Assignment 3: Inter-Process Communication Mechanisms
Variant: 22768
Student: Andreea-Alexandra Stan

1 Assignment description

1.1 Review of the SF File Format

This assignment makes few references to the SF file format (i.e. "**section file**") described in the first assignment's requirements. Though, in order to avoid the need to look in two different papers, we include in this one the parts needed from the first assignment's description.

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

```
VERSION: 4
NO_OF_SECTIONS: 1
SECTION_HEADERS: NO_OF_SECTIONS * sizeof(SECTION_HEADER)
    SECTION_HEADER: 6 + 4 + 4 + 4
        SECT_NAME: 6
        SECT_TYPE: 4
        SECT_OFFSET: 4
        SECT_SIZE: 4
HEADER_SIZE: 2
MAGIC: 4
```

The meaning of each field is the following:

- The MAGIC field identifies the SF files. Its value is specifies below.
- The HEADER_SIZE field indicates the size of the SF files' header, excluding the MAGIC and HEADER_SIZE fields.
- The VERSION field identifies the version of the SF file format, presuming the SF format could be changed a bit from one version to another, though not the case in your assignment, even if the version numbers could be different between different SF files — see below.

- The NO_OF_SECTIONS field specifies the number of the next SECTION_HEADER elements, which are covered in the description above by the SECTION_HEADERS name (field).
- The SECTION_HEADER's sub-fields are either self explanatory (e.g. SECT_NAME or SECT_TYPE) or their meaning is explained below.

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 "yeEI".
- VERSION's value must be between 121 and 153, including that values.
- The NO_OF_SECTIONS's value must be between 4 and 17, including that values.
- Valid SECT_TYPE's values are: 20 53 21 17 97 43 97 .

1.2 Pipe-Based Communication Protocol

Your program is required to communicate with our testing program using named pipes. The names of the pipes and the meaning of transferred information will be described below. Here we just want to explain the communication protocol. Let's suppose for our purpose that the communication pipes are named PIPE_1 and PIPE_2.

Let's also consider the scenario in which one of the programs (e.g. our tester) must send on PIPE_1 a request for some action to the other program (e.g. yours). The request is like a message consisting in a sequence of more fields in the following format:

Request Message Format

```
<req_name> ... <number_param> ... <string_param> ...
```

Each field consists in a specific number of bytes. There are two types of fields:

1. **string-fields**, which consist in a variable-size sequence of characters (bytes whose values correspond to codes of printable characters). Each variable-size string-field is composed of two parts (sub-fields): **size** and **contents** of that string-field. The size part consists in just a byte, which specifies the number of bytes of the following contents part. The two parts come one after another, in what we consider a single string-field. This kind of structure of the string-fields is needed for the receiving process to know how many bytes to read from the pipe for a particular string-field.

In the following sections we will specify string-fields of particular messages by including them between quotes, like for instance `"CONNECT"`, `"SUCCESS"` and `"ERROR"`. However, you have to take care that the corresponding bytes (hexadecimal values) sent on the pipe should be "`07 43 4f 4e 4e 45 43 54`", "`07 53 55 43 43 45 53 53`" and "`05 45 52 52 4f 52`" respectively, for the three examples mentioned (spaces are used only for better visibility, though they are not sent on the pipe, the illustrated consecutive bytes coming immediately one after another), including both string size (first byte) and its contents (the following bytes).

2. **number fields**, which consist in the bytes needed to represent that number. We consider number-fields as "`unsigned int`" values, represented on "`sizeof(unsigned int)`" bytes. In the following sections we will specify number-fields of particular messages as numbers not included between quotes, like for instance 1234, 5678 etc. Take care though that even if they look like strings of figures, this is only for clarity reasons, on the pipe being sent their binary representation. Such that, for an illustrated 1234 number-field, the following four bytes (hexadecimal values) must be actually sent on the pipe: "`00 00 04 D2`" (we just show here the hexadecimal equivalent of the decimal 1234, not taking into account the little-endian aspects, which should be transparent to you).

