Linux Kernel Introduction:

Linux kernel main role:

- *Manage all the hardware resources: CPU, memory, I/O.*
- *Provide a set of portable, architecture and hardware independent APIs to allow user space applications and libraries to use the hardware resources.*
- *Handle concurrent accesses and usage of hardware resources from different applications.*

⇒ Example: a single network interface is used by multiple user space applications through various network connections. The kernel is responsible to "multiplex" the hardware resource.

- The main interface between the kernel and user space is the set of system calls.

Pseudo filesystems:

- Linux makes system and kernel information available in user space through pseudo filesystems, sometimes also called virtual filesystems.

- 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. Information gathered by the kernel frameworks managing these devices.*

Linux license:

- The whole Linux sources are Free Software released under the GNU General Public License version 2 (GPL v2).

- This is means:

- *When you receive or buy a device with Linux on it, you should receive the Linux sources, with the right to study, modify and redistribute them.*
- *When you produce Linux based devices, you must release the sources to the recipient, with the same rights, with no restriction.*

Supported hardware architectures:

- See the arch/ directory in the kernel sources: arch/<arch>/Kconfig, arch/<arch>/README, or Documentation/<arch>/

Embedded Linux Kernel Usage

Getting Linux sources:

- The kernel sources are available from https://kernel.org/pub/linux/kernel as full tarballs (complete kernel sources) and patches(differences between two kernel versions).

- Fetch the entire kernel sources and history

- *git clone git: //git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git*

- Create a branch that starts at a specific stable version

- *git checkout -b <name-of-branch> v5.6*

- A minimum uncompressed Linux kernel just sizes 1-2 MB.

- No C library:

** You can't use standard C library functions in kernel code. (printf(), memset(), malloc(),...).*

** The kernel provides similar C functions for your convenience, like printk(), memset(), kmalloc().*

User space device drivers:

In some cases, it is possible to implement device drivers in user space!

Can be used when:

- *The kernel provides a mechanism that allows user space applications to directly access the hardware.*

Possibilities for user space device drivers:

- *USB with libusb, https://libusb.info/*
- *SPI with spidev, Documentation/spi/spidev*
- *I2C with i2cdev, Documentation/i2c/dev-interface*
- *Memory-mapped devices with UIO, including interrupt handling, driver-api/uio-howto*

Kernel source management tools:

***Cscope:** Tool to browse source code*

In Linux kernel sources, two ways of running it:

- *cscope -Rk: All files for all architectures at once*
- *make cscope*

  cscope -d cscope.out

  Only files for your current architecture

Allows searching for a symbol, a definition, functions, strings, files, etc.

***Elixir:** browsing the Linux kernel sources*

Building Kernel:

Kernel configuration

- The kernel configuration is the process of defining the set of options with which you want your kernel to be compiled

- The set of options depends

- *On the target architecture and on your hardware (for device drivers, etc.)*
- *On the capabilities you would like to give to your kernel (network capabilities, filesystems, real-time, etc.). Such generic options are available in all architectures.*

1. Specifying the target architecture

First, specify the architecture for the kernel to build:

- Set ARCH to the name of a directory under arch/: export ARCH=arm

- By default, the kernel build system assumes that the kernel is configured and built for the host architecture (x86 in our case, native kernel compiling)

- The kernel build system will use this setting to:

- *Use the configuration options for the target architecture.*
- *Compile the kernel with source code and headers for the target architecture.*

2. Kernel configuration and build system

- The kernel configuration and build system is based on multiple Makefiles. One only interacts with the main Makefile, present at the top directory of the kernel source tree.

- Interaction takes place

- *using the make tool, which parses the Makefile*
- *through various targets, defining which action should be done (configuration, compilation, installation, etc.).*

Example

- *cd linux-4.14.x/*
- *make <target>*

- The configuration is stored in the .config file at the root of kernel sources

- *Simple text file: CONFIG_PARAM=value*

- As options have dependencies, typically never edited by hand, but through graphical or text interfaces:

- *make xconfig, make gconfig (graphical)*
- *make menuconfig, make nconfig (text)*
- *You can switch from one to another, they all load/save the same .config file, and show the same set of options*

2.1: Initial configuration

- Desktop or server case:

Advisable to start with the configuration of your running kernel, usually available in /boot:

- *cp /boot/config-`uname -r` .config*

- Embedded platform case:

- *Default configuration files are available, usually for each CPU family.*
- *They are stored in arch/<arch>/configs/, and are just minimal .config files (only settings different from default ones).*
- *Run make help to find if one is available for your platform*
- *To load a default configuration file, just run: make cpu_defconfig ==> This will overwrite your existing .config file!*

⇒ Now, you can make configuration changes (make menuconfig...).

- To create your own default configuration file:

- *make savedefconfig: This creates a minimal configuration (non-default settings)*
- *mv defconfig arch/<arch>/configs/myown_defconfig: This way, you can share a reference configuration inside the kernel sources.*

Kernel VS module?

