UMass Boston CS 444
Project 2
Due 2 PM on October 27, 2022

1 Introduction

This project develops RAID 2 encoding and decoding capabilities based on the Hamming(7, 4) code. RAID 2 will be described in Chapter 5 of the textbook. Hamming(7, 4) code is described in the Hamming code notes for the class on September 30. Briefly, RAID 2 splits every nibble of a file into four data bits, and then adds three parity bits for error detection and correction. Thus, a nibble will be encoded as seven bits: P1, P2, D1, P4, D2, D3, and D4. The P1 bits of all nibbles in the file are concatenated and stored in one disk drive; similarly for the other six bits. Should one drive fail, Hamming(7, 4) code can detect and correct the error. For this project, however, we are not going to use seven real drives. Instead, we use seven files to keep the seven parts of an encoded file.

Typically we write the least significant bit on the right and the most significant bit on the left. Hamming code uses the reverse orientation. This endianness is critical to successful encoding, decoding, and error correction. Let's illustrate with an example. Consider a file of four bytes of the ASCII code of abcd: 0x61, 0x62, 0x63, and 0x64. The eight nibbles of the file are 0x61626364. The following table lists the corresponding Hamming(7, 4) bits.

nibble	P1	P2	D1	P4	D2	D3	D4
6	1	1	0	0	1	1	0
1	1	1	0	1	0	0	1
6	1	1	0	0	1	1	0
2	0	1	0	1	0	1	0
6	1	1	0	0	1	1	0
3	1	0	0	0	0	1	1
6	1	1	0	0	1	1	0
4	1	0	0	1	1	0	0
	0xEF	0xFA	0x00	0x51	0xAB	0xBE	0x44

Consider the row for the nibble 3, the third from the bottom. In binary, 3 is 0011_2, where the two 1's occupy the less significant half of the nibble. However, in the scheme of Hamming code, they are the data bits D3 and D4, corresponding to the more significant half. With this table, the original four bytes are encoded into seven bytes. We read the columns top-down — note this endianness. Column P1 is 11101111_2, which is 0xEF in hexadecimal.

The tasks for this project are to write two C programs. The first program shall be named `raid.c`. It encodes one file into seven files. It is run like this:

```
vm73> ./raid -f xyz
```

It creates seven files:

```
xyz.part0
xyz.part1
xyz.part2
xyz.part3
xyz.part4
xyz.part5
xyz.part6
```

These files store the bits of P1, P2, D1, P4, D2, D3, and D4, in that order. The command-line option `-f filename` is a required feature.

The second program shall be named `diar.c`. It decodes seven files back to the original file. It is run like this:

```
vm73> ./diar -f xyz -s 16
```

It looks for the seven files `xyz.part[0-6]`, decodes them, and creates a new file called `xyz.2`. The command-line options `-f filename` and `-s numberOfBytes` are required features.

2 Project Tasks

First, create a project folder under your `cs444` directory.

```
vm73> mkdir ~/proj2
```

Next, copy five files from the instructor's directory.

```
vm73> cp /home/ming/cs444/proj2/* ~/proj2
```

The files are the following.

```
-rw-r--r-- 1 hdeblois 5694072 Oct 16 21:33 completeShakespeare.txt
-rwxr-xr-x 1 hdeblois   13224 Oct 16 21:49 diarjhd
-rw-r--r-- 1 hdeblois     106 Oct 16 21:34 Makefile
-rwxr-xr-x 1 hdeblois   13128 Oct 16 21:49 raidjhd
-rw-r--r-- 1 hdeblois       8 Oct 16 21:42 test.txt
```

The two text files `completeShakespeare.txt` and `test.txt` will be used to test your programs. The `Makefile` can be used to compile your project. The two executables `raidjhd` and `diarjhd` are compiled from the instructor's code.

2.1 Task 1: to RAID

You can run `raidjhd` and see its output.

```
vm73> ./raidjhd -f test.txt
vm73> ls -l
vm73> ./raidjhd -f completeShakespeare.txt
vm73> ls -l
```

Your task is to write your own `raid.c` that encodes RAID 2. It accepts the command-line option `-f filename`.

2.2 Task 2: from RAID

You can run diarjhd and see its output.

```
vm73> ./diarjhd -f test.txt -s 8
vm73> ls -l
vm73> ./diarjhd -f completeShakespeare.txt -s 5694072
vm73> ls -l
```

Note that the sizes of these two test files are multiples of four. Thus, the last bytes of the RAID files are filled with real data bits. If the size is not a multiple of four, there will be trailing non-data bits, which must be ignored. However, you are not expected to handle this scenario — assume the sizes of all test files are multiples of four.

You can verify that the restored files are identical to the originals.

```
vm73> diff test.txt test.txt.2
vm73> diff completeShakespeare.txt completeShakespeare.txt.2
```

There should be no output from the `diff` command. When there are no failed drives, we can use just parts 2, 4, 5, and 6 to reconstruct the original file. RAID 2 can perform error correction when one drive fails. The techniques for error detection and correction are described in the notes for the class on September 30. You can test error correction by deliberately corrupting a RAID file. For example,

```
vm73> cp completeShakespeare.txt completeShakespeare.txt.part2
vm73> ./diarjhd -f completeShakespeare.txt -s 5694072
vm73> diff completeShakespeare.txt completeShakespeare.txt.2
```

There should be no output from the `diff` command. If you want to corrupt a different file, remember to run `raidjhd` again to restore the corrupted file back to the correct status before you corrupt another — RAID 2 can handle only one failed drive.

Your task is to write your own `diar.c` that decodes RAID 2 and corrects errors. It accepts the command-line option `-f filename` and `-s numberOfBytes`.

3 Grading Rubric

- (40 points) Your raid.c compiles, and the executable raid can encode.
- (40 points) Your diar.c compiles, and the executable diar can decode and correct errors.
- (20 points) Put a readMe.txt in your proj2 directory with 1) your name, 2) a brief description of your proj2, 3) a comment for each of the lines in the given Makefile explaining what it does (see GNU make utility) and 4) a description of how getopt.h and your code reads input from the command line.