CSC 412 – Operating Systems

Programming Assignment 05, Fall 2023

Friday, October 27th, 2023

Due date: Friday, November 10th, 11:59spm.

1 What This Assignment Is About

1.1 Objectives

The objectives of this assignment are for you to

- Get more experience with process creation in Unix;
- Start using pipes between processes, and between a script and a process.

This is an individual assignment. The programming language for this assignment is C++ (more specifically, C++20).

1.2 Handout

The handouts for this assignment are:

- this document as a `.pdf` file;
- a set of elevation maps.

2 The Steepest Descent Problem

2.1 General idea of the problem

Our data for this assignment consists in a 2D grid representing an elevation map (the altitude at each point on a grid).

Now, imagine that we can drop a skier at some random location on the grid. From that point, our valiant skier moves to the neighboring locating (using 8-connect[1], and once there moves again to the lowest neighbor, and keeps doing so until the skier's location is lower than that of all of its neighbors (in mathematical terms, we would say that the skier has reached a local minimum of the

[1] 8-connect is one of the two classical ways to define a neighborhood on a rectangular grid (the other being 4-connect). In our case, what it means is that a move is possible from a given square to any of its eight immediate neighbors (of course the number will be lower, 3 to 5 if the square lies on an edge of the map), allowing for diagonal displacements.

elevation function). At this point, the skier should report its final location and the elevation at that point and the path followed to reach that point. The central dispatcher could then arrange for the skier to be picked up and dropped to a new starting point, from which to repeat the procedure of going down to a new local minimum.

I am sure you see where this is headed: We are going to implement our dispatcher as a parent process and the skiers as its child processes.

