Advanced Operating Systems (labs)

Vittorio Zaccaria | Politecnico di Milano | '24/25

This repo organization

A few words on the organisation of this repo. Where we get the kernel from, what is a initramfs (hint: initramfs is used as the first root filesystem that your machine has access to) and what you can produce with it. Our machine will not need another disk; this initial file system is everything we need to play with the kernel.

> Initramfs is setup using busybox: The Swiss Army Knife of Embedded Linux

Building the demo container

```
make build-container # takes a while, compiles the kernel
make build-sys       # compiles modules, rebuilds the initramfs file system
make enter-container # enter the container
```

You can change architecture with the T flag (from amd64 to aarch64)

```
T=aarch64 make build-container
T=aarch64 make build-sys
T=aarch64 make enter-container
```

If you use `E=full` an nvim based IDE with LSP will be also installed

```
E=full make build-container
```

In the container

Once in the container, you can run the compiled kernel and file system into its own virtual machine:

```
/repo/stage/start-qemu.sh --arch amd64
```

Whenever you change a module, you can rebuild everything with:

```
cd /repo/modules && make build-modules
```

Modules and module lifecycle

- Modules are a **flexible way of handling** the operating system **image** at runtime.
- Instead of rebooting with a different operating system image, modules allow easy extension of the operating systems' capabilities as required.
- It is the most convenient way with which we are going to play with the kernel.

See lab-1-intro-hello-module

Module parameters

- Parameters can be declared and initialised in code

```
static int num = 5;
/* S_IRUGO: everyone can read the sysfs entry */
module_param(num, int, S_IRUGO);
```

- And defined at module instantiation time

```
insmod yourmodule.ko num=10
```

Kernel crashes

Your module has complete access to kernel code and data; as such it might compromise the kernel state and "crash it" due to:

- Memory access error (NULL pointer, out of bounds access, etc)
- Voluntarily panicking on error detection (using panic())
- Kernel incorrect execution mode (sleeping in atomic context)
- Deadlocks detected by the kernel (Soft lockup/locking problem)

Kernel oops

- CPU state when the oops happened
- Registers content with potential interpretation
- Backtrace of function calls that led to the crash
- Stack content (last X bytes)

> Sometimes, the crash might be so bad that the **kernel will panic** and halt its execution entirely by stopping scheduling application and staying in a busy loop.

Poorman's printk

Printing from kernel to the log buffer is done today with the `pr_*()` family of functions:

- `pr_emerg(), pr_alert(), pr_crit(), pr_err(), pr_warn(), pr_notice(), pr_info(), pr_cont()`,

For example: `pr_info("Booting CPU %d\n", cpu);`

For pointers:

- `%p` : Display the hashed value of pointer by default.
- `%px` : Always display the address of a pointer (use carefully on non-sensitive addresses).
- `%pK` : Display hashed pointer value, zeros or the pointer address depending on kptr_restrict sysctl value

> - How to get printk format specifiers right — The Linux Kernel documentation
> - bootlin.com/doc/training/debugging/debugging-slides.pdf

Kernel debugging

- Usually no magic formula, requires **creative detective work**
- This lab is not about this creative part (this requires time) but the tools that are available to you
- In general many ways in which you can arrive to the same result

GDB

Why use live GDB

- Understanding the code flow
- Dumping data structures and assembly
- Debugging hangs

Why not use live gdb

- Issue is not reproducible
- Don't know what to look for

Kernel debug

You must use `CONFIG_DEBUG_INFO` in your `.config` file otherwise..

```
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
0xffffffff81e8ccbf in default_idle ()
(gdb) bt
#0 0xffffffff81e8ccbf in default_idle () <----- No line info!
#1 0xffffffff81e8cf8c in default_idle_call()
#2 0xffffffff810da469 in do_idle ()
```

