Kernel debugging

Dongyoon Lee

Summary of last lectures

- Kernel data structures
  - list, hash table, red-black tree
  - radix tree, XArray, bitmap array
- Kernel module

Today’s agenda

- Kernel debugging
  - tools, techniques, and tricks

Kernel development cycle

- Write code → Build kernel/modules → Deploy → **Test and debug**
- *Debugging is the real bottleneck* even for experienced kernel developers due to limitations in kernel debugging
- It is important to get used to kernel debugging techniques to save your time and effort

Kernel debugging techniques

- Print debug message: `printk`
- Assert your code: `BUG_ON(c)`, `WARN_ON(c)`
- Analyze kernel panic message
- Debug with QEMU/gdb

Print debug message: `printk`

- Similar to `printf()` in C library
- Need to specify a log level (the default level is `KERN_WARNING` or `KERN_ERR`)

```
KERN_EMERG    /* 0: system is unusable                    */
KERN_ALERT    /* 1: action must be taken immediately      */
KERN_CRIT     /* 2: critical conditions                   */
KERN_ERR      /* 3: error conditions                      */
KERN_WARNING  /* 4: warning conditions                    */
KERN_NOTICE   /* 5: normal but significant condition      */
KERN_INFO     /* 6: informational                         */
KERN_DEBUG    /* 7: debug-level messages                  */

e.g., printk(KERN_DEBUG "debug message from %s:%d\n", __func__, __LINE__);
```

Print debug message: `printk`

- Prints out only messages, which log level is higher than the current level.

```
# Check current kernel log level
$ cat /proc/sys/kernel/printk
    4       4       1       7
#   |       |       |       |
# current default minimum  boot-time-default
# Enable all levels of messages:
$ echo 7 > /proc/sys/kernel/printk
```

- The kernel message buffer is a fixed-size circular buffer.
- If the buffer fills up, it warps around and *you can lose some message*.
- Increasing the buffer size would be helpful a little bit.
  - Add `log_buf_len=1M` to kernel boot parameters (power of 2)

Print debug message: `printk`

- Support additional format specifiers

```
/* function pointers with function name */
"%pF"     versatile_init+0x0/0x110 /* symbol+offset/length */
"%pf"     versatile_init

/* direct code address (e.g., regs->ip) */
"%pS"     versatile_init+0x0/0x110
"%ps"     versatile_init

/* direct code address in stack (e.g., return address) */
"%pB"     prev_fn_of_versatile_init+0x88/0x88

/* Example */
printk("Going to call: %pF\n", p->func);
printk("Faulted at %pS\n", (void *)regs->ip);
printk(" %s%pB\n", (reliable ? "" : "? "), (void *)*stack);
```

- Ref: How to get printk format specifiers right

`BUG_ON(c)`, `WARN_ON(c)`

- Similar to `assert(c)` in userspace
- `BUG_ON(c)`
  - if `c` is false, kernel panics with its call stack
- `WARN_ON(c)`
  - if `c` is false, kernel prints out its call stack and keeps running

Kernel panic message

```
[  174.507084] Stack:
[  174.507163]  ce0bd8ac 00000008 00000000 ce4a7e90 c039ce30 ce0bd8ac c0718b04 c07185a0
[  174.507380]  ce4a7ea0 c0398f22 ce0bd8ac c0718b04 ce4a7eb0 c037deee ce0bd8e0 ce0bd8ac
[  174.507597]  ce4a7ec0 c037dfe0 c07185a0 ce0bd8ac ce4a7ed4 c037d353 ce0bd8ac ce0bd8ac
[  174.507888] Call Trace:
[  174.508125]  [<c039ce30>] ? sd_remove+0x20/0x70
[  174.508235]  [<c0398f22>] ? scsi_bus_remove+0x32/0x40
[  174.508326]  [<c037deee>] ? __device_release_driver+0x3e/0x70
[  174.508421]  [<c037dfe0>] ? device_release_driver+0x20/0x40
[  174.508514]  [<c037d353>] ? bus_remove_device+0x73/0x90
[  174.508606]  [<c037bccf>] ? device_del+0xef/0x150
[  174.508693]  [<c0399207>] ? __scsi_remove_device+0x47/0x80
[  174.508786]  [<c0399262>] ? scsi_remove_device+0x22/0x40
[  174.508877]  [<c0399324>] ? __scsi_remove_target+0x94/0xd0
[  174.508969]  [<c03993c0>] ? __remove_child+0x0/0x20
[  174.509060]  [<c03993d7>] ? __remove_child+0x17/0x20
[  174.509148]  [<c037b868>] ? device_for_each_child+0x38/0x60
```

