Hanh Do Phung
University ID: 14252074
Email: hdd29@drexel.edu

CS 543: ASSIGNMENT 3
Implementing RAID 0 server on Inferno

SOFTWARE ARCHITECTURE:

1. Introduction

The purpose of this assignment is to implement an driver for a RAID 0 server on Inferno that stores even and odd blocks on two separate disks. This will have the effect of creating an abstraction layer where the user can only see 1 file being written in and read out while underneath it, data is stored interleavingly on two disks.

2. Software architecture

For this assignment, we are given a skeleton driver file to begin with. The driver file will get included in the compilation by editing the configuration file, emu. Most of the basic functions to get a device driver file compiled without error and attached to the system has already implemented, and my main job is to implement the read and write function for the 2 disks. The 2 "disks" will be seen as 2 files specified in the root directory via a command "bind".

My way of developing this server was to create 2 files, and programmed the control file within Dirtab to parse a command called bind. The syntax of this is "bind file0 file1", and it will automatically bind file0 to the even-block-storing-device, and file1 to odd-block-storing-device. To invoke this, issue command "echo bind file0 file1 > '#R'/raid0ctl". I have specified the unique name for this driver is #R, the data file as raid0data and the control file as raid0ctl.

After binding, we can read and write to the 2 files. The figure below illustrate how the abstraction level of RAID 0 works in this case:

The read function was given `Chan *c, void *va, long count, vlong offset` as its arguments, in which c is the channel from which we want to read, va is the buffer that will later on get displayed on stdout, count is the number of bytes to read, and offset is the number of bytes from the start of the channel. A read to the control file will only results in a print statement print(you have read from the control file), and nothing else, as we mostly are using the control file as a place to get commands from the user. On the other hand, a read to the data file will lead us in a while(count > 0) loop, which will read and decrement count until we finish reading, and count to decrease to 0.

In this while (count > 0), we cannot simply do read(to buffer va, read n bytes, starting from offset), since information is stored on 2 separate files. Therefore, we have to read block by block, each iteration of while(count >0) is processing 1 block from its offset in the block, o, to the end of the block, or to the end of count, whichever comes first. The data when read is called each time is written in a memory block in the form of a string and then copied to va at the end via readstr() function.

The heart of the while loop lies in

```
if (addr %2 == 0) {
        r = devtab[chan0→type]→read(chan0, data + nread, n, (addr >> 1)*BLOCKSIZE + o);
} else {
        r = devtab[chan1→type]→read(chan1, data + nread, n, (addr >>1)*BLOCKSIZE + o);
}
```

addr is the block number we are reading in respect to the raid0data file, which is divided by 2 to direct us to read from the respective file underneath, nread is the number of bytes read so far, and o, as explained above.

The logic for write is similar to read, in the way that there is a translation between what raid0data sees, and what each channel (chan0, and chan1) sees. The write function also obtains the values òf o, addr, nread, n each iteration, and instead of devtab[chan]→read, it does write. At the end of a write function call, the file size is updated and reported via raid0tab[Qdata]→length += count.

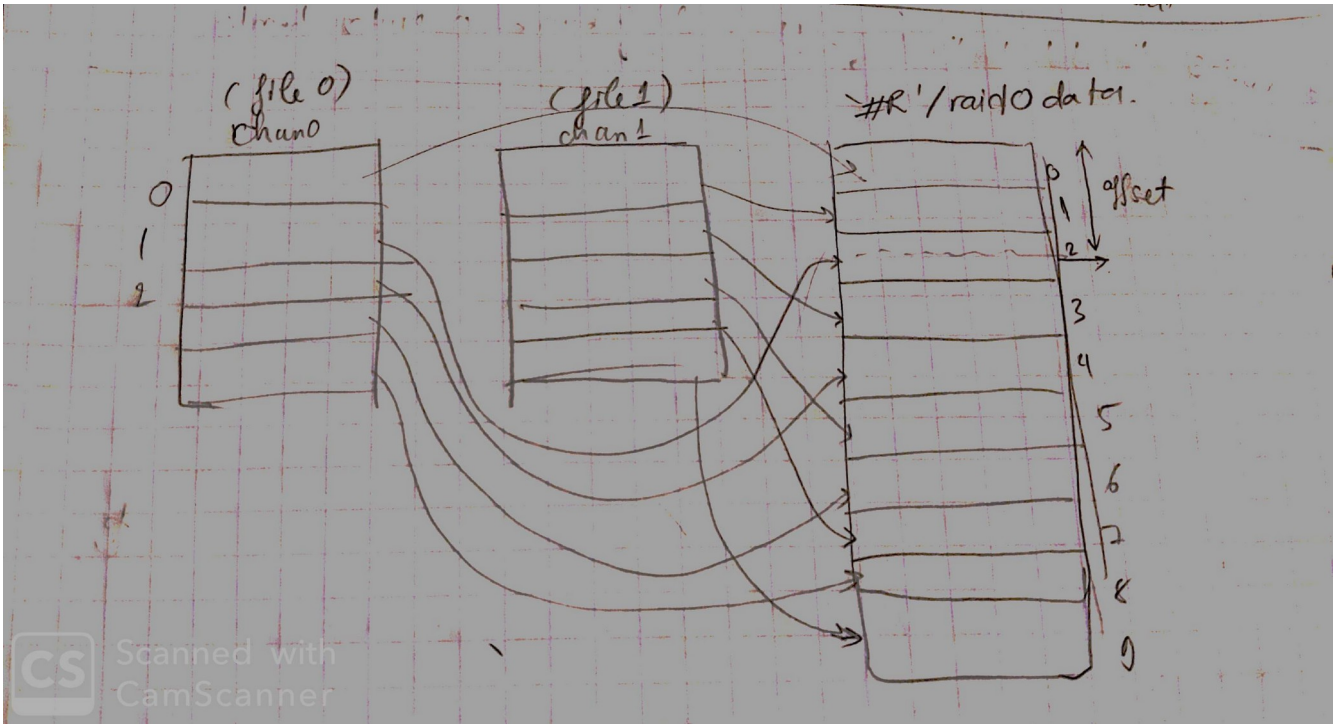

Figure 1: an illustration of how the blocks are mapped together.

