How debuggers work: Part 1 - Basics (https://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1)

January 23, 2011 at 07:40 **Tags** Articles (https://eli.thegreenplace.net/tag/articles) , Debuggers (https://eli.thegreenplace.net/tag/debuggers) , Programming (https://eli.thegreenplace.net/tag/programming)

This is the first part in a series of articles on how debuggers work. I'm still not sure how many articles the series will contain and what topics it will cover, but I'm going to start with the basics.

In this part

I'm going to present the main building block of a debugger's implementation on Linux - the `ptrace` system call. All the code in this article is developed on a 32-bit Ubuntu machine. Note that the code is very much platform specific, although porting it to other platforms shouldn't be too difficult.

Motivation

To understand where we're going, try to imagine what it takes for a debugger to do its work. A debugger can start some process and debug it, or attach itself to an existing process. It can single-step through the code, set breakpoints and run to them, examine variable values and stack traces. Many debuggers have advanced features such as executing expressions and calling functions in the debbugged process's address space, and even changing the process's code on-the-fly and watching the effects.

Although modern debuggers are complex beasts [1], it's surprising how simple is the foundation on which they are built. Debuggers start with only a few basic services provided by the operating system and the compiler/linker, all the rest is just a simple matter of programming (http://en.wikipedia.org/wiki/Small_matter_of_programming).

Linux debugging - `ptrace`

The Swiss army knife of Linux debuggers is the `ptrace` system call [2]. It's a versatile and rather complex tool that allows one process to control the execution of another and to peek and poke at its innards [3]. `ptrace` can take a mid-sized book to explain fully, which is why I'm just going to focus on some of its practical uses in examples.

Let's dive right in.

Stepping through the code of a process

I'm now going to develop an example of running a process in "traced" mode in which we're going to single-step through its code - the machine code (assembly instructions) that's executed by the CPU. I'll show the example code in parts, explaining each, and in the end of the article you will find a link to download a complete C file that you can compile, execute and play with.

The high-level plan is to write code that splits into a child process that will execute a user-supplied command, and a parent process that traces the child. First, the main function:

```c
int main(int argc, char** argv)
{
    pid_t child_pid;

    if (argc < 2) {
        fprintf(stderr, "Expected a program name as argument\n");
        return -1;
    }

    child_pid = fork();
    if (child_pid == 0)
        run_target(argv[1]);
    else if (child_pid > 0)
        run_debugger(child_pid);
    else {
        perror("fork");
        return -1;
    }

    return 0;
}
```

Pretty simple: we start a new child process with `fork` [4]. The `if` branch of the subsequent condition runs the child process (called "target" here), and the `else if` branch runs the parent process (called "debugger" here).

Here's the target process:

```c
void run_target(const char* programname)
{
    procmsg("target started. will run '%s'\n", programname);

    /* Allow tracing of this process */
    if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) {
        perror("ptrace");
        return;
    }

    /* Replace this process's image with the given program */
    execl(programname, programname, 0);
}
```

The most interesting line here is the `ptrace` call. `ptrace` is declared thus (in `sys/ptrace.h`):

```
long ptrace(enum __ptrace_request request, pid_t pid,
                 void *addr, void *data);
```

The first argument is a *request*, which may be one of many predefined `PTRACE_*` constants. The second argument specifies a process ID for some requests. The third and fourth arguments are address and data pointers, for memory manipulation. The `ptrace` call in the code snippet above makes the `PTRACE_TRACEME` request, which means that this child process asks the OS kernel to let its parent trace it. The request description from the man-page is quite clear:

> Indicates that this process is to be traced by its parent. Any signal (except SIGKILL) delivered to this process will cause it to stop and its parent to be notified via wait(). **Also, all subsequent calls to exec() by this process will cause a SIGTRAP to be sent to it, giving the parent a chance to gain control before the new program begins execution**. A process probably shouldn't make this request if its parent isn't expecting to trace it. (pid, addr, and data are ignored.)

I've highlighted the part that interests us in this example. Note that the very next thing `run_target` does after `ptrace` is invoke the program given to it as an argument with `execl`. This, as the highlighted part explains, causes the OS kernel to stop the process just before it begins executing the program in `execl` and send a signal to the parent.

