David Kravets

CMSC 421 Project 3

Design Document

In this project, a virtual device driver will be created in the form of a Linux loadable kernel module. This kernel module will implement a virtual character device to perform basic read and write operations. This driver will be used to play a game of Reversi in user space, with calls interacting with the kernel. This game of Reversi will be a single player, with the computer playing the “opponent”. The character board will be stored in a char string in kernel space. The driver function will run all operations of the game, and change the board state char string by writing to this char string. The driver function will also handle checking for win conditions, generating moves for the cpu, and verifying all inputs and outputs are correct and properly parsed and sanitized.

Character devices are devices that do not have physically addressable storage media, such as tape drives or serial ports, where I/O is normally performed in a byte stream. In this program, the board state char string will be stored in a character device. This string will be read/written to by a device driver which will execute program commands and modify the string based on the implemented functions.

In this implementation, a Miscellaneous character device was implemented. These Misc devices are under 10.x of the driver list, and are used for handling lightweight modules that do not need dedicated major numbers.

Kernel Module functions:

char opponent (void)

-returns the current players opponent (used for switching players)

void initialboard (void)

-initializes the board char array with correct format for a reversi game

void printboard (void)

-Copies board char array into kernel_buffer

int validp (int move)

-Checks if a player move is valid (within the board parameters).

int findbracketingpiece(int square, int dir)

-Finds all squares that “bracket” a reversi move, used for flipping and validating that a player can choose a certain square.

int wouldflip (int move, int dir)

Finds if a square on the board would flip. Calls findbracketingpiece to confirm that there is a bracketing piece that allows valid flip.

int legalp (int move)

-Checks if a suggested player move is legal according to the rules of reversi

void makeflips (int move, int dir)

-Flip all squares that are affected by a player move

void makemove (int move)

-place player marker on square requested. Call all associated functions to alter game board

int anylegalmove (void)

- Checks if any legal moves exist. Used for pass function and for computer turns

char nexttoplay (void)

- Returns the next player and updates board char accordingly

void legalmoves (void)

- Writes all legal moves to legalMoves[] int array

int reversi (void)

- Main run function for reversi game. Reads input buffer and calls correct functions to run a game of reversi. Input buffer starting with 00 call initializes the game. 01

prints the board. 02 allows for user to play their piece. 03 input allows the computer to play it's piece. 04 checks for available options and passes turn if none.

static int misc_open(struct inode *inode, struct file *file)

-not used but still needs to be declared

static int misc_close(struct inode *inodep, struct file *filp)

-not used but still needs to be declared

static ssize_t misc_write(struct file *filp, const char __user *ubuf,

size_t count, loff_t *ppos)

-This function uses copy_from_user to writes user input to a char buffer that our driver module can access. Also calls reversi() to execute any commands the user imputed.

static ssize_t misc_read(struct file *filp, char __user *ubuf,

size_t count, loff_t *f_pos)

-This function reads the char buffer from our driver and writes it into /dev/reversi so that user space programs can access the information. In this implementation the char board is often written to be printed to the user.

```
static const struct file_operations fops = {
        .owner          = THIS_MODULE,
        .write          = misc_write,
        .read           = misc_read,
        .open           = misc_open,
        .release        = misc_close,
        .llseek         = no_llseek,
};
//Misc device structure
```

The structure below is used to set the parameters for the reversi kernel module. This allows for read/write as well as open/close operations to be called from userspace programs with read() write() open() and close().

```
struct miscdevice misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "reversi",
    .fops = &fops,
};
```

This struct sets the name of /dev/reversi (our interface directory/file) as well as a minor number for our module. This means our character device will be dynamically allocated to a 10.xx driver location. User space programs will be able to read/write from /dev/reversi

static int __init misc_init(void)

-Function initializes our char device and directory /dev/reversi

static void __exit misc_exit(void)

-Function frees all allocated memory pertaining to the character device and stops running the module.

Several changes were made since the outset of this project. It was decided that a Miscellaneous character device would be a more logical implementation than a dedicated character device. The program was broken into many more functions, each with smaller scope. In the original design, it was assumed the entire program could be created in four or five functions. After developing it is more clear that smaller scope functions were easier to debug and develop. This also allowed for functions to be used for multiple commands, such as validation functions.

I ran into issues with copying character arrays into and out of buffers, and still do not have input validation operating correctly. I ran into many issues with

converting my user space implementation into kernel space. This included the rand() function, malloc() and free() and things like passing function pointers. I also ran into problems figuring out the way that char device drivers read and write to a buffer, and how they can run functions and interact with .c programs correctly. This project was fairly extensive and took a huge amount of time to research, understand, and implement.

1. Brandeburg, J. (n.d.). *Developing Out-of-Tree Drivers alongside In-Kernel Drivers*. Intel Corporation.
2. Chapter 10 drivers for character devices. (n.d.). Retrieved April 08, 2021, from https://docs.oracle.com/cd/E19683-01/806-5222/character-21002/index.html
3. Free_hatfree_hat, & Meuhmeuh. (2018, December 01). Emulating a character device from userspace. Retrieved April 08, 2021, from https://unix.stackexchange.com/questions/477117/emulating-a-character-device-from-userspace
4. Iigura. (2019, March 06). Reversi(Othello) game program which you can choose the first or the second in 7 lines code of C. Retrieved April 26, 2021, from https://dev.to/iigura/reversiothello-game-program-which-you-can-choose-the-first-or-the-second-in-7-lines-code-of-c-3799
5. Lyashko, A. (1970, January 01). Hijack linux system calls: Part ii. miscellaneous character drivers. Retrieved April 26, 2021, from http://syprog.blogspot.com/2011/10/hijack-linux-system-calls-part-ii.html
6. Mem.C - drivers/char/mem.c - linux source code (v5.12). (n.d.). Retrieved April 26, 2021, from https://elixir.bootlin.com/linux/latest/source/drivers/char/mem.c

7. Slr. (2021, April 24). Misc device driver - Linux device driver Tutorial PART 32 ⋆ EMBETRONICX. Retrieved April 26, 2021, from https://embetronicx.com/tutorials/linux/device-drivers/misc-device-driver/#Prerequisites