-The kernel image is a single file, resulting from the linking of all object files that correspond to features enabled in the configuration

- *This is the file that gets loaded in memory by the bootloader*
- *All included features are therefore available as soon as the kernel starts, at a time where no filesystem exists*

- Some features (device drivers, filesystems, etc.) can however be compiled as modules

- *These are plugins that can be loaded/unloaded dynamically to add/remove features to the kernel*
- *Each module is stored as a separate file in the filesystem, and therefore 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*

- Kernel option types:

There are different types of options, defined in Kconfig files:

- *bool options, they are either*
  - true (to include the feature in the kernel) or
  - false (to exclude the feature from the kernel)
- *tristate options, they are either*
  - true (to include the feature in the kernel image) or
  - module (to include the feature as a kernel module) or
  - false (to exclude the feature)
- *int options: to specify integer values*
- *hex options: to specify hexadecimal values. Example: CONFIG_PAGE_OFFSET=0xC0000000*
- *string options: to specify string values. Example: CONFIG_LOCALVERSION=-no-network*

⇒ Useful to distinguish between two kernels built from different options

- Kernel option dependencies:

There are dependencies between kernel options, for example, enabling a network driver requires the network stack to be enabled

- Two types of dependencies:

- *depends on dependencies:*
  In this case, option B that depends on option A is not visible until option A is enabled
- *select dependencies:*
  In this case, with option B depending on option A, when option A is enabled, option B is automatically enabled. In particular, such dependencies are used to declare what features a hardware architecture supports.

- With the Show All Options option:

- *make xconfig: allows to see all options, even the ones that cannot be selected because of missing dependencies.*
- *make xconfig: The most common graphical interface to configure the kernel.*
- *make gconfig: GTK based graphical configuration interface. similar to that of make xconfig.*
- *make menuconfig: Useful when no graphics are available. Pretty convenient too!*
- *make oldconfig: Useful to upgrade a .config file from an earlier kernel release.*

⇒ If you edit a .config file by hand, it's useful to run make oldconfig afterwards, to set values to new parameters that could have appeared because of dependency changes.

!!!!! Undoing configuration changes !!!!!

A frequent problem:

- *After changing several kernel configuration settings, your kernel no longer works.*
- *If you don't remember all the changes you made, you can get back to your previous configuration:*
  $ cp .config.old .config
- *All the configuration interfaces of the kernel (xconfig, menuconfig, oldconfig...) keep this .config.old backup copy.*

Compiling and installing the kernel

Choose a compiler

The compiler invoked by the kernel Makefile is $(CROSS_COMPILE)gcc

- *When compiling natively:*
  Leave CROSS_COMPILE undefined and the kernel will be natively compiled for the host architecture using gcc.
- *When using a cross-compiler:*
  To make the difference with a native compiler, cross-compiler executables are prefixed by the name of the target system, architecture and sometimes library.

Examples:

mips-linux-gcc: the prefix is mips-linux-
arm-linux-gnueabi-gcc: the prefix is arm-linux-gnueabi-

⇒ So, you can specify your cross-compiler as follows: export CROSS_COMPILE=arm-linux-gnueabi-

Specifying ARCH and CROSS_COMPILE:

There are actually two ways of defining ARCH and CROSS_COMPILE:

- *Pass ARCH and CROSS_COMPILE on the make command line:*
  make ARCH=arm CROSS_COMPILE=arm-linux- ...
- *Define ARCH and CROSS_COMPILE as environment variables:*
  export ARCH=arm export CROSS_COMPILE=arm-linux-

Kernel compilation:

make:

- *Run it In the main kernel source directory!*
- *Remember to run multiple jobs in parallel if you have multiple CPU cores. Our advice: ncpus * 2 or ncpus + 2, to fully load the CPU and I/Os at all times.*
  Example: make -j 8

⇒ To recompile faster use the ccache compiler cache: export CROSS_COMPILE="ccache riscv64-linux-"

⇒ Kernel compilation results:

- *vmlinux: the raw uncompressed kernel image, in the ELF format, useful for debugging purposes, but cannot be booted.*
- *arch/<arch>/boot/*Image: the final, usually compressed, kernel image that can be booted: bzImage for x86, zImage for ARM, vmlinux.bin.gz for ARC, etc.*
- *arch/<arch>/boot/dts/*.dtb: compiled Device Tree files.*
- *All kernel modules, spread over the kernel source tree, as .ko (Kernel Object) files.*

Kernel installation:

Native case:

- *make install: Does the installation for the host system by default, so needs to be run as root.*

Result Installs:

- */boot/vmlinuz-<version>: Compressed kernel image. Same as the one in arch/<arch>/boot*
- */boot/System.map-<version>: Stores kernel symbol addresses for debugging purposes*
- */boot/config-<version>: Kernel configuration for this version*

Embedded case:

-make install: is rarely used in embedded development, as the kernel image is a single file, easy to handle.
-Another reason is that there is no standard way to deploy and use the kernel image.
-Therefore making the kernel image available to the target is usually manual or done through scripts in build systems.
-It is however possible to customize the make install behavior in arch/<arch>/boot/install.sh