Thus, time is ripe to see what the parent does:

```
void run_debugger(pid_t child_pid)
{
    int wait_status;
    unsigned icounter = 0;
    procmsg("debugger started\n");

    /* Wait for child to stop on its first instruction */
    wait(&wait_status);

    while (WIFSTOPPED(wait_status)) {
        icounter++;
        /* Make the child execute another instruction */
        if (ptrace(PTRACE_SINGLESTEP, child_pid, 0, 0) < 0) {
            perror("ptrace");
            return;
        }

        /* Wait for child to stop on its next instruction */
        wait(&wait_status);
    }

    procmsg("the child executed %u instructions\n", icounter);
}
```

Recall from above that once the child starts executing the `exec` call, it will stop and be sent the `SIGTRAP` signal. The parent here waits for this to happen with the first `wait` call. `wait` will return once something interesting happens, and the parent checks that it was because the child was stopped (`WIFSTOPPED` returns true if the child process was stopped by delivery of a signal).

What the parent does next is the most interesting part of this article. It invokes `ptrace` with the `PTRACE_SINGLESTEP` request giving it the child process ID. What this does is tell the OS - *please restart the child process, but stop it after it executes the next instruction*. Again, the parent waits for the child to stop and the loop continues. The loop will terminate when the signal that came out of the `wait` call wasn't about the child stopping. During a normal run of the tracer, this will be the signal that tells the parent that the child process exited (`WIFEXITED` would return true on it).

Note that `icounter` counts the amount of instructions executed by the child process. So our simple example actually does something useful - given a program name on the command line, it executes the program and reports the amount of CPU instructions it took to run from start to finish. Let's see it in action.

A test run

I compiled the following simple program and ran it under the tracer:

```
#include <stdio.h>


int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

To my surprise, the tracer took quite long to run and reported that there were more than 100,000 instructions executed. For a simple `printf` call? What gives? The answer is very interesting [5]. By default, `gcc` on Linux links programs to the C runtime libraries dynamically. What this means is that one of the first things that runs when any program is executed is the dynamic library loader that looks for the required shared libraries. This is quite a lot of code - and remember that our basic tracer here looks at each and every instruction, not of just the `main` function, but *of the whole process*.

So, when I linked the test program with the `-static` flag (and verified that the executable gained some 500KB in weight, as is logical for a static link of the C runtime), the tracing reported only 7,000 instructions or so. This is still a lot, but makes perfect sense if you recall that `libc` initialization still has to run before `main`, and cleanup has to run after `main`. Besides, `printf` is a complex function.

Still not satisfied, I wanted to see something *testable* - i.e. a whole run in which I could account for every instruction executed. This, of course, can be done with assembly code. So I took this version of "Hello, world!" and assembled it:

```
section    .text
    ; The _start symbol must be declared for the linker (ld)
    global _start

_start:

    ; Prepare arguments for the sys_write system call:
    ;   - eax: system call number (sys_write)
    ;   - ebx: file descriptor (stdout)
    ;   - ecx: pointer to string
    ;   - edx: string length
    mov    edx, len
    mov    ecx, msg
    mov    ebx, 1
    mov    eax, 4

    ; Execute the sys_write system call
    int    0x80

    ; Execute sys_exit
    mov    eax, 1
    int    0x80

section   .data
msg db    'Hello, world!', 0xa
len equ    $ - msg
```

Sure enough. Now the tracer reported that 7 instructions were executed, which is something I can easily verify.

Deep into the instruction stream

The assembly-written program allows me to introduce you to another powerful use of `ptrace` - closely examining the state of the traced process. Here's another version of the `run_debugger` function:

```
void run_debugger(pid_t child_pid)
{
    int wait_status;
    unsigned icounter = 0;
    procmsg("debugger started\n");

    /* Wait for child to stop on its first instruction */
    wait(&wait_status);

    while (WIFSTOPPED(wait_status)) {
        icounter++;
        struct user_regs_struct regs;
        ptrace(PTRACE_GETREGS, child_pid, 0, &regs);
        unsigned instr = ptrace(PTRACE_PEEKTEXT, child_pid, regs.eip, 0);

        procmsg("icounter = %u.  EIP = 0x%08x.  instr = 0x%08x\n",
                    icounter, regs.eip, instr);

        /* Make the child execute another instruction */
        if (ptrace(PTRACE_SINGLESTEP, child_pid, 0, 0) < 0) {
            perror("ptrace");
            return;
        }

        /* Wait for child to stop on its next instruction */
        wait(&wait_status);
    }

    procmsg("the child executed %u instructions\n", icounter);
}
```

The only difference is in the first few lines of the `while` loop. There are two new `ptrace` calls. The first one reads the value of the process's registers into a structure. `user_regs_struct` is defined in `sys/user.h`. Now here's the fun part - if you look at this header file, a comment close to the top says:

