Developing and testing a device driver without the device

```
git clone --recurse-submodules git@github.com:mdionisio/linuxlab2018.git
```

Michele Dionisio | Powersoft SPA | michele.dionisio@powersoft.com

Pietro Lorefice | Develer S.r.l. | pietro@develer.com

BEFORE WE START

```
git clone --recurse-submodules  git@github.com:mdionisio/linuxlab2018.git
```

```
apt-get install build-essential
apt-get install crossbuild-essential-armhf
apt-get install gcc-arm-linux-gnueabihf
apt-get install gcc-arm-none-eabi
apt-get install libncurses5-dev
apt-get install bison flex libssl-dev bc
apt-get install python pkg-config libglib2.0-dev libpixman-1-dev
apt-get install sudo cpio lzop gzip
apt-get install gdb-multiarch
```

Who we are

What we will see

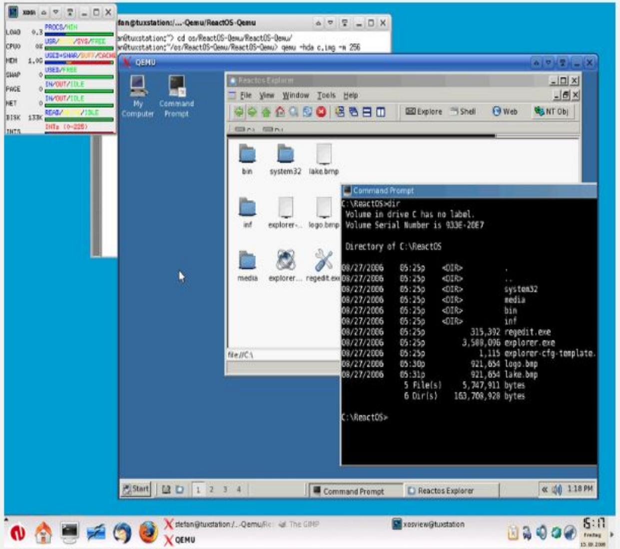
1. Qemu
    a. Introduction
    b. Developing a new simple emulated device
    c. Adding it to an existing machine
2. Test
    a. Simple bare-metal cortex a9 project
    b. Test and debug with qemu
3. Test with Linux
    a. Create a minimal rootfs
4. Develop a test a linux driver
    a. Developing device driver
    b. Adding it to and existing board configuring the device tree
    c. Test

QEMU

the FAST! processor emulator

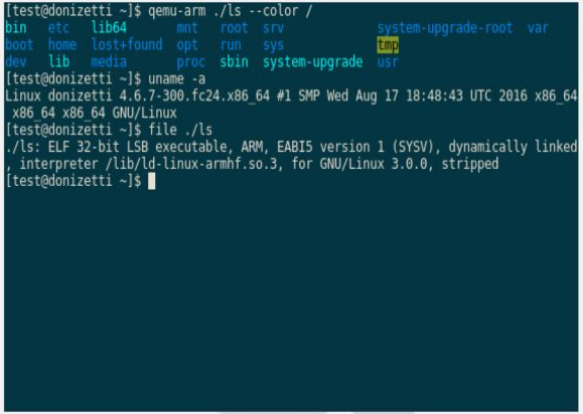
What is QEMU?

QEMU is a generic and open source machine emulator and virtualizer.



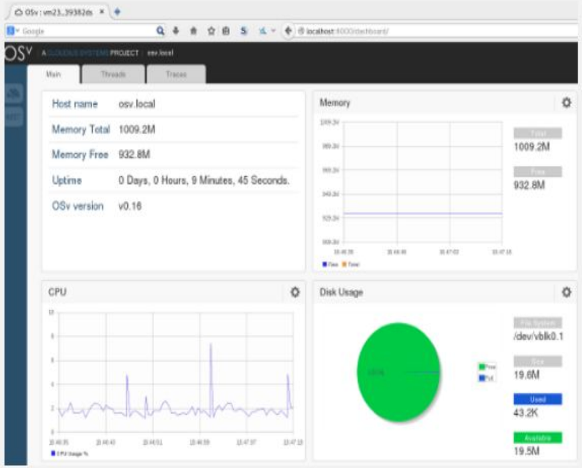
Full-system emulation

Run operating systems for any machine, on any supported architecture



User-mode emulation

Run programs for another Linux/BSD target, on any supported architecture



Virtualization

Run KVM and Xen virtual machines with near native performance

Device Specification

1. One memory mapped RW registry. You can read on it the last value write plus 1.
2. If you write 1 and interrupt is raised and writing 0 the interrupt is acknowledged

