Operating System Design and Implementation

Lecture 5: Linux Kernel

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm
Classroom: ED-302

Acknowledgements and Disclaimer

- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Bootloader Review

- The bootloader is a piece of code responsible for
  - Basic hardware initialization
  - Loading application binary, usually an operating system kernel, from flash or network
  - Possibly decompression of the application binary
  - Execution of the application
- Additional functions
  - Provide a shell with various commands
    - Memory inspection, hardware diagnostics and testing etc.

1st stage bootloader

- The main goal of the first stage bootloader
  - Configure the RAM controller
  - Load the second stage bootloader from storage (flash) to RAM
- The main porting steps are:
  - Finding the proper RAM timings and settings from the first stage
  - Configuring the storage IP
  - Copying the second stage to RAM

2^{nd} stage bootloader

- The main goal of the 2^{nd} stage bootloader
  - Load the Linux kernel from storage to RAM
  - Set the ATAGS or load the device tree depending on the kernel version
  - Load an initramfs to be used as the root filesystem
  - Also provices more debugging utilities like reading and writing to memory or Ethernet access

Outline

- U-Boot
- Linux kernel
  - Linux kernel structure
  - Linux kernel module
- Kernel debugging
  - kgdb

Booting kernel

- Device tree
  - Many embedded architectures have many non-discoverable hardware (serial, Ethernet, I2C, NAND flash, USC controllers …)
  - Such hardware is either described in BIOS ACPI table (x86) or
  - Using C code directly in the kernel or
  - Using a special hardware description language in a Device Tree
- The goal of device tree
  - To describe the hardware and its integration

Device tree

- A device tree source (DTS)
  - Compiled into a binary device tree blob (DTB)
  - Needs to be passed to the kernel at boot time
  - Each board/platform has its own device tree "arch/arm/boot/dts/<board>.dtb"
  - The boot loader must load both the kernel image and DTB in memory before starting the kernel

U-Boot configuring and Installing

- U-Boot is a bootloader
- **The “config/” directory in U-Boot source codes**
  - Contains configuration files for each supported board
  - Examples: configs/stm32mp15_basic_defconfig
  - It defines the CPU type, the peripherals and their configuration
- **Configuring and compiling U-Boot**
  - Configuration stored in a .config file
  - make BOARDNAME_defconfig
  - make menuconfig to further customize U-Boot’s configuration
  - cross-compiler: make CROSS_COMPILE=arm-linux-
  - The final result is a u-boot.bin file, which is the U-Boot image

Booting with U-boot

- **U-Boot**
  - load and boot a kernel image and change the kernel image and the root filesystem stored in flash
  - Through the network if U-Boot has drivers for such networking
  - Through a USB key, a SD, the serial port (loadb, loadx or loady command)
  - U-Boot can directly boot the zImage binary
    Example: tftp <address> <filename> => tftp 0x21000000 zImage
- **The typical boot process is:**
  - Load zImage at address X in memory
  - Load <board>.dtb at address Y in memory
  - Start the kernel with bootz X – Y
    The – in the middle indicates no initramfs

U-Boot prompt

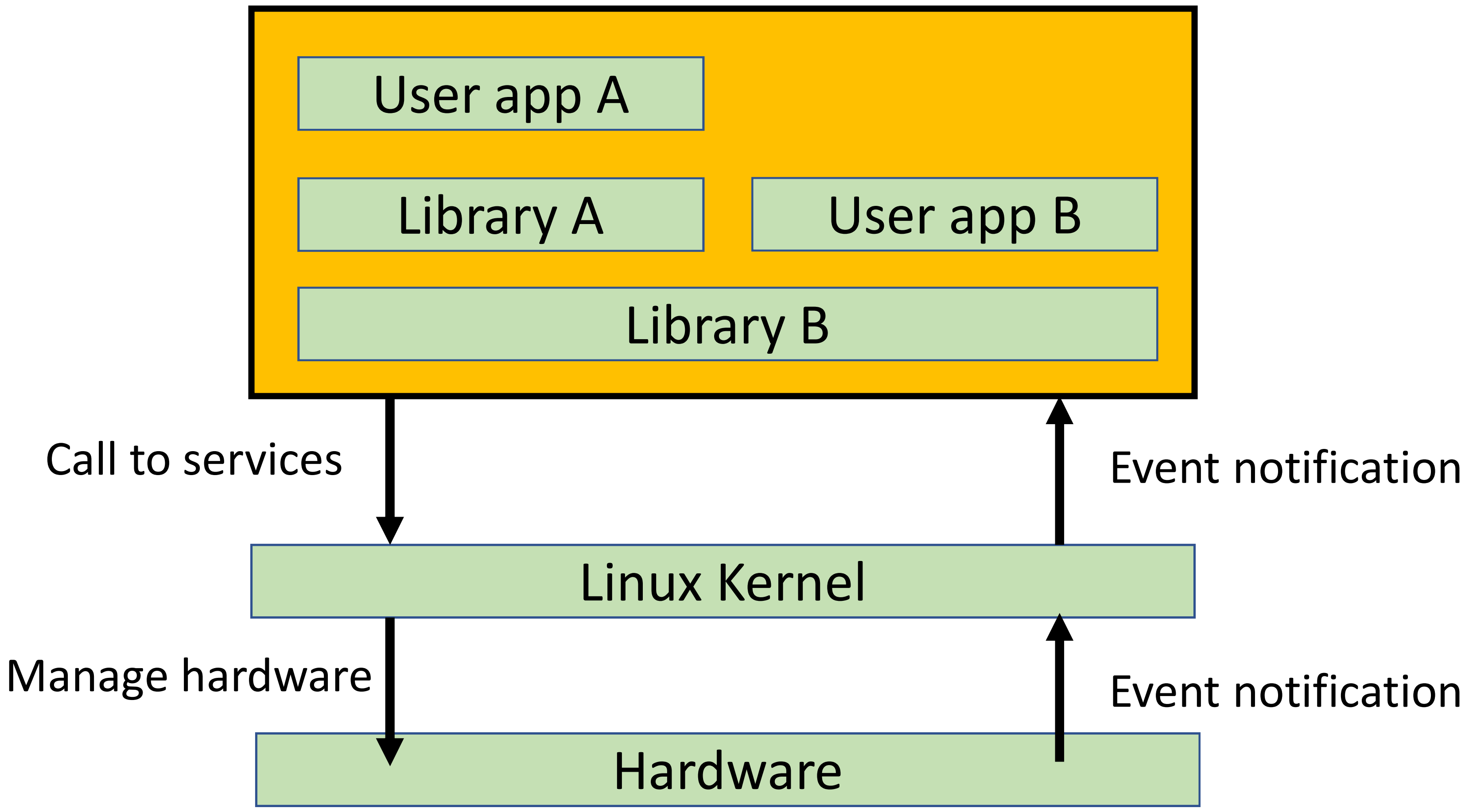
- U-Boot is usually be installed in flash memory
- Connect the target to the host through a serial console
- Power-up the board.
  On the serial console:

```
U-Boot 2020.04 (May 26 2020 - 16:05:43 +0200)

CPU: SAMA5D36
Crystal frequency:       12 MHz
CPU clock        :      528 MHz
Master clock     :      132 MHz
DRAM:  256 MiB
NAND:  256 MiB
MMC:   Atmel mci: 0, Atmel mci: 1
Loading Environment from NAND... OK
In:    serial@ffffee00
Out:   serial@ffffee00
Err:   serial@ffffee00
Net:   eth0: ethernet@f0028000
Error: ethernet@f802c000 address not set.

Hit any key to stop autoboot:  0
=>
```