2.2 A few examples of trails

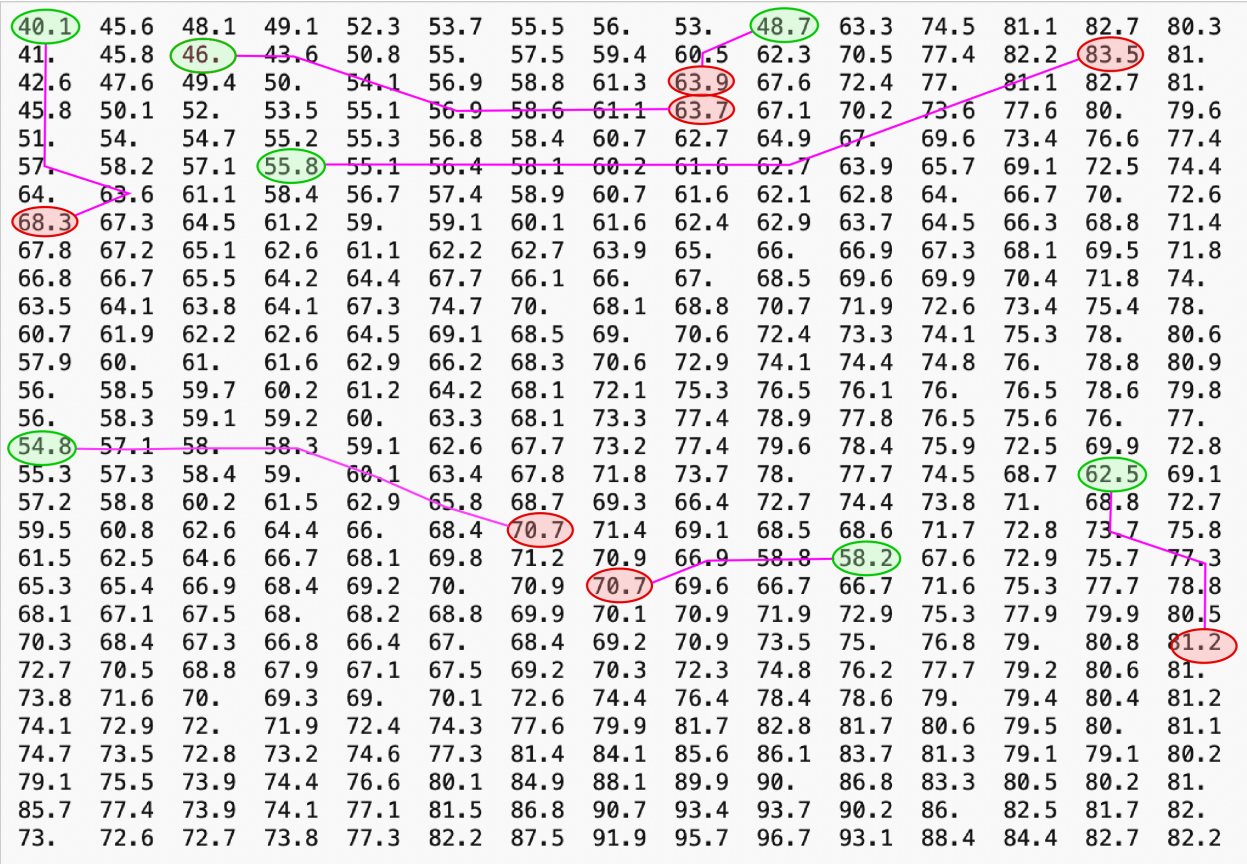

Figure 1: Examples of skier trails on the "map."

Figure 1 gives examples of skier trails obtained for different starting points on the map. Starting points are indicated by a red disk. The trails followed by different skiers are indicated as magenta-colored lines. All trails eventually end up at a point on the map that is lower than all the surrounding points, and there the skier's descent ends. Such endpoints are indicated by a green disk. As you can observe on the plot, two start points next to each other may give trails leading to very different endpoints.

2.3 Different sizes of maps

Figure 2 shows examples of elevation maps of different sizes displayed as gray-level images and as 3D surfaces. There is a family air between these different maps. It is an artifact of the algorithm that used to generate the maps (in case you care, it's a tweaked version of the Perlin noise generator, in which I had to force the presence of a few peaks and valleys for small-sized maps).