Device is installed in a custom board coming from Sambrelite board

This board is based on i.mx6 that is a ARM Cortex A9 MP processor so we will use a gcc for that platform for developing and testing

LINUXLAB BUILDING SMARTER DEVICES

And now …. A brief introduction to QOM.

And jump to the C code

We have the device emulated and we can start to add it to a machine emulated.

So go back to the QEMU source code

A bare metal CORTEX-A9

startup.S

And now LINUX

To develop and test a linux device driver we need a working Linux system.
So before start to develop we have to create a minimal Linux system.

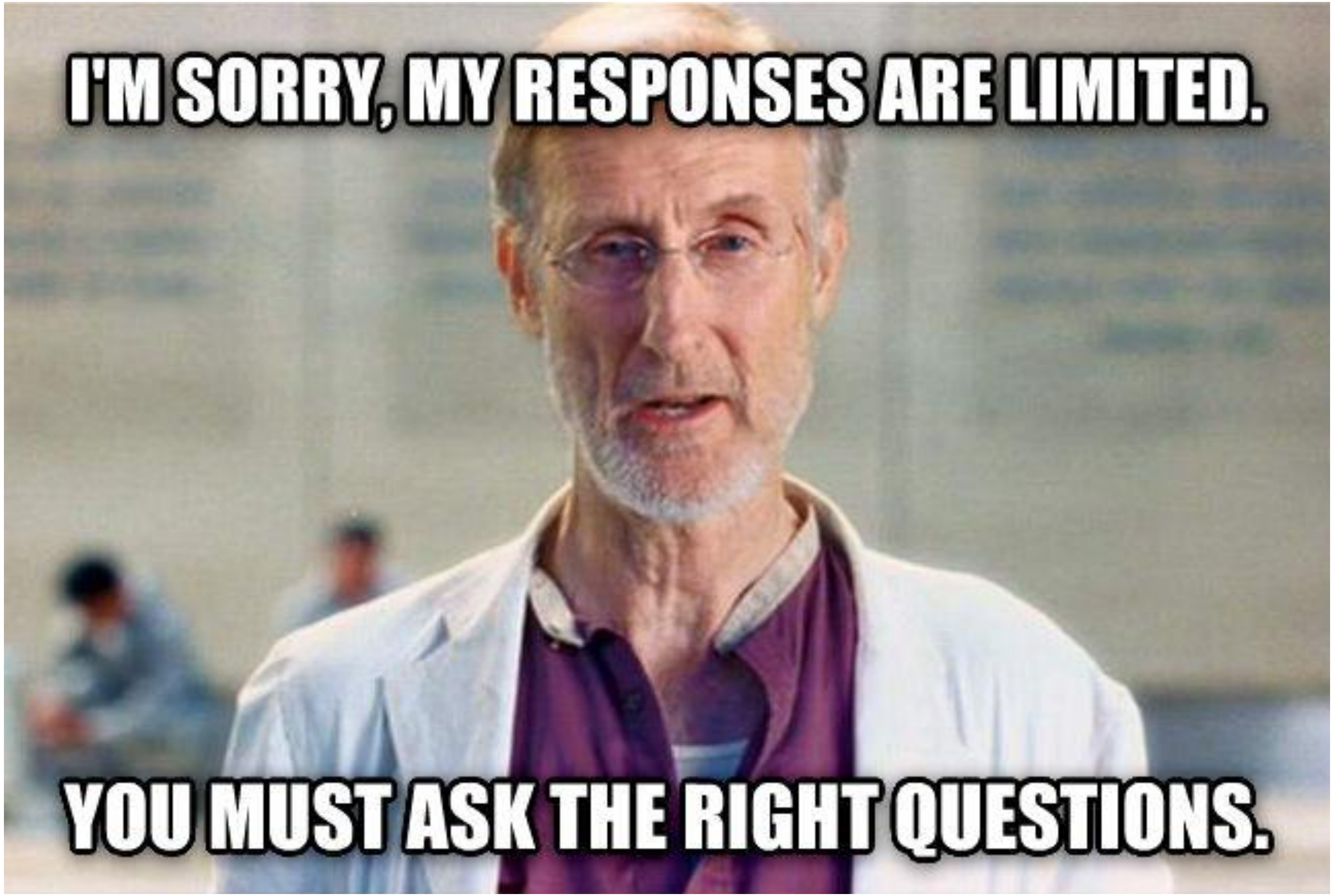
1. Build linux kernel
2. Build busybox
3. Build a minimal rootfs that can be mounted in ram. INITRAMFS

