Darren Mirano

CMSC 421

Project 3 – Preliminary Document

A basic description of the project itself and what you are expected to produce.

This project has tasked us to create a kernel module that runs the game Reversi. This will be done in a clean copy of the 5.5.0 kernel. We are expected to implement the game from scratch, with a bot playing as the other player. (since it requires two people to play) This module will have 5 commands that are used to play the game.

- 00 X/O – Starts a new game with the player's choice of pieces.
- 01 – Returns the current state of the board
- 02 C R – Attempts to place a piece for the current player at column C and row R
- 03 – Computer attempts to make a move
- 04 – Pass turn if there are no valid moves for the current player

Another note: The bot will simply play a move at random. So, if the bot has at least one valid move it should always take it's turn. The game should also always be checking whether the it has ended or not, and who's turn it is.

A brief description of what a character device is in the Linux kernel, how they interact with user-space, and the required functions to implement.

A character device in the Linux kernel is what let's us communicate with a module, which in our case would be our Reversi module. The required functions to implement the character device is located within the /dev directory. And the way the device interacts with the user-space is through the syscalls read() and write(). Data is read/written from the user-space to the character device which in turn allows communication to the module via the Linux kernel. Character devices are used for audio, video and input devices (keyboard/mouse), while its counterpart block devices focus on storage devices.

I just started working on the project, and character devices were hard to understand at first, but I think I got it down. Essentially in the broadest terms, we are implementing a character device that can run the game reversi. And this device would use the sample program given to us to run it, and the device directly speaks to the kernel. This is my understanding of the overall framework of the project. Once I get the character device working and I can code how the game works, it should be smooth sailing from there. I believe that the user will use the sample code given to us interact with the driver and send us the command via the device write function.

Ideas for how to implement the required algorithms that will be used to accomplish your device driver's task.

Here is a list of the algorithms and my approach for each command.

- 00 – Starts new game
  - Reset game board
  - Add in the starting pieces in the correct starting places
  - Make sure that the argument passed is the player’s piece
- 01 – Print out board
  - Should probably be passed in the game board as an argument
  - Loop over the array and print out which piece occupies that space
- 02 – Make a move
  - Check if it is the player’s turn
  - Check if the player has any valid moves
  - Get a list of valid moves by the player
  - Check if the move passed is valid
  - If valid make the move, appropriate return on error
  - If the move is made, check if the game has ended
- 03 – Bot makes a move
  - Check if it is the bot’s turn
  - Check if the bot has any valid moves
  - Get a list of valid moves by the bot
  - If the bot has a valid move, always use the 0 index of the list since it doesn’t matter which move the bot actually makes
  - If the move is made, check if the game has ended.

<u>Ideas of how to store the data required to accomplish your driver's task.</u>

I am thinking about storing the data in an array. Since the game board itself is constant (8x8), I know exactly how much space I would need. And since we only need to store what piece is in that place (X, O, or -) it can just be an array of chars. This would avoid the pain of allocating memory since we had to do that for every node in project 2. One allocation is needed when the device is started, and one free is needed when the device is stopped, plain and simple.

<u>A rough estimate of how many lines of code it will take to implement the driver, divided up by various functions you will be required to implement.</u>

My guess would be around 500-750 lines of code overall. Starting a new game (00) shouldn’t be that big, my guess would be around 50-100 lines. Printing (01) shouldn’t be that bad either, guess of around 50-100 lines of code as well. Making a move (02) would probably be the biggest function to code, guess would be about 250-300 lines since this also has to handle valid moves as well as checking if the game has ended or not. Making the computer (03) make a move is also in the same boat as 02, estimate of 200-250 lines of code. And skipping a turn (04) should also be a good chunk of code since you have to validate that the player has no valid moves. My guess would be 150-200 lines of code for this.

All in all, the functions that analyze the game board and make the current player’s moves is what’s going to make up the majority of my code. Especially checking when the game has ended or not. Functions such as printing out and starting a new game would be the smaller parts of the project since these do not rely on user inputs.

References

Oleg_kutkov. (2021, February 05). Simple Linux character device driver. Retrieved April 04, 2021, from https://olegkutkov.me/2018/03/14/simple-linux-character-device-driver/#:~:text=A%20character%20device%20is%20one,by%20byte%2C%20like%20a%20stream.

Character device drivers¶. (n.d.). Retrieved April 04, 2021, from https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html

How do character device or character special files work? (2013, February 01). Retrieved April 04, 2021, from https://unix.stackexchange.com/questions/37829/how-do-character-device-or-character-special-files-work