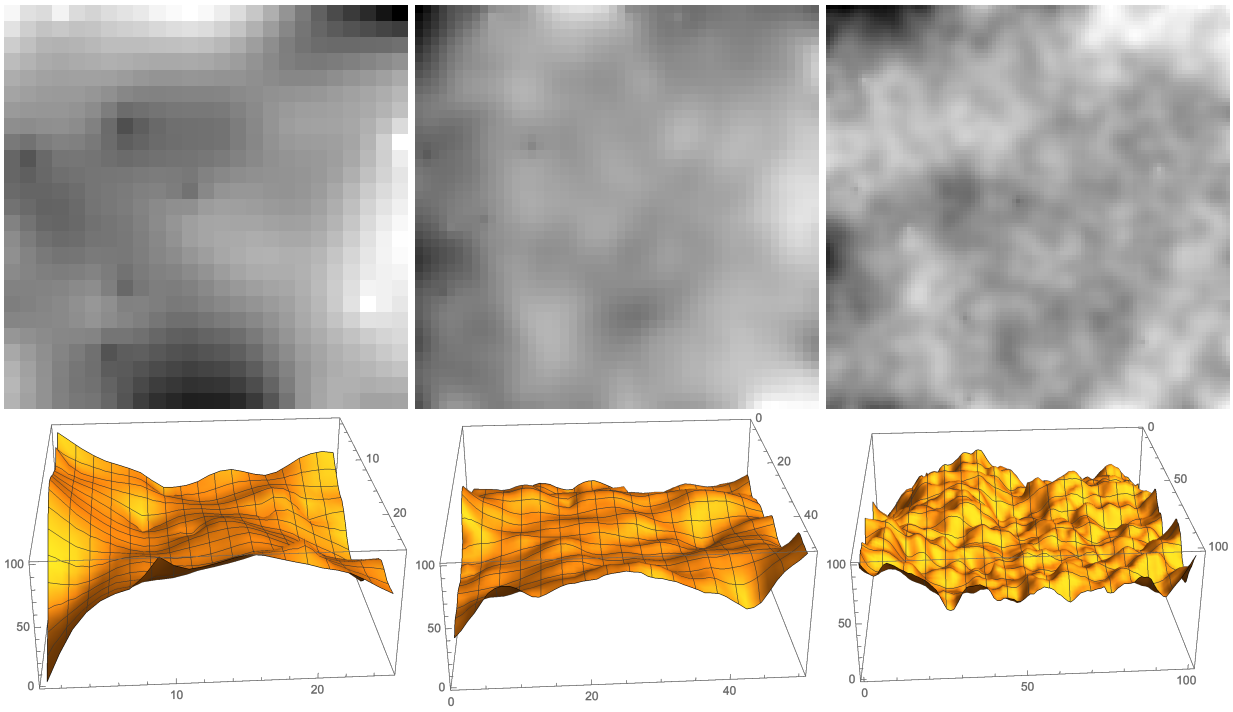

Figure 2: Examples of elevation maps of sizes 25×25, 50×50 and 100×100, respectively.

2.4 Map file format

The map files have the extension `.map`, but they are just plain text files. The file start with a line specifying the height and width of the map. The following lines (one per row of the map) list elevation values (a float number). Some maps are square (`_Sq`) suffix, others may be wider than they are high (`_Horiz` suffix) or higher than they are wide (`_Vert` suffix), but the naming of the files (other than the file extension) is not part of the format, just a convenience for the assignment. As with all grid-based problem that don't absolutely require a square grid (e.g. games such as chess, checkers, or go), it is preferable to use mostly non-square grids so as to maximize your chances of exposing and spotting erroneous row-column order swap in your code.

I also post pdf file representing the map as gray-level images or surface plots, in case you want to check if the paths of your skiers make sense.

3 C++ Implementation

All versions discussed below should reside in a folder named `Version1`, `Version2`, etc., residing inside the `Programs` subfolder of your root project folder.

3.1 Version 1: Single process

This version only exists to force you to get the basic functionality working before you start messing with forking and pipes. Really, you should think of it on your own, but many of you seem to be allergic to "wasting time" on temporary "scaffolding" code that helps complete the final product.

3.1.1 Arguments

This version will take as arguments

- Optionally, the flag `-t` indicating that program should run in "trace" mode (see below);
- The path to the map file to process;
- The row and column indices of a start point for a skier (in that order);
- The path to an output folder.

For example
```
./prog05v1 Data/smallMap_Vert.map 3 6 ./Output/
```
or
```
./prog05v1 -t Data/smallMap_Vert.map 3 6 ./Output
```

Note: In this assignment, we will use 1-based indices for rows and columns. So the top-left point of the grid (and therefore the first elevation value in the map file) will have row index 1 and column index 1.

3.1.2 Output of the program

The program should output a single text file named after the map file, but with the extension `.txt`. For example, in the case of the arguments given above, the program would output the file `./Output/smallMap_Vert.txt`. The content of that file would consist of:

- On the first line, the string `TRACE` or `NO_TRACE` indicating the mode in which the program was run.
- Then a line giving the row and column indices of the start point, the row and column indices of the final point, and the number of points on the path (including both endpoints), all separated by a single space character.
- If in trace mode, a line giving the row and column indices of all points on the path, all separated by a single space character.