The result is not a full linux distribution but is something perfect for develop and test.

Test the created Linux system

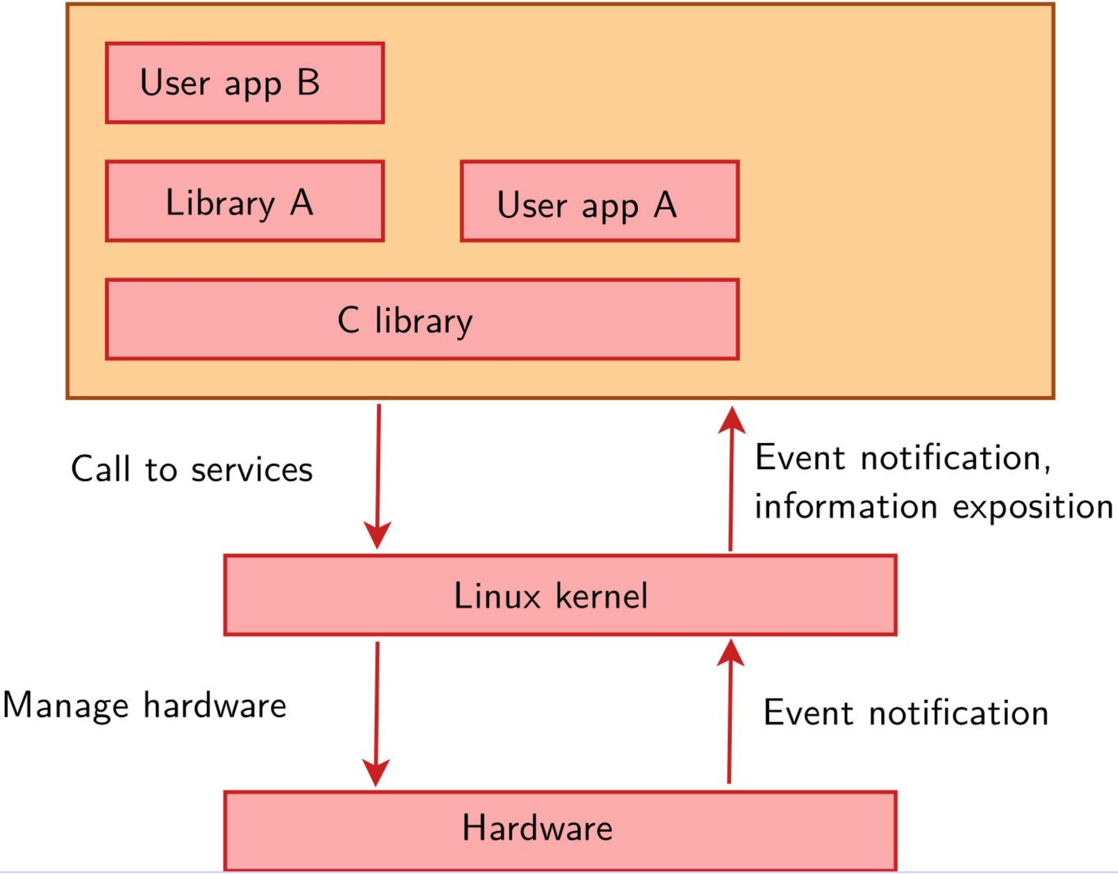
Questions ?

