Assignment 1: File System Module
Variant: 59153
Student: Erika-Timea Albert

1 Assignment description

You are given the following binary file format, which we will call from now on **SF** (i.e. "**section file**") format. A SF file consists in two areas: a **header** and a **body**. The overall SF file structure is illustrated below. It can be noted that the file header is situated at the end of the file, after the file's body.

```
SF FILE STRUCTURE
BODY
HEADER
```

A SF file's header contains information that identifies the SF format and also describes the way that file's body should be read. Such information is organized as a sequence of fields, each field stored on a certain number of bytes and having a specific meaning. The header's structure is illustrated in the HEADER box below, specifying a name for each header's field and the number of bytes that field is stored on (separated by ": "). Some fields are just simple numbers (i.e. MAGIC, HEADER_SIZE, VERSION, and NR_OF_SECTIONS), while others (i.e. SECTION_HEADERS and SECTION_HEADER) have a more complex structure, containing their own sub-fields. Such, the SECTION_HEADERS is actually composed by a sequence of SECTION_HEADER fields (elements), each such field in its turn being composed by four sub-fields: SECT_NAME, SECT_TYPE, SECT_OFFSET, and SECT_SIZE.

```
HEADER
SECTION_HEADERS: NO_OF_SECTIONS * sizeof(SECTION_HEADER)
    SECTION_HEADER: 7 + 2 + 4 + 4
        SECT_NAME: 7
        SECT_TYPE: 2
        SECT_OFFSET: 4
        SECT_SIZE: 4
NO_OF_SECTIONS: 1
VERSION: 2
HEADER_SIZE: 2
MAGIC: 4
```

The SF file's body, basically a sequence of bytes, is organized as a collection of sections. A section consists in a sequence of SECT_SIZE consecutive bytes, starting from the byte at offset SECT_OFFSET in the SF file. Consecutive sections could not necessarily be placed one near the other. In other words, there could be bytes between two consecutive sections not belonging to any section described in the SF file's header. A SF file's section contains printable characters and special line-ending characters. We would say thus that they are text sections. Bytes between sections could contain any value. They are however of no relevance for anyone interpreting a SF file's contents. The box below illustrates two sections and their corresponding headers in a possible SF file.

PARTIAL SF FILE's HEADER AND BODY

```
Offset  Bytes
...     ...
1000:   1234567890
...     ...
...     ...
2000:   1234567890
...     ...
xxxx:   SECT_2 TYPE_2 2000 10
yyyy:   SECT_1 TYPE_1 1000 10
...     ...
```

The following restrictions apply to the values certain fields in a SF file could take:

- The MAGIC's value is "Z78m".
- VERSION's value must be between 75 and 161, including that values.
- The NO_OF_SECTIONS's value must be between 3 and 14, including that values.
- Valid SECT_TYPE's values are: 42 44 .
- Section lines are separated by the following sequence of line-ending bytes (values are hexadecimal): `0A`.

2 Assignment's requirements

Your are required to write a C program named "*a1.c*" that implements the following requirements.

2.1 Compilation and Running

Your C program must be **compiled with no error** in order to be accepted. A sample compilation command should look like the following:

Compilation Command

```
gcc -Wall a1.c -o a1
```

Warnings in the compilation process will trigger a 10% penalty to your final score.

When run, your resulted executable (we name it "*a1*") **must provide the minimum required functionality and expected results**, in order to be accepted. What such minimum means, will be defined below. The following box illustrates the way your program could be run. The options and parameters your program must accept and their meaning will be explained in the following sections.

Running Command
```
./a1 [OPTIONS] [PARAMETERS]
```

2.2 Variant output

Each student receives a slightly modified variant of the SF format and the assignment's requirements. For the first task, you should be able to output the identifier of your assignment variant, when your program will be run like below.

Display Variant Command
```
./a1 variant
```

Output Sample for Display Variant Command
```
59153
```

2.3 Listing files

When being given the "`list`" option, your program must display on the screen the names of some files in the directory whose path is specified with the command line "`path`" option in the form illustrated in the following box.

List Directory Command
```
./a1 list [recursive] <filtering_options> path=<dir_path>
```

Which file names must be displayed is determined based on the filtering criteria mentioned below. Each file name must be displayed on a different line, with no blank lines between valid file names, in the form illustrated by the box below. If no file complying the requested criteria is found, nothing but the "SUCCESS" string must be displayed.

Sample Output for List Directory Command (Success Case)
```
SUCCESS
file_name_1
file_name_2
file_name_3
...
```

If the "`recursive`" option is also specified, your program should traverse the entire sub-tree starting from the given directory, entering recursively in all the sub-folders, sub-sub-folders and so on. In the recursive case, the filtering criteria do not apply to the sub-folder names, but only to file names. Found file names must be displayed prepended by their relative path in the file tree starting from the given directory. The two boxes below illustrate the way the program could be run recursively and a possible output for that case.

Sample List Directory Command
```
./a1 list recursive path=test_root/test_dir/
```

Sample Output for List Directory Command (Success Case)

```
SUCCESS
file_name_1
file_name_2
subdir_1/file_name_1
subdir_1/subdir_1_1/file_name_2
subdir_2/file_name_3
...
```

In case an error is encountered, an error message followed by an explanation must be displayed, in the form illustrated by the following box.

Output for List Directory Command (Error Case)

```
ERROR
invalid directory path
```

The filtering criteria used to select the files whose names must be displayed on the screen are the following:

- File size in bytes must be greater than the value specified with the filtering option "`size_smaller=value`".
- File permissions must be those specified with the filtering option "`permissions`=perm_string", where `perm_string` must be in the format displayed by "*ls*" and "*stat*" commands, like for instance "`rwxrw-r--`".

