

(gdb) break *0x972

Debugging, GNU± Linux and WebHosting and ... and ...

Research

Work news
PhD Thesis
Publications
CV
About me

Unrelated to work

Overview
Photo selection
Around Grenoble logs
Diving logbook
Personal library
Capoeira songbook

Weblog

Debugging (17)

How Does a C Debugger Work? (GDB Ptrace/x86 example)

When you use GDB, you can see that it has a complete control over your application process. Hit Ctrl-C while the application is running and the process execution stops, and GDB shows its current location, stack trace, etc.

But how can it do it?

How they don't work?

Let's start first with how it *doesn't* work. It doesn't simulate the execution, by reading and interpreting the binary instructions. It could, and that would work (that the way **valgrind** memory debugger works), but that would be too slow. Valgrind slows the application 1000x down, GDB doesn't. That's also the way virtual machines like **Qemu** work.

So, what's the trick? Black magic! ... no, that would be too easy.

Another guess? ... ? Hacking! yes, there's a good deal of that, plus help from the OS kernel.

Gdb (12)
Gdb.py (11)
Hack (11)
I3 (2)
Latex (1)
Project (6)
.*rc (4)
Strace (3)
Systemd (1)
Vulgarisation (6)
Webalbums (6)
Webhosting (1)
Work (12)

Last Posts

GDB, please set a
breakpoint on all my
functions

GDB scheduler
locking, function calls
and multi-threading

i3 named workspaces

Simple strace
debugging

Looking for a
Developer/Researcher

First of all, there's one thing to know about Linux processes: *parent* processes can get additional information about their children, in particular the ability to `ptrace` them. And, you can guess, the debugger is the parent of the debuggee process (or it becomes, processes can adopt a child in Linux :-).

Linux Ptrace API

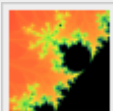
Linux `ptrace` API allows a (debugger) process to access low-level information about another process (the debuggee). In particular, the debugger can:

- read and write the debuggee's memory: `PTRACE_PEEKTEXT`, `PTRACE_PEEKUSER`, `PTRACE_POKE...`
- read and write the debuggee's CPU registers: `PTRACE_GETREGSET`, `PTRACE_SETREGS`,
- be notified of system events: `PTRACE_O_TRACEEXEC`, `PTRACE_O_TRACECLONE`, `PTRACE_O_EXITKILL`, `PTRACE_SYSCALL` (you can recognize the `exec` syscall, `clone`, `exit`, and all the other syscalls)
- control its execution: `PTRACE_SINGLESTEP`, `PTRACE_KILL`, `PTRACE_INTERRUPT`, `PTRACE_CONT` (notice the CPU single-stepping here)
- alter its signal handling: `PTRACE_GETSIGINFO`, `PTRACE_SETSIGINFO`

Position in April 2017

-see all-

Profiles



Kevin
3,424
2 28 50

GitHub (perso)
GitHub (research)

Flux RSS

: Blog

: Liens



How is Ptrace implemented?

Ptrace implementation is outside of the scope of this post, but I don't want to move the black-box one step above, so let me explain quickly how it works (I'm no kernel expert, please correct me if I'm wrong and excuse me if I simplify too much :-).

Ptrace [is part of Linux kernel](#), so it has access to [all](#) the kernel-level information about the process:

- [reading](#) and writing data? Linux's [copy_to/from_user](#)
- [accessing](#) CPU registers? easy with [copy_regset_to/from_user](#). (There is nothing complicated here, as CPU registers are saved somewhere in Linux's [struct task_struct](#) * scheduler structures, when the process is unscheduled.
- [changing](#) the signal handling? update field [last_siginfo](#)
- [single-stepping](#)? set the right flag ([ARM](#), [x86](#)) on the task structure and, before triggering the execution, on the processor.
- Ptrace is also *hooked* (search for function [ptrace_event](#)) in many scheduling operations, so that it can send a [SIGTRAP](#) signal to the debugger if requested ([PTTRACE_0_TRACEEXEC](#) option and its family).

What about systems without Ptrace?

The explanation above targeted Linux native debugging, but it's valid for most of the other environments. To get a clue on what GDB asks to its

different targets, you can take a look at the operations of its [target stack](#).

In this target interface, you can see all of the high-level operations required for C debugging:

```
struct target_ops
{
    struct target_ops *beneath;    /* To the target under this one. */
    const char *to_shortname; /* Name this target type */
    const char *to_longname; /* Name for printing */
    const char *to_doc;        /* Documentation. Does not include
trailing
                               newline, and starts with a one-line descrip-
                               tion (probably similar to to_longname). */

    void (*to_attach) (struct target_ops *ops, const char *, int);
    void (*to_fetch_registers) (struct target_ops *, struct regcache *,
int);
    void (*to_store_registers) (struct target_ops *, struct regcache *,
int);
    int (*to_insert_breakpoint) (struct target_ops *, struct gdbarch *,
                                struct bp_target_info *);
    int (*to_insert_watchpoint) (struct target_ops *,
                                CORE_ADDR, int, int, struct expression *);
    ...
}
```

The generic part of GDB calls these functions, and the target-specific parts implement them. It is (conceptually) shaped as a stack, or a pyramid: the top of the stack is quite generic, for instance:

- OS specific ([Linux](#))
- Local or [remote](#) debugging
- Debug-method specific ([ptrace](#), [ttrace](#))
- Instruction-set specific (Linux [ARM](#), Linux [x86](#))

The [remote](#) target is interesting, as it splits the execution stack between two "computers", through a communication protocol (TCP/IP, serial port).

