Project 3: Othello | Reversi

Preliminary Design Document

CMSC 421 Principles of Operating Systems

Caroline Vantiem

Table of Contents

Project Description & Requirements

In this project, a virtual device drive in the form of Linux a loadable kernel module will be created to simulate the gameplay of Othello. The loadable kernel module will implement a virtual character device to perform specific tasks related to Othello and other various tasks. Basic requirements for the project include, completing the project in the Virtual Machine that was set up earlier in the semester, cloning the correct repository containing the skeleton code for project 3, an initial setup of compiling the kernel, incremental development and design requirements, and implementing a basic simulation of Othello. For the incremental development requirement, it is required to make at least four non-trivial commits to the GitHub repository, and one must be done before April 7th. Another requirement of the project is to submit two design documents. One of these documents is the preliminary design document, which is this one. This document gives a brief overview of the projected implementation of the project. The final design document will give a more in-depth description of the project, how functions were implemented, and the overall functionality of the simulation of Othello.

Basic requirements of the project include, building and implementing a character device module, implementing various read and write system calls, and keeping track of the state of the game. A solution to the project does not need to implement any sort of AI, however, it does need to simply choose a random valid move. The module must support a set of commands executed from the user from user-space. It must also do error checking and with other functionality such as printing and keeping track of the game board.

Loadable Kernel Modules in Linux

A loadable kernel module (LKM), is an object file which contains code to extend the running kernel. LKM's are used to extend or add support to hardware, filesystems, or add system calls. LKM's in Linux are loaded and unloaded by commands located in the lib/modules directory, and have the extension .ko. Modules are pieces of code that can be loaded and unloaded into the kernel upon demand.

A character device driver in linux helps facilitate the transfer of data to and from a user. They behave like pipes or serial ports with read and write ability. A device driver also allows the kernel to access hardware connected to the system. The way that a user-space program can interact with a kernel module is through file opening, the user-space program will open a specific file in dev/ which corresponds to which file needs to be loaded. Character devices need to be registered and support a multiple of linux includes in the library.

The required functions that need to be implemented for the project include read, write, open, and release. These four main basic file operations will help facilitate to open the file from user-space, read any data from the user, and write back data to the user. It will then release the driver and class device and free any allocated memory. Other functions such as parsing, keeping track of the board, error checking, and other utility functions will need to be implemented to achieve the Reversi module with full functionality. There will need to be a function to check the validity of moves, whether a play has won, and keeping track of the board itself. All of these functions will need to be implemented into the module so that the user-space driver program can successfully use these functions to simulate the gameplay of Reversi.

Algorithm Implementation

The various algorithms and run-time complexities for the project will need to be optimized and efficient. For starters, most of the file operation functions can have a run-time of O(1) or O(n), there should be no need to have a larger complexity since the module will simply be reading, opening or writing to or from the user. Other functions such as parsing will need to at least be O(n) because the user can enter any number of commands. The algorithm to keep track of the game board will probably be split up into multiple functions, the game board itself will be stored as a global array to be able to access it outside of the scope of any function. This way passing the board game into other functions will be easy. The algorithms for determining whether a player has won will probably be called after each move because that is when the player wins, after a move.

The algorithm for parsing will be like in project 1. The idea is since the user can input many commands in the command line, the last command will be the one used. If the user enters more than one command then the program will parse the commands and only use the last command entered. The module will also check for error and bounds for the board. The module can easily check whether the given coordinates are within the board. However, the challenge will be whether or not a move is valid or what happens after a valid move. There will need to be separate functions to keep track of how places are flipped when a move is taken and if a move itself is valid on the game board. These functions will probably have a run-time complexity of O(n*m), since it will depend on how many pieces are flipped and which pieces aren't. Another way that this will be kept track of is there will be a way to denote spaces that are already taken.

Data Storage & Management

To store the data, specifically things like the game board, the module will use an array. This data structure will be the easiest to implement with this kind of project and should give the easiest structural and functionality. The game board will either be through a function or stored as a global. This way the array can be 2d so that multiple rows and columns can be easily indexed. Another reason why implementing an array is best is because it can be indexed easily. Searching is slow but usually there won't be searching because there are other ways to keep track of moves that are not valid versus valid.

To keep track of player moves, that will be dynamically allocated memory from the buffer. This way the module can use functions like copy_to and copy_from user to correctly get that memory.

Basic Functions & LOC

Some of the basic functions that will need to be implemented are user validation, error handling, parsing, and keeping track of the game board. With all of the functions in the module, the file should be no more than 2000 lines of code hopefully. The game and module will also only be contained in one file, called reversi.ko. Other basic functions will include read(), write(), open() and release(). These functions will help facilitate game play and easability. Other basic functions such as the init and exit functions will be included in the module. The module will also be compiled by the makefile and will be executed by sudo commands in the terminal.

References

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

AprioriIT. (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