Project 3: Othello | Reversi

Final Design Document

CMSC 421 Principles of Operating Systems

Caroline Vantiem

Table of Contents

Setting Up the Character Device

In this project, a virtual device drive in the form of Linux a loadable kernel module was created to simulate the gameplay of Reversi. The loadable kernel module implemented a virtual character device to perform specific tasks related to Reversi such as game logic and game functionality. For this project, the first setup was to create a character device driver. Specifically, a miscellaneous driver was created. Misc drivers are special and simple character drivers. The miscellaneous driver was chosen because the major and minor number are allocated automatically which will not interfere with any other devices on a system. It is also a simple driver with a boilerplate for setup which made setting up the device easier. The miscellaneous driver was created using the struct miscdevice and uses file operations. The miscellaneous device has a major number 10 and the kernel dynamically assigns a minor number. In the miscdevice, the name is reversi, using file operations, and permissions were set to required read/write access via mode. For the device file operations, open, read, write, and release are included.

Reversi Game Setup

For an initial setup of the game, driver, and game logic, global variables were used as well as static functions. The global variables include things like the cpu/user and player 2 indicators, whose turn it is, whether the game has started or is over, and which mode the game is being played in (cpu/two player). These globals were used to be accessible in all functions in the module. There are also globals for the error codes, arrays, and many constants for easibility. For the next setup, the filer operation functions were implemented. The open and release functions open the driver and release it once it’s done. The write() function uses kmalloc to allocate memory for a buffer pointer which stores the users commands. The function uses copy_from_user to get the data passed in and checks its validity. The read() function handles all other game logic which will be discussed a bit later. The read() function checks the commands for validity, runs the game logic such as checking if there’s a valid move, whether the game is over and who has won. As well as the file operation functions I also have helper functions for the game logic.

Reversi Game Logic

The first helper function is check_command() which checks the validity of the commands passed in by the user. This basically checks whether the format is correct, meaning no trailing or leading white space, or no extra numbers. It also checks if the command is one of the valid commands of 00-07 (extra credit). The second helper function is setup_reversi(), this function basically sets up the game which populates the array for the game which is stored in a 2D array. It sets which user and CPU characters the user wants to play as and sets the turn and game boolean. This function sets everything up when the 00 X/O command is called. The next helper function is populate_gameboard(), this just populates a 1D array for printing when the 01 command is called. This function populates the array with the actual game board with any necessary extra characters. The next helper function is end_game() this checks to see whether the game has ended by adding up the characters on the board or checking to see if there are no valid moves left. If either of these conditions are true it then tallies to see who has won the game. It keeps track of which piece the user is playing as so when the function is done it can send back the correct code of WIN, LOSE, or TIE. The next helper function is check_move(), this function checks whether there are any valid moves for either the CPU, user 1, or user 2. The last helper function is setup_move which sets up the move for either player. This checks whether to see the move is at an edge, corner, or an inner move. The way the game checks if a move is valid, is check whether the move is at a corner, edge or inner move and then checks the eight surrounding squares of the spot. If there is an opposite char next to it then it will then proceed to check if the move is legal, basically seeing if there is a char at the end of the line to flip. All of the helper functions can be used on any player and have a wide range of functionality. They are also used in the extra credit attempt. The dir_#() functions check respectively the directions next to the square. The directions are, diagonal up left, up, diagonal up right, right, diagonal down right, down, diagonal down left, and left. These check the directions of each square for a valid move. The game logic is implemented through all of these functions. If any of the functions return an error code then the corresponding code is returned using copy_to_user. Once the game is over, the pieces are counted and whether the user has won or not is sent back to the read() function. For the rules of the game, whoever chooses X always goes first, the initial setup of the board is in the correct OX XO pattern and for the two player reversi, the second player always goes second and is the white piece.

Algorithm & Data Structure Implementation

For the data structures, the board game is stored in a 2D array. The arrays for checking whether the user or CPU has a valid move are also stored in 2D arrays. This makes indexing the squares very easy and makes checking spaces easy because the arrays are by [Row][Column] in the array. The algorithm for the game logic is to check surrounding squares for a valid move, make the move and then go to the next turn. The game also checks whether the game is over and then presents the winner. The game shows error codes if the game is over and the user tries to move, or if the game hasn't started the user inputs a valid command. All error codes are shown with their own instance of when it should be shown, meaning if the user enters an invalid command it will print that, or if the user tries to move outside of their turn it will also display that. The feasibility of the project was made so that my helper functions could implement my algorithms no matter whose turn it is, that way no function or code had to be duplicated too many times. The efficiency of the code is greatly improved because there are helper functions which help facilitate game play and game logic. Without the helper functions a lot of code would have to be duplicated. The efficiency of the code was greatly improved for the extra credit and was applied to the extra credit as well.

Changes from Preliminary Design Document

Some changes I made were adding more functions which I hadn't thought of during the preliminary design, adding globals into my implementation and adding all of my helper functions. Other changes I made included adding more arrays into my implementation, but I still stuck with using the array data structure. Also the lines of code were changed from about 600 to roughly 1240. I added a new error code called, PASS\n, which is displayed when the cpu's turn cannot make a valid move. It displays PASS and then sets the turn to the user. Nothing else changed much from the implementation I proposed in the preliminary design document.

Extra Credit

I implemented the two player extra credit. This works just like the regular project and works for player 1 and player 2. The implementation for this was used with the help of the helper functions and for the 06 command, it works just like the 02 command. The extra credit was implemented fully. Just like the regular project guidelines, the command for 05 creates a game for two players, the 06 command takes in a column and row and the 07 command checks if the player two can pass their turn or not. For scoring this mode, the same algorithm and implementation was used. At the end the pieces are tallied up and if player 1 wins then it displays the correct code.

Development Issues & Shortcomings

The shortcomings I had included starting the project and learning about character and miscellaneous device drivers. I had to research a ton to learn how to set these up, change permissions and load and unload the kernel module. Other shortcomings I had were trying to implement the game logic, I had to do a bunch of tests to make sure the squares were checked correctly and that the pieces were being flipped correctly and the turn was changing and that the winner was being tallied correctly. I also had some issues with correctly storing the board and checking for valid moves efficiently. I had code that worked but it wasn’t efficient enough so I went back to improve the helper functions to be able to work with any turn and any players move.

References

1.1. What is a kernel module? (n.d.). Retrieved April 06, 2021, from https://linux.die.net/lkmpg/x40.html

AprioIT. (2020, August 06). Linux device DRIVERS: Tutorial for Linux driver development. Retrieved April 06, 2021, from https://www.apriorit.com/dev-blog/195-simple-driver-for-linux-os

Free_hatfree_hat 1333 bronze badges, & Meuhmeuh 41.5k11 gold badge4040 silver badges8585 bronze badges. (1967, August 01). Emulating a character device from userspace. Retrieved April 06, 2021, from https://unix.stackexchange.com/questions/477117/emulating-a-character-device-from-userspace

Staff, E. (2012, November 19). Device drivers in user space. Retrieved April 06, 2021, from https://www.embedded.com/device-drivers-in-user-space/

Writing a Linux kernel module - Part 2: A character device. (2015, April 25). Retrieved April 06, 2021, from http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/

Principles of operating systems. (n.d.). Retrieved April 06, 2021, from https://www.csee.umbc.edu/courses/undergraduate/421/spring21/project3.html