Image coming from film Io Robot



Organization of a Linux system

© CC BY-SA 3.0 Bootlin, 2004-2018



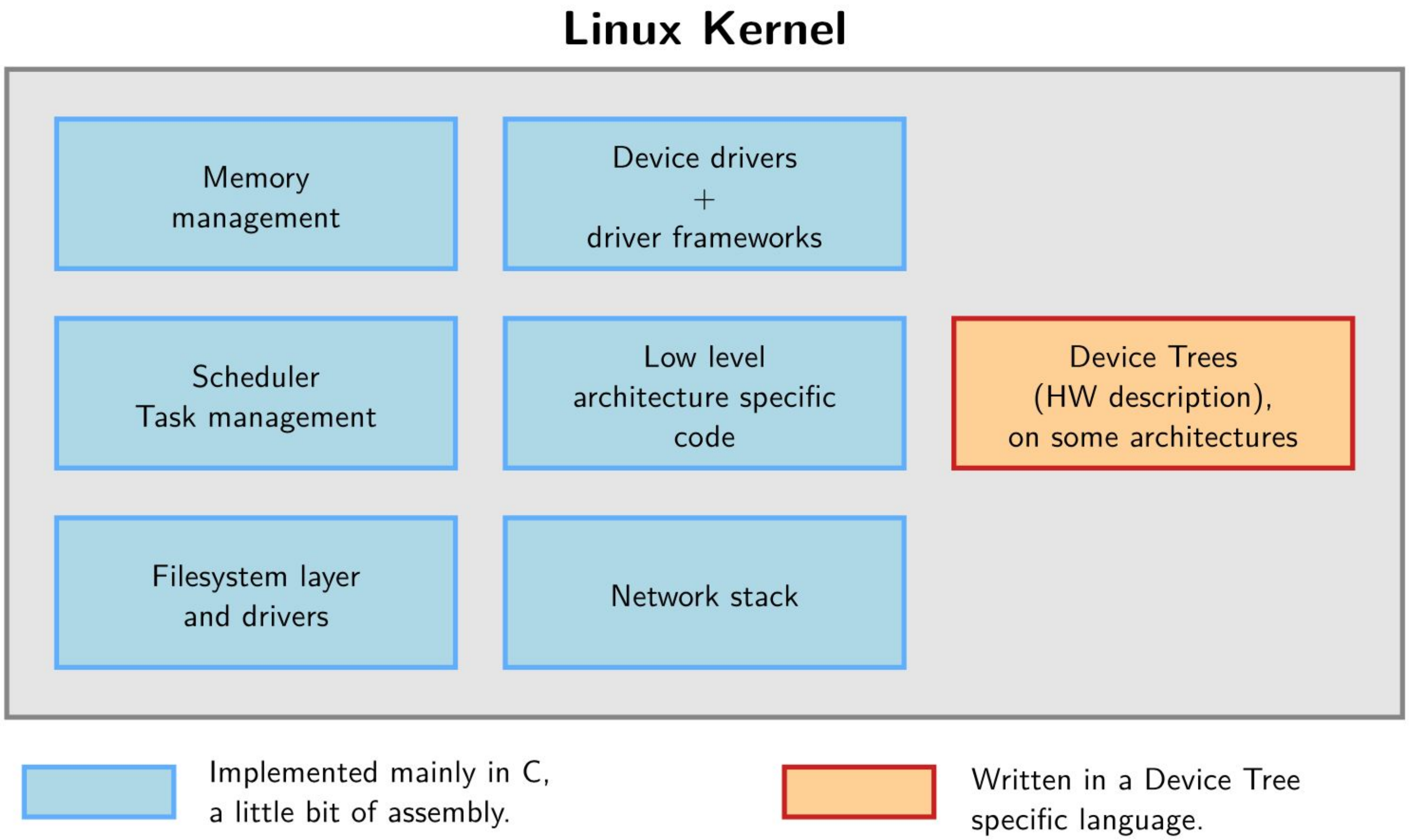
- Interact with the underlying hardware
- Provide a set of portable, architecture-independent API
- Handle concurrent access

- Kernel and userspace communicate using **system calls**, or *syscalls*
  - A modern Linux kernel implements around 300 syscalls
- Syscalls are propagated by the kernel to the dedicated **driver**
- Drivers handle the request and return a **status code** to the calling process

