CSC 412 – Operating Systems

Programming Assignment 05, Fall 2024

Wednesday, November 6th, 2024

Due date: Wednesday, November 20th, 11:55pm.

1 What this Assignment is About

1.1 Objectives

The objectives of this assignment are for you to

- Use the `fork()` function to distribute tasks among processes;
- Create a server-client architecture using named and/or unnamed pipes;
- Use blocking and non-blocking system calls for inter-process communications;
- Use scripting to solve an old, classical headache of cross-platform development.

This is an individual assignment.

1.2 The problem

The simplified/abstracted distributed computation task you are asked to solve involves the processing of a large number of data files. The data files have been produced/prepared so as to be treated by a specific number of processes, and each data file starts with a line giving the index of the process that should process it. All the data files are mixed together in a folder and your solution will involve the following steps:

1. Your script will search through all the data files to determine the total number of processes required for the project. It will then launch the parent/server process, with the desired number of processes and data folder's path as arguments.
2. The parent process will create appropriate communication channels (more on that later) and the proper number of child processes. The child processes will then proceed to distribute the data files between themselves, that is, to attribute each data file to the process that should work on it.
3. When each process has the full list of data files it is supposed to work on, it will launch a child process of its own, to actually process the data.
4. When the data processing processes, they will each report their result to their parent process, who in turn will report it to the root parent/server process. of files

Yes, that's a lot of process creation and a lot of pipelining between all these processes. Hopefully, the architecture of the whole system will become clearer as we progress through this document.

1.3 Handouts

The handouts for this assignment are this document (a pdf file) and datasets for you to practice with.

1.4 Reminder on specifications

When I tell you "name the program/script/function" `XYZ` and put it in a folder named `ABC`, then I expect you to do it exactly the way I say, and you will get points taken off if you don't. There are plenty of places in these assignments that you can make any design and implementation choices you want, but when I give specs, they are to be followed.

Feel free to think of me as your "point-haired boss." I am fine with that. When you are out there in industry you will be expected to follow company policies, no matter how bone-headed they may be or seem to be. These are my policies. Follow them or lose points.

2 Part I: `bash` Scripts

As usual, all scripts will go directly in the `Scripts` folder and will be run from the root folder of the assignment.

2.1 Data files

The data folder contains exclusively data text files. The files are named arbitrarily, so you will have to rely on their catalog order to distinguish them from one another. Each data file starts with a line giving the index of the process it should be assigned to.

2.2 Build script

This script named `build.sh` will build executables for all the versions of the assignment that you have completed.

2.3 Run scripts

For each version that you implemented, you should provide a script named after that version, e.g. `script01.sh`.

2.3.1 What a run script does

The script receives as arguments the path to the folder containing all the data files (and only the data files) and the path to the output file to produce. It searches throughout the data files to determine

the largest process index, then it launches the server/dispatcher process, passing as arguments the number of child processes to create, the path to the data folder, and that to the output file.

Figure 1 gives an example of such a task execution. The script having gone through all the files in the data folder determined that the largest process index is 3, and therefore it launches the root/server process passing as arguments 4, as the number of processes to create, and the path to the data folder.

