National University of Computer and Emerging Sciences, Lahore Campus

	Course:	**Operating Systems**	**Course Code:**	**CS 2006**
	Program:	**BSE (5B-5A)**	**Semester:**	**Fall 2024**
	Due Date	**Sunday 13th Sept 2024 (11:59 PM)**	**Total Marks:**	**75 marks**
	Type:	**Assignment 2**	**Page(s):**	**4**

Important Instructions:

- **Do not submit zip files else there will be deduction of marks**
- **Do not use Fstream and Fget in the questions.**
- **Late submission of your solution is not allowed.**
- **You are not allowed to copy solutions from other students. We will check your code for plagiarism using plagiarism checkers. If any sort of cheating is found, heavy penalties will be given to all students involved.**
- **Submit the code files. Do not submit Zip files.**

Question 1 : BattleShip Game **[50 marks]**

Design a **Battleship game** in C++ using **pipes**, where two players (**Player 1** and **Player 2**) play against each other on a 5x5 grid. The game will be played between two separate processes, and all communication between the players will happen through **pipes**. Each player controls a fleet of 3 ships, and the goal is to sink the opponent's ships by guessing their positions.

Requirements:

1. **Game Board**:
    - Each player has a **5x5 grid**, where each space can either be **empty** or **occupied by a ship**.
    - Each player controls **3 ships** placed at the start of the game. The ship sizes are fixed: 2, 3, and 4 spaces long, and can be placed either horizontally or vertically on the grid.
    - Players place their ships on their grid before the game starts, with proper validation to avoid overlapping ships.
2. **Pipes for Communication**:
    - The game will use **two pipes** for communication between **Player 1** and **Player 2** processes.
        - **Pipe 1**: Used for Player 1 to send their guesses to Player 2.
        - **Pipe 2**: Used for Player 2 to send their guesses to Player 1.

    - Players will also communicate their responses (hit or miss) using the same pipes, ensuring a turn-based gameplay loop.
    - Players **must wait** for the other player's guess and response before proceeding with their own turn.
3. **Turn-Based Gameplay**:
    - Players take turns guessing the opponent's ship positions by sending **(row, col)** coordinates through the pipe.
    - After a guess, the opponent responds with either:
        - **H** (Hit) if a ship is hit.
        - **M** (Miss) if the guess misses the ships.
    - The response is sent back through the same pipe.
4. **Game Termination**:
    - The game ends when one player successfully sinks **all 3 of the opponent's ships**.
    - Each ship requires multiple hits to sink, based on its size (e.g., a ship of size 3 requires 3 hits to sink).
    - Once a player sinks all the opponent's ships, the game declares the winner and both processes terminate gracefully.

Game Flow:

- **Player 1 Process**:
    1. **Place Ships**: Before the game starts, Player 1 places their ships on a private 5x5 grid.
    2. **Send Guess**: Player 1 sends a guess **(row, col)** to Player 2 through **Pipe 1**.
    3. **Receive Response**: Player 1 waits for Player 2's response (Hit/Miss) through **Pipe 1**.
    4. If all of Player 2's ships are hit, Player 1 wins, and the game ends.
    5. If Player 1 does not win, control is passed to Player 2.
- **Player 2 Process**:
    1. **Place Ships**: Player 2 places their ships on their private 5x5 grid.
    2. **Receive Guess**: Player 2 receives Player 1's guess **(row, col)** through **Pipe 1**.
    3. **Check Hit or Miss**: Player 2 checks if Player 1's guess hits or misses a ship on their grid and sends the response (Hit/Miss) back to Player 1 through **Pipe 1**.
    4. **Send Guess**: Player 2 then sends a guess **(row, col)** to Player 1 through **Pipe 2**.
    5. **Receive Response**: Player 2 waits for Player 1's response (Hit/Miss) through **Pipe 2**.
    6. If all of Player 1's ships are hit, Player 2 wins, and the game ends.
    7. If Player 2 does not win, control is passed back to Player 1.

Synchronization:

- The pipes are used for **communication**, ensuring that only one player can make a move at a time.
- The guessing process is synchronized using the turn-based system, where a player sends their guess and waits for the opponent's response before taking their next turn.
- There should be proper synchronization to ensure that one player cannot make multiple guesses in a row without waiting for the other player's move.

Pipes Error Handling:

- Implement error handling for the pipes to ensure proper communication. If there's a failure to send or receive data through the pipes, the game should handle this scenario gracefully.

Question 2 : File Descriptor Duplication and Redirection using dup() and dup2() **[25 marks]**

In this Question, you will use file descriptor duplication and redirection in C++ to manage file input/output in a controlled way. You will implement a program that reads input from a file, processes the data, and redirects the standard output and standard error using dup() and dup2().

Problem Statement:

Create a C++ program that performs the following tasks:

1. **File Input and Output Redirection**
    - The program should accept **three command-line arguments**:
        1. Input file (input.txt) to read data from.
        2. Output file (output.txt) to redirect standard output (stdout).
        3. Error file (error.txt) to redirect standard error (stderr).
    - The program should open the input file for reading, and the output and error files for writing.
2. **Duplicating and Redirecting stdout and stderr**
    - Use dup() to save the original stdout and stderr file descriptors.
    - Use dup2() to redirect stdout to the output file and stderr to the error file.
    - Output some processed data to the new stdout (which is redirected to the output file) and simulate an error message output to stderr (redirected to the error file).
3. **Restoring Original File Descriptors**
    - After writing to the files, restore the original file descriptors for stdout and stderr using dup() and dup2().
    - Output a confirmation message to the console that shows the file descriptors were restored.

Functionality :

1. **File Handling**:
    - Open the input file and read data (for example, read lines of text or numbers).
    - Redirect stdout and stderr using dup2() to output data and error messages into their respective files.
2. **Error Simulation**:
    - Trigger an error scenario by handling some incorrect data from the input (for example, reading invalid numbers or dividing by zero) and print an error message to stderr (which should now be redirected to the error file).
3. **Restoring Standard Output and Error**:
    - After processing is complete, restore the original stdout and stderr to their normal states (i.e., back to the console).

Constraints:

- **Error Handling**:
    - You should handle invalid file paths for the input, output, and error files.
    - If the input file cannot be opened, output an appropriate error message using stderr (without redirection).

Sample Input: 10 5 0 abc 15

Sample Output:

Output.txt:

Processing input file...
Read number: 10
Processed number: 10
Read number: 5
Processed number: 20
Read number: 0
Error: Division by zero encountered.
Read number: 15
Processed number: 6

Error.txt:
Error: Invalid number format for 'abc'
Error: Division by zero encountered.