Dave Benton

# Project 3: Reversi

## Description of Project

For this project we are required to create a linux kernel module that can be used to play a game of Reversi against a CPU. The module will need to implement a virtual character device that will be handled by the user using normal file read and write operations. This module works on the basis that the user will open a file descriptor to the module's /dev entry, writes commands to that descriptor using the write system call and reads the responses to those with the read system call. The module must also keep track of the game board, implement logic to read and parse these commands, along with checking their validity, and decide when the game is over.

## Description of a Character Device

Character devices have major and minor device numbers, traditionally major numbers identify which driver to handle the device and minor numbers identify which instance of the device is being managed. A module is any bit of runtime loaded kernel code, and a device driver is a module that controls access to a device. Drivers typically have three parts: Registration, Interrupt Handling, and User Space API. Registration is responsible for notifying the core kernel of driver ID, Interrupt handling handles hardware interrupts generated by the device, and User Space API lets programs interact with the driver. In order to handle User Space API we need to create device nodes as well. Creating device nodes involves registering callbacks which is done within a module's probe. The driver also must create a struct 'file_operations' which sets it's fields to the desired callbacks. A miscellaneous device is a simpler interface for smaller drivers.They are represented as a struct 'miscdevice' and need a struct 'file_operations'. Since a driver

is the interface between an application and hardware, we often have to access user-space data. Since accessing it cannot be done by dereferencing a user-space pointer, which can lead to incorrect behavior, access to user space data is done by calling macros/functions. These macros consist of copy_to_user() and copy_from_user(). We will also need to implement foo_read() and foo_write().

**Implemented Functions**

Other than the functions required to operate the device (init, exit, open, release, read, write) I had 8 functions. I will just go about explaining their functionalities in a downward order starting with the function that calls them. User commands are copied (using __copy_from_user) from the user space buffer into a char array I titled 'cmd' inside of the module's write function. From there I evaluate their command, in order to properly lock players out of running certain commands when a game is not in progress I created a global boolean that holds true after a player has run the '00 X/O' command, this is turned to false when a game ends as well. When the user enters a valid '00 X/O' command, the write function calls the function I created call new_game. I will go further in depth on what that does later, for now I'm detailing write. If the user enters 0 0 but invalid things after, they will be met with an 'INVFMT' error. If the user enters the command '01', the write function copies the game board and it's size into two globals that are then copied back to the user within the read function. These two commands are able to be ran by the user without a game in progress, for the rest the global bool for a game in progress must be true, if they attempt to run any of the following commands with a game not in progress they receive the 'NOGAME' error message. If the user enters '02', but something incorrect after they are met with another 'INVFMT' error, otherwise the place_move function is called, and it is passed indexes 3 and 5 of cmd. If the user enters '03', the cpu_move function is called which

handles making moves for the cpu. Finally, if the user enters ‘04’ the user_pass function is called which determines if the user has any valid moves left. Otherwise, anything else entered and the user receives a ‘UNKCMD’ error message.

**new_game(char piece)**

This function is quite simple. I created two global chars, one to hold the piece the user selected (either X or O supplied with the ‘00’ command, and passed to this function as ‘piece’). On top of this, I also created a global bool to hold whether it is the user’s turn or not. Since X always goes first, if the user selects X the bool is set to true and the user piece is set to X, otherwise the computer’s piece is set to X and the bool is false. After this the bool that holds if a game is in progress is set to true.