Module installation:

Native case:

- *make modules_install: Does the installation for the host system by default, so needs to be run as root ⇒ Installs all modules in /lib/modules/<version>/*
- *kernel/: Module .ko (Kernel Object) files, in the same directory structure as in the sources.*
- *modules.alias, modules.alias.bin: Aliases for module loading utilities. Used to find drivers for devices.*
  ***Example line:** alias usb:v066Bp20F9d*dc*dsc*dp*ic*isc*ip*in* asix*
- *modules.dep, modules.dep.bin: Module dependencies*
- *modules.symbols, modules.symbols.bin: Tells which module a given symbol belongs to.*

Embedded case:

-In embedded development, you can't directly use make modules_install as it would install target modules in /lib/modules on the host!
-The INSTALL_MOD_PATH variable is needed to generate the module related files and install the modules in the target root filesystem instead of your host root filesystem: make INSTALL_MOD_PATH=<dir>/ modules_install

Kernel cleanup targets

- make clean: Clean-up generated files (to force re-compilation)
- make mrproper: Remove all generated files. Needed when switching from one architecture to another.
Caution: it also removes your .config file!
- make distclean: Also remove editor backup and patch reject files (mainly to generate patches)

Booting the kernel:

Device Tree:

- The Device Tree (DT) was created for PowerPC, and later was adopted by other architectures (ARM, ARC...). Now Linux has DT support in most architectures, at least for specific systems (for example for the OLPC on x86).
- A Device Tree Source, written by kernel developers, is compiled into a binary Device Tree Blob, and needs to be passed to the kernel at boot time:

- There is one different Device Tree for each board/platform supported by the kernel, available in arch/arm/boot/dts/<board>.dtb.
- The bootloader must load both the kernel image and the Device Tree Blob in memory before starting the kernel.

Customize your board device tree:

Needed for embedded board users:

- To describe external devices attached to non-discoverable busses (such as I2C) and configure them.
- To configure pin muxing: choosing what SoC signals are made available on the board external connectors
- To configure some system parameters: flash partitions, kernel command line.

Booting with U-Boot:

- Recent versions of U-Boot can boot the zImage binary.
- Older versions require a special kernel image format: uImage
    - uImage is generated from zImage using the mkimage tool. It is done automatically by the kernel make uImage target.
    - On some ARM platforms, make uImage requires passing a LOADADDR environment variable, which indicates at which physical memory address the kernel will be executed.

-In addition to the kernel image, U-Boot can also pass a Device Tree Blob to the kernel.

⇒ The typical boot process is therefore:

1. Load zImage or uImage at address X in memory
2. Load <board>.dtb at address Y in memory
3. Start the kernel with bootz X - Y (zImage case), or bootm X - Y (uImage case).

The - in the middle indicates no initramfs.

Kernel command line:

-In addition to the compile time configuration, the kernel behavior can be adjusted with no recompilation using the kernel command line

-The kernel command line is a string that defines various arguments to the kernel. It is very important for system configuration

- root= for the root filesystem
- console= for the destination of kernel messages
  Example: console=ttyS0 root=/dev/mmcblk0p2 rootwait

-This kernel command line can be, in order of priority (highest to lowest):

- Passed by the bootloader. In U-Boot, the contents of the bootargs environment variable is automatically passed to the kernel.
- Specified in the Device Tree (for architectures which use it)
- Built into the kernel, using the CONFIG_CMDLINE option.

Using kernel modules

Advantages of modules:

- Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...
- Also useful to reduce boot time: you don't spend time initializing devices and kernel features that you only need later
- Useful to keep the kernel image size to the minimum
- To increase security, possibility to allow only signed modules, or to disable module support entirely.

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
- /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.
    - Kernel log messages are available through the dmesg command (diagnostic message)
    - Kernel log messages are also displayed in the system console (console messages can be filtered by level using the loglevel kernel command line parameter, or completely disabled with the quiet parameter).
      Example: console=ttyS0 root=/dev/mmcblk0p2 loglevel=5

⇒ Note that you can write to the kernel log from user space too: echo "<n>Debug info" > /dev/kmsg

Module Utility:

- 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.

When loading a module fails, insmod often doesn't give you enough details!Details are often available in the kernel log.

- *sudo modprobe <module_name>:*

Most common usage of modprobe: tries to load all the modules the given <module_name> depends on, and then this module. Lots of other options are available. modprobe automatically looks in /lib/modules/<version>/ for the object file corresponding to the given module name.

- *lsmod:*

Displays the list of loaded modules Compare its output with the contents of /proc/modules!

