CSCE A321: Operating Systems
Fall 2019. Homework Assignment 2
Due: 10/01/2019[1] 11:59PM AKDT
Group assignment

This second assignment is about utilizing some basic file system calls provided by a Linux/UNIX/POSIX environment in order to perform some basic file manipulation as well as some process creation.

It is to be done by the same group of 2 students who already worked together on the first homework assignment. Even though it is a team assignment, the instructor will determine each student's **individual** involvment. **Your grade will depend on your individual performance.**

For this assignment, you have to turn in an archive containing:

- the source file `head.c`,
- the source file `tail.c`,
- the source file `findlocationfast.c`,
- the source file `findlocationfastmemory.c`,
- the source file `pipeobserver.c` and
- your report (as a PDF) covering all tasks in all sections of this assignment.

The Linux manual pages, which you can open using the shell command e.g. `man 2 mmap`, are your friend. Read and understand them in their entirety for all system calls that are used in the programs to be written in this assignment.

Even though it is a call to a level-3 function, your programs may use calls to `strerror` for error messages. None of the programs is supposed to use `printf`, `fprintf` or `fputs`, as these are level-3 calls. You might want to start with the implementation of a function `int my_file_puts(int fd, const char *s);` which you base on the `write` system call (and your implementation of a function performing the task of `strlen`).

1 `head` and `tail`

As seen in class, the Linux/UNIX/POSIX `head` and `tail` programs allow the beginning resp. the end of a text file to be extracted.

In this section of the homework assignment, you are asked to re-program the Linux `head` and `tail` commands. Your programs must be written in C and must be based solely on POSIX file-system handling system calls (level-2 calls). This means, you can use `open`, `close`, `read` and `write` but calls to `fopen`, `fread` or `fprintf` are prohibited. You may use `malloc`, `calloc` and `free`; however, see the remark below.

[1] This date may seem far away in the future. However, there is quite some amount of code to write for this assignment. Start early! (i.e. now)

Your `head` and `tail` programs must handle the following use-cases and associated options:

- When run without any option nor filename argument, the `head` command must copy the first 10 lines of the input it receives on standard input (stdin) to standard output (stdout) and disregard any other lines seen on input. Similarly, the `tail` command must read the input on standard input and output only the last 10 lines of that input to standard output. Note that `head` can perform its task without any buffering (in memory), while `tail` needs to maintain a memory buffer of the current last 10 lines it just read in. When the input contains less than 10 lines, the input is completely copied from standard input to standard output.

- When run with a filename argument, the `head` and `tail` commands do not read from standard input but from the file designated by that filename argument. If they cannot open the file in read-mode, they display an appropriate error message on standard error and exit. Your `head` and `tail` commands just handle one filename argument, whereas the Linux/UNIX/POSIX programs handle several filename options.

- When run with the `-n [num]` option, the `head` and `tail` commands behave as specified above but replace the number of 10 lines by the appropriate amount specified by `[num]`. If that amount is zero, they do nothing. If the amount is negative, your `head` and `tail` programs do nothing either; the Linux/UNIX/POSIX commands execute a special behavior in this case. When the `-n` option is given incorrectly or incompletely, e.g. by specifying `-nose` instead of `-n` or by specifying `-n` without specifying a non-negative amount in the next argument, the `head` and `tail` commands display an appropriate error message on standard error and exit. The amount `[num]` is supposed to be less than or equal to $2^{64} - 1$; if it is larger, the behavior of the programs is unspecified. You are supposed to perform the string-to-integer conversion with code written on your own; not using `atoi` or `strtol`, which would be level-3 operations.

- The filename and `-n` options can be combined and used in any reasonable order, i.e. as `-n [num] [filename]` or as `[filename] -n [num]`.

- If no error condition arises, the `head` and `tail` commands return the *success* condition code, i.e. zero. If an error occurs, they return a non-zero *failure* condition code. Note that an error might occur for any system call, i.e. for any call to `read`, `write`, `malloc` etc. Before exiting, an error message must be displayed on standard error in these cases, too.

