Kernel debugging

Xiaoguang Wang

hw3 – system calls

How do you feel?

- start working on it early!

Q: using an ARM-based CPU, too slow to compile the kernel…

- Time to use CloudLab

Q: failed to ssh into the syzkaller QEMU VM

- provided new instructions to create an alpine-based VM image

Paper reading & final project topics

Everyone read and present a paper in systems (2~3 weeks)

Pick a final project topic

- Team of ~ 2
- If you choose to work on your research project, you need to differentiate them (and get consent from your advisor)
- You can reproduce a system in a paper you are interested in but should not use the open-sourced code (if they have).

https://docs.google.com/spreadsheets/d/1jtSceVvujo_8zCi0cOyJ26UvbNO8HvRj5F0w0uVvdTs/edit?usp=sharing

Recap

Memory allocation in the kernel

- `kmalloc(size, gfp_mask)` v.s. `vmalloc(size)`
- `gfp_mask`: `GFP_KERNEL` (caller may sleep) `GFP_ATOMIC` (no sleep)

More kernel data structures

- Radix tree
- XArray
- Bitmap

Kernel module

Recap: Linux radix tree

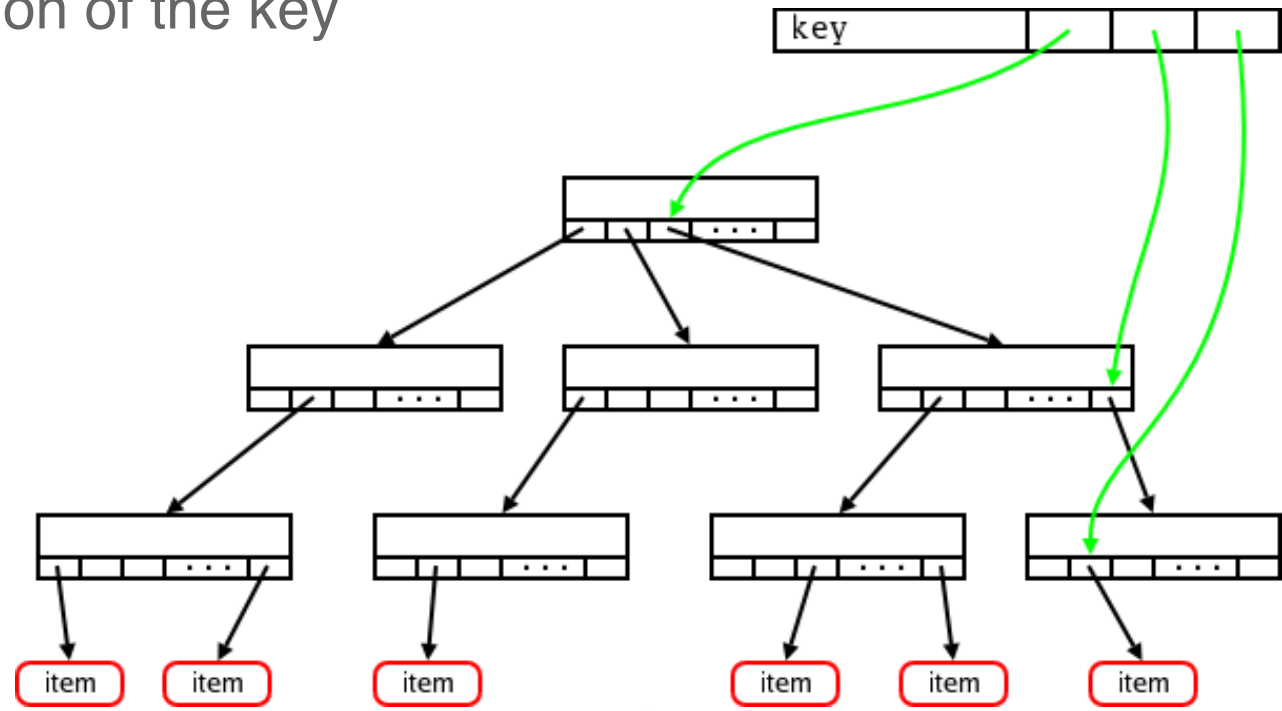
Compact prefix tree: map between a long integer key and a pointer value

Each node has 64 slots (leaves contain pointers)

