Assignment 1

This assignment was locked Mar 3 at 11:59pm.

Objectives

- Familiarize yourself with the C programming language
- Familiarize yourself with the shell / terminal / command-line of UNIX
- Learn how to use a proper code editor, such as cLion (or emacs or vi or...)
- Learn a little about how UNIX utilities are implemented

While the project focuses upon writing simple C programs, you can see from the above that even that requires a bunch of other previous knowledge, including a basic idea of what a shell is and how to use the command line on some UNIX-based systems (e.g., Linux or macOS), how to use a code editor such as cLion or emacs or vi, and of course a basic understanding of C programming.

Before beginning: If you need a refresher on the Unix/C environment, read this tutorial

(Links to an external site.)

.

To start cLion you can type cLion in the VM that you installed.

Summary of what gets turned in:

- A single .c file **summatrix.c**.
- It should compile successfully in cLion on the VM given.
- Check it passes the tests supplied to you.
- Also think of a few of your own tests and check it passes them.

matrix summation

For this assignment we will be implementing the matrix sum operation, which sums up all the values in a matrix. Sometimes you have a file of numbers that represent a matrix (see matrix.txt
Download matrix.txt
example, which has rows and columns space-separated). We want to write a program that sums all the numbers for us. We are going to write a utility that is missing from linux: **summatrix**. To create the **summatrix** binary, you'll be creating a single source

file, **summatrix.c**, which will involve writing C code to implement it. You could write a complicated function with awk, but the syntax is hard to remember, thus we will write **summatrix** with a clean interface.

Assume that you have a text file with a matrix row on each line, and rows are separated by newlines. The values in each row are space-separated into columns. You want to read the numbers in and sum them up. Since we want to be flexible on the type and size of numbers we can handle, you can use integers and you *expect a file to contain non-negative numbers from 0 upwards*.

You must be able to handle a file with any number of rows (including no numbers). You can assume that the input file contains only valid integer numbers and rows are separated by newlines; the input file may contain a few empty lines as well (which you will ignore).

For example, matrix.txt

Download matrix.txt

matrix.txt contains a matrix of numbers (integers). *Your program takes as its command-line parameters the filename and the number of columns N*. Then it prints the sum of numbers. *If there is a number in the input file that is less than 0, your program ignores it for the sum purposes* and prints a message to stderr. In other words, negative numbers count as 0 and the program sums the positive numbers only.

```
$ ./summatrix matrix.txt 4
```

which will output to **stdout**:

```
Sum: 72
```

In the example above, if your input file contained values less than 0 (negative) you should also print to **stderr** messages like:

```
Warning - Value -7 found.
```

Note: for this assignment you can assume each row has at least as many numbers as parameter *N* specified, and you can ignore numbers beyond *N*. You don't have to check if the matrix is well-formed, e.g. if each row has the same number of columns.

If you want to see just the stderr messages, you can redirect the stdout to /dev/null. For example:

```
$ ./summatrix matrix.txt 4 > /dev/null
```

If you want to ignore the stderr messages, you can redirect the stderr to /dev/null. For example:

```
$ ./summatrix matrix.txt 4 2> /dev/null
```

The "**./**" before the **summatrix** above is a UNIX thing; it just tells the system which directory to find the **summatrix** binary in (in this case, in the "**.**" (dot) directory, which means the current working directory).

Note: matrix.txt has an empty line in the end. If you download the file and open it in a text editor, there is an empty line in the end. This shouldn't bias your result. An empty line should be ignored by the program.

