TAIPEI TECH 國立臺北科技大學 National Taipei University of Technology

Department of Computer Science and Information Engineering

Embedded System

The Linux Kernel & Linux Kernel Module

Shuo-Han Chen (陳碩漢),
shchen@ntut.edu.tw

History

- The Linux kernel is one component of a system, which also requires libraries and applications to provide features to end users.
- The Linux kernel was created as a hobby in 1991 by a Finnish student, Linus Torvalds.
  - Linux quickly started to be used as the kernel for free software operating systems
- Linus Torvalds has been able to create a large and dynamic developer and user community around Linux.
- Nowadays, more than one thousand people contribute to each kernel release, individuals or companies big and small.

Linus Torvalds in 2014
Image credits (Wikipedia):
https://bit.ly/2UIa1TD

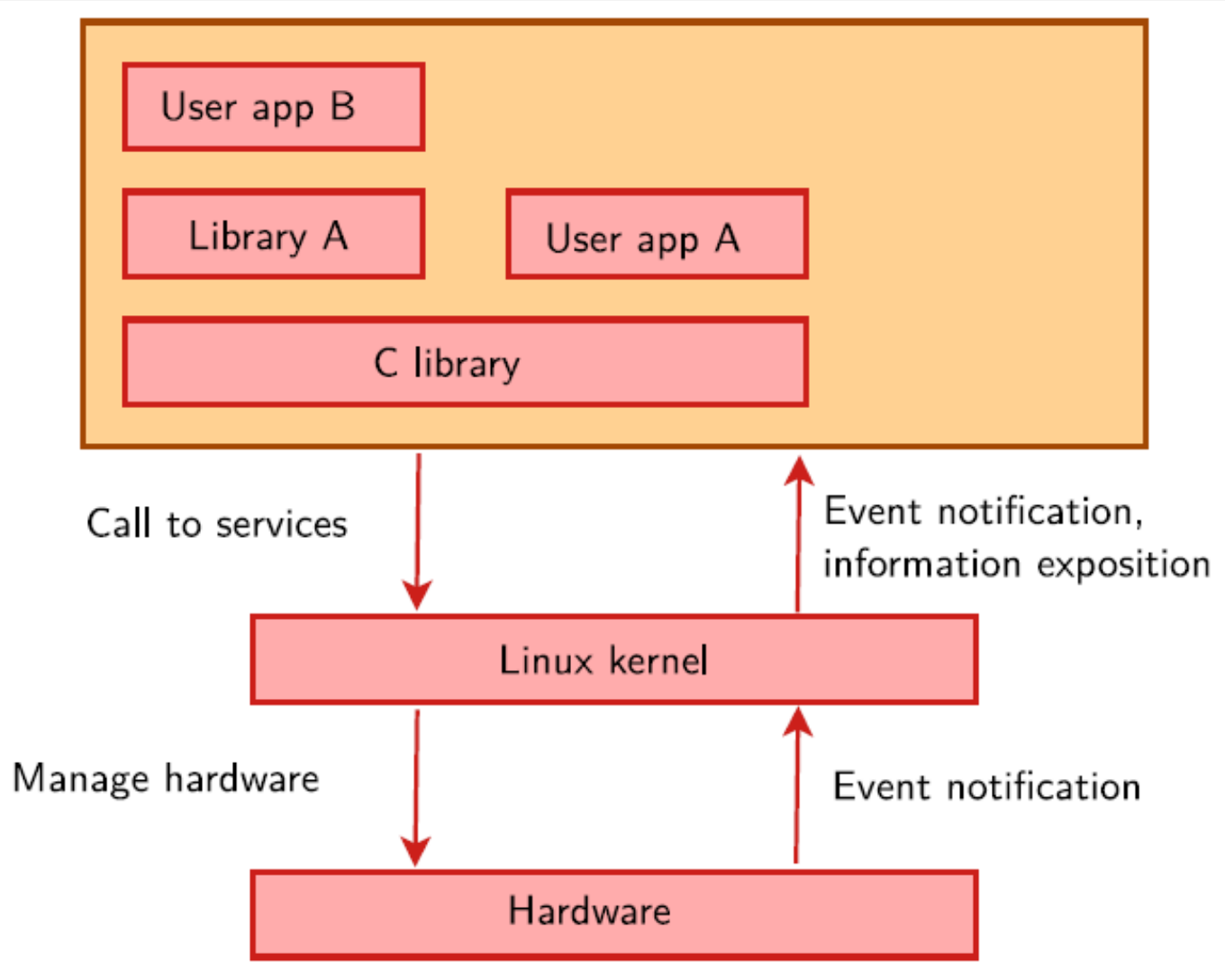
Linux Kernel Key Features

- Portability and hardware support. Runs on most architectures.
- Scalability. Can run on super computers as well as on tiny devices (4 MB of RAM is enough).
- Compliance to standards and interoperability
- Exhaustive networking support.
- Security. It can’t hide its flaws. Its code is reviewed by many experts.
- Stability and reliability.
- Modularity. Can include only what a system needs even at run time.
- Easy to program. You can learn from existing code. Many useful resources on the net.