The remote part can be [gdbserver](#), running in another Linux box. But it can also be an interface to a hardware-debugging port (JTAG) or a virtual machine hypervisor (e.g [Qemu](#)), that will play the role of the kernel+ptrace. Instead of querying the OS kernel structures, the remote debugger stub will query the hypervisor structures, or directly the hardware registers of the processor.

For further reading about this remote protocol, Embecosm wrote a [detail guide about the different messages](#). [Gdbserver event-processing loop](#) is there, and [Qemu gdb-server stub](#) is also online.

To sum up

We can see here that all the low-level mechanisms required to implement a debugger are there, provided by this [ptrace](#) API:

- Catch the `exec` syscall and block the start of the execution,
- Query the CPU registers to get the process's current instruction and stack location,
- Catch for `clone/fork` events to detect new threads,
- Peek and poke data addresses to read and alter memory variables.

But is that all a debugger does? no, that just the very low level parts ... It also deals with symbol handling. That's link between the binary code and the program sources. And one thing is still missing, maybe the most important one: breakpoints! I'll first explain how breakpoints work as it's quite interesting and tricky, then I'll come back on symbol management.

Breakpoints are not part of Ptrace API

As we've seen above, breakpoints are not part of `ptrace` API services. But we can alter the memory, and receive the debuggee's signals. You can't see the link? That's because breakpoint implementation is quite tricky and hacky! Let's examine how to set a breakpoint at a given address:

1. The debugger reads (`ptrace peek`) the binary instruction stored at this address, and saves it in its data structures.
2. It writes an *invalid* instruction at this location. What ever this instruction, it just has to be invalid.
3. When the debuggee reaches this invalid instruction (or, put more correctly, the processor, setup with the debuggee memory context), the it won't be able to

execute it (because it's invalid).

4. In modern multitask OSes, an invalid instruction doesn't crash the whole system, but it gives the control back to the OS kernel, by raising an *interruption* (or a fault).
5. This interruption is translated by Linux into a **SIGTRAP** signal, and transmitted to the process ... or to it's parent, as the debugger asked for.
6. The debugger gets the information about the signal, and checks the value of the debuggee's instruction pointer (i.e., *where* the trap occurred). If the **IP** address is in its breakpoint list, that means it's a debugger breakpoint (otherwise, it's a fault in the process, just pass the signal and let it crash).
7. Now that the debuggee is stopped at the breakpoint, the debugger can let its user do what ever s/he wants, until it's time to continue the execution.
8. To continue, the debugger needs to 1/ write the correct instruction back in the debuggee's memory, 2/ single-step it (continue the execution for one CPU instruction, with `ptrace single-step`) and 3/ write the invalid instruction back (so that the execution can stop again next time). And 4/, let the execution flow normally.

Neat, isn't it? As a side remark, you can notice that this algorithm will not work if not *all* the threads are stopped at the same time (because running threads may pass the breakpoint when the *valid* instruction is in place). I won't detail the way GDB guys solved it, but it's discussed in detail this paper: [Non-stop Multi-threaded Debugging in GDB](#). Put briefly, they write the instruction somewhere else in memory, set the instruction pointer to

that location and single-step the processor. But the problem is that some instructions are *address*-related, for example the jumps and conditional jumps ...

Symbol and debug information handling

Now, let's come back to the symbol and debug information handling aspect. I didn't study that part into details, so I'll only present an overview.

First of all, can we debug *without* debug information and symbol addresses? The answer is yes, as, as we've seen above, all the low-level commands deal with CPU registers and memory addresses, and not source-level information. Hence, the link with the sources are only for the user's convenience. Without debug information, you'll see your application the way the processor (and the kernel) see it: as binary (assembly) instructions and memory bits. GDB doesn't need any further information to translate binary data into CPU instructions:

```
(gdb) x/10x $pc # hexadecimal representation
0x402c60: 0x56415741 0x54415541 0x55f48949 0x4853fd89
0x402c70: 0x03a8ec81 0x8b480000 0x8b48643e 0x00282504
0x402c80: 0x89480000 0x03982484
(gdb) x/10i $pc # Instruction representation
=> 0x402c60: push    %r15
0x402c62: push    %r14
0x402c64: push    %r13
```



```
0x402c66:  push    %r12
0x402c68:  mov     %rsi,%r12
0x402c6b:  push    %rbp
0x402c6c:  mov     %edi,%ebp
0x402c6e:  push    %rbx
0x402c6f:  sub     $0x3a8,%rsp
0x402c76:  mov     (%rsi),%rdi
```

Now if we add symbol handling information, GDB can match addresses with symbol names:

```
(gdb) $pc
$1 = (void (*)( )) 0x402c60 <main>
```

