Yousuf Asfari

Project 2 Part 2 Design Document

Purpose

The objective of this undertaking was to design and construct a Linux chess character virtual device driver that the user can read/write with commands. The driver is to allow for typical two-player chess, where the user plays against the CPU by using character input of the format laid out by the project description. The driver is expected to perform user input error handling in addition to modifying a virtual chess board and abiding by the rules of chess.

Requirements and Goals

At the time of the construction of this document, the basic requirements of project 2 were met, without implementation of extra credit goals (although work on the stalemate verification functionality is currently in progress). The basic goals included the following:

- **I.** Verification of user input to match one of the four valid commands, including:
  - **a.** "00 C\n", to begin a new game with the user having the color *C* (either 'W' or 'B') and reset the character chess board
  - **b.** "01\n", to return the character chess board state unless no game was played
  - **c.** "02 *MOVE*\n", to make a specified move for the human player assuming validity
  - **d.** "03\n", to make a valid CPU move
  - **e.** "04\n", to resign the human player and declare CPU winner
- **II.** Assuming valid user input, verification of potential moves before making them (otherwise return an appropriate error code according to the project description). This entailed:
  - **a.** Checking for turn and disallowing turn skipping
  - **b.** Confirming that the selected source piece is of the player's color, and if there is a destination piece that it is of the enemy's color
  - **c.** Determining if there exist pieces as specified in the source and destination spots
  - **d.** Verifying that the move is possible for the selected piece in the source spot
    - **i.** If the move places/keeps the current player in check, then it is invalid and thus is reverted
    - **ii.** If the move causes the enemy king to be placed under attack, then a verification is run to determine if there are any possible moves for the enemy to evade check. If there are not, a checkmate is announced and the game is over.
- **III.** Moving pieces on the virtual board correctly, keeping track of captured pieces, and updating the character board array after any moves. Additionally, changing turns when appropriate (if the user attempts to make an invalid move, it is still their turn).

Data Design

As noted in the previous section, there is a requirement in place for having a character array representing the chess board (an 8 by 8 grid of boxes) that updates as a game progresses/resets. The character array is actually 129 characters in length, holding 128 positions for piece types and colors (for example 'B', 'P' for black pawn) and a terminating

newline. Relying on this array of characters for program logic would prove to be tricky and nonintuitive, so instead an alternative logical board was constructed.

A C data struct named *Piece* was created to represent a chess piece, as the name implies. The *Piece* struct contains five member variables:

1. char color – will be set either to the WHITE or BLACK macros 'W' and 'B' respectively
2. char type – will be set to one of the possible macros for piece types
3. char moved – will be incremented from 0 based on the number of moves for a piece
4. char index – is set to the index of the logical board the piece sits on (0 to 63)
5. int alive – is initially 1 for all pieces, and set to 0 if a piece is captured

Next, two *Piece* struct arrays were constructed to represent the collection of pieces of both players – whitePieces[] and blackPieces[]. At the beginning of a game, the arrays are initialized such that they each hold 16 pieces (2 of every piece, except for king and queen, and pawns which are 8) in order starting from the bottom right from the perspective of each player (the a1 or h8 rook). The member variables in each struct *Piece* are set appropriately.

Finally, the logical board is constructed. This is an struct *Piece** (pointer to *Piece* structs) array of size 64. After the two aforementioned *Piece* arrays are constructed, this array is set up such that the first 16 positions point to the corresponding whitePieces[] structs, and the final 16 pieces point to those in blackPieces[] in reverse order such that the final index of the logical board (63) is the first index in blackPieces (the h8 rook), all according to the rules of chess for initial game board setup. Additionally, for all indices in between (the middle 4 rows of the chess board), the pointers in the logical board are set to NULL to indicate empty positions.

Move Validity

In terms of determining the validity of moves, the procedure for the user player adds character array checks to the normal procedure for the CPU.

<u>Movement Validity Verification for User</u>

When a user char buffer arrives in the write() function of our driver, it is verified using the kernel access_ok function against the supplied length. If valid, it is copied over into a kernel buffer, then split into tokens using strsep() (separations by space) and stored in a char** tokenArray[]. For example:

"02 WPe2-e4\n"

Token 1	Token 2
0	W
2	P
	e
	2
	-
	e
	4

To save effort and reduce complexity, we can reject any input that does not meet the requirements for the number of tokens, and the length of the second token. In all cases, there will be at most 2 tokens in a valid command. Additionally, there are only three valid lengths for the second item in the tokenArray (tokenArray[1]). The length can be 7 (a simple move without a capture), 10 (a move with a capture OR transformation), or 13 (a move with a capture AND a transformation).