Linux kernel in the system



Linux kernel main roles

- **Manage all the hardware resources**
  - CPU, memory, I/O
- **Contains a set of hardware independent APIs**
  - Allow user applications to use the hardware resources
- **Handle concurrent accesses**
  - The use of hardware resources from different applications
  - E.g. a single network interface used by multiple user space applications through network connections. The kernel is responsible for multiplexing the hardware resource

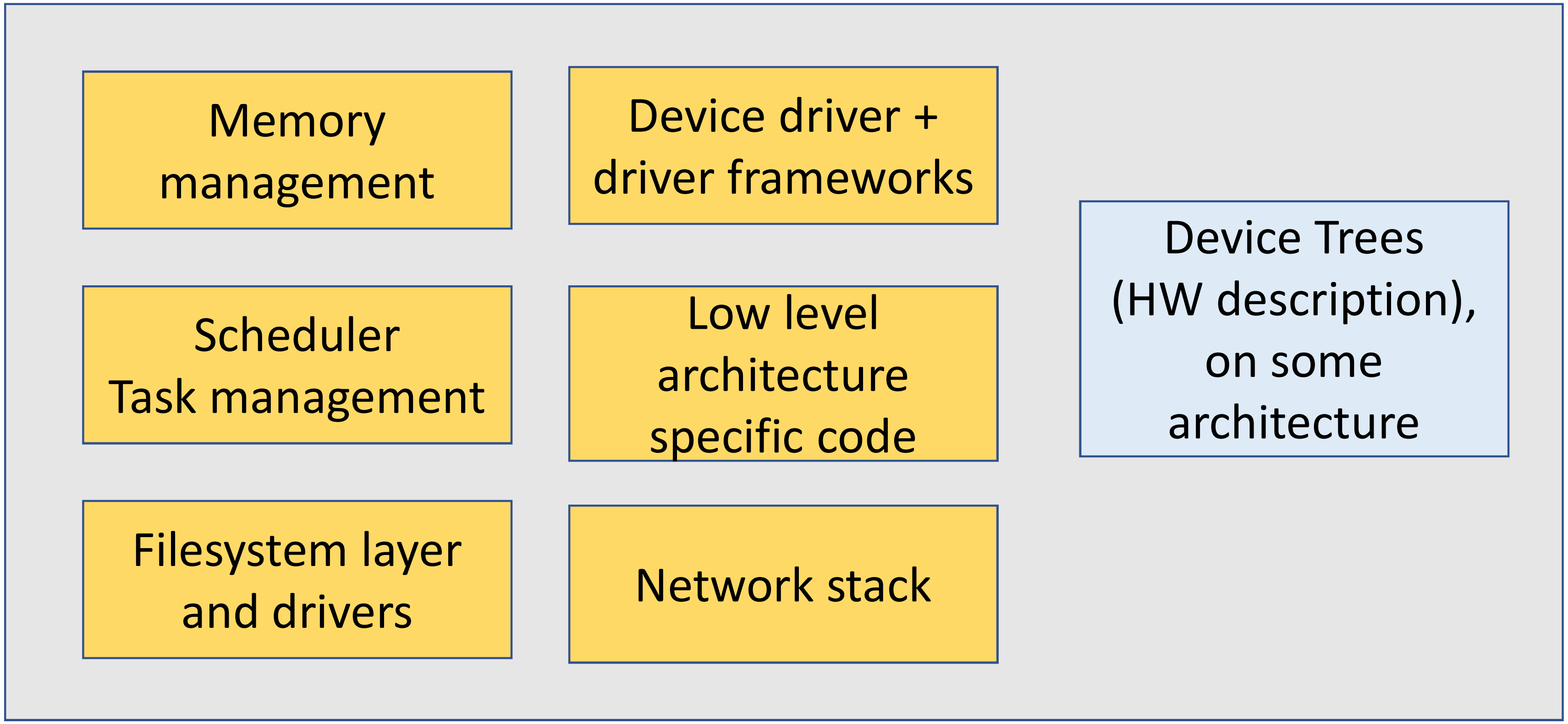
System calls

- **System calls**
  - The main interface between the kernel and user space
- About 400 system calls that provide the main kernel services
  - File and device operations, network operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, etc.
- These system call interfaces are wrapped by the **C library**
  - User space applications usually never make a system call directly but rather use the corresponding C library function

Pseudo filesystems

- **Pseudo filesystem**
  - Linux makes system and kernel information available in user space through **pseudo filesystems**, also call **virtual filesystems**
  - Allow applications to see directories and files that do not exist on any real storage: they are created and updated on the fly by the kernel
  - The two most important pseudo filesystems are
    - **proc**, usually mounted on **/proc**: operating system related information (processes, memory management parameters …)
    - **sysfs**, usually mounted on **/sys**: representation of the system as a tree of devices connected by buses.

Inside the Linux kernel



Supported hardware architectures

- See the **arch/** directory in the kernel sources
  - Minimum: 32 bit processors, with or without MMU, supported by gcc or clang
  - 32 bit architecture (arch/ subdirectories)
    Examples: arm, arc, m68k, microblaze (soft core on FPGA)
  - 64 bit architectures:
    Example: alpha, arm64, ia64 …
  - 32/64 bit architectures
    Example: mips, powerpc, riscv, sh, sparc, x86 …
- Find details in kernel sources:
  - arch/<arch>/Kconfig, arch/<arch>/README, or Documentation/<arch>/

Getting Linux sources

- Fetch the entire kernel sources and history