- *sudo rmmod <module_name>*

Tries to remove the given module. Will only be allowed if the module is no longer in use (for example, no more processes opening a device file)

- *sudo modprobe -r <module_name>:*

Tries to remove the given module and all dependent modules (which are no longer needed after removing the module)

Passing parameters to modules:

- *Find available parameters: modinfo usb-storage*
- *Through insmod: sudo insmod ./usb-storage.ko delay_use=0*
- *Through modprobe: Set parameters in /etc/modprobe.conf or in any file in /etc/modprobe.d/:*

options: usb-storage delay_use=0

- *Through 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 (useful for rotating devices).*
  - *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 (depends on the module code).*
- *Example: echo 0 > /sys/module/usb_storage/parameters/delay_use*

Developing Kernel Module

Hello Module Explanations:

-Headers specific to the Linux kernel: linux/xxx.h: No access to the usual C library, we're doing kernel programming

-An initialization function

- *Called when the module is loaded, returns an error code (0 on success, negative value on failure)*
- *Declared by the **module_init()** macro: the name of the function doesn't matter, even though <modulename>_init() is a convention.*

-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()

Symbols Exported to Modules

- *From a kernel module, only a limited number of kernel functions can be called*
- *Functions and variables have to be explicitly exported by the kernel to be visible to a kernel module*
- *Two macros are used in the kernel to export functions and variables:*

  - *EXPORT_SYMBOL(symbolname): which exports a function or variable to all modules*
  - *EXPORT_SYMBOL_GPL(symbolname): which exports a function or variable only to GPL modules*

Module License

- *Used to restrict the kernel functions that the module can use if it isn't a GPL licensed module*
- *Useful for users to check that their system is 100% free*

Compiling a Module

Two Solution:

1. *Out of tree: When the code is outside of the kernel source tree, in a different directory*
2. *Inside the kernel tree: Well integrated into the kernel configuration/compilation process.Driver can be built statically if needed.*

Compiling an out-of-tree Module:

Use a makefile: Just run make to build the hello.ko file

-The module Makefile is interpreted with KERNELRELEASE undefined, so it calls the kernel Makefile, passing the module directory in the M variable

-The kernel Makefile knows how to compile a module, and thanks to the M variable, knows where the Makefile for our module is. This module Makefile is then interpreted with KERNELRELEASE defined, so the kernel sees the obj-m definition.

Modules and Kernel Version:

-To be compiled, a kernel module needs access to the kernel headers, containing the definitions of functions, types and constants.

-Two solutions

- *Full kernel sources (configured + make modules_prepare)*
- *Only kernel headers (linux-headers-* packages in Debian/Ubuntu distributions, or directory created by make headers_install)*

-The sources or headers must be configured, Many macros or functions depend on the configuration.

- A kernel module compiled against version X of kernel headers will not load in kernel version Y

- *modprobe / insmod will say Invalid module format*

New Driver in Kernel Sources

-To add a new driver to the kernel sources:

1. Add your new source file to the appropriate source directory. Example: drivers/usb/serial/navman.c

2. Describe the configuration interface for your new driver by adding the following lines to the Kconfig file in this directory:

```
config USB_SERIAL_NAVMAN
    tristate "USB Navman GPS device"
    depends on USB_SERIAL
    help
    To compile this driver as a module, choose M here: the module will be called navman.
```

3. Add a line in the Makefile file based on the Kconfig setting: obj-$(CONFIG_USB_SERIAL_NAVMAN) += navman.o

⇒ It tells the kernel build system to build navman.c when the USB_SERIAL_NAVMAN option is enabled. It works both if compiled statically or as a module.

4. Run make xconfig and see your new options. Run make and your new files are compiled!

!!! Useful Kernel API 130 just for read

Processes, scheduling and interrupts

Process, thread?

- In UNIX, a process is created using fork() and is composed of

- *An address space, which contains the program code, data, stack, shared libraries, etc.*
- *One thread, entity known by the scheduler.*
- *Upon creation, a process contains one thread*

- Additional threads can be created inside an existing process: using pthread_create()

- *They run in the same address space as the initial thread of the process*
- *They start executing a function passed as argument to pthread_create()*

- In kernel space, each thread running in the system is represented by a structure of type struct task_struct

- From a scheduling point of view, it makes no difference between the initial thread of a process and all additional threads created dynamically using pthread_create()

** a Thread Life page 305*

Execution of system calls:

Process executed in user space -> system call or exception(kernel executed) -> process continued in user space.

Sleeping:

Sleeping is needed when a process (user space or kernel space) is waiting for data.

How to sleep?

- Must declare a wait queue, which will be used to store the list of threads waiting for an event.

