findlocation.sh

SWE decisions:

We most first confirm that the phone number is our first parameter.

Given that the parameter was given we must check that it is only made of digits.

If any of the above conditions is not fulfil, we would let the use know that he must enter a North American phone number prefix only containing the first 6 digits and we would exit the program.

We would now get the line from the file by using grep and save it on a variable.

If the length of the line is zero, let the user know that the phone number enter didn't match a North American phone number prefix in nanpa and exit the program.

If line was successfully found, we would eliminate the phone number, the white spaces at the end of the line and print the rest of the line (place where phone number is mainly use).

Problems encounter:

First problem encounter was find a way to check that all character on the input were actually all digits without having to hard code each character on a one to one check.

Problem was solve by using: [[! $1 =~ ^[[:digit:]]+$]]

Second problem encounter was to eliminate only the last white spaces and not all of them.

Problem was resolve by using: line=`echo $line | sed 's/ *$//'`

Testing:

Function got test with the following commands with the respective outputs.

Command: ./findlocation.sh 915760

Output: El Paso TX

Command: ./findlocation.sh 915761

Output: Code didn't match a North American phone number with prefix 915761 in nanpa.

Command: ./findlocation.sh 915-760

Output: Enter a North American phone number prefix with only the first 6 numbers.

head.c

SWE decisions:

We must first find if the argument "-n" was given on the parameters.

If the parameter was given, we read the next parameter which should be a number. If the conversion between string and number was successful, we set the number of lines to be printed to be that number, otherwise we display an error message and exit the program.

If the parameter was not given, we display an error message because "-n" was provided without a number following it.

While reading the parameters if we read something that is not "-n" or the number following it we assume a file to read the information from. Otherwise, we read the information from standard input.

An assumption was made to only read the information from one file so if multiple files were provided, we only read the information from the last one.

After all parameters were read, we open the file using open with read only restriction or set the standard input as the file descriptor (fd).

Now we proceed to read and print the information while we haven't passed the maximum number of lines to be printed or the fd hasn't run out of lines.

We read from fd using a static buffer, if we don't have any more on the file we exit the program, if read was unsuccessful we print an error message and exit the program, otherwise we proceed to print the line using write.

While writing the must iterate over the buffer to see if we have the new line character '\n', if we do we make multiple printings from the same reading. Another case is that if the buffer was fulfilled, we print everything that was read but don't increase the number of lines printed because we haven't finish with that line, so we pick it up were we left from the next reading.

Once we are done, we must close the file descriptor if we are reading from a file.

If closing was not successful, we print an error message.

Problems encounter:

The first problem encounter was to find if the parameter given was "-n".

We solve it by creating a function that simulates strcmp. We pass the parameter read and "-n" to the function and if it returns 0, we know they are the same.

The second issue was to print an error message without using the fprintf function.

The way to solve was to create a function which concatenates two strings…

Another issue was that when reading from files we get multiple lines from one reading.

So, we decide to read the buffer until we hit a new line character or to the end of the buffer and return the number of bytes to print from the beginning to the number of bytes to finish the line or until the end of the buffer.

Another issue was that the buffer may sometimes get fill and cut a line in half, so we are unable to assume that the buffer always has entire line on it.

We proceeded to check if the buffer was fill and if the last character is not a new line character, we don't increase the number of lines printed because the line would be cut in half.

Testing:

Command: ./head -n 2 nanpa

Output:

201200Jersey City NJ

201202Hackensack NJ

Command: ./head -n

Output:

head: option requires an argument -- n

usage: head [-n lines] [file ...]

Command: ./head -n 3

Output:

head: illegal line count -- -3

tail.c

SWE decisions:

(This part was created after head was finished so same approach to read the parameters was used)

A linked list was created to store the lines to be save until the end of the file to be printed.

The liked list has a pointer to the first and the last item. So, it can add to the end in O(1) and remove the head in O(1).

The items on the list have “strings” and the length of the string.

A copy “string” function was created that receive a pointer to a character and the number of bytes to be copy and returns an exact copy of the string, this way we can save the lines.

Once the entire file was read all the lines store on the linked list gets print.

Then we free the space use to save the strings, the items, and the linked list.

Problems encounter:

(This part was created after head was finished so previous error were already solve)

The main issue encounter was the way to store the lines read and save only the last n read.

The way it was solve was by making a linked list struct with pointers to the head and the tail items of the list as well as a counter to know the maximum number of lines to be store.

Another issue found was that if we save the lines from the buffer directly, we lost the information because it gets overwritten was, we read from the file descriptor.

The approach we use to solve this was to create a string copy function which make an exact copy of the string on a different place of memory by using malloc so we don’t lose the information.

Testing:

Command: ./tail nanpa -n 2

Output:

989992Saginaw MI
989996Saginaw MI

Command: ./tail nanpa

Output:

989971Saginaw MI
989975Bad Axe MI
989977Sebewaing MI
989979St Johns MI
989980Saginaw MI
989981Hubbardston MI
989983Vanderbilt MI

989984East Tawas MI
989992Saginaw MI
989996Saginaw MI

findlocationfast.c

SWE decisions:

We first check if the number of arguments is at least 2 (one for the filename and the other one with the phone number).

If we don't have enough argument, we print an error and finish the program. Otherwise, we continue.

We now open the file as read only.

If open was unsuccessful we print an error message and finish the program.

Now that we open the file we need to get to the end of the file.

We can get to the end of the file by using lseek which with the correct parameters return file size.

If lseek was unsuccessful we print an error message and finish the program. Otherwise, we continue.

If the size is not divisible by the number of characters, we finish the program because the file was not seekable.

We continue to map the file into memory by using mmap.

Again, with the right parameters mmap would return a pointer to the beginning of the file and it would only take as many bytes as the size of the file.

If the mapping was unsuccessful, we print an error and finish the program. Otherwise, we continue.

We prosed to look the line on the file that match the phone number provided.

Before starting this solution, we create a struct that match the structure of the line (6 numbers, 25 place, and 1 new line/end of file character).

We now pass the file to a function that receive the file using the struct just make so if the name is dict[1] it would go to position 32 of the file automatically because the size of each entry is 32 bytes. This is used to simplify the notation.

We now use iterative binary search to find the line that match the line.

Using binary search drop the look up time from O(n) to O(log(n)).

If the line wasn't found, we print an error message and terminate the program. Otherwise, we proceed to print the place followed the phone number without the extra spaces (how this was solve given on the problems encounter).

We now terminate the program.

Problems encounter:

Some repeated code was the closure of the file, so a function was created for it to avoid errors.

While the execution of the binary search we need to compare characters, so we create compare_entries.

It returns 0 if they match perfectly, a negative number if number on the file line is greater than the one searching for, a positive number otherwise.

Now, we need to print the trim version of the 25 characters place. We solve it by keeping track of the last character that was not a whitespace and then take the difference between the starting point and the last non whitespace character to get the number of bytes used and send it to be written with just the enough bytes count to not print extra whitespaces.

Testing:

Command: ./findlocationfast nanpa 915760
Output: El Paso TX
Command: ./findlocationfast nanpa 915761
Output: Code didn't match a North American phone number with prefix 915761 in nanpa.
Command: ./findlocationfast nanpa 915-760
Output: Enter a North American phone number prefix with only the first 6 numbers.