**place_move(char col, char row)**

This function gets a little messy, it is responsible for calling several other functions. I first initialze all local variables needed. Since commands passed in by the user are char arrays, I needed to converted their moves to ints. The chars holding their moves are supplied to this function as col and row. What I do is create several int variables: col2, row2, moveLoc, ret1, ret2, and i. Then, I created two char arrays with a size of two, titled col3 and row3. Seems a bit excessive but this was the way I decided to convert. I also create a char array called moves with a size of 9, I fill it in with default values but could be left empty. What I do from here is set col3[0] to col, and then col3[1] to ‘\n’, and then the same for row3. This is to null terminate them so I can pass them to kstrtoint, which converts strings to ints. Since kstrtoint returns a 0 upon success, I supplied row3/col3 as the first parameter, then 0 as the base for conversion and then finally a referenced row2/col2 to hold the new int. I set the returns of these converts to ret1 and ret2 and then compare those to 0, if they are not 0 the user is met with a ‘INVFMT’ error since

they did not enter valid ints, this is done the same as passing the board by copying the error message and it's size into the globals copied back to the user. After this I then convert the two indexes into a single index so that I can use it on the game board which is a 1d array with a size of 67, and the first 64 being the board. This is done with the statement 'moveLoc = (8 * row2 + col2)'. I now check to make sure that it is the user's turn, if not they are met with a 'OOT' error message. After this, I then check to make sure 'board[moveLoc]' is an empty piece signalled with '-'. If not, they are met with an 'ILLMOVE' error message. I now reset the moves array by filling index 0 with '-' and the rest with '0'. This is a sort of pseudo-boolean method. This array gets passed to my next function called valid_move and is populated there. I will explain how that works and is used when describing that function. If it comes back as a valid move, the user piece is then set at moveLoc on the board. Index 65 on the board holds the piece of whoever has the next move, at this point it becomes the CPUs turn and gets set to their piece, along with the user move bool being set to false. I know call my function designed to flip the pieces after a move called flip_pieces. After this is done, a win condition is checked using the check_winner function. If this function returns false, the user receives a "OK" message and then returns. If met with a true, it just returns.

**valid_move(int col, int row, char piece, char rets[])**

Here, col and row are self explanatory, they are the col and row of the move to check. Piece holds whoever we are checking a move for, so if for the user it will be the user's piece. Rets[] is the array we created in place_move. First thing I do inside the function is create an int array called dirs with a size of 8, with every index set to 0. I then create a col2 and row2 and set them equal to col and row, this is done so I always have access to the original ints supplied to the function. After this a create a char called oppPiece (holds the opposite piece supplied with

piece). This is given X as the default, but if piece == X it is set to O. I then again check to make sure the board at col, row is a '-' just to be sure. After this I then check every direction around col, row for an opposite piece. If an opposite piece is found, an index in dirs is set to 0. For example row-1 (up) is index 0 of dirs. I do this in order to avoid checking all the way in every direction. What happens after this is a series of while loops. There is 8, one for each direction and they operate while their respective index of dirs is equal to 1. I will only detail one while loop since it is relatively the same for the rest. Inside the while I first move row2/col2 in the right direction. So for dirs[0] I do row2 = row2-1. I now check to make sure row2 is still in bounds and if it is equal to piece. If it is, I set index 0 of rets to piece, this way when the function terminates, if that index is equal to piece, we found a valid move. I then set index 1 of rets (very similar to dirs, just moved one to the right) equal to 0, saying that direction needs to be flipped. From here I set row2 back to row and dirs[0] to 0 to end the loop. Within the while I have another if statement that handles if the row went out of bounds (meaning we never found another piece) or if we hit a '-' on the board. Within the if I set row2 back to row and dirs[0] to 0 to end the while loop.

**flip_pieces(int col, int row, char piece, char *moves)**

This function is always identical to the final section of valid_move. I tried to combine them into one and failed miserably, so here we are. Like valid_move we create a col2/row2 in order to keep the original value, and it follows the same scheme for oppPiece as well. From here, the while loops are set up the same as well. Since moves is a pointer I simply just advance it to check. I will go over the first while (which lines up with the while discussed above in valid_move). The condition is while(*(moves+1) == '1'), and like valid_move the first thing I do is subtract 1 from row2 in order to move to the next spot. I then check if it is in bounds and if it

equals an oppPiece. Since we know this direction needs to be flipped, if that is the case it flips that piece to piece. Then the other if checks if the next spot (row2 -1) equals piece, if so it resets row2 to row and sets *(moves+1) to '0' to end the loop.

