

Embedded System

Lecture 00: Course Information

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

Hong-Yue Technology Research Building 334
R 09:10 - 12:00

Course Information

- Embedded System
 - Full English
 - Designed for students in graduate school
- Course Time and Place
 - Lecture (3 Hours)
 - Thursday 09:10 - 12:00
 - Hong-Yue Technology Research Building 334e
- Course Website

Course Instructor & Teaching Assistant

- Course Instructor

- Prof. Shuo-Han Chen (陳碩漢)

- Office: 宏裕科技大樓 1532

- Office Hours: Monday 11:00 - 15:00

- Email: shchen@csie.ntut.edu.tw

- Teaching Assistant

- 吳承岳

- Office: 宏裕科技大樓 438

- Office Hours: Tuesday 09:00 – 12:00

- Email: t109598085@ntut.edu.tw



Prerequisites

- Addend & Interact
 - Comfortable with C
 - Already taken the Operating System Course
 - Not afraid of English
-
- And, of course, willing to learn more about Embedded System

Course Assessment

- **Assignment: 40 %**
 - Weekly or Biweekly
 - Upload to following gitlab:
 - You're requested to register and do the exercise before Next Thursday !
- **Midterm Exam: 40 %**
- **Survey Paper Presentation: 20 %**
 - Two people a group & each group will be requested to read a 12-page papers
 - Form a 30 minutes English presentation (15 minutes for each member)
 - You are required to meet with me in advance to practice
 - Your points is determined on how much efforts you put into this presentation
 - Not how fluently you can speak in English

Course Schedule

Full Online Course: To practice taking online course,
Required by the School (due to Covid-19)

W	Date	Lecture	Notes	Homework
1	Sept. 17	Lec01: Introduction		HW00
2	Sept. 24	Lec02: The Big Picture		
3	Oct. 1	Moon Festival	No Class	
4	Oct. 8	Lec04: Linux and Real time		
5	Oct. 15	Lec05: The Linux Kernel		
6	Oct. 22	Lec06: Kernel Arch for Device Drivers & Kernel Initialization		
7	Oct. 29	Lec07: Driver Allocation, Input Subsystem & Memory		
8	Nov. 5	Lec08: File System and Flash storage	Full Online Course	
9	Nov. 12	Lec09: Bootloaders		
10	Nov. 19	Lec10: Bootloaders	Full Online Course	
11	Nov. 26	Midterm on Nov. 26	3 Hour Paper-and-pencil Test (Slides & Notes Allowed)	
12	Dec. 3	Lec11: MTD System		
13	Dec. 10	Lec12: Busy box		
14	Dec. 17	Lec13: Device Driver Basics		
15	Dec. 24	Lec14: USB		
16	Dec. 31	Lec15: udev		
17	Jan. 7	Final Presentation	Full Online Course	
18	Jan. 14	Final Presentation	Full Online Course	

Big Question : What is Embedded System ?

- Should be mobile ?
- Should be small ?
- Should have battery ?
- Should be less powerful than desktop ?
- Should be inexpensive ?



- Actually, these are **not** deciding factors of embedded system

Big Question : What is embedded system ?

- A Combination of computer hardware, software, and input/output peripheral devices
 - Hardware = Motherboard, Processor, RAM, Storage
 - **Designed to perform a specific function.**
 - Very few people realize that a processor and software are involved in the preparation of their lunch or dinner !!!
 - General-purpose computer is not designed to perform a specific function.



Induction Heater

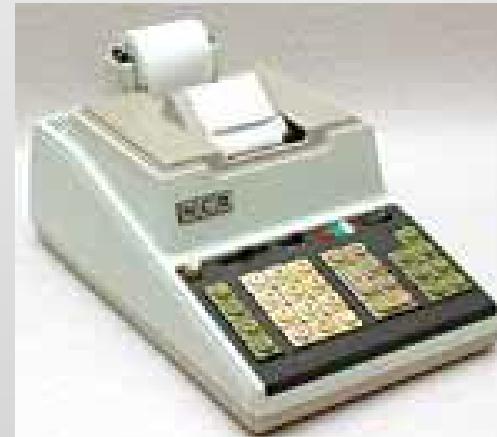
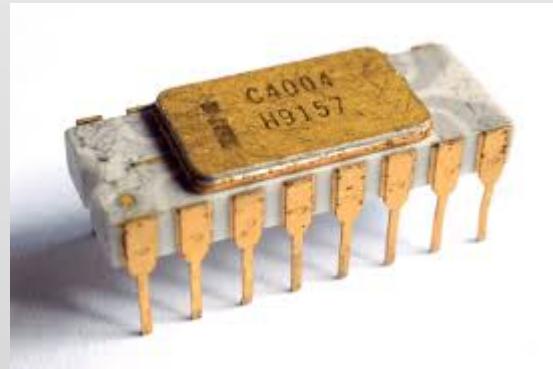


Big Question : What is embedded system ?

- Usually Installed in a larger system
- The existence of processor & software should be completely unnoticed by a device user
 - For example, cars and trucks contain many embedded systems
 - Anti-lock brake controller
 - Vehicle emission monitor and controller
 - Dashboard information display
 - Tesla has embedded system with AI chips for auto drive

Interesting Facts

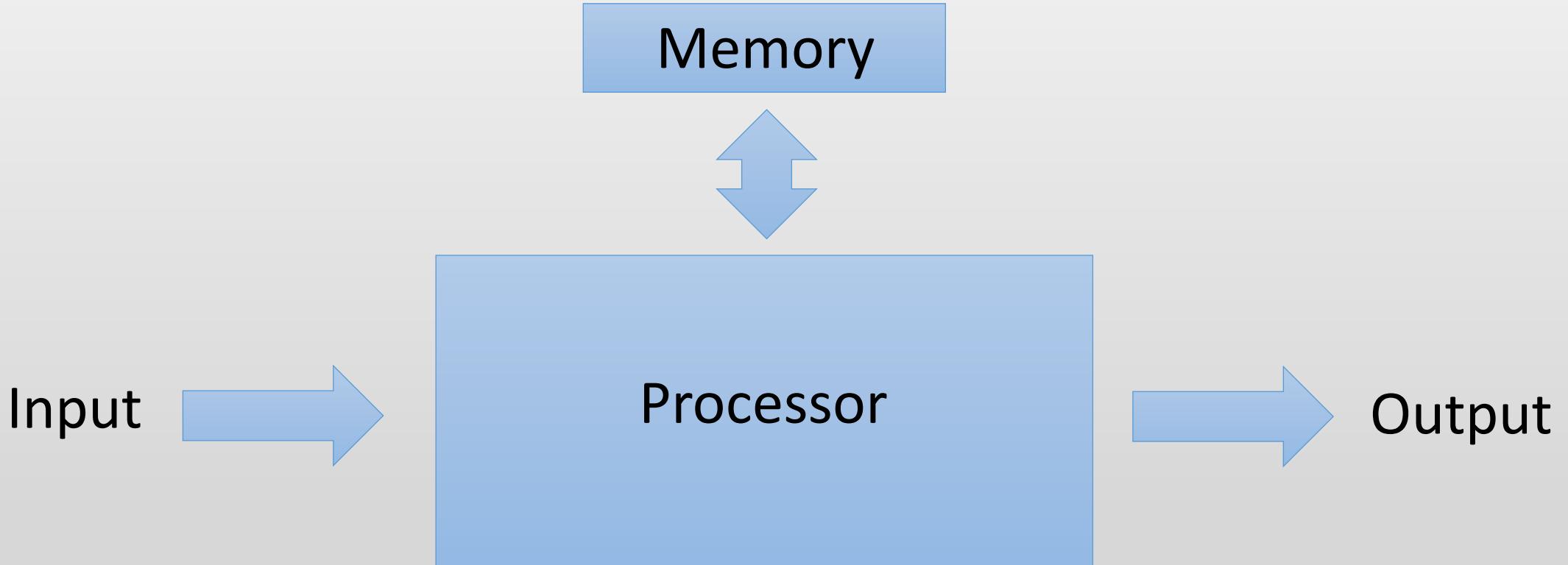
- When did embedded systems first appear?
 - Not before 1971, because that's when world's first microprocessor
 - Intel 4004, for use in Busicom 141-PF calculator of a Japan Company



- The microprocessor was an overnight success !
 - Increased use in the server future decade

Generic Embedded System

- Embedded systems will continue to increase



We are going to focus on Embedded Linux System

- First of all, what is Linux ?
 - Just like Windows, iOS, and Mac OS, Linux is an operating system
 - It's a software that manages all of the hardware resources
 - Contains of
 - Bootloader
 - Kernel
 - Init system
 - Daemons
 - Graphical server
 - Desktop environment
 - Applications
 - Very Popular and has many different Linux distribution



What's Embedded Linux ?

- Umbrella term:
 - Ad-hoc recipes for building Linux-based embedded systems
 - Very wide range of systems:
 - Very tiny (mmuless)
 - Smartphones (i.e. Android)
 - Cellular base stations
 - etc.
- An Embedded Linux system doesn't have any specific:
 - Kernel
 - User-space filesystem content
 - API
 - etc.

Why Linux ?

- Linux has been adopted for embedded products
 - Linux supports a vast variety of hardware devices, probably more than any other OS.
 - Linux supports a huge variety of applications and networking protocols.
 - Linux is scalable, from small consumer-oriented devices to large, heavy-iron, carrier-class switches and routers.
 - Linux can be deployed without the royalties required by traditional proprietary embedded operating systems.
 - Linux has attracted a huge number of active developers, enabling rapid support of new hardware architectures, platforms, and devices.
 - An increasing number of hardware and software vendors support Linux.

Why Linux and What are We Going to Learn ?

- It's open source and under GNU GPL (General Public License)
 - GPL states that “When we speak of free software, we are referring to freedom, not price.”
- Goals of this course
 - Linux and Real time
 - Architecture of Linux Kernel
 - Kernel & User Space Initialization
 - File System and Flash storage
 - MTD / Driver / USB



Q & A

Thank you for your attention.

Find your team member and report to TA before next class !

Embedded System

Lecture 01: The Big Picture

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

Hong-Yue Technology Research Building 334
R 09:10 - 12:00

Embedded or Not

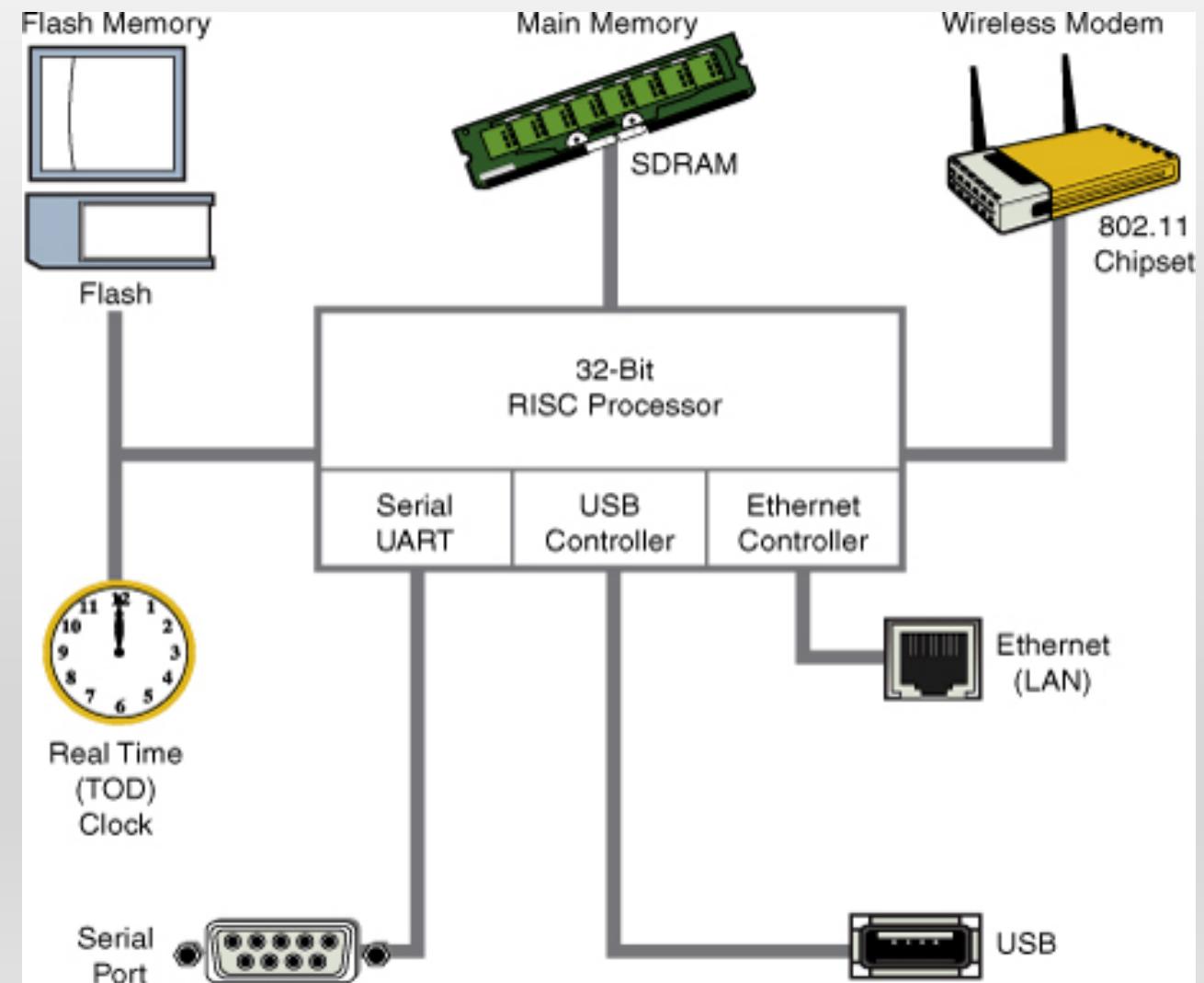
- Several key attributes are associated with embedded systems
- But you would think differently under different usage of a device
 - For example,
 - You wouldn't necessarily call your desktop PC an embedded system
 - But if a PC is used in a data center to perform a critical monitoring and alarm task
 - It fit the description we gave last week somehow
 - Designed for a specific task & Part of a bigger system
 - So, it's really hard to determine by the hardware itself
 - We should also look at what it's used for

Embedded or Not (Cont'd)

- Following are some of the usual characteristics of an embedded system
 1. Contains a processing engine, such as a general-purpose microprocessor
 2. Typically designed for a specific application or purpose
 3. Includes a simple (or no) user interface
 4. Often is resource-limited. For example, it might have a small memory footprint and no hard drive
 5. Might have power limitations, such as operating from batteries
 6. Not typically used as a general-purpose computing platform
 7. Generally has application software built in, not user-selected
 8. Ships with all intended application hardware and software pre-integrated
 9. Often is intended for applications without human intervention

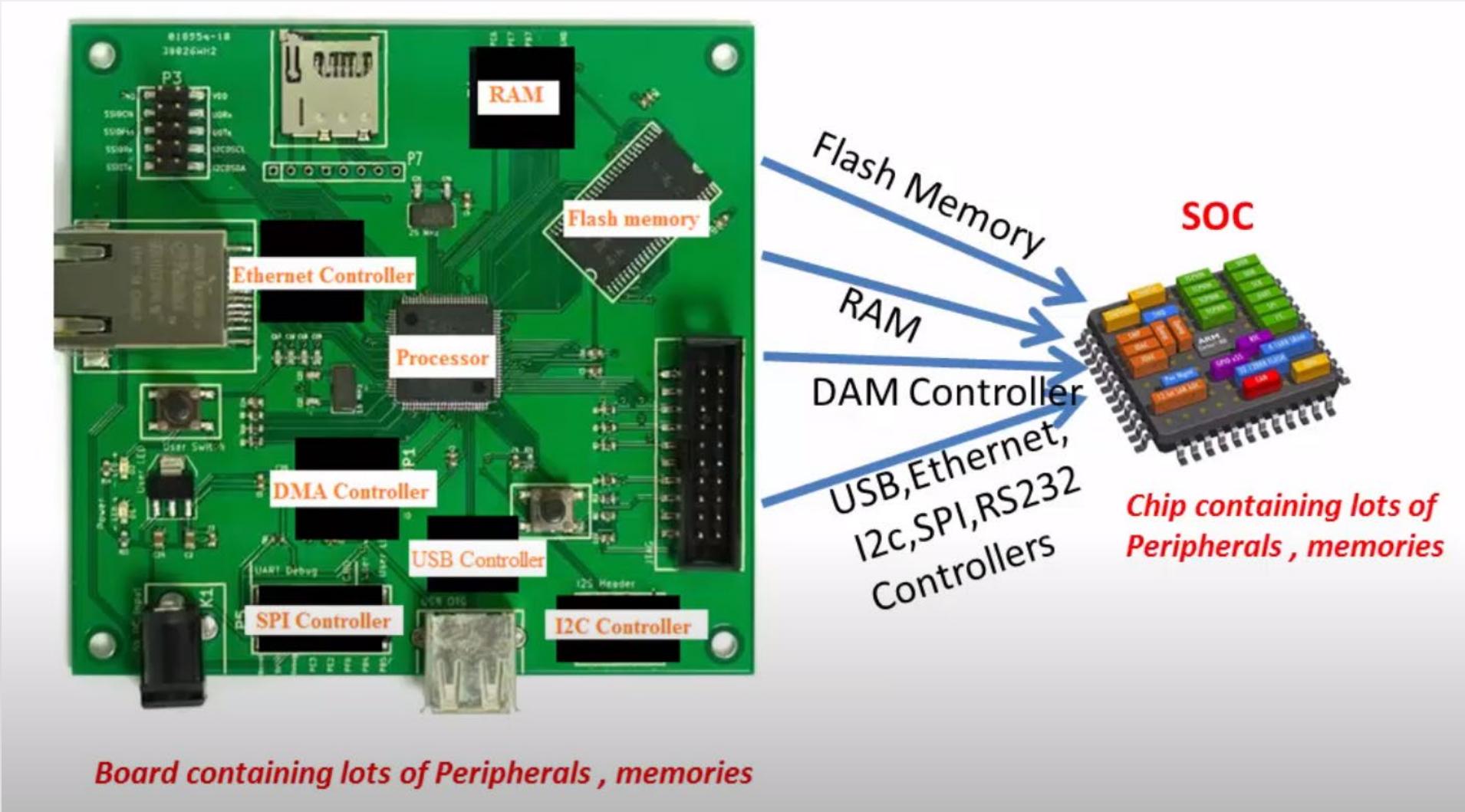
Anatomy of an Embedded System

- Example of typical embedded system
 - 32-bit RISC processor
 - Flash memory
 - SDRAM Main memory (a few megabytes to hundreds of megabytes)
 - Real-time clock module



Processor vs System-on-Chip

- Example

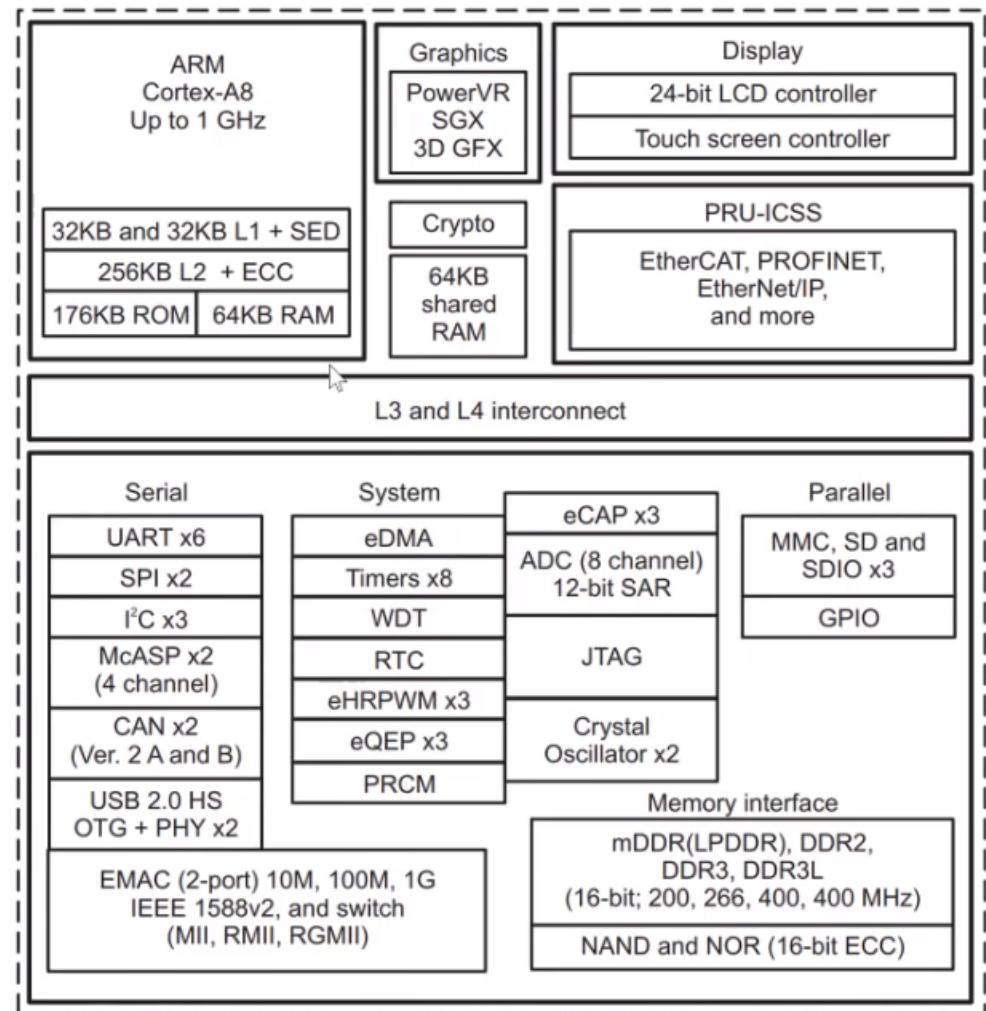


AM335x System-on-Chip

- System Architecture

1.4 Functional Block Diagram

Figure 1-1 shows the AM335x microprocessor functional block diagram.



Let's take about boot ROM

- Boot ROM, where the image is stored

2.1 ARM Cortex-A8 Memory Map

Table 2-1. L3 Memory Map

Block Name	Start_address (hex)	End_address (hex)	Size	Description
GPMC (External Memory)	0x0000_0000 ⁽¹⁾	0x1FFF_FFFF	512MB	8-/16-bit External Memory (Ex/R/W) ⁽²⁾
Reserved	0x2000_0000	0x3FFF_FFFF	512MB	Reserved
Boot ROM	0x4000_0000	0x4001_FFFF	128KB	
	0x4002_0000	0x4002_BFFF	48KB	32-bit Ex/R ⁽²⁾ – Public
Reserved	0x4002_C000	0x400F_FFFF	848KB	Reserved
Reserved	0x4010_0000	0x401F_FFFF	1MB	Reserved

Let's take about boot ROM

- Boot ROM, where the image is stored

2.1 ARM Cortex-A8 Memory Map

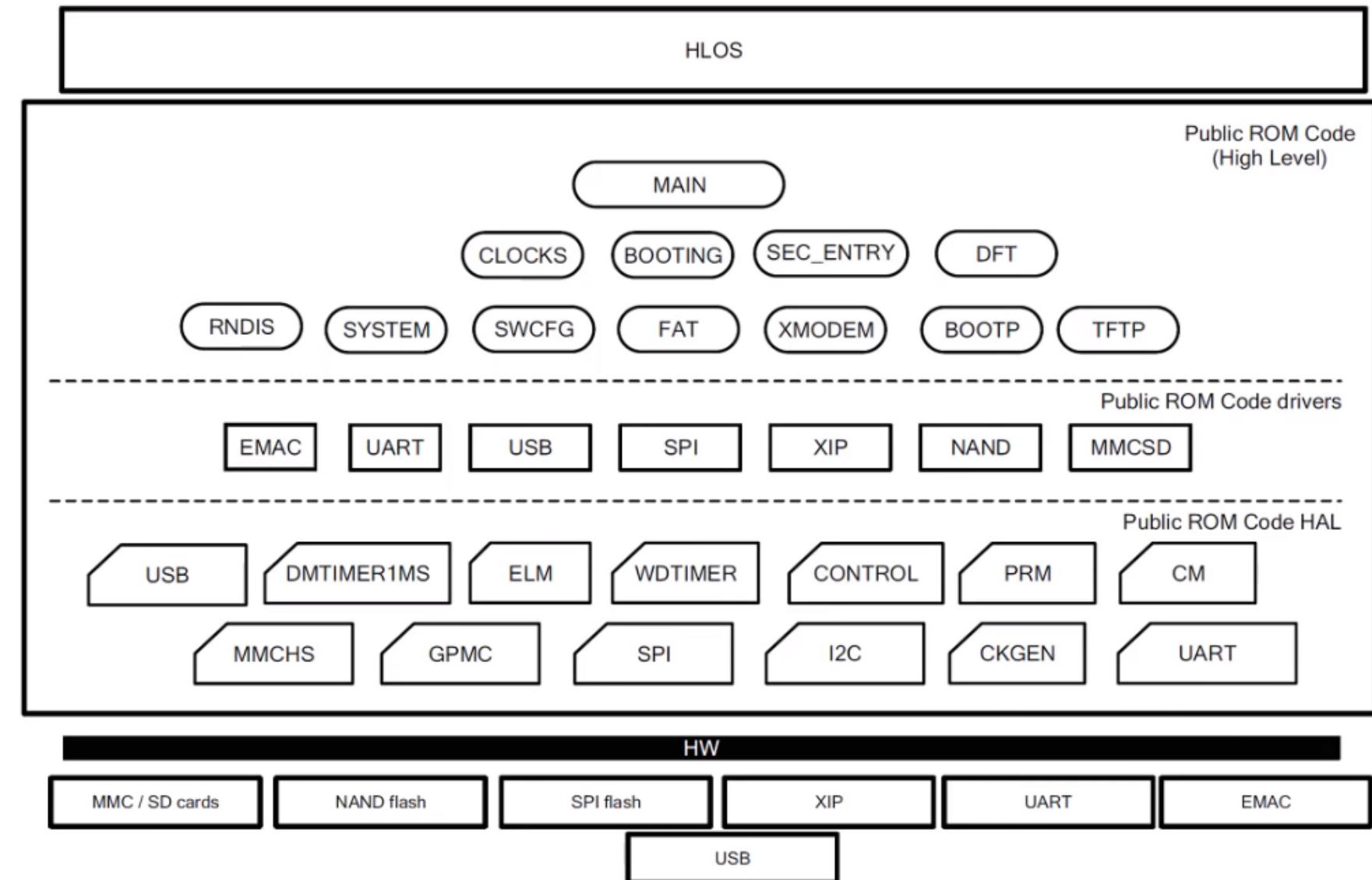
Table 2-1. L3 Memory Map

Block Name	Start_address (hex)	End_address (hex)	Size	Description
GPMC (External Memory)	0x0000_0000 ⁽¹⁾	0x1FFF_FFFF	512MB	8-/16-bit External Memory (Ex/R/W) ⁽²⁾
Reserved	0x2000_0000	0x3FFF_FFFF	512MB	Reserved
Boot ROM	0x4000_0000	0x4001_FFFF	128KB	
	0x4002_0000	0x4002_BFFF	48KB	32-bit Ex/R ⁽²⁾ – Public
Reserved	0x4002_C000	0x400F_FFFF	848KB	Reserved
Reserved	0x4010_0000	0x401F_FFFF	1MB	Reserved

Let's take about boot ROM

- Fairly complicated with only 178 KB

Figure 26-1. Public ROM Code Architecture



Bootloader

- ▶ In terms of booting process, no standardized *BIOS* or *firmware* like on x86 machines.
- ▶ Each ARM SoC comes with its own **ROM code** that implements a SoC-specific boot mechanism.
- ▶ The early stages of the boot process are therefore specific to each SoC.
- ▶ In general: capable of loading a small amount of code from non-volatile storage (NAND, MMC, USB) into a SRAM internal to the processor.
 - ▶ External DRAM not initialized yet.
- ▶ Often also provides a recovery method, to *unbrick* the platform. Over USB, serial or sometimes Ethernet.
- ▶ Used to load a *first stage* bootloader into SRAM, which will itself initialize the DRAM and load/run a *second stage* into DRAM.