Supported Hardware Architectures

- See the arch/ directory in the kernel sources
  - Minimum: 32 bit processors, with or without MMU, and gcc support
  - 32 bit architectures (arch/ subdirectories)
    - Examples: arm, arc, c6x, m68k, microblaze...
  - 64 bit architectures:
    - Examples: alpha, arm64, ia64...
  - 32/64 bit architectures
    - Examples: mips, powerpc, riscv, sh, sparc, x86...
  - Note that unmaintained architectures can also be removed when they have compiling issues and nobody fixes them.
  - Find details in kernel sources: arch/<arch>/Kconfig, arch/<arch>/README, or Documentation/<arch>/

Linux Kernel in the System



Linux Kernel Main Roles

- 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.

System Calls

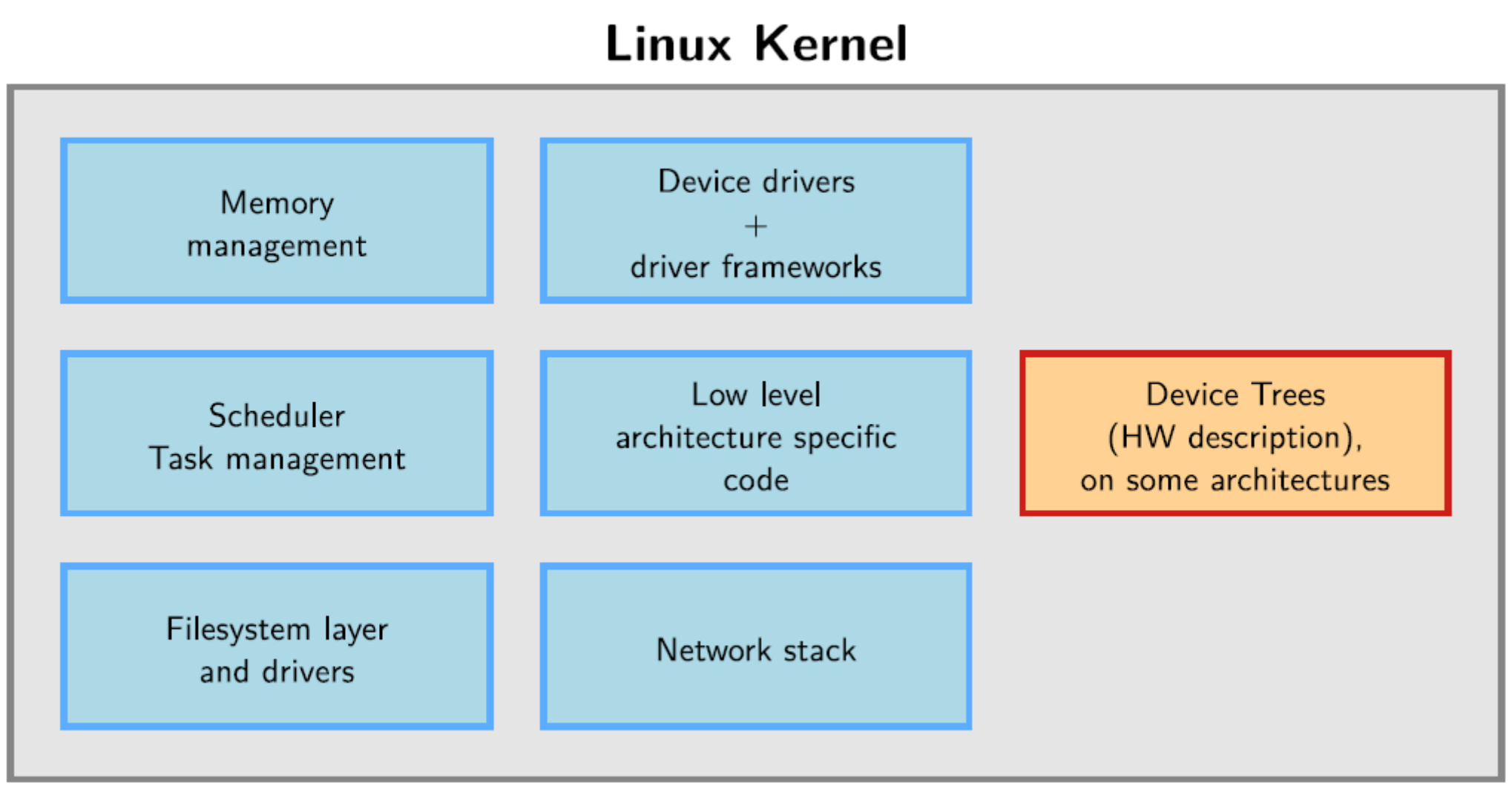
- The main interface between the kernel and user space is the set of system calls
- About 400 system calls that provide the main kernel Services
  - File and device operations, networking operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, etc.
- This interface is stable over time: only new system calls can be added by the kernel developers
- This system call interface is wrapped by the C library, and user space applications usually never make a system call directly but rather use the corresponding C library function

Image credits (Wikipedia):
https://bit.ly/2U2rdGB

Pseudo Filesystems

- Linux makes system and kernel information available in user space through pseudo filesystems, sometimes also called virtual filesystems
- Pseudo 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 set of devices and buses. Information about these devices.

