UMass Boston CS 444 Project 2
Posted Wednesday, 26 February 2025
Due Wednesday, 12 March 2025 at 11:59pm

1 Introduction

This project develops **RAID 2** encoding and decoding based on the **Hamming(7, 4)** code. See lecture **hamming.pdf**.

Briefly, **RAID 2** splits every 4-bit nibble of a file into four data bits and adds three parity bits for error detection and correction. So ASCII character 'A' which is one byte (hex 41) is first split into two nibbles (hex 4 = binary 0100 and hex 1 = binary 0001).

Then, using **Hamming(7, 4)**, each 4-bit nibble will be encoded as 7 bits named: P1, P2, D1, P4, D2, D3, and D4. Thus, 'A' which is two nibbles will require 14 bits. However, for **RAID 2** the bits go to 7 separate disks. We will use 7 files instead.

As a reminder, for **Hamming(7, 4)**, these are the parity rules: For P1, take even parity of D1, D2, D4. For P2, take even parity of D1, D3, D4. For P4, take even parity of D2, D3, D4.

This means that the P1 bits of each nibble in the encoded file are concatenated and stored on one disk drive; similarly for the other six types of bits. If one drive (for us, one file) fails, **Hamming(7, 4)** code can detect and correct the error.

Example: Consider an ASCII input file "ABCD", size 4 bytes. In hex bytes, it reads: 0x41, 0x42, 0x43, 0x44. The eight nibbles of the file are individual hex digits: 4,1,4,2,4,3,4,4. The following table lists the corresponding **Hamming(7, 4)** bits by nibble.

nibble	P1	P2	D1	P4	D2	D3	D4
4	1	0	0	1	1	0	0
1	1	1	0	1	0	0	1
4	1	0	0	1	1	0	0
2	0	1	0	1	0	1	0
4	1	0	0	1	1	0	0
3	1	0	0	0	0	1	1
4	1	0	0	1	1	0	0
4	1	0	0	1	1	0	0
	0xEF	0x50	0x00	0xFB	0xAB	0x14	0x44

In this table, the original four bytes (left column 1) are encoded into 7 bit-type bytes (columns 2,3,4,5,6,7,8). Read them top to bottom. For example, column 2 is bit-type P1 with data 11101111_2, which is 0xEF in hex. Check that you can decipher the other columns.

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

```
pe15> ./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` – which is ”raid” spelled backwards. It decodes seven files back to the original file. It is run like this:

```
pe15> ./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.

```
pe15> mkdir ~/proj2
```

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

```
pe15> cp /home/hdeblois/cs444/proj2/* ~/proj2
```

The files are the following.

```
-rw-r--r-- 1 hdeblois 5694072 Feb 26 21:33 completeShakespeare.txt
-rwxr-xr-x 1 hdeblois   13400 Feb 26 21:49 diarjhd
-rw-r--r-- 1 hdeblois     106 Feb 26 21:34 Makefile
-rwxr-xr-x 1 hdeblois   13344 Feb 26 21:49 raidjhd
-rw-r--r-- 1 hdeblois       8 Feb 26 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` were compiled from the instructor's code.

Prepare a `readMe.txt` to list how you developed your code in increments. For each time you work, make an entry in your readMe.txt file with date and what you worked on. Keep a list of sources you use. Put your name in the readMe.txt in a comment at the top.

Keep incremental copies of your code on the server. For example, each day copy your code to `<name>.c-Feb4`. Code will be collected from the server often. Enter your full name at the start of the readMe.txt file and in a comment in your code since some user names bear no resemblance to real names.
If you consult and/or use (small features of) code on the internet, list each source in your readMe.txt and cite the source in your code according to the MIT Writing Code formats. This applies to code you adapt as well. Do not cite fellow student's code because it is not a published source and you are not supposed to share your code. You do not have to cite any code given you by the instructor.
Regarding compiling with a Makefile, the `<name>` of your code must match the name(s) you include in the `Makefile`. Type: `man make` to see information about GNU Make Utility.

2.1 Task 1: to RAID

You can run `raidjhd` and see its output.

```
pe15> ./raidjhd -f test.txt
pe15> ls -l
pe15> ./raidjhd -f completeShakespeare.txt
pe15> 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.

```
pe15> ./diarjhd -f test.txt -s 16
pe15> ls -l
pe15> ./diarjhd -f completeShakespeare.txt -s 5694072
pe15> 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.

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`.

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

```
pe15> diff test.txt test.txt.2
pe15> 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.
More importantly, **RAID 2** can perform error correction when one drive fails. The techniques for error detection and correction are described in the Hamming slides. You can test error correction by deliberately corrupting a RAID file. For example,

```
pe15> cp completeShakespeare.txt completeShakespeare.txt.part2
pe15> ./diarjhd -f completeShakespeare.txt -s 5694072
pe15> 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.

3 Advice

It is best to write your C code on the server using a text editor, e.g., `nano` or `emacs`. Whether you write your code there or on a different machine, you need to compile and run on server each time you work on it because the libraries may differ between machines.

For command-line options, you may want to include the `getopt.h` library and use it to implement the options. As for variables, include int debug and the debug switch `-d` to set `int debug = 1` and use it to turn on `printf` statements.

4 Grading Rubric

Be sure to "make your code your own," by organizing it carefully and by knowing exactly how it works. Do not use AI generated code in your code. Also, the instructor and graders may require you to come in and explain how particular sections of your code work. You will find that knowing the algorithm before you start coding allows you to code your own style.

The first step in grading is to run your executable in your course directory and read your `readMe.txt`. After we check that your executable runs, we copy your code to another location, recompile it and test it again, and then run MOSS on all pairs of code. In grading, we run Stanford's Measure of Software Similarity (MOSS) to check that no one duplicates someone else's code, but if a too high level of duplication occurs, both students will be penalized.

Be sure to notice how the uppercase and lowercase letters are used in naming the files and directories that you are required to create (`Makefile`, `proj1`, `readMe.txt`). Names must match exactly or your submission will not be collected.

Any file changes made after 11:59 pm on the due date will not be taken into account in grading. Any request for an extension must be received before the due date/time and will only be considered if substantial (compiled/run) work has been done along the way and is on the server. Extensions may result in taking some points off your grade. Code submitted late will lose 15 points.

- (15 points) You have the right files in the right places
- (15 points) Both your programs compile without errors or warnings, the executables match the ones you submitted, and the input switches work
- (15 points) `readMe.txt` contains your full name, you list the six commands required (two to run your raid with each test file, two to run your diar with each test file and two to show how to display debug for each program), you list of what was worked on in each increment and you list of sources consulted if any.
- (20 points) Your full name is in both programs, your code reads the CL correctly, your code is correctly indented, names are well-chosen, comments are useful, and citing is specific – including URL, whether copied or copied and modified and the end of the cited section is marked so we know where your own code resumes
- (5 points) The sources you cite in your programs are a subset of the sources you cite in your readMe.txt and you did a code walkthru if requested

- (15 points) Tests with test.txt passed, output for completeShakespeare.txt is correct, and your debug switch shows intermediate results

- (15 points) After doing the suggested command that corrupts the output file your raid created, your diar can correct the error and diff shows no difference between original input file and corrected file.