Homework 4 Terminal Multiplexer - CSE 320 - Spring 2018

Professor Eugene Stark

Due Date: Friday 4/13/2018 @ 11:59pm

Introduction

The goal of this assignment is to become familiar with low-level Unix/POSIX system calls related to processes, signal handling, files, and I/O redirection. You will complete a simple terminal multiplexer program that allows several virtual terminal sessions to be multiplexed onto a single real terminal. This program, called `ecran`, is inspired by the classic `screen` program, which was originally written by Oliver Laumann about 1987 and which has evolved over the years since into GNU Screen Although this program originated back in the days before there were PCs with advanced windowing systems, this program and other related programs are still useful today for managing remote login sessions that can be detached and reattached without losing the session state. A more recent program along these lines is tmux. You can read general information about terminal multiplexer programs on Wikipedia.

Takeaways

After completing this assignment, you should:

- Understand process execution: forking, executing, and reaping.
- Understand signal handling.
- Understand the use of "dup" to perform I/O redirection.
- Have a more advanced understanding of Unix commands and the command line.
- Have gained experience with C libraries and system calls.
- Have enhanced your C programming abilities.

Hints and Tips

- We **strongly recommend** that you check the return codes of **all** system calls and library functions. This will help you catch errors.
- **BEAT UP YOUR OWN CODE!** Use a "monkey at a typewriter" approach to testing it and make sure that no sequence of operations, no matter how ridiculous it may seem, can crash the program.
- Your code should **NEVER** crash, and we will deduct points every time your program crashes during grading. Especially make sure that you have avoided race conditions involving session termination and reaping that might result in "flaky" behavior. If you notice odd behavior you don't understand: **INVESTIGATE**.
- You should use the `debug` macro provided to you in the base code. That way, when your program is compiled without `-DDEBUG`, all of your debugging output will vanish, preventing you from losing points due to superfluous output. It is a little more difficult to debug the program in the present

assignment, due to the fact that it takes over the terminal window on which it is started, but in the first part of this assignment you will implement a feature that will probably be helpful in debugging.

> :nerd: When writing your program, try to comment as much as possible and stay consistent with code formatting. Keep your code organized, and don't be afraid to introduce new source files if/when appropriate.

Reading Man Pages

This assignment will involve the use of many system calls and library functions that you probably haven't used before. As such, it is imperative that you become comfortable looking up function specifications using the `man` command.

The `man` command stands for "manual" and takes the name of a function or command (programs) as an argument. For example, if I didn't know how the `fork(2)` system call worked, I would type `man fork` into my terminal. This would bring up the manual for the `fork(2)` system call.

> :nerd: Navigating through a man page once it is open can be weird if you're not familiar with these types of applications. To scroll up and down, you simply use the **up arrow key** and **down arrow key** or **j** and **k**, respectively. To exit the page, simply type **q**. That having been said, long `man` pages may look like a wall of text. So it's useful to be able to search through a page. This can be done by typing the **/** key, followed by your search phrase, and then hitting **enter**. Note that man pages are displayed with a program known as `less`. For more information about navigating the `man` pages with `less`, run `man less` in your terminal.

Now, you may have noticed the `2` in `fork(2)`. This indicates the section in which the `man` page for `fork(2)` resides. Here is a list of the `man` page sections and what they are for.

Section	Contents
1	User Commands (Programs)
2	System Calls
3	C Library Functions
4	Devices and Special Files
5	File Formats and Conventions
6	Games et. al
7	Miscellanea
8	System Administration Tools and Daemons

From the table above, we can see that `fork(2)` belongs to the system call section of the `man` pages. This is important because there are functions like `printf` which have multiple entries in different sections of the `man` pages. If you type `man printf` into your terminal, the `man` program will start

looking for that name starting from section 1. If it can't find it, it'll go to section 2, then section 3 and so on. However, there is actually a Bash user command called `printf`, so instead of getting the `man` page for the `printf(3)` function which is located in `stdio.h`, we get the `man` page for the Bash user command `printf(1)`. If you specifically wanted the function from section 3 of the `man` pages, you would enter `man 3 printf` into your terminal.

> :scream: Remember this: **`man` pages are your bread and butter**. Without them, you will have a very difficult time with this assignment.

Getting Started

Fetch and merge the base code for `hw4` as described in `hw0`. You can find it at this link: https://gitlab02.cs.stonybrook.edu/cse320/hw4

NOTE: For this assignment, you need to run the following command in order for your Makefile to work:

```
$ sudo apt-get install libncurses5-dev
```

The `sudo` password for your VM is `cse320` unless you changed it.