**cpu_move(void)**

This function is very simple. Similar to place_move I create ints: col, row, moveLoc, and i. I then create a char array called moves, which is identical to that in place_move. I also create a bool called win to hold the return of check_winner after a move is made. After this, first thing I do is check to make sure user move is false, if not that user receives a 'OOT' error message. If it is false, I then create 2 for loops, each starting at 0 and operating less than 8. The outer loop runs on col, and the inner on row. I then set moveLoc equal to (8 * row + col) in order to get the board index. I yet again check to make sure the board at that spot is '-', this saves some executions from running completely and absolutely makes sure the spot is at least empty. From here we reset moves the same as in place_move and then called valid_move. If moves[0] is equal to the cpu's piece we then know we had a valid move. After this we place it, set the user's move to true, set index 65 on the board equal to the user's piece and then call flip_pieces. After flipping is done we check for a winner the same as in place_move.

**user_pass(void)**

This is extremely similar to cpu_move. Same local variables and for loops. All this does is check every empty spot on the board to see if it is valid for the user. If any are found, the user receives a 'ILLMOVE' error message and returns. If not then the user move is set to false, index 65 of the board set to the CPU, and then it checks for any winners.

**check_winner(void)**

Another simple function here, I create 3 local ints, one to hold the number of user pieces, one to hold the number of CPU pieces and then i for a for loop. I first call check_winner_search, if this returns true then I use a for loop to check every index on the board. If the index equals a user piece, increment user piece count and same for the cpu. After this I then compare those tallies. If user has more, user receives a 'WIN" message, if the cpu has more the user receives a 'LOSE' message and if tied, a 'TIE' message. After this we set the game bool to false to signal no game in progress and return true, indicating there was a win/tie. If check_winner_search does not return true, just return false.

**check_winner_search(void)**

Here, I create a char array Moves, used for valid_moves. Exactly the same as previous uses. I then create a char array movesRet which holds in index 0 if any valid user moves were found and at index 1 if any CPU moves were found. I also create 2 chars, cpuMove and userMove which will hold the return values of valid_move. I also have 3 ints: col, row and j. Col and row are used the same as in cpu_move and user_pass. I create two for loops to address every index of the board, and then call valid_move for that index, checking if there is a valid user or cpu move. This will be returned to userMove and cpuMove. If userMove is equal to the user's piece, movesRet[0] is set to '1' and same goes for cpuMove and movesRet[1]. I now realize I could have made this simpler but am running low on time so it will stay this way. After this I then check if movesRet[0] and [1] are equal to '0', if so then return true. Else return false.

**How my final differs from my prelim and what I had trouble with**

Well upon beginning the module, I quickly realized my prelim would not work very well. I first had trouble using a genuine char device, it just would not register. I'm still not sure what I was doing wrong, I followed every guide I could. Upon discovering that a misc device is a type

of char device, I went that route and it went very smoothly. When it became time to place and flip moves, I figured the code I had written for my 201 Othello project would work well. It did not, not even in the slightest. The added complexity of user's and cpu's placing moves and flipping dynamically threw that completely out the window. I definitely struggled greatly with determining valid moves and flipping them. Looking at it now, it's quite simple but at the time I was quite confused. This is when I decided to break it down in the manner it currently is. I started with placing moves, and then using that to flip them. As you can see, my functions are not even similarly named or remotely close in length to those detailed in my prelim. I absolutely should have done more prep for that.

**Helpful Sources:**

**Misc Device:**

gist.github.com/17twenty/6313566

**Modules in general:**

https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html

**Kstrtoint:**

kernel.org/doc/htmldocs/kernel-api/API-kstrtoint.html

**Misc:**

Slides linked on the project document, as well as the wiki article on othello