Slots are indexed by a 6-bit (2^6=64) portion of the key

```
/* Declare and initialize a radix tree, gfp_mask, and
how memory allocations are to be performed. */
RADIX_TREE(name, gfp_mask);

/* Initialize a radix tree at runtime */
struct radix_tree_root my_tree;
INIT_RADIX_TREE(my_tree, gfp_mask);
```



Recap: XArray API

A nicer API ~~wrapper~~ replacement for radix tree (automatically protected with locks)

```
void *xa_load(struct xarray *xa, unsigned long index);
/* Up to three single-bit tags can be set on any non-null XArray  entry; they are managed with: */
void xa_set_tag(struct xarray *xa, unsigned long index, xa_tag_t tag);
void xa_clear_tag(struct xarray *xa, unsigned long index, xa_tag_t tag);
bool xa_get_tag(struct xarray *xa, unsigned long index, xa_tag_t tag);
/* Iterate over present entries in an XArray: */
xa_for_each(xa, index, entry) {
   /* Process "entry" */
}
```

Recap: Linux bitmap

```
/* include/linux/types.h */
#define DECLARE_BITMAP(name,bits) \
        unsigned long name[BITS_TO_LONGS(bits)]
void set_bit(long nr, volatile unsigned long *addr);
void clear_bit(long nr, volatile unsigned long *addr);
void change_bit(long nr, volatile unsigned long *addr);

void bitmap_zero(unsigned long *dst, unsigned int nbits);
void bitmap_fill(unsigned long *dst, unsigned int nbits);
```

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.

```
# 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

```
/* Symbols/function pointers with function name */
"%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: %pS\n", p->func);
printk("Faulted at %pS\n", (void *)regs->ip);
printk(" %s%pB\n", (reliable ? "" : "? "), (void *)*stack);
```

How to get printk format specifiers right

BUG_ON(c), WARN_ON(c)

Similar to `assert(c)` in the user-space

`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

Experiment: Load the kernel module (kern_dbg) and trigger the BUG_ON statement

Kernel panic message

```
[13330.023038] The answer is 42 ...
[13330.023040] The int_param is 41 ...
[13330.023061] ------------[ cut here ]------------
[13330.023062] kernel BUG at /users/xgwang/kern_dbg/lkp.c:14!
[13330.047941] invalid opcode: 0000 [#1] SMP PTI
[13330.067443] CPU: 8 PID: 104686 Comm: insmod Tainted: G          OE     5.15.0-86-
[13330.108661] Hardware name: HP ProLiant m510 Server Cartridge/ProLiant m510 Server
[13330.154877] RIP: 0010:lkp_init+0x41/0x1000 [lkp]
[13330.175638] Code: 00 00 00 48 c7 c7 69 70 71 c0 e8 c5 42 0f f5 8b 35 e0 1f 0f 00 4
[13330.260479] RSP: 0018:ffffb63700f8fcd8 EFLAGS: 00010297
[13330.284406] RAX: 0000000000000017 RBX: 0000000000000000 RCX: 0000000000000027
[13330.316710] RDX: 0000000000000000 RSI: 0000000000000001 RDI: ffff9c8c7fc20580
[13330.349003] RBP: ffffb63700f8fcd8 R08: 0000000000000003 R09: fffffffffffd2ef8
[13330.381072] R10: 7369206d61726170 R11: 5f746e6920656854 R12: ffffffffc0626000
[13330.413455] R13: ffff9c7d5e4e20f0 R14: 0000000000000000 R15: ffffffffc0718040
[13330.445836] FS:  00007ffaae924c40(0000) GS:ffff9c8c7fc00000(0000) knlGS:0000000000
[13330.482422] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
                                                                    3706e0
                                                                    000000
                                                                    000400
[13330.604307] Call Trace:
[13330.615246]  <TASK>
[13330.624592]  ? show_trace_log_lvl+0x1d6/0x2ea
[13330.644348]  ? show_trace_log_lvl+0x1d6/0x2ea
[13330.664011]  ? do_one_initcall+0x49/0x1e0
[13330.682061]  ? show_regs.part.0+0x23/0x29
[13330.700028]  ? __die_body.cold+0x8/0xd
[13330.717077]  ? __die+0x2b/0x37
[13330.730456]  ? die+0x30/0x60
```

How can we find where **RIP: lkp_init+0x41/0x1000** in the source?

Analyze kernel panic message

Where **RIP: lkp_init+0x41/0x1000** in the source?

Method 1: objdump

- `objdump -S lkp.o | less`

Q: which instruction causes this kernel panic?

Analyze kernel panic message

Where **RIP: lkp_init+0x41/0x1000** in the source?

Method 2: gdb

- `gdb lkp.o`
- `(gdb) list *(lkp_init+0x41)`

