CS 334: Operating Systems
Lab 2: System calls and programming primitives
Spring 2024. (100 pts possible)
Due Date: Wednesday 01/31/2023. NTL Midnight

Your Name : _____

Introduction:

This is an individual exercise. Everyone will turn in their own write-up, but please work with a partner so your learning progress is faster and more thorough.

Throughout the different programming classes, you have seen that there is more than one way of implementing a solution. This exercise focuses on getting you more up to speed in C and learning about the trade-offs of different implementations. In this lab you will see the difference between various programming primitives, from the kernel level to the library level calls. The different libraries are used to represent the same abstract constructs or methods but with very different implications on the application's performance: its running time, the space considerations and reliability.

Educational objectives (student will be able to):

* identify different level functions (and calls)
* profile application behavior in terms of run-time, calls used,
* differentiate between using the system level, low level and high level user level C calls

Readings:

* man intro
* man syscalls
* man man
* try as much as possible NOT TO GOOGLE, but use the man pages instead – you will learn more this way.

Materials:

* Access to UP's Linux VDI or a Debian flavor Linux system

Part 1 (15 pts). System basics

1. From CLI 'man -k device' will return a list of commands with a number following each command, for example: 'mmap64 (3)' is one of the commands in the list.

   a. What commands does the list 'man -k memory' returned? Don't give me the list of commands, instead, please explain what does the command do. In particular, what does the flag '-k' do?
   Answer:

   b. What do the numbers following the command mean? For example the number 3, in the following command 'mmap64 (3)'. Please list all values 1-9 and what they refer to. Hint: the manual on how to use the manual might useful.
   Answer:

   c. What is the section number for the system calls?
   Answer:

2. What is a /dev/tty and what is it used for? Try commands: ‘info tty’ and ‘who’ (run ‘who’ from a Mac console or a local Linux subsystem – not VM).
   Answer:

3. *Execute ‘man -k write’*
   a. How many ‘write’ commands are there
   Answer:

   b. What do you need to type on the CLI to view the man pages for the write (1) command? Similarly, for write (2).
   Answer:

   c. Why are there two ‘write’ calls and how do they differ from each other? (Think speed)
   Answer:

4. Similar to the questions above, let’s looks at two similar and closely related calls: open and fopen.
   a. What is the returned value type for each of the functions? It is worth your time to read the “DESCRIPTION” section for each of these calls.
   Answer:

   b. The returned values from either of the calls is used in one of the subsequent function such as write and fprintf:
   1. If my program uses a lot of custom pretty-prints (formatted string such as “%d, %s, %f, %s”), should I open an output file using `open` and use `write`, or should I use `fopen` and `fprintf`? Hint: the function signature give you the answer.
      Answer:

   2. If my program writes strings that are always the same fixed length, think of an airline reservation system, should I open an output file using open and use write, or should I use `fopen` and `fprintf`?
      Answer:

Part 2 (25 pts). Application profiling
There are several ways of measuring the execution times of an application. The simplest is to run the application inside of the ‘time (1)’ utility, while the more sophisticated measurement is done using the ‘valgrind’ tool. Yes, the same tool you used previously to check for your program’s memory leaks.

Log into your UP’s Linux VDI. On the P:\ drive, create your course CS334 folder with the sub-folder called lab2. Download and extract the lab2_startercode into your lab2 sub-folder.

Open a terminal window and navigate to the lab’s sub-folder called Pt2_profiling. Folder contains my solution to one of your 305 assignments that implemented Dijkstra to find all flight routes between a source and destination airports. The name of the executable is routeFinder. The other files in this directory are: AllUSRoutes.txt which is the command

line argument to the program (executable). The file has all commercial flights flown in the US in one calendar year. Finally, the last file is called ‘searches.txt’ which contains 2573 flight route queries to be executed as an input from the standard-in.