Inside the Linux Kernel



Programming Language

- Implemented in C like all UNIX systems. (C was created to implement the first UNIX systems)
- A little Assembly is used too:
  - CPU and machine initialization, exceptions
  - Critical library routines.
- No C++ used, see http://vger.kernel.org/lkml/#s15-3
- All the code compiled with gcc
  - Many gcc specific extensions used in the kernel code, any ANSI C compiler will not compile the kernel
  - See https://gcc.gnu.org/onlinedocs/gcc-6.2.0/gcc/C-Extensions.html
- Ongoing work to compile the kernel with the LLVM C compiler too.

No C library

- The kernel has to be standalone and can’t use user space code.
- Architectural reason: user space is implemented on top of kernel services, not the opposite.
- Technical reason: the kernel is on its own during the boot up phase, before it has accessed a root filesystem.
- Hence, kernel code has to supply its own library implementations (string utilities, cryptography, uncompression...)
- So, you can’t use standard C library functions in kernel code. (printf(), memset(), malloc(),...).
- Fortunately, the kernel provides similar C functions for your convenience, like printk(), memset(), kmalloc(), ...

Portability

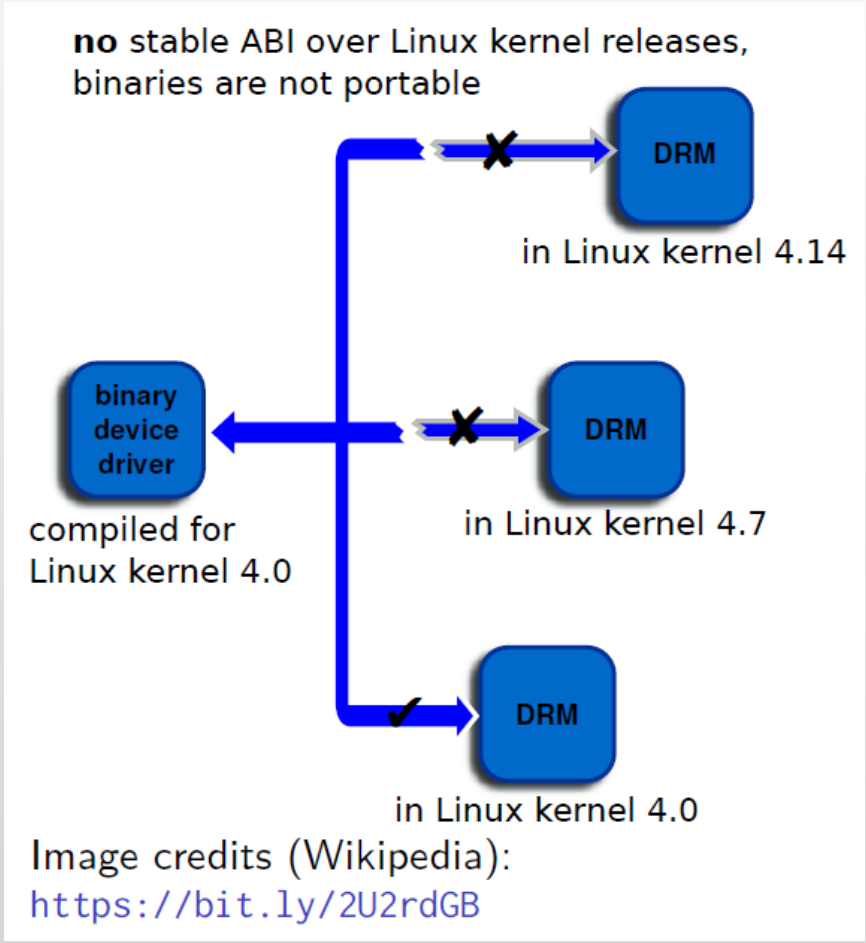
- The Linux kernel code is designed to be portable
- All code outside arch/ should be portable
- To this aim, the kernel provides macros and functions to abstract the architecture specific details
  - Endianness
    - cpu_to_be32()
    - cpu_to_le32()
    - be32_to_cpu()
    - le32_to_cpu()
  - I/O memory access
  - Memory barriers to provide ordering guarantees if needed
  - DMA API to flush and invalidate caches if needed
- Never use floating point numbers in kernel code. Your code may need to run on a low-end processor without a floating point unit.

Kernel Memory Constraints

- No memory protection
- The kernel doesn't try to recover from attempts to access illegal memory locations. It just dumps *oops* messages on the system console.
- Fixed size stack (8 or 4 KB). Unlike in user space, no mechanism was implemented to make it grow.
- Swapping is not implemented for kernel memory either.

No Stable Linux Internal API

- The internal kernel API to implement kernel code can undergo changes between two releases.
- In-tree drivers are updated by the developer proposing the API change: works great for mainline code.
- An out-of-tree driver compiled for a given version may no longer compile or work on a more recent one.
- See process/stable-api-nonsense in kernel sources for reasons why.
- Of course, the kernel to user space API does not change (system calls, /proc, /sys), as it would break existing programs.



application binary interface (ABI)

Benefit of Writing Driver in Linux Kernel

- What is in-tree? There is also another term out-of-tree
  - All modules start out as "out-of-tree" developments, that can be compiled using the context of a source-tree.
  - Once a module gets accepted to be included, it becomes an in-tree module.
- Once your sources are accepted in the mainline tree...
  - There are many more people reviewing your code, allowing to get cost-free security fixes and improvements.
  - You can also get changes from people modifying internal kernel APIs.
  - Accessing your code is easier for users.
  - You can get contributions from your own customers.
- This will for sure reduce your maintenance and support work

