Chess Kernel

Overview:

The program aims at implementing a kernel module that is capable of playing chess. The game would be between a human player and the computer.
Although the logic for the game chess can be very complex and hard to comprehend few specific details, we have tried to simplify some specific implementation for this project.

Goals:

The goals to the project is as follows:

- To learn how to implement kernel modules
- To develop a game which helps in understanding the key implementation ideas
- To strong sense on implementation logic and control flow
- To help get a strong implementational knowledge of C programming in Linux kernel

Proposed Solutions:

The solution to the chess logic can be very complex, wherein, we could be overwhelmed by the various cases that might occur ending up not finding any generic solution to it. So, for the most part of the assignment, we have tried to simulate the real chess game.
To implement the logic, we didn't use any exaggerated data structure and kept it simple for the most part. We used a static 3d matrix to retain the board state between function calls. It's an 8x8 matrix where each cell can contain 2 characters to define the piece at that cell.
The user input is read and sent to the kernel buffer which we use to analyze if the command received is valid.
The analysis to decide if the command received is valid or not is done through checking each character of the buffer. They should start with a 0followed by either 0 or 1 or 2 or 3 or 4, depending on which is decided which task to perform.

- If the second character is 0 it would imply that the game has started or restarted depending on if the game was previously continuing.
- If the 2nd character is 1 it would imply to show the board to the screen. In this case the kernel buffer is filled with the board string characters
- If the 2nd character is 2. In this case the following checks need to be done:
    - The 3rd character needs to be a space.
        - If the space is miss placed with character 'W' or 'B' that would imply that the following characters have been written without taking into

consideration the space. This would mean the command is in INVALID FORMAT and thus the kernel buffer is filled with INVFMT to specify that.
        - If the space is miss placed with some other symbol then probably the command entered is WRONG and so UNKCMD is filled in the kernel buffer to specify UNKNOW COMMAND
    - The 4th character needs to match with whatever the player has chosen to pick in the beginning. If not then:
        - For example, if the player has chosen “W” but the character here is “B” or vide versa, then the player is playing OUT OF TURN and OOT sent in the kernel buffer.
        - If some other character is present, then the command is probably UNKOWN, so UNKCMD is copied to kernel buffer.
    - The 5th character should be within the range of ‘a’ to ‘h’ and the 6th character should be within ‘1’ to ‘8’. These denotes the starting square of the move.
    - The 7th character should be ‘- ‘to separate the two (starting and ending) coordinates.
    - The 8th and the 9th character should be same as the 5th and the 6th character.
    - If any of these checks fails, we copy “UNKCMD” into the kernel buffer.
    - After the check succeeds, we calculate the matrix index of the starting move point and the matrix index for the ending move.
        - This is done by simply subtracting from 5th character of the kernel buffer, ‘a’ since that would give a 0 based index which is suitable for the matrix indexing. This now gives us the x coordinate of the starting move
        - To get the y coordinate of the start move, we use the 6th character and subtract from it character ‘0’ which tells how much distance 6th character is from ‘0’ in a 0 based index notation.
        - We do the same calculation with the 8th and the 9th character as well. This gives us the coordinates of end moves.
    - Once we are done calculating the moves coordinates in terms of matrix indices, we verify if the moves are valid. To do so,
        - We check if the piece mentioned in the command is present in the calculated start moves coordinates.
        - If the piece is not present already, in the starting coordinates then the move is invalid and the kernel buffer is filled with “UNKCMD”
        - Otherwise, we move on to check for the 10th character.
        - If the 10th character is a new line character, then, it means the command has ended until there, in which case we check if the ending position of the board that we calculated has an empty cell (i.e. “**”) in position.
            - If an empty cell is found
            - Then we check if the movement we are making on the piece is valid. Like the pawn can only move in forward direction, while the bishop always moves diagonally, etc.
            - If the movement of the pieces are found to be correct, we

