

Crossing borders between user space and kernel space

Connecting GDB with Kernel debugging tools

Alexandra Petlanová Hájková

October 4, 2026



GDB: crossing the user–kernel boundary

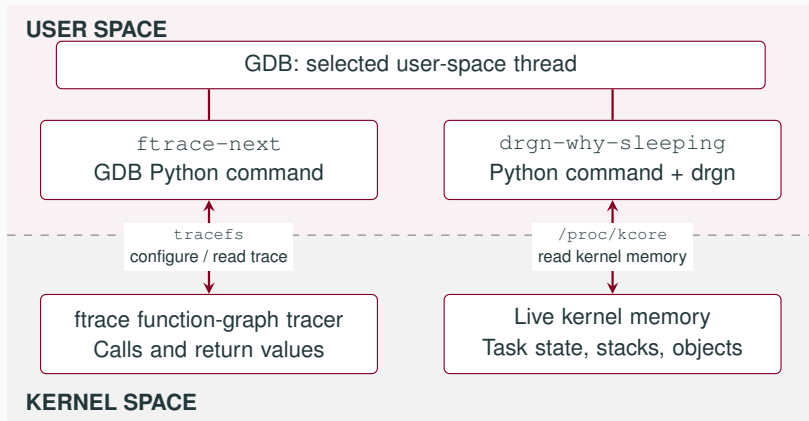
USER SPACE DEBUGGING

- Tracing failing system calls
 - Inspecting a sleeping thread
-

KERNEL DEBUGGING

- Inspecting a vmcore

How GDB reaches into the kernel



Both paths identify the kernel thread using its Linux TID.

Tracing failing syscalls

- Empty non-blocking pipe
- Both ends are kept open
- Trying to read 1 byte results in `EAGAIN`

```
pipe2 (pipe_fds, O_NONBLOCK | O_CLOEXEC);
```

```
/* No data written; the write end remains open. */  
result = read (pipe_fds[0], &byte, sizeof (byte));  
saved_errno = errno;
```

New ftrace-next command

- Enables kernel tracing for the current thread
- Written using GDB's python API
- Uses ftrace kernel's infrastructure
- Records kernel execution during one GDB `next`
 - ▶ runs current source line, stops at the following line
 1. Enable tracing for the current thread
 2. Resume execution using `next`
 3. When GDB stops, disable tracing and show results
- Captures function return values using function-graph tracer
- If error was returned, shows recorded call paths
- Prototype only, not submitted to any Mailing List

New ftrace-next command

Using tracefs

- Filesystem interface to Linux's tracing infrastructure
- ftrace-next GDB's python command uses tracefs to
 - ▶ Create a separate tracing instance
 - ▶ Select the function-graph tracer and filter by thread ID
 - ▶ Turn recording on and off
 - ▶ Read the recorded trace

New ftrace-next command

```
(gdb) source /home/ahajkova/binutils-gdb/gdb_fttrace.py
(gdb) ftrace-next
```

Tracing instance: /sys/kernel/tracing/instances/...

```
21         saved_errno = errno;
```

__x64_sys_read returned -11. Matching return paths:

```
ksys_read -> vfs_read -> anon_pipe_read returned -11
```

```
__x64_sys_read -> ksys_read -> vfs_read returned -11
```

```
__x64_sys_read -> ksys_read returned -11
```

```
__x64_sys_read returned -11
```

```
(gdb) c
```

Continuing.

```
read returned -1, errno = 11 (EAGAIN)
```

```
[Inferior 1 (process 385621) exited normally]
```

Limitations

- Requires permission to configure tracefs
 - ▶ Running GDB as root
- Tracing must start before the failure
 - ▶ No kernel history is available after syscall have failed
- Recording covers one GDB `next`
 - ▶ Starting too early causes missing the syscall
- Syscall detection uses x86_64 names
 - ▶ Currently matches `__x64_sys_` wrappers
- A blocking call keeps the command waiting
 - ▶ The internal `next` cannot finish
 - ▶ interrupt with `Ctrl-C`