  - git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux
- Create a branch that starts at a specific stable version
  - git checkout -b <name-of-branch> v5.6
- Linux 5.10.11 sources
  - 70,639 files (git ls-files | wc -l)
  - 29,746,102 lines (git ls-files | xargs cat |wc -l)
  - 862,810,769 bytes (git ls-files | xargs cat | wc -c)
  - A minimum uncompressed Linux kernel just sizes 1-2 MB
  - Why are these sources so big ?

Linux kernel size

- As of kernel version 5.7 (in percentage of total number of lines)
  - drivers/: 60.1%
  - arch/: 12.9%
  - fs/: 4.7%
  - sound/: 4.2%
  - net/: 4.0%
  - include: 3.6%
  - tools/: 3.2%
  - Documentation/: 3.2%
  - Kernel/: 1.3%

lib/:	0.6%
mm/:	0.5%
scripts/:	0.4%
crypto/:	0.4%
security/:	0.3%
block/:	0.2%
samples/:	0.1%
virt/:	0.1%

Linux sources structure (1/5)

- **arch/<ARCH>**
  - Architecture specific code
  - arch/<ARCH>/mach-<machine>, SoC family specific code
  - arch/<ARCH>/include/asm, architecture-specific headers
  - arch/<ARCH>/boot/dts, Device Tree source files, for some arch.
- **block/**
  - Block layer core
- **certs/**
  - Management of certificates for key signing

Linux sources structure (2/5)

- **crypto/**
  - Cryptographic libraries
- **documentation/**
  - Kernel documentation sources
- **drivers/**
  - All device drivers except sound ones (usb, pci)
- **fs/**
  - Filesystems (fs/ext4, etc.)
- **include/linux**
  - Linux kernel core headers

Linux sources structure (3/5)

- **include/uapi**
  - User space API headers
- **init/**
  - Linux initialization (including init/main.c)
- **ipc/**
  - Code used for inter process communication
- **Kbuild**
  - Part of the kernel build system
- **Kconfig**
  - Top level description file for configuration parameters
- **kernel/**
  - Linux kernel core (very small !)
- **lib/**
  - Misc library routines (zlib, crc32 …)

Linux sources structure (4/5)

- **mm/**
  - Memory management code (small too!)
- **net/**
  - Network support code (not drivers)
- **samples/**
  - Sample code (markers, kprobes, kobjects, bpf …)
- **scripts/**
  - Executables for kernel building and debugging
- **security/**
  - Security model implementations (SELinux)

Linux sources structure (5/5)

