Assignment 5: autoscroll - A Self-Scrolling File Viewer

Overview

This assignment is designed to give you practice with several parts of the kernel API: signal handling, a bit of terminal control, and file I/O. It is probably the most algorithmically challenging of the assignments you've had so far, but it does not require a long program. Because you haven't yet learned how to control terminals, the program will not use many features of terminals. It will use the ANSI escape sequences that I explained and showed examples of in class to control the position of the cursor and the screen contents.

The assignment is specified by the following man-page-like description. Your job is to write a program that satisfies these requirements. The information regarding program submission is at the end of the assignment.

NAME

`autoscroll -- display a file, automatically scrolling upward one line per second if needed`

SYNOPSIS

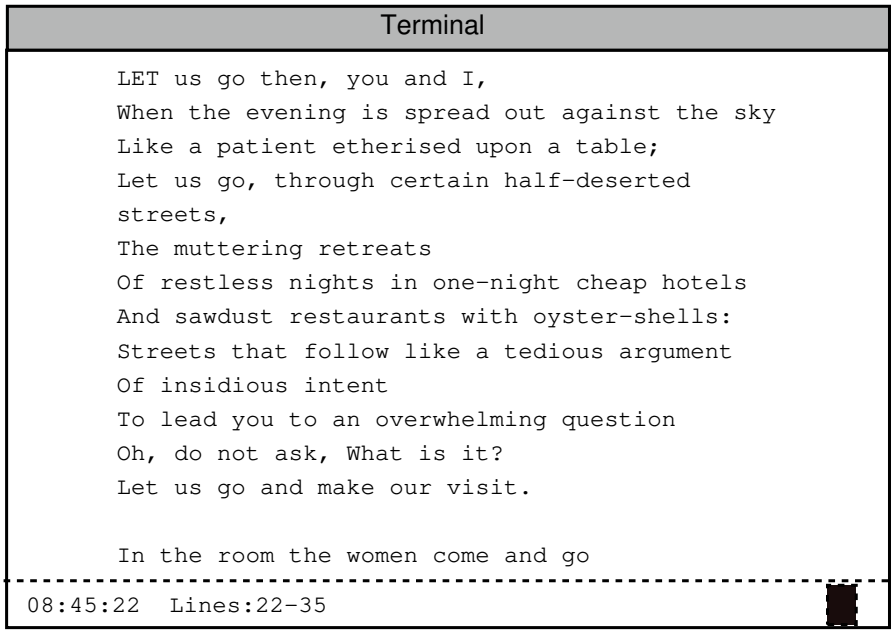
`autoscroll` *`[-s secs] textfile`*
where secs is a positive integer < 60

DESCRIPTION

When run with any size plain text file in a terminal window with R rows and C columns, autoscroll displays the first *R-1* lines of the file in the terminal. If the number of lines in the file is *R-1* or less, it terminates after displaying these lines. Otherwise, every N seconds, with *N=1* by default, it scrolls the file up one line, so that the topmost line disappears and the next line in the file is displayed; but see below for further details. It also updates a clock display and an indicator of the visible lines in the file.

Terminal windows have a coordinate system in which the upper-left corner is (1,1). The bottom row of the terminal window, which I'll call the *status bar*, is reserved for status information and never shows any file content. In particular, it displays the current time in the format "*HH:MM:SS*" in the leftmost position of row R, and starting in column 10 of row R , it displays the range of line numbers of the file that are currently displayed in the format "Lines: *num1-num2*", where *num1* and *num2* are integers, such as "Lines: 100-122".

While the program is auto-scrolling, the cursor is parked at position *(R,C-2)*. At any time, the user can enter one of two key combinations to pause and restart autoscrolling: `CTRL-Z` or `CTRL-C`. `CTRL-Z` pauses scrolling. When the user enters `CTRL-Z`, the clock display continues to keep time but the file is not scrolled upward in the terminal. When the display is paused and the user enters `CTRL-C`, autoscrolling restarts. If scrolling is not paused, `CTRL-C` has no effect and when it is paused, `CTRL-Z` has no effect. The user can terminate the program at any time by sending any other terminating signal, such as `SIGTERM`, `SIGQUIT`, or `SIGKILL` or by by entering `CTRL-\` on the keyboard. When any terminating signal is delivered to the process, the screen is cleared and the shell prompt returns. The following figure shows a sample of how the terminal's area is used.



- If the `-s` option is supplied, the rate at which the file is scrolled should be once every `secs` seconds.