```
/* The whole purpose of this file is for GDB and GDB only.
   Don't read too much into it. Don't use it for
   anything other than GDB unless know what you are
   doing.  */
```

Now, I don't know about you, but it makes *me* feel we're on the right track :-) Anyway, back to the example. Once we have all the registers in `regs`, we can peek at the current instruction of the process by calling `ptrace` with `PTRACE_PEEKTEXT`, passing it `regs.eip` (the extended instruction pointer on x86) as the address. What we get back is the instruction [6]. Let's see this new tracer run on our assembly-coded snippet:

```
$ simple_tracer traced_helloworld
[5700] debugger started
[5701] target started. will run 'traced_helloworld'
[5700] icounter = 1.  EIP = 0x08048080.  instr = 0x00000eba
[5700] icounter = 2.  EIP = 0x08048085.  instr = 0x0490a0b9
[5700] icounter = 3.  EIP = 0x0804808a.  instr = 0x000001bb
[5700] icounter = 4.  EIP = 0x0804808f.  instr = 0x000004b8
[5700] icounter = 5.  EIP = 0x08048094.  instr = 0x01b880cd
Hello, world!
[5700] icounter = 6.  EIP = 0x08048096.  instr = 0x000001b8
[5700] icounter = 7.  EIP = 0x0804809b.  instr = 0x000080cd
[5700] the child executed 7 instructions
```

OK, so now in addition to `icounter` we also see the instruction pointer and the instruction it points to at each step. How to verify this is correct? By using `objdump -d` on the executable:

```
$ objdump -d traced_helloworld

traced_helloworld:     file format elf32-i386


Disassembly of section .text:

08048080 <.text>:
 8048080:     ba 0e 00 00 00          mov    $0xe,%edx
 8048085:     b9 a0 90 04 08          mov    $0x80490a0,%ecx
 804808a:     bb 01 00 00 00          mov    $0x1,%ebx
 804808f:     b8 04 00 00 00          mov    $0x4,%eax
 8048094:     cd 80                   int    $0x80
 8048096:     b8 01 00 00 00          mov    $0x1,%eax
 804809b:     cd 80                   int    $0x80
```

The correspondence between this and our tracing output is easily observed.

Attaching to a running process

As you know, debuggers can also attach to an already-running process. By now you won't be surprised to find out that this is also done with `ptrace`, which can get the `PTRACE_ATTACH` request. I won't show a code sample here since it should be very easy to implement given the code we've already gone through. For educational purposes, the approach taken here is more convenient (since we can stop the child process right at its start).

The code

The complete C source-code of the simple tracer presented in this article (the more advanced, instruction-printing version) is available here (https://github.com/eliben/code-for-blog/blob/master/2011/simple_tracer.c). It compiles cleanly with `-Wall -pedantic --std=c99` on version 4.4 of `gcc`.

Conclusion and next steps

Admittedly, this part didn't cover much - we're still far from having a real debugger in our hands. However, I hope it has already made the process of debugging at least a little less mysterious. `ptrace` is truly a versatile system call with many abilities, of which we've sampled only a few so far.

Single-stepping through the code is useful, but only to a certain degree. Take the C "Hello, world!" sample I demonstrated above. To get to `main` it would probably take a couple of thousands of instructions of C runtime initialization code to step through. This isn't very convenient. What we'd ideally want to have is the ability to place a breakpoint at the entry to `main` and step from there. Fair enough, and in the next part of the series I intend to show how breakpoints are implemented.

References

I've found the following resources and articles useful in the preparation of this article:

- Playing with ptrace, Part I (http://www.linuxjournal.com/article/6100?page=0,1)
- How debugger works (http://www.alexonlinux.com/how-debugger-works)

[1] I didn't check but I'm sure the LOC count of `gdb` is at least in the six-figures range.

[2] Run `man 2 ptrace` for complete enlightment.

[3] *Peek* and *poke* are well-known system programming jargon (http://www.jargon.net/jargonfile/p/peek.html) for directly reading and writing memory contents.

[4] This article assumes some basic level of Unix/Linux programming experience. I assume you know (at least conceptually) about `fork`, the `exec` family of functions and Unix signals.

[5] At least if you're as obsessed with low-level details as I am :-)

[6] A word of warning here: as I noted above, a lot of this is highly platform specific. I'm making some simplifying assumptions - for example, x86 instructions don't have to fit into 4 bytes (the size of `unsigned` on my 32-bit Ubuntu machine). In fact, many won't. Peeking at instructions meaningfully requires us to have a complete disassembler at hand. We don't have one here, but real debuggers do.

For comments, please send me ✉ an email (mailto:eliben@gmail.com), or reach out on on Twitter (https://twitter.com/elibendersky).

🡅 Back to top