2.4 Identifying and parsing section files

When being given the "`parse`" option, your program must check if the file whose path is specified with the command line "`path`" option complies or not the SF format. The way the program could be run for this is illustrated in the following box.

Check SF Format Command

```
./a1 parse path=<file_path>
```

The way the SF format compliance must be checked is based on the following criteria:

- The value of the magic field is the one mentioned before, i.e. "Z78m".
- The values of the file version must be one from the interval mentioned above, i.e. between 75 and 161, including that values.
- The number of sections must be between the mentioned values, i.e. 3 and 14, including that values.
- The existing sections' type must be only in the set mentioned above, i.e. 42 44 .

When all the above criteria are met, the file could be considered as being compliant with the SF format, even if some other inconsistencies could still be found in it, like the size of the file being less than at least one section's beginning (i.e. section's offset) or end (i.e. section's offset plus section's size), or having overlapping sections.

In case the given file does not comply the SF format, an error message must be displayed on the screen, followed by the failure reason mentioning the first field the checking failed on, in the form illustrated by the box below.

Sample Output for an Invalid SF File

```
ERROR
wrong magic|version|sect_nr|sect_types
```

In case the given file complies the SF format, some of its checked fields must be displayed on the screen in the form illustrated by the box below.

Sample Output for a Valid SF File

```
SUCCESS
version=<version_number>
nr_sections=<no_of_sections>
section1: <NAME_1> <TYPE_1> <SIZE_1>
section2: <NAME_2> <TYPE_2> <SIZE_2>
...
```

2.5 Working with sections

When being given the "`extract`" option, your program find and display some part of a certain section of a SF file. In particular, a single line must be extracted. The needed command line arguments are given using the "`path`", "`section`" and "`line`" options, for the file path, section number and line number respectively, as illustrated in the following box.

Extract Section Line Command

```
./a1 extract path=<file_path> section=<sect_nr> line=<line_nr>
```

In case an error is encountered, an error message followed by an explanation must be displayed, in the form illustrated by the following box.

Sample Output for Extract Section Line Command (Error Case)

```
ERROR
invalid file|section|line
```

In case the given file is of the SF format and the searched section and line are found, the result should be displayed as illustrated in the box below.

Sample Output for Extract Section Line Command (Success Case)

```
SUCCESS
<line_content>
```

The line numbers are counted from the section start, the first line being line 1.

The lines should be displayed in reversed order (from the last character to the first).

2.6 Sections filtering

When being given the "`findall`" option, your program must function similar to the case when run with the "`list recursive`" options, though it must search only for SF files that have at least 4 section(s) of type 44.

The way the program could be run for this is illustrated in the following box.

Find Certain SF Files Command

```
./a1 findall path=<dir_path>
```

In case an error is encountered, an error message followed by an explanation must be displayed, in the form illustrated by the following box.

Output for Find Certain SF Files Command (Error Case)

```
ERROR
invalid directory path
```

Normal output must be displayed in the form illustrated by the box below.

Output for Find Certain SF Files Command (Success Case)

```
SUCCESS
file_path_1
file_path_2
...
```

3 User Manual

3.1 Self Assessment

How to generate tests and run them.

How to interpret the test results.

3.2 Restrictions and Recommendations

You are required and restricted to use only the OS system calls, i.e. low-level functions, not higher-level one, in your entire solution, in all lab assignments. For instance, regarding the file accesses, you MUST use system calls like `open()`, `read()`, `write()` etc., but NOT higher-level functions like `fopen()`, `fgets()`, `fscanf()`, `fprintf()` etc. The only accepted exceptions from this requirement are the functions to read from `STDIN` or display to `STDOUT` / `STDERR`, like `scanf()`, `printf()`, `perror()` and functions for string manipulation and conversion like `sscanf()`, `snprintf()`.

For string tokenization (i.e. separate a string into elements based on specific separators, like spaces) we recommend you using the `strtok()` or `strtok_r()` functions.

Do not take code from others. Review the "Examination and Plagiarism Policy" in order to make clear aspects related to cheating and cooperation.

Try to follow a minimum coding style. Review our coding style recommendations.

3.3 Evaluation and Minimum Requirements

Your program will be evaluated automatically by a script similar to “`tester.py`” (provided in the same archive with this document). The first time you run “`tester.py`”, a folder with all the test cases will be generated randomly (this may take a couple of minutes). A file called “`tests.json`”, containing the input parameters and the desired output for each test will also be generated. You can check this file in order to see the desired output for failed tests.

The tester will use `valgrind` for running the tests, if found on the system. If not present, the tester will still work, but keep in mind that your score may be 10% lower due to the memory leaks penalty.

How to run the tester:

`Running the tester`

```
$ python tester.py
```

The optional argument “`novalgrind`” can be used to disable valgrind checking.

- If your program has compilation warnings, a 10% penalty is applied.
- If your program has memory leaks (found by the `valgrind` tool), a 10% penalty is applied.
- If your program doesn’t comply to the required coding style, a penalty up to 10% could be applied (decided by your instructor).

Note that the provided tests are not exactly the tests we will run on your provided solution. Do not write solutions displaying expected results based on testing file names. Besides, check on your code that the required functionality is completed and correctly provided as just running some set of tests does not necessarily cover all possible cases and prove the solution if perfect. It would thus be possible that some exceptional cases to be covered only be tests that we will run.