CS 4420/5420 Operating Systems
Programming Assignment 2

Utilizing the ptrace System Call

PART 1

Total Points: 20

Due: Wednesday, November 6, 2019 (11:59pm)

Overview:

In the second project, you will familiarize yourself with and make use of important Linux/Unix system calls, including fork(), wait(), ptrace(), mkfifo(), etc.

The first part of this project involves using ptrace() to monitor the system call activities of a process. More specifically, ptrace allows a parent process (aka the tracer) to observe and control the execution of another process (aka the tracee) and access the tracee's memory and registers.

You will start by reading the following article ("Playing with ptrace, Part 1", published in the Linux Journal)

https://www.linuxjournal.com/article/6100

Additionally, read the following note that describes how to adopt the code presented in the above article to more modern x86_64 processors.

http://theantway.com/2013/01/notes-for-playing-with-ptrace-on-64-bits-ubuntu-12-10/

In addition to these two sources, also look at the Linux manpages for ptrace() (use the "man ptrace" command), and review the lecture notes (slides chapter 3) on Unix systems programming, especially fork(), wait(), various exec() variants.

Technical Details:

As described in the Linux Journal article, on an x86 (32-bit) system, a Linux system call, such as write(2, "Hello", 5) will translate to something like

```
movl 4, %eax       // system call number
movl 2, %ebx       // first syscall argument: file descriptor STDOUT in this case)
movl $hello, %ecx  // second syscall argument: address of write buffer
movl 5, %edx       // third syscall argument: number of bytes to write
int 0x80           // system call instruction on x86
```

The registers eax, ebx, ecx, edi, esi, ebp are all x86 general purpose registers. A 64-bit version of x86 (referred to as x86_64) has the same registers, but also provides registers rax, rbx, rcx, rdi, rsi, rbp. The 'r' versions refer to the 64-bit register extensions of the original registers.

Anytime Linux enters (or exits) a system call, it first checks whether the process executing the system call is a tracee. If this is the case, it stops and allows the tracer to control the tracee (i.e., once upon starting the system call and once upon returning from the system call). The example in the Linux Journal article (I patched the code to run on x86_64 and provide these files on blackboard) explains in detail how the tracing works.

Enabling a process to be traced is done after the parent executed a fork() system call (but in the child branch of the program), and before executing the actual program. The ptrace() call for this is

ptrace(PTRACE_TRACEME, 0, NULL, NULL);

In the parent branch of the program, the parent should execute wait() in a loop. The wait() system call will return anytime the child performs certain actions. These actions include an exit(), but also anytime a traced child enters or exits a system call. A parent would call wait() with an argument (example: wait(&status)) that upon return of wait() contains an encoding of the reason for wait() to return. In the example program, a call (WIFEXITED(status)) (see "man -s2 wait" for more details) will be true if wait() returned because the child performed an exit().

If wait() returns for reasons other than an exit() of the child, we assume here that it is because ptrace() stopped the child. If that happens, the parent can make calls to ptrace() to obtain various information about the child. The kind of information/action requested depends on the first argument to ptrace(). Examples include

- ptrace(PTRACE_PEEKUSER, child, 8 * ORIG_RAX, NULL)
- ptrace(PTRACE_PEEKDATA, child, addr , NULL)

The first example (PEEKUSER is used to retrieve child register values) retrieves the original value of the rax register (i.e., the register storing the system call number). The second example (PEEK_DATA is used to access the child's address space) obtains a data value (from address 'addr') from the tracee.

Please note that 32-bit and 64-bit Linux versions use different registers for passing system call arguments (If in doubt, it is much more likely that the system you're working on is a 64-bit version).

32-bit processors use the following registers

syscall	arg0	arg1	arg2	arg3	arg4	arg5
%eax	%ebx	%ecx	%edx	%esi	%edi	%ebp

In contrast, 64-bit processors use

syscall	arg0	arg1	arg2	arg3	arg4	arg5
%rax	%rdi	%rsi	%rdx	%r10	%r8	%r9

Detailed Requirements:

Use the provided example (modified from the Linux Journal article) as a starting point and extend the code in the following way:

- The parent will track the tracee and log the events in a log file (the name of the log file should be provided as command argument when the parent is started from a shell.

- The parent should track entry and exit of the following system calls: open(), read(), write(), close() and exit().

- For each call, the program should extract the system call arguments, and the return values and write them into the log file.

  Example output:

  ```
  write(2, “Hello World”, 12) - - > 1
  ```

  In this case, 2 is the file descriptor, “Hello World” is the buffer content, 12 is the number of bytes to write, and 1 is the return value from write(). (Note: limit the number of bytes for a string to be written to a reasonable limit (like max. 20 bytes). You can do this with the printf() function; example: printf(“%.*s”, limit, long_string) prints only limit characters of string ‘long_string’).

  Note that the output of this part of the program is very similar to (or identical) to the output of the Linux "strace" command.

- Finally, provide a summary about the total volume of data written to files (by write() system calls) and the total amount of data read from files (by the read() system calls). This should be reported per file descriptor (the first argument to read() and write()).

  Example output:

  ```
  file descriptor: 0    read volume: 10000 bytes   write volume: 412 bytes
  file descriptor: 1    read volume: 4 bytes       write volume: 1900000 bytes
  file descriptor: 2    read volume: 10 bytes      write volume: 4 bytes
  …
  ```

  Note: It's reasonable to assume that any program you use this with (see below) is running for a limited amount of time and then exits (i.e., no infinite while() loops or something like that).

- Your program should be named "4420_ptrace_1". Make sure you create (and submit) a makefile

- Your program should be called (i.e., from a shell) the following way:

```
bash> ./4420_ptrace_1 <logfile_name>  <tracee_program_name>  <tracee_arg0> < tracee_arg1>…
```

where <logfile_name> is the name of the file for all the output and <tracee_program_name> is the name of the tracee command (i.e., program = executable file) to execute.

Example:

```
bash> ./4420_ptrace_1 my_log_file ls -l -d
```

Note: In the Linux Journal example, the example program simply always executes the "ls" command. In your program, you will need to be able to execute any program a user specifies, including passing the correct command arguments to the program. There are various variations of the Unix exec() system call (example: execlp(), execvp(), etc.; see lecture slides Chapter 3) that allow you to execute a program and pass command arguments to it.

- Submit the first part of this project in the usual way by using the command: "/home/drews/bin/442submit 3" from pu1, pu2, or pu3 (NOT p2!)

Grading Criteria:

Max. points: 20

Breakdown of points:

- Error handling: calling the tool with invalid commands is caught properly and an error is reported (2 points)
- The basic mechanism of fork(), exec(), wait()-loop is overall correctly implemented (10 points)
- The system call operands and the return values are correctly printed (4 points)
- The read()/write() volume summary is correctly reported in the logfile (4 points)