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.
  - There is no need to leverage an existing kernel subsystem such as the networking stack or filesystems.
  - There is no need for the kernel to act as a “multiplexer” for the device: only one application accesses the device.
- 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

User Space Device Drivers

- Certain classes of devices (printers, scanners, 2D/3D graphics acceleration) are typically handled partly in kernel space, partly in user space.
- Drawbacks
  - Less straightforward to handle interrupts.
  - Increased interrupt latency vs. kernel code.
- Advantages
  - No need for kernel coding skills. Easier to reuse code between devices.
  - Drivers can be written in any language and can be kept proprietary.
  - Driver code can be killed and debugged. Cannot crash the kernel.
  - Can be swapped out (kernel code cannot be).
  - Can use floating-point computation.
  - Less in-kernel complexity.
  - Potentially higher performance, especially for memory-mapped devices, thanks to the avoidance of system calls.

Time to Look at the Source Structure

- 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 architectures
- block/
  - Block layer core
- certs/
  - Management of certificates for key signing
- COPYING
  - Linux copying conditions (GNU GPL)
- CREDITS
  - Linux main contributors

Time to Look at the Source Structure

- crypto/
  - Cryptographic libraries
- Documentation/
  - Kernel documentation sources. Also available on https://kernel.org/doc/ (includes functions prototypes and comments extracted from source code).
- drivers/
  - All device drivers except sound ones (usb, pci...)
- fs/
  - Filesystems (fs/ext4/, etc.)
- include/
  - Kernel headers
- include/linux/
  - Linux kernel core headers

Time to Look at the Source Structure

- include/uapi/
  - User space API headers
- init/
  - Linux initialization (including init/main.c)
- ipc/
  - Code used for 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...)

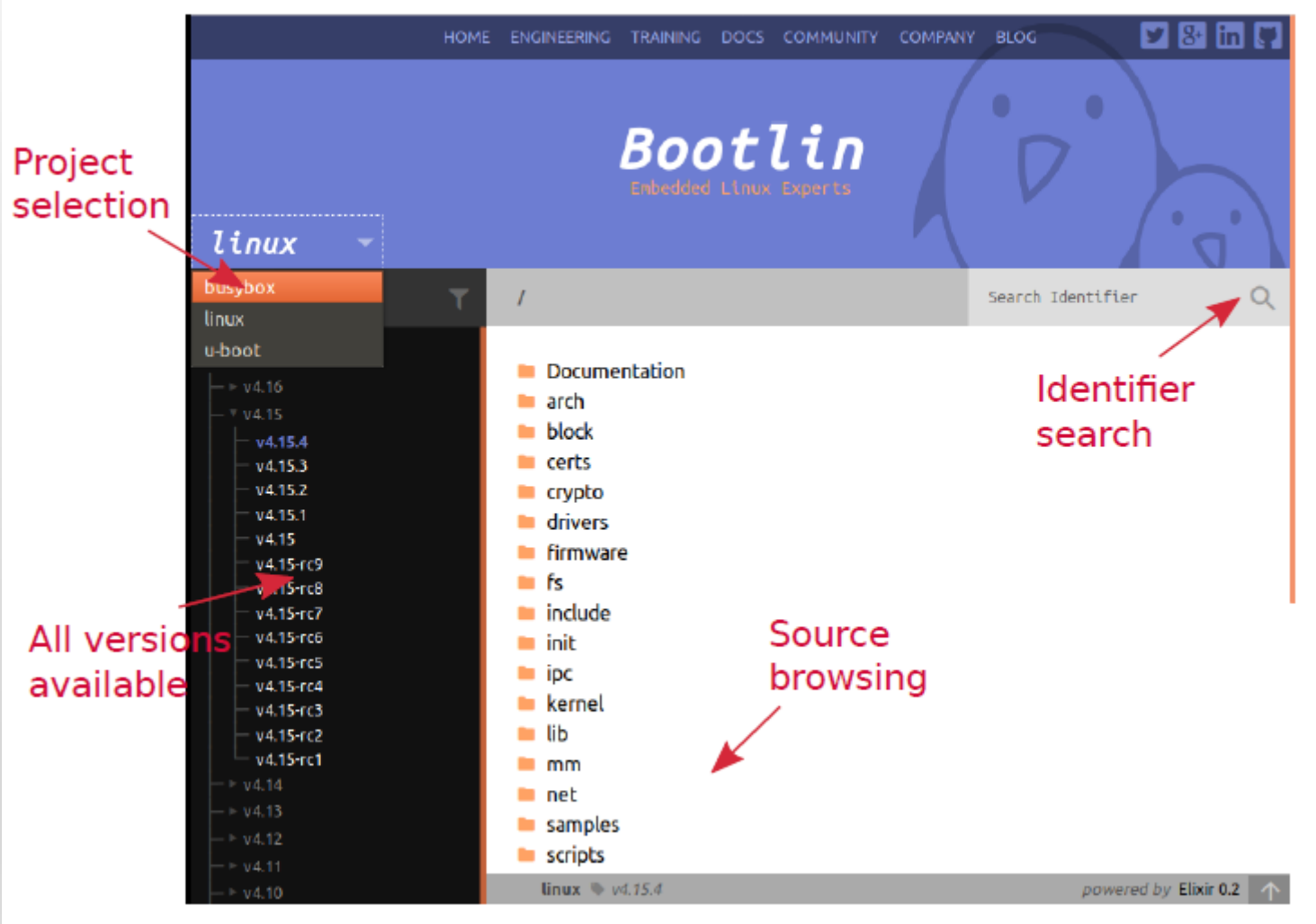
Time to Look at the Source Structure

- MAINTAINERS
  - Maintainers of each kernel part. Very useful!
- Makefile
  - Top Linux Makefile (sets version information)
- mm/
  - Memory management code (small too!)
- net/
  - Network support code (not drivers)
- README
  - Overview and building instructions
- samples/
  - Sample code (markers, kprobes, kobjects, bpf...)