Note, that different by the regular C strings, our string-fields are not terminated with `'\0'`. So, if you need working with them as NUL-terminated strings (e.g. displaying them on the screen with *printf* or copying them with *strncpy*), you have to terminated them explicitly with `'\0'`.

Also note that in the above message structure the space characters are used only for a better visibility, otherwise, bytes of two consecutive fields are not separated by other bytes. Similar remarks hold for the quotes characters (`"`) used to emphasize the string-fields and the "`...`" sequences, which just suggest that there could be other parameters of different types between the illustrated ones.

The request's name, i.e. "`<req_name>`", is a string-field, whose supported values are described below. A specific request could be followed by zero or more parameters of the two field types described above. Such, the "`<number_param>`" represents a possible number-field and "`<string_param>`" a string-field. The number and meaning of the parameters are described below in accordance to each supported request type.

Once the request message is read from the pipe by the receiving program and the corresponding request handled, a response must be sent back, containing the results of request handling. In our example scenario PIPE_2 is used for sending back the response message, whose structure is illustrated below:

Response Message Format

```
<req_name> <response_status> ... <number_param> ... <string_param> ...
```

We can note that the response message's structure is similar to that of the request messages, following the convention for its fields. The response message contains the corresponding request name for checking purposes, followed by a request handling status (given by the "`<response_status>`" string-field) and zero or more fields (number or string) needed to transfer back the results of handling a specific request. The supported values for each response message fields will be described below in relation to each specific request.

2 Assignment's requirements

Your are required to write a C program named "*a3.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 a3.c -o a3 -lrt
```

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

When run, your resulted executable (we name it "*a3*") **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.

Running Command

```
./a3
```

2.2 Pipe-Based Connection

Your program must establish a communication to our tester using two named pipes. In order to establish such a communication, your program must perform the following steps:

1. creates a pipe named "RESP_PIPE_22768";
2. opens for reading a pipe named "REQ_PIPE_22768", supposed to be created automatically by our testing program;
3. opens for writing the pipe "RESP_PIPE_22768" that was previously created at step 1;
4. writes the following request message on the "RESP_PIPE_22768" pipe

   Connection Request Message

   ```
   "CONNECT"
   ```

If all the above steps could be executed successfully, your program must display on the screen the following success message:

Connection Successful Message

```
SUCCESS
```

Otherwise, it must display an error message like that below.

Connection Failure Message

```
ERROR
cannot create the response pipe | cannot open the request pipe
```

In case of a successful connection, your program must execute a loop performing the following steps:

1. reads from the pipe "REQ_PIPE_22768" a request message sent by our testing program;
2. handles that request as described below for each supported request type;
3. writes back on the pipe "RESP_PIPE_22768" the result of handling the received request.

2.3 Ping Request

The request message looks like

Ping Request Message

```
"PING"
```

In response to a ping request, your program must simply send back the following response

Ping Response Message

```
"PING" "PONG" 22768
```

2.4 Request to Create a Shared Memory Region

The request message looks like

SHM Creation Request Message

```
"CREATE_SHM" 2407090
```

In response to getting such a request, your program must create a shared memory region of 2407090 bytes, using the name "/2EjJwK". The permission rights for that shared memory region must be "664". You must use POSIX functions for creating the shared memory area and adjusting its size. If successfully created, the shared memory region must be mapped by your program in its own virtual address space.

The response message indicate a success or a failure respectively in the following way

Successful SHM Creation Response Message

```
"CREATE_SHM" "SUCCESS"
```

Error SHM Creation Response Message

```
"CREATE_SHM" "ERROR"
```

2.5 Write to Shared Memory Request

The request message looks like

Write to SHM Request Message

```
"WRITE_TO_SHM" <offset> <value>
```

The fields "`offset`" and "`value`" are number-fields and indicate an offset and a value that must be written in the shared memory region at the specified offset (the type `unsigned int`).

Your program must validate if the given offset is inside the shared memory region (i.e. between 0 and 2407090)) and if all the bytes that should be written also correspond to offsets inside the shared memory region.

After writing in the shared memory, your program must send a response message indicating the success or failure of the write operation, in a format like those below.

Successful Write to SHM Response Message