For example, in the case of the map shown in Figure 1, if the start point was at row 8 and column 1, then the output file in no-trace mode would contain

```
NO_TRACE
8 1 1 1 8
```

If in trace mode, that file would contain

```
TRACE
8 1 1 1 8
8 1 7 2 6 1 5 1 4 1 3 1 2 1 1 1
```

If the start point was at row 19 and column 7, then the output file in no-trace mode would contain

```
NO_TRACE
19 7 16 1 7
```

If in trace mode, that file would contain

```
TRACE
19 7 16 1 7
19 7 18 6 17 5 16 4 16 3 16 2 16 1
```

3.1.3 Error handling

1. If the output folder doesn't exist, the program won't produce any output file and will terminate with exit code 11;
2. If the map file doesn't exist, the program will produce an output file that only contains the string `File not found` and with exit with code 12.
3. If the map file exists but the start point is not within the map, the program will produce an output file that only contains a string telling that the start point is invalid and exit with code 13. For example, in the case of the map of Figure 1, executing the program with a start point at row 40 and column 5 would produce the output file `Start point at row=40, column=5 is invalid`.

We won't deal with the case of invalid map files. All the data files that we will feed your program will be legit.

3.2 Version 2: Skier dispatcher and skiers (no pipe)

We add forking to this version, with multiple startpoint provided.

3.2.1 Arguments

This version of the program will take as arguments

- Optionally, the flag `-t` indicating that program should run in "trace" mode (see below);
- The path to the map file to process;
- The row and column indices of at least one start point for a skier (in that order);
- The path to an output folder.

For example
```
./prog05v1 Data/smallMap_Vert.map 8 1 19 7 3 9 ./Output/
```
or
```
./prog05v1 -t Data/smallMap_Vert.map 8 1 19 7 3 9 ./Output
```

both of which give three starting points $(S_1(r = 8, c = 1), S_2(r = 19, c = 7), S_3(r = 3, c = 9))$ for a skier's descent.

3.2.2 What the parent, "skier dispatcher" process does

We want each skier run, corresponding to a particular starting point, to be performed by a separate process. Reading the data file can take a significant amount of time, particularly when the map is large. It is therefore something that we would rather not do more often than needed. This means that we should read the map in the parent process *before* the call to `fork()`, and then keep the data in the child "skier" process. This means that we will not be making an `execXY()` call, and therefore that the parent "skier dispatcher" process and "skier" child processes will run off the same program (but of course, different sections of that program).

Each skier process will produce an output file (the file format, naming convention, and location are up to you). The parent process will read these output files to produce the output file for the map.

3.2.3 Output file

This is the file that will end up in the output folder. Its format is just an extension of that of Version 1. For example, in the case of the arguments given above, the program would output the file `./Output/smallMap_Vert.txt`. The content of that file would consist of:

- On the first line, the string `TRACE` or `NO_TRACE` indicating the mode in which the program was run.
- Then a line telling how many skier runs were performed.
- Then, for each skier run
  - A line giving the row and column indices of the start point, the row and column indices of the final point, and the number of points on the path (including both endpoints), all separated by a single space character;

- If in trace mode, a line giving the row and column indices of all points on the path, all separated by a single space character.

For example, in the case of the argument list given above, the output file in no-trace mode would contain

```
NO_TRACE
3
8 1 1 1 8
19 7 16 1 7
3 9 1 10 3
```

If in trace mode, that file would contain

```
TRACE
3
8 1 1 1 8
8 1 7 2 6 1 5 1 4 1 3 1 2 1 1 1
19 7 16 1 7
19 7 18 6 17 5 16 4 16 3 16 2 16 1
3 9 1 10 3
3 9 2 9 1 10
```

3.2.4 Error handling

Error handling is identical to that of Version 1 in the case of a missing output folder or map file. If some of the start points are invalid, then the output file will report this as well. For example, for the following list of arguments

```
./prog05v2 Data/smallMap_Vert.map 8 1 40 5 3 9 ./Output/
```

the following output file would be produced in trace mode:

```
TRACE
3
8 1 1 1 8
8 1 7 2 6 1 5 1 4 1 3 1 2 1 1 1
Start point at row=40, column=5 is invalid
3 9 1 10 3
3 9 2 9 1 10
```

3.2.5 Implementation constraints and advice

Since the next versions will use pipes, which for most purposes work like files, it probably would be a good idea to use C-style syntax for opening files and writing/reading them.