Bootloader

- ▶ Grub(2) typically **not widely used** on ARM platforms
- ▶ **U-Boot**, the de-facto standard, found on most development boards and community platforms.
- ▶ **Barebox**, less widely used, but very interesting.
- ▶ Homemade bootloaders, especially when security/DRM are involved (phone, set-top boxes, etc.)
- ▶ Grub starts to gain some traction, especially on ARM64, for the server market
- ▶ RaspberryPi is a very special case, with some firmware executed on the GPU, and directly loading the Linux kernel.

Boot Sequence

B
O
O
T
L
O
A
D
E
R

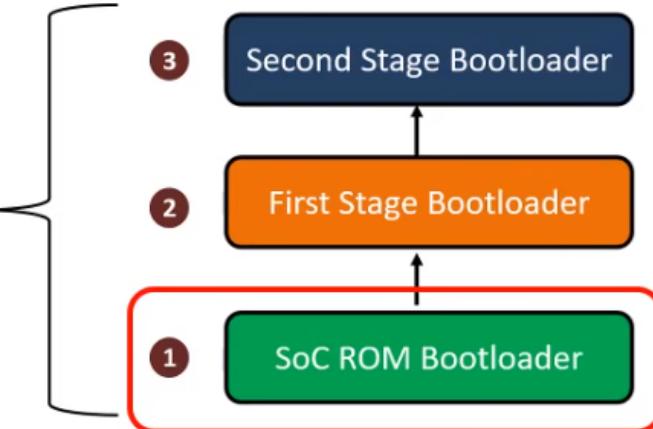
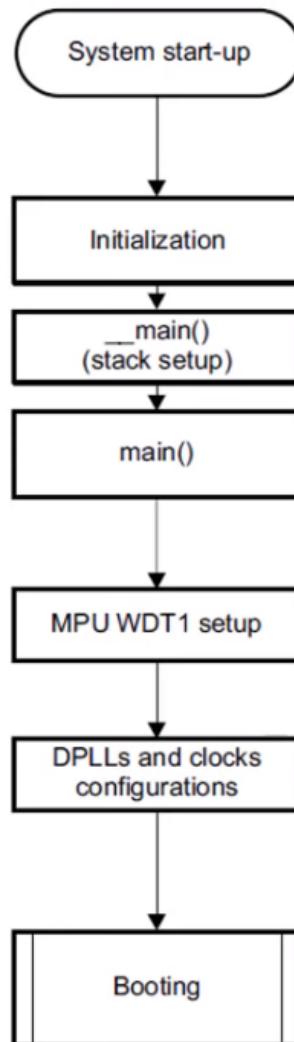


Figure 26-8. ROM Code Startup Sequence



- System initialization
- Sets up the Stack
- Sets up the Watchdog Timer
- Configures the system clock
- Starts the booting procedures

Boot Sequence

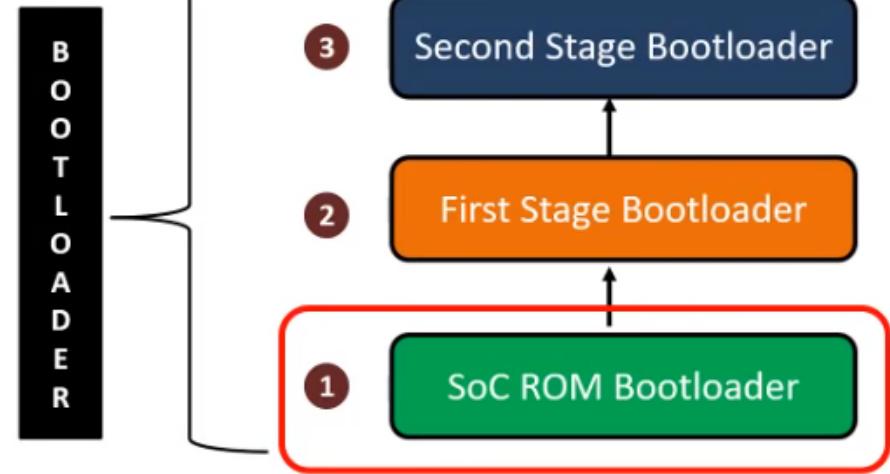
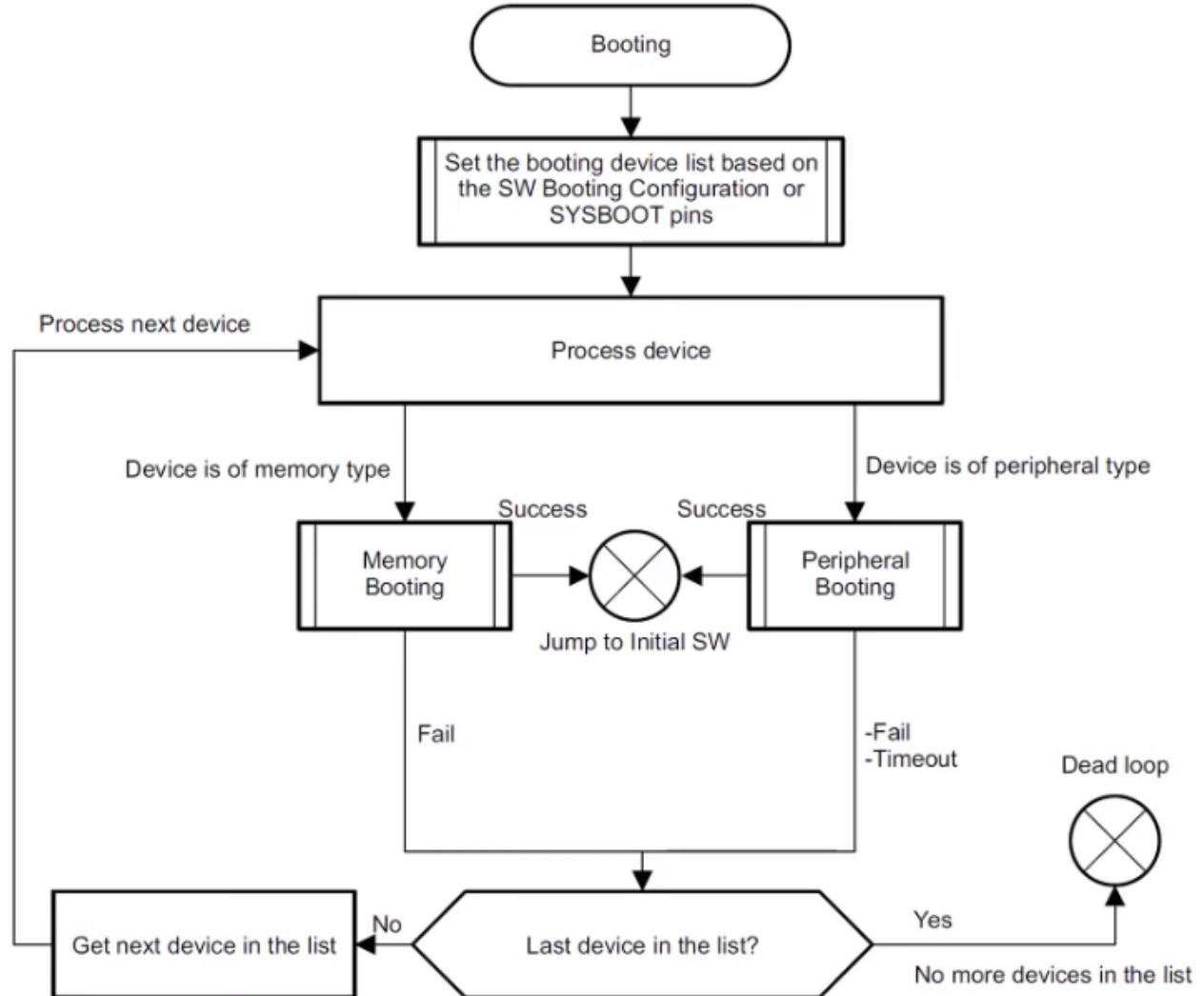


Figure 26-10. ROM Code Booting Procedure



Boot Sequence

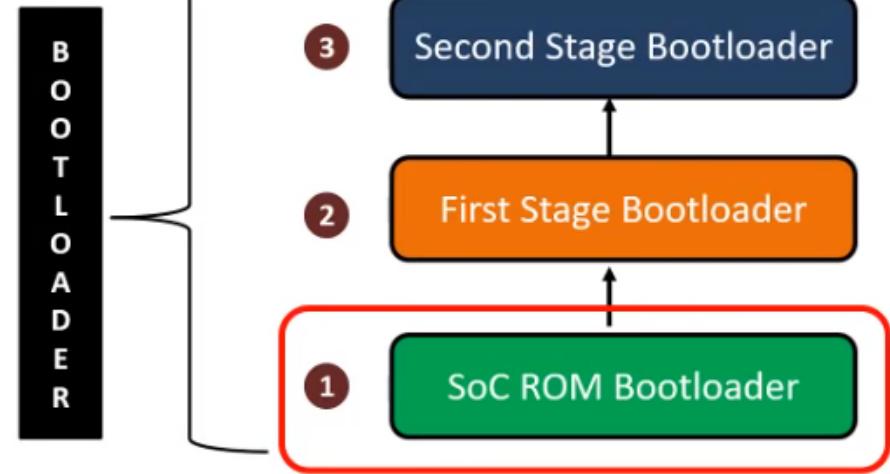
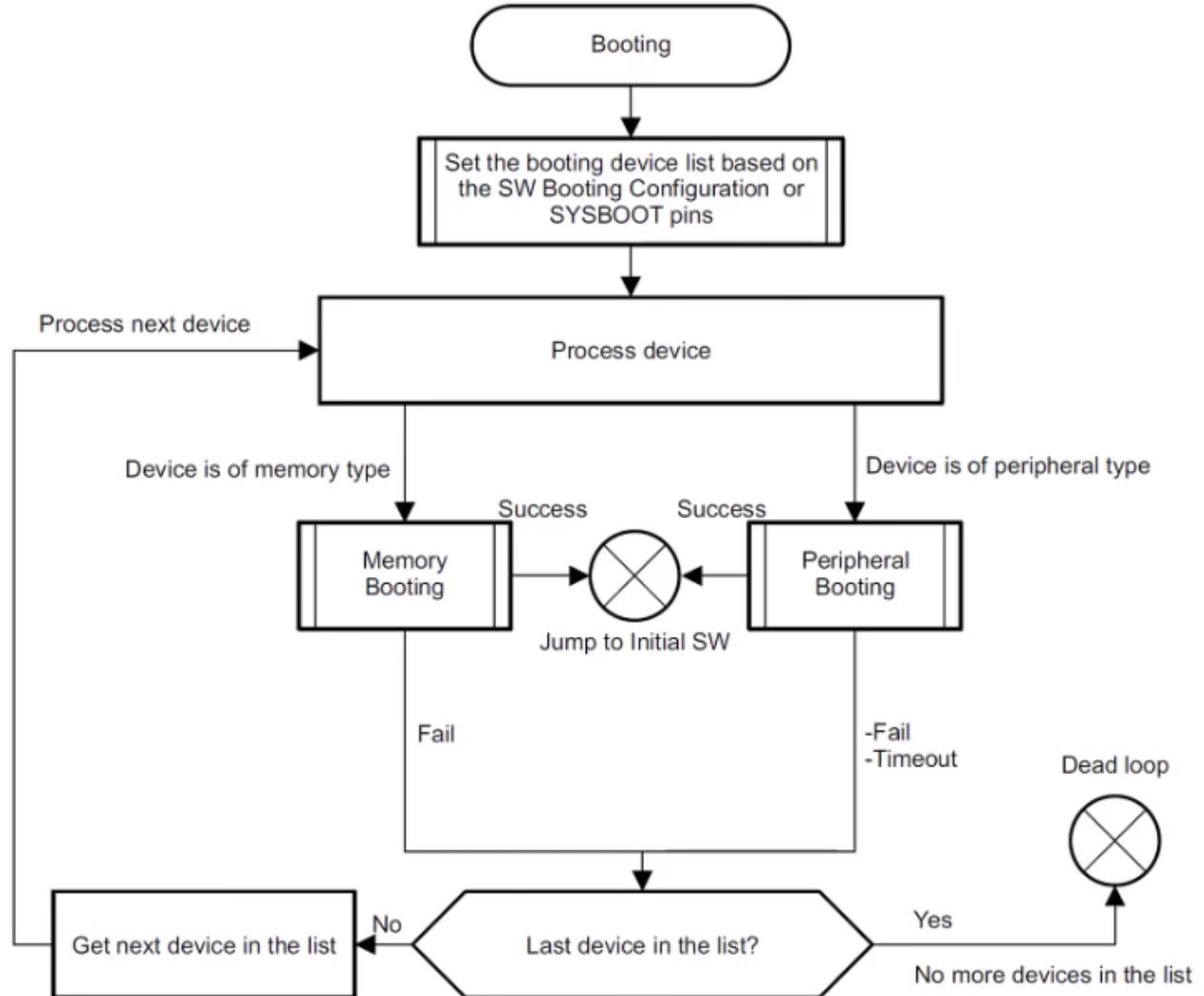
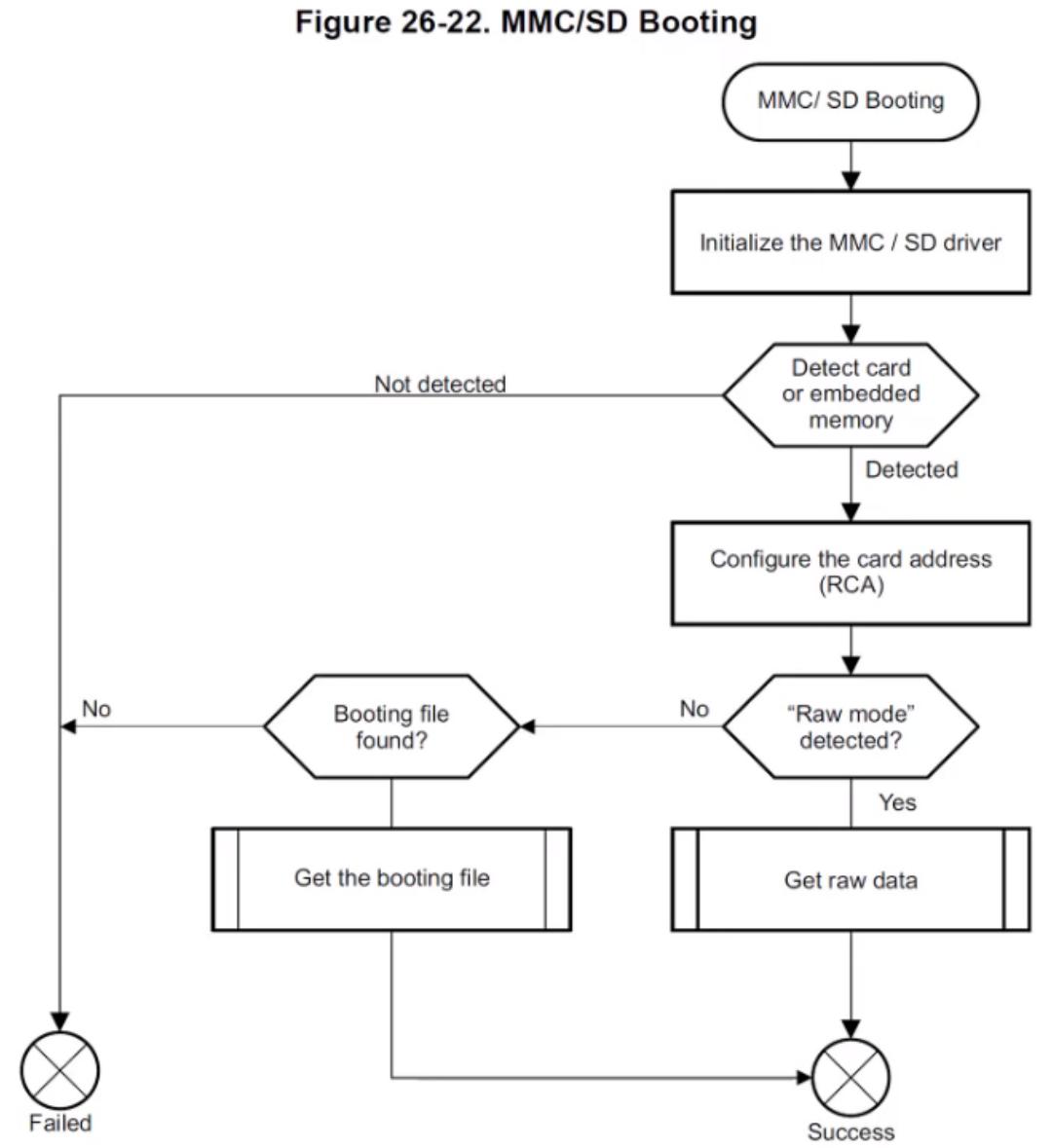
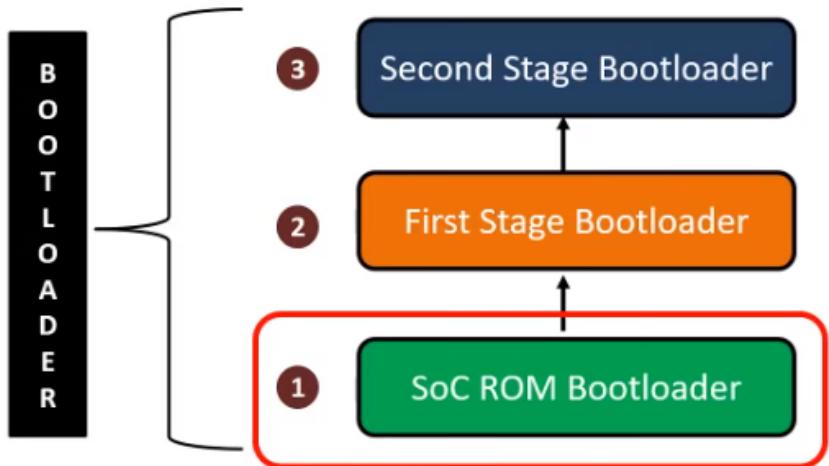


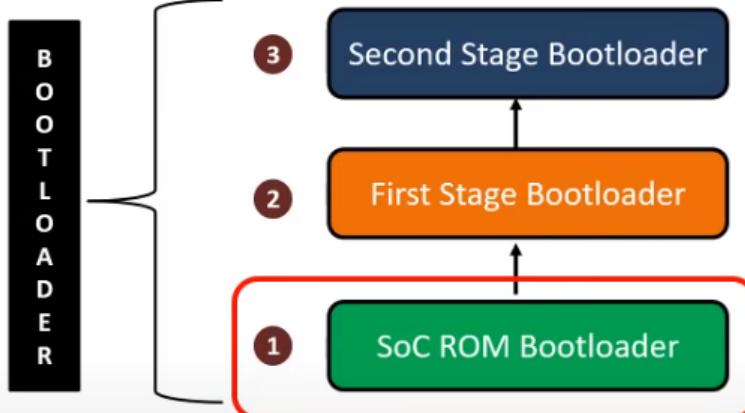
Figure 26-10. ROM Code Booting Procedure



Boot Sequence



Booting Sequence



26.1.8.5.5 MMC/SD Read Sector Procedure in Raw Mode

In raw mode the booting image can be located at one of the four consecutive locations in the main area: offset 0x0 / 0x20000 (128KB) / 0x40000 (256KB) / 0x60000 (384KB). For this reason, a booting image shall not exceed 128KB in size. However it is possible to flash a device with an image greater than 128KB starting at one of the aforementioned locations. Therefore the ROM Code does not check the image size. The only drawback is that the image will cross the subsequent image boundary.

The raw mode is detected by reading sectors #0 and #1024. The content of these sectors is then verified for presence of a TOC structure, as described in [Section 26.1.10](#). In the case of a GP Device, a Configuration Header (CH) must be located in the first sector, followed by a GP header (see [Section 26.1.10](#)). The CH might be void (only containing a CHSETTINGS item for which the valid field is zero).

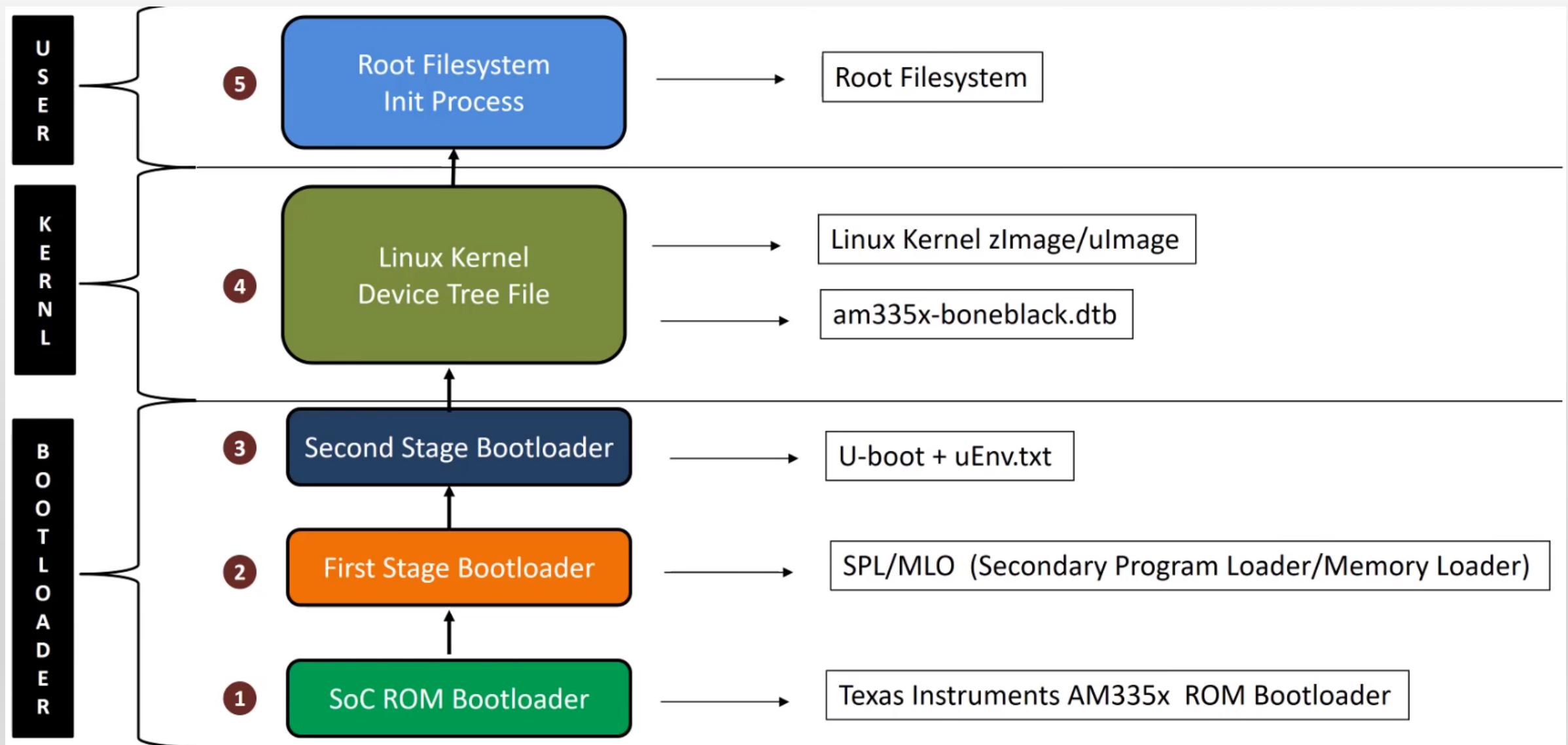
26.1.8.5.6 MMC/SD Read Sector Procedure in FAT Mode

MMC/SD Cards or eMMC/ eSD devices may hold a FAT file system which ROM Code is able to read and process. The image used by the booting procedure is taken from a specific booting file named “MLO”. This file has to be located in the root directory on an active primary partition of type FAT12/16 or FAT32.

An MMC/SD card or eMMC/ eSD device can be configured either as floppy-like or hard-drive-like.

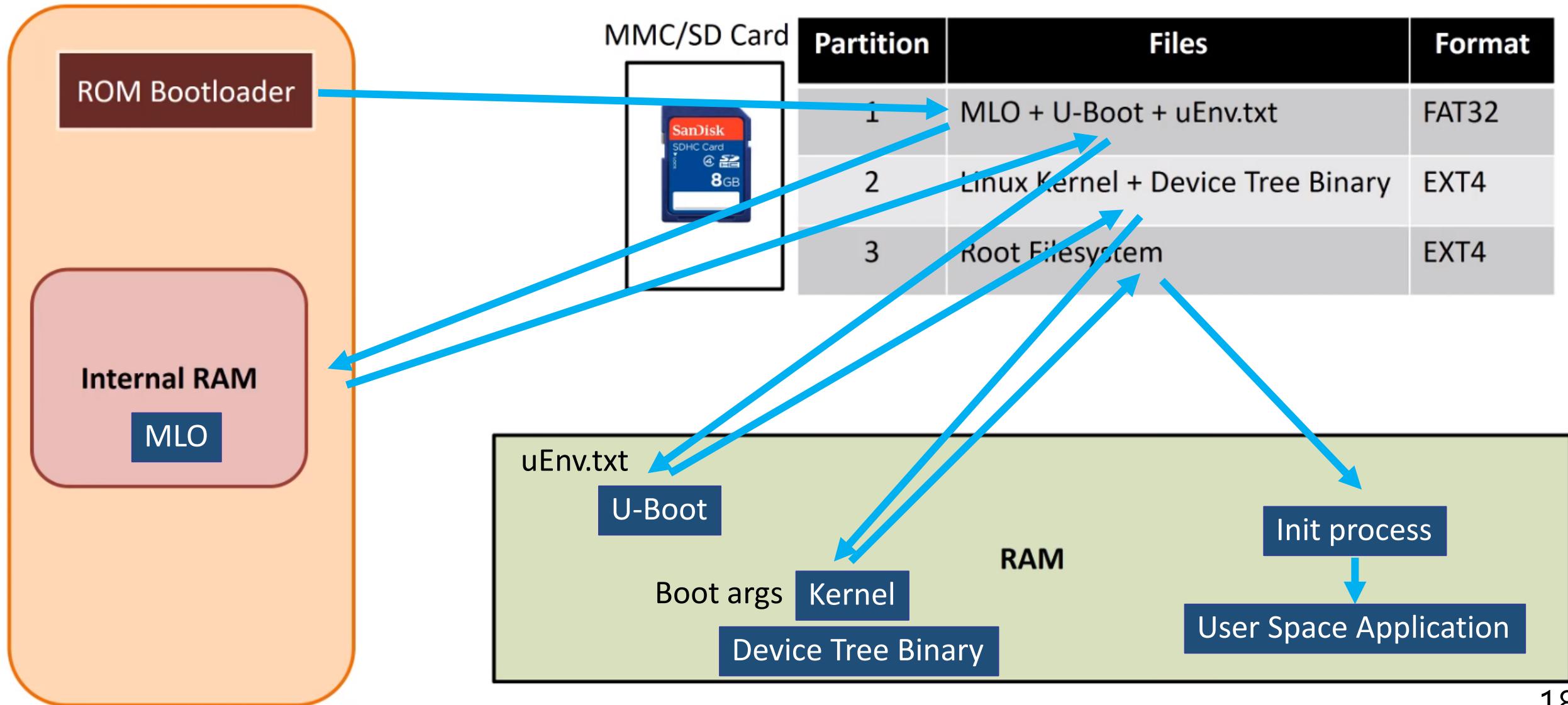
- When acting as floppy-like, the content is a single file system without any Master Boot Record (MBR) holding a partition table
- When acting as hard-drive-like, an MBR is present in the first sector. This MBR holds a table of partitions, one of which must be FAT12/16/32, primary and active.