Here is the structure of the base code:

```
hw4
├── .gitlab-ci.yml
├── include
│   ├── ecran.h
│   ├── session.h
│   └── vscreen.h
├── Makefile
├── src
│   ├── ecran.c
│   ├── mainloop.c
│   ├── session.c
│   └── vscreen.c
└── tests
```

If you run `make`, the code should compile correctly, resulting in an executable `bin/ecran`. If you then run this program, what should happen is that the terminal contents are cleared and you will see an error message at the top line, advising you that you need to fill in some code before the program will function. Exit `ecran` by typing `CTRL-a` followed by `q`.

- To get the program to function, you first have to fill in a few lines of code that have been left out of the `session_init()` function in `session.c`. Find the place in this function where the error

message you saw is generated. At this point, you need to arrange for the standard input (file descriptor 0) and the standard output (file descriptor 1) of the newly forked process to be suitably set.**HINT:** the standard error (file descriptor 2) was already set up a few lines earlier and it was what was used to output the error message. You need to "dup" this file descriptor to point the standard input and standard output to the same place.You also need to fill in a call to the proper `exec()` variant in order to execute the program at `path` with the specified `argv` argument vector.

Once you have successfully filled in the missing lines of code, what should happen when you run the program is that a prompt from the shell will appear at the top line of the screen instead of the error message. At this point, you are actually talking to a shell that is the session leader for a virtual terminal session. You should be able to run commands with this shell, essentially as if it were a normal terminal session. However, you will probably notice that the display is not as featureful as a normal terminal window would be. For example, if you try to run `man sh` you will receive a message `WARNING: terminal is not fully functional` and the output will be peppered with funny sequences that look like `[m`. This is because the virtual screen emulated by the base version of `ecran` is a "dumb" terminal that does not support the escape sequences for manipulating the screen contents that a normal terminal window supports. If you get the basic terminal multiplexing functionality to work, you will have an opportunity to earn extra credit points by improving the terminal emulation.

The normal behavior of `ecran` is to send characters that you type to the virtual terminal session and to take output coming from the virtual terminal session and display it in the real terminal. To exit from `ecran`, it is necessary to cause it to take something that you type and **not** send it to the virtual terminal session but rather interpret as a command. `Ecran` recognizes the special character CTRL-a as an escape to command mode from normal processing. If you type CTRL-a, followed by, say, `x`, you should see the screen flash. This indicates that the command character `x` is not currently understood by `ecran`. However, if you type CTRL-a followed by `q`, then `ecran` should exit and restore the contents of your screen to what they were before the program was started.

The basecode that we have provided to you implements some features essential to managing virtual terminal sessions, but in its current form it can only manage a single session. Your objective is to improve the program so that it can seamlessly handle up to 10 virtual terminal sessions, with commands for creating sessions, switching from one session to another, terminating sessions, and so on.

Add Debugging Support

To manipulate the terminal window, the `ecran` program uses a library called `ncurses`. This library has evolved over many years from the original `curses` library that was part of Unix System V. It is currently a part of essentially every Linux and Unix distribution, and it is used to support the behavior of terminal-based programs like `less` and `top`. Your Linux Mint system is no exception, however that system does not assume that you are going to be doing program development with the `ncurses` library. That is the reason why you had to run the `apt-get install` command indicated above.

When the `ncurses` library is initialized, it co-opts the normal terminal behavior. This can make debugging awkward. In order to use `gdb` to debug the program, you have to start `gdb` in a separate terminal window and then attach to the process to be debugged. Here is how you would do this: First, start `bin/ecran` in one terminal window (which it will take over with `ncurses`). Then, in another terminal window, run `ps -a` to find out the process ID of the process running `bin/ecran`. Start `gdb` via `gdb bin/ecran` as usual. Once `gdb` is running, issue the command `attach nnn`, where `nnn` is the process ID of the process you want to debug. If all goes well, `gdb` should attach to the process, the process should be stopped, and you will be able to browse the stack, variables, etc. It might be that when you try to attach you will get a permission failure. This is because restrictions are placed on which processes can attach to which other processes for debugging, to avoid various security holes. If this occurs, then on your VM, you should be able to work around it by starting `gdb` with `sudo`.

Aside from using `gdb`, it would be usual to debug your program by using the macros in `debug.h` to generate debugging printout. Once again, this is made inconvenient by the fact that the entire terminal window has been taken over by `ncurses`. To work around this, you are to implement an option to redirect `stderr` output to a file, instead of to the terminal.