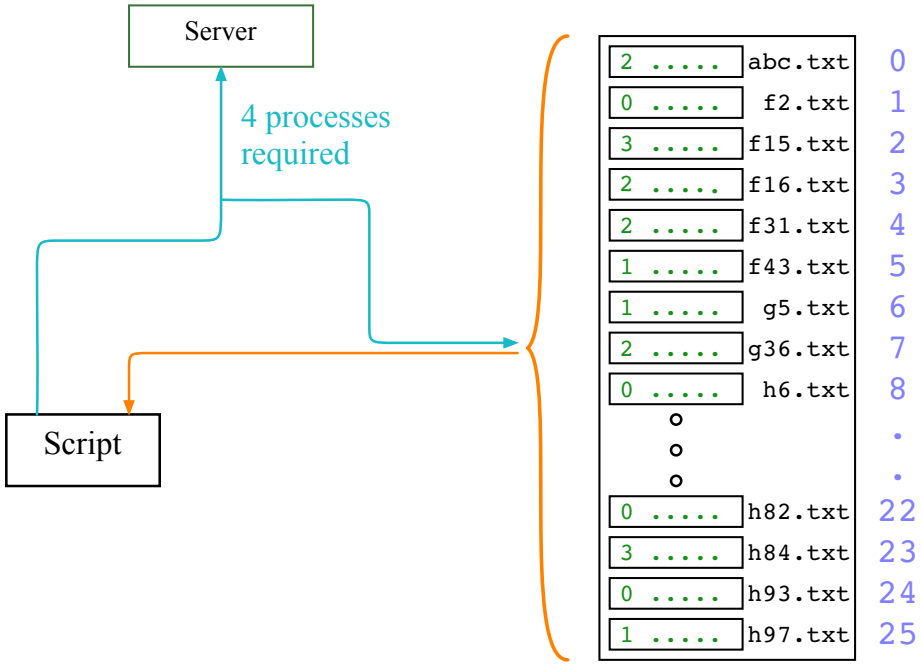

Figure 1: The data distribution task.

2.3.2 Implementation details

Of course you could try to read the first line of each data file, but I want you to start learning about *regular expressions* and the `grep` command. There are plenty of online references on either topic. For Mac people, I highly recommend the chapter on regular expressions in the user manual of the text editor BBEdit.

2.4 one more script: The end of line headache

The purpose of this script named is to address a very old, vexing problem that all of us who have to use different platforms must run into, sooner or later.

2.4.1 The problem

The issue as stake here is that of the characters used to indicate a new line in a text file, that is, the end of the current line. This can be referred to as the “newline,” “end of line,” or line ending

problem. It carries over from the days of mechanical typewriters. When typists arrived to the end of a line, they had to execute a "linefeed" (LF) to make the paper sheet move up by one line, and then a "carriage return" (CR) to reposition the head at the beginning of the new line.

When the different operating systems were developed in the 70s and 80s, they came with different scheme to represent the end of line, most based on the linefeed (LF, `'\n'`, `0x0A`, 10 in decimal) or the carriage return (CR, `'\r'`, `0x0D`, 13 in decimal).

Most notably:

- Early on, Unix systems (and later on, Amiga, BeOS, and Linux) opted for LF;
- The TRS 80, the Apple II, and the original Macintosh System (then Mac OS until Mac OS 9 up to the early 2000's), chose CR;
- CP/M, and therefore DOS, Windows, and OS/2 (along with Atari TOS and others) picked CR+LF.

As you can imagine, this posed several problems. For a start, typically text editors on Windows (for cause of being the dominant OS, with 95% of the installed base) and Unix (for "true OS" religious reasons) only recognized the native endline format. You can see this at work if you try to view in Notepad[1] one of your Linux programs freshly unzipped from the archive. Other common problems are that the "same" text file has a different length on Windows and Unix or that the same C/C++ code is not guaranteed to give the same results on the different platforms. Finally, developers of file transfer clients had to offer different transfer mode: one for text file, in which end of line characters had to be translated, and one for "binary" files in which inserting a character `0x0A` before every occurrence of a `0x0D` could corrupt images, videos, etc.

Since then, Mac OS X (and now macOS and iOS) being a Unix system, Apple has switched to LF as its supported end of line format. So we are pretty much down to two formats: LF vs. CR+LF. This is the problem that you are going to address in this assignment. We are going to provide some text files (representing "images" and "patterns" to search for in the images) to be used as input. All files will be in the Windows format, and your script will "translate" the files into the Unix LF end of line format.

2.4.2 What the script should do

Your script should be named `eolFix.sh` and reside in the `Scripts` folders (and will be run from the project's root folder). It should take as arguments:

- the path to the directory containing a number of "data" files with problematic eol characters;
- the path to a "scrap folder" where to store any temporary files that your script may need to create;
- the path to the directory where to write the "fixed" data files with the proper line ending characters. Be careful that the user may choose the same folder as the input folder (this is the main purpose of the "scrap folder").

[1] Of course, you all have installed on your PC a decent text editor, Notepad++ or better, so you would never open a text file by default in Notepad, I am sure.

The files in the input directory are encoded using Windows line endings. The script must replace these line endings with Unix line endings. If your script creates temporary files, then it should create them in the indicated "scrap folder" (and create the scrap folder first in that case). At the end of execution, the script should delete the scrap folder and its contents if it created it. If you don't need a scrap folder, then there is nothing to create and delete.