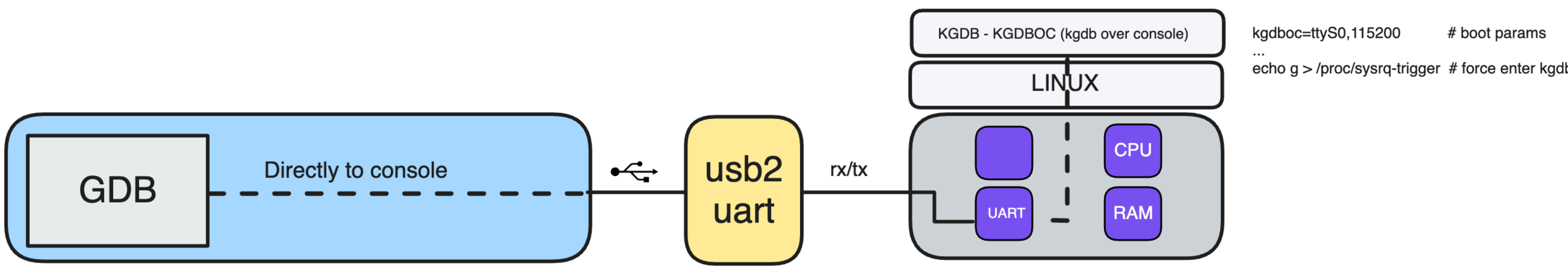
Kernel debug

You must boot with `kaslr` your kernel otherwise you'll get random and unintelligible stack frame info

```
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
0xffffffff81e8ccbf in ??
(gdb) bt
#0 0xffffffff81e8ccbf in ?? <----- No symbol info!
#1 0xffffffff81e8cf8c in ??
#2 0xffffffff810da469 in ??
```

GDB > KGDB

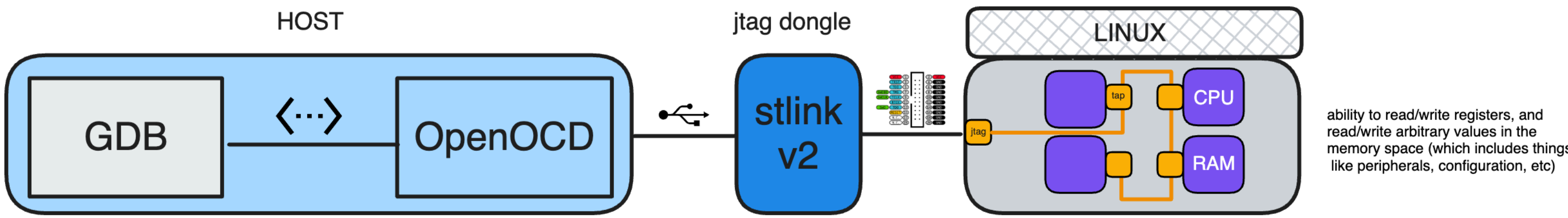
The execution of the kernel can be fully controlled by gdb from another machine, connected through a serial line (enabled by `CONFIG_KGDB_SERIAL_CONSOLE`).



> GDB connected to serial port to a live Linux machine

It can do almost everything, including inserting breakpoints in interrupt handlers and provides GDB python scripts to ease debugging.