- Dynamic queue declaration:

- *Typically one queue per device managed by the driver*
- *It's convenient to embed the wait queue inside a per-device data structure.*

- Static queue declaration:

- *Using a global variable when a global resource is sufficient*
- *DECLARE_WAIT_QUEUE_HEAD(module_queue);*

- Several ways to make a kernel process sleep:

void wait_event(queue, condition);

- *Sleeps until the task is woken up and the given C expression is true. Caution: can't be interrupted (can't kill the user space process!)*

int wait_event_killable(queue, condition);

- *Can be interrupted, but only by a fatal signal (SIGKILL). Returns -ERESTARTSYS if interrupted.*

int wait_event_interruptible(queue, condition);

- *Can be interrupted by any signal. Returns -ERESTARTSYS if interrupted.*

int wait_event_timeout(queue, condition, timeout);

- *Also stops sleeping when the task is woken up or the timeout expired (a timer is used). Returns 0 if the timeout elapsed, non-zero if the condition was met.*

int wait_event_interruptible_timeout(queue, condition, timeout);

- *Returns 0 if the timeout elapsed, -ERESTARTSYS if interrupted, positive value if the condition was met.*

Waking up!

Typically done by interrupt handlers when data sleeping processes are waiting for become available.

wake_up(&queue);

- *Wakes up all processes in the wait queue*

wake_up_interruptible(&queue);

- *Wakes up all processes waiting in an interruptible sleep on the given queue*

Exclusive vs. non-exclusive

wait_event_interruptible(): puts a task in a non-exclusive wait.

⇒ All non-exclusive tasks are woken up by wake_up() / wake_up_interruptible()

wait_event_interruptible_exclusive(): puts a task in an exclusive wait.

⇒ wake_up() / wake_up_interruptible() wakes up all non-exclusive tasks and only one exclusive task

⇒ wake_up_all() / wake_up_interruptible_all() wakes up all non-exclusive and all exclusive tasks

- Exclusive sleeps are useful to avoid waking up multiple tasks when only one will be able to "consume" the event.

- Non-exclusive sleeps are useful when the event can "benefit" to multiple tasks.

Interrupt Management

Registering an interrupt handler

- The managed API is recommended:

*int devm_request_irq(struct device *dev, unsigned int irq, irq_handler_t handler, unsigned long irq_flags, const char *devname, void *dev_id);*

- *device for automatic freeing at device or module release time.*
- *irq is the requested IRQ channel. For platform devices, use platform_get_irq() to retrieve the interrupt number.*
- *handler is a pointer to the IRQ handler*
- *irq_flags are option masks (see next slide)*
- *devname is the registered name (for /proc/interrupts)*
- *dev_id is an opaque pointer. It can typically be used to pass a pointer to a per-device data structure. It cannot be NULL as it is used as an identifier for freeing interrupts on a shared line.*

Releasing an interrupt handler:

*void devm_free_irq(struct device *dev, unsigned int irq, void *dev_id);*

- Explicitly release an interrupt handler.

⇒ Here are the most frequent irq_flags bit values in drivers (can be combined):

- *IRQF_SHARED: interrupt channel can be shared by several devices.*
  When an interrupt is received, all the interrupt handlers registered on the same interrupt line are called.
  This requires a hardware status register telling whether an IRQ was raised or not.
- *RQF_ONESHOT: for use by threaded interrupts (see next slides). Keeping the interrupt line disabled until the thread function has run.*

Interrupt handler constraints:

-No guarantee in which address space the system will be in when the interrupt occurs: can't transfer data to and from user space.

-Interrupt handler execution is managed by the CPU, not by the scheduler. Handlers can't run actions that may sleep, because there is nothing to resume their execution. In particular, need to allocate memory with GFP_ATOMIC.
-Interrupt handlers are run with all interrupts disabled on the local CPU. Therefore, they have to complete their job quickly enough, to avoiding blocking interrupts for too long.

Interrupt handler prototype:

*irqreturn_t foo_interrupt(int irq, void *dev_id)*

- *irq: the IRQ number*
- *dev_id: the per-device pointer that was passed to devm_request_irq()*

⇒ Return value

- *IRQ_HANDLED: recognized and handled interrupt*
- *IRQ_NONE: used by the kernel to detect spurious interrupts, and disable the interrupt line if none of the interrupt handlers has handled the interrupt.*

Typical interrupt handler's job:

- Acknowledge the interrupt to the device (otherwise no more interrupts will be generated, or the interrupt will keep firing over and over again)
- Read/write data from/to the device
- Wake up any process waiting for such data, typically on a per-device wait queue:

wake_up_interruptible(&device_queue);

Threaded interrupts

The kernel also supports threaded interrupts:
-The interrupt handler is executed inside a thread.
-Allows to block during the interrupt handler, which is often needed for I2C/SPI devices as the interrupt handler needs to communicate with them.
-Allows to set a priority for the interrupt handler execution, which is useful for real-time usage of Linux:

*int devm_request_threaded_irq(struct device *dev, unsigned int irq, irq_handler_t handler, irq_handler_t thread_fn, unsigned long flags, const char *name, void *dev);*

- *handler, "hard IRQ" handler*
- *thread_fn, executed in a thread*

Top half and bottom half processing:

Splitting the execution of interrupt handlers in 2 parts:

- *Top half: This is the real interrupt handler, which should complete as quickly as possible since all interrupts are disabled. It takes the data out of the device and if substantial post-processing is needed, schedule a bottom half to handle it.*
- *Bottom half: Is the general Linux name for various mechanisms which allow to postpone the handling of interrupt-related work. Implemented in Linux as softirqs, tasklets or workqueues.*