TESTING and EVALUATION

The file server was tested by specifying 2 files as 2 disks, then echo different things at it. I also used dd to test the write functions with different configurations on input and output block sizes (-ibs -obs), writing from the middle of the file with -iseek, or writing to the middle of the file by -oseek, while specifying the number of blocks written by -count (block size specified by dd is different from block size of odd and even blocks as seen by the file server). The read function was doing great as I cross

check a cat '#R'/raid0data with cat file0 and cat file1, it turns out as expected. The write function works well with echo > and echo >>, and was able to write to the middle of the file using -oseek in dd. The only bug still persist is that with a write using dd with multiple block (specified by -count x), only the first block was written successfully to the correct spot if -oseek is specified, and correct block size as specified by -bs, while subsequent calls to write results in an error: write: file does not exist.

Also, kfs was not mounted successfully on raid0 driver. I will try fixing that and update you with a new submission before Monday, but if it did not work out, please consider this my last submission.

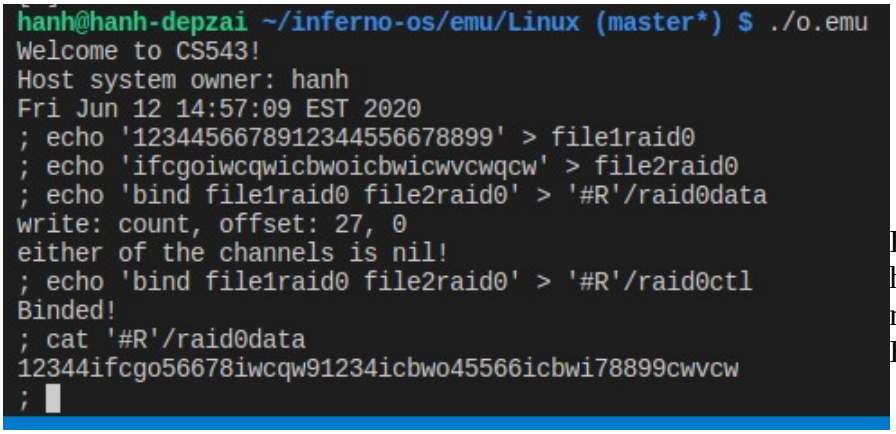

Figure 1: Demonstration of how bind command and the raid0read function works with BLOCKSIZE equals 5 bytes

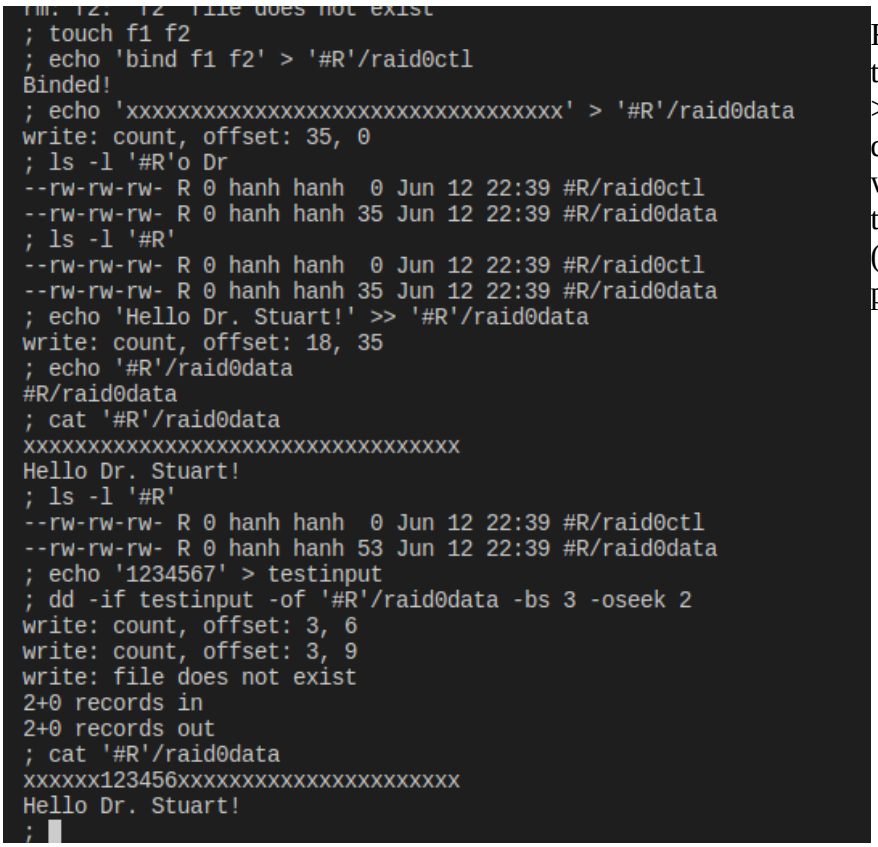

Figure 2: Demonstration of testing made on echo >, echo >>, cat, updating file size, and dd (writing to middle of file) works. The system passed all the test cases in this example. (please disregard the debug out put 'write: ...")