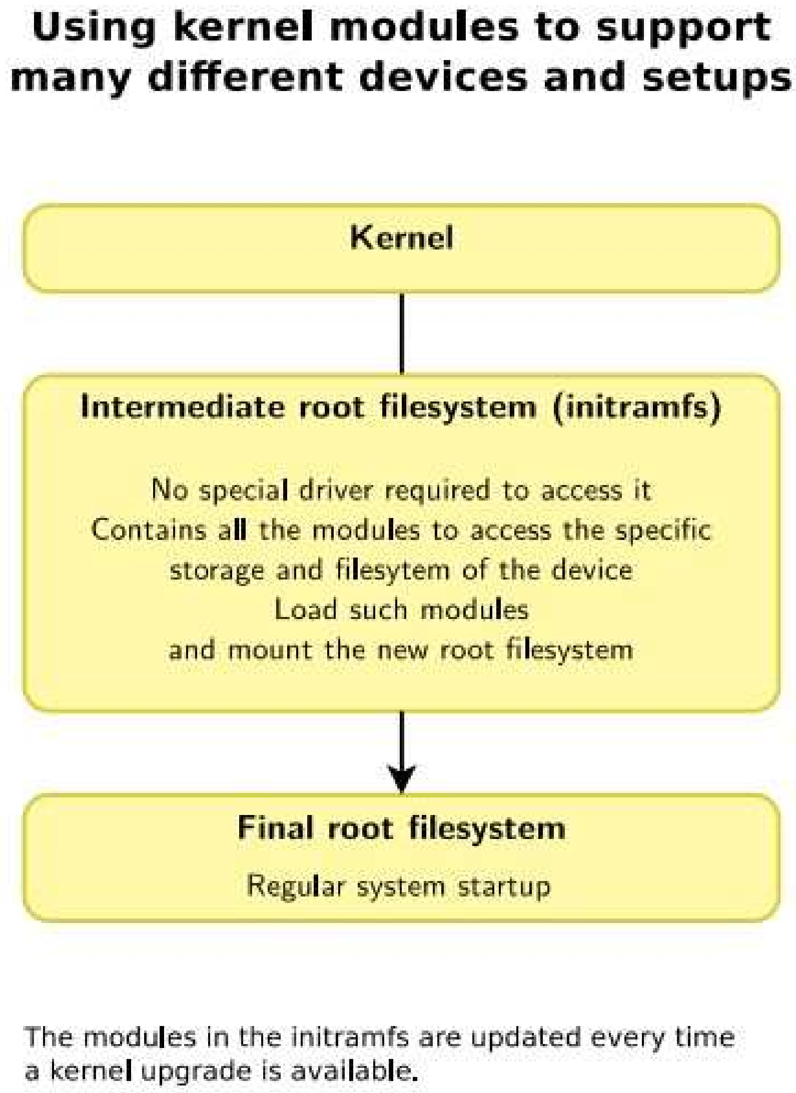
- **sound/**
  - Sound support code and drivers
- **tools/**
  - Code for various user space tools (mostly C, example: perf)
- **usr/**
  - Code to generate an initramfs cpio archive
- **virt/**
  - Virtualization support (KVM)

Kernel modules

- **Kernel or module ?**
  - The kernel image is a single file, resulting from the linking of all object files that correspond to features enabled in the configuration
  - The kernel is loaded in memory by boot loader
- Some features (device drivers, filesystems, etc.) can be compiled as **modules**
  - Modules are plugins that can be load/unloaded dynamically to the kernel
  - Each module **is stored as a separate file in the filesystem**
  - Access to a filesystem is mandatory to use modules
  - This is not possible in the early boot procedure of the kernel, because no filesystem is available

Advantages of modules

- Easy to **develop drivers without rebooting**
- Useful to **keep the kernel image size to the minimum**
- Also useful to **reduce boot time**: don't spend time on device initialization
- **Caution**
  - Once loaded, have full control and privileges in the system
  - No particular protection
  - Only the root user can load and unload modules



Module dependencies

- Some kernel modules can depend on other modules, which need to be loaded first
  - Example: the ubifs module depends on the ubi and mtd modules
- Dependencies are described both in
  - /lib/modules/<kernel-version>/modules.dep and in
  - /lib/modules/<kernel-version>/modules.dep.bin (binary hashed format)
  - These files are generated when you run “make modules_install”

Kernel log

- When a new module is loaded, related information is available in the kernel log
  - The kernel keeps its messages in a circular buffer (so that it doesn't consume more memory with many messages)
  - Kernel log messages are available through the dmesg command
  - Kernel log messages are also displayed in the system console Example: **console=ttyS0 root=/dev/mmcblk0p2 loglevel = 5**
  - Can write to kernel log from user space too. Example: **echo "<n>Debug info"**

Module utilities (1)

- **<module_name>**
  - name of the module file without the trailing .ko
- **modinfo** <module_name> (for modules in /lib/modules)
- **modinfo** <module_path>.ko
  - Gets information about a module without loading it: parameters, license, description and dependencies
- **sudo insmod <module_path>.ko**
  - Tries to load the given module
  - The full path to the module object file must be given

Understanding module loading issues

- When loading a module fails
  - Insmod often doesn’t gave you enough details
  - Details are often available in the kernel log
  - Example:

```
$ sudo insmod ./intr_monitor.ko
insmod: error inserting './intr_monitor.ko': -1 Device or resource busy
$ dmesg
[17549774.552000] Failed to register handler for irq channel 2
```

Module utilities (2)

- sudo **modprobe** <top_module_name>
  - Tries to load **all the dependencies** of the given top module, and then this module
  - Automatically looks in /lib/modules/<version>/modules.dep for the object file corresponding to the given module name
- **lsmod**
  - Display the list of loaded modules

Module utilities (3)

- sudo **rmmod** <module_name>
  - Remove the given module
  - Will only be allowed if the module is no longer in use
- sudo **modprobe** –r <top_module_name>
  - Remove the given top module and all its no longer needed dependencies