```
(gdb) list *(lkp_init+0x41)
0xc0 is in lkp_init (/users/xgwang/kern_dbg/lkp.c:14).
9       {
10              printk(KERN_INFO "Module loaded ...\n");
11              printk(KERN_INFO "The answer is %d ...\n",
12              printk(KERN_INFO "The int_param is %d ...\n
13
14              BUG_ON(answer != int_param);
15
```

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

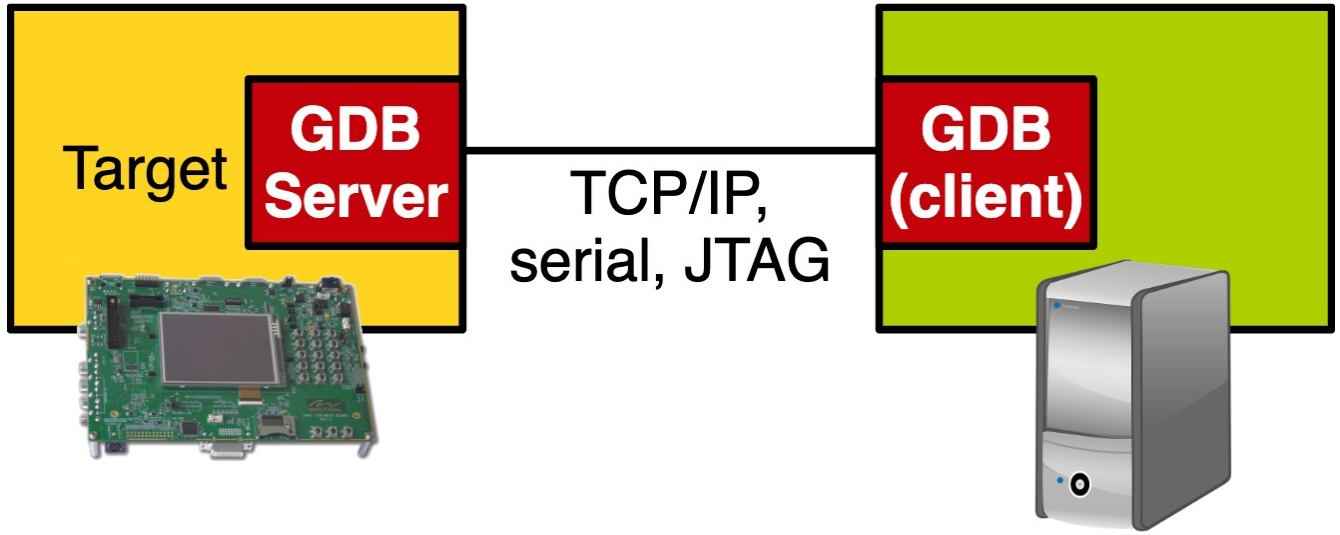
Can also be used in conjunction with hardware virtualization extensions to provide high performance virtualization

- **KVM**: In-kernel support for CPU/memory virtualization + extensions to QEMU

GDB server

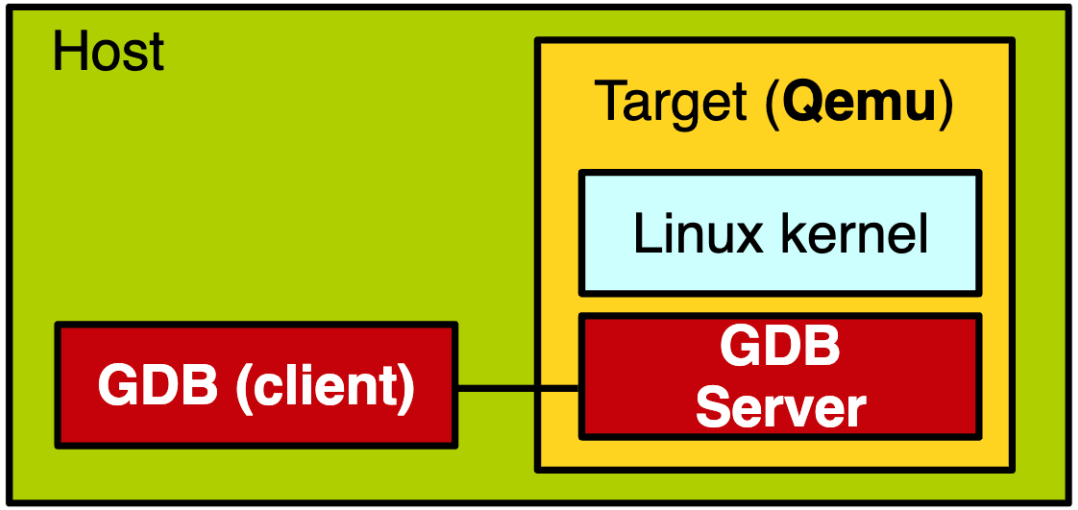
Originally used to debug a program executing on a remote machine

- When GDB is not available on that remote machine
- E.g., low performance embedded systems



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 in QEMU, contacting with the virtual machine emulation logic
- So, it can fully control Linux kernel running on QEMU



Build kernel for QEMU/gdb debugging