These are examples of well-formed calls to `head` and `tail`:

```
cat nanpa | ./head
cat nanpa | ./head -n 42
./head -n 42 nanpa
./head nanpa
./head nanpa -n 42
echo -e 'Hello\nworld!\nHave\na\nnice\nday.' | ./head
cat nanpa | ./tail
cat nanpa | ./tail -n 42
./tail -n 42 nanpa
./tail nanpa
```

```
./tail nanpa -n 42
echo -e 'Hello\nworld!\nHave\na\nnice\nday.' | ./tail
```

Your `head` and `tail` commands must not access any memory out of their addressing space and must not leak any memory. This means any memory allocated on the heap with `malloc` or `calloc` must be returned using `free`. All accesses to arrays must use correct indices; you may want to double-check on this. All files opened with `open` must be closed with `close`. These memory and file handling specifications must be met even though the operating system would perform these tasks automatically on process termination if they are forgotten. Your instructor will check for these points, using, amongst others, `strace` and `valgrind`. You may want to utilize these two tools for testing, too.

You must write C code (not C++). You are supposed to do some software engineering in order to end up with well structured source code (no spaghetti!). Your source code must contain some comments at the most decisive places, i.e. before functions to document their meaning and at places where the functioning of the code is not obvious to someone reading your code for the first time.

As the `head` and `tail` commands exist on the system you will be developing on, be sure to execute **your** `head` and `tail` implementations when testing!

For this section, you have to submit:

- The source file `head.c` which can be compiled to `head` using `gcc -Wall -O3 -o head head.c`.
- The source file `tail.c` which can be compiled to `tail` using `gcc -Wall -O3 -o tail tail.c`.
- A section in your report explaining the software engineering decisions you made, the problems you encountered and the testing you performed.

2 File searching

With the first homework assignment you worked on last week, you downloaded a file called `nanpa`. This file contains quite a comprehensive list of North American phone number prefixes, six digits long each, followed by a string describing the location served by phone numbers starting with this prefix. On each line, the string is right next to the six digit prefix. It is suffixed with spaces so it is always exactly 25 digits long. Each line is separated from the next line by one next-line character `'\n'`. Hence each line consists of exactly $6 + 25 + 1 = 32$ characters. The `nanpa` file is sorted by ascending prefixes. However, there are certain prefixes that do not correspond to any location: these prefixes do not figure in the file.

You saw last week that searching for a certain prefix in the `nanpa` file is possible in $\mathcal{O}(n)$ using the Linux/UNIX/POSIX tools `grep` and `sed`. However, this method is not optimal from a complexity standpoint: a look-up in an ordered array or file can be done in $\mathcal{O}(\log n)$.

For this section of the homework assignment, you must write, in C, a tool named `findlocationfast` that always takes two arguments: a filename and a prefix. The filename corresponds to a file organized like the `nanpa` file. The prefix must consist of six digits, which are all numerical (`'0'` through `'9'`). The tool must open the file and search it for the location identified by the prefix specified by the second argument. If a corresponding line is found, the line must be output on standard output, after stripping the superfluous spaces in the end (and, of course, without the prefix). The look-up must be performed in $\mathcal{O}(\log n)$ whenever the underlying filesystem allows this complexity to be reached; however, see the remark below.

When the look-up is successful, a zero exit condition code must be returned. When no line corresponding to the sought prefix figures in the file, the exit condition code must be non-zero; no error message is displayed in this case. If the file cannot be opened, the prefix received in argument is not well-formed (not exactly 6 digits, other characters than the digits `'0'` through `'9'`) or any other error occurs, an error message must be displayed on standard error and the tool must exit with a non-zero condition code. In any case, the file, once opened, must be closed before exiting. The tool must not perform any access outside its memory space in any case and must terminate execution with a proper error message (on standard error) and exit code, even if the file it is given to work on is not formatted like the `nanpa` file[2].

You are supposed to base your program only on the system calls `open`, `read`, `write`, `lseek` and `close`. You are not supposed to use level-3 buffered file manipulation calls, like `fopen`, `fread`, `fprintf`, `printf`, etc. Your tool may assume that the file to search is less than 2147483648 bytes in size. Your tool is not supposed to allocate any memory (using `malloc` or `calloc`); using statically sized buffers on the stack is fine. The tool is not supposed to use `strlen`, `strcmp`, `memset`, `memcpy` or `memmove`; these functions are trivial to implement: just implement your own.

Not all Linux/UNIX files and corresponding file descriptors are seekable, i.e. the `lseek` system call fails on them. Your tool may assume that the filename it is given in argument corresponds to a file that is seekable. However, the tool must **properly** exit with an appropriate error message (on standard error) and non-zero exit condition, if the file is not seekable. You are required to find an example with which this particular error can be triggered: report on this example in your write-up. Your `findlocation` script you wrote for the last homework assignment should work on that example. To help you, the instructor recommends looking into what the Linux/UNIX/POSIX command `mkfifo` does and working with the kind of "file" created with `mkfifo`.

