Hafsa Chaudhry

I created a kernel module in my OS image of Debian in my Virtual Box software. This kernel module uses the character device driver to implement a chess engine to play the game of chess. It then creates a bridge between the chess viewer, which is a tool to “help interact with your chess driver and visualize the state of your driver”, and the Virtual Box. To start recreating this kernel chess game, you will need to set up a kernel module like normal. It should include the licensing type, a init and exit function, and should create a dev/chess accordingly. It will need to have a file_operation structure that gives ownership to your module and that allows for the creation of your chess read, write, release and open functions. The file_operation structure will be used as a prototype function for the character driver. Then you will need to create a header file that hold all the general functions of the chessboard and the structure of the chess board itself. The general functions for this chess game module include the functions relating to how to move pieces for both the CPU and the user and check for checkmate as well as the basic functions for the file operations. In addition, you will define global variable names for the chess game that include the color and status of your chess pieces (i.e. black pawn) and the move specification and/or error conditions (i.e. NOGAME, CHECK, OK). Then you will need to create c files that hold all of your header file functions, formally known as the character device driver. You can split them up into multiple different C files or keep them all in one file, depending on your coding preferences. I will not be going into detail about the files themselves but instead of the functionality of the implementation of the chess engine. I will note, however, that this module is written in C, and the C language is not object oriented. Instead, they are object based, meaning simply that this module has objects and methods that can be created and destroyed and be used to communicate and send messages to each other, and “if an object doesn’t know what to do with a message then it can send it to another object”. For the chess character device driver, you create your functions based on your object type such as the game implementations, the cpu moves, the user moves, and the rules for chess themselves.

When implementing the file operations, open and release should be relatively the same as any other module character device, with read and write being the file operation functions that are modified. It is good to note that the read and write functions for this chess engine are “backwards”, meaning, that read is used when the module is sending messages and communicating with the viewer, i.e. when the device is being read by the user space, and write is used when the module is receiving communication and messages from the viewer i.e. the user space is being read by the driver. Therefore, the read function is more trivial, and will be used to send commands such as INVFMT or CHECK to the server, by copying the kernel buffer over to the user buffer. The write function takes care of user command input from the viewer such as 00 W/B to start a new game, 01 to print the current board, 02 to make a valid player move. Finally, 03 to make a CPU move and 04 to end the game. At the start of a new game, you will need to create a void function that takes the board pointer and handles the board set-up. The board will be initialized and you will need a buffer of at least 129 chars (two per square, times sixty-four plus a new space character). A 3D array is used to store the pieces, the first two conditions of the array representing the size of the board, 8 spots for length and 8 spots for height, and the third representing two chars for color and type. The board is eight by eight so you will need a nested if-statement. The inner if condition will be used to set up the players pieces or the cpu pieces in the gameboard. The pieces will have a back row with the non-pawn pieces and a front row with all pawn pieces. To check for valid input, you check if each element of the command being sent to the driver is valid. In other words, you’ll have to go char by char with each character being compared to a character for equivalency with anything unexpecting throwing an exception error. This step is validating grammar. You first check if it’s a correct piece, and if the current position and the position to move to are valid and in bounds and then check if the length of the command is valid and if it is the player’s turn. For each possible command length, you check if the commands sent are valid per position of the command string being sent. If all of these are true, then the chess engine will check if the player is trying to capture the opponents piece or its own and return the appropriate message. It will do the same for promotions and then validate the move as well. Then it will check if the piece it is trying to move to is reachable and based on the chess rules for each piece. For each piece, create a function that checks movement based on chess move. So for a pawn, you have to check if it can kill diagonally forward, and you have to make sure that the pawn can not move backwards and that it is your own pawn being moved and not the opponents. It needs to also check if it is making an invalid diagonal move, or in other words, it is trying to move diagonally to kill when there is nothing to kill. You also need to check if the pawn can move forward by checking if there is another piece blocking it. Knight is the only piece that can jump in chess, to enable this move, you need to find the distance it can travel horizontally and vertically. You need to check if a piece is blocking the move and if the knight is jumping illegally, likewise allow the piece to kill. For the rook, you have to check if it can move horizontally or vertically based on if it is being blocked and return an error. Since some of the pieces move similarly to each other (i.e. the bishop moves diagonally, the rook moves vertically and horizontally while the queen does both), you need to ensure that the piece being moved can correctly move in that fashion. And since all three pieces can move in any distance, you need to check each slot in the path that it can move. You need to also get the position of the slot by using the x and y coordinates for that slot. If a king is being moved, it needs to validate that it is not putting itself into check and that the move is valid as well. Then each piece will check whether or not it is placing the king in check based on their next valid available move. Once the king is in check, to check for mate, the king will see if there is any valid move it can make to get itself out of check when it is the king’s turn. In addition, if a piece is killed then it is removed from the board. The CPU will follow similar directions to ensuring valid moves and kill, capture, promote. But the CPU will move more randomly. However, the CPU will check to see if it can kill based on if it can move and attack with one of its pieces when it is their turn. Each move and attack is checked for validity the same way as the player pieces. Again, the king’s position is implemented to update automatically when necessary. Every time a piece moves from either side, and the move is valid, the old position will be replaced with a ‘*’ until a new piece takes that slot again. Once checkmate is detected on either side, the game is then set to game over.

To check if the king is checked , it checks the knights differently, since knights are the only pieces that can jump, and therefor they are checked in a rotation space around the king. and alongside this , your function should also check every direction for other pieces. Then it moves in that direction until it reaches a piece of edge of the board. Importantly it also checks if that piece is able to reach the king.

To check if the king is checkmate, you will need to temporarily removes the king to avoid conflicts with “check” which is aforementioned, it is important to take away the king in order for the commands to follow through smoothly. Your function should then check every possible destination before adding back the king. Two functions are needed to move both the CPU pieces as well as the player pieces . Also be able to get possible distance from the king in order to gather if check for mate will occur. Once again if checkmate is detected on either side then the game is then over.

References:

http://www.informatrix.ch/informatrix/obc/index.htm

http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/