Middle East Technical University Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2024 - 2025

Homework 1 - Multiplayer Tic-Tac-Toe Game

Due date: 30 03 2025, Sunday, 23:59

1 Overview

In this homework, you will implement a multiplayer tic-tac-toe game that functions in real-time rather than turn-by-turn. Multiple player processes, each running its executable, will communicate with a central server process through interprocess communication (IPC). Your primary goal is to implement the server, which manages the shared game state, processes player messages, and decides the winner based on a chosen streak size.

Keywords: *tic-tac-toe, multiplayer, real-time, ipc, pipe*

2 Introduction

The tic-tac-toe is generally played on a grid 3×3, where players alternately place marks to achieve three in a row, column, or diagonal. In this homework, we extend this concept to a larger grid and allow multiple players to compete in real-time, not turn-based. A separate process represents each player, and the game continues until one player achieves the specified streak size in a row, column, or diagonal. The game ends in a draw if the entire grid is filled without any player meeting the winning condition.

Each position on the grid can be marked only once; once a player places their mark on a position, it cannot be overwritten by another player.

You will focus on implementing the server, which is responsible for:

- Reading the game configuration (grid size, streak size, player details) from `stdin`.
- Creating bidirectional pipes to communicate with each player process.
- Forking and executing each player process, redirecting their `stdin` and `stdout` to these pipes.
- Managing messages from players:
  - **START**: Indicates a player has begun execution; the server should respond with the current game state.
  - **MARK**: Sent when a player attempts to mark a specific point on the grid. The server checks if the move is valid and updates the game state accordingly.
- Determining when a player has won by achieving the streak size or when the game ends in a draw, and notifying all players.

- Reaping child processes to avoid zombies.

The game is not turn-based; players can send **MARK** messages at any time, and the server must handle these concurrently. The server must ensure that only valid moves are accepted and that the game state is updated consistently.

Figure 1 illustrates the possible winning streaks on a larger grid, reflecting the updated game version with a customizable grid and streak sizes.

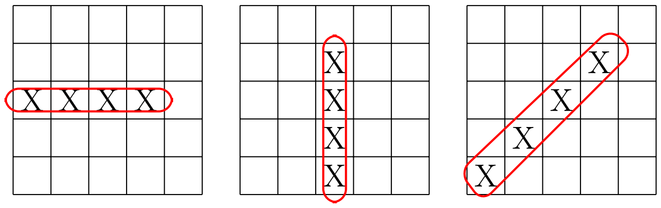

Figure 1: Examples of winning streaks on a 5×5 grid with streak size 4: horizontal, vertical, and diagonal.

Figure 2 illustrates a partially filled 6×6 board with three players, each striving to achieve a streak of 4. At this stage, no one has met that requirement.

X			O		O
	X	Y		O	
	X	X	Y		Y
Y		X	O	O	
X	O	Y		X	Y
O	O		X	Y	X

Figure 2: A partially filled 6×6 grid with three players (X, O, and Y), each aiming for a streak of 4.

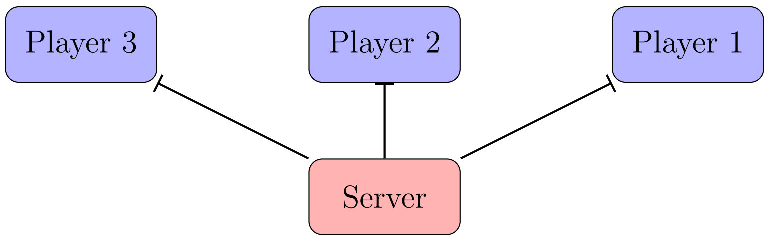

Figure 3: Server communicating with player processes

3 Game and Implementation Details

The server begins by reading the grid dimensions, streak size, and player details from standard input. It then creates bidirectional pipes to communicate with each player process. After forking the player processes, redirecting their standard I/O to these pipes, and executing the player executables with their respective arguments, the server will wait for and process incoming messages.