Softirqs:

-Softirqs are a form of bottom half processing
-The softirqs handlers are executed with all interrupts enabled, and a given softirq handler can run simultaneously on multiple CPUs
-They are executed once all interrupt handlers have completed, before the kernel resumes scheduling processes, so sleeping is not allowed.
-The number of softirqs is fixed in the system, so softirqs are not directly used by drivers, but by complete kernel subsystems (network, etc.)
-The list of softirqs is defined in include/linux/interrupt.h:
***Example:** usage of softirqs - NAPI: New API*
-Interface in the Linux kernel used for interrupt mitigation in network drivers
-Principe: when the network traffic exceeds a given threshhold ("budget"), disable network interrupts and consume incoming packets through a polling function, instead of processing each new packet with an interrupt.
-This reduces overhead due to interrupts and yields better network throughput. The polling function is run by: napi_schedule(), which uses NET_RX_SOFTIRQ.

Tasklet:

-Tasklets are executed within the HI_SOFTIRQ and TASKLET_SOFTIRQ softirqs. They are executed with all interrupts enabled, but a given tasklet is guaranteed to execute on a single CPU at a time.
-Tasklets are typically created with the tasklet_init() function, when your driver manages multiple devices, otherwise statically with DECLARE_TASKLET(). A tasklet is simply implemented as a function. Tasklets can easily be used by individual device drivers, as opposed to softirqs.
-The interrupt handler can schedule tasklet execution with:

- *tasklet_schedule(): to get it executed in TASKLET_SOFTIRQ*
- *tasklet_hi_schedule(): to get it executed in HI_SOFTIRQ (highest priority)*

Workqueues:

-Workqueues are a general mechanism for deferring work. It is not limited in usage to handling interrupts. It can typically be used for background work which can be scheduled.

-The function registered as workqueue is executed in a thread, which means:

- *All interrupts are enabled*
- *Sleeping is allowed*

-A workqueue, usually allocated in a per-device structure, is registered with INIT_WORK() and typically triggered with queue_work()

-The complete API, in include/linux/workqueue.h.

*** Interrupt management summary ****

Device driver:

- *In the probe() function, for each device, use devm_request_irq() to register an interrupt handler for the device's interrupt channel.*

Interrupt handler

- *Called when an interrupt is raised.*
- *Acknowledge the interrupt*
- *If needed, schedule a per-device tasklet taking care of handling data.*
- *Wake up processes waiting for the data on a per-device queue*

Device driver

- *In the remove() function, for each device, the interrupt handler is automatically unregistered*

Concurrent Access to Resources: Locking

Sources of concurrency issues

-Concurrency arises because of:

- *Interrupts, which interrupts the current thread to execute an interrupt handler. They may be using shared resources (memory addresses, hardware registers...)*
- *Kernel preemption, if enabled, causes the kernel to switch from the execution of one system call to another. They may be using shared resources.*
- *Multiprocessing, in which case code is really executed in parallel on different processors, and they may be using shared resources as well.*

⇒ The solution is to keep as much local state as possible and for the shared resources that can't be made local (such as hardware ones), use locking.

Linux mutexes

-The kernel's main locking primitive. It's a binary lock. Note that counting locks (semaphores) are also available, but used 30x less frequently.

-The process requesting the lock blocks when the lock is already held. Mutexes can therefore only be used in contexts where sleeping is allowed.

Mutex definition:

- *#include <linux/mutex.h>*

Initializing a mutex statically (unusual case):

- *DEFINE_MUTEX(name);*

*-Or **initializing a mutex dynamically** (the usual case, on a per-device basis):*

- *void mutex_init(struct mutex *lock);*

Locking and Unlocking Mutexes

*void mutex_lock(struct mutex *lock): Tries to lock the mutex, sleeps otherwise.*

*int mutex_lock_killable(struct mutex *lock): Same, but can be interrupted by a fatal (SIGKILL) signal. If interrupted, returns a non zero value and doesn't hold the lock.*

*int mutex_lock_interruptible(struct mutex *lock): Same, but can be interrupted by any signal.*

*int mutex_trylock(struct mutex *lock): Never waits. Returns a non zero value if the mutex is not available.*

*int mutex_is_locked(struct mutex *lock): Just tells whether the mutex is locked or not.*

*void mutex_unlock(struct mutex *lock): Releases the lock. Do it as soon as you leave the critical section.*

Spinlocks

-Locks to be used for code that is not allowed to sleep (interrupt handlers), or that doesn't want to sleep (critical sections). Be very careful not to call functions which can sleep!

-Spinlocks never sleep and keep spinning in a loop until the lock is available. The critical section protected by a spinlock is not allowed to sleep.