Passing parameters to modules

- Find available parameters: **modinfo usb-storage**
- Using **insmod**:
  - sudo insmod ./usb-storage.ko delay_use=0
- Using **modprobe**:
  - Set parameters in /etc/modprobe.conf or in any file in /etc/modprobe.d/:
    options usb-storage delay_use=0
- Using the kernel command line, when the driver is built statically into the kernel:
  - **usb-storage.delay_use=0**
  - **usb-storage** is the driver name
  - **delay_use** is the driver parameter name. It specifies a delay before accessing a USB storage device
  - **0** is the driver parameter value

Check module parameter values

- How to find/edit the current values for the parameters of a loaded module ?
  - Check /sys/module/<name>/parameters
  - There is one file per parameter, containing the parameter value
  - Also possible to change parameter values if these files have write permissions
  - Example:
    - echo 0 > /sys/module/usb_storage/parameters/delay_use

Developing kernel modules

- Hello module

```c
// SPDX-License-Identifier: GPL-2.0
/* hello.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void)
{
  pr_alert("Good morrow to this fair assembly.\n");
  return 0;
}

static void __exit hello_exit(void)
{
  pr_alert("Alas, poor world, what treasure hast thou lost!\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Greeting module");
MODULE_AUTHOR("William Shakespeare");
```

https://frama.link/Q3CNXnom

Hello module

- Code marked as **__init:**
  - Removed after initialization (static kernel or module)
  - See how init memory is reclaimed when the kernel finishes booting

```
[    2.689854] VFS: Mounted root (nfs filesystem) on device 0:15.
[    2.698796] devtmpfs: mounted
[    2.704277] Freeing unused kernel memory: 1024K
[    2.710136] Run /sbin/init as init process
```

- Code marked as **__exit:**
  - Discarded when module compiled statically into kernel, or when module unloading support is not enabled

Hello module explanations

- Headers specific to the Linux kernel: linux/xxx.h
  - No access to the usual C library, we are doing kernel programming
- An initialization function
  - Called when the module is loaded, return an error code (0 on success, negative value on failure)
  - Declared by the **module_init**() macro
- A cleanup function
  - Called when the module is unloaded
  - Declared by the **module_exit()** macro
- Metadata information declared using
  - **MODULE_LICENSE(), MODULE_DESCRIPTION(),** and **MODULE_AUTHOR()**

Compiling a module

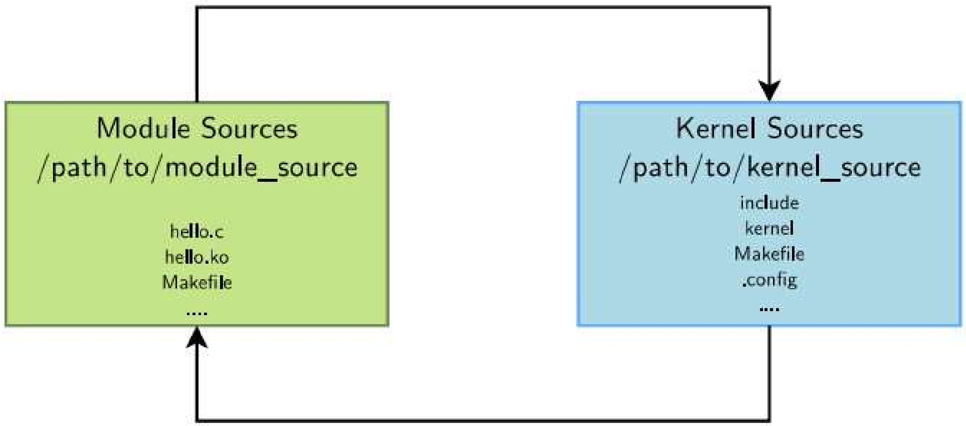
- Two solutions
  - Out of tree, when the code is outside of the kernel source tree, in a different directory
    - Not integrated into the kernel configuration/compilation process
    - Needs to be built separately
    - The driver cannot be built statically, only as a module
  - Inside the kernel tree
    - Well integrated into the kernel configuration/compilation process
    - The driver can be built statically or as a module

Compiling an out-of-tree module

- The source file is hello.c
- Just run make to build the hello.ko file
- KDIR: kernel source or headers directory
- To use below Makefile for any single-file out-of-tree Linux module

```
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
else
KDIR := /path/to/kernel/sources

all:
<tab>$(MAKE) -C $(KDIR) M=$$PWD
endif
```



https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Kernel debugging

- Debugging using messages
  - **printk()**, no longer recommended for new debugging messages
  - The pr_*() family of functions: pr_emerg(), pr_alert(), pr_crit(), pr_err(), pr_warning(), pr_notice(), pr_info(), pr_cont()
  - Defined in “include/linux/printk.h”
  - Example:
    pr_info(“Booting CPU %d\n”, cpu)
  - Here is what you get in the kernel log:

```
[  202.350064] Booting CPU 1
```

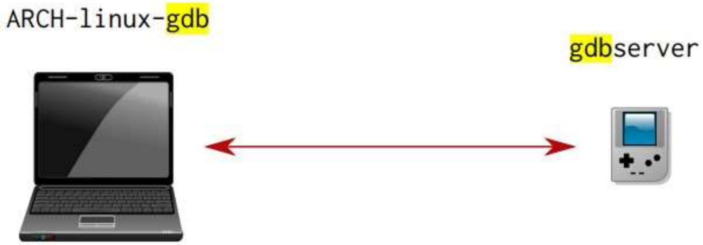
Debugging using messages

- The dev_*() family of functions:
  - dev_emerg(), dev_alert(), dev_crit(), dev_err(), dev_warn(), dev_notice(), dev_info()
  - Take a pointer to **struct device** as first argument, and then a format string with arguments
  - Defined in **“include/linux/dev_printk.h”**
  - To be used in drivers integrated with the Linux device model
  - Example:
    dev_info(&pdev->dev, “in prob\n”)