Player processes may send two types of messages:

- **START**: Sent when a player process is initialized. The server responds by sending the current state of the board.
- **MARK**: Sent when a player attempts to mark a specific position on the grid. The server must check if the position is valid (within bounds and not already marked). If valid, the server updates the grid with the character of the player and checks if this move results in a win or a draw.

The server must handle these messages concurrently, as players can send messages at any time. After processing a message, the server responds with either:

- **END**: If a player has won or the game is a draw, indicating the game is over.
- **RESULT**: For messages **START** and **MARK**, indicating whether the mark was successful (for **MARK**) and providing the current game state.

Game play continues until one player achieves the required streak size or until the board is fully occupied and no winner emerges. In either scenario, the server must notify all players and terminate the game.

The algorithm 3 provides a high-level overview of the main server loop.

Algorithm 1 High-level Server Loop

```
Initialize game state from input
Set up communication channels with players
while game is not over do
    Wait for player messages using select or poll
    Process each incoming message
    Update game state if the message is a valid mark
    Check if the game is won or if the grid is full
    Send appropriate responses to players
end while
Announce the winner or declare a draw
Clean up resources and terminate
```

Note that the server must handle multiple players sending messages concurrently. The server should process messages in the order they are received, updating the game state accordingly.

4 Inter Process Communication

4.1 Bidirectional Pipe

The communication between the server and player processes will be carried out via bidirectional pipes, created using:

```
#include <sys/socket.h>
#define PIPE(fd) socketpair(AF_UNIX, SOCK_STREAM, PF_UNIX, fd)
```

This setup allows for bidirectional communication, where each end of the pipe can read and write data.

The server must handle multiple players concurrently. It should use `select()` or `poll()` to check for incoming data on the player pipes without blocking.

4.1.1 Handling Multiple Connections

To manage multiple player connections concurrently without blocking, the server will use the `select()` or `poll()` system calls. These functions are essential for handling I/O operations on multiple file descriptors efficiently.

- `select()`: This function allows the server to monitor multiple file descriptors and wait until one or more of them become "ready" for some class of I/O operation (e.g., reading or writing). The server can specify a timeout, after which `select()` returns even if no file descriptors are ready. This prevents the server from blocking indefinitely. For example, the server can use `select()` to check all player pipes simultaneously, processing a message as soon as its pipe becomes readable.

- `poll()`: Similarly to `select()` , `poll()` waits for one or more file descriptors to become ready for I/O. It offers more flexibility by allowing the server to specify which events to monitor for each file descriptor (e.g., readability, writability) via a struct array. This makes `poll()` particularly useful when dealing with a large number of connections, as it avoids the file-descriptor limit issues that `select()` might encounter.

In this assignment, the server will use these functions to monitor the pipes connected to each player. When a player sends a message, the corresponding pipe becomes readable and the server can read and process the message immediately. This ensures that the server can handle messages from multiple players in real time without being blocked by any single player's I/O operation. Students are expected to learn how to use these functions by referring to the man pages (e.g. `man select`, `man poll`) or other relevant resources, focusing on their basic usage and their role in enabling concurrent message handling.

4.2 Messages

Players and the server exchange messages through simple C structs. These messages are processed immediately upon arrival; there is no buffering or message queue.

Client Messages (from player to server):

```
typedef enum client_message_type {
    START,
    MARK
} cmt;

typedef struct client_message {
    cmt type;
    coordinate position;  // Used only for MARK message
} cm;
```

Server Messages (from server to player):

```
typedef enum server_message_type {
    END,
    RESULT
} smt;

typedef struct server_message {
    smt type;
    int success;  // 1 if mark was successful, 0 otherwise
    int filled_count;  // Number of filled positions
} sm;

typedef struct grid_data {
    coordinate position;
    char character;
} gd;
```

- For **START** and **MARK** messages, the server responds with a `RESULT` message containing:
  - `success`: 0 for **START**, 1 if **MARK** was successful, 0 otherwise.
  - `filled_count`: Number of positions currently marked on the grid.
- The message is followed by `filled_count` `grid_data` structures detailing each marked position.
- If a player wins or the game is a draw, the server sends an `END` message to all players, indicating that the game is over.

