Assignment 1: Computing parity

- Due Jan 30 by 10p.m.
- Points 1

Note: There will be some automated tests made available in the next day or so for you to ensure that your code compiles and runs.

Introduction

Later in this course, you will build a simple simulation of a RAID

Links to an external site. file system. RAID (Redundant Array of Independent Disks) is a method of storing data across multiple data storage devices, and was created to improve performance and reliability. One way in which RAID can improve reliability is by storing "parity blocks" that provide enough information to reconstruct a block if one of the data storage devices fail.

In this assignment, you will complete a program to compute parity bits. You will be able to use some of the code you write for this assignment as part of assignment 3. Each of the functions that you are asked to implement are fairly small, so you will not need helper functions and you should not be surprised if it doesn't take much time to complete this assignment.

Git Repository

As you have already done in your labs, log into MarkUs using your UTORid and password. Navigate to Assignment 1 and add the starter code to your repository.

Then, pull the changes to the checked out local version of your repo using `git pull`, or you can clone a fresh copy using `git clone` with the URL on your local machine and/or the lab machines.

As you are working on the assignment, use the git commands `add` and `commit` to commit changes to your local repository, and `git push` to push the changes to the remote repository managed by MarkUs. If you do not run `git push` you have not submitted your work and will not get credit for it.

Tip: Use `git add` for each file you want to commit rather than adding all the files in the directory. For example, run `git add parity.c` instead of `git add .` or `git add --all`. This will prevent you from having to worry about committing files that you shouldn't. You can

also create your own `.gitignore` file to prevent git from committing files that you don't want to commit.

Background: Why Parity?

Parity in mathematics refers to whether an integer is even or odd. In the context of *error detection*, a parity *bit* is used to store whether a set of bits has an odd number of 1s.

This parity operation can be implemented easily, using the Boolean xor function. If one of the bits change, the parity of all the bits will no longer match the previous parity bit, allowing us to detect the error and even restore missing data. For a simple example, suppose the data is a binary number of value 0b00101011 (in C we can write numbers in binary by prepending 0b, much like how we can write hexadecimal numbers by prepending 0x). Then we can compute a parity bit for the string: 0, as there is an even number of 1s. This parity bit can then be used as a simple check for whether the binary number has changed erroneously. If one bit gets changed, then recomputing its parity bit would no longer match the previous parity bit of 0.

As a larger example of using parity to *recover data* in the context of RAID, suppose we want to store the string "CSC209", but split up across three blocks of two characters each. We can then extend the parity bit to a parity *block*, where it stores the parity of the bits at all the blocks in each position.

Block 1	`'C' == 67 == 0b01000011`	`'S' == 83 == 0b01010011`
Block 2	`'C' == 67 == 0b01000011`	`'2' == 50 == 0b00110010`
Block 3	`'0' == 48 == 0b00110000`	`'9' == 57 == 0b00111001`
Parity block	`'0' == 48 == 0b00110000`	`'X' == 88 == 0b01011000`

This parity block can be used, as before, to detect errors in the data. If part of one block changes erroneously, then a newly computed parity block will no longer match the previous parity block. But this parity block can also be used to *recover* data in the event that one block is lost (e.g. if the device that block is stored on fails). Suppose block 2 is lost. Then we can recover the two characters in the block using the parity block, using a similar method as computing the parity block. Try taking the xor of the remaining blocks and the parity block, and see what you get! This should be `'C' ^ '0' ^ '0'` for the first byte and `'S' ^ '9' ^ 'X'`

for the second where `^` is the xor operator in C. (Note that the byte is the same whether we interpret it as a character, as a decimal value or as a binary number.)

Logistics

Because this is the first C programming assignment, we have avoided using some features of C that we will get to later in the course. In particular, we are not using strings, dynamically allocated memory or header files. We are also not using the best method to read command line arguments. All of the tests we use will have valid command line argument values.

To compile the programs, include all .c files needed to compile a specific main function on the compile line:

```
gcc -Wall -g -o compute_parity compute_parity.c parity.c
gcc -Wall -g -o restore_block restore_block.c parity.c
```

2D arrays in C

We haven't discussed 2D arrays in C. You can see in the starter code how a 2D array is declared.

```
char data[num_blocks][block_size];
```

Here `num_blocks` is the number of rows in data and `block_size` is the number of columns. You can also see from the call to `get_data` and the parameters in the definition of `get_data` how we pass a 2D array into a function. In this case, we need to specify the second dimension or number of columns, so that the compiler knows how long each row is. This allows us to use the normal indexing into a 2D array, as you can see in the body of `get_data`.

Part 1: Compute parity

The file `compute_parity.c` contains the `main` function that calls the functions defined in `parity.c` The functions in `parity.c` include those that you need to write and several helper functions that are provided for you. Please read the comments in the code carefully to learn what each function does.

In this program, we are reading data from a file and storing each line as a *block* in the 2d array `data`. Each row of `data` contains bytes from one line of the file and represents one block. The number of rows and number of columns in `data` are specified by the first two arguments to the program: `num_blocks` and `block_size`. Read through `get_data` to see how lines from the file are stored in `data`. Read through `compute_parity.c` to see how the data structures are declared.