- Modify `ecran` so that if it is started with an option `-o filename`, then it should arrange to redirect the standard error output to the file whose name is `filename`. This file should be created if it does not already exist, and truncated to 0 length if it does. To implement this redirection, you will need to use the `open()` and `dup2()` system calls. Read the man pages for these and refer to the textbook as well. If you implement this feature correctly, then you will be able to monitor the debugging output as `ecran` runs using the command `less filename` (executed in a separate terminal window). Once `less` is started, if you type `F` it will monitor the contents of the file and show you immediately each time additional output is appended.

Improve Terminal Emulation and Add Status Line

Before going on to the harder stuff, make the following easy improvements to the terminal emulation implemented by virtual screens:

- The only control characters for which support is provided in the base code are carriage return (`\r`, ASCII code 13) and line feed (`\n`, ASCII code 10) Add support for the `BEL` character (`\a`, ASCII code 7). If this character is output, the screen should flash. You can use the `flash()` function provided by `ncurses` to do this.:nerd: Documentation for the `ncurses` API can be found online here.
- The base code terminal emulation does not implement scrolling. Normal terminal behavior would be that if a line feed is output when the cursor is at the bottom line of the screen, then all the lines are scrolled up by one, the top line disappears, and a blank line is inserted as the new bottom line. Implement that behavior.

Also, it will be useful to have a "status line", separate from the display of the foreground virtual screen, in which messages for the user can be displayed.

- Arrange for the bottom line of the terminal window (the actual window, not the virtual screens) to be reserved as a status line. This can be done by using the `ncurses` function `newwin()` to create

two "windows": one that covers all but the last line of the screen and the other that covers only the last line of the screen. Set the `main_screen` variable in `ecran` to be a pointer to the big window and create another variable to refer to the status window. Implement a function `set_status(char *status)` to set the contents of the status line.

Error Handling Code Review

- The base code supplied to you does not do a good job of checking for error returns from library functions and system calls. Review the entire base code (you may skip the code in `mainloop.c`) to ensure that error returns from all functions are checked and handled gracefully (without crashing). For some kinds of errors, the appropriate response will be to terminate the program with status `EXIT_FAILURE`. Instead of exiting abruptly, possibly leaving "leaked" session processes and the terminal in an indeterminate state, an attempt should be made to kill existing sessions and to cleanly shut down `curses` processing before exiting. For other kinds of errors, displaying a message in the status line would be appropriate.

Session Management

Now you are ready to proceed to the "meat" of the assignment: extending `ecran` to handle multiple terminal sessions. Specifically, you are to do the following:

- Implement a command `CTRL-a n` (analogous to the already-implemented `CTRL-a q`) to create a new virtual terminal session. When you enter this command, `ecran` should start a new session, fork a new instance of the default shell, make the new session the foreground session, and cause the contents of the terminal window to correspond to the contents of the new virtual terminal. The original session does not disappear; it just goes temporarily into the background, awaiting recall at a later time.
- Implement commands `CTRL-a 0`, `CTRL-a 1`, ..., `CTRL-a 9` for switching between virtual terminal sessions. Your `ecran` program should be able to manage up to 10 virtual terminal sessions, with session ID's 0 to 9. The initial session should have session ID 0, and when you create an additional session it should be assigned the lowest unused session ID. Entering the command `CTRL-a 0` should switch the original session back to the foreground and display the current contents of its virtual screen in the terminal window. Entering the command `CTRL-a 1` should switch back again to the newer session. If you enter, say, `CTRL-a 5` and there is currently no session with session ID 5, then no change should occur and the screen should flash (use `ncurses` function `flash()` for this).
- Now that you can create new virtual terminal sessions, you also have to handle their termination. For this, you should install a handler for the `SIGCHLD` signal. Upon receipt of `SIGCHLD` you should arrange for any terminated session leaders to be reaped and the session deallocated. Be sure to pay close attention to what the textbook and lectures have to say about what it is and is not safe to do in a signal handler. If it is the foreground session that has terminated, then another session should be set as the new foreground session. If there are no more sessions, then `ecran` should terminate cleanly.
- Implement commands `CTRL-a k 0`, `CTRL-a k 1`, ..., `CTRL-a k 9` for killing (forcibly terminating)

terminal sessions. That is, when the user types `CTRL-a k 5` the virtual terminal session with session ID 5 is terminated by sending `SIGKILL` to its process group. If the user types `CTRL-a k` followed by a digit that does not correspond to an existing session, or if the user types `CTRL-a k` followed by a non-digit character, the screen should flash and nothing should be killed.
- Extend the implementation of the `CTRL-a q` (quit) command given in the base code so that it behaves as follows: First, terminate all sessions by sending `SIGKILL` to their respective process group. After cleaning up **all** resources (reaping all session leader processes, closing ptys, freeing `VSCREEN` and `SESSION` structures and their contents, and shutting down `curses` processing to restore the original screen contents), `ecran` should exit with status `EXIT_SUCCESS`. (Note that the version of this command that is implemented in the base code already shuts down `curses` and exits the program, but it does not terminate sessions or clean up properly.)

