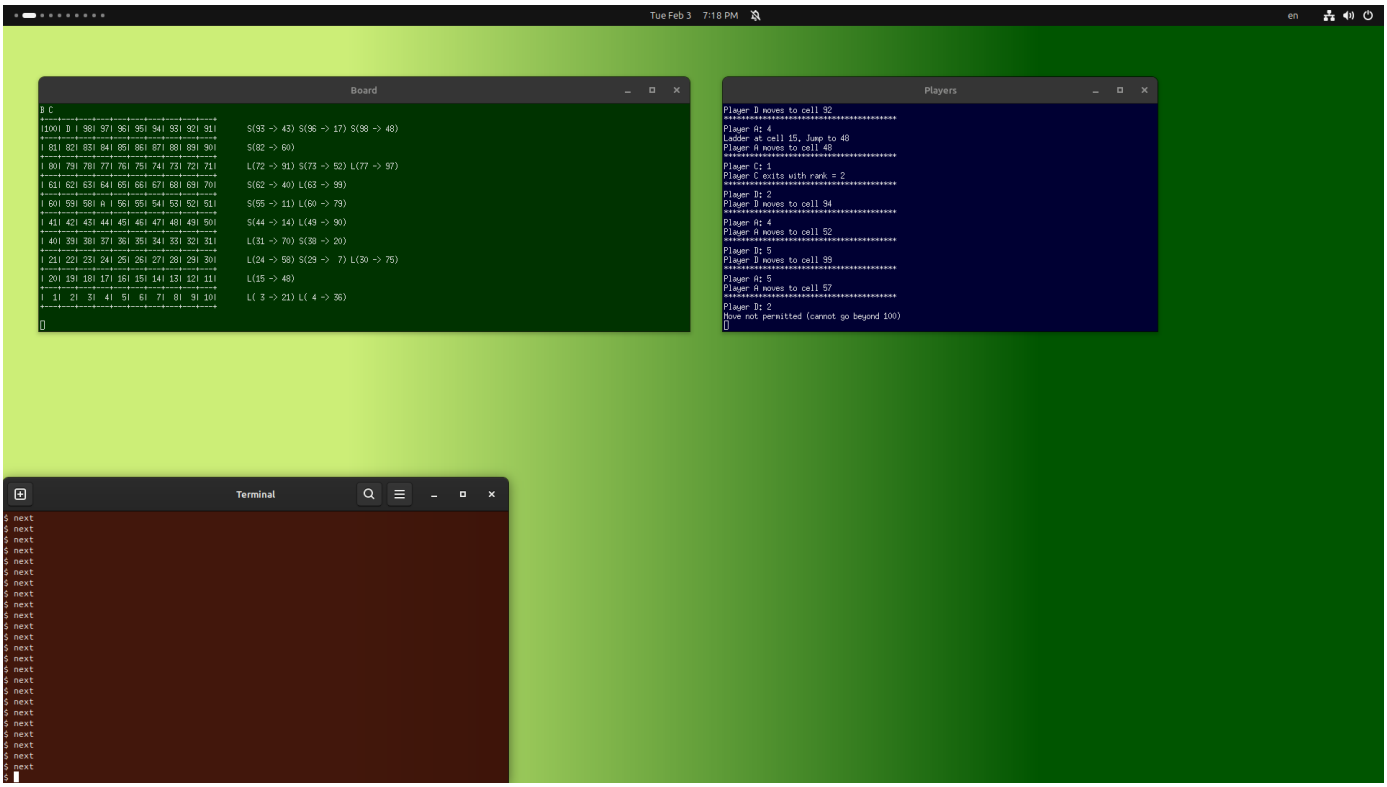
After the first six moves. Player *D* is still home. *A*, *B*, *C* have advanced. *B* has already encountered two ladders, *C* only one, and *A* none.