- **Q: How can I find where `sd_remove+0x20/0x70` is at source code?**

Analyze kernel panic message

1. Find where `sd_remove()` is (e.g., linux/driver/scsi/sd.c)
2. Load its object file with gdb
3. Use gdb command, `list *(function+0xoffset)` command

```
manjo@hungry:~/devel/kernel/build/build-generic/drivers/scsi$ gdb sd.o
This GDB was configured as "x86_64-linux-gnu".
(gdb) list *(sd_remove+0x20)
0x1650 is in sd_remove (/home/manjo/devel/linux/drivers/scsi/sd.c:2125).
2120    static int sd_remove(struct device *dev)
2121    {
2122            struct scsi_disk *sdkp;
2123
2124            async_synchronize_full();
```

- **Q: Can I debug kernel using gdb? → It is possible using QEMU/gdb**

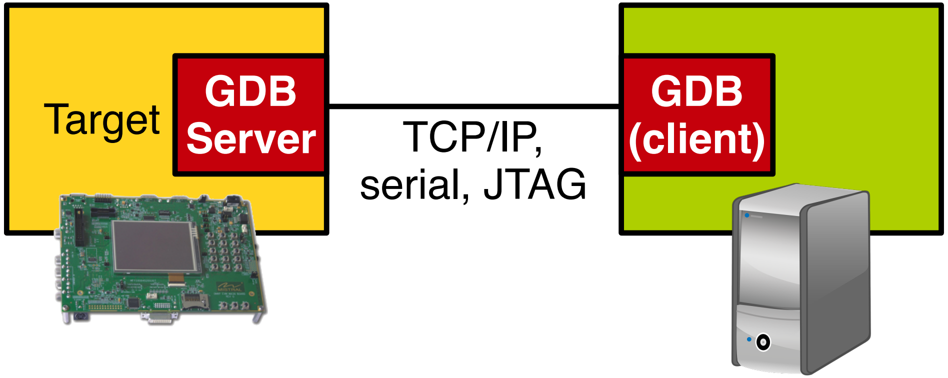
QEMU

- Full system emulator: emulates an entire virtual machine
  - Using a software model for the CPU, memory, devices
  - Emulation is slow
- Can also be used in conjunction with hardware virtualization extensions to provide high performance virtualization
  - **KVM**: In-kernel support for virtualization + extensions to QEMU
- To install QEMU, libvirt, GDB, etc.

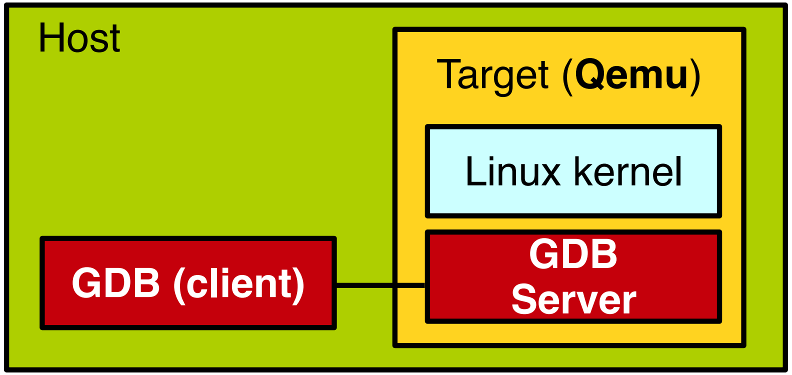
```
$ sudo apt install qemu qemu-system qemu-kvm libvirt-daemon-system \
                   libvirt-clients bridge-utils gdb
```