Booting Sequence



Booting Sequence

AM335x



Device Tree

- ▶ On x86, most hardware can be dynamically discovered at run-time
 - ▶ PCI and USB provide dynamic enumeration capabilities
 - ▶ For the rest, ACPI provides tables describing hardware
 - ▶ Thanks to this, the kernel doesn't need to know in advance the hardware it will run on
- ▶ On ARM, no such mechanism exists at the hardware level
 - ▶ In the old days (prior to ~2011), the kernel code itself contained a description of all HW platforms it had to support
 - ▶ In ~2011, the ARM kernel developers switched to a different solution for HW description: **Device Tree**
 - ▶ Done together with an effort called **multiplatform ARM kernel**

Device Tree

- ▶ A **tree of nodes** describing non-discoverable hardware
- ▶ Providing **information** such as register addresses, interrupt lines, DMA channels, type of hardware, etc.
- ▶ Provided by the **firmware** to the **operating system**
- ▶ Operating system agnostic, **not Linux specific**
 - ▶ Can be used by bootloaders, BSDs, etc.
- ▶ Originates from the PowerPC world, where it has been in use for many more years
- ▶ Source format written by developers (**dts**), compiled into a binary format understood by operating systems (**dtb**)
 - ▶ One **.dts** for each HW platform

Device Tree

sun5i.dtsi

```
/ {  
    cpus {  
        cpu0: cpu@0 {  
            device_type = "cpu";  
            compatible = "arm,cortex-a8";  
            reg = <0x0>;  
        };  
    };  
  
    soc@01c00000 {  
        compatible = "simple-bus";  
        ranges;  
  
        uart1: serial@01c28400 {  
            compatible = "snps,dw-apb-uart";  
            reg = <0x01c28400 0x400>;  
            interrupts = <2>;  
            clocks = <&apb1_gates 17>;  
            status = "disabled";  
        };  
  
        uart3: serial@01c28c00 {  
            compatible = "snps,dw-apb-uart";  
            reg = <0x01c28c00 0x400>;  
            interrupts = <4>;  
            clocks = <&apb1_gates 19>;  
            status = "disabled";  
        };  
        [...]  
    };  
};
```

sun5i-r8-chip.dts

```
/ {  
    model = "NextThing C.H.I.P.";  
    compatible = "nextthing,chip", "allwinner,sun5i-r8",  
                 "allwinner,sun5i-a13";  
  
    leds {  
        compatible = "gpio-leds";  
  
        status {  
            label = "chip:white:status";  
            gpios = <&xp_gpio 2 GPIO_ACTIVE_HIGH>;  
            default-state = "on";  
        };  
    };  
    [...]  
  
&uart1 {  
    pinctrl-names = "default";  
    pinctrl-0 = <&uart1_pins_b>;  
    status = "okay";  
};
```

Device Tree

- ▶ Used for almost **all ARM platforms** in Linux, and all ARM64 ones
- ▶ Used for a few platforms in bootloaders such as U-Boot or Barebox
- ▶ Device Tree source code **stored in the Linux kernel tree**
 - ▶ Duplicated in U-Boot/Barebox source code as needed
 - ▶ Plan for a *central* repository, but never occurred
- ▶ Supposed to be **OS-agnostic and therefore backward compatible**
 - ▶ In practice, are changed quite often to accommodate Linux kernel changes
- ▶ Loaded in memory by the bootloader, together with the Linux kernel image
- ▶ Parsed by the Linux kernel at boot time to know which hardware is available

Linux Kernel

- ▶ Support for the ARM core is generally done by ARM engineers themselves
 - ▶ MMU, caches, virtualization, etc.
 - ▶ In `arch/arm` and `arch/arm64`
 - ▶ Generally in Linux upstream even before actual ARM SoCs with this core are available
- ▶ Support for the ARM SoC and HW platform is a different story
 - ▶ Requires drivers for each and every HW block, inside the SoC and on the board, in `drivers/`
 - ▶ Requires Device Tree descriptions, in `arch/arm(64)/boot/dts`
 - ▶ Sometimes supported only in vendor forks, sometimes supported in the upstream Linux kernel

Linux Kernel : Typical Support for an SoC

- ▶ Core drivers
 - ▶ Clock controllers (`drivers/clk`), reset controller (`drivers/reset`), pin-muxing controllers (`drivers/pinctrl`), interrupt controller (`drivers/irqchip`), timers (`drivers/clocksource`), GPIO controllers (`drivers/gpio`)
- ▶ Peripheral drivers
 - ▶ Bus controllers: I2C (`drivers/i2c`), SPI (`drivers/spi`), USB (`drivers/usb`), PCI (`drivers/pci`)
 - ▶ Display controller (`drivers/gpu/drm`), camera interface (`drivers/media`), touchscreen or other input devices (`drivers/input`), Ethernet controller (`drivers/net`)
- ▶ Platform code
 - ▶ On ARM, minimal amount of platform code in `arch/arm/mach-<foo>` for power management and SMP support
 - ▶ On ARM64, no platform code at all, power management and SMP activities handled using *PSCI*

Linux Kernel : from Vendor to Upstream

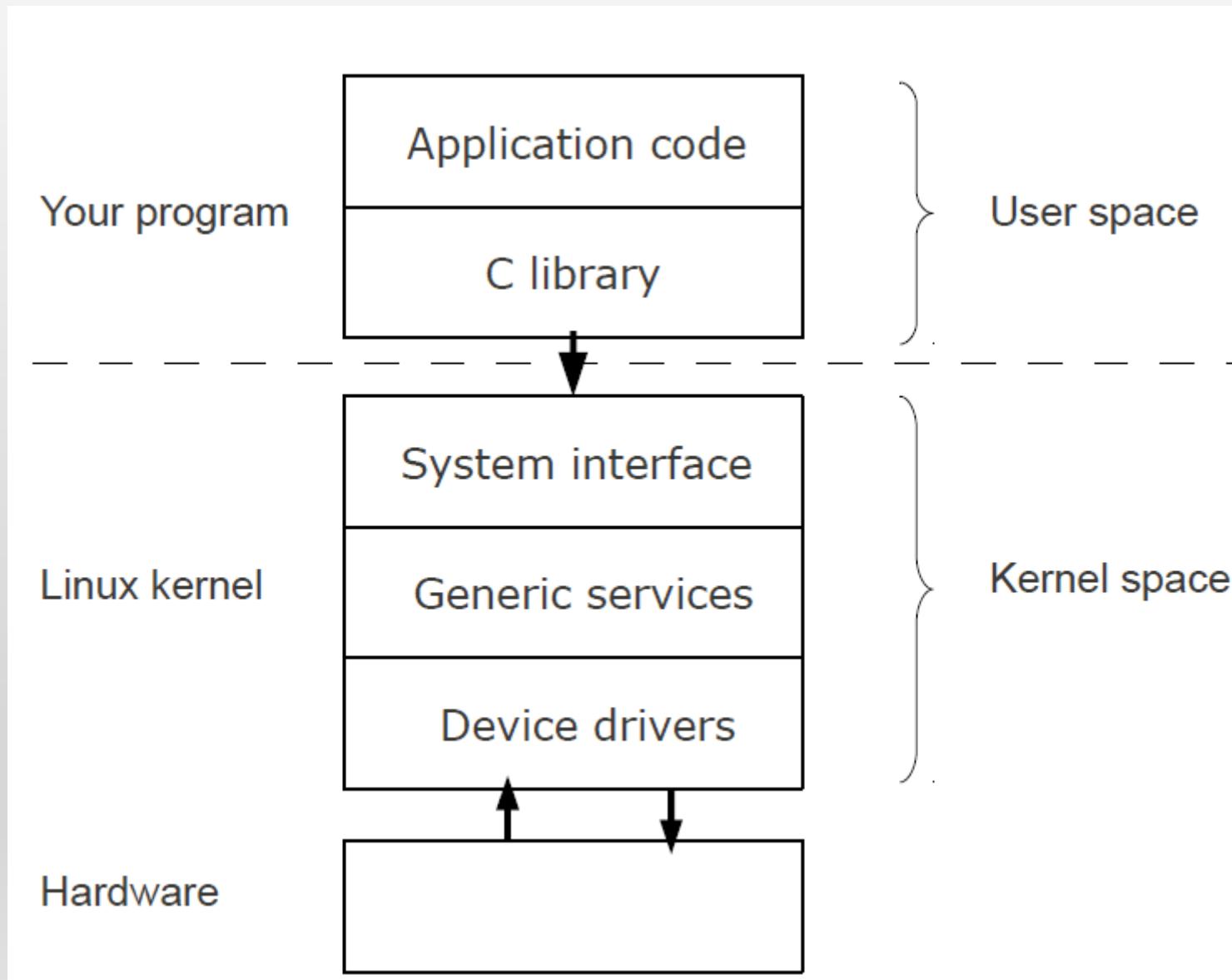
- ▶ Most vendors **fork the Linux kernel**, and add support for their SoC to their own fork
- ▶ Leads to kernel forks with sometimes **millions of added lines** for SoC support
 - ▶ Users **cannot easily change/upgrade** their kernel version
 - ▶ Generally of **poor quality**
 - ▶ Situation got somewhat worse with Android
- ▶ Some vendors engage with the **upstream** Linux kernel community, and submit patches
 - ▶ More and more vendors taking this direction
 - ▶ Mileage may vary depending on the vendor, and sometimes the SoC family
- ▶ The **community** also significantly contributes to upstream Linux kernel support for ARM SoCs
 - ▶ Example: Allwinner support is fully community-contributed, no involvement from the vendor

Root Filesystem

- ▶ Regular desktop-style **distributions**: Debian, Ubuntu, Raspbian, Fedora, etc.
- ▶ **Specialized** systems: Android, Tizen, etc.
- ▶ Embedded Linux **build systems**
 - ▶ Widely used for embedded systems
 - ▶ Produce a Linux root filesystem through cross-compilation
 - ▶ Allows a much more customized and stripped down system than a full-blown distribution
 - ▶ Examples: OpenEmbedded/Yocto, Buildroot, OpenWRT, etc.



Kernel vs User Space



Kernel Modules

- Kernel code that is loaded after the kernel has booted
- Advantages
 - Load drivers on demand (e.g. for USB devices)
 - Load drivers later – speed up initial boot
- Disadvantages
 - Adds kernel version dependency to root file system
 - More files to manage

What is User Space?

- A sane (POSIX) environment for applications (unlike the kernel)
- The main components are
 - Programs – e.g. init and a shell
 - Libraries - e.g. libc
 - Configuration files in /etc
 - Device nodes in /dev
 - User data in /home

Busybox

- Web - <http://www.busybox.net>
- Very common in embedded systems
- Single binary that masquerades as many Linux utilities, including
 - init
 - ash (a Bourne shell)
 - file system utilities: mount, umount,...
 - network utilities: ifconfig, route,...
 - and of course, the vi editor

Busybox

```
# ls -l /bin
lrwxrwxrwx 1 root root      7 2008-08-06 11:44 addgroup -> busybox
lrwxrwxrwx 1 root root      7 2008-08-06 11:44 adduser -> busybox
lrwxrwxrwx 1 root root      7 2008-08-06 11:44 ash -> busybox
-rwxr-xr-x 1 root root 744480 2008-05-16 15:46 busybox
lrwxrwxrwx 1 root root      7 2008-08-06 11:44 cat -> busybox
...
...
```

So when you type (for example)
cat /etc/inittab

... launches /bin/busybox with argv [0] = "/bin/cat"

Busybox main() parses argv[0] and jumps to cat applet

init

- /sbin/init is the first program to be run
 - change by setting kernel parameter “init=...”
- Two common versions of init
 - Busybox init
 - e.g. by buildroot
 - System V init
 - e.g. by Angstrom

Busybox init

- Begins by reading /etc/inittab, for example:

/etc/inittab

```
::sysinit:/etc/init.d/rcS
::respawn:-/sbin/getty -L ttyS0 115200 vt100
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
::restart:/sbin/init
```

/etc/init.d/rcS

```
#!/bin/sh
echo "Starting rcS"
mount -t proc proc /proc
mount -t sysfs sysfs /sys
ifconfig lo 127.0.0.1
ifconfig eth0 192.168.1.101
```



Q & A

Thank you for your attention.

HW00 is due today and HW01 is going to be released

Embedded System

Lecture 04: Linux and Real time

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

Hong-Yue Technology Research Building 334
R 09:10 - 12:00

What is Real Time System ?

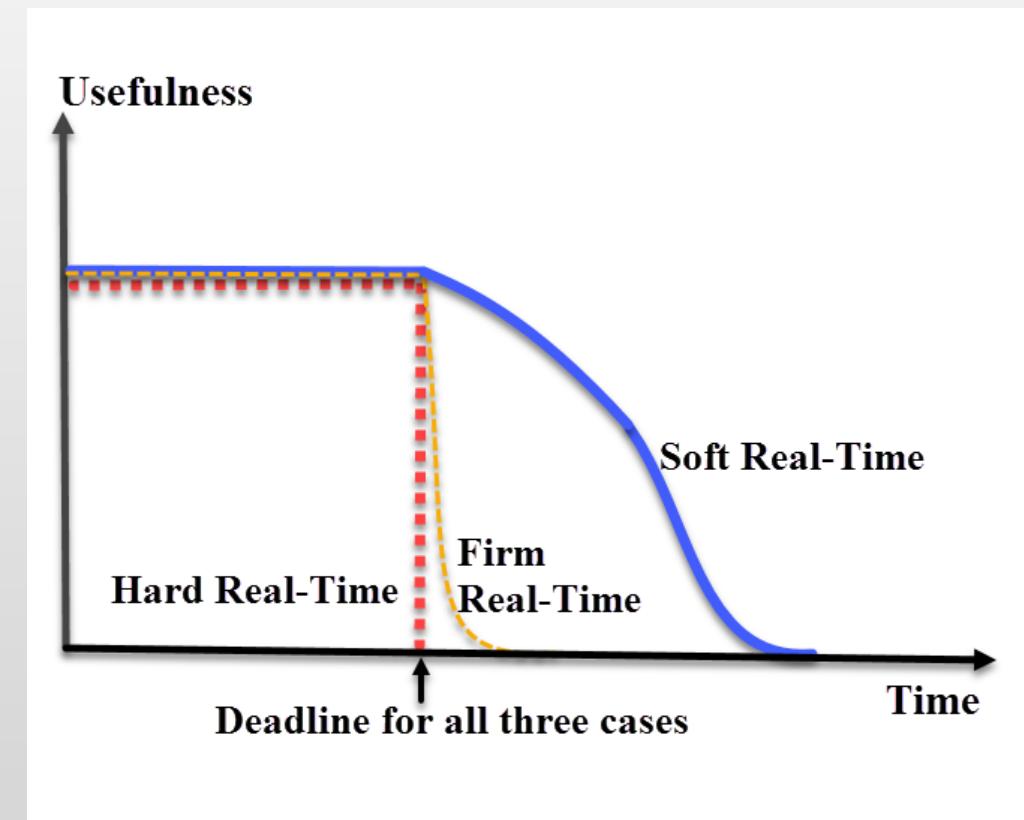
- Ask five people what “real time System” means
 - Chances are, you will get five different answers.
 - Possible answers that are incorrect
 - Return results in a few second ?
 - Return results with unnoticeable latency ?
 - Return results with very high speed ?
 - In the world of computer, real time means that the correctness of the system depends
 - Not only on **the logical result of computation**
 - But also on **the time at which the results are produced**

Different Types of Real Time Requirement

- Based on how tolerable the real time system is
- Hard real-time systems -> Usually Life critical
 - (e.g., Avionics, Command & Control Systems)
- Firm real-time systems -> Usually Money related
 - (e.g., Banking, Online transaction processing)
- Soft real-time systems -> Not so big deal
 - (e.g., Video streaming)

Penalty of Real Time Requirement

- Hard deadline:
 - Penalty due to missing deadline is a higher order of magnitude than the reward in meeting the deadline
- Firm deadline:
 - Penalty and reward are in the same order of magnitude
- Soft deadline:
 - Penalty often lesser magnitude than reward



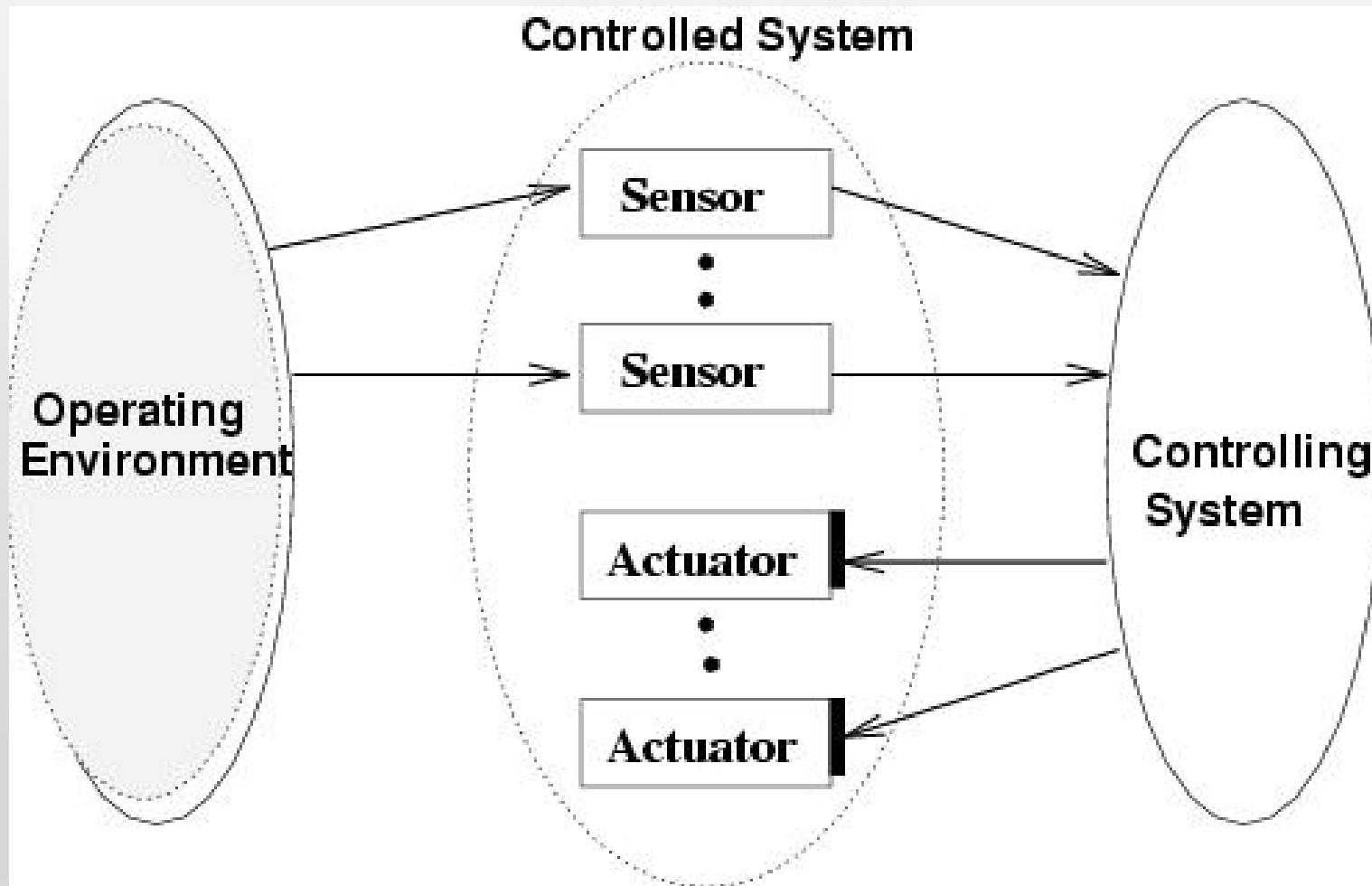
These Two are Usually Discussed Together

- Hard real time
 - Tasks have to be performed on time
 - Failure to meet deadlines is fatal
- Soft real time
 - Tasks are performed as fast as possible
 - Late completion of jobs is undesirable but not fatal.
 - System performance degrades as more & more jobs miss deadlines

Common Misconceptions

- Real-time computing is equivalent to fast computing.
- Real-time programming is assembly coding, priority interrupt programming, and writing device drivers.
- Real-time systems operate in a static environment.
- The problems in real-time system design have all been solved in other areas of computer science.

A Typical Real Time System



Most Real-Time Systems are embedded

- An embedded system is a computer built into a system but not seen by users as being a computer
- Examples
 - FAX machines
 - Copiers
 - Printers
 - Scanners
 - Routers
 - Robots

Role of an OS in Real Time Systems

- Standalone Applications
 - Often no OS involved
 - Micro controller based Embedded Systems
- Some Real Time Applications are huge & complex
 - Multiple threads
 - Complicated Synchronization Requirements
 - File system / Network / Windowing support
 - OS primitives reduce the software design time

Foreground/Background Systems

- Small systems of low complexity
 - These systems are also called “super-loops”
- A scheduling algorithm that is used to control an execution of multiple processes on a single processor
- Two waiting lists
 - Interrupt service routines (ISRs) handle asynchronous events (**foreground**)
 - An application consists of an infinite loop of desired operations (**background**)

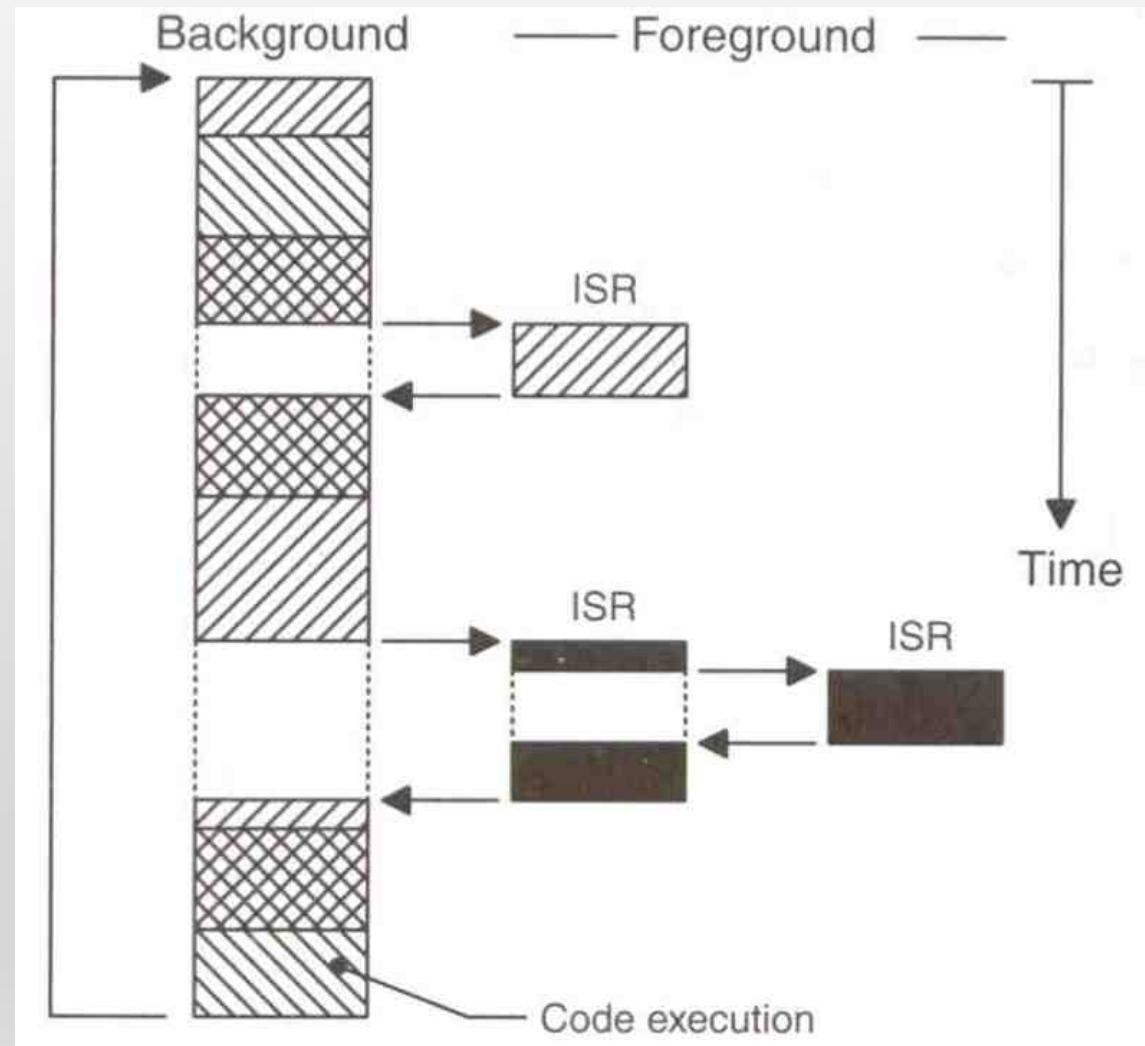
Foreground/Background Systems (cont.)