To compile this program outside of cLion, you can also do the following:

```
$ gcc -o summatrix summatrix.c -Wall -Werror
$
```

This will make a single *executable binary* called **summatrix**, which you can then run as above.

grading

submitted code compiles cleanly (no errors/warnings, check with **gcc -Wall -Werror**)	10 %
overall requirements of assignment are satisfied	20 %
output correctly for given test case with input file matrix.txt	10 %

outputs correctly for tester's test cases	20 %
program flow is understandable	10 %
number of lines in file (matrix rows) that can be processed is not limited	10 %
code is commented and indented (use of whitespace)	10 %
please submit **summatrix.c** inside a zip file called **proj1.zip**	10 %

submission

Upload a zip file called **proj1.zip** that contains a single file: **summatrix.c** . The submitted zip file contains **summatrix.c** and it doesn't need any other files to compile (besides the standard C libraries).

further details

- In all non-error cases, **summatrix** should exit with status code 0, usually by returning a 0 from **main()** (or by calling **exit(0)**). This is called the exit code of a program and is different from what your program prints to stdout/stderr.
- If *no file* or other parameters are specified on the command line, or an empty file with no numbers is given, **summatrix** should just exit and return 0.
- If the program tries to **fopen()** a file and fails, it should print the exact message "range: cannot open file" (followed by a newline) and exit with status code 1.

You may need to learn how to use a few library routines from the C standard library (often called **libc**) to implement the source code for this program, which we'll assume is in a file called **summatrix.c**. All C code is automatically linked with the C library, which is full of useful functions you can call to implement your program. Learn more about the C library here

(Links to an external site.)

and perhaps here

(Links to an external site.)

.

For this project, we recommend using the following routines to do file input and output: **fopen()**, **fgets()**, **fscanf()**, and **fclose()**. Whenever you use a new function like this, the first thing you should do is read about it -- how else will you learn to use it properly?

On UNIX systems, the best way to read about such functions is to use what are called the **man** pages (short for **manual**). In our HTML/web-driven world, the man pages feel a bit antiquated, but they are useful and informative and generally quite easy to use.

To access the man page for **fopen()**, for example, just type the following at your UNIX shell prompt:

```
prompt> man fopen
```

Then, read! Reading man pages effectively takes practice; why not start learning now?

We will also give a simple overview here. The **fopen()** function "opens" a file, which is a common way in UNIX systems to begin the process of file access. In this case, opening a file just gives you back a pointer to a structure of type **FILE**, which can then be passed to other routines to read, write, etc.

Here is a typical usage of **fopen()**:

```
FILE *fp = fopen("numbers.txt", "r");
if (fp == NULL) {
    printf("cannot open file\n");
    exit(1);
}
```

A couple of points here. First, note that **fopen()** takes two arguments: the *name* of the file and the *mode*. The latter just indicates what we plan to do with the file. In this case, because we wish to read the file, we pass "r" as the second argument. Read the man pages to see what other options are available.

Second, note the *critical* checking of whether the **fopen()** actually succeeded. This is not Java where an exception will be thrown when things goes wrong; rather, it is C, and it is expected (in good programs, i.e., the only kind you'd want to write) that you always will check if the call succeeded. Reading the man page tells you the details of what is returned when an error is encountered; in this case, the macOS man page says:

```
Upon successful completion fopen(), fdopen(), freopen() and
fmemopen() return
a FILE pointer.  Otherwise, NULL is returned and the global
variable errno is
set to indicate the error.
```

Thus, as the code above does, please check that **fopen()** does not return NULL before trying to use the FILE pointer it returns.

Third, note that when the error case occurs, the program prints a message and then exits with error status of 1. In UNIX systems, it is traditional to return 0 upon success, and non-zero upon failure. Here, we will use 1 to indicate failure.

Side note: if **fopen()** does fail, there are many reasons possible as to why. You can use the functions **perror()** or **strerror()** to print out more about *why* the error occurred; learn about those on your own (using ... you guessed it ... the man pages!).

Once a file is open, there are many different ways to read from it. We suggest **fscanf()** or **fgets()**, which is used to get input from files, one line at a time.

scanf uses **any whitespace as a delimiter**, so if you just say scanf("%d", &var) it will skip any whitespace and then read an integer (digits up to the next non-digit) and nothing more.

To print out file contents, you may use **printf()**. For example, after reading in a line with **fgets()** into a variable **buffer**, you can just print out the buffer as follows:

```
printf("%s", buffer);
```

Note that you should *not* add a newline (\n) character to the printf(), because that would be changing the output of the file to have extra newlines. Just print the exact contents of the read-in buffer (which, of course, many include a newline).