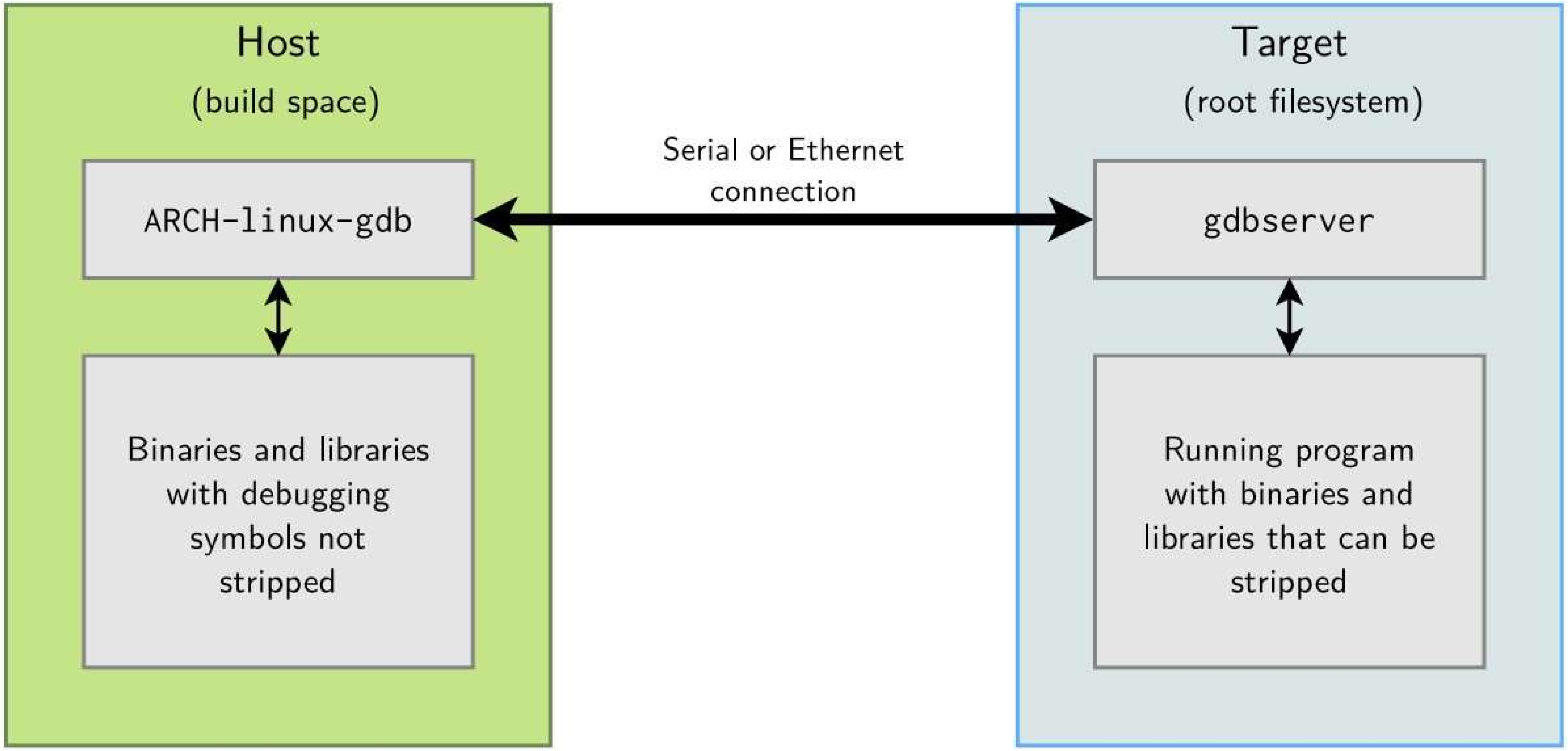
```
[   25.878382] serial 48024000.serial: in probe
[   25.884873] serial 481a8000.serial: in probe
```

Remote debugging

- In a non-embedded environment
  - Debugging takes place using gdb or one of its front-ends
- In an embedded context
  - The target platform environment is too limited to allow direct debugging with gdb (2.4 MB on x86)
- Remote debugging is preferred
  - **ARCH-linux-gdb** is used on the development workstation
  - **gdbserver** is used on target system (only 100 KB on ARM)