- Critical operations must be performed by the ISRs to ensure the timing correctness
- Thus, ISRs tend to take longer than they should
- Task-Level Response
 - Information for a background module is not processed until the module gets its turn

Foreground/Background Systems

- The execution time of typical code is not constant
- If a code is modified, the timing of the loop is affected
- Most high-volume microcontroller-based applications are F/B systems
 - Microwave ovens
 - Telephones
 - Toys

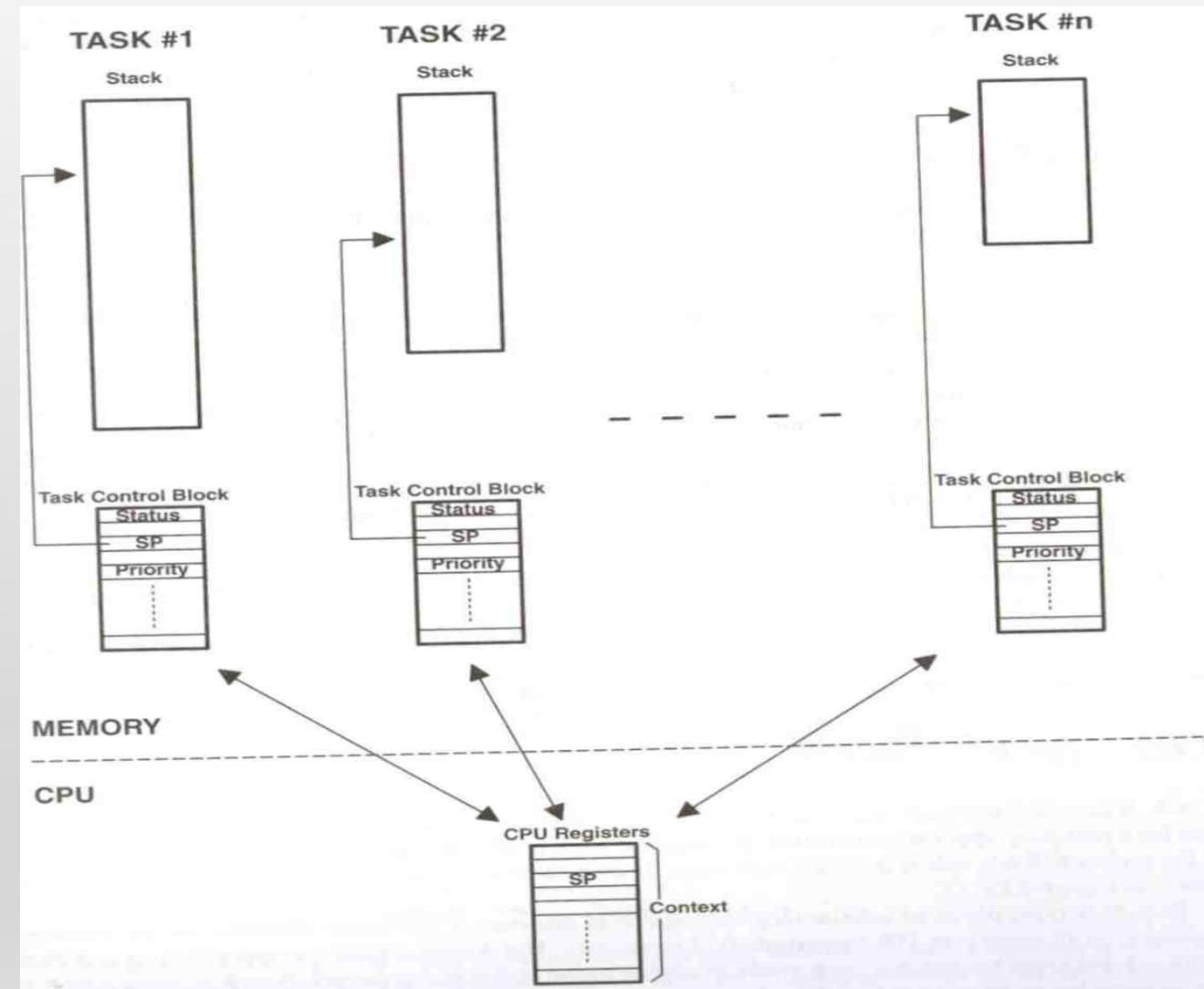
Foreground/Background Systems (cont.)



- From a power consumption point of view, it might be better to halt and perform all processing in ISRs

Multitasking Systems

- Like F/B systems with multiple backgrounds
- Allow programmers to manage complexity inherent in real-time applications



Scheduling in RTOS

- More information about the tasks are known
 - Number of tasks
 - Resource Requirements
 - Execution time
 - Deadlines
- Being a more deterministic system better scheduling algorithms can be devised.

Scheduling Algorithms in RTOS

- Clock Driven Scheduling
- Weighted Round Robin Scheduling
- Priority Scheduling

Scheduling Algorithms in RTOS (cont.)

- Clock Driven
 - All parameters about jobs (execution time/deadline) known in advance.
 - Schedule can be computed offline or at some regular time instances.
 - Minimal runtime overhead.
 - Not suitable for many applications.

Scheduling Algorithms in RTOS (cont.)

- Weighted Round Robin
 - Jobs scheduled in FIFO manner
 - Time quantum given to jobs is proportional to it's weight
 - Example use : High speed switching network
 - QOS guarantee.
 - Not suitable for precedence constrained jobs.
 - Job A can run only after Job B. No point in giving time quantum to Job B before Job A.

Scheduling Algorithms in RTOS (cont.)

- Priority Scheduling
 - Processor never left idle when there are ready tasks
 - Processor allocated to processes according to priorities
 - Priorities
 - Static - at design time
 - Dynamic - at runtime

Priority Scheduling

- Earliest Deadline First (EDF)
 - Process with earliest deadline given highest priority
- Least Slack Time First (LSF)
 - slack = relative deadline – execution left
- Rate Monotonic Scheduling (RMS)
 - For periodic tasks
 - Tasks priority inversely proportional to it's period

Schedulers

- Also called “dispatchers”
- Schedulers are parts of the kernel responsible for determining which task runs next
- Most real-time kernels use priority-based scheduling
 - Each task is assigned a priority based on its importance
 - The priority is application-specific

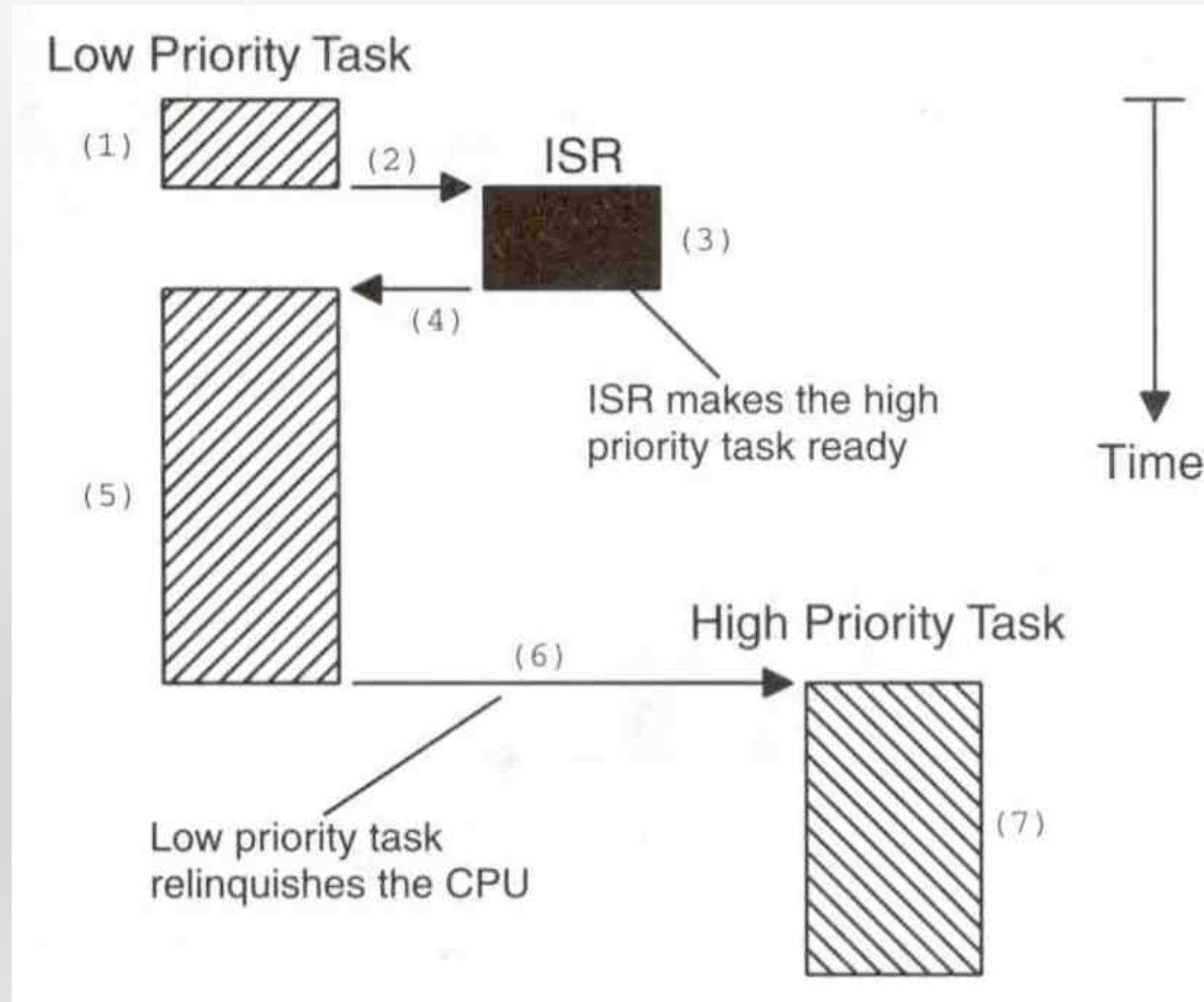
Priority-Based Kernels

- There are two types
 - Non-preemptive
 - Preemptive

Non-Preemptive Kernels

- Perform “cooperative multitasking”
 - Each task must explicitly give up control of the CPU
 - This must be done frequently to maintain the illusion of concurrency
- Asynchronous events are still handled by ISRs
 - ISRs can make a higher-priority task ready to run
 - But ISRs always return to the interrupted tasks

Non-Preemptive Kernels (cont.)



Advantages of Non-Preemptive Kernels

- Interrupt latency is typically low
- Can use non-reentrant functions without fear of corruption by another task
 - Because each task can run to completion before it relinquishes the CPU
 - However, non-reentrant functions should not be allowed to give up control of the CPU
- Task-response is now given by the time of the longest task
 - much lower than with F/B systems

Advantages of Non-Preemptive Kernels (cont.)

- Less need to guard shared data through the use of semaphores
 - However, this rule is not absolute
 - Shared I/O devices can still require the use of mutual exclusion semaphores
 - A task might still need exclusive access to a printer

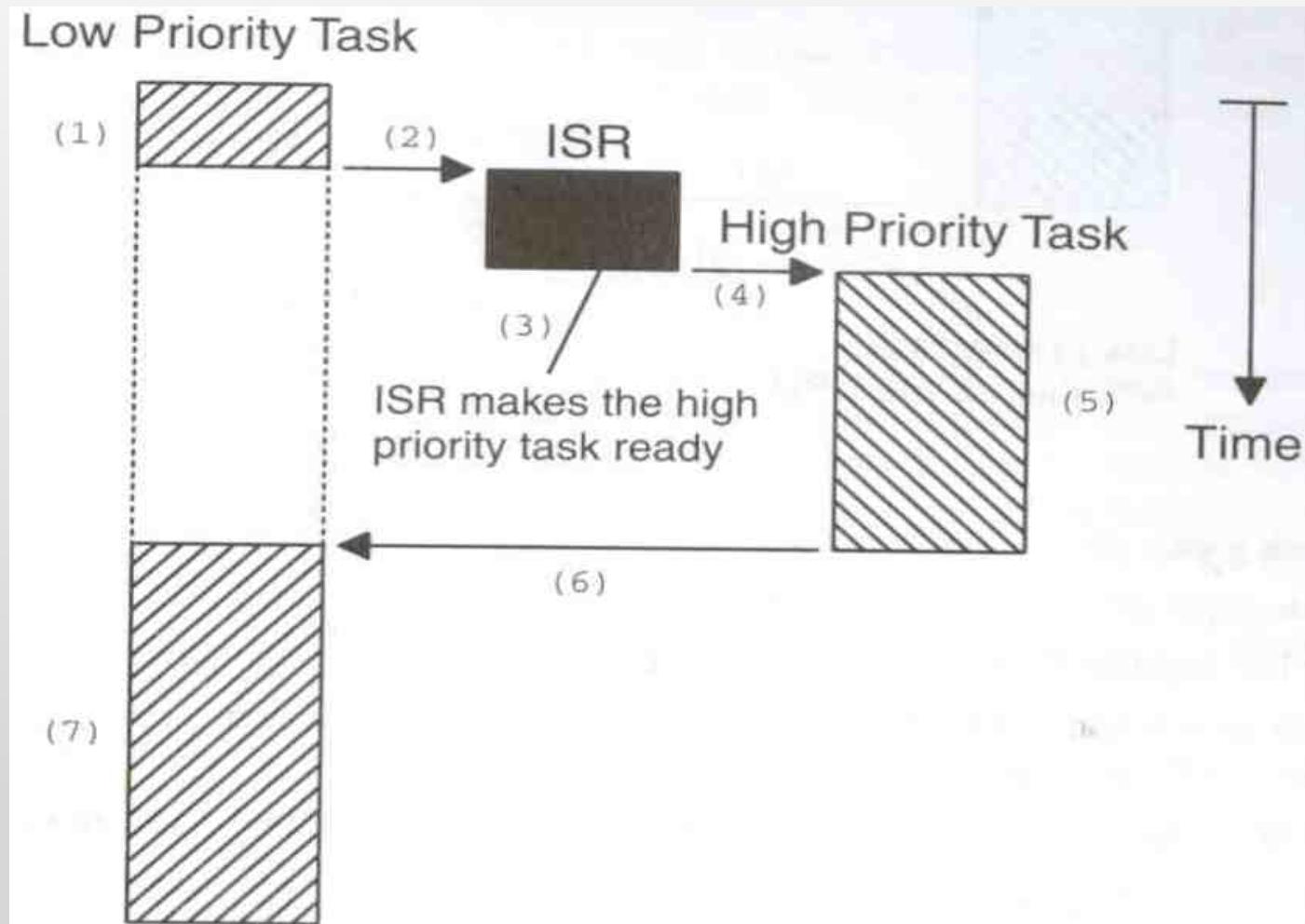
Disadvantages of Non-Preemptive Kernels

- Responsiveness
 - A higher priority task might have to wait for a long time
 - Response time is nondeterministic
- Very few commercial kernels are non-preemptive

Preemptive Kernels

- The highest-priority task ready to run is always given control of the CPU
 - If an ISR makes a higher-priority task ready, the higher-priority task is resumed (instead of the interrupted task)
- Most commercial real-time kernels are preemptive

Preemptive Kernels (cont.)



Preemptive Kernels (cont.)

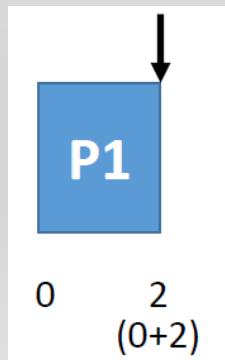
- Advantages
 - Execution of the highest-priority task is deterministic
 - Task-level response time is minimized
- Disadvantages
 - Should not use non-reentrant functions unless exclusive access to these functions is ensured

Preemptive Shortest Job First (SJF)

- Also known as Shortest-remaining-time-first
- Starvation will happen -> Resolved by using priority aging

Current Process : P1

P2 Arrive



Process ID	Next CPU Burst Length	Priority	Arrival Time
P1	10	1	0
P2	7	2	2
P3	5	4	3
P4	8	3	6

Preemptive Shortest Job First (SJF) (cont.)

- Also known as Shortest-remaining-time-first
- Starvation will happen -> Resolved by using priority aging

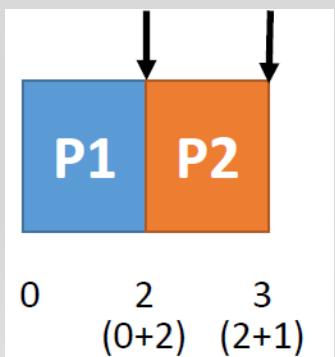
Current Process : P1、P2

P1 burst time : $10 - 2 = 8$

P1 : 8 > P2 : 7 \rightarrow P2

P3 Arrive

P2 Arrive



Process ID	Next CPU Burst Length	Priority	Arrival Time
P1	8	1	0
P2	7	2	2
P3	5	4	3
P4	8	3	6

Preemptive Shortest Job First (SJF) (cont.)

- Also known as Shortest-remaining-time-first
- Starvation will happen -> Resolved by using priority aging

Current Process : P1、P2、P3

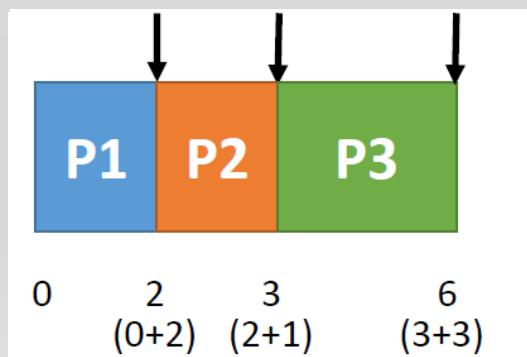
P2 burst time : $7-1=6$

P1 : 8 > P2 : 6 > P3 : 5 → P3

P3 Arrive

P2 Arrive

P4 Arrive



Process ID	Next CPU Burst Length	Priority	Arrival Time
P1	8	1	0
P2	6	2	2
P3	5	4	3
P4	8	3	6

Preemptive Shortest Job First (SJF) (cont.)

- Also known as Shortest-remaining-time-first
- Starvation will happen -> Resolved by using priority aging

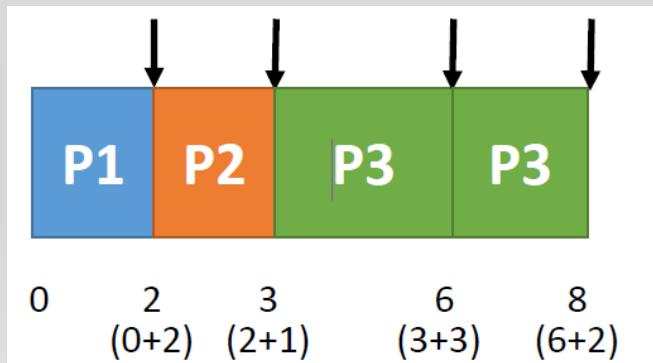
Current Process : P1、P2、P3、P4

P3 burst time : 5-3=2

1 : 8 = P4 : 8 > P2 : 6 > P3 : 2 → P3

P3 Arrive P3 end

P2 Arrive P4 Arrive



Process ID	Next CPU Burst Length	Priority	Arrival Time
P1	8	1	0
P2	6	2	2
P3	2	4	3
P4	8	3	6

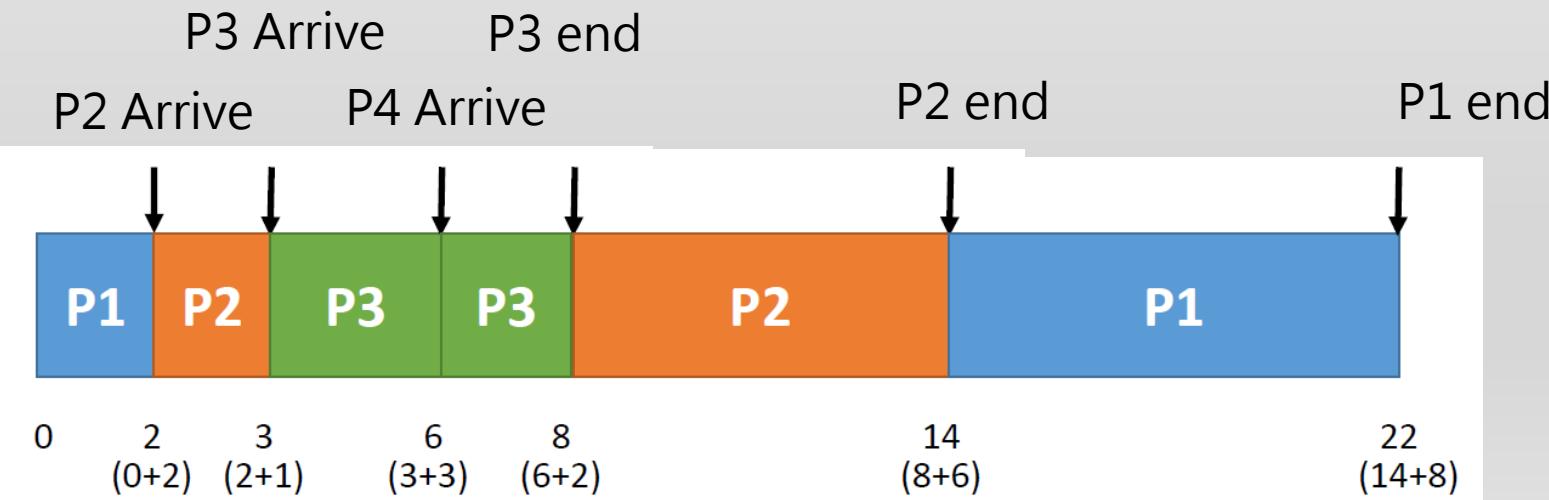
Preemptive Shortest Job First (SJF) (cont.)

- Also known as Shortest-remaining-time-first
- Starvation will happen -> Resolved by using priority aging

Current Process : P1、P4

P2 burst time : 6-6=0

P1 : 8 = P4 : 8 → P1



Process ID	Next CPU Burst Length	Priority	Arrival Time
P1	8	1	0
P2	0	2	2
P3	0	4	3
P4	8	3	6

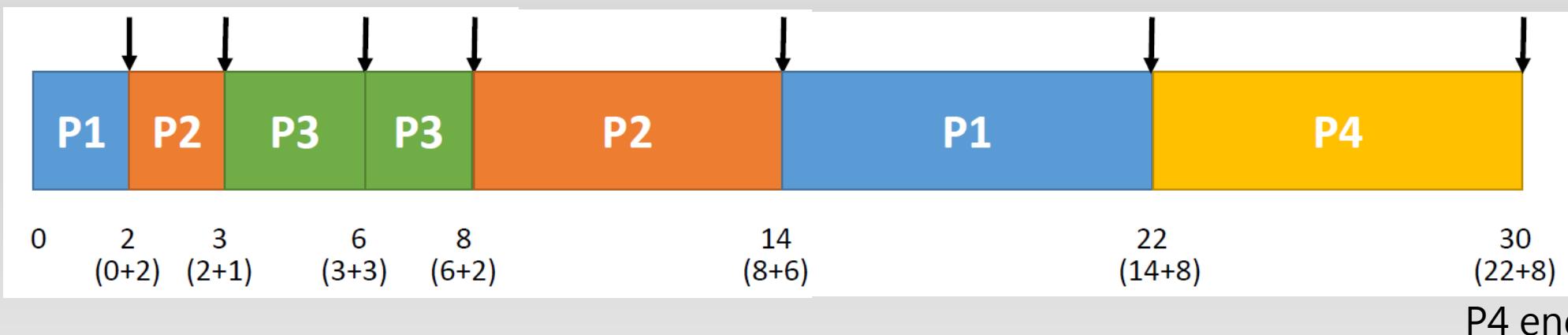
Preemptive Shortest Job First (SJF) (cont.)

- Also known as Shortest-remaining-time-first
- Starvation will happen -> Resolved by using priority aging

Current Process : P4

P1 burst time : $8-8=0$

P3 Arrive P3 end
P2 Arrive P4 Arrive P2 end P1 end



Process ID	Next CPU Burst Length	Priority	Arrival Time
P1	0	1	0
P2	0	2	2
P3	0	4	3
P4	8	3	6

Reentrant Functions

- A reentrant function can be used by more than one task without fear of data corruption
- It can be interrupted and resumed at any time without loss of data
- It uses local variables (CPU registers or variables on the stack)
- Protect data when global variables are used

Non-Reentrant Function Example

```
int Temp;
```

```
void swap(int *x, int *y)
```

```
{
```

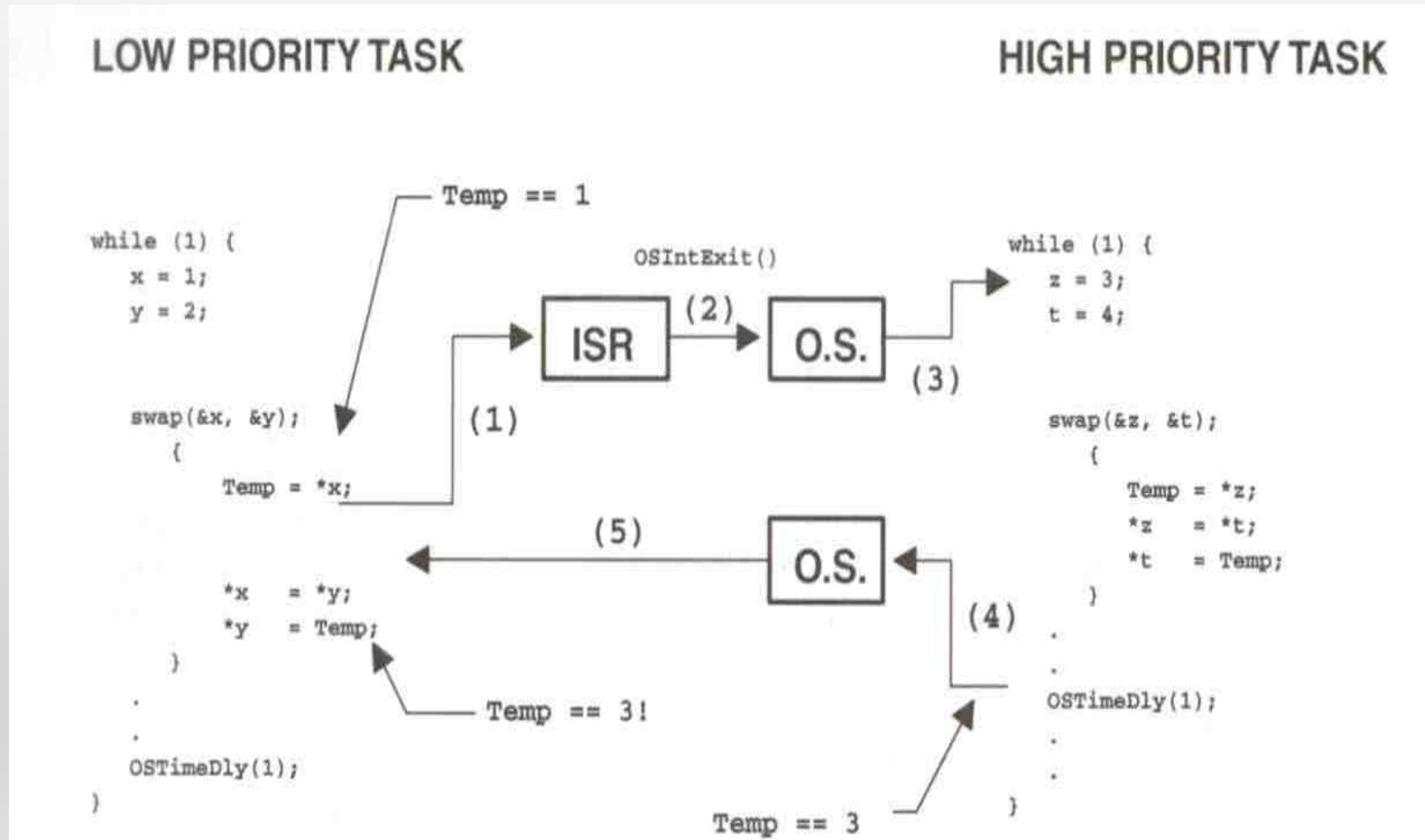
```
    Temp = *x;
```

```
    *x = *y;
```

```
    *y = Temp;
```

```
}
```

Non-Reentrant Function Example (cont.)



