Lab 9

Due Mar 13 by 6:30pm **Points** 1

Lab 9: Signals

Due: Friday 13 March before 6:30pm

Introduction

The purpose of this lab is to practice using signal handlers and alarms and to review operations on binary files. We will use these techniques to explore how quickly your machine performs read and write operations. The lab provides links to help you find information about the timer and other functions used, but it useful to have TAs nearby while you are working, so you can ask questions.

Timing Reads: The Overall Task

In this lab, you will explore how various operations affect the runtime of a program. In short, you will be doing some profiling. You will write a C program that takes an integer argument `s` representing a number of seconds and a string representing the name of an existing binary file `f`, which contains 100 integers. The program will repeatedly generate a random integer from 0 to 99 and use that number as an index to read the corresponding integer from `f`. The program runs for `s` seconds and then reports how many reads were completed.

1. Create a test file

Since your program has to read integers from a binary file, you need to create a test file containing 100 integers. Write a small program, `write_test_file.c`, to do this task. It should take a single argument representing a filename and create a file with that name that contains 100 integers. The values of the integers themselves don't matter, so start off by writing the integers 0 through 99 in order.

Because your newly-created test file is binary, you can't display the integers with cat, and a normal text editor won't work either. Instead, look at the contents with the tool `od` (https://linux.die.net/man/1/od). The `-vtu1` flag will display the contents in a format that is easier for you to read. Once you have convinced yourself that the 100 integers are being correctly written, change your `write_test_file` to write random integers between 0 and 99. To generate random numbers, use the function `random` (https://linux.die.net/man/3/random).

2. Write a program to read random locations in the file

The next step is to complete the first draft of the `time_reads.c` program. The starter code in the `lab9` folder just handles the command-line arguments.

For now, you will need to write code that goes into the loop body. Your program should seek to a random location in the file, read the integer at that location, and print it. Change your program to seek and read in the infinite loop.

At the moment, the only way to stop the program is to send it a signal. You can do this from the keyboard directly (with `ctrl-C`) or from another terminal window **by looking up the process id (https://linux.die.net/man/1/ps)** and using the **`kill`** **(https://linux.die.net/man/1/kill)** command). Try both ways.

3. Add a signal handler

Now, use **`sigaction`** **(https://linux.die.net/man/2/sigaction)** to add a signal handler to your program. Start with something simple that just prints a message to standard out and exits (with termination code `0`) when it receives a `SIGPROF`. Then, to test it, run your program and use `kill` to send it a `SIGPROF` signal from the shell in another terminal window. (Check the man page for **kill (https://linux.die.net/man/1/kill)** to see how to get a listing of signals and their numbers.)

4. Add timing

Add code to set up a timer (using **`setitimer`** **(https://linux.die.net/man/2/setitimer)**) for `s` seconds (where `s` is obtained from the command line). You shouldn't use real time, since your results will be affected if your system is busy. Instead, use `ITIMER_PROF` to make the `itimer` send a `SIGPROF`. **Here (https://www.gnu.org/software/libc/manual/html_node/Setting-an-Alarm.html)** is an extra manual page that gives more advice on how to set an alarm. Part of the learning goals of this lab is practicing using a man page to learn a new system call so we are intentionally not giving you any more instruction about how to set the alarm.

Change your signal handler to print a message to standard out (use the `MESSAGE` format in the starter code) that provides both the total number of reads completed and the time elapsed (`s` seconds). Run your code a few times to see how it works.

Once you have done this step, you can submit your lab. However, there's more to do to explore how your computer works ...

5. Explore! (Not for lab credit)

Try different amounts of time (1 second? 5 seconds? 10 seconds?). Is the number of reads that your program can perform per second fairly constant? (It's slower the first second. Why do you think that is?) What happens if you remove the print statement in the loop (not the one in your signal handler)? Can

you explain why the number of read operations per second changes? What does that tell you about the cost of printing?

Try opening the file for reading and writing, and instead of reading a value, write a random value at the offset. Is one operation more expensive than the other, and if so, can you speculate as to why?

Now, change the size of your data file (and be sure to change the possible offsets generated in your program). Does the number of reads (or writes) per second change if the file size is much larger? If so, is there a specific file size where the change occurs? Do you have an explanation as to why?

If you've been printing a message for every piece of data being read, you may notice that the alarm often happens while the statement is being printed, so only part of the output will be delivered. Optionally, use `sigprocmask` to mask the `SIGPROF` signal during the print statement so that each line of output is fully printed before the alarm is delivered and the signal handler executes and finishes the program.

Submission

Submit both `write_test_file.c` and `time_reads.c` files to MarkUs under the `lab9` folder in your repository.