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().

Ideas For Implementing Required Algorithms

The majority of my ideas for the algorithms will be based on my othello project from CMSC201. Since the board will always start the same for this project, I do not need to worry about opening up a board file. So from that what I guess is left would be to initialize the board, and I would call that function create_board(), which I plan to be a 2d array, this will correspond with the command 00 X/O. This function will store the board in a global in order to maintain its state, this will also allow me to reset it anytime by calling create_board. I will also need a function I will call print_board(), which will be used for the command ‘01’, responsible for displaying the board, and just print across each row in one line. The function place_piece will correspond with the 02 C R command, which will check that spot in the array, if taken or nonexistent it will return the correct error, if not it will place the piece. I will create a place_cpu function for the 03 command. This will generate a random coordinate and continue to do so until it has a valid move. I may decide to just keep a list of valid moves and pick one from it. The function pass() will handle the 04 command, which I believe I will use a boolean to control who’s turn it is, if there is no valid move, it will change the boolean to allow the cpu to move and respond with “OK” as 02 would.

Estimate of lines of code

create_board() should take roughly 3-5 lines or so. First to create the 2d array, then to set the player and cpu's pieces.

print_board() will roughly take 5 lines, a nested for loop will print the array on one line and then 2 other lines will handle the display who has the next move and the new line.

place_piece() will take roughly 10-20. First to check if the player's move is valid, which will require checking the spaces around it, if valid then placing it. It will then need to check if anyone has won. If an invalid move it needs to return the correct error .

pass() will take roughly 10-20 lines, first to check for any valid user moves. If one exists it needs to return an error, if one does not it will need to print OK and allow the cpu to move.

References

"Introduction." *Linux Character Drivers | Introduction*, sysplay.github.io/books/LinuxDrivers/book/Content/Part04.html.

"Character Device Drivers." *Character Device Drivers - The Linux Kernel Documentation*, linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html#implementation-of-operations.

Tang, Jason. "L18DeviceDrivers."

Salzman, Peter Jay, et al. "The Linux Kernel Module Programming Guide."