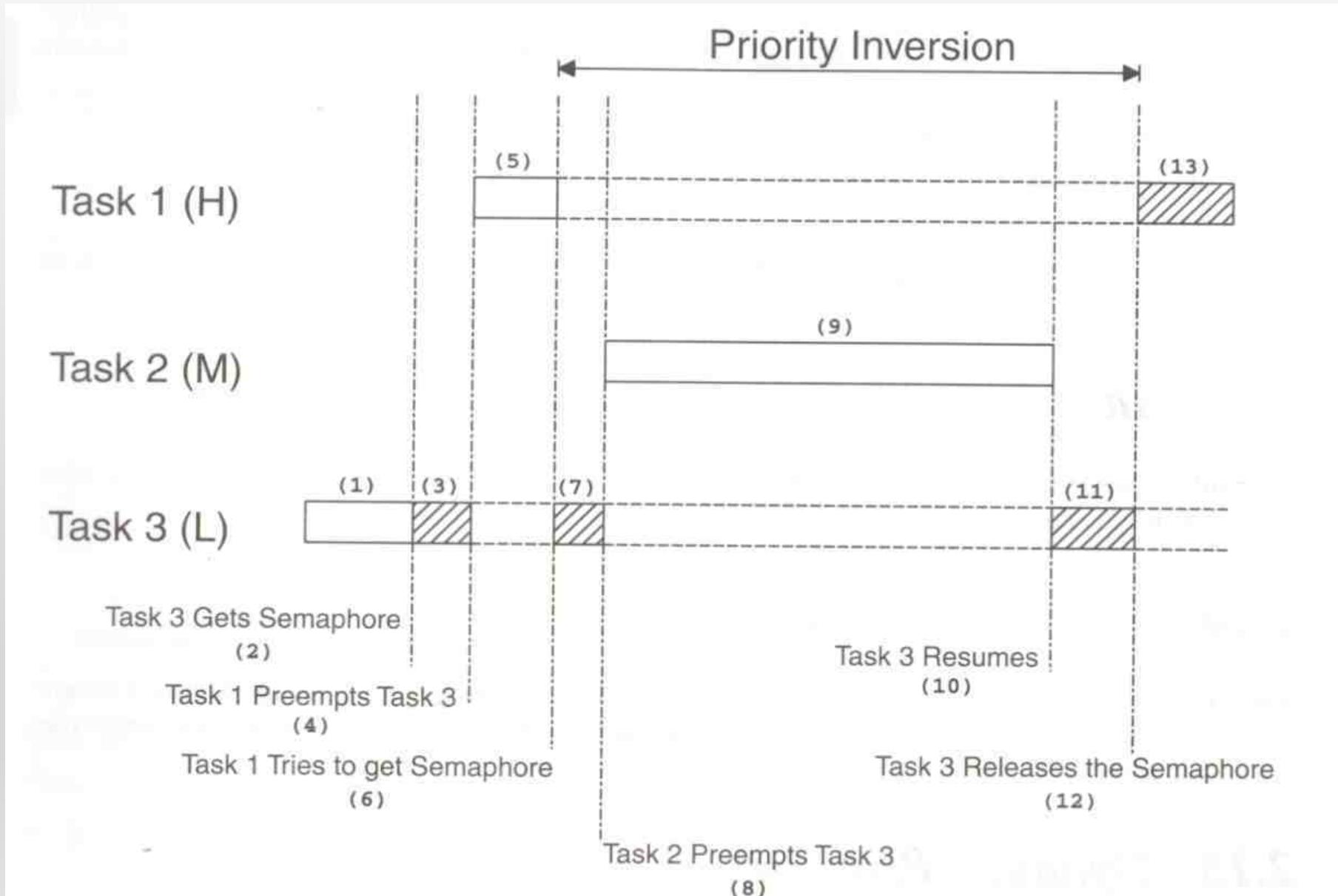
Resource Allocation in RTOS

- Resource Allocation
 - The issues with scheduling applicable here.
 - Resources can be allocated in
 - Weighted Round Robin
 - Priority Based
- Some resources are non preemptible
 - Example: semaphores
- Priority inversion problem may occur if priority scheduling is used

Priority Inversion Problem

- Common in real-time kernels
- Suppose task 1 has a higher priority than task 2
- Also, task 2 has a higher priority than task 3
- If mutual exclusion is used in accessing a shared resource, priority inversion may occur

Priority Inversion Problem



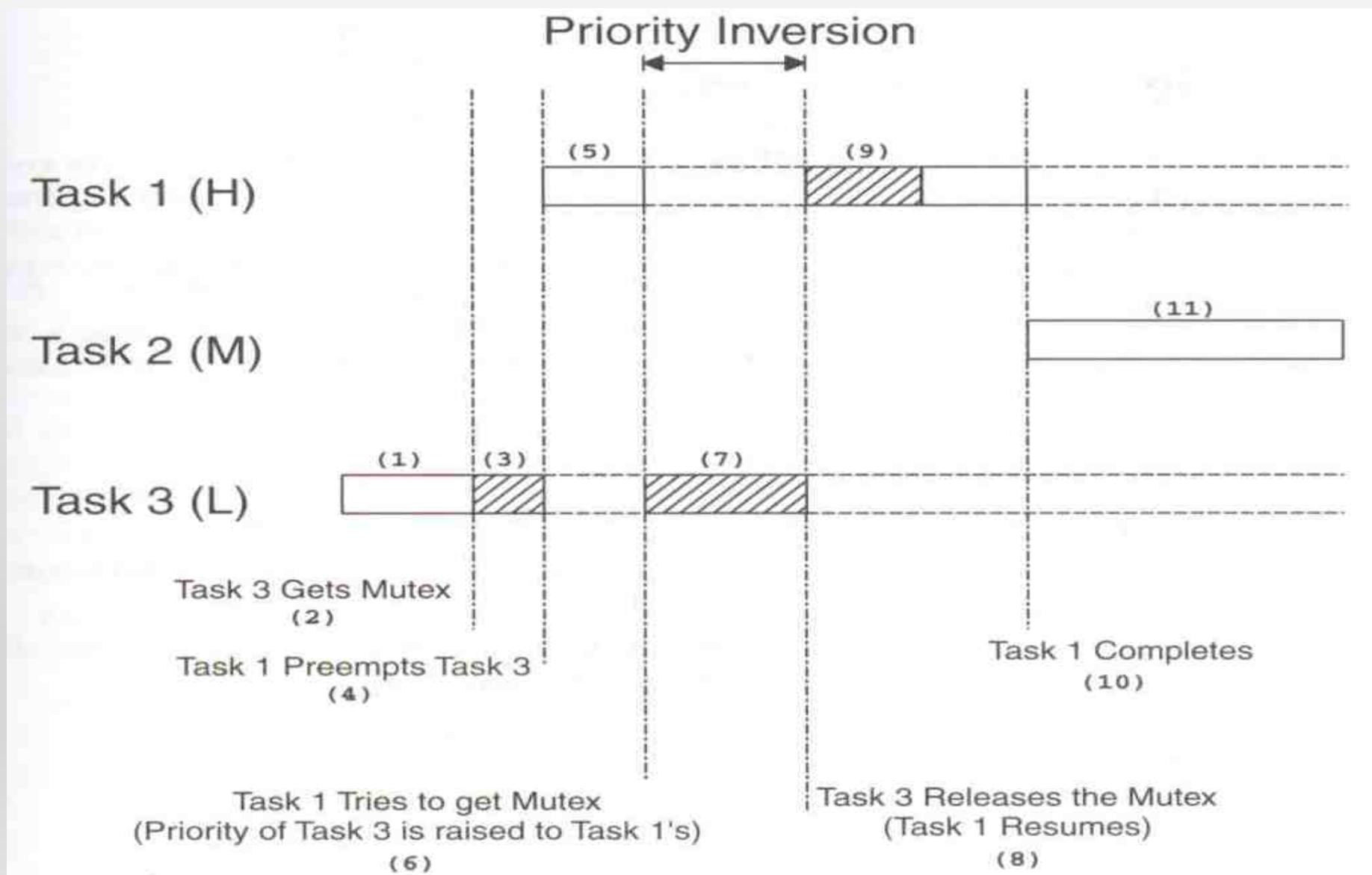
A Solution to Priority Inversion Problem

- We can correct the problem by raising the priority of task 3
 - Just for the time it accesses the shared resource
 - After that, return to the original priority
 - What if task 3 finishes the access before being preempted by task 1?
 - incur overhead for nothing

A Better Solution to the Problem

- Priority Inheritance
 - Automatically change the task priority when needed
 - The task that holds the resource will inherit the priority of the task that waits for that resource until it releases the resource

Priority Inheritance Example



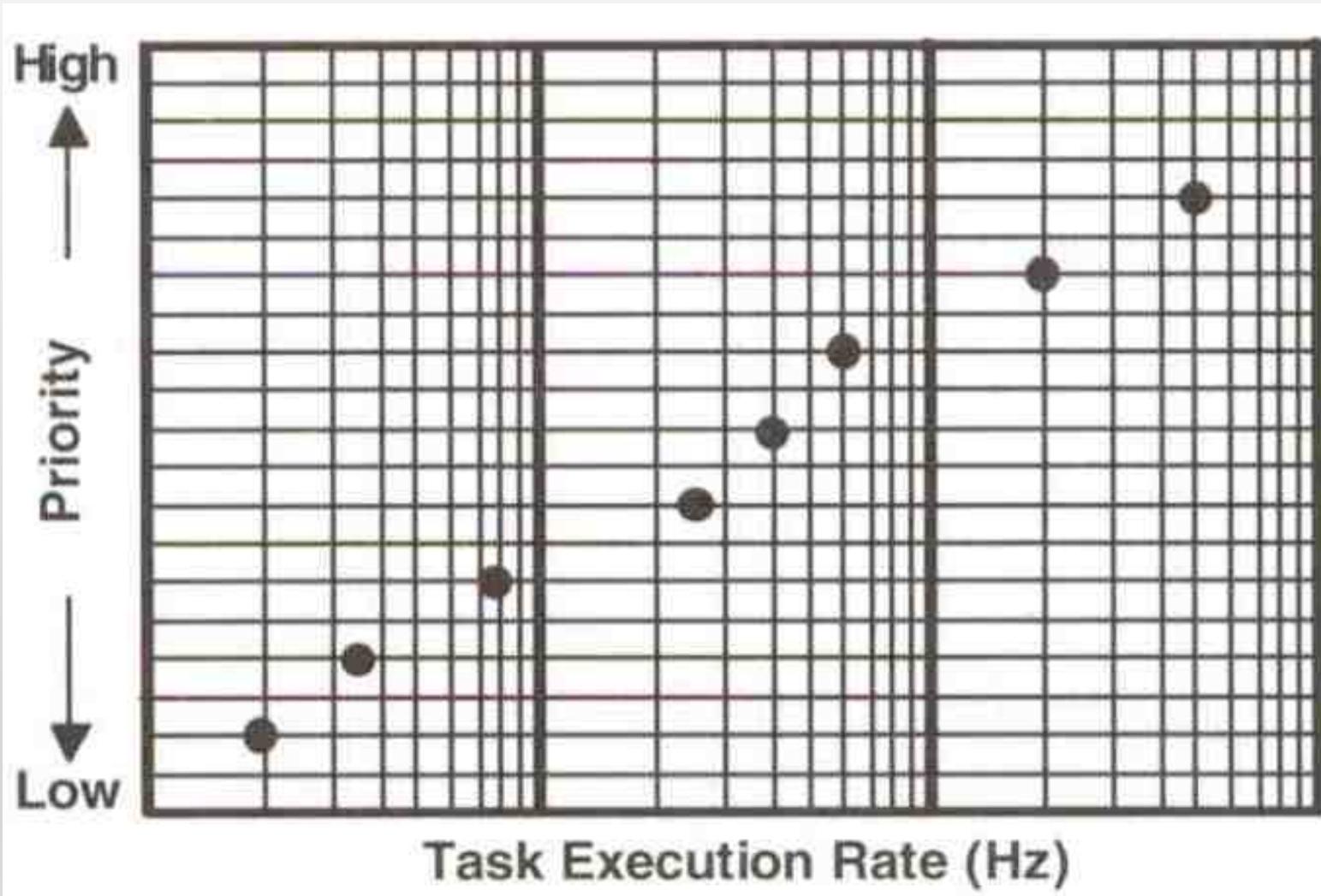
Assigning Task Priorities

- Not trivial
- In most systems, not all tasks are critical
- Non-critical tasks are obviously low-priorities
- Most real-time systems have a combination of soft and hard requirements

A Technique for Assigning Task Priorities

- Rate Monotonic Scheduling (RMS)
 - Priorities are based on how often tasks execute
- Assumption in RMS
 - All tasks are periodic with regular intervals
 - Tasks do not synchronize with one another, share data, or exchange data
 - Preemptive scheduling

RMS Example



RMS: CPU Time and Number of Tasks

<i>Number of Tasks</i>	$n(2^{1/n} - 1)$
1	1.000
2	0.828
3	0.779
4	0.756
5	0.743
.	.
.	.
.	.
-	0.693

RMS: CPU Time and Number of Tasks (cont.)

- The upper bound for an infinite number of tasks is 0.6973
 - To meet all hard real-time deadlines based on RMS, CPU use of all time-critical tasks should be less than 70%
 - Note that you can still have non-time-critical tasks in a system
 - So, 100% of CPU time is used
 - But not desirable because it does not allow code changes or added features later

RMS: CPU Time and Number of Tasks (cont.)

- Note that, in some cases, the highest-rate task might not be the most important task
 - Eventually, application dictates the priorities
 - However, RMS is a starting point

Other RTOS issues

- Interrupt Latency should be very small
 - Kernel has to respond to real time events
 - Interrupts should be disabled for minimum possible time
- For embedded applications Kernel Size should be small
 - Should fit in ROM
- Sophisticated features can be removed
 - No Virtual Memory
 - No Protection

Mutual Exclusion

- The easiest way for tasks to communicate is through shared data structures
 - Global variables, pointers, buffers, linked lists, and ring buffers
 - Must ensure that each task has exclusive access to the data to avoid data corruption

Mutual Exclusion (cont.)

- The most common methods are:
 - Disabling interrupts
 - Performing test-and-set operations
 - Disabling scheduling
 - Using semaphores

Disabling and Enabling Interrupts

- The easiest and fastest way to gain exclusive access
- Example:
 - Disable interrupts;
 - Access the resource;
 - Enable interrupts;

Disabling and Enabling Interrupts (cont.)

- This is the only way that a task can share variables with an ISR
- However, do not disable interrupts for too long
- Because it adversely impacts the “interrupt latency”!
- Good kernel vendors should provide the information about how long their kernels will disable interrupts

Test-and-Set (TAS) Operations

- Two functions could agree to access a resource based on a global variable value
- If the variable is 0, the function has the access
 - To prevent the other from accessing the resource, the function sets the variable to 1
- TAS operations must be performed indivisibly by the CPU (e.g., 68000 family)
- Otherwise, you must disable the interrupts when doing TAS on the variable

TAS Example

```
Disable interrupts;  
if (variable is 0) {  
    Set variable to 1;  
    Enable interrupts;  
    Access the resource;  
    Disable interrupts;  
    Set variable to 0;  
    Enable interrupts;  
} else {  
    Enable interrupts;  
}  
}
```

Disabling and Enabling the Scheduler

- Viable for sharing variables among tasks but not with an ISR
- Scheduler is locked but interrupts are still enabled
 - Thus, ISR returns to the interrupted task
 - Similar to a non-preemptive kernel (at least, while the scheduler is locked)

Disabling and Enabling the Scheduler (cont.)

- Example:
 - Lock scheduler;
 - Access shared data;
 - Unlock scheduler;
- Even though this works well, you should avoid it because it defeats the purpose of having a kernel

Semaphores

- Invented by Edgser Dijkstra in the mid-1960s
- Offered by most multitasking kernels
- Used for:
 - Mutual exclusion
 - Signaling the occurrence of an event
 - Synchronizing activities among tasks

Semaphores (cont.)

- A semaphore is a key that your code acquires in order to continue execution
- If the key is already in use, the requesting task is suspended until the key is released
- There are two types
 - Binary semaphores
 - 0 or 1
 - Counting semaphores
 - ≥ 0

Semaphore Operations

- Initialize (or create)
- Value must be provided
- Waiting list is initially empty
- Wait (or pend)
- Used for acquiring the semaphore
- If the semaphore is available (the semaphore value is positive), the value is decremented, and the task is not blocked
- Otherwise, the task is blocked and placed in the waiting list

Semaphore Operations

- Initialize (or create)
 - Value must be provided
 - Waiting list is initially empty

Semaphore Operations (cont.)

- Wait (or pend)
 - Used for acquiring the semaphore
 - If the semaphore is available (the semaphore value is positive), the value is decremented, and the task is not blocked
 - Otherwise, the task is blocked and placed in the waiting list
 - Most kernels allow you to specify a timeout
 - If the timeout occurs, the task will be unblocked and an error code will be returned to the task

Semaphore Operations (cont.)

- Signal (or post)
 - Used for releasing the semaphore
 - If no task is waiting, the semaphore value is incremented
 - Otherwise, make one of the waiting tasks ready to run but the value is not incremented
 - Which waiting task to receive the key?
 - Highest-priority waiting task
 - First waiting task

Semaphore Example

```
Semaphore *s;
```

```
Time timeout;
```

```
INT8U error_code;
```

```
timeout = 0;
```

```
Wait(s, timeout, &error_code);
```

```
Access shared data;
```

```
Signal(s);
```

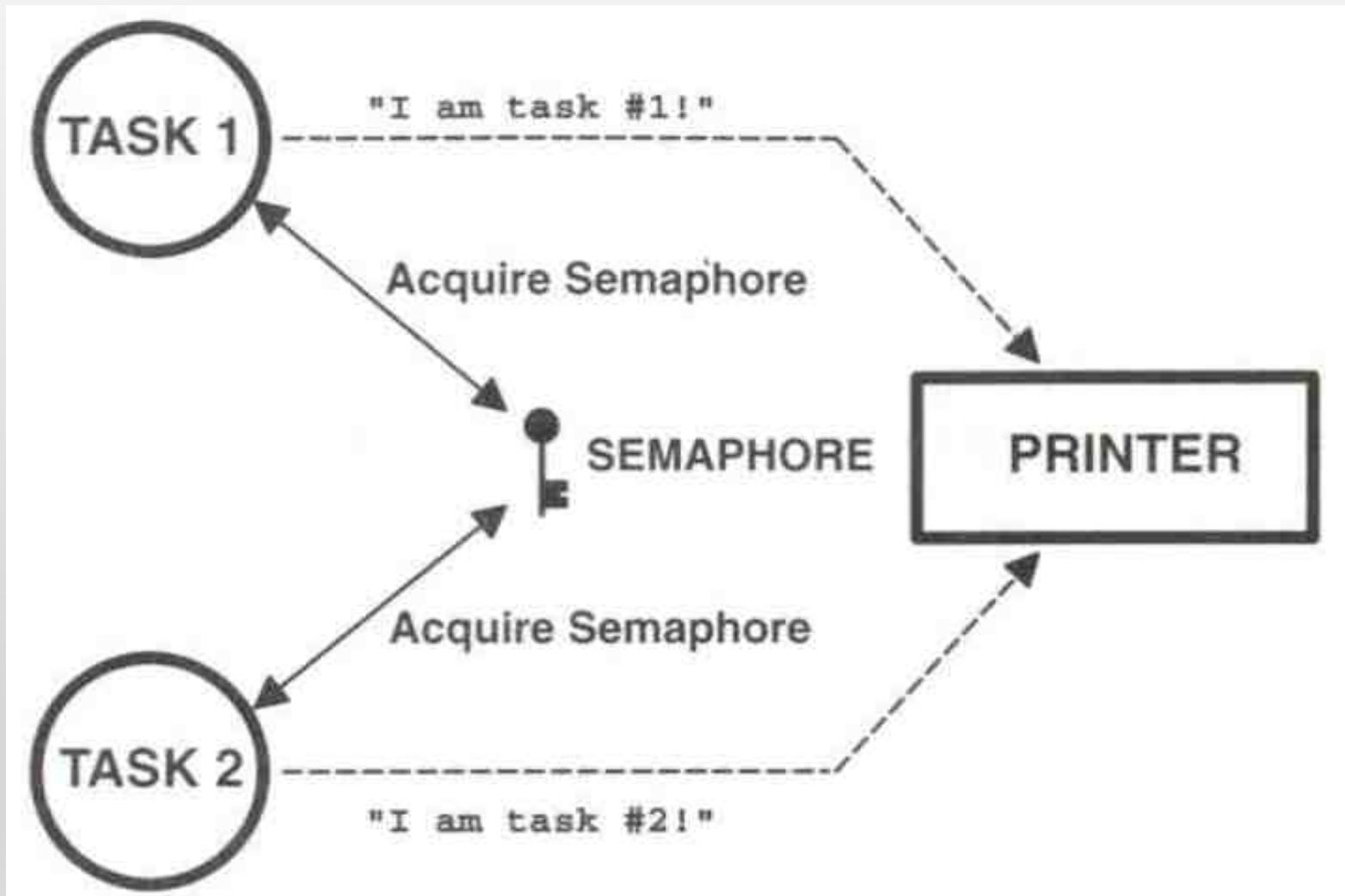
Applications of Binary Semaphores

- Suppose task 1 prints “I am Task 1!”
- Task 2 prints “I am Task 2!”
- If they were allowed to print at the same time, it could result in:

I la amm T Tasask k1!2!

- Solution:
 - Binary semaphore

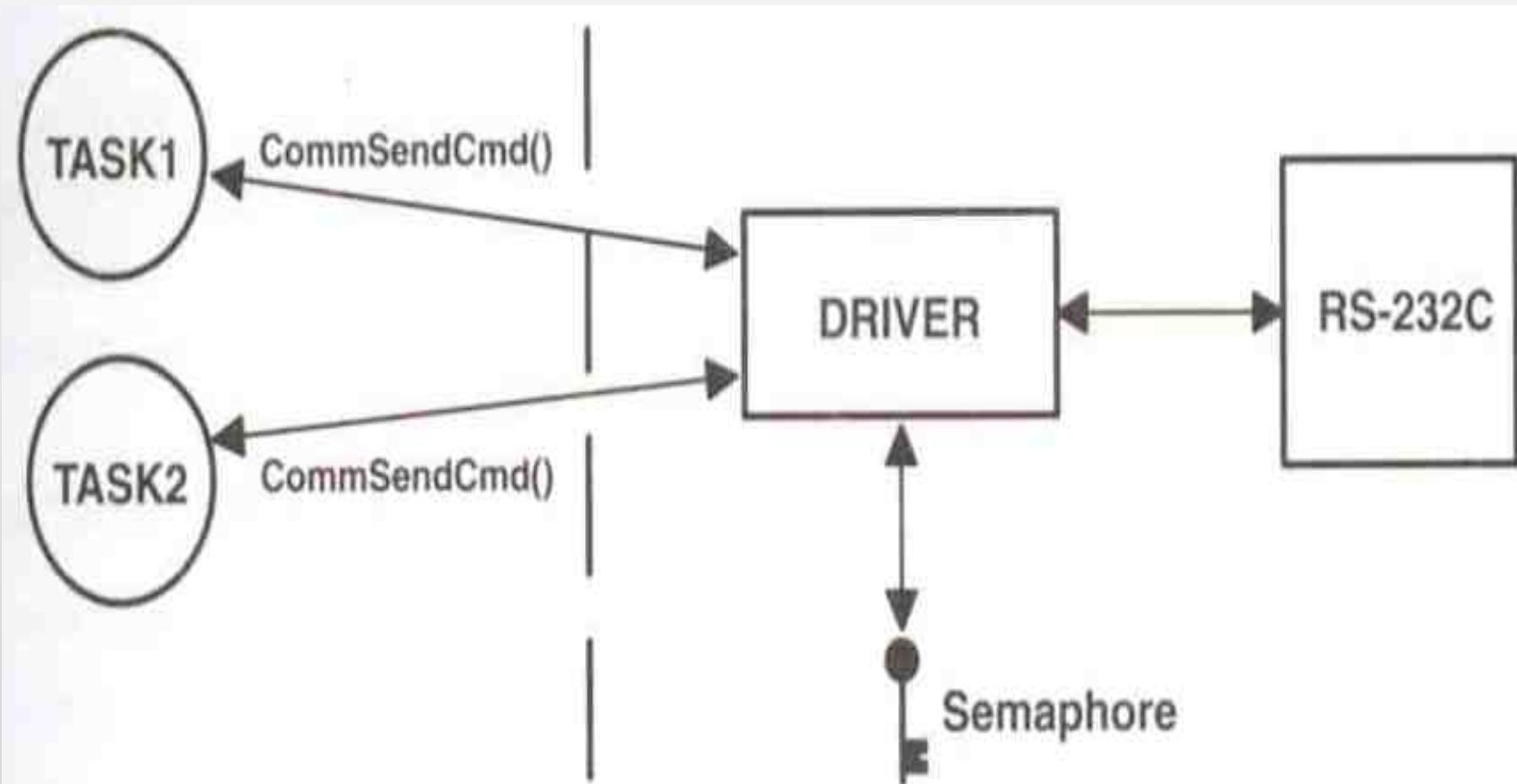
Sharing I/O Devices



Sharing I/O Device (cont.)