Initializing Spinlocks:

-Statically (unusual): DEFINE_SPINLOCK(my_lock);

-Dynamically (the usual case, on a per-device basis): void spin_lock_init(spinlock_t *lock);

Using spinlock:

Several variants, depending on where the spinlock is called:

void spin_lock(spinlock_t *lock);

void spin_unlock(spinlock_t *lock);

- Used for locking in process context (critical sections in which you do not want to sleep).
- Kernel preemption on the local CPU is disabled. We need to avoid deadlocks because of preemption from processes that want to get the same lock

void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);

void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);

- Disables / restores IRQs on the local CPU.
- Typically used when the lock can be accessed in both process and interrupt context.
- We need to avoid deadlocks because of interrupts that want to get the same lock.

void spin_lock_bh(spinlock_t *lock);

void spin_unlock_bh(spinlock_t *lock);

- Disables software interrupts, but not hardware ones.
- Useful to protect shared data accessed in process context and in a soft interrupt (bottom half).
- No need to disable hardware interrupts in this case.

More deadlock Situations

Rule 1: don't call a function that can try to get access to the same lock.

Rule 2: if you need multiple locks, always acquire them in the same order!

⇒ locking can have a strong negative impact on system performance. In some situations, you could do without it:

- By using lock-free algorithms like Read Copy Update (RCU).
- RCU API available in the kernel.
- When available, use atomic operations.

Atomic Variable:

-Useful when the shared resource is an integer value. Even an instruction like n++ is not guaranteed to be atomic on all processors!

- Atomic operations definitions: #include <asm/atomic.h>

- atomic_t

- Atomic operations (main ones):

Set or read the counter:

- void atomic_set(atomic_t *v, int i);
- int atomic_read(atomic_t *v);

Operations without return value:

- void atomic_inc(atomic_t *v);
- void atomic_dec(atomic_t *v);
- void atomic_add(int i, atomic_t *v);
- void atomic_sub(int i, atomic_t *v);

Atomic Bit Operations

-Supply very fast, atomic operations. On most platforms, apply to an unsigned long * type.

- Apply to a void * type on a few others.

Set, clear, toggle a given bit:

- void set_bit(int nr, unsigned long * addr);
- void clear_bit(int nr, unsigned long * addr);
- void change_bit(int nr, unsigned long * addr);

Test bit value:

- int test_bit(int nr, unsigned long *addr);

Test and modify (return the previous value):

- int test_and_set_bit(...);
- int test_and_clear_bit(...);
- int test_and_change_bit(...);

* Kernel Locking Summary:
-Use mutexes in code that is allowed to sleep
-Use spinlocks in code that is not allowed to sleep (interrupts) or for which sleeping would be too costly (critical sections)
-Use atomic operations to protect integers or addresses

Kernel debugging

Debugging using message:
Three APIs are available

- The old 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() and the special pr_debug(): Defined in include/linux/printk.h
- The dev_*() family of functions: dev_emerg(), dev_alert(), dev_crit(), dev_err(), dev_warn(), dev_notice(), dev_info() and the special dev_dbg(): Defined in include/linux/device.h

pr_debug() and dev_dbg()
-When the driver is compiled with DEBUG defined, all these messages are compiled and printed at the debug level.
DEBUG can be defined by #define DEBUG at the beginning of the driver, or using:
ccflags-$(CONFIG_DRIVER) += -DDEBUG in the Makefile
-When the kernel is compiled with CONFIG_DYNAMIC_DEBUG, then these messages can dynamically be enabled on a per-file, per-module or per-message basis
-When neither DEBUG nor CONFIG_DYNAMIC_DEBUG are used, these messages are not compiled in.
Configuring the priority
-Each message is associated to a priority, ranging from 0 for emergency to 7 for debug, as specified in include /linux/kern_levels.h.
-All the messages, regardless of their priority, are stored in the kernel log ring buffer

- Typically accessed using the dmesg command

-Some of the messages may appear on the console, depending on their priority and the configuration of

- The loglevel kernel parameter, which defines the priority number below which messages are displayed on the console. Details in admin-guide/kernel-parameters.
  Examples: loglevel=0: no message, loglevel=8: all messages
- The value of /proc/sys/kernel/printk: which allows to change at runtime the priority above which messages are displayed on the console.

DebugFS:
A virtual filesystem to export debugging information to user space.

- Kernel configuration: CONFIG_DEBUG_FS

-The debugging interface disappears when Debugfs is configured out.
-You can mount it as follows:

- sudo mount -t debugfs none /sys/kernel/debug

DebugFS API
- Create a sub-directory for your driver:

- struct dentry *debugfs_create_dir(const char *name, struct dentry *parent);

- Expose an integer as a file in DebugFS. Example:

- struct dentry *debugfs_create_u8(const char *name, mode_t mode, struct dentry *parent,u8 *value);

- Expose a binary blob as a file in DebugFS:

- struct dentry *debugfs_create_blob(const char *name, mode_t mode, struct dentry *parent, struct debugfs_blob_wrapper *blob);

kgdb - A kernel debugger
- The execution of the kernel is fully controlled by gdb from another machine, connected through a serial line.
- Can do almost everything, including inserting breakpoints in interrupt handlers.