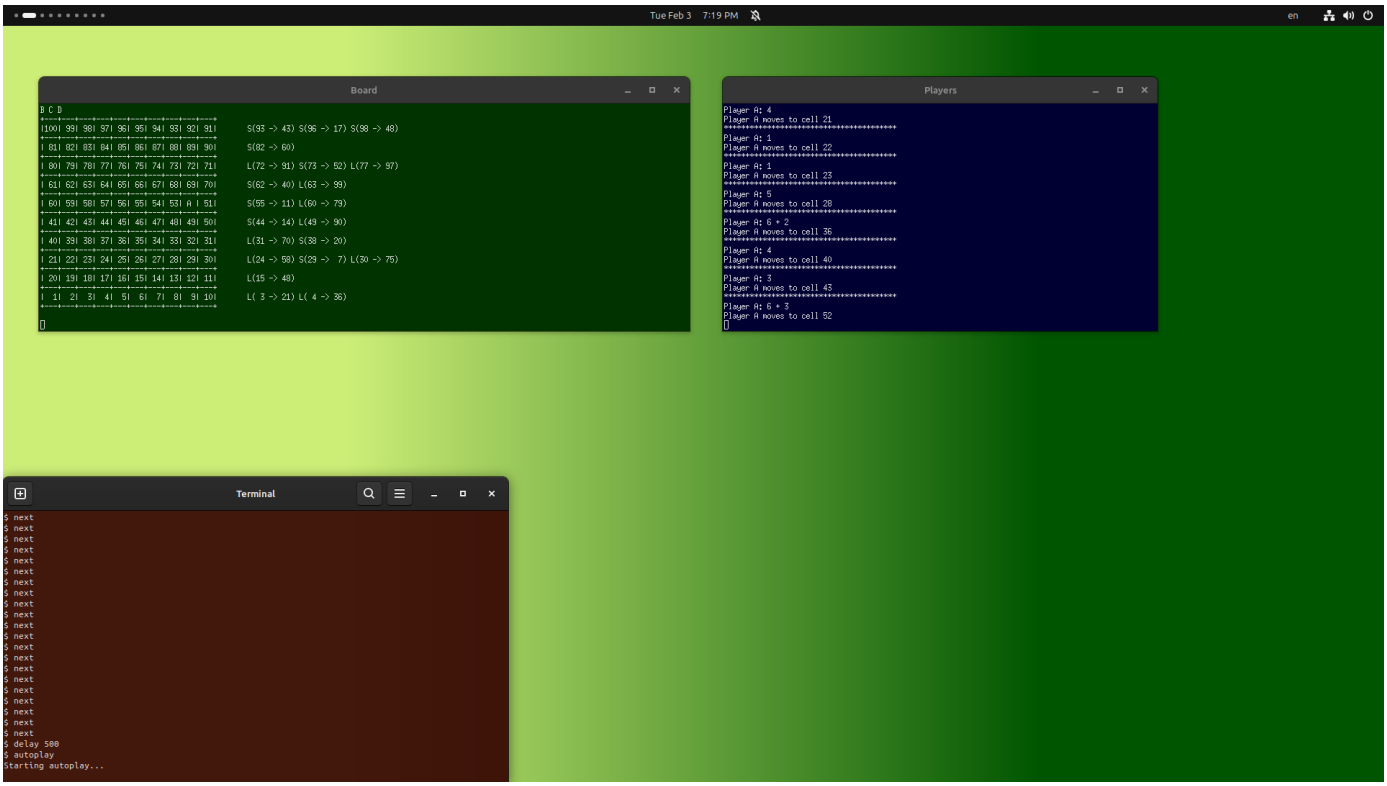
After a few more moves, *B* is almost at the destination, but its last two moves are not valid; it needs 1 to reach 100. Also look at *C*'s move near the bottom. Three 6's are canceled (shown by X), so its next move is 2 (not 20).