IMPLEMENTATION ISSUES

- This program uses timing to schedule the execution of selected sequences of instructions. In particular, updating the clock, the lines o display and the line information in the status bar takes place only at specific times. Because you have not learned much about timers, your program should use the `alarm()` system call for scheduling all time-dependent actions, such as updating the clock on the status line and the scrolling of the file. The handler for `alarm()` will need to call all functions that perform these actions. Because scrolling is not necessarily performed at each second, the handler for the alarm must keep track of when to scroll.
- Because `alarm()` is not a timer, in order to get repeated alarms, the handler must set it every time it runs.
- In order for `autoscroll` to scroll a file, after it opens and reads the file into memory, it finds and records the positions of all newline characters. Your program can assume that every file has a terminating newline character. Since long lines may be wrapped by the terminal driver in a terminal window, when the program is calculating which lines fit in the window above the status line, it has to calculate how many extra lines are required for wrapping these lines. For example, if the window is 80 characters wide and a line has 200 characters, then it will use 3 lines in the terminal window instead of 1 line (80+80+40 on each of 3 lines). `autoscroll` needs to reduce the range of lines it displays by 2 because of this long line. If the next line to be scrolled into view in the file is a long line, it should not display it until there is room in the terminal for the entire line, including wrapped content, to be visible. This implies that there will be space at the bottom until it can fit that line into the text display region.
- In order to display the file as if it is scrolled, the program clears the screen above the status line and redraws the new range of lines. It does this every N seconds. Most terminals have a scroll buffer that saves what was in the terminal as it advances. `autoscroll` should clear that buffer as well.
- The signal handler for the signal that `alarm()` sends must be able to access the text loaded into memory from the file, the positions of newline characters and many other variables. Because signal handlers do not have hooks for any data, and that data is most likely initialized in the `main()` function, these variables must be file-scoped.

- Once it has set up signal handling, read and loaded files, initialized all data structures and so on, the `main()` function does very little other than detecting when the user enters one of the two keyboard sequences. It just needs to make sure that it detects these key presses and always keeps the cursor parked in the lower left corner. The main() function should use the `sigwait()` system call with suitable processing when it returns. Read its man page.

LIMITATIONS

- There is no limit on the size of a file to be displayed, up to your computer's own limits on memory and disk storage. The program should be able to open any text file that other applications can open.
- The file can have lines that are longer than the width of the terminal window, but your program can assume that no line is longer than 4KB.
- The program can assume that the file's last character is a newline. Alternatively, it can allows files that do not end in a newline and it can add a newline at the end. You might find it is easier when it always ends in a newline.

ERROR HANDLING

If the required file argument is missing, it is an error that must be handled with a suitable usage message. If an option is supplied but is anything other than `-s` followed by a positive integer less than 60, it is an error. The program should report all errors on standard error along with the correct form of usage.

SEE ALSO

`alarm(2), ioctl(3), sigwait(3), console_codes(7)`

Hints

Debugging a program that has signal handlers and screen manipulation can be challenging. You should use a debugger such as `gdb`, and also use `valgrind`. You can also print to a second terminal window — open a second window, enter the command `tty`, record the output (such as `/dev/pts/2`) and in your program, write diagnostics to that file.

Code Re-Use and Program Structure

You are free to use any of the code in the class's repository. There are a few demo programs in Chapter 9 that will be useful. All of them have GNU General Public Licenses. If you do use any of that code, ***the documentation should indicate which parts are not written by you***. The program should be a single source code file. **All functions other than `main()` should precede `main()` in the file.**

Instructions for Submitting the Assignment

1. Use the `submithwk_cs49366` program to submit your program. To submit, if your file is `~/autoscroll.c`, you'd enter

   ```
   $ cd ~
   $ submithwk_cs49366  -t  5 autoscroll.c
   ```

   The program will copy `autoscroll.c` into the directory

   ```
   /data/biocs/b/student.accounts/cs493.66/hwks/hwk5/
   ```

   and if it is successful, it will display the message, "`File hwk5_`*`username`*`.c successfully submitted.`" where *`username`* is your username. You will not be able to read this file, nor will anyone else except for me. But you can double-check that the command succeeded by typing the command

   ```
   ls -l /data/biocs/b/student.accounts/cs493.66/hwks/hwk5
   ```

   and making sure you see a non-empty file named `hwk5_`*`username`*`.c` where *`username`* is your user name and whose date of last modification is the time at which you ran the command.

2. ***You can do step 1 as many times as you want. Newer versions of the file will overwrite older ones.***

Deadline

You must complete this assignment before its ***deadline***, which is ***Monday, May 13, at 7:00 PM. After that, you will not receive credit for completing it.***

Grading Rubric

This assignment is **18%** of your final grade. The output must be correct and the program must conform to the rules written in the ***Programming Rules*** document on the class's webpage. It must be thoroughly documented.