Project 3 - Tail

The Unix `tail` program prints the last n lines of a file. You will write a subset of this program.

Valgrind must report NO errors in memory management.

The point of this project

The point of this project is to:

- emphasize memory management.
- use the lowest level file I/O routines.
- learn usage of a fundamental data structure in Computer Science, the circular (or ring) buffer.

Fixed algorithm and data structures

Because I am testing specific skills, **you cannot choose your data structure or the high level algorithm.**.

Fixed (maximum) number of lines to be printed

The real `tail` program allows you to select the number of lines to be printed. In this program, use a fixed constant of 10. Use the following to define the constant. `TAILLEN` says that at most 10 lines will be printed.

```
.EQU    TAILLEN, 10
```

Certainly, if the file is shorter than `TAILLEN` lines, you'll print less.

Reading one line at a time

There is a `getline()` but you may **not** use it.

Rather, you must write it yourself. To do so, you will allocate (and at the end, free) a fixed temporary buffer of 4096 bytes. Then, read one character at a time into the buffer until you find a newline. You already have experience using `read()`. You'll use this again.

In the first project, you read 1 character at a time into a buffer just large enough to hold the character (actually a few but ignore that). In this project, you will be reading into successive bytes of a larger buffer. This is easy to do...

Once you've found a newline, add a null terminator. You have completed a line which is now ready to transfer into your circular buffer (see below).

If you reach **4094** bytes (ensuring there is space available for a newline followed by a null terminator) without reading a newline, truncate the line and ignore any remaining characters in the file until a newline is found.

Remember to finalize your temporary buffer with a newline and null terminator.

Recap: In all cases, your temporary buffer should end in a newline then a null terminator. If the line you're reading is too long, ignore any remaining characters.

With a completed and finalized line in your temporary buffer, you'll dynamically allocate a buffer pointed to by your circular buffer. See next.

Circular buffer

A circle has no end. A circular buffer is ideal when you have to pass an effectively infinite amount of data through a finite buffer. The circular buffer is a fundamental data structure in Computer Science.

Wikipeda has a nice article on circular buffers. Your circular buffer will be `TAILLEN char *`. That is, the buffer is an array of `TAILLEN` pointers to complete lines terminated with new lines and null bytes. Each `char` pointer begins as null. Then, the pointers are filled in until you've filled the buffer... then keep right on going circling back to the beginning. However, before reusing a slot in the buffer you must free what the slots previously pointed to.

At the end of the program, all non-null entries in the buffer must be freed.

Storing lines

Each line you read will be of a different length. You are to dynamically allocate a *perfectly* sized buffer to hold the finalized line and put a pointer to this buffer in your circular buffer.

Again, you may assume that no line of input will be larger than 4K - 2 bytes. You may not simply allocate `TAILLEN` 4K buffers. This would defeat one of the main purposes of this assignment and will be dealt with harshly.

Instead, you will dynamically allocate a "right sized" buffer for every line of text read with appropriate freeing of previously allocated lines which rotate off the circular buffer.

The data structure you are required to use is a circular buffer of `TAILLEN char *` (except in assembly language):

```
char * buffers[TAILLEN];
```

Finally, the circular buffer *itself* must be dynamically allocated.

You must comment

Commenting will be graded this time. You must comment functions like so:

```
/*
C version of the function's signature

High level description of the function

Parameters:
    <register>    <description>
    as many as needed

Returns:
    <register>    <description>
    typically just one

Registers preserved:
    <register>    <description>
    as many as needed - this is *SO* here to help you
*/
```

Required error messages

```
usage:      .asciz  "File name must be given."
badopen:    .asciz  "Open file failed"
noline:     .asciz  "Allocating line buffer failed."
badtail:    .asciz  "Allocating tail pointer buffer failed."
badalloc:   .asciz  "Allocating a tail line failed."
```

`usage` is printed if you don't specify a file as your command line argument.

`badopen` is printed if the file specified doesn't open. It is used with `perror()`.

`noline` is printed if the 4096 character array used to read text cannot be allocated.

`badtail` is printed if a TAILLEN long array of pointers to char cannot be allocated.

As each of the above relate to unrecoverable errors, after printing these, exit *cleanly*.

Additional requirements

Runtime library and system call use

There exists a C version of `getline()`. You may **not** use it. You must read and parse lines yourself.

There exists a C library call `getc()`. You may **not** use it. You must read using `read()`.

File handling calls you are permitted to use:

- open
- close
- read

That's it. Research these.

Other C library or system calls you are likely to use:

- malloc
- free
- memset
- strlen
- puts
- printf

Use of perror

If a file cannot be opened, you must use `perror()` to print out why.

Notice that my error string labeled `badopen` has no trailing punctuation like all the other error messages. This is because it must be used as the prefix to the message printed by `perror`. `perror` will supply a `:` between the prefix and the error message it prints.

Testing

Your program must match letter for letter the output expected by my test programs.

For example - this file is only three lines long (the numbers printed come from the file):

```
./a.out test1.txt
1. This is a test. There are three lines total.
2. This is another test.
3. And last one.
```

This file is very long:

```
./a.out main.s
usage:      .asciz  "File name must be given."
badopen:    .asciz  "Open file failed"
```

```
noline:     .asciz  "Allocating line buffer failed."
badtail:    .asciz  "Allocating tail pointer buffer failed."
dbgprnt:    .asciz  "Bytes read: %d String: %s\n"
badalloc:   .asciz  "Allocating a tail line failed."
pstring:    .asciz  "%s"

        .end
```

Here is the error of specifying no file:

```
./a.out
File name must be given.
```

Here is the error of specifying a bad file name:

```
./a.out
Open file failed
```

`valgrind`

Amongst all its output, `valgrind` must produce this (ignore the number at the beginning of the line):

```
==21779== All heap blocks were freed -- no leaks are possible
```

On the Mac, `leaks` will be used for your testing.

Partner rules

You can work solo or with a partner.

What to hand in

Only one partner should submit code. The code must including the names of the partners (if working as a partnership). The partner who is NOT handing in code must submit a text file containing the name of the person submitting the code on behalf of your partnership.