INFO0940-1: Operating Systems (Groups of 2)
Project1: Tracer

Prof. L. Mathy - G. Gain - J. Francart

Submission before Wednesday, March 31, 2021 (23:59:59)

1 Introduction

In this project, you must implement in *C* a custom tracer which will gather and display different information about a particular running process. Your tracer will take two arguments:

1. the first one is a string which represents the mode. There are two different modes: *"profiler"* or/and *"syscall"*;
2. the second one is also a string which represents the process which will be traced (also called the "tracee"). This string can either be a relative or an absolute path to the executable of the tracee.

According to its first parameter, the tracer will have a different behaviour. The *"profiler"* mode will display all functions called by the tracee as well as the number of instructions for each of them. The second mode will display all system calls used by the tracee during its execution. In this project, the executable of the **tracee** must be compiled with the *static* flag of GCC with the following command:

```
gcc --static -g -o tracee tracee.c
```

2 Sample Archive

For this project, we provide you a sample archive which contains:

1. a `Makefile` that you need to complete;
2. a sample source file (`tracee.c`) which must be statically compiled to generate the tracee program;
3. a text file which contains a list of system calls (*syscalls.txt*).

3 Requirements

The following requirements must be satisfied:

- Your tracer must respect a given call: `./tracer <mode> <prog>` where `<mode>` is either `-p` to profile the program (profiler mode) or `-s` to get the list of system calls used by the program (syscall mode). The `<prog>` argument represents the name of the executable which will be traced (it can be prefixed by a relative or an absolute path);
- Your tracer must be tested and run on an x86 architecture (32bits);
- Your tracer must respect a given output (see Sections 4 and 5);
- You must redirect the output of the traced process to "dev/null";
- For any additional data structures, you **must** use `malloc()`, `realloc()` and `free()` to manage memory. Use valgrind[1] to detect memory leak;
- Your tracer must exclusively use `fork()` system call to manage process creation and `ptrace()` to trace the tracee; Nevertheless you can use other system calls such as `waitpid`, `wait`, ... to manage process execution;
- Your code must be readable. Use common naming conventions for variable names and comments;
- You are free regarding the implementation of the program however you must provide a well structured code where each part has a well-defined role ("library" or "business" code). In other words, it is strictly forbidden to have a single file containing all the code;
- Your code must be robust and must not crash. In addition, errors handling must be managed in a clean way. Only fatal errors such as failed memory allocation will stop the tracer. Other errors must be displayed on `stderr`.

4 The profiler mode

When you use the tracer with the *-p* argument, it must detect and record all **relative** `call` and `ret` instructions of the tracee. With this information, your program must construct a call tree/graph which contains the number of instructions executed by each function.

Concerning the output, your must respect a specific convention. For each lower level, you must add 4 spaces before the function name. In a more conventional way, it gives:

$(d \times 4\_)$ fct_name: nb_instructions, where d represents the depth of the tree

[1] http://www.valgrind.org

In the case of a recursive function, it is slightly different. Indeed, when a function performs a recursive call, you must count the number of recursive calls and adapt output as follow:

($d \times 4\_$) fct_name [rec call: nb_rec_call]: nb_instructions, where d represents the depth of the tree

Nevertheless functions called by a recursive function must add a new level in the call tree/graph as it is shown in Listing 1. In that case, the function `f8` performs 4 recursive calls and each of them adds a new level in the call tree/graph.

In a general way, the number of instructions of a function is equal to the number of instructions performed by the function plus the number of instructions made by its children. When a function calls another one, the `call` instruction is counted for the caller. Finally, when a function terminates and calls the `ret` instruction, this instruction is counted for this function only.
Here is an example of the output:

```
$ ./tracer -p bin
f1: 8806
    f2: 8794
        f3: 132
            f4: 34
        f5: 8652
            f6: 327
            f7: 8
            f8 [rec call: 4]: 199
                f9: 21
                f9: 21
                f9: 21
                f9: 21
                f9: 21
            f10: 299
                f11: 294
                    f12: 8
```

Listing 1: Sample output in syscall mode. Note that we do not use real data in this example.

Hint: There are several methods to gather the list of functions of a program. For this part, we let you investigate nevertheless it can be interesting to take a look at the slides of the second practical session. Note that you **must not use** an external library.

5 The syscall mode

When the tracer is called with the `-s` argument, it will intercept and display all system calls which are called by the tracee process. We provide you a text file

(called `syscall.txt`) which associates the system call number to its respective name. In order to test your program, you can use the `strace` command on the tracee and check if your program displays system calls in the same order.

Concerning the output, your program must display `syscall: syscall_name` for each system call encountered during the execution of the tracee. System calls must be displayed in the order in which they are called. Here is an example of the program output:

```
$ ./tracer -s bin
syscall: uname
syscall: brk
syscall: brk
syscall: set_thread_area
syscall: readlink
syscall: brk
syscall: brk
syscall: access
syscall: write
syscall: write
syscall: write
syscall: write
syscall: exit_group
```

Listing 2: Sample output in syscall mode

6 Report

You are asked to write a very short report (**max 2 pages**) in which you briefly explain your implementation. In addition, you must answer to the three following questions:

1. What are the opcodes that you have used to detect the `call` and `ret` instructions?
2. Why are there many functions called before the "real" program and what is/are their purpose(s)?
3. According to you, what is the reason for statically compiling the "tracee" program?

Note that, you can draw diagrams to illustrate your explanations but note that these ones **will not** be considered as appendices.

Finally, we would also appreciate an estimation of the time you passed on the assignment, and what were the main difficulties that you have encountered.

7 Evaluation and tests

Your program can be tested on the submission platform. A set of (minimal) automatic tests will allow you to check if your program satisfies the requirements.

Depending on the tests, a **temporary** mark will be attributed to your work. Note that this mark does not represent the final mark. Indeed, another criteria such as the structure of your code, the memory management, the correctness and your report will also be considered. You are however **reminded** that the platform is a **submission** platform, not a test platform.

8 Submission

Projects must be submitted before the deadline. After this time, a penalty will be applied to late submissions. This penalty is calculated as a deduction of 2^{N-1} marks (where N is the number of started days after the deadline).

Your submission will include your code (all the required `.c|.h` files in a directory named `src`) along with a `Makefile` which produces the binary file `tracer`. Please also add your report (`.pdf` or `.txt`) in the `src` directory.

Submissions will be made as a `src.tar.gz` archive, on the submission system. Failure to compile will result in an awarded mark of 0.
Bon travail...