```
$ cat .config
# Compile-time checks and compiler options
CONFIG_DEBUG_INFO=y
CONFIG_GDB_SCRIPTS=y

# or (use `/ GDB_SCRIPTS` in `make menuconfig`
$ make menuconfig
 |-> Kernel hacking
 |   -> Compile-time checks and compiler options
 |     -> Debug information (Rely on the toolchain's implicit default DWARF version)
 |     -> Provide GDB scripts for kernel debugging
```

Run kernel with QEMU/gdb

```
#!/bin/bash
KNL_SRC=~/linux-6.7.1      # TODO: Change with your kernel base location
BZIMAGE=${KNL_SRC}/arch/x86/boot/bzImage
CMDLINE="nokaslr console=ttyS0 root=/dev/sda3"

sudo qemu-system-x86_64 \
    -s -nographic -smp 2 -m 2G \
    -nic user,host=10.0.2.10,hostfwd=tcp:127.0.0.1:2222-:22 \
    -net nic,model=e1000 \
    -drive file=alpine.qcow2,format=qcow2 \
    -kernel ${BZIMAGE} -append "${CMDLINE}"
```

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

CMDLINE

- pass the boot parameters to the kernel
- nokaslr: disable the kernel address space layout randomization

Kernel address space layout randomization

Connect to the kernel on QEMU/gdb

```
$ cd /path/to/linux-build
$ gdb 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
- [n]ext
- [s]tep
- [p]rint <variable or *memory address>
```

Linux-provided gdb helpers

Load module and main kernel symbols

```
$ gdb vmlinux
Type "apropos word" to search for commands related to "word"...
Reading symbols from vmlinux...
(gdb) target remote :1234
Remote debugging using :1234
native_irq_disable ()
    at ./arch/x86/include/asm/irqflags.h:37
37              asm volatile("cli": : :"memory");
(gdb) lx-symbols
loading vmlinux
(gdb)
```

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
Continuing.
```

Linux-provided gdb helpers

Dump the kernel log buffer of the target kernel:

```
(gdb) lx-dmesg
[    0.000000] Linux version 6.7.1 (xgwang@x86-server) (gcc (Ubuntu
11.4.0-1ubuntu1~22.04) 11.4.0, GNU ld (GNU Binutils for Ubuntu) 2.38) #1
SMP PREEMPT_DYNAMIC Wed Jan 24 21:44:23 MST 2024
[    0.000000] Command line: nokaslr console=ttyS0 root=/dev/sda3
```

Examine fields of the current task struct:

```
(gdb) p $lx_current().pid
$1 = 0
```

Linux-provided gdb helpers

Help

```
(gdb) apropos lx

function lx_clk_core_lookup -- Find struct clk_core by name

function lx_current -- Return current task.

function lx_dentry_name -- Return string of the full path of a dentry.

function lx_device_find_by_bus_name -- Find struct device by bus and name
(both strings)

function lx_device_find_by_class_name -- Find struct device by class and name
(both strings)

function lx_i_dentry -- Return dentry pointer for inode.
```

Tips for QEMU kernel debugging

Cursor disappears in QEMU window (if you don't add `-nographic`)?

- Ctrl Alt (right)

Always terminates QEMU VM with the `poweroff` command otherwise the disk image could be corrupted

QEMU is too slow

- Try KVM (`-enable-kvm`)
- It works only when your host is Linux (not work inside the VirtualBox VM).

Further reading

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

Feedback

UIC COMPUTER SCIENCE