GDB and JTAG

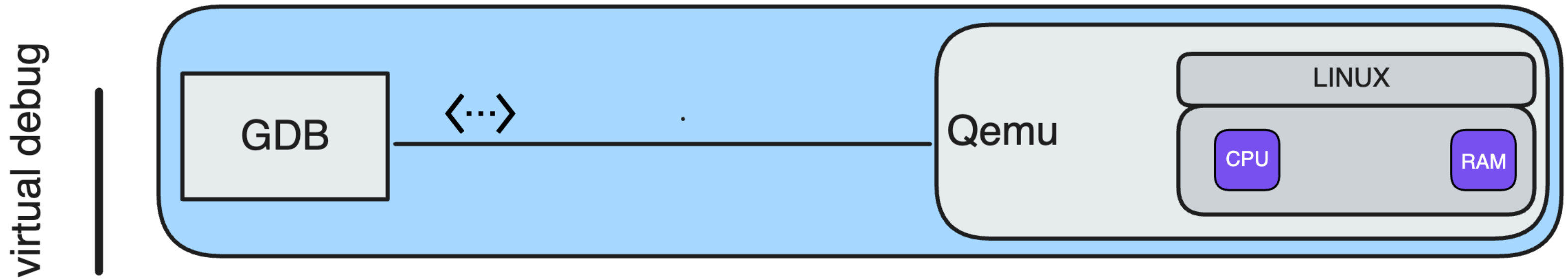


> GDB connected to a jtag port to a live Linux machine, no need kgdb

JTAG is allows to debug a system with almost no functioning software. You use JTAG on system bring up if you are not sure that you even reach a point where kgdb could work.

> - https://docs.kernel.org/dev-tools/gdb-kernel-debugging.html
> - bootlin.com/doc/training/debugging/debugging-slides.pdf
> - https://elinux.org/images/1/1b/ELC19_Serial_kdb_kgdb.pdf

GDB > QEMU



> Gdb connected to QEMU; no intermediator necessary

With QEMU, we could emulate access to our linux system as it was over a jtag, even if kgdb is not enabled. GDB commands are the same though.

Inspect Linux with gdb

TMUX shell 1

```
cd /sources/linux
make scripts_gdb
echo "add-auto-load-safe-path /sources/linux/vmlinux-gdb.py" > ~/.gdbinit
gdb vmlinux
gdb> target remote localhost:1234
```

TMUX shell 2

```
```

```
cd /repo/modules && make build-modules
/repo/stage/start-qemu.sh --arch amd64 --debug
```

Essentially runs the following command:

```
qemu-system-amd64 ... -s -S -kernel ... -append "... nokaslr"
```

GDB commands > most used

```
break <sym> # inserts a break point
continue    # continues from a breakpoint to the next
step        # enters the subroutine and breaks
next        # does not enter subroutines and breaks
```

GDB commands > Linux specific scripts

```
lx-lsmod # list modules
lx-ps    # list processes
apropos lx

p *(struct task_struct *) <address of process from lx-ps>
p $container_of(listheadp, "struct <type x>", "name of the list head in <type x")
p $lx_task_by_pid(<PID>)
set $t = $lx_task_by_pid(<PID>)
p $t -> __state # print state
```

Debugging a module

Debugging a timer handler (/repo/modules/lab-1-list-manip)

TMUX 1

```
cd /sources/linux
make scripts_gdb
echo "add-auto-load-safe-path /sources/linux/vmlinux-gdb.py" > ~/.gdbinit
gdb vmlinux
gdb> target remote localhost:1234
# or `gdbfrontend -G vmlinux -l 0.0.0.0 -p 6000`
```

TMUX 2

```
cd /repo/modules && make build-modules
/repo/stage/start-qemu.sh --arch amd64 --dbg
```