        - we replace the characters in the cell with the characters for the piece mentioned and verified previously in the command.
        - Otherwise, the move is designated as INVALID. So “ILLMOV’ is copied to kernel buffer
- If the 10th character is ‘x’ it means that the player piece is taking out some opponent’s piece. In order to verify it:
    - We check if the ending coordinates in the board is has any piece of opponent’s exists in the mentioned position.
    - If the ending coordinates has an opponent’s piece
        - We 1st check if the piece that we are trying to move is of player’s
        - Then we check if the movement we are making on the piece is valid. Like the pawn can only move in forward direction, while the bishop always moves diagonally, etc.
        - If the movement of the pieces are found to be correct, we replace the characters in the ending move’s cell with the player’s piece mentioned and previously verified in the command, while also replacing the starting cell’s characters with “**” to signify empty cell.
        - We copy into kernel buffer “OK” to signify that all the checks were passed and the necessary movements were made or “UNKCMD” if incase any of the checks failed.
    - If the ending moves coordinates does not have any opponent’s piece, the command is marked as INVALID and “ILLMOV” response is copied to the kernel buffer
- If the 10th character in the buffer is ‘y’, this means that the player is trying to promote a pawn to a piece that is denoted by the 11th and 12th character.
    - We 1st check if the piece player is trying to move is that of player
    - We then check if the piece the player is trying to promote is a pawn (“P”) since only pawns can be promoted.
    - We then check the piece the player is trying to promote to is that of player
    - We check if the starting piece contains the piece that the player mentioned in the command.
    - We also check if the ending coordinated of the move has an empty cell (denoted by “**”)
    - And lastly, we check if the piece that the user is trying to promote to is either a Queen or a Bishop or Knight or Rook. No other piece can the pawn be promoted to.
    - If all the checks have passed, we replace the characters in the ending coordinates with the characters of the piece the player is trying to promote to and the characters in the starting coordinates with “**” to signify the empty cell that the pawn has left.

- If any of the checks fail, we copy “ILLMOV” or “UNKCMD” accordingly.

- Case of Check:
  - To check if any checks has been placed on the player’s king or the opponent’s king, we do the following:
    - We 1st check whose turn it is. If its player’s turn, we search for the CPU’s king and vice versa.
    - We move in the diagonal direction from the king’s position and check if the 1st cell that is encountered that is not an empty cell, is a Bishop or a Queen of the opponent’s (can be player or CPU depending on whose turn it is) color.
    - If we find an opponent’s bishop or queen it’s a check
    - If not we continue looking in the up, down, left and right direction from the king’s position for opponent’s Rook or Queen
    - If we find an opponent’s rook or queen it’s a check
    - Otherwise we check if any of the opponent’s knight is in a position to check the king. We already know the moves a knight can make, (‘L’ shape). Accordingly, we check if there is any knight that can check.
    - If no knight is present to check the king, we lastly check if any opponent’s pawn can check diagonally one step forward. If there is threatening pawn then the check is placed otherwise not.
    - If a check is placed, then we copy “CHECK” into the kernel buffer and depending on whose turn it is the opponent is placed a check.

- Case of Checkmate:
  - The case of check mate was the most complex to handle. We have tried an easier and simpler version of the check mate that usually happens in real chess and is more complex than that implemented here.
  - We have implemented, that, if a king is taken down in any valid manner already described above by any of the opponent’s king, the game is over and the player whose king has been taken down is checkmated.
  - If check mate has been done, the game is over and the “MATE” string is copied into the kernel buffer.
  - When the game is over no other moves by either the user or the computer is allowed.

Questionable:

Here we would like to discuss how we could have improved the game implementation and features. The 1st thing to improve is the check mate feature. Here we have only implemented the easiest version, but, we can use complex logic to check if the opponents king, which has to be currently at check, has any valid moves left such that that king can be at least temporarily safe. If no such moves are found and the check cannot be avoided by taking down any of the opponent's pieces to threat then the king is mate, otherwise not.
We can check if the surrounding squares of the checked king that is free, has any potential threat on them. If not then its not a case of checkmate.

We can also improve the functionality by making the CPU move in calculated direction. This can be done by some of the AI/ML algorithms and minmax algorithm.