5 Input&Output

5.1 Input

The server reads the configuration from `stdin`, following this format:

```
<grid_width> <grid_height> <streak_size> <player_count>
<player1_character> <player1_total_argument_count>
<player1_executable_path> <player1_arg1> ... <player1_argM>
...
<playerK_character> <playerK_total_argument_count>
<playerK_executable_path> <playerK_arg1> ... <playerK_argL>
```

- `<grid_width>` and `<grid_height>` define the size of the grid.
- `<streak_size>` is the number of consecutive marks required to win.
- `<player_count>` is the number of players.
- For each player:
  - `<player_character>` is the character used by that player to mark the grid.
  - `<player_total_argument_count>` is the number of arguments (excluding the executable path).
  - `<player_executable_path>` is the path to the player's executable.
  - Then follow the player's arguments.

5.2 Output

You will use a provided function to log key game events. The relevant structures and function prototype are shown below:

```
typedef struct client_message_print {
    pid_t process_id;
    cm *client_message;
} cmp;

typedef struct server_message_print {
    pid_t process_id;
    sm *server_message;
} smp;

typedef struct grid_update {
    coordinate position;
    char character;
} gu;

void print_output(cmp *client_msg, smp *server_msg, gu *grid_updates, int update_count);
```

Call `print_output` in the following scenarios:

1. When receiving a message from a player:

   ```
   print_output(client_msg, NULL, NULL, 0);
   ```

2. When sending a message to a player:

   - For `RESULT` messages, include grid updates:

     ```
     print_output(NULL, server_msg, grid_updates, filled_count);
     ```

   - For `END` messages:

     ```
     print_output(NULL, server_msg, NULL, 0);
     ```

When the game ends, the server must print the outcome to standard output:

- If a player wins, print: `Winner: Player$` (where '$' is the letter of the winning player).
- If the game is a draw, print: `Draw`.

For example, in your code, you could use:

```
printf("Winner: Player%c\n", winner_character);  // For a win
printf("Draw\n");                                 // For a draw
```

This output should be printed just before the server terminates, after notifying all players with the `END` message.

6 Specifications

- The server must terminate when a player wins by achieving the required streak size or when the grid is fully marked without a winner.
- All player processes run concurrently, leading to non-deterministic output order. The evaluation will focus on correctness, not output order.
- The coordinates (x, y) correspond to columns and rows, with $(0, 0)$ at the top-left corner.
- Player characters must be single uppercase or lowercase letters (A-Z or a-z). No other characters (e.g., numbers, symbols) are allowed.
- Use a 1 ms sleep in the server loop to prevent CPU hogging. This can be done with a timeout.
- The server must reap all child processes to avoid zombie processes.
- Do not modify the provided header and source files; they will be replaced during grading.
- Students are not responsible for implementing the player processes; precompiled executables will be provided for testing and grading.
- Ensure no unnecessary output; evaluation is based on black-box testing.

7 Regulations

- **Programming Language:** Your program should be coded in C or C++. Your submission will be compiled with `gcc` or `g++` on department lab machines. Make sure your code compiles successfully.
- **Late Submission:** Late submission is allowed but with a penalty of $5 * day * day$.
- **Cheating:** Everything you submit must be your work. Any work used from third-party sources will be considered as cheating, and disciplinary action will be taken under our "zero tolerance" policy. This includes large-language models (LLMs).
- **Newsgroup:** You must follow the ODTUClass for discussions and possible updates daily.
- **Grading:** This homework will be graded out of 100. It will make up 10% of your total grade.

8 Submission

Submit via ODTUClass a tar.gz file named `eXXXXXXX.tar.gz` containing your source code (excluding the provided files) and a makefile. The makefile should produce an executable named `server`. Ensure that the tar file does not have directories. Test with:

```
$ tar -xf eXXXXXXX.tar.gz
$ make
$ ./server
```