Project #1: Tracing, System Calls, and Processes

Objectives

In this project[1], you will learn about system calls, process control and several different techniques for tracing and instrumenting process behaviors. You will develop two (2) small programs: one that uses a **shim** library (and the LD_PRELOAD environment variable) to detect simple memory leaks, and another that uses the **ptrace** API to report statistics about the system calls made by another program.

Program #1: Memory Leak Detector

Your leak detector will be called **leakcount** and will be run from the command-line with the syntax shown in Listing 1. It will accept a series of arguments—specifying a program or shell command—run that command (in another process) and inspect its calls to **malloc** and **free**. For this program, a leak is defined as a call to **malloc** that returns a pointer that is never freed during the course of the program's execution.

Your programs should be written in C and should consist of two parts: the **leakcount** program, and a shared library, called **memory_shim.so** that will act as a shim for intercepting calls to **malloc** and **free**.

The **leakcount** program will run the specified command in a separate process (using either **fork+exec∗**, **popen**, or **system**). However you create the process, you will need to set the **LD_PRELOAD** environment variable to be the path to your shim library **memory_shim.so**. (Hint: There are multiple ways to do this. If you go the **fork+exec** route, **execvpe** allows you to specify the environment explicitly as an argument, but this is not the only way to accomplish this task).

Setting the **LD_PRELOAD** environment variable tells the operating system to load your library before any of the other system libraries (like **libc.so**). This allows you to effectively replace any library call you want, with your own implementation. This can be extremely useful

[1]This project is used with permission by Dr. Jacob Sorber

Listing 1: Syntax and output format for your memory leak detector.

leakcount ¡command¿
Example: `./leakcount cat ./mouse.txt`
Example: `./leakcount ./my_test_program`

Output: (to stderr)

```
LEAK<tab><num bytes>
LEAK<tab><num bytes>
TOTAL<tab><total count><tab><bytes leaked>
```

Example Output: (to stderr)

```
> ./leakcount ./mytest
LEAK  128
LEAK  35
TOTAL 2 163
```

Listing 2: Memory leak detector hints.

```
//implement these functions
void free (void *ptr);
void *malloc(size_t size);

//call the original versions
#include <dlfcn.h>
void *(*original_malloc) (size_t size);
original_malloc = dlsym(RTLD_NEXT, "malloc");
void *ptr =  original_malloc(17);
```

in testing programs or modifying system functionality. Make sure you wrap your head around what we are doing here, because this will come up again in this course.

In its simplest form, your shim library will be implemented as a single C file, that implements the two functions shown in Listing 2. Your code should keep track of allocations that occur and which ones get freed. When the program or command finishes, your program (**leakcount**) should report the number of leaks that occurred, to `stderr` in the format described in Listing 1. Please print out the leaks in the order that they were allocated. Your program should print out a TOTAL leak count even if there are no memory leaks. Your program should not produce any other output.

There are multiple ways to implement this program. Communication between your two processes is optional, and how your two processes communicate is left up to you. Files, pipes, sockets, etc are all fair game.

Listing 3: Syntax and output format for your system call tracer.

sctracer <command>
Example: ./sctracer ../my_random_test_program <output_file>
Example: ./sctracer ‘‘../my_test_program arg1 arg2’’
<output_file>

Output: (to specified output file)
<syscall number><tab><num calls>
<syscall number><tab><num calls>
...

You may also find the `constructor` and `destructor` attributes that are provided by GCC to be useful in this part of the project. A very good explanation of how these work is provided at the following url: `http://phoxis.org/2011/04/27/c-language-constructors-and-destructors-with-gcc/`

Also, note that there are several things that the child process can do that can make your job more difficult. A program run by leakcount can close `stderr` before returning from `main`. This will cause anything printed by your library's destructor from appearing in the terminal. A program can also terminate abnormally (by calling `abort`) which will prevent library destructors from running, at all. You are not required to handle either of these two cases, though feel free to tackle them, if you are up for an additional challenge.

Program #2: System Call Tracer

Your second small program will be called **sctracer**. It will also be run from the command-line, with the syntax shown in Listing 3. Unlike the first program, which accepted multiple arguments, **sctracer** will take a shell command as a single argument, which it will run, in another process, and report statistics on the system calls that it uses. These statistics will be reported in an output file, specified on the command-line.

In this program, we will focus on system calls, rather than library calls, and we want to watch *all* system calls, rather than a small set of calls. Using a shim in this instance would require us to write a wrapper for every system call, which would be really tedious. So, instead of using **LD_PRELOAD** and a shim library, in this program you will use the

ptrace API.