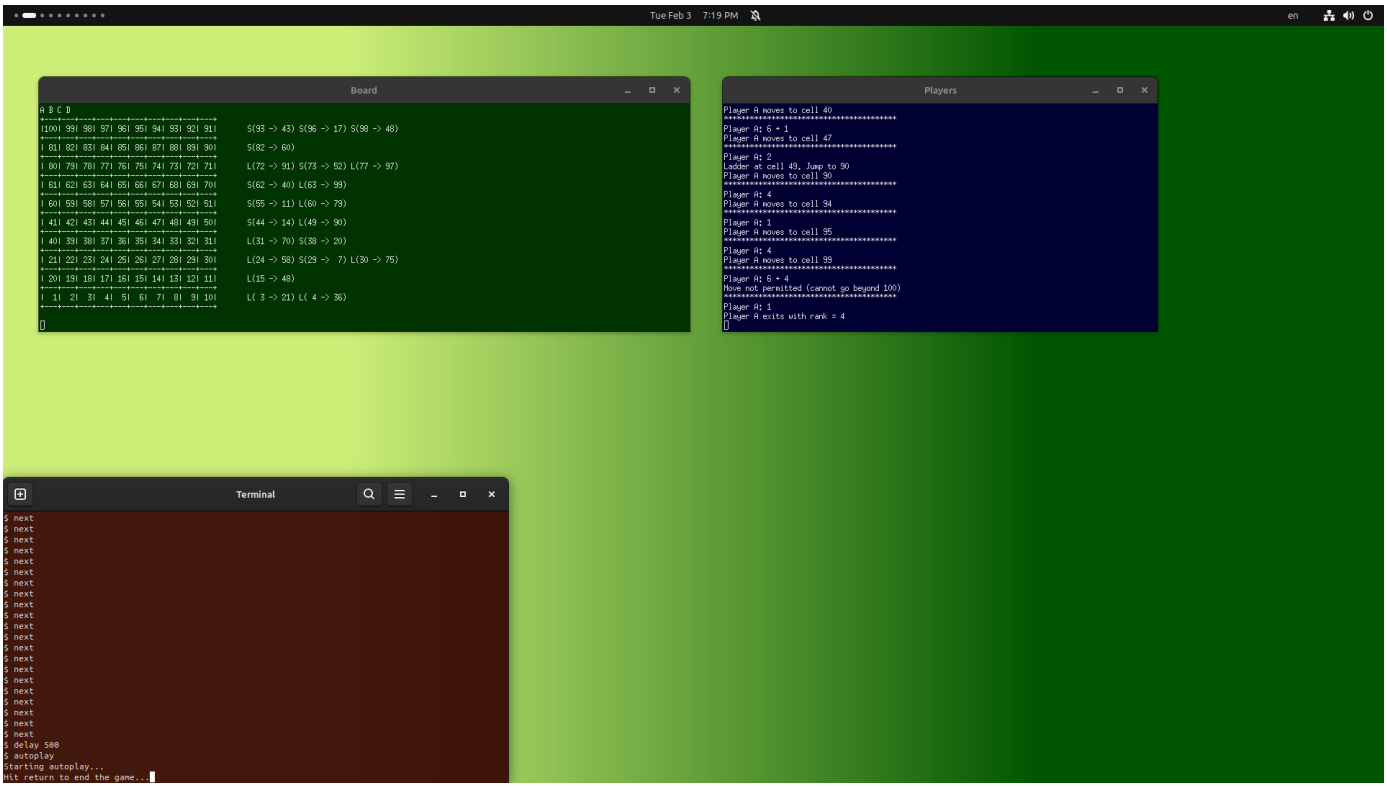
Look at the next move of Player *A*. The dice outcomes are 6 + 2, and *A* moves to Cell 82. At that cell, it hits a snake, and falls to Cell 60. This new position has a ladder to Cell 79, so the ladder is taken, and *A* ends up at Cell 79.



