Darren Mirano

CMSC 421

Project 3 – Final Documentation

How my preliminary differed from my final product:

My preliminary design didn't change that much from my final design. There was only one change that differed. Originally in my preliminary design I thought it would be a good idea to get a list of all the valid moves for the current player, and then compare that to the move passed. However, on second glance this seemed like an inefficient way of validating moves. So, I changed my design to only validate the move the user passed. This saves a lot of computation time since it does not have to find every valid move for the player.

Other than that, my design stayed roughly the same. Obviously getting a list of all the valid moves and validating one move are two completely different algorithms, but I will go over that later in the document.

Original steps when starting the project

When I first started this project, I needed to understand how kernel modules function. Because of this I spent a couple of hours scouring the internet researching on how to properly create and open kernel modules. The slides that were given to us also really helped me understand. The slides also gave a clue that we are supposed to use miscdevices which was a good hint.

Once I figured out that our character device needed to be a miscdevice, this led me on the path of miscdevice_create(). I got the device compiling and working correctly, but it never showed up in my /dev directory. Hours of researching led me to believe that we need to use the "mknod" command to manually create the entry in /dev. I asked in the 421 discord and this was not the case. I was informed that the "insmod" command should be creating the /dev directory automatically. More hours of research later, I came across this miscdevice_create() function. This was the key. This is what makes the character device automatically in /dev. It was a struggle to find, but the linux cross-reference really helped me here .

Once I got that down I figured I was done with the whole character device part of the project, but there was still one this I had to figure out. I was getting the incorrect permissions. When I created my character device I had "crw-------" in my "ls -l /dev/reversi". It came to my attention that these are the permissions, and it should be "crw-rw-rw-". This didn't take me as long as finding miscdevice_create(), but I still spent quite a bit of time finding what had to be done. In our reversi miscdevice, we had to add a ".mode = 0666". This is what gives the correct permissions. And with that done, I could finally focus on programing the game reversi.

Moving on from the character device setup:

Once I had the character device set up and ready to go, I could finally start coding reversi. I decided to study the driver file given to us, since that would control how we operate our device. This driver file that was given to us was a really big help since it meant we did not have to code a driver file ourselves. I had to create an output() function so my reversi.c and the reversi-program.c could communicate. After that, it was just a matter of coding the game.

It is worth noting that figuring out this part took me quite a bit. Having my reversi.c and the reversi-program.c communicate was a big problem for me. I was having trouble figuring out how to get the command that the user passed in. This was a big roadblock as I could not proceed in the project without getting the command being passed. Eventually, after hours of trying to get the command, I decided to take a break from that part of the project. I then just coded reversi in userspace.

<u>Coding reversi:</u>

My first steps to coding reversi was to know how to set up the board. The board is constant, so we can simply create an 8x8 2d char array. This was easy, I had a for loop iterate through my array and fill the board with “-“ to represent the empty spaces. After that there is also starting pieces that must be placed. So, I just overwrote the “-“ with the correct pieces after I populated the board. That is command 00 down. Printing the board is also easy. It is simply just a nested for loop. Loop over the 2d array and print what is at each index. Command 01 is done.

The rest of the commands would be the hard part of programming reversi. The first thing I focused on was validating a move, so I studied a standard reversi board. I figured out that each valid move must be at an empty square, and that the 8 surrounding square must have at least one opponent’s piece. That’s not all though, once you find an opponent’s piece you would then have to continue in that direction until you find one of your own pieces. If you don’t find one of your pieces in that direction, that move is not valid. This is the criteria for a valid move in reversi.

Once you find a valid move, you would then have to flip all the opponent’s pieces that were sandwiched in between the current player’s pieces. The turn is then switched when a move is made, and if the current player does not have any valid moves, they must skip their turn. The game follows this formula until it ends, meaning there are no more empty spots on the board. So, after each piece is placed you would have to check if the game has ended. Here are the helper functions I added to help me achieve all of this.

<u>check_adj_cells(row, col, piece, validate)</u>

This functions purpose is to check the surrounding 8 cells around board[row][col]. If an opponent’s piece was found at any of the cells, I would then call my next helper function check_and_flip(). Piece is the current players piece, I passed this in so I could know what the opponent’s piece is. Validate is used for checking if there are any valid moves for the current player. Since I only want to check if there are any valid moves, I use this parameter as sort of a flag to let the function know that I do not actually want to make the move.