Remote debugging: architecture



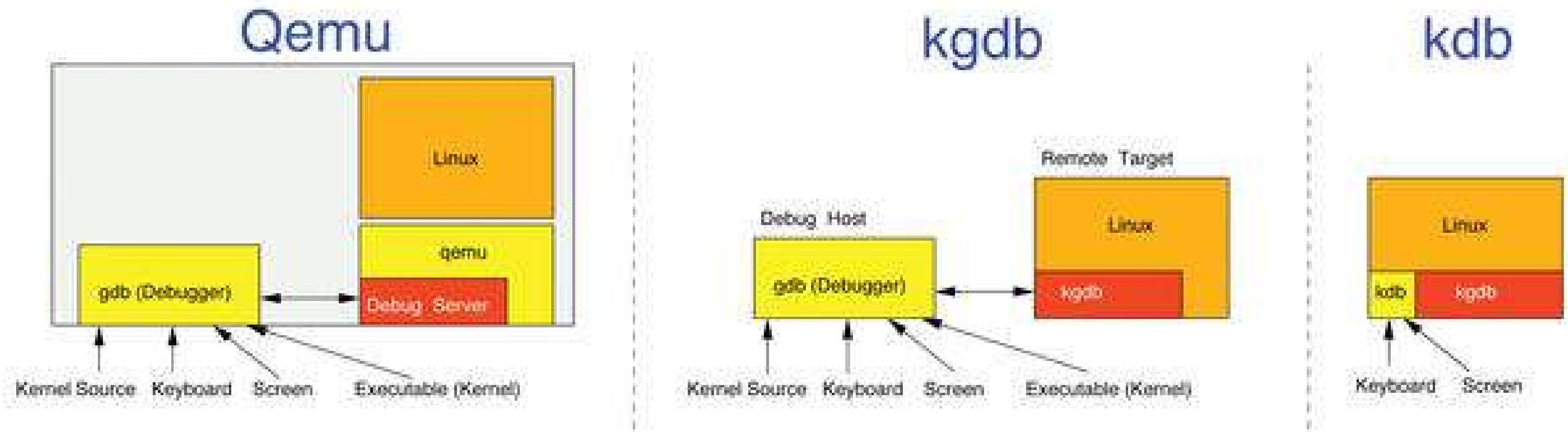
https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Remote debugging: usage

- On the target, run a program through gdbserver
  - gdbserver localhost:<port> <executable> <args>
  - gdbserver /dev/ttyS0 <executable> <args>
- Otherwise, attach gdbserver to an already running program
  - gdbserver –attach localhost:<port> <pid>
- Then, on the host, start ARCH-linux-gdb <executable>, and using the following gdb commands
  - To connect to the target:
  - gdb> target remote <ip-addr>:<port> (networking)
  - gdb> target remote /dev/ttyUSB0 (serial link)

kgdb –A kernel debugger

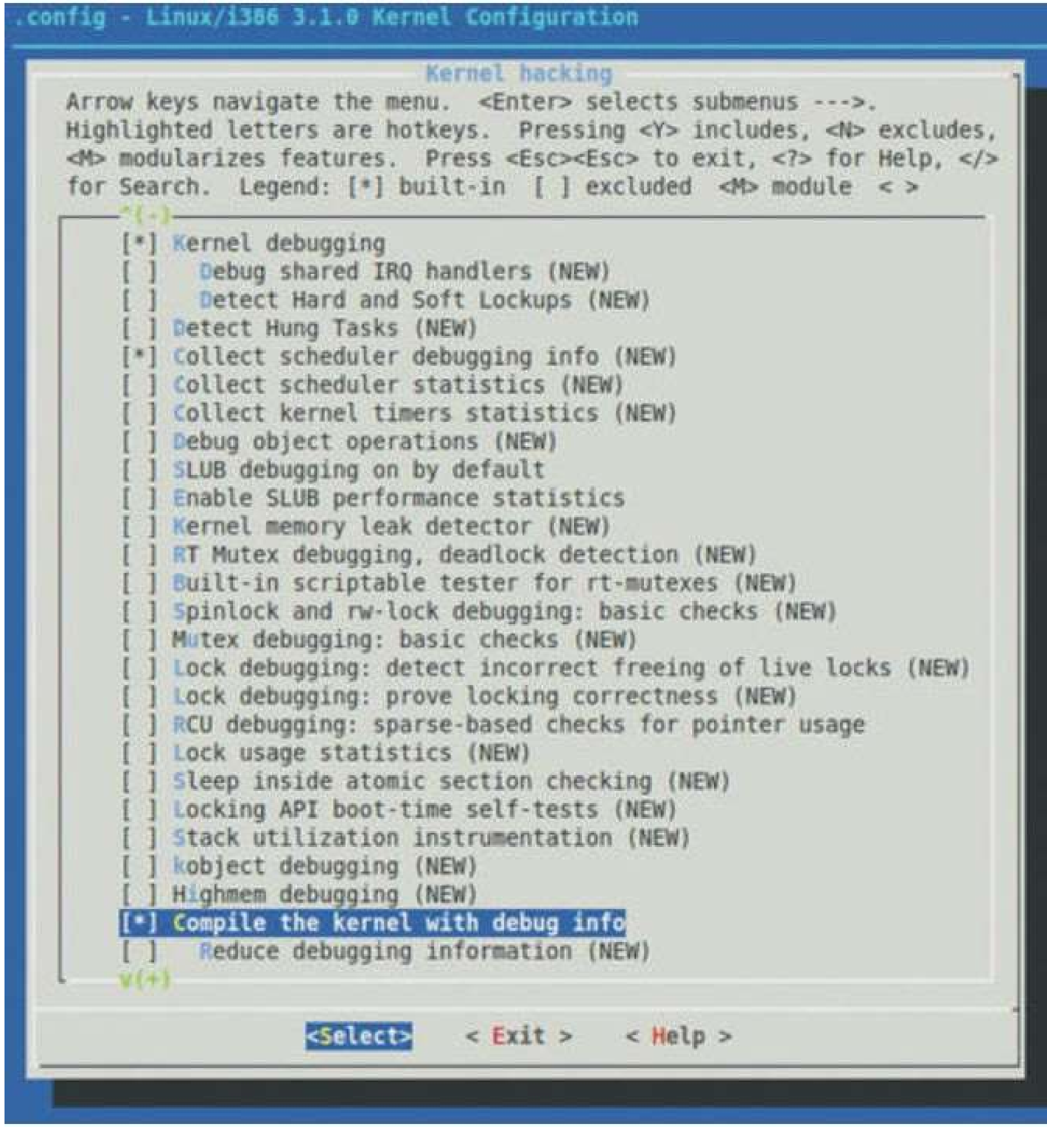
- The execution of the kernel is fully controlled by gdb from another machine, connected through a serial line
- You must include a kgdb I/O driver over serial console, enabled by CONFIG_KGDB_SERIAL_CONSOLE.



https://www.linux-magazine.com/Online/Features/Qemu-and-the-Kernel

How to use kdb/kgdb ?