2.4.3 What the script should *not* do

There exist utilities that allow you to perform the end-of-line fix on an entire file with a single-line command. You are not allowed to use these commands. Again, I want to force you to write a loop in `bash` to iterate through every line of a text file, because this is an fundamental building block of `bash` programming. If you also happen to know a few extra convenient commands, this is great, but you won't always find the "magic one-line command that solves exactly the problem." And in that case, you will have to be able to roll your own loop.

3 Part II: (Final) Organization of the C++ Program(s)

The task to perform can be decomposed into three major subtasks:

1. Data distribution: The files found in the data folder are distributed among processes;
2. Data processing: Once each process has a list of data files to work on, it creates a child "processor" and assigns it to work on the data files;
3. Integration: When the processors have finished their work, they report their result to their parent process, which in turn report it to the grand-parent server process, which outputs the collated results.

3.1 Data distribution

3.1.1 The parent/server process

This process is the root process for our problem. It is launched by the script and receives as arguments the path to the data folder and the number of processes to create.

It is going to create the required number of child "distributor" processes, set up proper communication channels with them (not necessarily in this order), and then handle communications with the distributor processes and *between* the distributor processes.

3.1.2 The "distributor" client processes

These are the child processes created by the server process. Each distributor gets a set of data files (defined by a range in the catalog list) and gets on to determine which process this data file was destined for. Figure 2 summarizes the situation from the point of view of distributor process `d0`. On the right side of the figure, we see the catalog list, with an index assigned to each file. We also get a peek into the content of each data file so that which process this file should be processed by. Each distributor process gets a range of file indices to sort and distribute. For example, we can see

that Distributor `d0` will be sorting the files with indices 0 to 5. Among those, it is only going to "keep" the second file (index 1), all the others being assigned to another process (e.g. the file with index 0 should be processed by Distributor `d2`.)

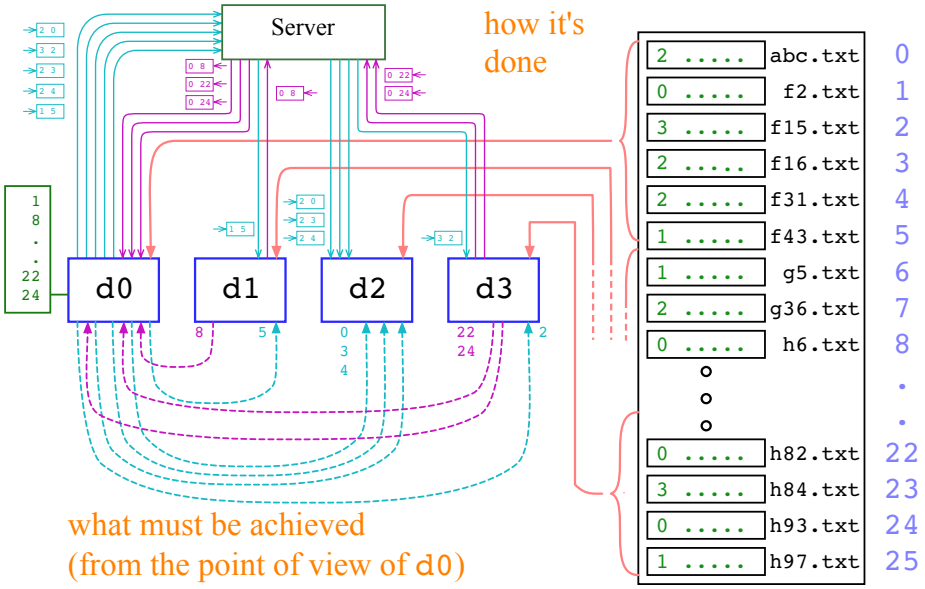

Figure 2: The data distribution task.

The dashed arrows in the lower half of the figure summarize what should be accomplished from the point of view of `d0`. It must inform If the data file is one of those it was supposed to process, it ads the file's name/id to its "to do list." If the file was destined for another process, it sends a message to the server with the index of the distributor process to which the message should be forwarded and the index of the data file that the destination process should add to its own "to do list."