That was for the advice part. Now, for constraints, your program should not litter the project folder with junk files. Definitely, by the time the skier dispatcher terminates, the output files

produced by the individual skiers should have been deleted. Back on the advice track, it's also not a good sight if your project folder suddenly fills with a buch of junk files, even just temporarily. So maybe consider creating these files in a separate *temporary* folder. And before I get asked: Yes, it's your job to make sure that these files don't overwrite each other.

3.3 Version 3: Skier dispatcher and pipes

This version will take the same arguments and will produce the same output (including error messages) as Version 2. The only difference between the two versions is how the job gets done: This version replaces the skier-produced temporary files by pipes. We are here in a configuration of forking with no `exec` call, and therefore unnamed pipes are called for.

3.4 Version 4: General dispatcher

This version of the program cannot work by itself: It is meant to work jointly with Version 3 of the script. So, be careful as you read this that you understand what is done by the script and what is done by the program.

3.4.1 What this version adds

In this version, skier dispatcher processes are not launched directly by the script, but by a "general dispatcher" process. The C++ program run by this general dispatcher is a new piece for this version. The system's architecture now looks as follows:

- The script builds the executables and launches the general dispatcher process.
- The script starts its folder watching behavior and sends to the general dispatcher the path to any map or task file that it finds, ignoring all others.
  - If the path is to a map file, then the general dispatcher launches a new skier dispatcher process.
  - If the path is to a task file, then the general dispatcher forwards the path to the appropriate skier dispatcher process.

thre

3.4.2 Argument

The general dispatcher program will take as its sole argument the path to an output folder.

3.4.3 Output

The output of the program is identical to that of Version 2 or Version 3, and is produced by the different skier dispatchers.

3.4.4 Development strategy

This version of the program needs to work together with Version 3 of the script (Subsection 4.5), and neither the program nor the script can run just by itself *as specified in the assignment*. It's probably not a good idea to try to develop them both at the same time. At least, I wouldn't try to do it. Rather, I would first go for some intermediate version of the script, of the program, or of both, knowing full well that the time "wasted" developing this intermediate, not-for-grade version is going to be considerably less than that attempting to tackle a "multiple balls up in the air" development problem.

I cannot force you to develop an intermediate version, and I won't tell you what it should be (this is the kind of thing that you must learn to do on your own, but of course we can give you feedback on the way). This is just my word of advice.

3.5 Extra credit

3.5.1 Extra Credit 1: Selective termination (8 pts)

This EC can be applied either to

- A combination of Version 2 of the script and either Version 2 or Version 3 of the C++ program. In that case, joint modification of script and program(s) will be required.
- A combination of Version 3 of the script and Version 4 of the C++ program. In that case, only the C++ program(s) should need modification.

I repeat this EC in Subsection 4.7.1, but the bonus points are for the functionality. There is no double dipping for the program *and* the script.

This version should be placed in a subfolder named `EC1` of the `Programs` folder, and the companion script (even if unmodified in the second option) should be named `script05EC1.sh`.

What this version adds is a more "granulated" form of process ending. We now distinguish to forms of `END` task file:

- `END ALL` produces the same result as in the original Version 4: the end of execution of all skier dispatchers, of the general dispatcher, and of the script (in this order).
- `END <name of map file>` results in the end of execution of the skier dispatcher for that particular map, while all other processes and the script keep running.
  - If the script has never encountered this map file, then it should print out
    `Unknown map: <map file name>`
  - If the script has encountered this map file before, but its skier dispatcher has been stopped, then it should print out
    `Skier dispatcher has already ended: <map file name>`

Don't forget to update the list of maps being processed.

3.5.2 Extra Credit 3: Skier dispatcher relaunch (8 pts)

This version should be placed in a subfolder named `EC3` of the `Programs` folder. This version allows the user to relaunch a skier dispatcher process for a given map after the previous one has terminated.

3.5.3 More EC

I may post an updated version of this document with more EC options (if I get ideas).