- In the example, each task must know about the semaphore in order to access the device
- A better solution:
 - Encapsulate the semaphore

Encapsulating a Semaphore



Encapsulating a Semaphore (cont.)

```
INT8U CommSendCmd(char *cmd, char *response, INT16U timeout)
```

```
{
```

```
    Acquire semaphore;
```

```
    Send command to device;
```

```
    Wait for response with timeout;
```

```
    if (timed out) {
```

```
        Release semaphore;
```

```
        return error code;
```

```
    } else {
```

```
        Release semaphore;
```

```
        return no error;
```

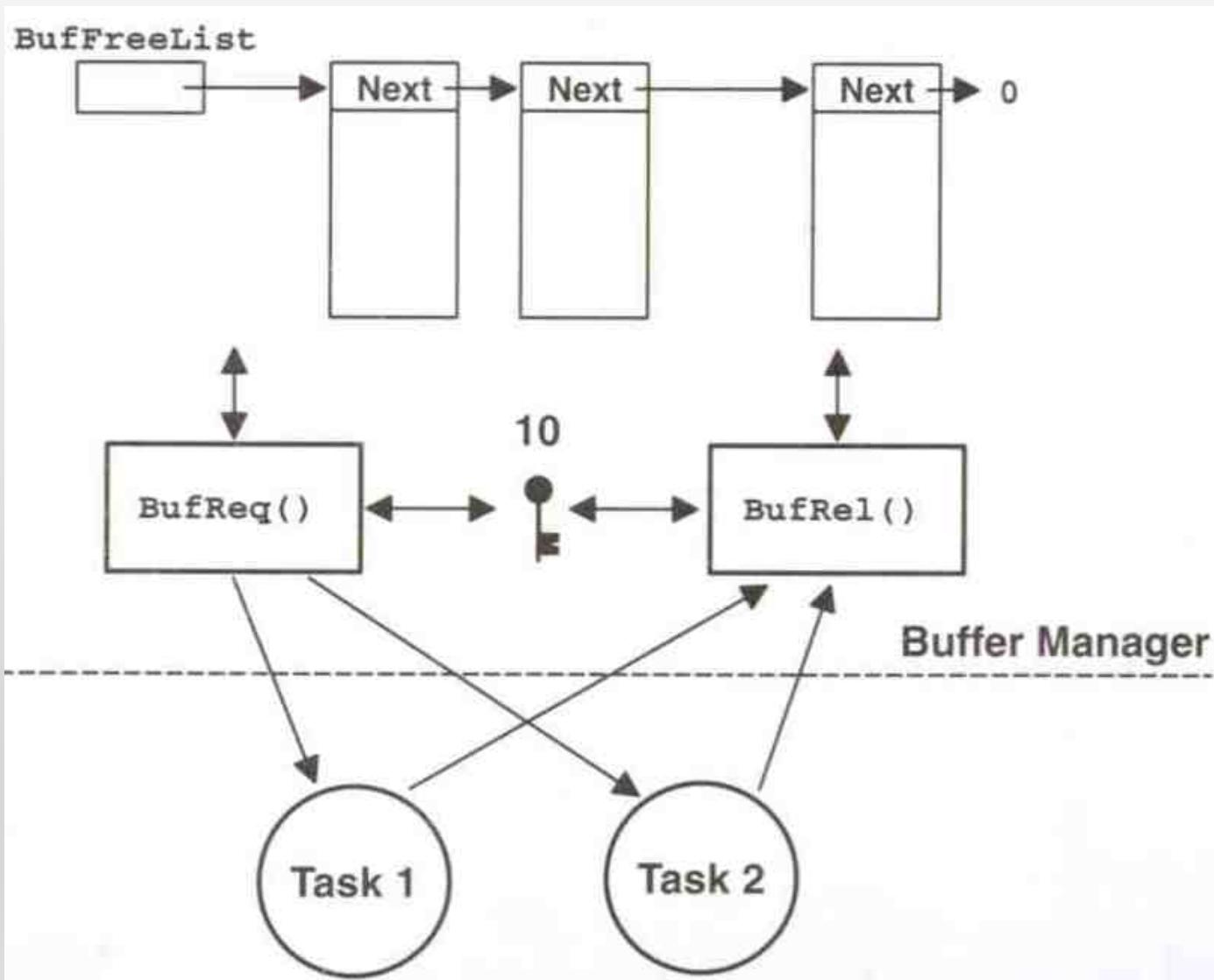
```
}
```

```
}
```

Applications of Counting Semaphores

- A counting semaphore is used when a resource can be used by more than one task at the same time
- Example:
 - Managing a buffer pool of 10 buffers

Buffer Management



Buffer Management (cont.)

```
BUF *BufReq(void)
```

```
{
```

```
    BUF *ptr;
```

```
    Acquire a semaphore;
```

```
    Disable interrupts;
```

```
    ptr = BufFreeList;
```

```
    BufFreeList = ptr->BufNext;
```

```
    Enable interrupts;
```

```
    return (ptr);
```

```
}
```

Buffer Management (cont.)

```
void BufRel(BUF *ptr)
{
    Disable interrupts;
    ptr->BufNext = BufFreeList;
    BufFreeList = ptr;
    Enable interrupts;
    Release semaphore;
}
```

Buffer Management (cont.)

- Semaphores are often overused
- The use of semaphore to access simple shared variable is overkill in most situations
- For this simple access, disabling interrupts are more cost-effective
- However, if the variable is floating-point and CPU does not support floating-point in hardware, disabling interrupts should be avoided

Linux for Real Time Applications

- Scheduling
 - Priority Driven Approach
 - Optimize average case response time
 - Interactive Processes Given Highest Priority
 - Aim to reduce response times of processes
 - Real Time Processes
 - Processes with high priority
 - There was no notion of deadlines

Linux for Real Time Applications (cont.)

- Resource Allocation
 - There was no support for handling priority inversion
 - Since version 2.6.18, priority inheritance available in both kernel and user space mutexes for preventing priority inversion

Interrupt Handling in Linux

- Interrupts were disabled in ISR/critical sections of the kernel
- There was no worst case bound on interrupt latency available
 - eg: Disk Drivers might disable interrupt for few hundred milliseconds

Interrupt Handling in Linux (cont.)

- Linux was not suitable for Real Time Applications
 - Interrupts may be missed
- Since 2.6.18, Hard IRQs executed in kernel thread context
 - (where differing priority levels can be assigned)
 - allows developers to better insulate systems from external events

Other Problems with Linux

- Processes were not preemptible in Kernel Mode
 - System calls like fork take a lot of time
 - High priority thread might wait for a low priority thread to complete it's system call
- Processes are heavy weight
 - Context switch takes several hundred microseconds
- Linux 2.6.18 adds preemption points in kernel, with the goal of achieving microsecond-level latency;
 - all but "kernel-critical" portions of kernel code become preemptible involuntarily at any time

Why Linux ?

- Coexistence of Real Time Applications with non Real Time Ones
 - Example http server
- Device Driver Base
- Stability

Why Linux? (cont.)

- Several real-time features are now integrated into the main distribution of Linux
 - Improved POSIX compliancy
 - including Linux's first implementation of message queues and of priority inheritance,
 - as well as an improved implementation of signals less apt to disregard multiple inputs
 - Hard IRQs executed in thread context

Why Linux? (cont.)

- Three levels of real-time preemptibility in kernel, configurable at compile time (throughput or real-time predictability):
 - Voluntary preemption
 - Preemptible kernel (ms level)
 - Full real-time preemption (microsecond level)
- Priority inheritance available in both kernel and user space mutexes

Why Linux? (cont.)

- High-resolution timers
 - Added in 2.6.21, this allows the kernel to actually use the high-resolution timers built into most processors, enabling,
 - For example, POSIX timers and `nano_sleep` calls to be "as accurate as hardware allows,"
 - The kernel's entire time management gets to the level of microseconds

Why Linux? (cont.)

- Various kernel config options for monitoring real-time behavior of kernel:
 - `CONFIG_LATENCY_TRACE`
 - Track the full trace of maximum latency
 - `CONFIG_WAKEUP_TIMING`
 - Tracking of the maximum recorded time between
 - waking up for a high-priority task, and
 - executing on the CPU, shown in microseconds



Q & A

Thank you for your attention.

HW01 is due today

Embedded System

Lecture 05: The Linux Kernel

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

Hong-Yue Technology Research Building 334
R 09:10 - 12:00

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>

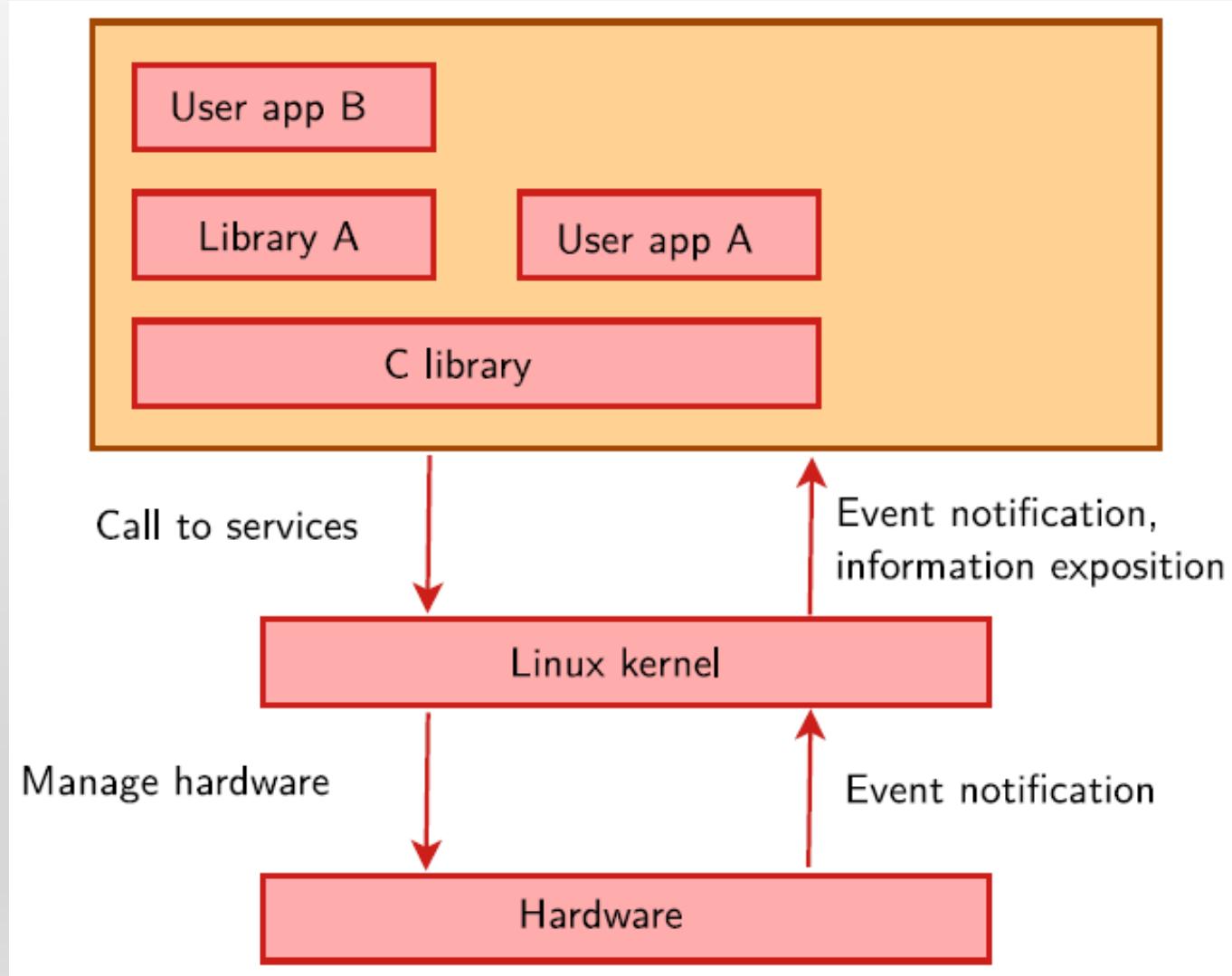
Linux Kernel Key Features

- 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

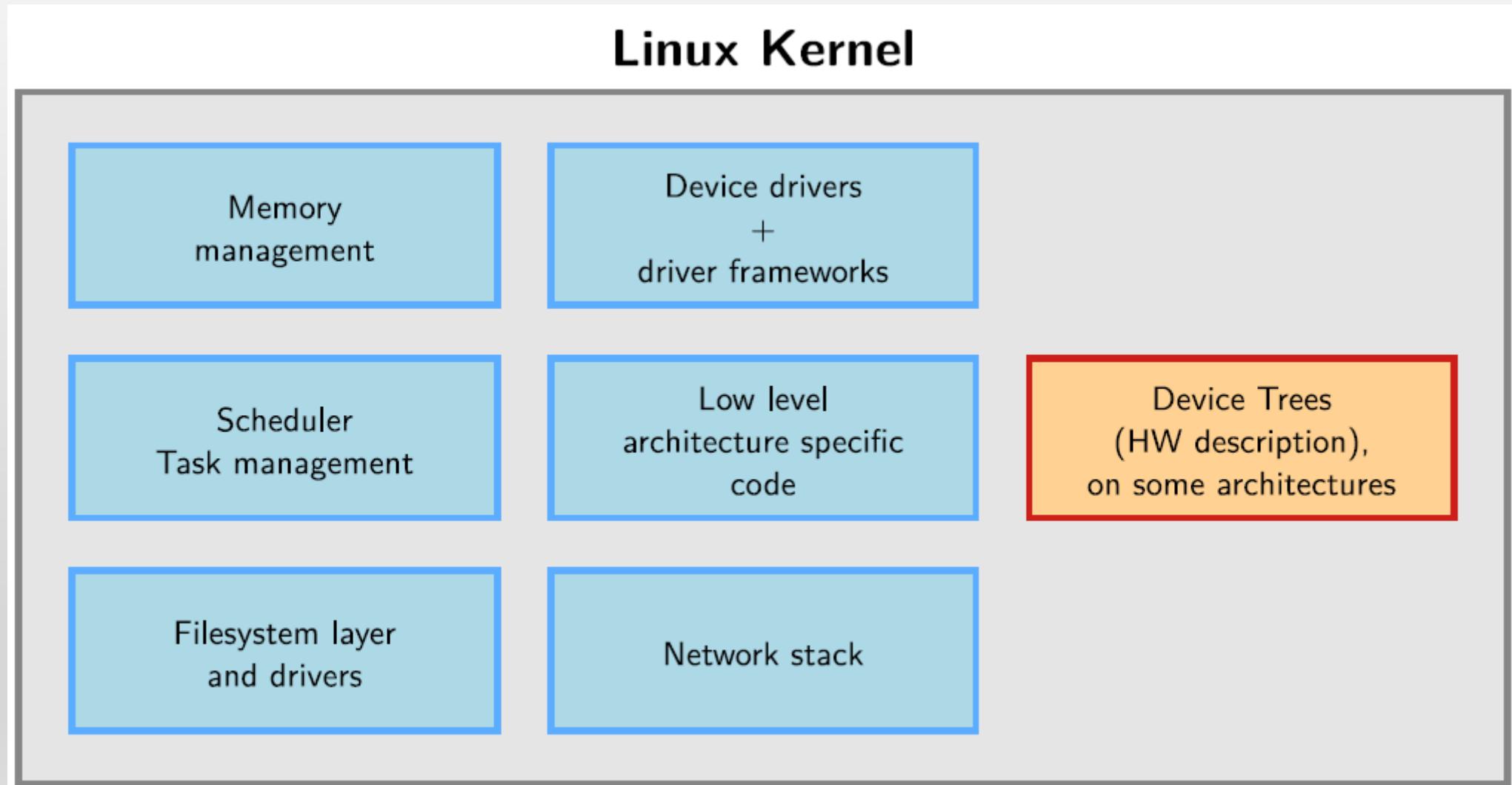
- The main interface between the kernel and user space is the set of system calls
- About 400 system calls that provide the main kernel Services
 - File and device operations, networking operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, 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



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.

no stable ABI over Linux kernel releases,
binaries are not portable

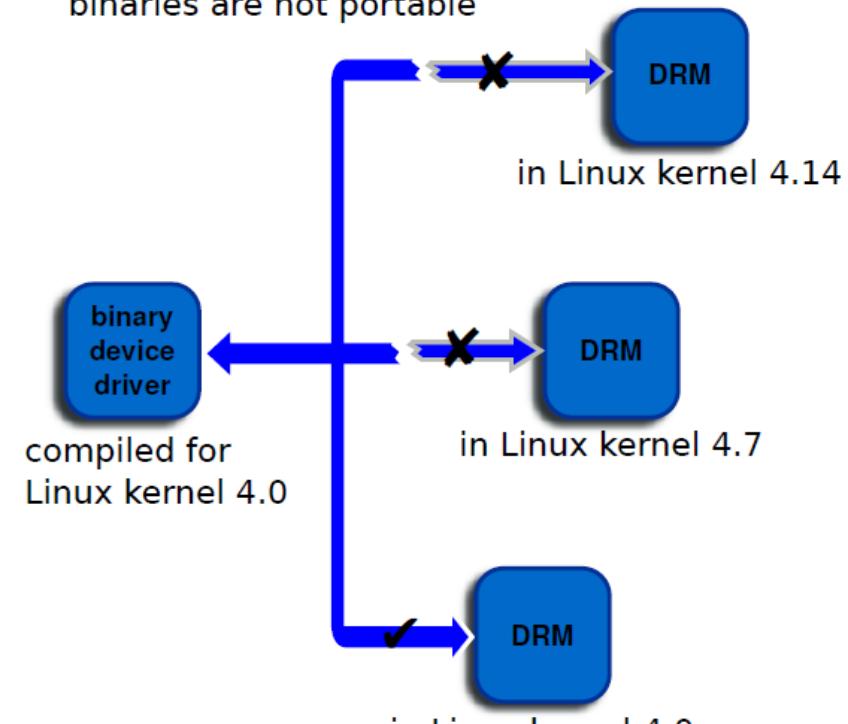


Image credits (Wikipedia):
<https://bit.ly/2U2rdGB>

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...)

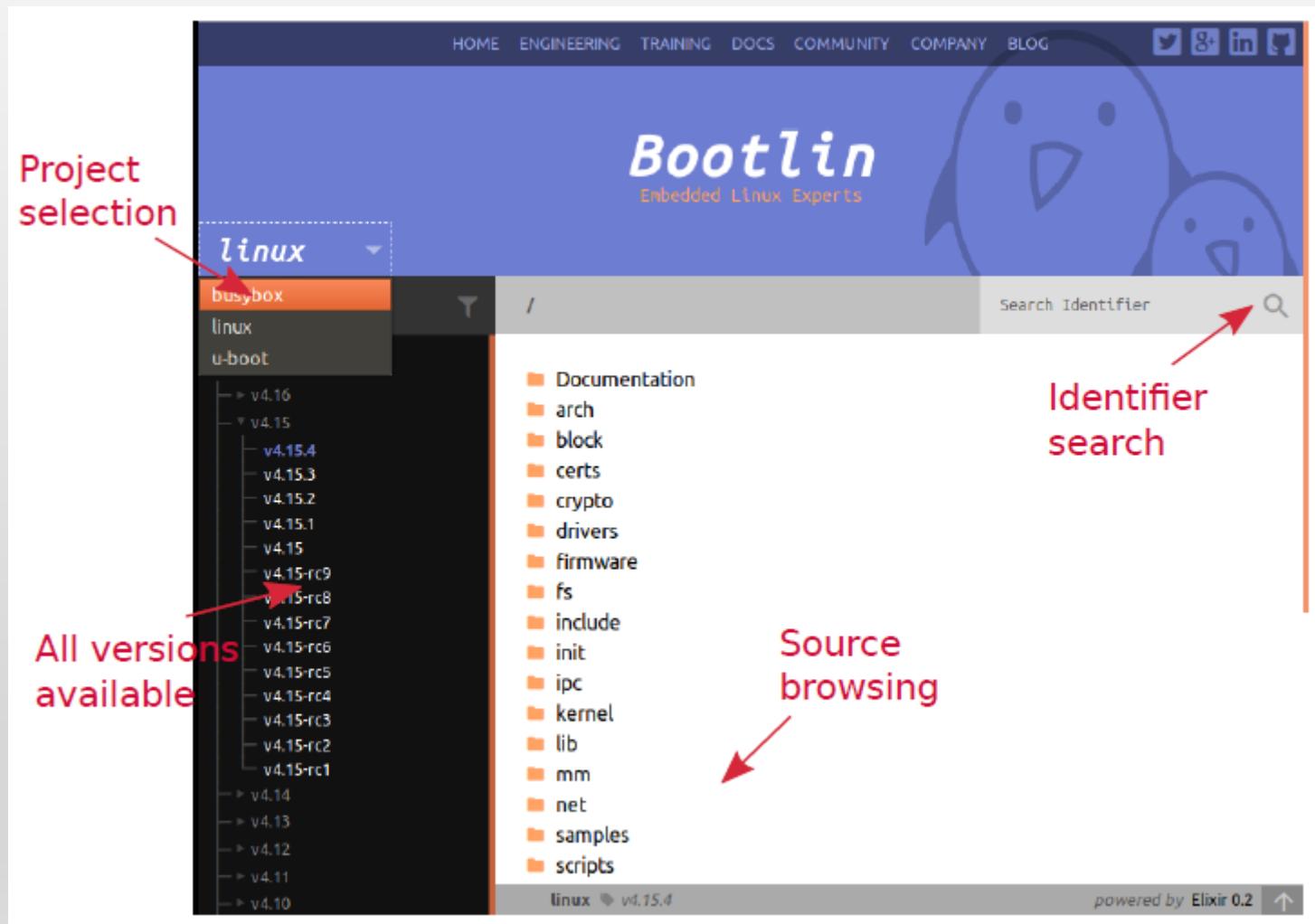
Time to Look at the Source Structure

- [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 ACPI tables (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 Source
- Older versions require a special kernel image format: ulmage
 - ulmage is generated from zImage using the mkimage tool. It is done automatically by the kernel make ulmage target.
 - On some ARM platforms, make ulmage 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 ulmage 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 (ulmage 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

```
// 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>

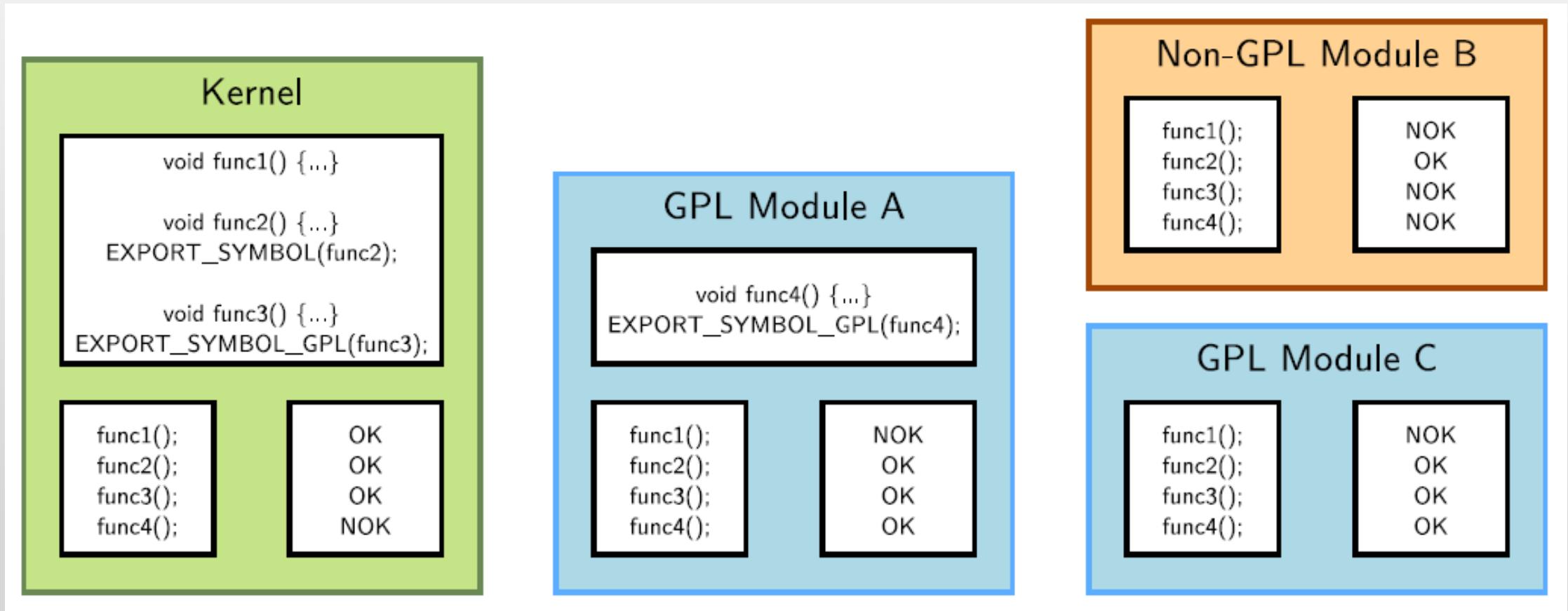
Hello Module Explanations

- 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

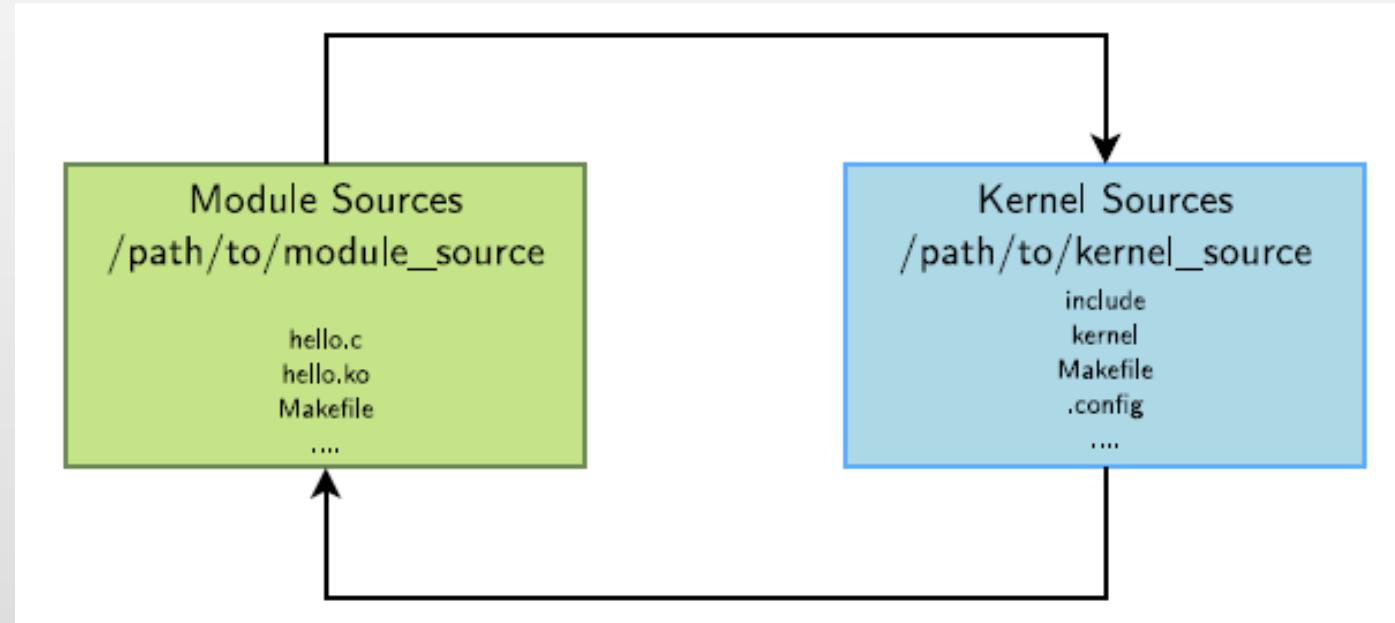
- The below **Makefile** should be reusable for any single-file out-of-tree Linux module
- The source file is **hello.c**
- Just run **make** to build the **hello.ko** file

```
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

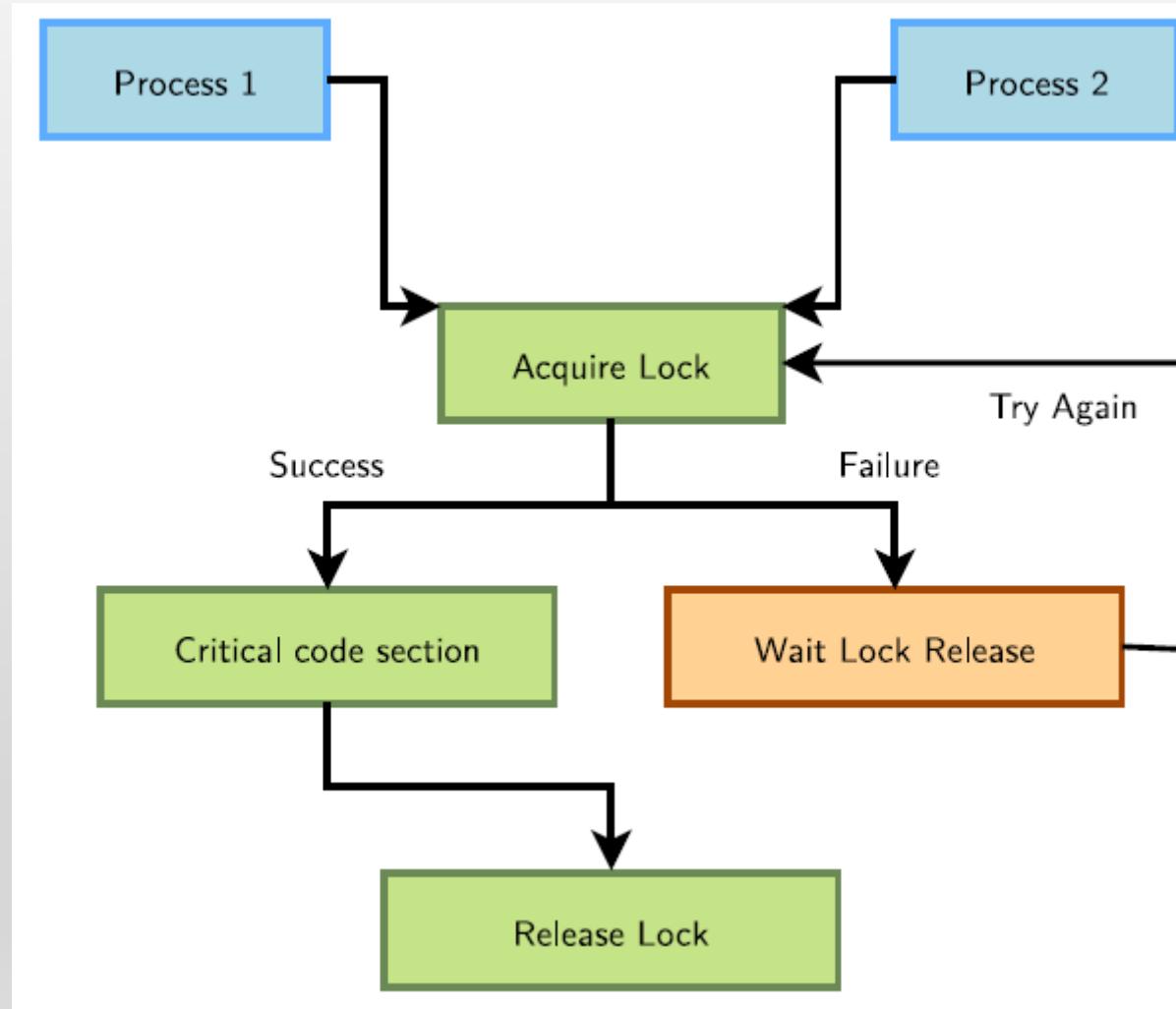


- The module **Makefile** is interpreted with **KERNELRELEASE** undefined, so it calls the kernel **Makefile**, passing the module directory in the **M** variable
- The kernel **Makefile** knows how to compile a module, and thanks to the **M** variable, knows where the **Makefile** for our module is. 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 = mutual exclusion
- 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.
- 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);
```

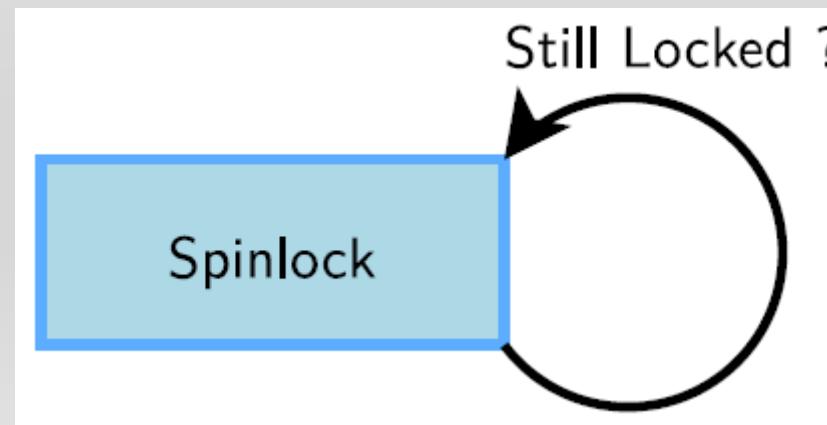
- Just tells whether the mutex is locked or not.