The plain arrows in the top part of the figure show how this "passing around" of information is implemented. It would be absurd (and very complex) to establish 2-way communication between all pairs of processes (that would be $n(n-1)$ pipes for n processes), so we use a client-server architecture: All communications go through the server. In Figure 2, we see that the first data file that Distributor 0 looks at should be processed by Process 2. Therefore, it sends to the server the message `2 0` (indicated by a small blue box next to an arrow going from `d0` to the server). The server forwards the message to Distributor 2, and we see the message in a small blue box next to an arrow arriving to `d2`[2]. Conversely, while going through its own list of files, `d1` finds that that the File 8 should be processed by Process 0, and we see the message mbox`0 8` in a magenta box next to arrows going from `d2` to the server and from the server to `d0` respectively. When `d0` receives the message, it adds file Index 8 to its "to do list" represented by the green box next to `d0`.

[2]One could argue that the index 2 is superfluous in the message sent by the server to `d2`. On the other hand we could this as an opportunity for a verification that the wrong message didn't accidentally get sent to the wrong process.

3.2 Data processing

3.2.1 Data processing processes

At this point, we could simply say that the distributor processes, having compiled the list of files that must be processed, would do the processing themselves. Instead, we are going to add another layer of process creation[3]. The distributor processes are therefore each going to create a "processor" child process and give it the list of files to process (see Figure 3).

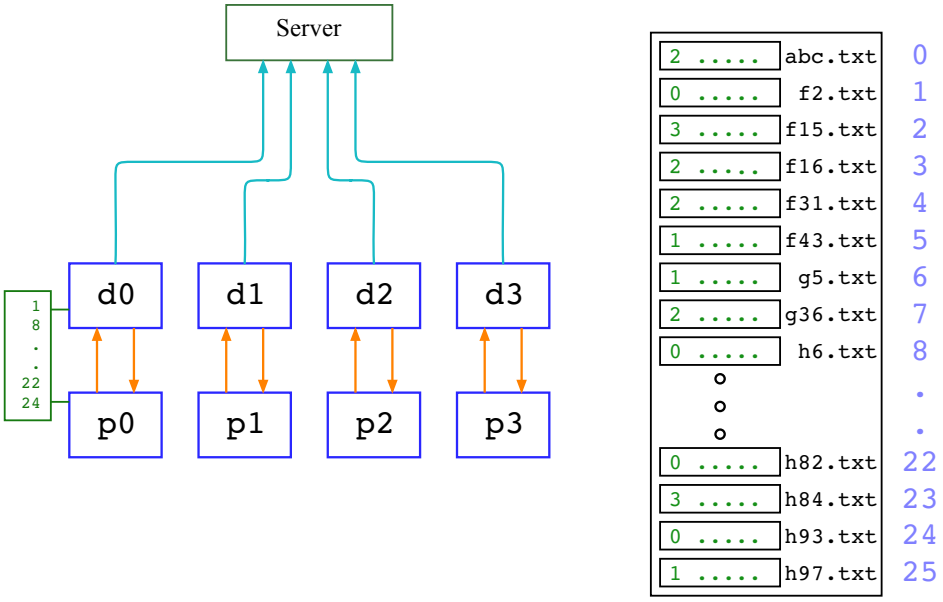

Figure 3: The data processing task.

3.2.2 Data files

The data processing itself is not the main purpose of the assignment, so I want to keep it as simple as possible. On the other hand, I want you to compute *something*, and to have a simple way to verify that your system works as expected.

For this dual purpose, I build my data files out of a single C source file. Let's for example consider the small "Hello World!" program of Figure 4. I decided that two processes would be sufficient for this task, and that the first half of the program should be handled by Process 0 and the second half by Process 1. Next, I prefix each line by the index of its designated processor and by the line number itself, then I save it as a separate, randomly-named text file (see Figure 5). Please note that the data files contain the end of line character at the end of each line of code.

[3]The rationalization of this added layer (which, quite frankly, exists for the sole purpose of making use an `exec`-family system call) would be that one could build a new data processing module and put it in place to replace the old one without changing a line to the client/server programs.

```
//  Hello  -----  main.c
//  Created by Jean-Yves Hervé on 2017-10-20.

#include <stdio.h>
                                                    Process 0
int main(int argc, const char * argv[])
{
    printf("Hello, World!\n");
    printf("CSC412 - Programming assignment 03\n");
    return 0;
}
                                                    Process 1
```

Figure 4: The source file used to generate a data set.