First, let’s use ‘time’ utility to measure program’s run-time:

1. Execute the program inside of the ‘time’ utility, using the AllUSRoutes.txt as the command line input that builds the graph for the Dijkstra algorithm. The searches to be executed by the routeFinder should be all 2573 source and destination queries listed in the searches.txt file. Using the redirect of the search file to appear as the standard input into the executable. The final execution command then would be:
   ./routeFinder AllUSRoutes.txt <searches.txt

   a. Give the output of the program running in the time utility measuring how long it tool to execute the above program with searches listed in the text document
      Answer:

   b. What do these three reported times mean? Which one should we be looking at, and what kind of information does it tell us?
      Answer:

2. We would like to have more detail, to do this, program often have a verbose flag. Time also has a verbose flag, but we have to be very specific which time program we would like to use. Find where the ‘time’ program is located at by using the ‘which time’ command to get the program’s location. Then rerun the routeFinder the same as in question Pt2 Q1, but this time, type in the entire path (absolute path) of the time utility starting from the root, and use the verbose flag when running the routeFInder program. (use ‘which’ utility to locate the executable’s location)

   a. What is the command line to do this?
      Answer:

   b. Give the output of the time utility
      Answer:

   c. What information is reported this time? Give short (couple words) explanation of each output line that is printed using the verbose of the time utility. Hint: the man page might be super useful.
      Answer:

3. Now, let’s move on to an industrial quality tool called ‘valgrind’ to do the similar analysis as above.
   Profile the code execution of the above using valgrind with its profiler flag to record all access times:
   `valgrind --tool=callgrind ./routeFinder AllUSRoutes.txt <searches.txt`
   (be patient this will take a bit)

   a. List the content of the directory. There is a new file there. What is this newly created file called? What does this file record (look into the file for markers)?
      Answer:

b. List the fist 150 lines of the log file. Focus on the lines after the header (the first couple of line of the file) that start with – ob, fl, fn, calls…. Take a crack at explaining at least 3 different lines in this file that start with the prefix ob, fl, fn, calls….– copy and paste the line and explain what the lines mean/record.
Answer:

4. Open `kcachegrind` with the new file you have in your directory after running `valgrind`. The file name's is likely prefixed by "`callgrind.out…`". From the command line use the following: "`kcachegrind <file callgrind.out… here>`"

   a. What is `kcachegrind` and what does kcachegrind do?
   Answer:

   b. How many times were the following functions called?
   Answer:
   - Dijkstra =
   - buildGraph =
   - addEdge =
   - removeMinQueue =

   c. Which C library was called the most number of times? What is the file name of the compiled object that this library function-call came from?
   Answer:

   d. How would you use this tool as a programmer to optimize the run-time performance of your code?
   Answer:

Part 3 (30 pts). System vs. library vs. user calls
Below is a very similar program to what we analyze in class that uses the open, read, and write primitives. Navigate to the sub-folder Pt3_bug

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <errno.h>
5. #include <fcntl.h>
6. #include <unistd.h>
7.
8. int main(const int argc, const char * argv []) {
9.   if (argc != 2) {
10.         printf("Usage: %s <FILE NAME>\n", argv[0]);
11.         exit(EXIT_FAILURE);
12.       }
13.
14.       const char * filename = argv[1];
15.       int fd = open(filename, O_WRONLY);
16.       int errnum = errno;
17.
18.       if (fd != -1) {
19.         printf("Opened %s\tfd = %d\terror = %s\n",
20.           filename, fd, strerror(errnum));
```