Command Execution

- Extend the behavior of `ecran` so that any command-line arguments given following the initial `-o filename` option are treated as the name of a command to be executed and its arguments. So, for example, if you run `bin/ecran ps a` the command `ps a` will be executed in the initial virtual session, rather than the default shell. Likewise, if you run `bin/ecran -o xxx ps a`, the command `ps a` will be executed in the initial virtual session and the standard error output of `ecran` (**not** that of `ps`) will be redirected to the file `xxx`.Note that it should not be necessary to use a complete pathname (e.g. `/bin/ps`) to specify the program to be executed; the value of the `PATH` environment variable should be used to determine the search path for locating the executable.

Extra Credit

Some options for extra credit are available for this assignment. Before attempting extra credit, make sure that the basic requirements (discussed above) work perfectly. You will have to tell us what extra credit to look for in your submission. Do this by creating (and committing) a file `EXTRA_CREDIT` in your `hw4` directory. This file should simply consist of a sequence of keywords, each one on a separate line, corresponding to the extra credit items that you attempted and wish to have evaluated. The keyword for each of the extra credit items is given below, along with the number of points that item is worth. One hundred points of extra credit is comparable to the value of a single normal homework assignment (HW1 - HW5). To receive extra credit for an item, it must work perfectly; there is no partial credit given on extra credit items.

ANSI Terminal Emulation (Keyword: `ANSI_EMULATION`, Points: 20)

Find out the control characters and escape sequences that are supported by an ANSI terminal (e.g. see here). These control characters and escape sequences are sent to a terminal to cause it to do things such as positioning the cursor and clearing portions of the screen. Besides carriage return and line feed, which are already implemented, other important control characters are backspace (`\b`, ASCII code 8), which moves the cursor back one column, horizontal tab (`\t`, ASCII code 9), which moves the cursor forward by at least one column to the next tab stop (tab stops are every eight columns, starting with the ninth column), and form feed (`\f`, ASCII code 12), which clears the screen.

ANSI escape sequences all start with the two character sequence `ESC [`, where `ESC` has ASCII code 27.

Extend the terminal emulation capabilities supported by the `ecran` virtual screen to handle these control characters and escape sequences. (You may omit those having to do with "graphics mode", "set mode", "reset mode", and "set keyboard strings".) Change the code in `session.c` so that it sets the `TERM` environment variable for a new session to `ansi`, as opposed to the original `dumb`. If you do this correctly, programs such as `less` and `top` that use cursor-positioning capabilities should behave correctly when run in an `ecran` virtual screen.

Split Screen (Keyword: `SPLIT_SCREEN`, Points: 20)

Implement a "split screen" command `CTRL-a s` that splits the main screen in half vertically, thereby creating side-by-side windows that can be used independently to view any of the `ecran` virtual sessions. For simplicity, the `CTRL-a s` command should "toggle" the split screen function: the first time it is typed, the screen should be split and the current session displayed in both halves. If `CTRL-a s` is typed again when in split screen mode, the screen should revert to its normal behavior, displaying as the foreground session what was previously the foreground session in the left half-screen.

Enhanced Status (Keyword: `ENHANCED_STATUS`, Points: 10)

Arrange for the current time of day and the number of existing virtual sessions to be always shown at the right-hand side of the status line. The time of day should be in HH:MM:SS format, and the information should update once per second. One way to do this is to use the `alarm()` system call and a `SIGALRM` handler. Another way to do it would be to modify the parameters to the `select()` call in `mainloop()`, but you should not attempt to do it that way unless you are very keen to understand how `select()` works and you have some time to spend on debugging.

Help Screen (Keyword: `HELP_SCREEN`, Points: 10)

Implement a command `CTRL-A h` which when typed, switches to a virtual screen that displays a summary of the commands `ecran` understands and the sessions that are currently active. This screen should be displayed until the `ESC` key is pressed, at which time the display switches back to the foreground session.

Hand-in instructions

As usual, make sure your homework compiles before submitting. Test it carefully to be sure that doesn't crash or exhibit "flaky" behavior due to race conditions. Use `valgrind` to check for memory errors and leaks. Besides `--leak-check=full`, also use the option `--track-fds=yes` to check whether your program is leaking file descriptors because they haven't been properly closed.

Submit your work using `git submit` as usual. This homework's tag is: `hw4`.