Debugging kernel and modules via gdb

The kernel debugger kgdb, hypervisors like QEMU or JTAG-based hardware interfaces allow to debug the Linux kernel and its modules during runtime using gdb. Gdb comes with a powerful scripting interface for python. The kernel provides a collection of helper scripts that can simplify typical kernel debugging steps. This is a short tutorial about how to enable and use them. It focuses on QEMU/KVM virtual machines as target, but the examples can be transferred to the other gdb stubs as well.

Requirements

- gdb 7.2+ (recommended: 7.4+) with python support enabled (typically true for distributions)

Setup

- Create a virtual Linux machine for QEMU/KVM (see www.linux-kvm.org and www.qemu.org for more details). For cross-development, https://landley.net/aboriginal/bin keeps a pool of machine images and toolchains that can be helpful to start from.

- Build the kernel with CONFIG_GDB_SCRIPTS enabled, but leave CONFIG_DEBUG_INFO_REDUCED off. If your architecture supports CONFIG_FRAME_POINTER, keep it enabled.

- Install that kernel on the guest, turn off KASLR if necessary by adding "nokaslr" to the kernel command line. Alternatively, QEMU allows to boot the kernel directly using -kernel, -append, -initrd command line switches. This is generally only useful if you do not depend on modules. See QEMU documentation for more details on this mode. In this case, you should build the kernel with CONFIG_RANDOMIZE_BASE disabled if the architecture supports KASLR.

- Enable the gdb stub of QEMU/KVM, either

    - at VM startup time by appending "-s" to the QEMU command line

  or

    - during runtime by issuing "gdbserver" from the QEMU monitor console

- cd /path/to/linux-build

- Start gdb: gdb vmlinux

  Note: Some distros may restrict auto-loading of gdb scripts to known safe directories. In case gdb reports to refuse loading vmlinux-gdb.py, add:

```
add-auto-load-safe-path /path/to/linux-build
```

to ~/.gdbinit. See gdb help for more details.

- Attach to the booted guest:

```
(gdb) target remote :1234
```

Examples of using the Linux-provided gdb helpers

- Load module (and main kernel) symbols:

```
(gdb) lx-symbols
loading vmlinux
scanning for modules in /home/user/linux/build
loading @0xffffffffa0020000: /home/user/linux/build/net/netfilter/xt_tc
loading @0xffffffffa0016000: /home/user/linux/build/net/netfilter/xt_pk
loading @0xffffffffa0002000: /home/user/linux/build/net/netfilter/xt_li
loading @0xffffffffa00ca000: /home/user/linux/build/net/packet/af_packe
loading @0xffffffffa003c000: /home/user/linux/build/fs/fuse/fuse.ko
...
loading @0xffffffffa0000000: /home/user/linux/build/drivers/ata/ata_gen
```

- Set a breakpoint on some not yet loaded module function, e.g.:

```
(gdb) b btrfs_init_sysfs
Function "btrfs_init_sysfs" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (btrfs_init_sysfs) pending.
```

- Continue the target:

```
(gdb) c
```

- Load the module on the target and watch the symbols being loaded as well as the breakpoint hit:

```
loading @0xffffffffa0034000: /home/user/linux/build/lib/libcrc32c.ko
loading @0xffffffffa0050000: /home/user/linux/build/lib/lzo/lzo_compres
loading @0xffffffffa006e000: /home/user/linux/build/lib/zlib_deflate/zl
loading @0xffffffffa01b1000: /home/user/linux/build/fs/btrfs/btrfs.ko

Breakpoint 1, btrfs_init_sysfs () at /home/user/linux/fs/btrfs/sysfs.c:
36              btrfs_kset = kset_create_and_add("btrfs", NULL, fs_kobj
```

- Dump the log buffer of the target kernel:

```
(gdb) lx-dmesg
[     0.000000] Initializing cgroup subsys cpuset
[     0.000000] Initializing cgroup subsys cpu
[     0.000000] Linux version 3.8.0-rc4-dbg+ (...
[     0.000000] Command line: root=/dev/sda2 resume=/dev/sda1 vga=0x314
[     0.000000] e820: BIOS-provided physical RAM map:
[     0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff]
[     0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff]
....
```

- Examine fields of the current task struct(supported by x86 and arm64 only):

```
(gdb) p $lx_current().pid
$1 = 4998
(gdb) p $lx_current().comm
$2 = "modprobe\000\000\000\000\000\000\000"
```

- Make use of the per-cpu function for the current or a specified CPU:

```
(gdb) p $lx_per_cpu("runqueues").nr_running
$3 = 1
(gdb) p $lx_per_cpu("runqueues", 2).nr_running
$4 = 0
```

- Dig into hrtimers using the container_of helper:

```
(gdb) set $next = $lx_per_cpu("hrtimer_bases").clock_base[0].active.nex
(gdb) p *$container_of($next, "struct hrtimer", "node")
$5 = {
  node = {
    node = {
      __rb_parent_color = 18446612133355256072,
      rb_right = 0x0 <irq_stack_union>,
      rb_left = 0x0 <irq_stack_union>
    },
    expires = {
      tv64 = 1835268000000
    }
  },
  _softexpires = {
    tv64 = 1835268000000
  },
  function = 0xffffffff81078232 <tick_sched_timer>,
  base = 0xffff88003fd0d6f0,
  state = 1,
  start_pid = 0,
  start_site = 0xffffffff81055c1f <hrtimer_start_range_ns+20>,
  start_comm = "swapper/2\000\000\000\000\000\000"
}
```

List of commands and functions

The number of commands and convenience functions may evolve over the time, this is just a snapshot of the initial version:

```
(gdb) apropos lx
function lx_current -- Return current task
function lx_module -- Find module by name and return the module variable
function lx_per_cpu -- Return per-cpu variable
function lx_task_by_pid -- Find Linux task by PID and return the task_stru
function lx_thread_info -- Calculate Linux thread_info from task variable
lx-dmesg -- Print Linux kernel log buffer
lx-lsmod -- List currently loaded modules
lx-symbols -- (Re-)load symbols of Linux kernel and currently loaded modul
```

Detailed help can be obtained via “help <command-name>” for commands and “help function <function-name>” for convenience functions.