Once your tool works correctly, test it on various examples. Also run it through `strace` and try to understand the output produced by `strace`. Based on what you see with `strace`, write some explanation why you are convinced that the look-ups take $\mathcal{O}(\log n)$ time on your typical (SSD) hard-disk and why the instructor should share this conviction. Then think of some storage technology whose ugly interface the Operating System will try to hide in order to allow you to use `open`, `read` and `lseek` for your search but for which the look-up will still be in $\mathcal{O}(n)$.

For this section, you have to submit:

- The source file `findlocationfast.c` which can be compiled to `findlocationfast` using `gcc -Wall -O3 -o findlocationfast findlocationfast.c`.
- A section in your report explaining the software engineering decisions you made, the problems you encountered, the testing you performed and the explanations you found.

3 Mapping files to memory

A typical POSIX-compatible environment supports the `mmap` system call. This system call allows for mapping the contents of a file on the filesystem, described by a file handle obtained with `open`, into memory: the `mmap` call returns a pointer which, when dereferenced, allows for reading the memory-mapped file by reading at the address given by the pointer, and for writing to the file, by writing to memory at the address provided by the pointer.

For this section of the homework assignment, you must write, in C, a tool named `findlocationfastmemory` which performs the same task as the tool `findlocation` you wrote

[2]Try executing it on various other files, like a JPEG picture of a cat, a MP3 file with a song etc.

for the last section of this homework assignment. The difference is that `findlocationfastmemory` uses `mmap` to map the used file into memory.

You are supposed to base your program only on the system calls `open`, `lseek`, `mmap`, `munmap` and `close`. You are not supposed to use level-3 buffered file manipulation calls, like `fopen`, `fread`, `fprintf` etc. Your tool may assume that the file to search is less than 2147483648 bytes in size. Your tool must not leak any memory, the memory-mapped region must be properly unmapped using `munmap`. All files opened must be properly closed. The status codes to be returned are the same as for `findlocationfast`. Your tool is not supposed to allocate any memory (using `malloc` or `calloc`) other than the memory obtained with `mmap`; using statically sized buffers on the stack is fine. The tool is not supposed to use `strlen`, `strcmp`, `memset`, `memcpy` or `memmove`; these functions are trivial to implement: just implement your own. The lookup algorithm must have $\mathcal{O}(\log n)$ time complexity. The tool may gracefully[3] fail on files that are non-regular or non-seekable.

For this section, you have to submit:

- The source file `findlocationfast.c` which can be compiled to `findlocationfast` using `gcc -Wall -O3 -o findlocationfastmemory findlocationfastmemory.c`.
- A section in your report explaining the software engineering decisions you made, the problems you encountered, the testing you performed and the explanations you found.

4 Pipe observer

As we have seen in class, a Linux/UNIX/POSIX process has two standard file descriptors, standard input and standard output. These file descriptors can be connected to a keyboard, a screen, a file, or, through a pipe, to the corresponding file descriptor of another process. For example, the following `bash` command launches two processes, one for `ls`, one for `wc`, and connects the standard output file descriptor of `ls` through a pipe to the standard input file descriptor of `wc`:

```
ls | wc
```

Typically, the information that travels through the pipe cannot be observed; it is consumed at the same time it is produced[4]. In the example above, the lines with filenames produced by `ls` and consumed by `wc` that traveled over the pipe are lost. If this information needs to be kept for further reference in a file, the `tee` tool can be used. This tool copies all information it receives on standard input to standard output, while duplicating it into a file whose name is given in argument. For example, in `bash`, this would lead to a statement like:

```
ls | tee allfiles | wc
```

Here, the information produced by `ls` is kept in the file `allfiles`. Note that there are at least three processes involved in the execution of that statement: one for `ls`, one for `tee`, one for `wc`. To these processes adds the `bash` process that executed the whole compound statement and that keeps running, waiting for its children (corresponding `ls`, `tee` and `wc`) to exit. Sometimes there might even be yet another process involved, controlling the second pipe between `tee` and `wc`.

Suppose the `bash` shell would not allow for such an arrangement. The following tool –let's name it `pipeobserver`– might be considered:

[3] with a proper error message (on standard out) and deallocating memory and closing all files.

[4] The Operating System provides for some buffering so the two processes do not necessarily have to write and read the same amount of information in a lock-step fashion. However, that buffer is transparent.

```
./pipeobserver allfiles [ ls ] [ wc ]
```

That tool `pipeobserver` hence takes several arguments; in the example, these are

1. the filename `allfiles`,
2. an opening bracket `[`,
3. the name of an executable to run, here `ls`,
4. an closing bracket `]`,
5. an opening bracket `[`,
6. the name of a second executable to run, here `wc`, and
7. a final closing bracket `]`.