4 Scripting

4.1 Location

As usual, your script(s) will be located in the subfolder `Scripts` of the root `Prog05` folder, and will be executed from the project's root folder. The script(s) will implement the behaviors discussed in the next subsections. Make sure to understand what the final version of your script is meant to accomplish, so that you make the right design decisions as you work your way through the versions.

4.2 Build executables

All versions of your script will build executables for all the version (including EC) of the C++ program that you have *completed* (Don't try to build versions that cause compiling errors, though.).

4.3 Version 1

This version of the script should be named `script05v1.sh` and is meant to work with Version 2 or Version 3 of the program.

4.3.1 Data files

These files will be text files and will have the extension `.task`. A task file will contain

- On the first line, the string `TRACE` or `NO_TRACE` indicating the mode in which skiers should run.
- Then a line giving the path to a map file.
- Then a line telling how many skiers to launch
- Then, for each skier run a line giving the row and column indices of the start point, separated by a single space character.

An example of a task file for this version of the script would be

```
TRACE
./prog05v1 Data/smallMap_Vert.map
3
8 1
40 5
3 9
```

4.3.2 Argument list

The arguments of the script will be (in this order):

- The path to the data folder.
- The path to an output folder. If this folder doesn't already exists, you script should create it. If the folder already exists, then the script should empty the folder.

4.3.3 What the script does

The script looks for task files in the data folder, and will then launch a "skier dispatcher" for each of these task files, **to be run concurrently**. The script terminates when all tasks have completed.

4.3.4 Error handling

There is no special error handling for this script, except for the data folder not being found. All data files provided will be valid. In the basic version of this script (see next EC) there won't be two task files using the same map, so there is no risk of file overwriting.

4.3.5 Extra credit 2 (6 points)

(This EC is numbered 2 in order to give to the next one an index matching that of the C++ program it works with).

This version of `script05v1.sh` should be named `script05EC2.sh`. If different task files refer to the same map, then you have to "do something" so that the different output files don't overwrite one another. Note that this may involve making a change to the list of arguments of the C++ program. If that is the case, then create a new folder `EC2` for your modified Version 2 or Version 3 program.

4.4 Version 2

This version of the script should be named `script05v2.sh` and is meant to work with Version 2 or Version 3 of the program. This script uses the same type of "task files" as in script Version 1, with one addition: A task file could contain the single line

```
END
```

When the script encounters this task file, then it should send a termination command (syntax up to you) to all skier dispatcher proceses.

4.4.1 Watch folder

If you didn't complete this part of the previous assignment, this is your chance to get it right.

Your script should keep watch on a specific folder and process any file that the user drops in that folder. If the user drops a task file in the watch folder, then the script should launch a skier dispatcher process for this task and return to its watch behavior without waiting for the skier process to terminate. If the task file contains an `END` task, then the script should terminate, after making sure that all skier processes have terminated. Any other file (including folders) should be ignored.

4.4.2 Argument list

The arguments of the script will be (in this order):

- The path to the watch folder. If the folder doesn't really exist, then your script should create it. If the script should create it. If the folder already exists, the script should process all task files in the folder, if any, before starting on with its watch behavior.
- The path to an output folder. If this folder doesn't already exists, you script should create it. If the folder already exists, then the script should empty the folder.

4.4.3 Error handling

There is no special error handling for this script. All data files provided will be valid. In the basic version of this script (see next EC) there won't be two task files using the same map, so there is no risk of file overwriting.

4.5 Version 3 (complete system)

This version of the script should be named `script05v3.sh` and is meant to work with Version 4 of the program.

As I already said in Section 3.4, this version of the script is meant to work jointly with Version 4 of the program. So, be careful as you read this that you understand what is done by the script and what is done by the program.

- If the user dropped a new map file, then the script should send the path to that file to the "general dispatcher" program (which will then launch a "skier dispatcher" for the map).
- If the user dropped a "task file" for a known map, then the script should forward the list of drop points to the "general dispatcher" process. If the drop task concerns an unknown map, then the script should print out an error message.
- If the user dropped any other kind of file (including a folder), then the script should simply ignore that file.