You can list the symbols of an ELF binary with `nm -a $file`:

```
nm -a /usr/lib/debug/usr/bin/ls.debug | grep " main"
0000000000402c60 T main
```

GDB will also be able to display the stack trace (more on that later), but with a limited interest:

```
(gdb) where
#0  write ()
#1  0x0000003d492769e3 in _IO_new_file_write ()
#2  0x0000003d49277e4c in new_do_write ()
```

```
#3 _IO_new_do_write ()
#4 0x0000003d49278223 in _IO_new_file_overflow ()
#5 0x00000000004085bb in print_current_files ()
#6 0x000000000040431b in main ()
```

We've got the PC addresses, the corresponding function, but that's it. Inside a function, you'll need to debug in assembly!

Now let's add debug information: that's the DWARF standard, `gcc -g` option. I'm not very familiar with this standard, but I know it provides:

- address to line and line to address mapping
- data type definitions, including typedefs and structures
- local variables and function parameters, with their type

Try `dwarfdump` to see the information embedded in your binaries. `addr2line` also uses this information:

```
$ dwarfdump /usr/lib/debug/usr/bin/ls.debug | grep 402ce4
0x00402ce4 [1289, 0] NS
$ addr2line -e /usr/lib/debug/usr/bin/ls.debug 0x00402ce4
/usr/src/debug/coreutils-8.21/src/ls.c:1289
```

Many source-level debugging commands will rely on this information, like the command `next`, that sets a breakpoint at the address of the next line,

the `print` command that relies on the types to display the variables in the right type (`char`, `int`, `float`, instead of binary/hexadecimal!).

Last words

We've seen many aspects of debugger's internals, so I'll just say a few words of the last points:

- the stack trace is "unwinded" from the current frame (`$sp` and `$bp/#fp`) upwards, one frame at a time. Functions' name, parameters and local variables are found in the debug information.
- `watchpoints` are implemented (if available) with the help of the processor: write in its registers which addresses should be monitored, and it will raise an exception when the memory is read or written. If this support is not available, or if you request more watchpoints than the processor supports ... then the debugger falls back to "hand-made" watchpoints: execute the application instruction by instruction, and check if the current operation touches a watchpointed address. Yes, that's very slow!
- Reverse debugging can be done this way too, record the effect of each instruction, and apply it backward for reverse execution.
- Conditional breakpoints are normal breakpoints, except that, internally, the debugger checks the conditions before giving the control to the user. If the condition is not matched, the execution is silently continued.

And play with `gdb gdb`, or better (way better actually), `gdb --pid $(pidof gdb)`, because two debuggers in the *same* terminal is insane :-).
Another great thing for learning is system debugging:

```
qemu-system-i386 -gdb tcp::1234
gdb --pid $(pidof qemu-system-i386)
gdb /boot/vmlinuz --exec "target remote localhost:1234"
```

but I'll keep that for another article!

Thursday, November 13, 2014 - 11 comments

Publié dans : [debugging](#)

1 - Andrey :

Thanks! Nice article! Can i translate article to russian language?

le Wednesday, November 26, 2014 à 21:25:26

[@répondre](#) [#lien](#)

2 - Antoine :

Very good article, thanks :-)

le Saturday, May 09, 2015 à 11:35:23

[@répondre](#) [#lien](#)

3 - Sergey :

Thanks! This really helped :)

le Friday, August 14, 2015 à 07:45:08

[@répondre](#) [#lien](#)

4 - JY :

thanks! great article!

le Thursday, July 14, 2016 à 11:42:39

@répondre #lien

5 - Ravi :

Thank you so much for your detailed information about gdb internals.

le Monday, December 12, 2016 à 16:12:08

@répondre #lien

6 - Shakaiba Majeed :

Very Good article... easier to grasp!

le Wednesday, January 04, 2017 à 05:52:05

@répondre #lien

7 - Darren :

You non-stop multi-threaded debugging in gdb link is dead

le Tuesday, July 04, 2017 à 22:11:50

@répondre #lien

8 - Kevin :

@Darren : I fixed it, thanks !

--> <https://communities.mentor.com/docs/DOC-3037>

le Wednesday, July 05, 2017 à 07:14:25

@répondre #lien

9 - Prash :

Very well and neatly explained. Thank you so much.

le Monday, April 16, 2018 à 23:20:48

10 - Gang :

@répondre #lien

Nice article. Thanks.

le Tuesday, May 08, 2018 à 12:21:42




@répondre #lien

11 - AJ :

Awesome explanation!!

le Wednesday, October 31, 2018 à 06:26:40

@répondre #lien

B *I* U ~~S~~ |   

comment

Pseudo:

E-mail (optionnal):

Website (optionnal):

Sum of: **eight + five**

☐ Use a cookie to remember these informations?

Send

Preview

Blog tenu par **kevin**, licence CC BY-SA 4.0 - Publié avec BlogoText - Thème par Timo van Neerden
Hébergé par l'Association Groupe PulseHeberg - 91 rue du Faubourg St Honoré - 75008 Paris