The **ptrace** API is extremely powerful. You can read or write to another process's memory or registers, monitor the signals received by the process, stop and start the process when a system call occurs, and of course kill the process. In this project, we will only focus on a subset of this functionality. At the bare minimum, you will need to use the PTRACE_TRACEME, PTRACE_SYSCALL, PTRACE_PEEKUSER, PTRACE_SETOPTIONS requests and the PTRACE_O_TRACESYSGOOD option, and I will provide a **ptrace** example in the Git repository; however, you should start this project by reading the `man` page on **ptrace**. Not reading this documentation carefully, would be unwise. I also recommend brushing up on **waitpid**, **getpid**, and other process management functions.

In this program, I strongly recommend that you use **fork** to create the new process. The child process should call **ptrace(PTRACE_TRACEME)** before calling **exec∗**.

Once the child process has completed, your program will write the results into the output file, in the format specified in Listing 3. Each system call should be on its own line. You should report the system call number, followed by a tab character, followed by the number of times that system call was called. The results should be ordered in order of increasing system call number (smallest sys-call number first).

Other thoughts, hints, and warnings

Your programs should be written in C or C++ and use good programming style. You may use only the standard libraries and those provided by glibc or your compiler. You code should be `-Wall`[2] clean, and most importantly it should be your own. Both programs will be tested on single-threaded terminal programs. You don't need to correctly support multithreaded or multiprocess applications. Also, your **leakcount** program is not required to correctly count leaks in programs that close `stderr` or terminate with `_exit`, `abort`, or any other method other than returning from main. Depending on how you approach this project, handling these cases can be tricky.

For this project, there are several tools that will be very helpful. The **strace** and **ltrace** tools can be used to sanity check your results. I

[2] **"-Wall clean"** = If compiled with the `-Wall` flag, it should not produce any warnings.

strongly recommend you use **valgrind** for all of your projects to help you find memory leaks and other such issues. Finally, those of you who use **printf** to debug will find that this may not work well on the first program. The reason is that **printf** allocates memory, so calling it within your shim library's **malloc** or **free** function will likely cause the process being checked to go down in flames (infinite recursion). I'm often surprised at how many students don't use **gdb**, which I think makes life much easier. I strongly recommend it.

Make sure that your code compiles, runs and has been thoroughly tested on the lab machines.

Submission Instructions

This project is due by 17:00 on September 14th, 2018. Absolutely no late assignments will be accepted, and deadline extensions typically require a natural disaster.

When your project is complete, archive your source materials, using the following command:

```
> tar cvzf project1.tgz README Makefile <list of source files>
```

The **Makefile** should build your program (by running **make**). It should produce three files, called **leakcount, memory_shim.so**, and **sctracer**.

The **README** file should include your name, a short description of your project, and any other comments you think are relevant to the grading. It should have two clearly labeled sections titled with KNOWN PROBLEMS, and DESIGN. The KNOWN PROBLEMS section should include a specific description of all known problems or program limitations. This will not hurt your grade, and may (in rare cases) improve it. I am more sympathetic when students have thoroughly tested their code and clearly understand their shortcomings, and less so when I find bugs that students are ignorant of or have tried to hide. You should also include references to any materials that you referred to while working on the project. Please do not include special instructions for compilation. The compilation and running of the tests will be automated (see details below) and your instructions will not be followed.

Please make sure you include only the source files, not the object files. Also, make sure that your files are not in a subdirectory within the

tar ball. Before you submit this single file, please use the following commands to check and make sure you have everything in the `project1.tgz` file, using the following script:

```
> tar xvzf project1.tgz
> make
```

This should put all of your source files in the current directory, and compile your two programs.

Submit your `project1.tgz` file via **handin.cs.clemson.edu**. You must name your archive **project1.tgz**, and it must compile and run. We will compile your code using your **Makefile**, and run it using the syntax described in Listings 1 and 3.

1 Grading

Your project will be graded based on the results of functional testing and the design of your code. We will run several tests to make sure it works properly and correctly handles various error conditions. We will test your programs against custom test programs that we have written. Your code should not crash. You will receive 10% credit if your code successfully compiles, and 10% for code style and readability. The rest of your score will be determined by the number of tests that your programs pass.

Your source materials should be readable, properly documented and use good coding practices (avoid magic numbers, consistent indentation, use appropriately-named variables, etc). Your code should be -Wall clean (you should not have any warnings during compilation). **Our testing of your code will be thorough. Be sure you test your application thoroughly.**

2 Collaboration

You will work independently on this project. You must *not* discuss the problem or the solution with classmates, and all of your code must be your own code.

You may, of course, discuss the project with me, and you may discuss conceptual issues related to the project that we have already discussed in lecture on Canvas. For example, it would be OK to discuss how

ptrace works (in general) since this is something we have discussed in class. It is not OK to discuss how you used **ptrace** to solve any part of this project. **Collaborating with peers on this project, in this or any other manner will be treated as academic misconduct.**