Finally, when you are done reading and printing, use **fclose()** to close the file (thus indicating you no longer need to read from it).

Assignment 2

Objectives

- Understand how processes work and how file get processed in parallel
- Understand how multi-processing can speed-up execution

- Understand that processes run independently but they always have a parent process
- Learn how processes communicate through pipes
- Learn to program processes in such a way that a slow process won't cause the faster processes to stall; eliminate bottlenecks.

For this assignment you will extend **summatrix.c** (from assignment #1) to compute the sum of multiple matrices in a parallel fashion using multiple processes. For example, the ith child process will process the ith matrix.

You will develop a program **summatrix_parallel** that takes multiple matrices on the command line. You will need to compute the sum of each matrix similar to assignment #1 using the standard matrix summation method you implemented.

Assume that you have several files with numbers in columns separated by newlines. You want to read the numbers in from all the files. You should use the same integer data type as in assignment #1 over all files.

If a file appears multiple times on the command line, those numbers will be processed multiple times.

You should use **fork()** to start a process for each of the matrices, such that we can compute a sum for each matrix individually in parallel. To communicate between processes you may use **pipe()**. You need to wait for the processes to finish using **wait()**.

You must be able to handle files with any number of numbers (including no numbers). You can assume that the files are well formed: they will contain only valid numbers separated by newlines (and possibly an occasional empty line, same as in assignment 1).

The code/ipc directory contains examples of using fork, pipe and wait, which you can consult for help.

For example, running on the matrix.txt file will output:

```
$ ./summatrix_parallel matrix.txt matrix.txt 4
```

which will output to **stdout**:

```
Sum: 144
```

In the example above, if your input file contained values less than 0 (negative) you should also print to **stderr** messages like:

```
Warning - Value -7 found.
```

```
$ ./summatrix_parallel matrix.txt matrix.txt morematrix.txt 4
```

will output the following, if `morematrix.txt` only contains one row of "1"s:

```
Sum: 148
```

Same as assignment 1, empty lines in a file don't count and can be ignored.

An objective of this assignment is to learn to program multiple processes in such a way that a slow process won't cause the faster processes to stall (eliminate bottlenecks). For example, assume the first matrix involves lots of floating-point operations (slow), while the second matrix contains all zero numbers and your linear algebra library detects and optimizes the run, such that no additions are needed and it could finish instantly. The *wrong* way to implement the code would be for your second process to have to wait for the first process (which does more processing) to completely run through completion before the second process even starts. That would resemble the situation where a line-up of cars are trailing a slow-speed tractor on a single-lane road. For this reason, the parent should fork processes that will run independently and wait for any process to finish; the OS will handle the context switching between processes.

Grading

overall requirements of assignment are satisfied	20 %
compiles cleanly (no errors/warnings, use **gcc -Wall -Werror**)	5%
output correctly for given test case	10 %
outputs correctly for tester's test cases	10 %
is fork() used correctly	10 %
is wait*() used correctly	10 %

is pipe() used correctly	10 %
program flow is understandable	5%
code is commented and indented (use of white space)	10 %
number of numbers that can be processed is not limited	5%
submission of **summatrix_parallel.c** inside a zip file called **proj2.zip**	5%

Submission

Upload a zip file called proj2.zip that contains a single file: **summatrix_parallel.c**

further details

The exit codes are the same as for assignment #1.

- In all non-error cases, **summatrix_parallel** should exit with status code 0, usually by returning a 0 from **main()** (or by calling **exit(0)**). This is called the exit code of a program and is different from what your program prints to stdout/stderr.
- If *no file* or other parameters are specified on the command line, or an empty file with no numbers is given, **summatrix_parallel** (possibly a child process) should just exit and return 0.
- If the program (possibly a child process) tries to **fopen()** a file and fails, it should print the exact message "range: cannot open file" (followed by a newline) and exit with status code 1.