Note that there are spaces between the different arguments, including the brackets, and remark that these spaces are required for `bash` to properly separate the arguments. In C, the `pipeobserver`'s `main` function hence would receive an `argc` argument with the value 8 and an `argv` argument containing, for example, in `argv[3]` the string `"ls"` and `argv[4]` the string `"]"`.

The bracket arguments are necessary as the two processes might have arguments and a means is required to separate the arguments of the one process for the other's arguments: For example,

```
./pipeobserver allfiles [ ls -lat ] [ wc -l ]
```

would also be a valid call; `ls` would receive the string `"-lat"` in argument, `wc` would receive `"-l"`. Note that brackets might also be valid arguments to the processes indicated: For example,

```
./pipeobserver allfiles [ echo [ Hello ] ] [ wc -l ]
```

would be a valid call, where `echo` receives three arguments, viz. `"["`, `"Hello"` and `"]"`. The `pipeobserver` tool can however look for brackets starting both from the left and the right, which allows it to find the separating bracket combination, `"]"` followed by `"["`, quite easily. In particular, this way, there are as many opening as closing brackets on each side[5].

For this section of the assignment, you are supposed to write, in C, this `pipeobserver` tool. The tool starts by checking if the first argument it receives is a filename it can execute `open` on to open a corresponding file for write access. It then parses its own arguments, separating them into a first executable's name with corresponding arguments and a second executable's name with corresponding arguments. If any of these operations (file opening, parsing etc.) fails, it displays a corresponding error message (on standard error) and exits with a non-zero exit condition.

The tool then calls `pipe` to obtain a new process pipe. It continues by creating a first child process of itself. This child process connects its standard output to the write end of the pipe it inherited from its parent (and closes the other end), using the `dup2` (and `close`) system call(s). The child then replaces itself by a process corresponding to the first executable to run, using the `execvp` system call for this replacement. It passes on the appropriate arguments.

After creation of its first child, the parent creates a second child. That child connects its standard input to the read end of the pipe it inherited (and closes the other end).

[5]There are funny cases when the bracketing is ambiguous. Your tool should take a guess there. The instructor will not try to run your tool on such strange cases.

The child then creates a second process pipe and continues by creating a first child of its own; we shall call this child of the child the grand-child A. The grand-child A connects its standard output to the write end of the pipe it got from its parent (and closes the other end). It then behaves like `tee`[6] on the file descriptor its grand-parent created for the indicated filename: it reads on standard input, using `read` and writes on standard output and the file descriptor using `write`. When encountering a end-of-file condition, the grand-child A stops copying data and closes the file descriptor it got from its grand-parent.

The child then creates a second child of its own; we shall call this child of the child the grand-child B. The grand-child B connects its standard input to the read end of the pipe it got from its parent (and closes the other end). It then replaces itself by the second executable to run, using again `execvp`.

The instructor recommends that you thoroughly understand this tree of processes that fork off children before starting to code. Make some drawings and discuss with your team partner. In the code, start with just the process tree structure and test, then only insert the calls to `execve`. You may use the Linux command line tool `pstree` to display the process tree.

Each process that creates children waits for all its children to exit before exiting itself, with a zero exit condition to signal success. If any system call fails, an appropriate error message is displayed on standard error and the corresponding process exits with a non-zero exit condition. Each process which inherits the file descriptor corresponding to the file whose name is given as a first argument to `pipeobserver` but which does not need that file descriptor (as only the grand-child A writes into it), closes the file descriptor as soon as possible.

Your `pipeobserver` tool may assume that none of the two executables to run receives more than 8192 arguments. This assumption allows you to allocate all memory statically, which prevents you from hunting down complex memory leaks.

Once your `pipeobserver` tool starts working correctly, test it **thoroughly**. Try to cover all use cases, even those leading to error conditions.

It is clear that you need to use proper software engineering and, in particular care and lots and lots of comments, to write a piece of software as intricate as the `pipeobserver` tool.

For this section, you have to submit:

- The source file `pipeobserver.c` which can be compiled to `pipeobserver` using `gcc -Wall -O3 -o pipeobserver pipeobserver.c`.
- A section in your report explaining the software engineering decisions you made, the problems you encountered and the testing you performed.

[6] You must not `execve tee` though, but implement your own `tee`.