The arguments for this script, which must be named `script05.sh`, are, **in this specific order**):

1. Optionally, the flag `-t` indicating that the skier processes should run in "trace" mode;
2. The path to a "watch folder." If the folder doesn't already exist, your script should create it;
3. The path to an output folder for the results of the skier processes.

Your script must be able to handle any proper path. You cannot use hard-coded paths; you cannot assume that the path is relative to the local folder (i.e. the path doesn't necessarily start with `./` or `../`; it could very well be a full, absolute path starting from the root of the file system `/`); the path may end with a `/`,... or not.

4.6 Watch folder and general behavior of the script

I repeat here what I already wrote when discussing Version 4 of the C++ program:

- The script builds the executables and launches the general dispatcher process.
- The script starts its folder watching behavior and sends to the general dispatcher the path to any map or task file that it finds, ignoring all others.
  - If the path is to a map file, then the general dispatcher launches a new skier dispatcher process.
  - If the path is to a task file, then the general dispatcher forwards the path to the appropriate skier dispatcher process.

4.7 Extra credit

4.7.1 Extra Credit 1: Selective termination (8 pts)

I repeat here what I already said in Subsection 3.5.1. Again, the full bonus is for the functionality. It won't apply twice for the program and the script.

This EC can be applied either to

- A combination of Version 2 of the script and either Version 2 or Version 3 of the C++ program. In that case, joint modification of script and program(s) will be required.
- A combination of Version 3 of the script and Version 4 of the C++ program. In that case, only the C++ program(s) should need modification.

What this version adds is a more "granulated" form of process ending. We now distinguish to forms of `END` task file:

- `END ALL` produces the same result as in the original Version 4: the end of execution of all skier dispatchers, of the general dispatcher, and of the script (in this order).
- `END <name of map file>` results in the end of execution of the skier dispatcher for that particular map, while all other processes and the script keep running.

- If the script has never encountered this map file, then it should print out
  `Unknown map: <map file name>`
- If the script has encountered this map file before, but its skier dispatcher has been stopped, then it should print out
  `Skier dispatcher has already ended: <map file name>`

Don't forget to update the list of maps being processed.

4.7.2 Extra credit 2: Enable/disable trace mode per map: (5 pts)

This EC script should be named `script05EC2.sh` and can only work with Version 4 or one of the EC version of the program. If the user enters `trace <map name>` (resp. `notrace <map name>`), then skier tracing should be enabled (resp. disabled (for that specific map), overriding what the task file says. If the map is unknown, then the script should print out an error message:
`unknown map: <map name>`

5 What to submit

5.1 The pieces

All your work should be organized inside a folder named `Prog05`. Inside this folder you should place:

- Your report;
- A folder named `Programs`, in which there should be a subfolder for all of the versions that you completed
- A folder named `Scripts` containing the scripts that you implemented;
- Your report.

Please note that we may test your program and script with data other than the ones that you provide, but at least we want to be able to test your code in the same conditions that you did.

5.2 What's in the report

First, the report must imperatively be a pdf file. The main sections I want to see in this report are:

- An explanation of the communications between script and parent process, and parent process and child processes (how and when communication was set up, what get passed around, in what format, etc.);
- If you attempted to do any extra credit work, a list of what you did and how much of it was completed;

- A discussion of possible difficulties encountered in this assignment[2].
- A discussion of possible current limitations of your C++ program and script. Is it possible to make them either crash or fail?

5.3 Grading

- The C++ program follows the specifications (process creation): 20 points
- The C++ program performs the task specified (execution): 35 points
- the `bash` script performs the task specified (execution): 15 points
- C++ and `bash` code quality (readability, indentation, identifiers, etc.): 15 points
- Report: 15 points

[2]Here, I don't mean "my laptop's battery is dying" or "I couldn't get a pizza delivered to my dorm room."