```
"WRITE_TO_SHM" "SUCCESS"
```

Unsuccessful Write to SHM Response Message

```
"WRITE_TO_SHM" "ERROR"
```

2.6 Memory-Map File Request

The request message looks like

Map File Request Message

```
"MAP_FILE" <file_name>
```

Your program should map in memory the file whose name is given by the string-field "`<file_name>`" for reading.

After mapping in memory the requested file, your program must send a response message indicating the success or failure of that operation, in a format like those below respectively.

Successful Map File Response Message

```
"MAP_FILE" "SUCCESS"
```

Unsuccessful Map File Response Message

```
"MAP_FILE" "ERROR"
```

2.7 Read From File Offset Request

The request message looks like

Read From File Offset Request Message

```
"READ_FROM_FILE_OFFSET" <offset> <no_of_bytes>
```

Your program should read from the currently memory-mapped file "`<no_of_bytes>`" bytes from offset "`<offset>`".

If your program does not read from the memory-mapped file, i.e. from memory, but uses the `read()` function to read directly from the file, your implementation will be considered only partially corrected and some penalties applied (see below in Section 3.2).

The read bytes must be copied at the beginning of the shared memory region, before sending the response message back to the testing program.

The response message must have the following structure, corresponding to success or failure cases of request handling, in one of the two forms below respectively.

Successful Read From File Offset Response Message

```
"READ_FROM_FILE_OFFSET" "SUCCESS"
```

Unsuccessful Read From File Offset Response Message

```
"READ_FROM_FILE_OFFSET" "ERROR"
```

The read is successful if:

- there already exists a shared memory region,
- a file is already mapped in memory (or just opened, if you chose to work directly with the file), and
- the required offset added with the number of bytes to be read gives a number smaller than the file size.

2.8 Read From File Section Request

The request message looks like

Read From File Section Request Message

```
"READ_FROM_FILE_SECTION" <section_no> <offset> <no_of_bytes>
```

Your program should read from the requested "`<section_no>`" section of the currently memory-mapped file (considered a SF file) "`<no_of_bytes>`" bytes from the offset "`<offset>`" in that section. The "`<section_no>`" takes valid values between 1 and the number of sections of the memory-mapped SF file.

If your program does not read from the memory-mapped file, i.e. from memory, but uses the `read()` function to read directly from the file, your implementation will be considered only partially corrected and some penalties applied (see below in Section 3.2).

The read bytes must be copied at the beginning of the shared memory region, before sending the response message back to the testing program.

The response message must have the following structure, corresponding to success or failure cases of request handling, in one of the two forms below respectively.

Successful Read From File Section Response Message

```
"READ_FROM_FILE_OFFSET" "SUCCESS"
```

Unsuccessful Read From File Section Response Message

```
"READ_FROM_FILE_OFFSET" "ERROR"
```

2.9 Read From Logical Memory Space Offset Request

The request message looks like

Read From Logical Space Offset Request Message

```
"READ_FROM_LOGICAL_SPACE_OFFSET" <logical_offset> <no_of_bytes>
```

Your program should read from the currently memory-mapped file "`<no_of_bytes>`" bytes from an offset obtained by some calculations from the given "`<logical_offset>`". The "`<logical_offset>`" is an offset in what we call the file's "*logical memory space*". Such a logical memory space could be imagined as a contiguous memory area in the process' memory, whose contents could be obtained in the following way, assuming the memory-mapped file is a SF file:

- the SF file' sections should be loaded in your process' memory starting from a given address, considered the beginning of the SF file' logical memory space (note that the address itself is of no interest to us, but only the structure of the memory area starting from that address);
- the SF file's header should only be used to locate and read the SF file's sections, but otherwise will not be loaded into the corresponding logical memory space;
- each SF file' section must be loaded in the logical memory space to the next available multiple of 2048 bytes after the end of the previously loaded section;
- the SF file's sections must be considered in the order they appear in the SF file's header; note that this order could not necessarily be the same order the contents of the sections are placed in the SF file, i.e. the contents of a section whose header is before another section's header could be located in the SF file at an offset greater than that of that other section (so after it).

In order to make the explanation clearer, let's consider the case of a SF file having three sections of 2049, 20, and 2048 bytes respectively and the required alignment 2048. Supposing for simplification the starting address of the logical memory space as being 0 (zero), the sections would be placed in that logical space starting from the following offsets:

- 0, the first section;
- 4096, the second section;
- 6144, the third section.

Note however, that we did not mention in the above example any file offset of the considered sections. Obviously, such offsets are different by those in the logical memory space and should be taken from the SF file's header.

Also note that we do not really ask you creating the corresponding logical memory space of a SF file, but only to be able find (using some formula) for one particular byte in the logical memory space its corresponding byte in the SF file.

If your program does not read from the memory-mapped file, i.e. from memory, but uses the `read()` function to read directly from the file, your implementation will be considered only partially corrected and some penalties applied.

The read bytes must be copied at the beginning of the shared memory region, before sending the response message back to the testing program.

The response message must have the following structure, corresponding to success or failure cases of request handling, in one of the two forms below respectively.

Successful Read From Logical Space Offset Response Message