Porting the Linux Kernel to an ARM Board

-The Linux kernel supports a lot of different CPU architectures, Each of them is maintained by a different group of contributors.
-The organization of the source code and the methods to port the Linux kernel to a new board are therefore very architecture-dependent:

- For example, some architectures use the Device Tree, some do not.

Architecture, CPU and Machine

-In the source tree, each architecture has its own directory

- *arch/arm/ for the ARM architecture*

-This directory contains generic ARM code

- *boot/, common/, configs/, kernel/, lib/, mm/, nwfpe/, vfp/, oprofile/, tools/ and several others.*

- And many directories for different SoC families

- *mach-* directories: mach-pxa/ for PXA CPUs, mach-imx/ for Freescale iMX CPUs, etc.*

*- Some CPU types share some code, in directories named plat-**

- Device Tree source files in arch/arm/boot/dts/.

- Each board supported by the kernel was associated to an unique machine ID. The Linux kernel was defining a machine structure for each board, which associates the machine ID with a set of information and callbacks.The bootloader had to pass the machine ID to the kernel in a specific ARM register.

⇒ This way, the kernel knew what board it was booting on, and which init callbacks it had to execute.

The Device Tree and the ARM cleanup

-First, the Device Tree was introduced on ARM: instead of using C code to describe SoCs and boards, a specialized language is used.

-Second, many driver infrastructures were created to replace custom code in arch/arm/mach-<soc>:

- *The common clock framework in drivers/clk/*
- *The pinctrl subsystem in drivers/pinctrl/*
- *The irqchip subsystem in drivers/irqchip/*
- *The clocksource subsystem in drivers/clocksource/*

Adding the support for a new ARM board

Provided the SoC used on your board is supported by the Linux kernel:

1. Create a Device Tree file in arch/arm/boot/dts/, generally named <soc-name>-<board-name>.dts, and make it include the relevant SoC .dtsi file.

⇒ Your Device Tree will describe all the SoC peripherals that are enabled, the pin muxing, as well as all the devices on the board.

2. Modify arch/arm/boot/dts/Makefile to make sure your Device Tree gets built as a DTB during the kernel build.

3. If needed, develop the missing device drivers for the devices that are on your board outside the SoC.

***Example:** the Crystalfontz CFA-10036 platform see page 377.*

Understanding the SoC support

-Let's consider another ARM platform here, the Marvell Armada 370/XP.

-For this platform, the core of the SoC support is located in arch/arm/mach-mvebu/

-The board-v7.c file contains the "entry point" of the SoC definition, the DT_MACHINE_START .. MACHINE_END definition:

- *Defines the list of platform compatible strings that will match this platform, in this case marvell,armada-370-xp. This allows the kernel to know which DT_MACHINE structure to use depending on the DTB that is passed at boot time.*
- *Defines various callbacks for the platform initialization, the most important one being the .init_machine callback, running initialization code for the associated SoC.*

Components of the minimal SoC support

The minimal SoC support consists of

- *An SoC entry point file, arch/arm/mach-mvebu/board-v7.c*
- *At least one SoC .dtsi DT and one board .dts DT, in arch/arm/boot/dts/*
- *A interrupt controller driver, drivers/irqchip/irq-armada-370-xp.c*
- *A timer driver, drivers/clocksource/timer-armada-370-xp.c*
- *An earlyprintk implementation to get early messages from the console, arch/arm/Kconfig.debug and arch/arm/include/debug/*
- *A serial port driver in drivers/tty/serial/. For Armada 370/XP, the 8250 driver drivers/tty/serial/8250/ is used.*

-This allows to boot a minimal system up to user space, using a root filesystem in initramfs.

Extending the minimal SoC support

Once the minimal SoC support is in place, the following core components should be added:

- *Support for the clocks. Usually requires some clock drivers, as well as DT representations of the clocks.*
  See drivers/clk/mvebu/ for Armada 370/XP clock drivers.

- *Support for pin muxing, through the pinctrl subsystem. See drivers/pinctrl/mvebu/ for the Armada 370/XP drivers.*
- *Support for GPIOs, through the GPIO subsystem. See drivers/gpio/gpio-mvebu.c for the Armada 370/XP GPIO driver.*
- *Support for SMP, through struct smp_operations. See arch/arm/mach-mvebu/platsmp.c.*

Adding device driver:

Once the core pieces of the SoC support have been implemented, the remaining part is to add drivers for the different hardware blocks:

- *Ethernet driver, in drivers/net/ethernet/marvell/mvneta.c*
- *SATA driver, in drivers/ata/sata_mv.c*
- *I2C controller driver, in drivers/i2c/busses/i2c-mv64xxx.c*
- *SPI controller driver, in drivers/spi/spi-orion.c*
- *PCIe controller driver, in drivers/pci/controller/pci-mvebu.c*
- *USB controller driver, in drivers/usb/host/ehci-orion.c*