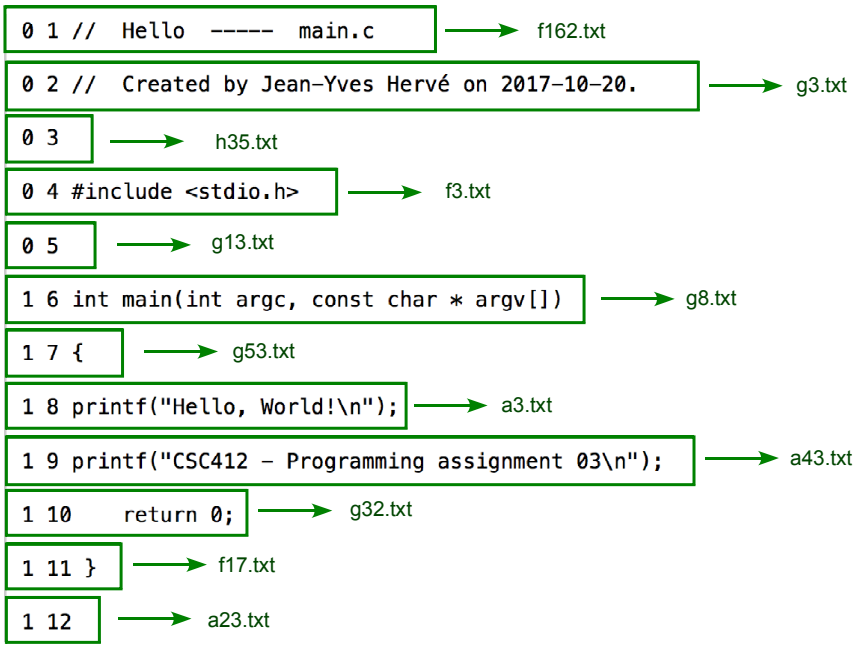

Figure 5: The source file exported as a set of data files.

3.2.3 What the processors should do

If you followed me until now, each processor must put into the right order the consecutive lines of a block of code. Since each data file contains the original line number in the code, this is a very simple sorting application. I am not asking you to implement your own sorter, but instead to use the built-in `sort` function of the standard C++ library[4]. All you have to do is implement the *comparator* to pass along to `sort`.

When `sort` has finished its task, your processor can put back together the block of code it received in pieces. Don't forget to remove the process index and line number from the strings

[4]It's good that you know how to implement various sorting algorithms, but at this point, if a functionality exists in the standard library, use it.

before putting back the code fragment together.

Once the code block has been reconstructed, the processor has finished its task: It passes back the reconstructed code block to its parent process (the distributor process with the same index), then terminates.

3.3 Putting it all back together

The distributor processes, upon receiving the reconstructed code block from their child process, pass that data back to the server process, and terminate. The server process puts the different blocks together (in the proper order) and outputs the reconstructed source file. If all went well, then you should be able to `gcc` the output file.

Data validation Depending how the user of your script may produce the data folder's path, the string passed as argument to your program may or may not end with a `/` character. You have to check for that and, if the `/` is missing, append the character to the path before sending it to the distributor processes.

Similarly, your must verify whether the name of the output file has a `.c` extension, and add one if it is missing.

4 Development of the C++ Programs

Because this assignment may be somewhat overwhelming, I am going to break down the development into several major steps that you will submit as separate versions of your assignment. As you implement the early versions, please keep in mind where this is all headed (what the final implementation's architecture looks like) so as to avoid making design decisions that are going to "paint you in a corner," coding-wise.

4.1 Version 1: One process does all the work

This is the most dangerous step, because many of you will have to fight their natural tendency to write all their code into one big function. The more modular you write it, with little to no reliance on global variables, the easier it will be to transition to a multi-process version (and the less penalized you will be on the "quality of code" rubric).
Let's review what your program should do:

- It receives as arguments the number n of child processes to—eventually—create and the path to the data directory.
- It splits the work into n more or less equal parts: It determines that (future) Child number k will process the files from Index i_{inf} to Index i_{sup} on the file list.
- Instead of creating a child process to work on the files (that will come in the next version), it calls a function that processes the files. That function is going to perform first the work of the distributor processes of Subsection 3.1.2. In other words, it is going to build n lists

of files: First the files that it should be in charge of, then a list of files for its of its—future siblings.

- The The main function gets all these lists back from the n function calls. It then calls n times the "data processing" function, passing it the n partial lists of files it is supposed to process. The function reorders the fragments and returns the result as a long string that reconstructs the part of the source file it was working with.
- The main function simply concatenates the n parts of the file and outputs the reconstituted source file.