Time to Look at the Source Structure

- scripts/
  - Executables for internal or external use
- security/
  - Security model implementations (SELinux...)
- 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)

http://elixir.bootlin.com/



Let's see the Booting of Linux Kernel

- Device Tree need to be provided
- Many embedded architectures have a lot of non-discoverable hardware (serial,Ethernet, I2C, Nand flash, USB controllers...)
- Depending on the architecture, such hardware is either described in BIOS ACPItables (x86), using C code directly within the kernel, or using a special hardware description language in a *Device Tree*.
- 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.
  - See arch/arm/boot/dts/at91-sama5d3_xplained.dts for example.
- The bootloader must load both the kernel image and the Device Tree Blob in memory before starting the kernel.

Booting with U-Boot

- The Universal Boot Loader
  - Can boot the zImage binary.
  - zImage is the outcome after you compiled Linux Kernel Srouce
- 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 (covered later)
  - console= for the destination of kernel messages
  - Example: console=ttyS0 root=/dev/mmcblk0p2 rootwait
  - Many more exist. The most important ones are documented in admin-guide/kernel-parameters in kernel documentation.
- 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.

Kernel Modules

- Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...
- Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).
- Also useful to reduce boot time: you don’t spend time initializing devices and kernel features that you only need later.
- Caution: once loaded, have full control and privileges in the system. No particular protection. That’s why only the root user can load and unload modules.
- 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 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 (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 Utilities

- <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 give 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

- sudo modprobe <module_name>
  - Most common usage of modprobe: tries to load all the modules the given module 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!