Blocked system call

```
Breakpoint 1, wait_for_pipe_data (read_fd=3) at /home/ahajkova/binutils-gdb/gdb_fttrace.py:17
      result = read (read_fd, &byte, sizeof (byte))
(gdb) source /home/ahajkova/binutils-gdb/gdb_fttrace.py
(gdb) ftrace-next
Tracing instance: /sys/kernel/tracing/instances/...
^C
Program received signal SIGINT, Interrupt.
0x00007ffff7e1854e in __internal_syscall_cancel () from
__x64_sys_read returned -512. Matching return paths:
vfs_read -> anon_pipe_read -> prepare_to_wait_event
ksys_read -> vfs_read -> anon_pipe_read returned -512
__x64_sys_read -> ksys_read -> vfs_read returned -512
__x64_sys_read -> ksys_read returned -512
__x64_sys_read returned -512
```

Blocked system call

- `read()` waiting on an empty blocking pipe instead of failing
- `ftrace-next` is waiting because GDB's `next` cannot finish
- `Ctrl-C` produces a trace of the interruption
`ERESTARTSYS`
 - ▶ kernel interrupted a syscall and may need to restart it
- Why the process is waiting inside the kernel?
 - ▶ inspect the thread/process while it is waiting

New drgn-why-sleeping command

- Submitted to gdb-patches Mailing list: patches
- Bridging user space GDB with live kernel memory
- Written using GDB's python API
- drgn provides access to kernel data and stack traces
- The command maps GDB's current thread to a kernel task using its Linux LWP ID
 - ▶ Lightweight Process ID identifies the thread to the kernel, differs from GDB's thread number
- Inspects the kernel through `/proc/kcore`
- Requires kernel-memory access and debug information
 - ▶ `kernel-debuginfo` installed or using debuginfod

Where is the process sleeping?

- An empty pipe, with both ends kept open
- Nothing writes any data
- A blocking read waits (not returning `EAGAIN`)

```
pipe (pipe_fds); /* No O_NONBLOCK. */
```

```
/* Empty pipe; the write end remains open. */  
result = read (pipe_fds[0], &byte, sizeof (byte));
```

- Example running in a separate terminal
- GDB will attach to it

Where is the process sleeping?

- Attach and inspect the user-space backtrace
- Resume in the background, process returns to pipe wait
- Inspect the running threads kernel state with drgn

```
(gdb) attach PID
```

```
(gdb) bt
```

```
(gdb) continue &
```

```
(gdb) drgn-why-sleeping
```

```
Task state: S
```

```
...
```

```
schedule
```

```
anon_pipe_read
```

```
new_sync_read
```

```
vfs_read
```

```
ksys_read
```

Where is the process sleeping?

<code>schedule</code>	process is sleeping
<code>anon_pipe_read</code>	pipe-read implementation
<code>new_sync_read</code>	generic read wrapper
<code>vfs_read</code>	common file-read layer
<code>ksys_read</code>	syscall handling

- `drgn-why-sleeping` shows kernel stack
- We know `wait` is inside the pipe-read path

Inspecting a kernel frame

- The kernel stack gives us the frame name: `vfs_read`
- Asking for that frame shows its arguments and locals

```
(gdb) drgn-why-sleeping vfs_read
```

```
Task state: S
```

```
vfs_read
```

```
file = (struct file *)0xffff8be4f1b7a600
```

```
buf = (char *)0x7fffffffef227
```

```
count = (size_t)<optimized out>
```

```
pos = (loff_t *)0x0
```

```
ret = (ssize_t)<optimized out>
```

- The `file` argument gives us a starting point for inspecting the open file

Limitations

- Every time the command is called, load kernel and module debug information
- Interpreting kernel frames requires kernel knowledge
- For local targets only
- Text output of kernel frames
- Is this useful in real life?

Further experiments

Haven't tried this yet!

- Look at vmcore when calling drgn from GDB
- Inspect saved kernel tasks, stacks, and objects
- drgn interprets saved kernel memory and exposes kernel objects
- GDB provides the familiar interface