If the number of tokens and the length of the second token are valid, there are a series of checks that are performed based on the length of the second token. In all cases, the first position is checked against being a 'W' or 'B', and the second against being a valid chess piece (for example an 'R' for rook). The remaining positions aside from the dash "-" are casted to integers and compared against appropriate ASCII decimal values to ensure that the column and row values are within the range of a to h and 1 to 8, respectively.

For lengths of 10, if an x is appended, the following position for color is checked to be different than the color of the attacking piece (index 0 in the second token), and the piece to be valid and not a king. If a y is appended, the following character which represents color must be the same as index 0, and the following index must contain a valid transformation piece in chess. Additionally, the second index in the second token must represent a pawn ("P"). The check for lengths of 13 basically does the same check as described in the previous sentence, in addition to the checks for length 10 and length 7 (using goto's).

<u>Movement Validity Verification After Character Array Manipulation</u>

For both the player and CPU, after the variables for the moving and destination (if any) pieces are set, the driver performs a number of checks before reflecting the move on the logical board and on the character array that is returned by the "01\n" command.

When the user or CPU attempts to move a piece, they supply the driver with a row and column value (which are both checked for out-of-bounds). Note that in the case of the CPU, it tries all possible moves for each piece of its color until one can be done without place it under check. In both cases, a function *getIndex()* is called, passed both values as parameters, and returns the corresponding index in the logical board:

Logical Board Index = Column + (8 * Row)

Then, using the data structure, we can make many verifications for validity easy. The source/moving piece position can be checked for pointing to NULL, being captured (by inspecting the alive member variable), matching the color specified, and type of the piece. Before we even attempt to look at the destination index, we can confirm whether or not the source position piece can move to the destination, based on type. To do this, a switch statement is put in place that uses the type variable as its condition, and a helper function for observing whether or not the piece can travel to the destination is used for each case (each piece has its own helper function since chess pieces have different movement styles). For pawn movement, we can also check the move member variable to determine if it may move two spots (once a pawn moves from start it cannot move two positions forward).

If movement is possible, the destination index can also be observed to see whether or not it is empty (NULL), and if not inspect to ensure the color is not the same as the source index. To logically perform the move, the destination index's pointer in the logical board can be set to the struct pointed to the source index pointer, and the source pointer can then be set to NULL indicating that the position has been vacated. Member variables of the moving piece such as index and moved are updated. Additionally, if the moving piece has captured the enemy, then the destination's pointer sets its alive variable to 0 beforehand.

After performing the move, a helper function for checking whether the current player is under check is called (we can use the index member variable of the king struct in the white or black pieces arrays to find where it is). If the move places/keeps the player under check, then it is invalid. In the case of the user, "illmove\n" is returned, but for the CPU, another valid move must be found and performed.

Verifying Checks and Checkmate

As for determining a check, the inspection must be done after every move by either player. To do so, after a valid move is performed (the conditions above are passed), the helper function for determining a check is called, but instead the position of the enemy king is passed (using the member variable index of the struct *Piece* for the enemy king in the enemy's pieces array). This helper function basically loops through the friendly pieces array (for example whitePieces) and checks if it is possible to reach the supplied enemy king's location given the friendly piece type. This function also has a switch statement that calls on the helper functions for each piece if their case is called. If any piece can reach the king's position, then for sure the enemy king is under check, but are they checkmated?

To determine this, the driver loops through all the enemy pieces, then tries to move to every possible position on the board for each given piece. For every trial, the program tests whether the enemy king remains under check. If some move releases the move from check, a variable is set and the driver returns "CHECK\n". If the nested loop for all enemy pieces and board positions (0 through 63) completes and no possible move releases the enemy king from check, the driver returns "MATE\n" and ends the game.

Note that there is no need to verify whether the current player is under check, as once they attempt a move, if it does not free them from check, it will be invalid.

Potential Improvements and Alternatives

In addition to implementing extra credit features such as detecting and responding to stalemate situations, I would like to improve and clean my code further, given more time. Although I made a significant improvement (compared to project 2 part 1) in terms of reducing redundant code and adding helper functions which may be called several times, I did not completely eliminate redundancy. For example, I have two switch statements for determining possible moves in different parts of the code that could be made into a function.

In addition, I would like to address the perhaps excessive use of goto's in my code. While helpful in parsing situations where a longer input will still have the components of a shorter

one, I am sure that when it boils down to check and checkmate code that I could have done better, most likely with functions. This would also help make my code cleaner and more readable.