- To turn on KDB over serial console
- 'make menuconfig'
  - go to "Kernel Hacking" sub-menu
  - turn on "KGDB: kernel debugger", and choose "<Select>" to go to sub-menu
  - verify that "KGDB: use kgdb over the serial console" is set
  - set "KGDB_KDB: include kdb frontend for kgdb"
- save and exit



https://www.linux-magazine.com/Online/Features/Qemu-and-the-Kernel
https://elinux.org/KDB

Enabling kdb

- Connect to the board's console port
  - agent-proxy 2223^2222 localhost /dev/ttyUSB0,115200
  - telnet localhost 2223

https://git.kernel.org/pub/scm/utils/kernel/kgdb/agent-proxy.git

- Configure kgdboc to use the console device
  - **echo ttyO2 > /sys/module/kgdboc/parameters/kgdboc**
  - The console returns a confirmation:
  - kgdb: Registered I/O driver kgdboc.
- Enter kdb mode by sending the **sysrq-g** magic sequence
  - **# echo g > /proc/sysrq-trigger**
  - The console returns:

```
SysRq : DEBUG
Entering kdb (current=0xde63da40, pid 543) due to
Keyboard Entry
kdb>
```

Enabling kdb

- Enter kgdb mode from the **kdb** prompt
  - kdb> kgdb
- Launch the **gdb** debugger on the host workstation
  - (gdb)
- Connect **gdb** to the target
  - (gdb) target remote localhost:2222
- use of kgdb over serial - Start up the agent-proxy and connect and hit a breakpoint a sys_sync
  - https://www.youtube.com/watch?v=nnopzcwvLTs

https://docs.windriver.com/bundle/Wind_River_Linux_Users_Guide_6.0_1/page/1565589.html

Useful commands in kdb

Commands	Meaning
lsmod	Shows where kernel modules are loaded
ps	Displays only the active processes
ps A	Show all the processes
summary	Show kernel version info and memory usage
bt	Get a backtrace of the current process usingn dump_stack()
dmesg	View the kernel syslog buffer
go	Continue the system
bph	Set or display hardware breakpoint

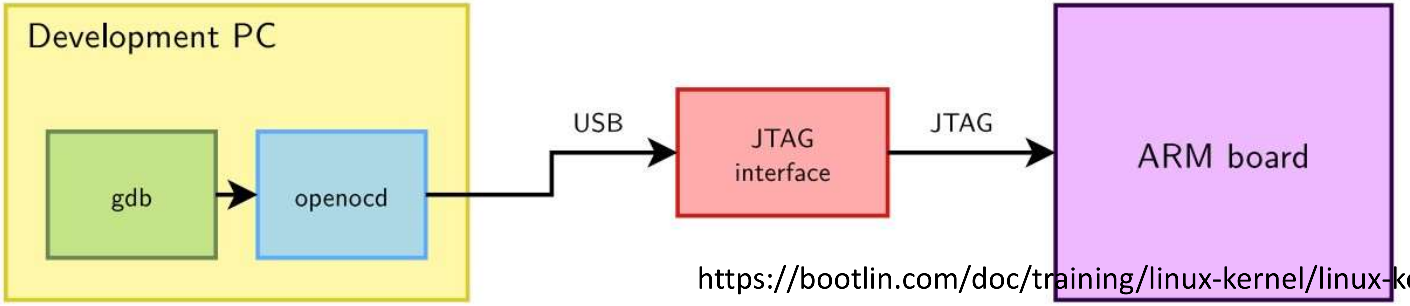
https://www.kernel.org/doc/html/v5.0/dev-tools/kgdb.html

Demo kdb/kgdb

- Example of a call to panic from a test module (without debugger)
  - https://www.youtube.com/watch?v=V6Qc8ppJ_jc
- Example of catching the panic with KDB, and looking up the source line with gdb
  - https://www.youtube.com/watch?v=LqAhY8K3XzI
- Example of a bad access request, and looking up the source line with gdb
  - https://www.youtube.com/watch?v=bBEh_UduX04
- Example of using a hardware breakpoint with kdb
  - https://www.youtube.com/watch?v=MfJU2E0aJwg

Debugging with a JTAG interface

- Two types of JTAG dongles
  - The ones offering a gdb compatible interface, over a serial port or an Ethernet connection
  - The ones not offering a gdb compatible interface are generally supported by OpenOCD (Open On Chip Debugger)
    - OpenOCD is the bridge between the gdb debugging language and the JTAG interface of the target CPU
    - For each board, you need an OpenOCD configuration file



https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Summary

- U-Boot demonstrates the processing of the bootloader
- Linux kernel designs to manage hardware resource with multiple abstractions
- Modules in the Linux kernel enables to load/unload additional features dynamically
- Kgdb – Linux kernel debugger