Interacting with the kernel: pseudo-filesystems

- Kernel can also present statistics and configurations using **pseudo-filesystems**
- Usually mounted automatically at boot
- Several pseudo-filesystems exist with different roles
  - `proc`, mounted on `/proc`
  - `sysfs`, mounted on `/sys`

© CC BY-SA 3.0 Bootlin, 2004-2018



Working on the Linux kernel

- By default, Linux is configured and built for the host system
- Use **`ARCH`** and **`CROSS_COMPILE`** variables to change target platform
- Different configuration options are available for different platforms

- To use an existing configuration: **`make <config-name>_defconfig`**
  - Configurations available in *`arch/<arch>/configs/`*

- To modify the current configuration: **`make menuconfig`**

- To build the kernel: **`make -jN`**
- Build outputs
  - `./vmlinux`
  - `./arch/<arch>/boot/*Image`
  - `./arch/<arch>/boot/dts/*.dtb`
  - `./**/*.ko`

Writing a kernel driver

- (*Optional*) Create a dedicated directory under `drivers/`
- Create the necessary **Kconfig** options and **Makefile** rules
- Write code!

- A kernel module is not a userspace application!
  - Does not execute sequentially
  - Does no automatic cleanup
  - Can be interrupted
  - Does not have floating-point support

Device drivers

- Access to the hardware is abstracted as a stream of bytes
  - Serial ports and text consoles are example of char devices
- Many hardware peripherals can be abstracted as char devices
  - Flash memories
  - Frame grabbers
  - ADC, DAC, shift registers

- Char devices are accessed using *device nodes*
  - Usually located in the `/dev` directory
- A combination of *major* and *minor* is used to identify the device
  - The major number identifies the device driver
  - Minors usually identify a device instance of a given driver

Platform drivers

- A device located on a bus with minimal infrastructure
  - Most peripheral controllers on ARM SoCs are platform devices
- Usually addressed directly by the CPU
  - Eg. using memory-mapped registers
- In Linux, they are handled using **platform drivers**

- Implemented using `struct platform_driver`
  - Registered using `platform_driver_register()`
- Enumeration is performed by the kernel
  - During enumeration, `.probe()` is invoked
  - During shutdown, `.remove()` is invoked
- The driver should actually check that the hardware *exists*

Device Tree

- Embedded architectures often have non-discoverable peripherals
    - This is the case for ARM processors, for example
- We need a way to describe the hardware connected to such platforms
- Enter the Device Tree!

- Organized in `.dts` and `.dtsi` files
- Compiled in `.dtb` files
- Several roles in the Linux kernel
  - Describing devices attached to non-discoverable buses
  - Configure pin muxing on SoCs
  - Configure some system parameters (memory layout, flash partitioning etc.)

```
#include "imx6q.dtsi"
#include "imx6qdl-sabrelite.dtsi"

/ {
    model = "Freescale i.MX6 Quad SABRE Lite Board";
    compatible = "fsl,imx6q-sabrelite", "fsl,imx6q";

    soc {
        qemudev: qemu-test@50000000 {
            compatible = "linuxlab,qemu-test";
            reg = <0x50000000 4>;
            status = "okay";
        };
    };
};
```

I/O operations

- I/O operations performed on device nodes are propagated to the driver
  - `open()`, `read()`, `write()`, `close()`, ...
- Drivers need to define a callback for each handled syscall
- This is done using the `struct file_operations`
  - Structure members are all function pointers

- **open()** – called each time the device is opened from userspace
- **read()** – called when data is sent from the device to userspace
- **write()** – called when data is sent from userspace to the device
- **release()** – called when the device is closed in userspace

- Special care must be used to access user memory pointers in I/O callbacks
  - User memory is different from kernel memory, and can’t be accessed directly

- **`copy_to_user()`** – Copy data from kernel memory to user memory

- **`copy_from_user()`** – Copy data from user memory to kernel memory

Interrupt handling

- Most peripherals or devices use interrupts to signal asynchronous events
  - ADC conversion performed, SPI transfer completed, etc.
- In Linux, drivers can request to serve one or more specific interrupts
  - Usually each peripheral controller has its own interrupt line
- A device tree entry can be used to specify the interrupt index
  - Interrupt numbering is platform- and controller-specific