# CMSC 421
# Project Design Document

[Jeffrey Carino]
[Last Modified: 5/4/20]

1. Introduction

1.1.System Description

This document provides an outline for building a kernel module that handles a game of chess. There will be a kernel module that communicates through the chess viewer program provided and then handles commands passed by the user and updates the game accordingly.

2. Design Considerations

To create the kernel module, place the appropriate headers to place the module into the kernel for use. After all the connections are done, then create all the necessary functions such as open, release, read, write, init, and exit, to handle the module. Then I was thinking about using separate functions to handle all the movements of certain pieces such as the rook, bishop, queen, and knight, which all have very dynamic movements. These functions will alter the board accordingly depending on the move given to it and the function will return a char that gives the letter for an error or success. The write () function will do most of the board modifications and setup, while read () mostly communicates to the user on what is going on, giving appropriate error messages and responses. Must use a three-dimensional array with dimensions 8, 8, and 2 because its an 8x8 board and each element in the board has two characters in it that signify the color of the piece and the type of piece. Must have conditional statements to determine which command was passed and act depending on that command. The 01 command would just return the board with modifications, if any. 00 W/B will create a board and place the white places or black pieces on the bottom of the board facing the user depending on the color the user chooses. The 02 command will take a move and break it down from which

piece is going to move, the start position, and end position. There are some moves that promote the pawn to any piece, so there must be a portion of code to account for that. The 03 command will be inserted by the user when it is the computers turn to make a move and make a valid move. The 04 command will declare the CPU the winner and cancel the current game, but the 01 command should still be able to print the most recent game. I will go into further detail on the implementation I used as this is just more of a design template.

## 3. System Design

### 3.1. Pre-requisites

Create a three-dimensional array board and have the sizes of each go in this order: 8, 8, 2. This will give you a nice 8 by 8 board that has two characters inside each element that describe the color and the type of piece. Then, create a variable that signifies whether it is the CPUs turn or the users turn. Next, declare a char array that holds the commands that are being passed in for global use because the read function needs access to that information as well. There also must be a variable that holds the game state because if there is no game in progress then there can not be a current board to print out until the 00 W/B command is sent in which starts a new game. As stated in the design considerations, usage of the built-in kernel functions must be used such as read, write, release, open, init, and exit are necessary for the kernel module. When you are creating your miscellaneous character device make sure that you adjust the root permissions so that you can use the module without using sudo command. Along with that you also must give it a dynamic minor and access to the fops functions which are built in functions mentioned earlier. In the init and exit function you must register and unregister your device for proper usage.

### 3.2. The Module Write Function

There must be a variable to take in the number of bytes that have been read in order to log it into the kernel. There will be a copy_from_user to put the passed down command list into a local variable so that it is readable in kernel space. There will then be a while loop to iterate the number of bytes in the command given. This function is going to hold all the conditions of the commands. The first command is going to be 00 W/B which will create a new board and make either black or white be the users' pieces. There will be three for loops that will iterate over the board and place a * inside each element to initialize an empty board. Then there will be an if statement that will ask whether a 'W' or 'B' was passed. If all three necessary things needed for the command are inputted correctly then change the is_game variable to true because a game has started. If a 'W' is passed that means the user decided to play as white and goes first. Therefore, you must set the turn to player one and put all the white chess pieces on the near side so that the user can see all their pieces. If a 'B' is passed, then the turn will be set to 2 because the computer gets the first turn and the board will insert black pieces on the near side. Each of these if statements will have their own for loops for putting the white and black pieces in their proper place. Then return the bytes that were read, which should be stored in a variable.

For the 01 command, it is going to be simple. If there is a game in progress, then there is a game board to return. If there is a game, then return the number of bytes that were read in. If there is no game in progress, then return the bytes needed for the error message.

For the 02 command, there is going to be a large command passed in, so I make sure that everything that is passed is valid before trying to make a move. I implemented functions that will make the program look neater and work more efficiently. I implement functions based on the piece so there will be a pawn(), knight(), King(), Queen(), Bishop(), and Rook() that will take in parameters and then execute the move if it is valid or give the appropriate error message if invalid. These functions also hold all the rules for each certain piece and returns an integer; it returns 0 if an error was found and returns 1 if the move was valid.

For the 03 command, it will make the computer make the most basic valid move such as moving each of the pawns one space ahead until you can not anymore then move to the next available piece. If there is a game in progress it will make that basic move but if there is no game in progress, then it will return the number of bytes for the error message that was written.

For the 04 command, the user quits and the CPU wins. If there is a game in progress, then it returns the number of bytes for the response which should be “OK \n.” If there is no game in progress, then it will return the number of bytes needed to write the error message.

Near the end of the write function there will be a return 9 which returns the unknown command error because if none of the above commands are inserted, then a proper command was not inserted.

3.3.The Module Read Function

At the beginning of the read function there will be char* variables that hold the error and success responses for the commands passed through write. If the 00

W/B command is passed, then the read function will read it from the global array that holds all the current commands and give the proper response. If the command was inserted correctly then the response was “OK \n.” For the 01 command if there is no game in progress, then there is no board to print, so it returns the NOGAME error message. If there is a game in progress, then there will be three for loops to iterate over the board and send every element back to the user using put_user for each element in the board. It also returns the number of bytes that were written which should be 129 including the newline character. The 02 command gives the proper response depending on what the write functions said. If the write function tells read that the move is invalid, then it will throw an invalid move error, but if the move is valid, then read will return the number of bytes from the “OK\n” response. The rest of the commands will do this as well because the read function essentially just communicates to the user space program what is going on.