4.2 Version 2: $2n$ child processes, no `execXY` call

In this version, your code still comes under the form of a single source file, but we are going to make `fork()` calls to delegate the processing to child processes. Your main process now does the following:

- It receives as arguments the number n of child processes to create and the path to the data directory.
- It splits the work into n more or less equal parts: It determines that Child number k will process the files from Index i_{inf} to Index i_{sup} on the file list.
- It makes n calls to `fork()` to create a child process. The child process makes a call to a function that does only half of the work of the one you developed for Version 1: it is going to build n lists of files: First the files that it should be in charge of, then a list of files for its of its—future siblings. Before terminating, it is going to write these lists to one or more files (your choice).
- The parent process waits for its n children to finish. Then it is going to create again n child processes that will do the actual data processing. How each "data processing" child process is informed of which data files it should work with is up to you. A "data processing" child process reorders the fragments it is working with and writes its reconstructed source part as a text file.
- The parent process waits for its second generation of n children to finish. It reads the n files produced by its children and simply concatenates the n parts of the file and outputs the reconstituted source file.

4.3 Version 3: $2n$ child processes, with `execXY` calls

This version only differs from the previous one in that the code of "distribution" and "processing" child processes is now going to be written into separate programs, each with its own `main` function, and that these programs will be launched by a system call to a function of the `execXY` family of functions.

4.4 Version 4: n child and n grandchild processes, some pipes

This version constitutes a bigger jump from the previous one. The root, parent process only creates the n "distributor" processes, and these in turn each create one "processor" child process. Distributor processes still communicate to each other (to pass lists of data files) by writing to text files. One problem you need to address now is how/when a distributor (or processor?) will know when its siblings have finished writing and the lists of data files can be read.

Each processor process writes in a text file the code fragment that it has reconstructed, and then terminates its execution. Its parent process (the corresponding distributor process), upon detecting that its child process has terminated, reads that file, and sends its content to the root process via a pipe. The root process collects and concatenate together all the pieces of the reconstructed file, and writes the reconstituted file.

4.5 Version 5: All communications between processes are done via pipes

This is the final version of the project, conform to the architecture presented in Subsection 3.1.2, and in particular in Figure 2.

Advice
It should be obvious that during the distribution phase, exchanges between the server and the distributors should take place using non-blocking `read` calls. Other than that, we don't tell you whether you should be using named or unnamed pipes. The decision is up to you; just make sure to justify it in your report.

4.6 Extra credit (up to 8 points)

I don't anticipate producing for this assignment a data set that would challenge the capacity of pipes on a modern OS, but pipe capacity would definitely be an issue with a more realistic processing side of the assignment (e.g., as I said in class, if the application was one of map building out of local topographical surveys). To get the extra credit points, set in your program a—small—hard limit for the size of data blocks to pass along the pipes, and fragment your messages in blocks that don't exceed that size.

4.7 More extra credit on the way

I will post on BrightSpace more opportunities for extra credit as questions start arriving on the discussion boards.

5 What to submit

5.1 `Script` folder

This folder should only contain the bash scripts that you wrote for this assignment. In case there are peculiarities of your scripts that you would like to attract our attention to, feel free to add a `README.txt` file to explain that.

5.2 `Propgrams` folder

This folder should contain a subfolder named `Version n` (for $n = 1 \ldots 5$ for each version that you implemented. That folder should contain all the source and header files necessary to build your executable for that version.

5.3 Report

Along with the source code of the C++ program and shell script, you should submit a report in which you explain your various design decisions regarding named vs. unnamed pipes and blocking vs. non-blocking read calls that were not already discussed in your **mandatory** javadoc-style comments.

Identify any current limitation of your code: Can it crash? Under what circumstances? Do you make any assumptions in addition to the ones allowed in this assignment?

5.4 Grading

- C++ program execution: 40%
  - Version 1: 10%
  - Version 2: 8%
  - Version 3: 7%
  - Version 4: 7%
  - Version 5: 8%
- `bash` script: 15%
- Report: 15%
- Code design: 15%
- Code quality (readability, identifiers, indentation, etc.): 15%

Various penalties:

- Folder organization (incl. name): 10 points
- File formats: 10 points