Let’s see the Booting of Linux Kernel

- 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

Passing Parameters to Modules

- 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

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");
```

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 the kernel, or when module unloading support is not enabled.
- Code of this example module available on https://frama.link/Q3CNXnom

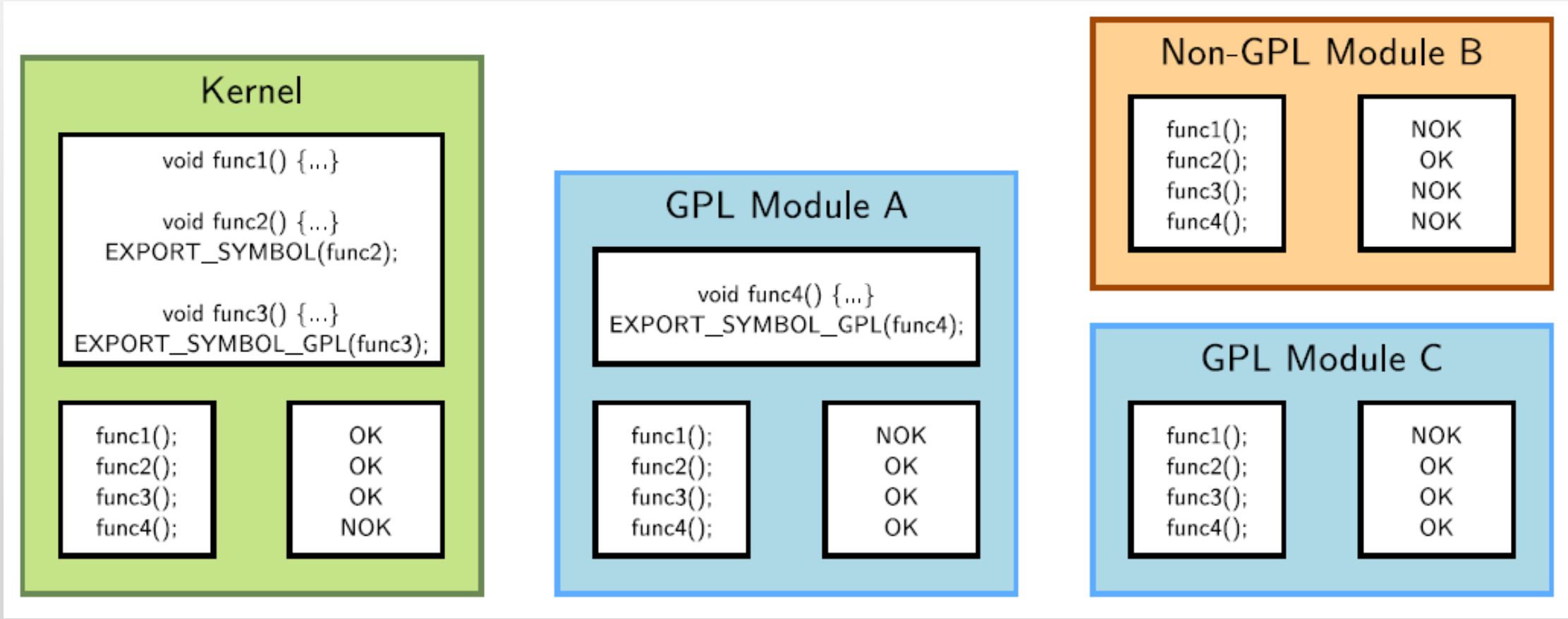
Hello Module Explanations

- 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
  - Linux 5.3: contains the same number of symbols with EXPORT_SYMBOL() and symbols with EXPORT_SYMBOL_GPL()
- A normal driver should not need any non-exported function.

Symbols Exported to Modules



Compiling a Module

- Two solutions
- Out of tree
  - When the code is outside of the kernel source tree, in a different directory
  - Advantage: Might be easier to handle than modifications to the kernel itself
  - Drawbacks: Not integrated to the kernel configuration/compilation process, needs to be built separately, the driver cannot be built statically
- 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

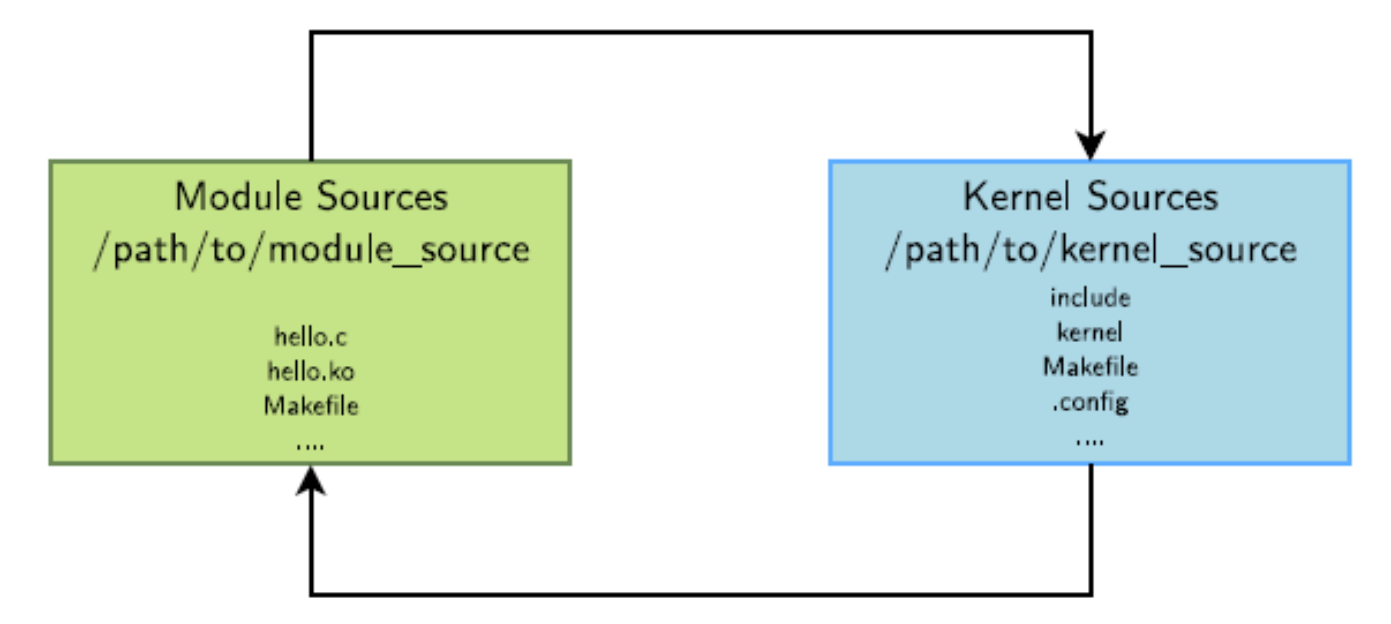
- The below Makefile should be reusable for any single-file out-of-tree Linux module
- The source file is hello.c
- Just run make to build the hello.ko file

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

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

- KDIR: kernel source or headers directory (see next slides)

Compiling an out-of-tree Module

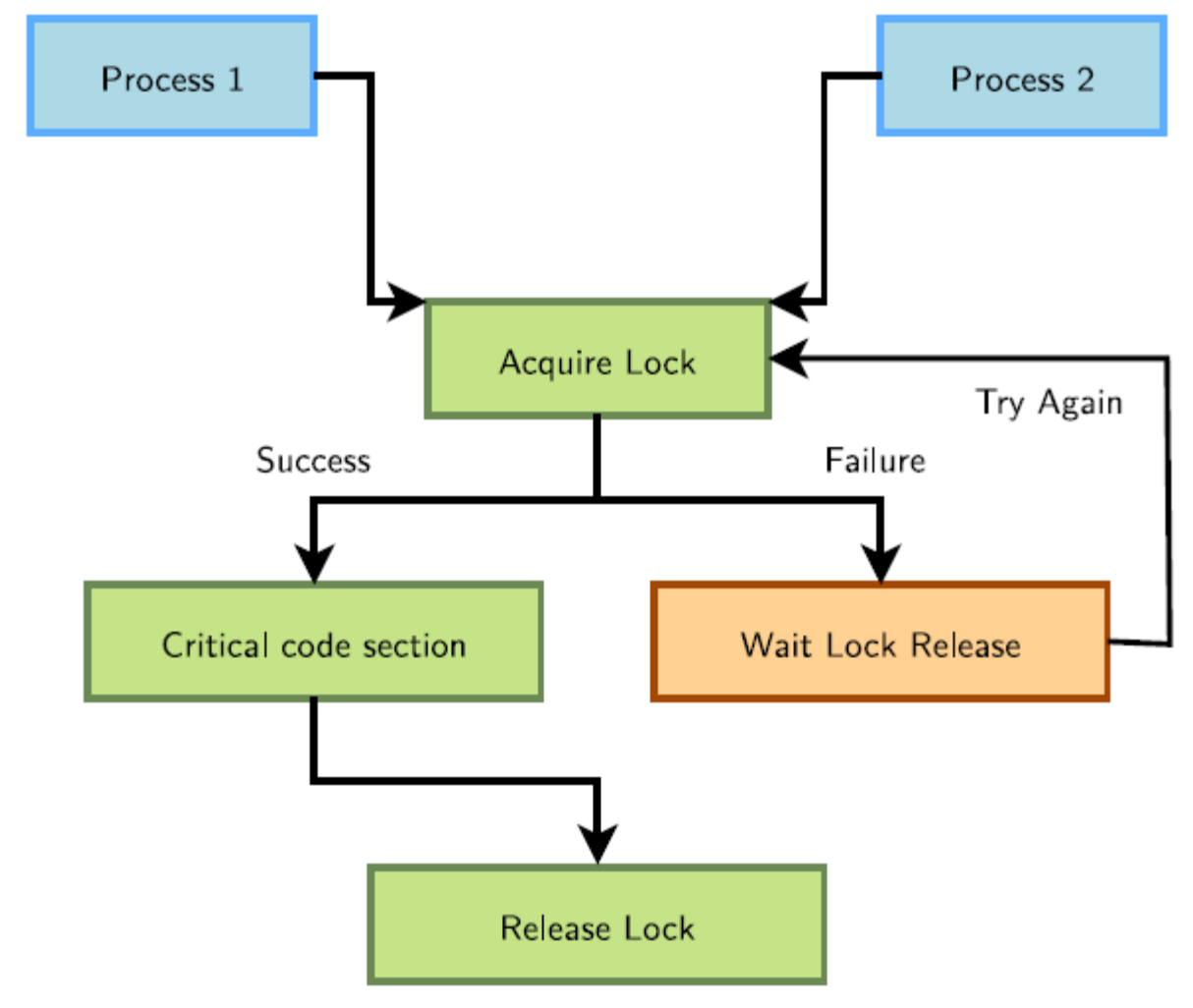


- The module Makefile is interpreted with KERNELRELEASE undefined, so it calls the kernel Makefile, passing the module directory in the M variable
- The kernel Makefile knows how to compile a module, and thanks to the M variable, knows where the Makefile for our module is. This module Makefile is then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

Sources of Concurrency Issues

- In terms of concurrency, the kernel has the same constraint as a multi-threaded program: its state is global and visible in all executions contexts
- 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.

Concurrency Protection with Locks



Linux mutexes

- mutex = **mut**ual **ex**clusion
- 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. Mutexescan 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.
- Caution: can’t be interrupted, resulting in processes you cannot kill!

