Tània Pazos, u185115
Júlia López, u214964

Report Lab 1

1. Explain the parsing of the arguments, and the checks that you have implemented to verify that they are correct.

The parsing of the arguments is done in the main part of our main.c file which is where we have implemented the checks to make sure that we only accept correct arguments and in an accepted order.

The first thing to take into account before performing the checks is where and how do we pass the arguments. The int main function, has two arguments: int argc, which holds the integer number of arguments (which since the main is parsed, it will always be at least one), and char* argv[], an array holding all the arguments (argv[0] will be main).

We have started our checks assuring that the number of arguments passed is a valid amount. The argc cannot be less than 3 or more than 5. So, we evaluate if it falls within this range. If it does not, we exit with code 22.

```
if (argc < 3 || argc > 5){
        printMenu();
        exit(22);
}
```

One thing we assume will be correct is that the first parsed argument -argv[1]- will have the correct file format.

To perform the checks for the other arguments, we have created flag variables to know which one we need to perform. Then, we loop through each argument and compare them to the list of options. We start the loop with the second argument since the previous ones will always be main and filename.

What we compare is an argument stored in the argv array and the string. To do so, we use the string processing function strcmp(const char * s1,const char * s2) which when both strings are equal, returns zero. If this is the case, we set to one the flag variable according to the option.

```
if(strcmp(argv[i], "-generate") == 0){
            generate = 1;
}
```

For the “-maxNumErrors” we evaluate inside the if condition if there is another argument after that one seeing that it would not be valid without the numErrors value. If there is, then we use atoi(const char * str) -another string processing function- to convert string argument to integer to store it in maxNumErrors. Then, we skip one argument to loop since we already checked for the numErrors.

```
else if (strcmp(argv[i], "-maxNumErrors") == 0){
      if(i+1 < argc){
        maxNumErrors = atoi(argv[i+1]);
        i++;
}
```

At the end of the loop, if there are no accepted arguments, we print the menu with the options again, and exit with code 22.

```
} else {
    printMenu();
    exit(22);
}
```

The last thing we need to check for, is that there are no arguments that can not be together, or if none of the mandatory arguments appear.

```
if((generate && verify) || (!generate && !verify)){
      printMenu();
      exit(22);
 }
```

The arguments -generate and -verify cannot appear together, but in order to perform some operations there has to be one of them among the parsed arguments.

2. Suppose that you have a variable blocksize, that is given per command line. What would you need to modify?

If we add a variable given per command line, we will first need to modify the checks by adding the argument “-blockSize” which will need to be followed by “-numBlockSize” integer, similar to the check we perform for “-maxNumErrors”.

```
else if (strcmp(argv[i], "-blockSize") == 0){
      if(i+1 < argc){
        numBlockSize = atoi(argv[i+1]);
        i++;
}
```

We will also need to change the way in which we read from a file and, therefore, compute the crc. Instead of assuming that the block size is 256 bytes, we will need to pass the “numBlockSize” each time we read from a file as the number of bytes to read.

```
// Read block from input file
bytesRead = read(fd, &buff, numBlockSize);
```

3. Explain the handling of the last block, which might be smaller than the blocksize.

We designed our code in a way in which we do not especially treat only the last block differently. What we did, was defining the number of bytes for each block as 256, using the function read(int fd, void* buff, int nbytes) which return the number of bytes successfully read.

Once we read the first block, the condition to enter the loop is that the number of bytes read is bigger than 0. This condition is where the code allows to adapt to the actual size of the file because it does not request each read to be exactly 256 bytes. The loop continues, until no bytes are read.

```
startTimer();
bytesRead = read(fd, &buff, 256);
totalTimeRead += endTimer();
while (bytesRead > 0){
```

Here is where we can see the implementation of the read function (line 2), and then the condition of the loop (line 4), in the generateCRCFile function.

For the verifyCRC function the idea is the same, but we implemented it differently. We have written a condition that when the bytes read are 0, then it exits the loop.

```
if (bytesRead <= 0) break;
```

4. Compute the total timing for computing the CRC, and reading, of the file ”test large.txt”. Assuming that your computer only has a single CPU, and knowing that the CRC does not use any I/O operation, discuss about the potential acceleration of using concurrency on this program.

As we can see below, the total timing for reading the file “large_file.txt” is of 7801 microseconds, and for computing the CRC is of 118240 microseconds, which sums up to be 126041 microseconds (following the image below).

```
Microseconds spent in read are 7801 (generate)
Microseconds spent in computing the CRC are 118240 (generate)

Microseconds spent in read are 3463 (generate)
Microseconds spent in computing the CRC are 91389 (generate)
```

We have done this on different computers and several times, and we have obtained different results. We assume that it is because we have different CPUs and that each process is different.

Using concurrency on this program will be beneficial for a more rapid CPU computation time. If we break down the CRC calculation using separate threads with smaller blocks and

running them separately will decrease the running time and be more efficient since the CPU will be switching jobs with a response time of less than 1 second and the switch time will be reduced.