CS4375 Operating Systems Concepts
Fall 2024. Homework Assignment 1
Due: 09/29/2024 2:59AM MDT
Team Assignment

This first assignment is a basic Linux get-to-know that will provide the necessary basis for the following assignments, in particular by training everyone in C to get everyone up to speed with this programming language. The assignment is to be done in teams of two students. However, your individual performance will be evaluated.

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

- the source file `head.c`,
- the source file `tail.c`,
- the source file `findlocation.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.

None of the programs is supposed to use `printf`, `fprintf` or `fputs`, as these are level-3 calls. You can use `strerror`, as a special exception. 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 Linux `ssh` Login

This first part of the assignment is really easy: you are just required to access the machine `dandelion` at `dandelion.cs.utep.edu` through `ssh` on **port 22**. You will receive individualized email with your username and login credentials for `dandelion`. Using `ssh` on some Linux/MacOS/FreeBSD/whatever machine or Putty on Windows and using the login and private key you will be sent, access the machine. The following rules govern the use of the `dandelion` system:

- The system runs `fail2ban`. If you fail to authenticate correctly too many times, your IP will get banned for 24 hours. If you need to get unbanned, send email (urgent cases only).
- **Any attempt at exploiting the system for the purpose of sending spam, mining bitcoins or "just" hacking the UTEP network or any other purpose which is not related to this class will result in a notice to UTEP and other consequences, which might include you failing this class!**
- If you f**k up on the system (have a process that takes all memory, uses all CPUs, writes to disc like crazy etc.) but it's just that you happen to do that inadvertedly, **no biggie! Just send email as soon as possible**.
- **Never take the system's resources for granted**. If the computer needs rebooting or there's a power failure, your data/processes etc. may just go away. This should never happen, but who knows. **Make sure enough time is left for homework submissions. Make sure to backup your important data that is on that server.**
- You can use `scp` to move data from that server resp. to that server.

2 Getting to know the `bash` shell

Open a terminal and perform the following on command line. Keep a log of your commands and the answers you got back; this will provide you a basis for the write-up you turn in for this assignment.

1. Using `ssh`, log into the remote system `dandelion` on `dandelion.cs.utep.edu`.
2. On that system find out
   - how many users are logged in concurrently with you,
   - how long the system is already running since last rebooted,
   - what processors it has,
   - how much RAM is available.
3. On `dandelion` or a private Linux installation, open the file `nanpa` using `less`. The file contains quite a comprehensive list of North American phone number prefixes (first 6 digits, excluding `+1`), followed by the location this phone number prefix is attached to. For example, for `915220`, the location `El Paso TX` is listed. Still inside `less`, find the entries for `907519`, `503526` and a couple of other phone numbers you know in the country, as such your home phone, your parents' phone, the phone of a loved one etc.
4. Find out how many lines connecting prefixes to locations are contained in the file `nanpa`. Which Linux command line tool do you use to count lines?
5. List the first 17 lines of the `nanpa` file on command line. Also list the last 42 lines of the file. You can use the Linux tools `head` and `tail` for this task.

3 `head` and `tail`

As seen above, 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.

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.

4 File searching

In the Linux shell get-to-know part above, you already worked with the 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. Again, 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.

A typical POSIX-compatible environment supports the `mmap` system call. This system call allows the contents of a file on the filesystem, described by a file handle obtained with `open`, to be mapped 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 `findlocation` which does the following:

- When called with no argument, your tool must fail, displaying a usage message on standard error.
- When called with one argument, your tool must read the contents of the `nanpa` file (or a file formatted the same way) from standard input, look for the location associated with the number in the single argument and display that location. If the single argument is not properly formatted (as ten digits `0` through `9`), the tool must fail with an error message. The tool must read the input first, in $\mathcal{O}(n)$ time, into memory and then perform a lookup in memory in $\mathcal{O}(\log n)$ time. If ever `lseek` works on the instance of standard input you have, you must use `mmap` in the manner described below and perform the full memory-mapping and lookup in $\mathcal{O}(\log n)$ time. This case is rare but you must handle it.
- When called with two (or more) arguments, your tool must open the file whose filename is given as the second argument and use its content for the lookup of the first argument's number, similar to what is described above.
- If the file descriptor obtained by calling `open` on the second argument is seekable, meaning that the system call `lseek` works on it, your tool must use the system call `mmap` to map the file to memory. The Operating System's implementation of `mmap` guarantees a time complexity of $\mathcal{O}(1)$, meaning that your tool must be able to perform its complete task in $\mathcal{O}(\log n)$ time. If the file descriptor is not seekable, meaning that the system call `lseek` fails, your tool must read in the contents of the file at this descriptor by performing several `read` calls and calls to `malloc` and `realloc`.

You are supposed to base your program only on the system calls `open`, `lseek`, `mmap`, `munmap`, `read`, `write` and `close`, as well as `malloc`, `realloc` and `free`. You are not supposed to use level-3 buffered file manipulation calls, like `fopen`, `fread`, `fprintf` etc, not even for error messages. In case of an error, display `An error occured while xxx: yyy`[1] on standard error which just a call to write. Write your own `display_error_message` function for that purpose. 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 in the regular fashion. Your tool is not supposed to allocate any memory using `malloc` or `calloc`, unless `lseek` does not work on the input file descriptor you use. If `lseek` works, you must not use any 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 for file descriptors where `lseek` works and $\mathcal{O}(n)$ time complexity for file descriptors where `lseek` does not work.

[1] Where `xxx` stands for the operation you just attempted to execute and `yyy` is the string returned by `strerror`.

For this section, you have to submit:

- The source file `findlocation.c` which can be compiled to `findlocation` using `gcc -Wall -O3 -o findlocation findlocation.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.

Use all what you know about good software development for the `findlocation` program to be developed in this section! Your instructor will take the readability of your code into account when grading your assignment. **No spaghetti code allowed!**