```
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

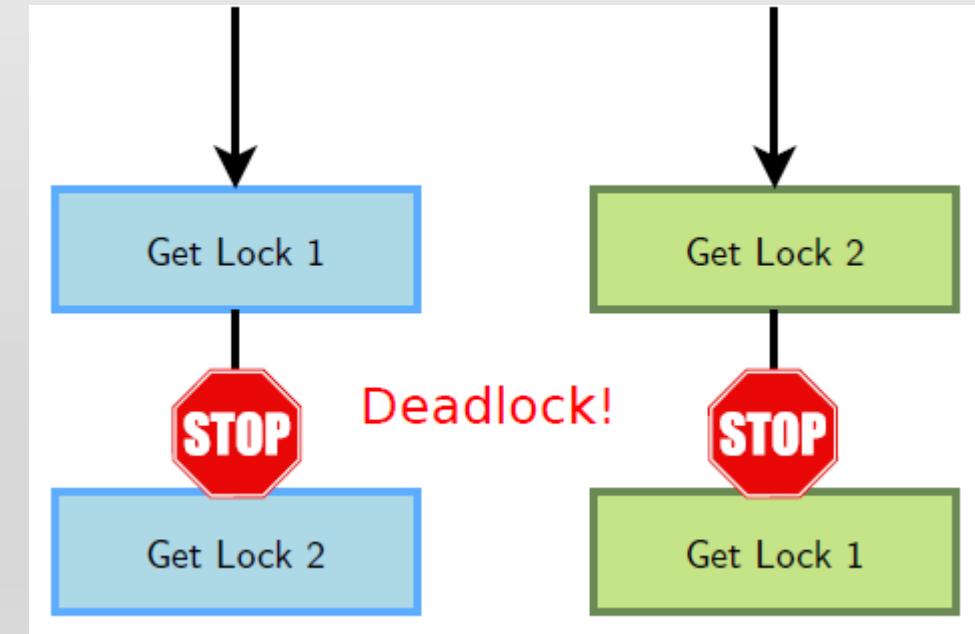
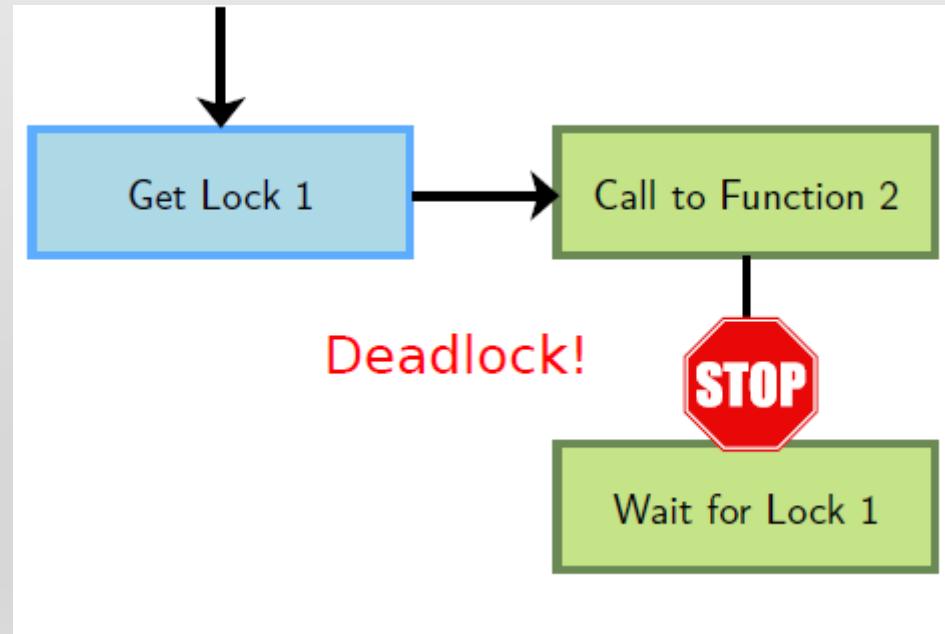
```
struct uart_port {  
    spinlock_t lock;  
    /* Other fields */  
};
```

- 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.

Embedded System

Lecture 06: Kernel Arch for Device Drivers & Kernel Initialization

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

Hong-Yue Technology Research Building 334
R 09:10 - 12:00

Kernel Arch for Device Drivers

- Userspace sees three main types of devices:
 1. **Character devices**
 - The most common type of devices.
 - Initially for devices implementing streams of bytes, it is now used for a wide range of devices: serial ports, framebuffers, video capture devices, sound devices, input devices, I2C and SPI gateways, etc.
 2. **Block devices**
 - for storage devices like hard disks, CD-ROM drives, USB keys, SD/MMC cards, etc.
 3. **Network devices**
 - for wired or wireless interfaces, network connections and others

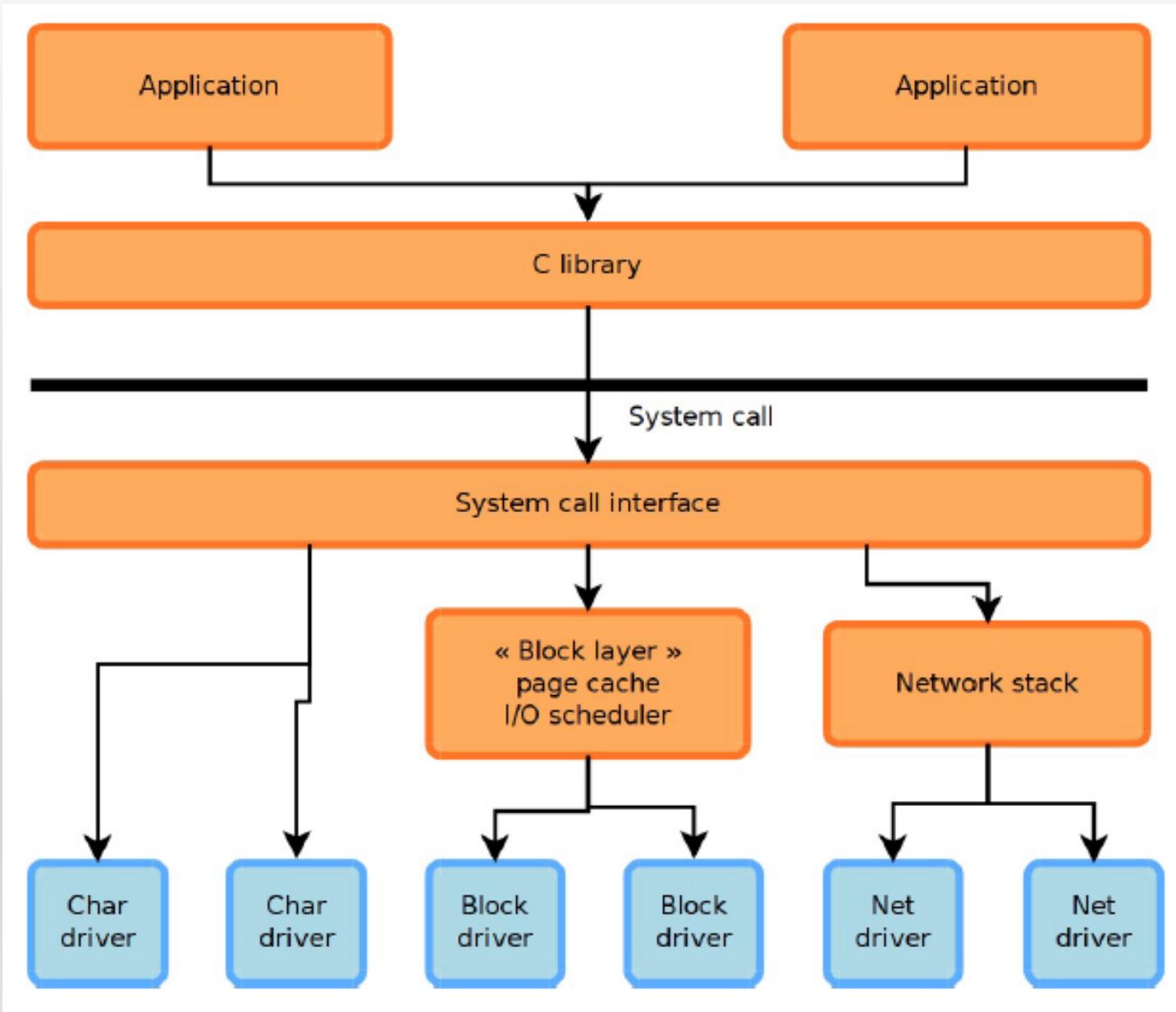
Accessing the devices

- Network devices are accessed through **network-specific APIs and tools** (socket API of the standard C library, tools such as ifconfig, route, etc.)
- Block and character devices are represented for userspace applications as files than can be manipulated using the traditional file API (open(), read(), write(), close(), etc.)
 - Special file types for block and character devices, associating a name with a couple (**major, minor**)
 - The kernel only cares about the (type, major, minor), which is the unique identifier of the device
 - Special files traditionally located in **/dev**, created by **mknod**, either manually or automatically by **udev**

Inside the Kernel

- Device drivers must register themselves to the core kernel and implement a set of operations specific to their type:
 - **Character drivers** must instantiate and register a `cdev` structure and implement `file_operations`
 - **Block drivers** must instantiate and register a `gendisk` structure and implement `block_device_operations` and a special `make_request` function
 - **Network drivers** must instantiate and register a `net_device` structure and implement `net_device_ops`
- For the following, we will first focus on character devices as an example of device drivers.

General architecture



File operations

- The file operations are generic to all types of files: regular files, directories, character devices, block devices, etc.

```
struct file_operations {  
    struct module *owner;  
    loff_t (*llseek) (struct file *, loff_t, int);  
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);  
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);  
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);  
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);  
    int (*readdir) (struct file *, void *, filldir_t);  
    unsigned int (*poll) (struct file *, struct poll_table_struct *);  
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);  
    int (*mmap) (struct file *, struct vm_area_struct *);  
    int (*open) (struct inode *, struct file *);  
    int (*flush) (struct file *, fl_owner_t id);  
    int (*release) (struct inode *, struct file *);  
    int (*fsync) (struct file *, struct dentry *, int datasync);  
    int (*fasync) (int, struct file *, int);  
    int (*flock) (struct file *, int, struct file_lock *);  
    [...]  
};
```

Character Driver Skeleton

- Implement the `read()` and `write()` operations, and instantiate the `file_operations` structure.

```
static ssize_t demo_write(struct file *f, const char __user *buf,
                         size_t len, loff_t *off)
{
    [...]
}

static ssize_t demo_read(struct file *f, char __user *buf,
                        size_t len, loff_t *off)
{
    [...]
}

static struct file_operations demo_fops =
{
    .owner = THIS_MODULE,
    .read = acme_read,
    .write = acme_write
};
```

Character Driver Skeleton

- Register and unregister the driver to the kernel using
 - `register_chrdev_region` & `unregister_chrdev_region`
 - `cdev_add` & `cdev_del`

```
static dev_t demo_dev = MKDEV(202,128);
static struct cdev demo_cdev;

static int __init demo_init(void)
{
    register_chrdev_region(demo_dev, 1, "demo");
    cdev_init(&demo_cdev, &demo_fops);
    cdev_add(&demo_cdev, demo_dev, demo_count);
}

static void __exit demo_exit(void)
{
    cdev_del(&demo_cdev);
    unregister_chrdev_region(demo_dev, 1);
    iounmap(demo_buf);
}

module_init(demo_init);
module_exit(demo_exit);
```

Driver Usage in Userspace

- Making it accessible to userspace application by creating a device node:
- `mknod /dev/demo c 202 128`
- Using normal the normal file API :

```
fd = open("/dev/demo", O_RDWR);  
  
ret = read(fd, buf, bufsize);  
  
ret = write(fd, buf, bufsize);
```

From the Syscall to Your Driver

- In `fs/read write.c`

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
    struct file *file;
    ssize_t ret = -EBADF;
    int fput_needed;

    file = fget_light(fd, &fput_needed);
    if (file) {
        loff_t pos = file_pos_read(file);
        ret = vfs_read(file, buf, count, &pos);
        file_pos_write(file, pos);
        fput_light(file, fput_needed);
    }

    return ret;
}
```

From the Syscall to Your Driver

- In `fs/read/write.c`

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;

    if (!(file->f_mode & FMODE_READ))
        return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
        return -EFAULT;

    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }

    return ret;
}
```

ioctl mechanism

- The **file operations** set of operations, while being sufficient for regular files, isn't sufficient as an API to the wide range of character and block devices
- Device-specific operations such as changing the speed of a serial port, setting the volume on a soundcard, configuring video-related parameters on a framebuffer **are not** handled by the file operations
- One of the operations, **ioctl()** allows to extend the capabilities of a driver with driver-specific operations
- In user space: **int ioctl(int d, int request, ...);**
 - d, the file descriptor
 - request, a driver-specific integer identifying the operation
 - ..., zero or one argument.
- In kernel space: **int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);**

Kernel

- Implement the demo `ioctl()` operation and reference it in the `file_operations` structure:

```
static int demo_ioctl(struct inode *inode,
                      struct file *file,
                      unsigned int cmd,
                      unsigned long arg)

{
    char __user *argp = (char __user *)arg;

    switch (cmd) {

        case DEMO_CMD1:
            /* Something */
            return 0;

        default:
            return -ENOTTY;
    }
}

static const struct file_operations demo_fops =
{
    [...]
    .ioctl = demo_ioctl,
    [...]
};
```

ioctl example, userspace side

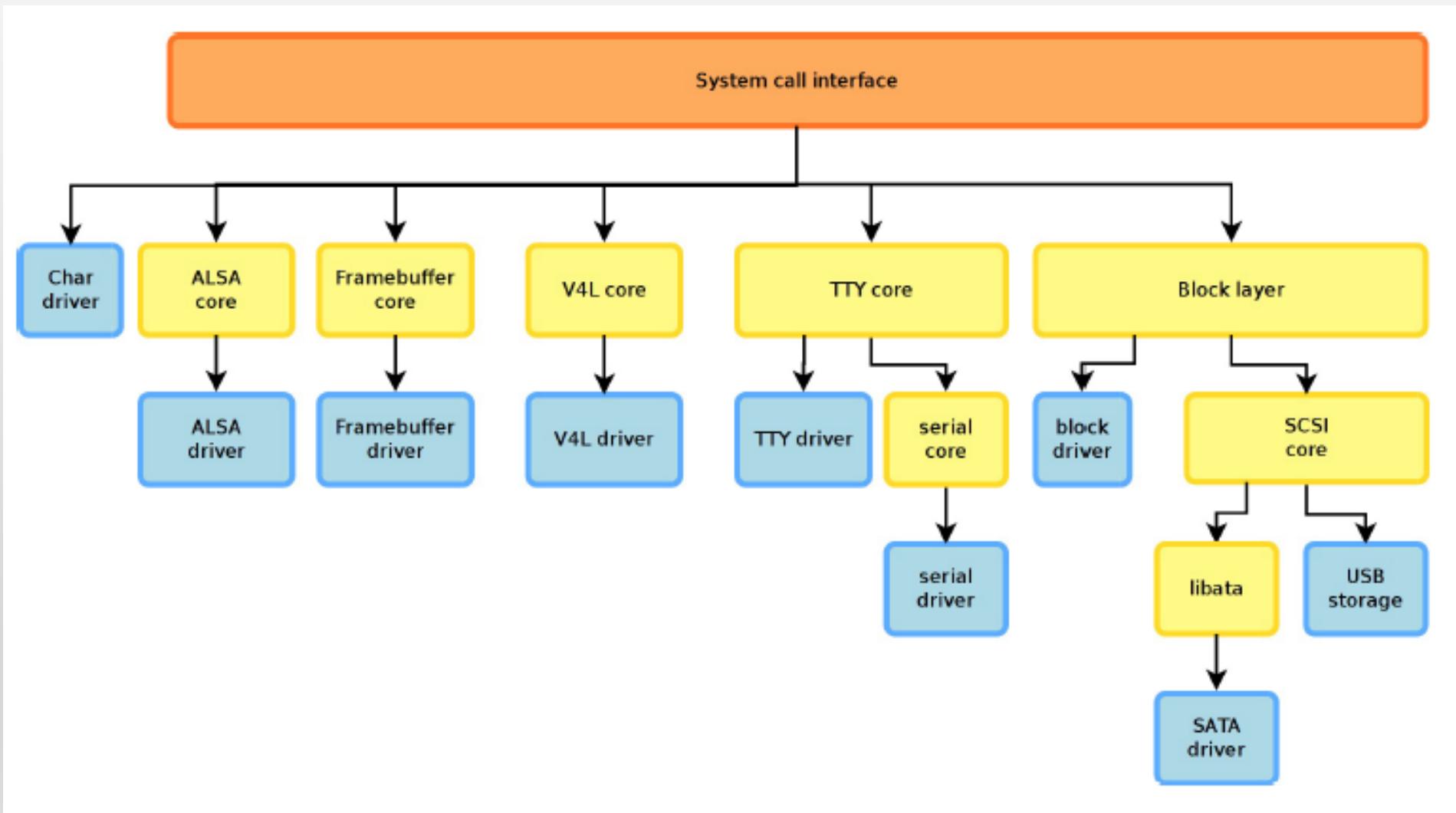
- Use the `ioctl()` system call.

```
int fd, val;  
  
fd = open("/dev/demo", O_RDWR);  
  
ioctl(fd, DEMO_CMD1, & val);
```

Kernel Framework

- Most device drivers are not directly implemented as character devices or block devices
- They are implemented under a **framework**, specific to a device type (framebuffer, V4L, serial, etc.)
 - The framework allows to factorize the common parts of drivers for the same type of devices
 - From user space, they are still seen as normal character devices
 - The framework allows to provide a coherent user space interface (ioctl numbering and semantic, etc.) for every type of device, regardless of the driver

Example of frameworks



Example of the framebuffer framework

- Kernel option `CONFIG_FB`
- Implemented in `drivers/video/`
 - `fb.c`, `fbmem.c`, `fbmon.c`, `fbcmap.c`, `fbsysfs.c`, `modedb.c`, `fbcvt.c`
- Implements a single character driver (through file operations), registers the major number and allocates minors, allocates and implements the user/kernel API
 - First part of `include/linux/fb.h`
- Defines the set of operations a framebuffer driver must implement and helper functions for the drivers
 - struct `fb_ops`
 - Second part of `include/linux/fb.h`

The framebuffer Driver

- Must implement some or all operations defined in `struct fb_ops`. Those operations are framebuffer-specific.
 - `xxx_open()`, `xxx_read()`, `xxx_write()`, `xxx_release()`, `xxx_checkvar()`, `xxx_setpar()`, `xxx_setcolreg()`, `xxx_blank()`, `xxx_pan_display()`, `xxx_fillrect()`, `xxx_copyarea()`, `xxx_imageblit()`, `xxx_cursor()`, `xxx_rotate()`, `xxx_sync()`, `xxx_get_caps()`, etc.
- Must allocate a `fb_info` structure with `framebuffer_alloc()`, set the `->fbops` field to the operation structure, and register the framebuer device with `register framebuffer()`

Skeleton example

```
static int xxx_open(struct fb_info *info, int user) {}
static int xxx_release(struct fb_info *info, int user) {}
static int xxx_check_var(struct fb_var_screeninfo *var, struct fb_info *info) {}
static int xxx_set_par(struct fb_info *info) {}

static struct fb_ops xxx_ops = {
    .owner          = THIS_MODULE,
    .fb_open        = xxxfb_open,
    .fb_release     = xxxfb_release,
    .fb_check_var   = xxxfb_check_var,
    .fb_set_par     = xxxfb_set_par,
    [...]
};

init()
{
    struct fb_info *info;
    info = framebuffer_alloc(sizeof(struct xxx_par), device);
    info->fbops = &xxxfb_ops;
    [...]
    register_framebuffer(info);
}
```

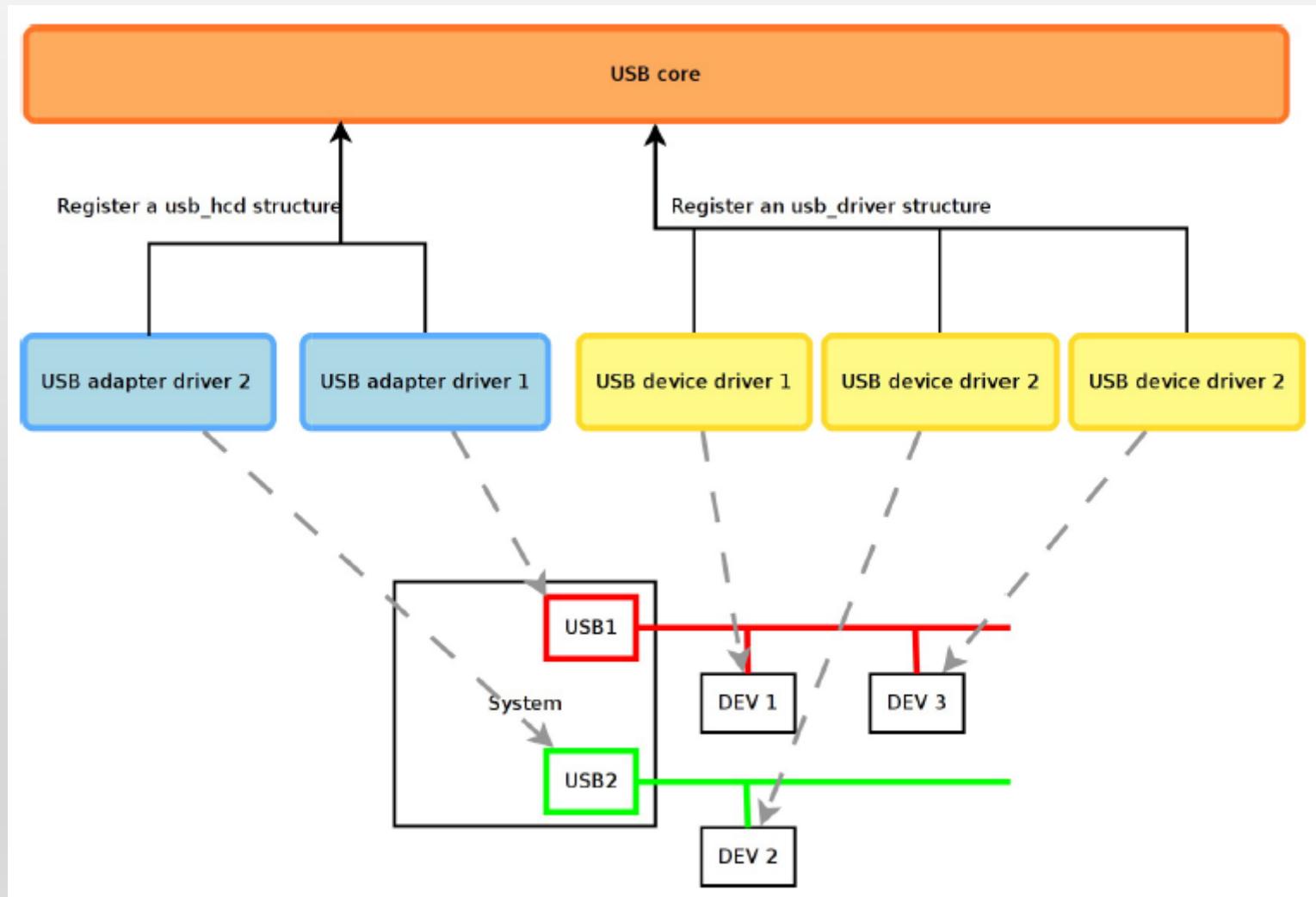
Other example of framework: serial driver

- 1. The driver registers a single `uart_driver` structure, that contains a few informations such as major, starting minor, number of supported serial ports, etc.
 - Functions `uart register driver()` and `uart unregister driver()`
- 2. For each serial port detected, the driver registers a `uart_port` structure, which points to a `uart_ops` structure and contains other informations about the serial port
 - Functions `uart_add_one_port()` and `uart_remove_one_port()`
- 3. The driver implements some or all of the methods in the `uart ops` structure
 - `tx empty()`, `set mcrl()`, `get mcrl()`, `stop tx()`, `start tx()`, `send xchar()`, `stop rx()`, `enable ms()`, `break ctl()`, `startup()`, `shutdown()`, `flush buffer()`, `set termios()`, etc.
 - All these methods receive as argument at least a `uart port` structure, the device on which the method applies. It is similar to the `this` pointer in object-oriented languages

Device and Driver model

- One of the features that came with the 2.6 kernel is a **unified device and driver model**
- Instead of different ad-hoc mechanisms in each subsystem, the device model unies the vision of the devices, drivers, their organization and relationships
- Allows to minimize code duplication, provide common facilities, more coherency in the code organization
- Defines base structure types: struct device, struct driver, struct bus type
- Is visible in **userspace** through the **sysfs** filesystem, traditionally mounted under **/sys**

Adapter, Bus and Device Drivers



Example of Device Driver

- To illustrate how drivers are implemented to work with the device model, we will use an USB network adapter driver. We will therefore limit ourselves to **device drivers** and won't cover adapter drivers.