GDB server

- Originally used to debug a program executing on a remote machine
- For example when GDB is not available on that remote machine
    - E.g., low performance embedded systems



Kernel debugging with QEMU/gdb



- Linux kernel runs in a virtual machine (KVM or emulated on QEMU)
- Hardware devices are emulated with QEMU
- GDB server runs at QEMU, emulated virtual machine, so it can fully control Linux kernel running on QEMU
- *It is fantastic for debugging and code exploration!*

Three steps for QEMU/gdb

1. Build kernel for QEMU/gdb debugging
2. Create the root filesystem
3. Run Linux on QEMU and attach GDB

Build kernel for QEMU/gdb debugging

- Rebuild kernel with debug info, gdb script, and virtio enabled
- Following should be built-in, not built as a kernel module.

```
$ scripts/config -e CONFIG_DEBUG_INFO
$ scripts/config -e CONFIG_GDB_SCRIPTS
$ scripts/config -e CONFIG_E1000
$ scripts/config -e CONFIG_VIRTIO
$ scripts/config -e CONFIG_VIRTIO_NET
$ scripts/config -e CONFIG_VIRTIO_BLK

$ cat .config
CONFIG_DEBUG_INFO=y      # debug symbol
CONFIG_GDB_SCRIPTS=y     # qemu/gdb support
CONFIG_E1000=y           # default network card

CONFIG_VIRTIO=y          # virtio
CONFIG_VIRTIO_NET=y      # virtio network
CONFIG_VIRTIO_BLK=y      # virtio block device
```

Build kernel for QEMU/gdb debugging

```
# or (use `/ GDB_SCRIPTS` in `make menuconfig`
$ make menuconfig
  | Prompt: Provide GDB scripts for kernel debugging                  |
  |   Location:                                                       |
  |     -> Kernel hacking                                             |
  |       -> Compile-time checks and compiler options                 |
  |         -> Provide GDB scripts for kernel debugging               |
  |            Compile the kernel with frame pointers                 |

# then build the kernel
$ make -j8; make -j8 modules
```

- You don’t need `make modules_install; make install` because all necessary features will be built-in to ease of deployment.

Build kernel for QEMU/gdb debugging

- You can find the following files under the Linux source folder:

```
[path-to-linux]/arch/x86/boot/bzImage    # kernel binary image
[path-to-linux]/vmlinux                  # taget for GDB
[path-to-linux]/vmlinx-gdb.py            # GDB helper script
```

- Add *vmlinux-gdb.py* script to GDB’s auto load path so that you can later use the lx- helper commands.

```
$ echo "add-auto-load-safe-path [path-to-linux]/vmlinux-gdb.py" >> ~/.gdbinit
```

Create the root filesystem

- We need a root filesystem for the kernel to boot on QEMU.
- Two options
  - Create a new image using the buildroot project.
  - Convert a VirtualBox image (.vdi) to a QEMU (.img). See this.

Create the root filesystem with buildroot

```
# Clone the buildroot project

$ git clone git://git.buildroot.net/buildroot

# Configure buildroot to include the following options

$ cd buildroot
$ make menuconfig

    Target options -- Target architecture -- select [x86_64]
    Toolchain -- Enable C++ support [*]
    Filesystem images -- ext2/3/4 root filesystem -- choose [ext4]
    Target packages -- Network applications -- select openssh [*]

# Build

$ make -j<number of CPUs>

# Find the output image below

$ ls output/images/rootfs.ext4
output/images/rootfs.ext4
```