```
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. Test the return value!!!

```
int mutex_lock_interruptible(struct mutex *lock);
```

- Same, but can be interrupted by any signal.

Locking and Unlocking Mutexes

```
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);
```

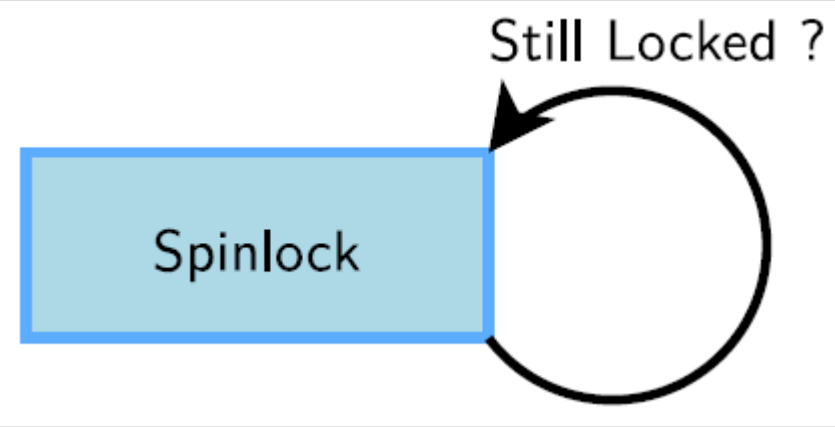
- Just tells whether the mutex is locked or not.

```
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!
- Originally intended for multiprocessor systems
- Spinlocks never sleep and keep spinning in a loop until the lock is available.
- Spinlocks cause kernel preemption to be disabled on the CPU executing them.
- 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 Spinlocks

- Several variants, depending on where the spinlock is called:

```
void spin_lock(spinlock_t *lock);
void spin_unlock(spinlock_t *lock);
```

  - Doesn't disable interrupts. Used for locking in process context (critical sections in which you do not want to sleep).