```
"READ_FROM_FILE_OFFSET" "SUCCESS"
```

Unsuccessful Read From Logical Space Offset Response Message

```
"READ_FROM_FILE_OFFSET" "ERROR"
```

2.10 Exit Request

The request message looks like

Exit Request Message

```
"EXIT"
```

When getting this type of message, your program must close the connection / request pipe, close and remove the response pipe and terminate.

3 User Manual

3.1 Self Assessment

In order to generate and run tests for your solution you could simply run the "tester.py" script provided with your assignment requirements.

Sample Command for Tests Generation

```
python3 tester.py
```

Even if the script could be run in MS Windows OS (and other OSes), we highly recommend running it in Linux, while this is how we run it by ourselves, when evaluating your solutions.

When running the script, it generates a "test_root" directory, containing all sort of files used to check your solution. Even if the contents of the "test_root" directory is randomly generated for each student, for the same student it is (with a high probability) the same, independently of how many times you run the script. The only difference could be between different OSes.

After creating the testing directory's contents, the "test_root" script also runs your program against it, displaying the results of the different tests it runs. The maximum grade (10) is gained when all tests pass and no penalty is applied (see below). The precise grade is calculated by scaling the number of points your solution gains due to test passing to the 0-10 scale.

Restrictions:

- 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()`;
- Regarding the access to SF files, as mentioned before you are restricted from reading from the file using the `read()` function, being required instead to map the file in memory and read directly from memory, when required. However, this is not a very strict restriction, though if you do not comply, your grade will be applied some penalties

Recommendations:

- 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.
- If it happens to observe an unexpected strange behavior of your program, like getting hanged, getting unexpected bytes from the pipe etc., this could be (among other reasons, like bugs in your program) because of some hanged processes from your previous tests. In order to be sure your program's behavior is not influence in this way, you could try executing the command "`killall -9 python3 a3`".
- If you want to get execution details of your program in order to debug it, you could set the "VERBOSE=True" option (i.e. change its default False value) in your "tester.py" file.

When your assignment is graded, it is run inside a *docker* container. While a correct and deterministic solution will behave in the same way, a bug in your solution may pass unnoticed on your system but may cause a crash / undefined behavior during the "official" evaluation. For this reason, we encourage you to also test your solution using docker before submitting it. To do this, you need to:

- install *docker* on your machine
- create a Docker Hub account and login to it from the command line (it is needed for downloading the evaluation image)
- install the *docker* module for Python

To test your solution using the *docker* container, you need to provide the *docker* argument to the testing script.

Sample Command for testing with Docker

```
python3 tester.py docker
```

3.2 Evaluation and Minimum Requirements

- If your program has compilation warnings, a 10% penalty is applied.
- If your program does not map the required file in memory and read its contents from the corresponding mapped memory area, but instead use the `read()`, a 30% penalty is applied for the corresponding tests.
- If your program doesn't comply to the required coding style, a penalty up to 10% is applied (decided by your instructor).

Do not cheat as we run plagiarism detection tools on all provided solutions.

Notes: The provided tests could not be 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.