Step 1: Complete `print_data_block` and `print_data`, so that you can see what the values in the data array are.

Example: `print_data(4, 64, data)` where `data` is read from `quotes.txt` will print the following

"Humans are allergic to change. They love to say, 'We've always
"Computers are incredibly fast, accurate, and stupid. Human bein
"Computers are good at following instructions, but not at readin
"If debugging is the process of removing software bugs, then pro

Step 2: Complete `compute_parity_block` to compute in the parity values for each column of data. The parity value for column j is the xor of all of the rows in column j. The xor operation in C is ^. Note that the xor operator will apply to each element of the array. (You don't somehow have to figure out how to get at individual bits.)

Step 3: Complete `print_parity_block` to print the values in `parity_block`. Note that the values in `parity_block` are not necessarily printable characters within the range of ASCII values. This means that it isn't helpful to print them as characters. Instead we will print them as hexadecimal numbers since one byte can be represented using 2-digit hexadecimal numbers. Use `"%02x "` as the format specifier and leave a space in between each hexadecimal number.

Example: `print_parity(64, parity_block)` where `parity_block` is computed using the 4 blocks printed above with a `block_size` of 64 will print:

00 01 13 4d 05 0b 11 55 06 15 0c 4e 06 4c 05 18 53 1f 17 43 05 19 5f 45 16 41 18 13 43 17 0a 47 67 07 05 10 50 16 00 17 1d 1a 15 09 43 08 53 1d 49 04 0c 28 43 00 32 51 4d 48 15 5c 42 08 01 4f

Something to think about: If a parity value is `00`, does that mean that all the characters in that column (or at that byte number in the block) have the same value?

Now you have written all the code necessary to compute the parity block for a list of blocks and can think about how to test it.

Part 2: Restoring a block using a previously computed parity block

In a RAID array, each of our data blocks would be stored on a different storage device, and an extra storage device would be used to store the parity block. This means that if one storage device fails, we can reconstruct the missing data using the data from the other storage devices

and the parity block. We will see how to implement this in this part of the assignment. It turns out to be quite simple. The `restore_block.c` file is a program with a main function that you can use to test the `reconstruct_block` function. (It also first computes the parity blocks, but leaves out some print statements.)

Step 4: Implement `delete_data_block`. To simulate a storage device failing we will set all of the values in row `block_num` to `'\0'` so that the data from that block has been deleted. (This is equivalent to setting the values to `0`.

Step 5: Implement reconstruct_block to fill in the data for row `block_num`. Instead of xor'ing all the data blocks, this function will xor together all the parity block and all of the data blocks *except* the deleted one.

Part 3: Testing

To ensure that your program works correctly, you will need to identify some good test cases. While it is fun to try out your code on large file, that doesn't necessarily help you understand whether your program works well. A good set of test cases will require you to create a few small text files probably even smaller than quotes.txt. The reason you want small input is so that you can run it, read the output and quickly tell whether your program is working or what might be wrong with it. You should also be able to create some small examples like the diagram above so that you can test `compute_parity_block` before you write `reconstruct_block`.

Your task is to describe 3 test cases for `get_data`, 3 test cases for `compute_parity_block`, and 3 test cases for `restore_block` in a plain text file called `testing.txt`. Your file should clearly distinguish between the test cases and will contain 1) a brief description of the test that explains why you think it is a good test, 2) the command used to run the test on a line by itself, 3) the expected output. Any test files used for these tests should committed to your A1 directory. For example my `testing.txt` file might contain the following test description:

Test 1: Test that compute_parity_block will work in the example case with 4 data blocks and a block_size of 8
We expect the first parity value to be 0 because the first character on each line is the same.
./compute_parity 4 8 quotes.txt
Expected output:
Data blocks:
"Humans
"Compute
"Compute
"If debu
Parity block:
00 01 13 4d 05 0b 11 55

Every test case must test a different case, and your description needs to explain why it is a different case. You will not get credit for repeating any test case that is used as an example in this handout. Only one test case out of the 9 cases can test for an error.

Submission and Marking

Your submission should be committed and pushed to your git repository by the due date.
We are using automated grading tools to provide functional feedback, so it's important that your submission be in the correct location in your repository (under A1), have the correct filenames.

Files to submit:

- parity.c - all your coding work will be done in this file.
- compute_parity.c
- restore_block.c
- quotes.txt
- testing.txt
- any text files you create for testing your program.

Your program must compile on a teach.cs lab machine, so please test it there before submission. We will be using gcc to compile program as follows:

gcc -Wall -g -o compute_parity compute_parity.c parity.c
gcc -Wall -g -o restore_block restore_block.c parity.c

Your program must not produce any error or warning messages when compiled. Programs that do not compile will receive a 0. Programs that produce warning messages will be penalized.

We may use other programs to test the functions you have written in `parity.c`, so you may not change any function signatures.

We will be running automated tests on your program, so the names of the files, the names of the functions, and the function signatures must not be changed from the starter code.