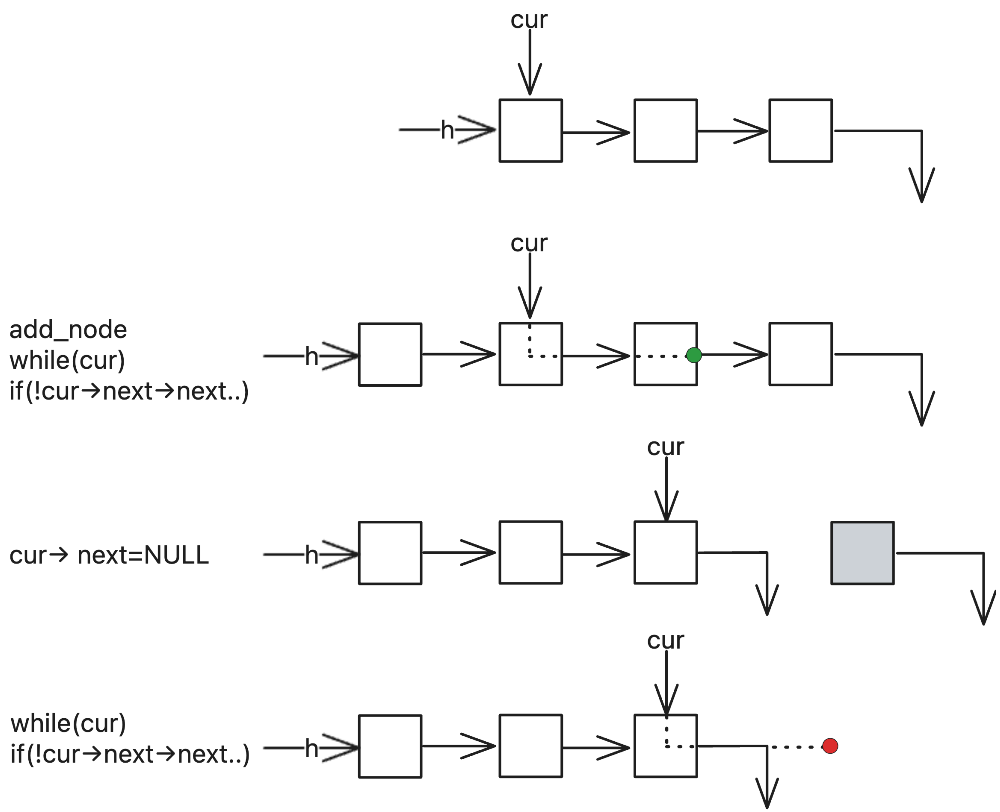
Debugging your module (cont'd)

GDB

```
target remote :1234
c
hbreak do_init_module
c
# execute boot and insmod (see below), when it stops at do_init_module
# note it is a good point to show the stack trace of a syscall because init_module is a system call!
# use lx-symbols to load its symbols
lx-symbols /repo/modules/lab-1-list-manip
# now you should see the symbols of your module as well
hbreak my_timer_handler
c  # and then goes into panic

# to list the program at a certain line indicated by the panic
list *(my_timer_handler+0x2d)
```

Debugging your module (cont'd)



> The culprit, the second time we check, we get a null pointer

Graphical debug with gdbfrontend

Container must have been launched with:

```
-p 8080:6000
```

Launch in the container

```
# Shell 1
cd /sources/linux
make scripts_gdb
echo "add-auto-load-safe-path /sources/linux/vmlinux-gdb.py" > ~/.gdbinit
gdbfrontend -G vmlinux -l 0.0.0.0 -p 6000

# Shell 2
cd /repo/modules && make build-modules
/repo/stage/start-qemu.sh --arch amd64 --dbg
```

and access the graphical interface of gdb over: http://localhost:8080/

Anatomy of a syscall

Insmod is just an example of syscall

```
(gdb)
#0  do_init_module (mod=mod@entry=0xffffffffc0002040) at ./include/linux/slab.h:590
#1  0xffffffff811038d2 in load_module (info=info@entry=0xffffc90000277e78, uargs=uargs@entry=0x6661e1 "", flags=flags@entry=0) at kerne
l/module.c:4133
#2  0xffffffff81103d59 in __do_sys_init_module (umod=<optimized out>, len=<optimized out>, uargs=0x6661e1 "") at kernel/module.c:4198
#3  0xffffffff81bd1b1b in do_syscall_x64 (nr=<optimized out>, regs=0xffffc90000277f58) at arch/x86/entry/common.c:50
#4  do_syscall_64 (regs=0xffffc90000277f58, nr=<optimized out>) at arch/x86/entry/common.c:80
#5  0xffffffff81c0007c in entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:113
#6  0x0000000000000000 in ?? ()
(gdb)
```

Printing the syscall table

```
x/255x (unsigned long*) sys_call_table
```

Linkography

- https://bmeneg.com/post/kernel-debugging-with-qemu/
- https://docs.kernel.org/fault-injection/provoke-crashes.html
- https://blog.k3170makan.com/2020/11/linux-kernel-exploitation-0x0-debugging.html