```
21.       } else {
22.         printf("Failed to open %s\tfd = %d\terror = %s\n",
23.           filename, fd, strerror(errnum));
24.         exit(EXIT_FAILURE);
25.       }
26.       const int BUFFER_SIZE = 128;
27.       char buffer [BUFFER_SIZE];
28.
29.       while (read(STDIN_FILENO, buffer, BUFFER_SIZE) > 0) {
30.         if (write(fd, buffer, BUFFER_SIZE) == -1) perror("write");
31.       }
32.
33.       close(fd);
34.       return EXIT_SUCCESS;
35.     }
```

Compile the code: gcc -o mystery mystery.c and run the code: ./mystery out2.txt

1. Your program failed to execute, because of a file problem

a. What is the line number that printed the error message?

Answer:

b. What is the line number the opened or attempted to open the file?

Answer:

What is the function signature – what are the function's arguments to open the file?

Answer:

What are the actual argument values that are used?

Answer:

c. Fix the bug in the file opening call so that: (1) when the output file does not exist a new file is created, (2) when an existing file is opened successfully it is writable. Copy and paste the fixed line of code. (Note: Do not write any new lines of code. Fix the existing code by fixing the exiting function call.)

Answer:

d. Describe what was wrong.

Answer:

2. Recompile and re-run your program: ./mystery out.txt. As soon as you executed the program, it looks like it just hangs, but it is waiting for the standard input from the command line. Type in a line, or two, then press "ctrl-d" which will terminate the input.

a. Now, we can look closely what should the program do in the first place? In detail, look at the source code lines 29-31. Give a short, laymen description of what the program should do? (1 sentence).

Answer:

b. List the out.txt, obviously there is something broken in the lines 29-31. Please don't change the semantics of the code, don't add additional code, don't delete anything. Program has a small bug. Fix the bug in lines 29-31 to fix the output. Hint: exactly, how many bytes (characters) did you type in the terminal? How

many characters were successfully read? How many characters did the write routine print?
Copy and paste the fixed source code line(s):
Answer:

c. Describe what was wrong.
Answer:

3. Recompile and run your improved version: ./mystery out.txt. Re-run your program 5 times: ./mystery out1.txt;./mystery out2.txt; ./mystery out3.txt;./mystery out4.txt; ./mystery out5.txt. Hmm, this time, it did create new files but:

a. What are the file permissions? Can you print the content of all of the created files using the CLI using the 'cat' utility?
Answer:

b. It looks like our original file creation fix did not cut it. Go back to the code of line that creates the output file and fix it so that when a file is created, it is always created with 'r--' permissions, in other words, user has read permissions (00400). Again, do NOT add any new lines, instead fix the problem in-place. HINT: Check the file open man pages to see if you can explicit set the file permissions when a file created. Give the fixed source code line(s)
Answer:

d. Describe what was wrong.
Answer:

Note: it is worth your while to think about what is causing this undesired behavior.

Part 4 (30 pts). Fixed length record files
Now, let's take advantage of fast byte reads from a file. Navigate to the sub-folder Pt4_database. There are two source files: writeDB.c and readDB.c. writeDB.c will create a file, where each line of the file has a fixed length. The readDB.c will access these records by reading them in random order. Let's look at the details:

1. Let's take apart the lines that create the fixed record (LOC: 39, 44, 49...). It is a two step process, first, program creates a formatted buffer with the fixed record, then it will write to the file. To create the record:
```
sprintf(buff, "%s%*u", txt, (REC_LEN - (int)strlen(txt)+2), ' ');
```

a. What does the sprintf call do?
Answer:

b. Where is the data written by the sprintf call?
Answer:
1.

c. What does the formatted string `%s%*u` do? In particular, what does the '*' do and what is its ('*') parameter?
Answer:

d. What type of information is 'u', what value does it have, and what data is printed?
Answers:
u is:

u has value:
u prints a value:
if “Alf” is passed:
u has value
u prints:

2. The second step is to write the buffer to a file. `write(fd_out, buff, REC_LEN);` will write the record to a file.
   a. What are the write call’s parameters?
   Answer:

   b. How do these parameters differ from the fprintf? Can fprintf write a byte record this way?
   Answer:

   c. Looks like there are tradeoffs between using write and fprintf. List and explain 2 trade-offs? (Please don’t use two opposites of the same trade-offs).
   Answer:

3. The readDB.c reads records in random order:
   a. What do the lines 43-48 do? What is ‘fstat’, and what are its arguments?
   Answer:

   b. How is the number of records calculated?
   Answer:

   c. How does the program decide which record to retrieve?
   Answer:

   d. How is the file descriptor positioned to the right place in the file? How does this function do the re-positioning? Hint: Is this in any way similar to array access?
   Answer:

Part 5 (0 pts): Additional information about his module:
Please give me any feedback about this module. What worked, what would you change, what additional questions would you include?

What to turn in on Moodle:
* A compressed archive with the pdf of the above lab report as a single document.