This function also has a lot of edge cases. These being the four corners and the four edge rows. (top, bottom, left, right) For example, you would not want to check the bottom adjacent cells of a piece trying to be placed in the bottom row, as there are no spots to check and would throw an out of bounds error. This had to be done with all eight edge cases.

check_and_flip(row, col, opp_row, opp_col, piece, validate)

This function branches off check_adj_cells(). In check adj_cells_() if I find an opposing piece, I pass the empty space index and the index the opposing piece was found. I pass these two because with these two indexes I can determine which direction to look. So, for example if the rows are the different, but the columns are the same, I know I'm moving either up or down. Knowing these two indexes let me know the exact cardinal direction I would need to check. So, I would iterate in that direction, and if I find the current player's piece, I now know that move is valid. Again, validate is used for checking if there are any valid moves, so my validate flag can prevent the move from being played.

check_game_end(void)

This function checks if the game has ended. It simply iterates through the 2d array and if it finds an empty space the game has not ended. If no empty spaces are found, the game has ended. This function would be run whenever a move was made.

count_pieces(void)

This function runs when the game has ended. It just iterates through the array and counts the pieces to determines a winner.

I used all these helper function to code my placing a piece command. (02)

How I handled making the bot move

Making the bot move was relatively simple. I just iterated through the array and ran check_adj_cells() on each index that was empty. So that mean the bot always made the first move that was available. If it didn't find a move, that means that the bot has no moves and must skip their turn.

How I handled skipping a turn

This is where the parameter validate comes into play. In my check_adj_cells() and check_and_flip() I pass in an int parameter named validate. Essentially what this does is if validate is 0 make the move, and if validate is 1 don't make the move. This allows me to check if there are any valid moves for the current player without actually making the move.

check_for_valid_moves(void)

This function just iterates through the array and calls check_adj_cells() with a validate of value 1, which means no moves would be made.

Locking

Locking was relatively simple, well simple because we had to learn about it for project 2. And this locking is far simpler compared to the mailbox locking. I just used a write lock whenever I called check_adj_cells() (since this also goes into check_and_flip). I also called a read lock whenever I ran check_game_end(), count_pieces() and check_for_valid_moves(). I only locked places where the game board was being read and written to.

Places where I struggled

As explained before, the first place I started to struggle was reading in the commands that the user entered. I eventually figured this out though, I figured it would have to do with the ubuf parameter being passed. So, I just did a copy_from_user() on that and it gave me the command the user entered.

The next place I really struggled was validating moves. At first, I was super confused on how validating moves worked, which is probably due to the fact that I just dove first head in, without fully understanding the core mechanics of reversi. So, I then decided to study the reversi rules and first figured out what made a valid move. Once I figured out how a move was valid, I could then code around that.

One last place where I struggled was when making the moves. At first, I had all my check_and_flips() in an if-else if statement, but this is completely wrong. Since multiple rows can be affected by one piece, each check_and_flip() must be in their own if statement. This allows the program to detect if multiple rows and columns could be affected. This was a big problem when I first started testing my game out, but once I figured out the concept as to why, it was a simple fix.

Code estimate

I was spot on with my code estimates. I guessed that the length of my program would be around 750 – 1000 lines of code and I clocked in at about 975. I was also right in guessing that the functions used to validate moves are going to be the largest functions of the whole project.

What I could’ve done better

Overall, I am pleased with how my project turned out. Looking back there is one way I could’ve improved my functions to reduce the lines of code. In my check_adj_cells() I have an if-else if for each edge case, meaning I call check_and_flip() in each of those statements. I could’ve determined where the piece was trying to be placed, and then create flags based on what edge condition they would be. Meaning I would not have to call check_and_flip() so many times. I decided not to implement this because it doesn’t affect run times. It would simply just make the code look more condensed; however, I liked the look of each edge case being in their own condition. This made it easier for me to implement the game and allowed me to be more organized when debugging.

Reflection

This had to be my favorite project of the semester by far. My dream is to create video games for a living, and this was a fun start. I thoroughly enjoyed coding reversi and it wasn’t even that hard. (coding the game, not the whole project) I coded reversi in userspace in about 10 hours, and I loved every second of it. Even the debugging was fun. Also getting beaten by the bot who literally just chooses the very first move available gave me a chuckle. This was a really fun project to end the semester on!