```
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, to prevent preemption by interrupts

Using Spinlocks

```
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.

- Note that reader / writer spinlocks also exist, allowing for multiple simultaneous readers.

Spinlock example

- From drivers /tty/serial/uartlite.c
- Spinlock structure embedded into struct uart_port

```
struct uart_port {
        spinlock_t lock;
        /* Other fields */
};
```

- Spinlock taken/released with protection against interrupts

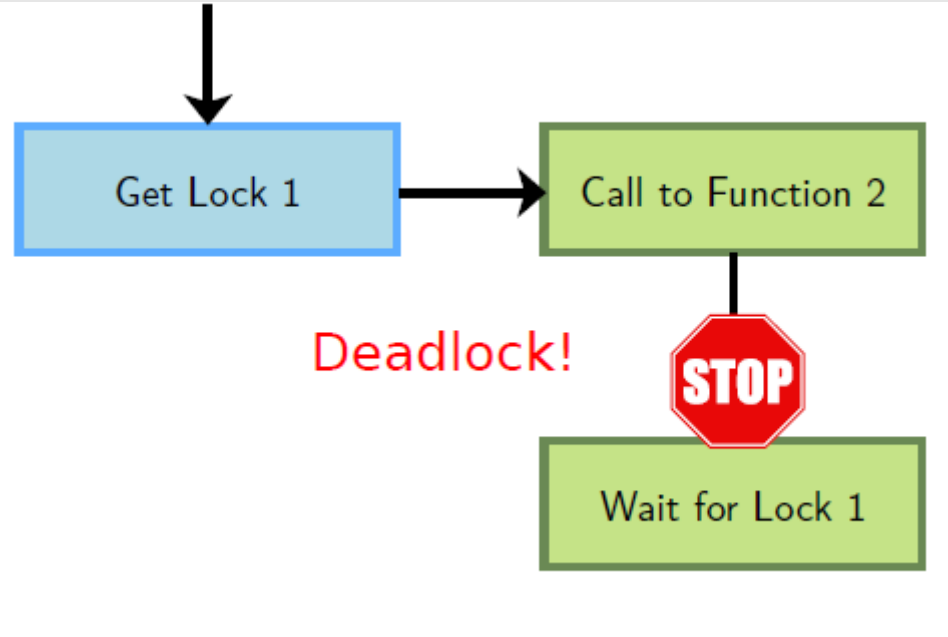
```
static unsigned int ulite_tx_empty
    (struct uart_port *port) {
        unsigned long flags;

        spin_lock_irqsave(&port->lock, flags);
        /* Do something */
        spin_unlock_irqrestore(&port->lock, flags);
}
```

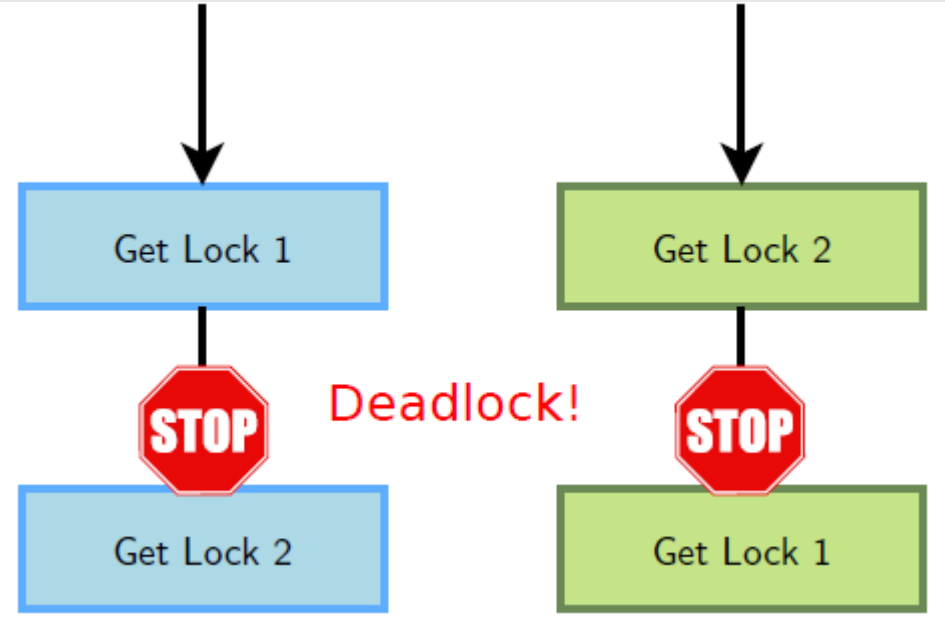
Deadlock Situations

- They can lock up your system. Make sure they never happen!

- 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!



Alternatives to Locking

- 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 (See https://en.wikipedia.org/wiki/RCU).
  - When available, use atomic operations.

Atomic Variables

- 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
  - Contains a signed integer (at least 24 bits)
- 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 Variables

- Similar functions testing the result:

```
int atomic_inc_and_test(...);
int atomic_dec_and_test(...);
int atomic_sub_and_test(...);
```

- Functions returning the new value:

```
int atomic_inc_return(...);
int atomic_dec_return(...);
int atomic_add_return(...);
int atomic_sub_return(...);
```

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 and References

- 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

- See kernel-hacking/locking in kernel documentation for many details about kernel locking mechanisms.

That's all for today.