Run kernel with QEMU/gdb

```
# run QEMU/gdb

sudo qemu-system-x86_64 \
  -s \                                                # enable qemu-gdb debugging
  -S \                                                # pause on the first instr.
  -kernel <path-to-linux>/arch/x86_64/boot/bzImage \  # kernel image
  -nographic \
  -drive format=raw,file=<path-to-buildroot>/output/images/rootfs.ext4,if=virtio \
  -append "root=/dev/vda console=ttyS0 nokaslr other-paras-here-if-needed" \
  -m <mem 4G> \
  -cpu host \
  -smp <num of cpus> \
  -net nic,model=virtio \
  -net user,hostfwd=tcp::10022-:22 \                  # port forwarding for ssh
  -enable-kvm                                         # use kvm

# Ctrl-a x: terminating QEMU
```

Run kernel with QEMU/gdb

- QEMU options
  - `-kernel vmlinux` : path to the vmlinux of the kernel to debug
  - `-s` : enable the GDB server and open a port 1234
  - `-S` : (optional) pause on the first kernel instruction waiting for a

    GDB client connection to continue

Connect to the kernel on QEMU/gdb

- Run gdb and attach to QEMU

```
$ gdb <path-to-linux>/vmlinux
(gdb) target remote :1234
```

- You can use all *gdb commands* and *Linux-provided gdb helpers*!

```
- [b]reak <function name or filename:line# or *memory addres>
- [hbreak] <start_kernel or any function name> # to debug boot code
- [d]elete <breakpoint #>
- [c]continue
- [b]ack[t]race
- [i]nfo [b]reak
- [n]ext
- [s]tep
- [p]rint <variable or *memory address>
- Ctrl-x Ctrl-a: TUI mode
```

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_tcpudp.ko
loading @0xffffffffa0016000: /home/user/linux/build/net/netfilter/xt_pkttype.ko
loading @0xffffffffa0002000: /home/user/linux/build/net/netfilter/xt_limit.ko
loading @0xffffffffa00ca000: /home/user/linux/build/net/packet/af_packet.ko
loading @0xffffffffa003c000: /home/user/linux/build/fs/fuse/fuse.ko
...
loading @0xffffffffa0000000:
/home/user/linux/build/drivers/ata/ata_generic.ko
```

Linux-provided gdb helpers

- 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

loading @0xffffffffa006e000: /home/user/linux/build/lib/zlib_deflate/zlib_deflate.ko
loading @0xffffffffa01b1000: /home/user/linux/build/fs/btrfs/btrfs.ko

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

Linux-provided gdb helpers

- 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] usable
[     0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff] reserved
```

- Examine fields of the current task struct:

```
(gdb) p $lx_current().pid
$1 = 4998
```

Linux-provided gdb helpers

- Help

```
(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_struct variable
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
modules
```

Tips for QEMU-gdb kernel debugging

- `(gdb) p my_var` → `$1 = <value optimized out>`
  - `my_var` is optimized out
  - Since it is not possible to disable optimization for the entire kernel, we need to disable optimization for a specific file.

```
# linux/fs/ext4/Makefile
obj-$(CONFIG_EXT4_FS) += ext4.o

CFLAGS_bitmap.o = -O0  # disable optimization of bitmap.c

ext4-y := balloc.o bitmap.o dir.o file.o \
#...
```

Tips for QEMU-gdb kernel debugging

- Cursor disappears in qemu window?
  - Ctrl Alt (right)
- Always terminates QEMU with the `halt` command otherwise the disk image could be corrupted
- QEMU is too slow
  - Try KVM (`enable-kvm`)
  - It works only when your host is Linux.

Further readings

- Debugging by printing
- Kernel Debugging Tricks
- Kernel Debugging Tips
- Debugging kernel and modules via gdb
- gdb Cheatsheet
- Speed up your kernel development cycle with QEMU
- Installing new Debian systems with debootstrap
- Migrate a VirtualBox Disk Image (.vdi) to a QEMU Image (.img)
- The kernel’s command-line parameters

Next lecture

- Process management