Final Design Document
Muhammed Muktar
CMSC 421: Project 3
April 25, 2021

The main piece of the project is the development of the virtual character device. The project is to implement a character device that will take in commands given in user space and read and write data to the device file that will be /dev/reversi. A character device is a device that functions through transmission of reads and writes of characters. In terms of hardware, the driver would help organize data from user space and kernel space through the device file to work with the device. In the project I will be creating a virtual character device will be able to compute the commands and play the game of Reversi from the module (driver) that is working in kernel space.

The module requires an initialization function for the device and a cleanup. During module initialization, the miscellaneous device will need to be registered using misc_register. In clean up the device will need to be unregistered using misc_deregister (). Next, the main file operations needed to communicate is .read, .write, .open, .release. The operations, similar to:

static struct file_operations fops

.owner = THIS_MODULE, .read = read function, .write = write function, .open = signal open function, .release = signal release function.

(Salzman, Burian, & Pomerantz)

Read Function – Device file is being read from; module will need to copy to user the response message. The message will respond accordingly with respect to the current command after the message is done the buffer will be cleared and the function will return the size copied

Write function – Device file is being written to; module will need to copy from user the command that is given from user space. The command given will be used in runCommand() to be parsed and executed the command is valid. The function will return the size copied

Open function – Device file is being opened; module will need to keep track of device being opened to specify occupied status.

Release function – Device file is being closed; module will need to update status of device of being unoccupied.

For the actual game, I will split the game into different functions. The functions planned include runCommand, printBoard, updateBoard, computerRun, checkMove, checkPossibleMove, createBoard and checkWin.

void runCommand(void) – After device file has been written to command will be passed to this function. Command will be parsed in this function and complete the command specified.

Command Description:

00 X/O: if the command is found the player piece will be updated, the board will be created with createBoard() and the game will begin. If no command is found message will respond accordingly.

01: if the command is found printBoard() will be called to update the message that will be receive when the character device is read.

02 R C: if the command is found, the game must be created with 00 and it must be the player's turn. It will then call checkMove() to make sure the move given is a valid move for the player, if

it is valid, it will then update the board with updateBoard() and check if there is a winner with checkWin(). It then will move on to allow for the computers' turn.

03: if the command is found, the game must be created with 00 and it must be the computer's turn. It will then call computerRun(), which will make sure there is a possible move for the computer with checkPossibleMove(). If there is a move it will loop to randomly generate row and column until that move is valid. The board will then update accordingly with updateBoard() and check if there is a winner with checkWinner(). It then will move on to allow for the player's turn.

04: if the command is found, the game must be created, and it must be the player's turn. It will check if there is a possible move for the player with checkPossibleMove(). If there is no possible move the player's turn will be skipped and the computer can play, else the player must go.

void printBoard(void) – runCommnad will call print board that will turn the board into a character string, stored as the message.

void updateBoard(int, int, char) – runCommand will call updateBoard. The function will update the board based on the direction to update the board in. The function checkMove() will determine which direction (up, down, left, right, top right, etc.) the game board will be updated. If the direction needs to be updated the function will convert the computer's pieces within the player's pieces to the player's pieces of the specified direction and vice versa.

void computerRun(void) – The function will first check if there is a possible move for the computer to play. If there is a possible move a random move will the generated and checked with checkMove(), this will continue as long as there is a possible move. The function will then call updateBoard().

int checkMove(int, int, char) – To check if the location is legal. The location specified will be checked. The move is legal if the location specified is empty and there is another player's piece in between the opposing player's piece. The function will then check each direction for an opposing player's piece, if it found it will continue to move in that direction checking if the opposing player's is in that direction. It will end checking that direction if the current player's piece is found, else the location specified is not legal and will return an illegal message. The function will return true if a possible direction is found and false if there is no direction is found.

int checkPossibleMove(char) – the function will run through the entire board checking if that location is a valid location for the specified piece. The function will use checkMove() for each of the locations on the board, if a location is found it will return true else it will return false.

int checkWin(void) – Using checkPossibleMoves() if there are no more possible moves for both the player and the computer, the function will run through the board counting the number of pieces. The function will then update the message specifying if the player won, if the computer won, or if there was a tie.

void createBoard(void) – The function will initialize the board game to all empty ('-') character. It will then set up the middle of the board to the standard layout of reversi.

Data Storage:

For communicating between kernel space and user space I will be using a buffer called bufferStatement. It will store what command is written to the character device and the message that will be read from the character device. The variables will be parsed within runCommand() and will be updated, depending on the required response.

static char bufferStatement[buffer len]

For the game, the game board will be stored in a 2d character string. The pieces of the player and computer in a variable. The game also requires variable to track moves and the current game state. I store the player's turn, game state, and the directions (updateBoard() requires) in variables.

Preliminary Changes and Update:

The biggest change of my preliminary is the use of miscellaneous character device rather than using a regular character device. I decided to change to a miscellaneous device due the simplicity of both the driver we are creating and the device itself. I was able to simplify register the device, automating device node and files (SLR). I also decided to use one buffer with the device to communicate with user space, the command and message will be both passed through one buffer, this decision was just to simplify the look of the code and make everything look uniform. It also allowed be to simplify the look of the file operations read and write. As for the game, not much has changed for the design of the game. The most I change was the use of checkPossibleMove(). I ended up using checkMove() which helped simplify the design of the function, easily checking moves for the given piece. The function checkMove() also allowed me to simplify updateBoard() as checkMove() already indicates if a location of the board is valid for a piece, updateBoard() can use that location without any issues. I also added the createBoard() fuction which just resets or creates a new game board.

In developing the character device, the main issues I was presented was the read() and write() file operations. Copying from and to the user became an issue as I was having issues copying the proper length of bytes within the operations and tracking the size of each command and message, however I was able to easily solve these issues. I simply made sure to check for proper length before copying and I properly parsing and cleared the buffer to track size of the buffer.

Lines of code:

Initialization function; 15 lines.

Clean up; 5 lines.

File Operations:

Read function; 27 lines. Write function; 21 lines. Open function; 11 lines. Release function; 7 lines.

Game Functions:

runCommand: 198 lines. createBoard: 21 lines. printBoard: 35 lines. updateBoard: 170 lines. computerRun: 33 lines. checkMove: 337 lines of code. checkPossibleMove: 17 lines. checkWin: 55 lines.

Total Number of Lines: 1118

References

Rules for Reversi and Othello. (n.d.). Retrieved April 04, 2021, from https://www.mastersofgames.com/rules/reversi-othello-rules.htm

Salzman, P. J., Burian, M., & Pomerantz, O. (2005, December 31). 4.1. character device drivers. Retrieved April 04, 2021, from https://linux.die.net/lkmpg/x569.html

Tang, J. (n.d.). Lecture 18: Device Drivers. Retrieved April 04, 2021, from https://www.csee.umbc.edu/~jtang/archives/cs421.f19/lectures/L18DeviceDrivers.pdf

SLR. (n.d.). Misc device driver - Linux device driver Tutorial PART 32. Retrieved April 28, 2021, from https://embetronicx.com/tutorials/linux/device-drivers/misc-device-driver/