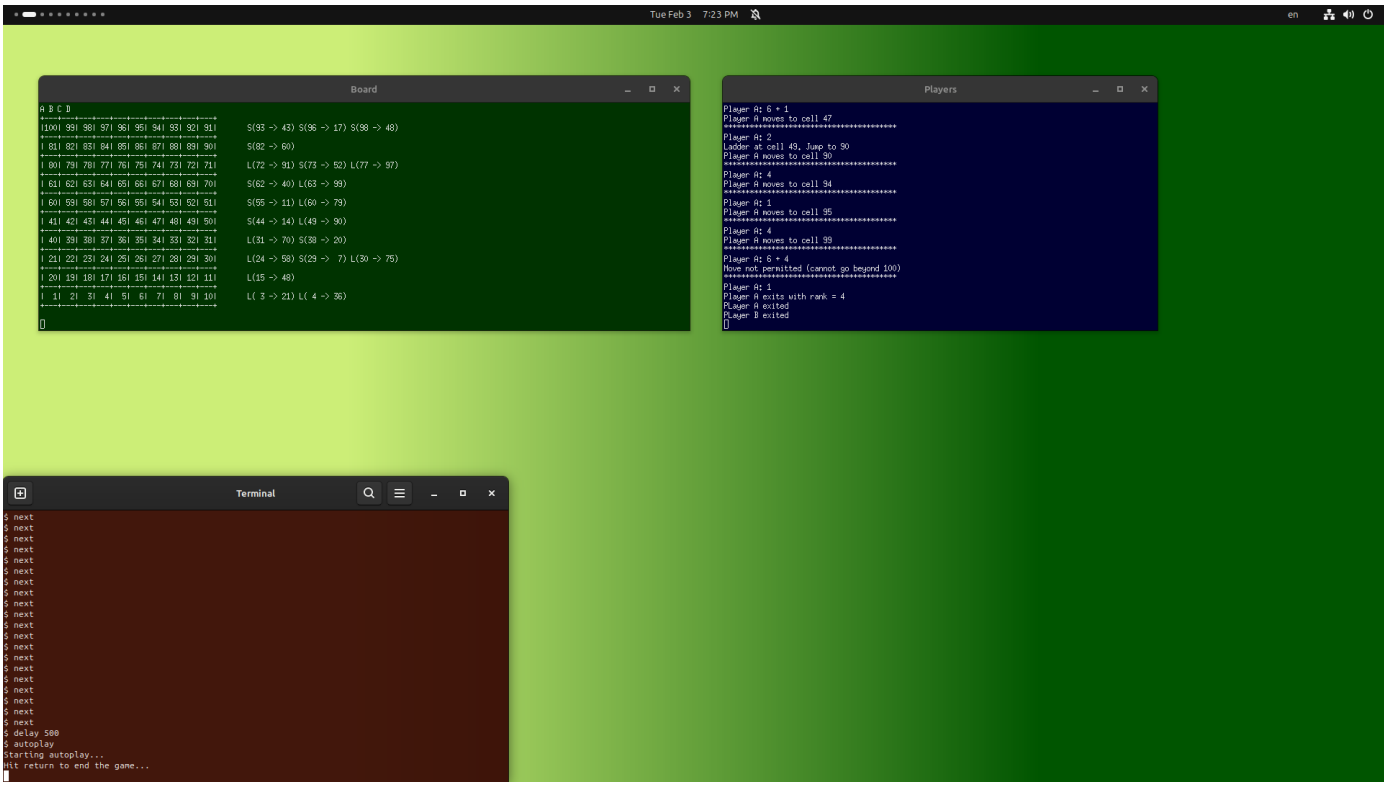
B and *C* are done, that is, reached the destination 100. This is shown above the board (not in the cell 100 itself). The moves now alternate between *A* and *D*.



The user prefers to switch to the autoplay mode. A *delay* of 500ms is set, and then the next command entered is *autoplay*. Now, the game will continue itself, making 2 moves per second. The user can see (not from this still image) how the game unfolds.



All the players have reached the destination. The user may have a final look at the transcripts before hitting return.



PP waits for catching the termination signal from the player processes. A wait of 1 second is added after each waitpid, so that this creates a "slow" animation (the delay is otherwise unnecessary).

After the terminations of all the player processes, the remaining processes *PP*, *XPP*, *BP*, and *XBP* terminate, and the two xterms (green and blue) close. Eventually, the process *CP* at the launch window also terminates, and the shell prompt returns.