Device Identifiers

- Defines the set of devices that this driver can manage, so that the USB core knows which devices this driver can handle.
- The `MODULE_DEVICE_TABLE` macro allows `depmod` to extract at compile time the relation between device identifiers and drivers, so that drivers can be loaded automatically by udev.
- See `/lib/modules/$(uname -r)/modules.{alias,usbmapg}`

```
static struct usb_device_id rtl8150_table[] = {
    {USB_DEVICE(VENDOR_ID_REALTEK, PRODUCT_ID_RTL8150)},
    {USB_DEVICE(VENDOR_ID_MELCO, PRODUCT_ID_LUAKTX)},
    {USB_DEVICE(VENDOR_ID_MICRONET, PRODUCT_ID_SP128AR)},
    {USB_DEVICE(VENDOR_ID_LONGSHINE, PRODUCT_ID_LCS8138TX)},
    {USB_DEVICE(VENDOR_ID_OQO, PRODUCT_ID_RTL8150)},
    {USB_DEVICE(VENDOR_ID_ZYXEL, PRODUCT_ID_PRESTIGE)},
    {}
};

MODULE_DEVICE_TABLE(usb, rtl8150_table);
```

Device identifiers

- Instantiates the `usb_driver` structure. This structure is a specialization of `struct_driver` defined by the driver model. We have an example of inheritance here.

```
static struct usb_device_id rtl8150_table[] = {
    {USB_DEVICE(VENDOR_ID_REALTEK, PRODUCT_ID_RTL8150)},
    {USB_DEVICE(VENDOR_ID_MELCO, PRODUCT_ID_LUAKTX)},
    {USB_DEVICE(VENDOR_ID_MICRONET, PRODUCT_ID_SP128AR)},
    {USB_DEVICE(VENDOR_ID_LONGSHINE, PRODUCT_ID_LCS8138TX)},
    {USB_DEVICE(VENDOR_ID_OQO, PRODUCT_ID_RTL8150)},
    {USB_DEVICE(VENDOR_ID_ZYXEL, PRODUCT_ID_PRESTIGE)},
    {}
};

MODULE_DEVICE_TABLE(usb, rtl8150_table);
```

Instantiation of usb_driver

- Instantiates the `usb_driver` structure. This structure is a specialization of `struct driver` defined by the driver model. We have an example of inheritance here.

```
static struct usb_driver rtl8150_driver = {  
    .name =          "rtl8150",  
    .probe =         rtl8150_probe,  
    .disconnect =   rtl8150_disconnect,  
    .id_table =     rtl8150_table,  
    .suspend =      rtl8150_suspend,  
    .resume =       rtl8150_resume  
};
```

Registration of the Driver

- When the driver is loaded and unloaded, it simply registers and unregisters itself as an USB device driver.

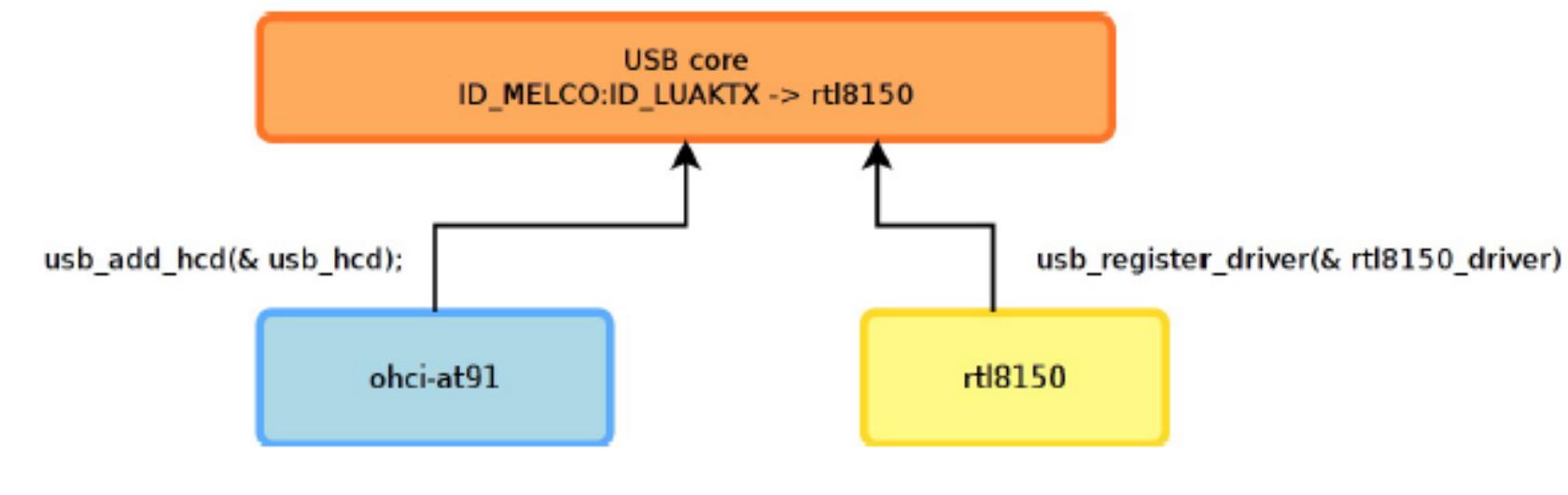
```
static int __init usb_rtl8150_init(void)
{
    return usb_register(&rtl8150_driver);
}

static void __exit usb_rtl8150_exit(void)
{
    usb_deregister(&rtl8150_driver);
}

module_init(usb_rtl8150_init);
module_exit(usb_rtl8150_exit);
```

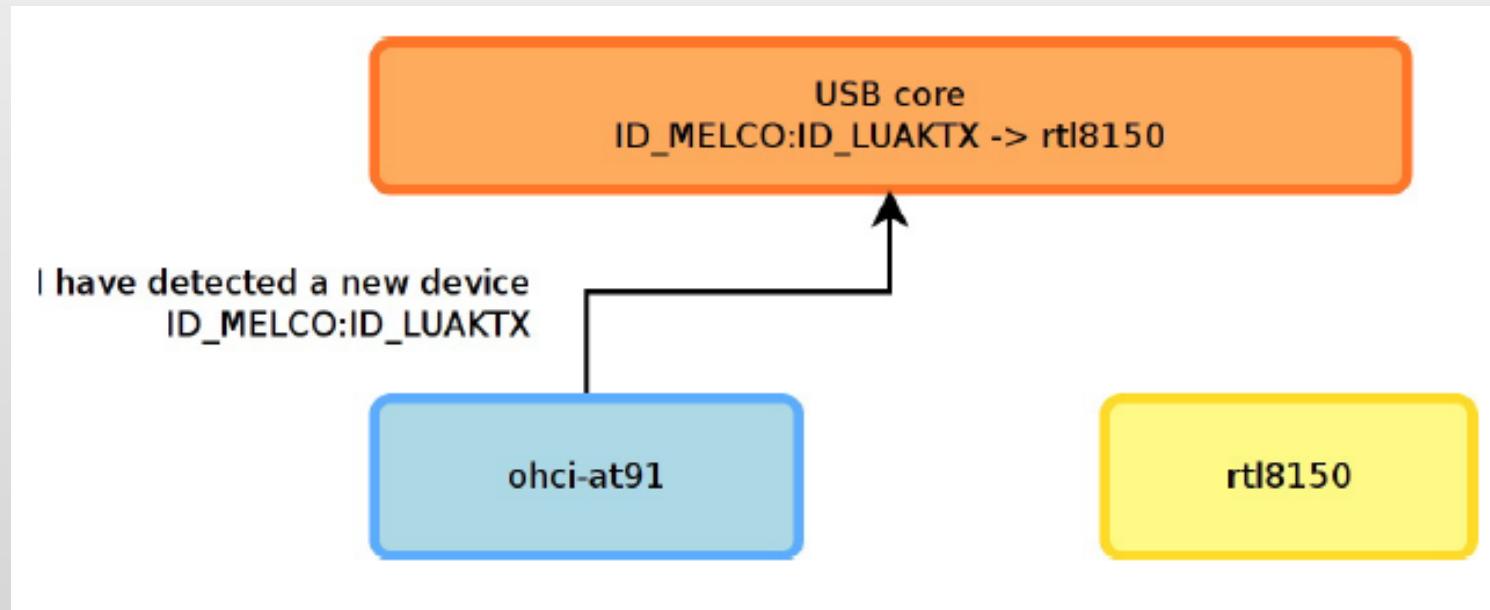
Probe Call Sequence (1/3)

- At boot time, the USB device driver registers itself to the generic BUS infrastructure



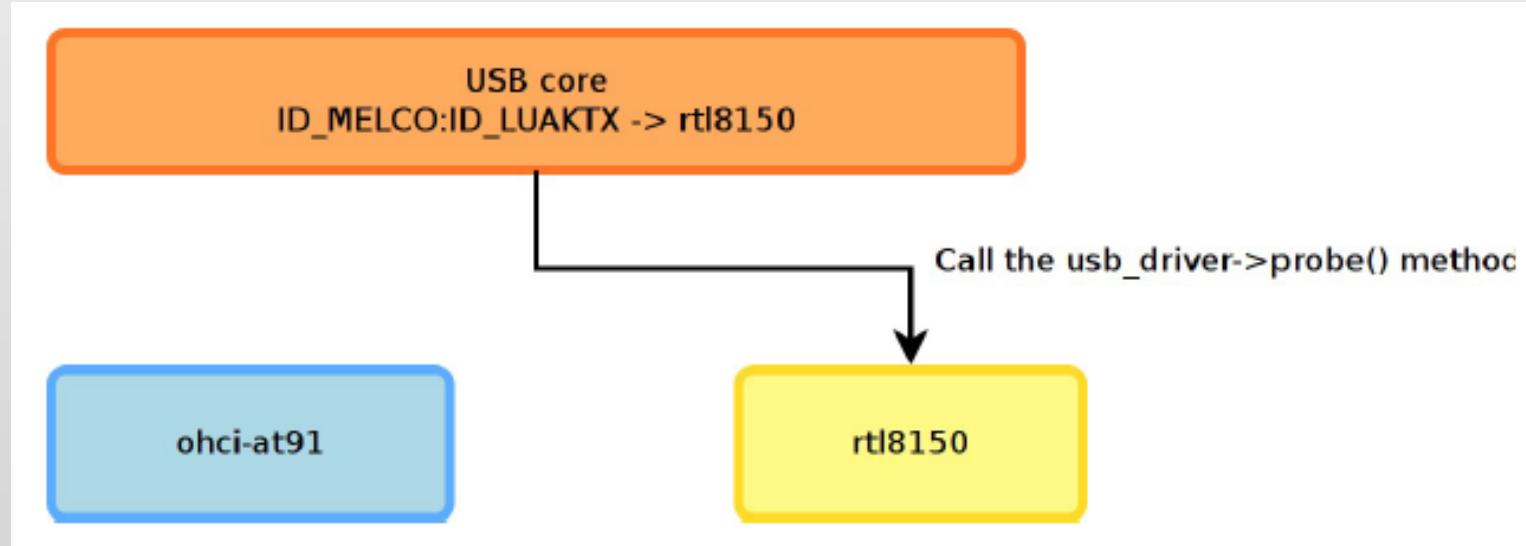
Probe Call Sequence (2/3)

- When a bus adapter driver detects a device, it notifies the generic USB bus infrastructure



Probe Call Sequence (3/3)

- The generic USB bus infrastructure knows which driver is capable of handling the detected device. It calls the `probe()` method of that driver



Probe Method

- The `probe()` method receives as argument a structure describing the device, usually specialized by the bus infrastructure (`pci_dev`, `usb_interface`, etc.)
- This function is responsible for
 - Initializing the device, mapping I/O memory, registering the interrupt handlers. The bus infrastructure provides methods to get the addresses, interrupts numbers and other device-specific information.
 - Registering the device to the proper kernel framework, for example the network infrastructure.

rtl8150 probe

```
static int rtl8150_probe(struct usb_interface *intf,
                         const struct usb_device_id *id)
{
    rtl8150_t *dev;
    struct net_device *netdev;

    netdev = alloc_etherdev(sizeof(rtl8150_t));

    dev = netdev_priv(netdev);
    tasklet_init(&dev->tl, rx_fixup, (unsigned long)dev);
    spin_lock_init(&dev->rx_pool_lock);

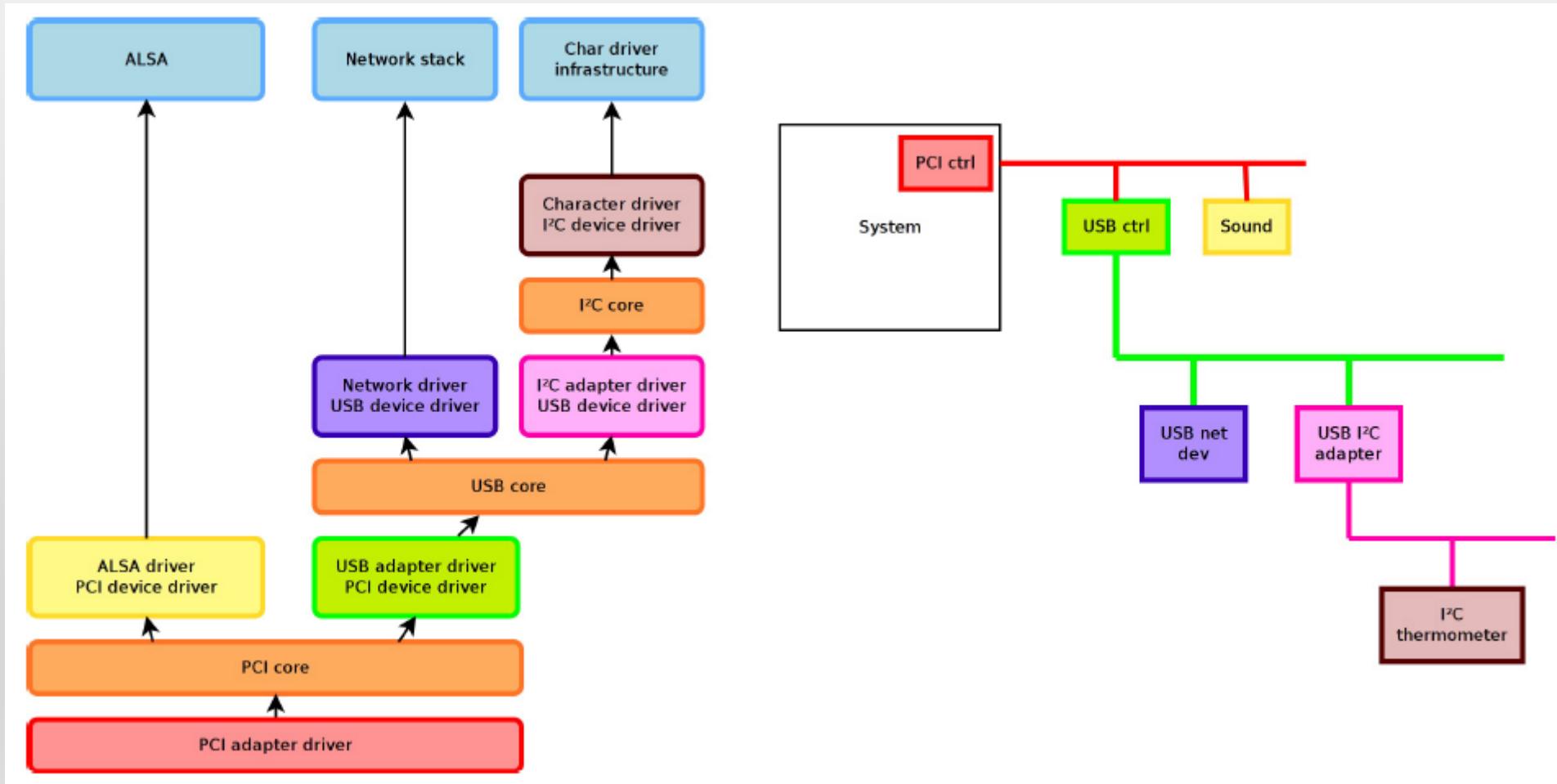
    netdev->netdev_ops = &rtl8150_netdev_ops;
    alloc_all_urbs(dev);

    usb_set_intfdata(intf, dev);
    SET_NETDEV_DEV(netdev, &intf->dev);
    register_netdev(netdev);

    return 0;
}
```

Device Model is Recursive

- Drivers can be connected to another driver



Platform Drivers

- On embedded systems, devices are often not connected through a bus allowing enumeration, hotplugging, and providing unique identifiers for devices.
- However, we still want the devices to be part of the device model.
- The solution to this is the platform driver / platform device infrastructure.
- The platform devices are the devices that are directly connected to the CPU, without any kind of bus.

Initialization of a Platform Driver

- Example of the iMX serial port driver, in [drivers/serial/imx.c](#). The driver instantiates a platform driver structure:

```
static struct platform_driver serial_imx_driver = {
    .probe          = serial_imx_probe,
    .remove         = serial_imx_remove,
    .driver         = {
        .name      = "imx-uart",
        .owner     = THIS_MODULE,
    },
};
```

- And registers/unregisters it at init/cleanup:

```
static int __init imx_serial_init(void)
{
    platform_driver_register(&serial_imx_driver);
}
static void __exit imx_serial_cleanup(void)
{
    platform_driver_unregister(&serial_imx_driver);
}
```

Initialization of a Platform Device

- As platform devices cannot be detected dynamically, they are statically defined:
 - by direct instantiation of `platform_device` structures, as done on ARM
 - by using a `device tree`, as done on PowerPC
- Example on ARM, where the instantiation is done in the board specific code ([\(arch/arm/mach-imx/mx1ads.c\)](#))

```
static struct platform_device imx_uart1_device = {
    .name          = "imx-uart",
    .id            = 0,
    .num_resources = ARRAY_SIZE(imx_uart1_resources),
    .resource      = imx_uart1_resources,
    .dev = {
        .platform_data = &uart_pdata,
    }
};
```

- The matching between a device and the driver is simply done using the name.

Registration of Platform Devices

- The device is part of a list:

```
static struct platform_device *devices[] __initdata = {  
    &cs89x0_device,  
    &imx_uart1_device,  
    &imx_uart2_device,  
};
```

- And the list of devices is added to the system during the board initialization

```
static void __init mx1ads_init(void)  
{  
    [...]  
    platform_add_devices(devices, ARRAY_SIZE(devices));  
    [...]  
}  
  
MACHINE_START(MX1ADS, "Freescale MX1ADS")  
    [...]  
    .init_machine = mx1ads_init,  
MACHINE_END
```

The resource mechanism

- Each device managed by a particular driver typically uses different hardware resources: different addresses for the I/O registers, different DMA channel, different IRQ line, etc.
- These information can be represented using the kernel [struct resource](#), and an array of resources is associated to a platform device definition.

```
static struct resource imx_uart1_resources[] = {
    [0] = {
        .start  = 0x00206000,
        .end    = 0x002060FF,
        .flags  = IORESOURCE_MEM,
    },
    [1] = {
        .start  = (UART1_MINT_RX),
        .end    = (UART1_MINT_RX),
        .flags  = IORESOURCE_IRQ,
    },
};
```

The platform_data mechanism

- In addition to the well-defined resources, some driver require driver-specific configuration for each platform device
- These can be specified using the `platform_data` field of the `struct device`
- As it is a `void *` pointer, it can be used to pass any type of data to the driver
- In the case of the iMX driver, the platform data is a `struct imxuart_platform_data` structure, referenced from the `platform_device` structure

```
static struct imxuart_platform_data uart_pdata = {  
    .flags = IMXUART_HAVE_RTSCTS,  
};
```

Driver-specific data structure

- Typically, device drivers **subclass** the type-specific data structure that they must instantiate to register their device to the upper layer framework
- For example, serial drivers subclass [uart_port](#), network drivers subclass [netdev](#), framebuffer drivers subclass [fb_info](#)
- This **inheritance** is done by aggregation or by reference

```
struct imx_port {  
    struct uart_port        port;  
    struct timer_list       timer;  
    unsigned int            old_status;  
    int                     txirq,rxirq,rtsirq;  
    unsigned int            have_rtscts:1;  
    unsigned int            use_irda:1;  
    unsigned int            irda_inv_rx:1;  
    unsigned int            irda_inv_tx:1;  
    unsigned short          trcv_delay; /* transceiver delay */  
    struct clk              *clk;  
};
```

probe() method for Platform Devices

- Just like the usual `probe()` methods, it receives the `platform_device` pointer, uses different utility functions to find the corresponding resources, and registers the device to the corresponding upper layer.

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imx_port *sport;
    struct imxuart_platform_data *pdata;
    void __iomem *base;
    struct resource *res;

    sport = kzalloc(sizeof(*sport), GFP_KERNEL);
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    base = ioremap(res->start, PAGE_SIZE);

    sport->port.dev = &pdev->dev;
    sport->port.mapbase = res->start;
    sport->port.membase = base;
    sport->port.type = PORT_IMX,
    sport->port.iotype = UPIO_MEM;
    sport->port.irq = platform_get_irq(pdev, 0);
    sport->rxirq = platform_get_irq(pdev, 0);
    sport->txirq = platform_get_irq(pdev, 1);
    sport->rtsirq = platform_get_irq(pdev, 2);

    [...]
```

probe() method for Platform Devices

```
    sport->port.fifosize = 32;
    sport->port.ops = &imx_pops;

    sport->clk = clk_get(&pdev->dev, "uart");
    clk_enable(sport->clk);
    sport->port.uartclk = clk_get_rate(sport->clk);

    imx_ports[pdev->id] = sport;

    pdata = pdev->dev.platform_data;
    if (pdata && (pdata->flags & IMXUART_HAVE_RTSCTS))
        sport->have_rtscts = 1;

    ret = uart_add_one_port(&imx_reg, &sport->port);
    if (ret)
        goto deinit;
    platform_set_drvdata(pdev, &sport->port);

    return 0;
}
```

Other non-dynamic busses

- In addition to the special platform bus, there are some other busses that do not support dynamic enumeration and identification of devices. For example: I2C and SPI.
- For these busses, a list of devices connected to the bus is hardcoded into the board-specific information and is registered using `i2c_register_board_info()` or `spi_register_board_info()`. The binding between the device is also done using a string identifier.

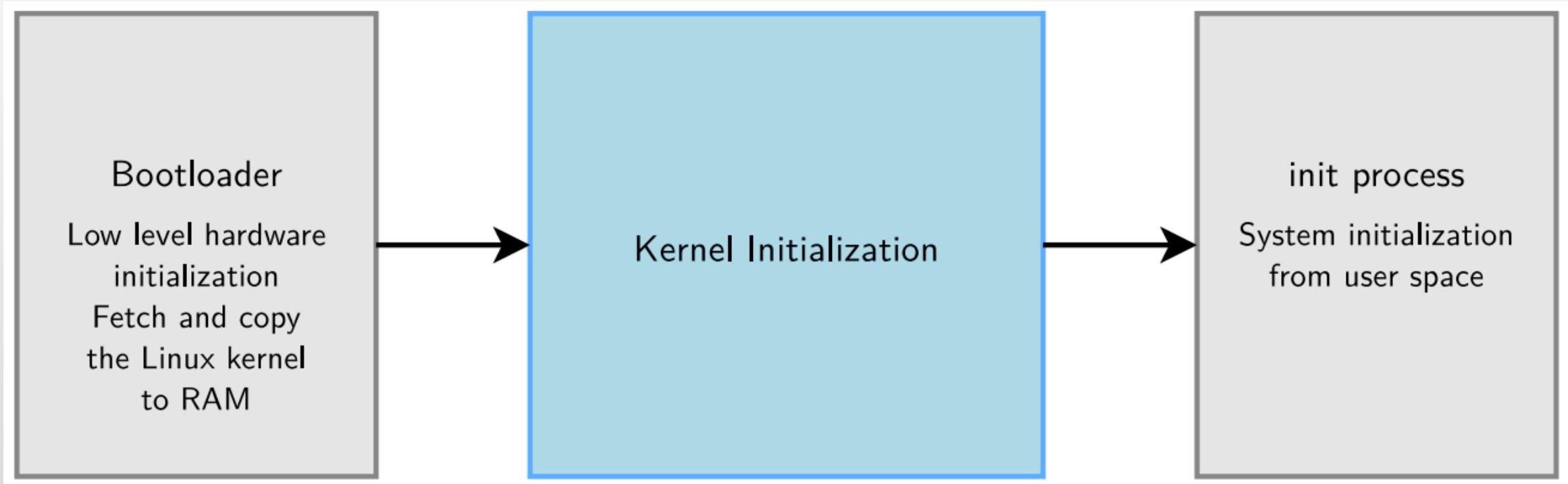
```
static struct i2c_board_info pcm038_i2c_devices[] = {
    {
        I2C_BOARD_INFO("at24", 0x52),
        .platform_data = &board_eeprom,
    },
    {
        I2C_BOARD_INFO("pcf8563", 0x51),
    },
    {
        I2C_BOARD_INFO("lm75", 0x4a),
    }
};

static void __init pcm038_init(void) {
    [...]
    i2c_register_board_info(0, pcm038_i2c_devices,
                           ARRAY_SIZE(pcm038_i2c_devices));
    [...]
}
```

Typical Organization of a Driver

- A driver typically
 - Defines a **driver-specific data structure** to keep track of per-device state, this structure often subclass the type-specific structure for this type of device
 - Implements a set of **helper functions**, interrupt handlers, etc.
 - Implements some or all of the **operations**, as specified by the framework in which the device will be subscribed
 - Instantiate the **operation table**
 - Defines a **probe()** method that allocates the "state" structure, initializes the device and registers it to the upper layer framework. Similarly defines a corresponding **remove()** method
 - Instantiate a **SOMEBUS_driver structure** that references the **probe()** and **remove()** methods and give the bus infrastructure some way of binding a device to this driver (by name, by identifier, etc.)
 - In the **driver initialization function**, register as a device driver to the bus-specific infrastructure. In the **driver cleanup function**, unregister from the bus-specific infrastructure

Kernel Initialization : From Bootloader to User Space



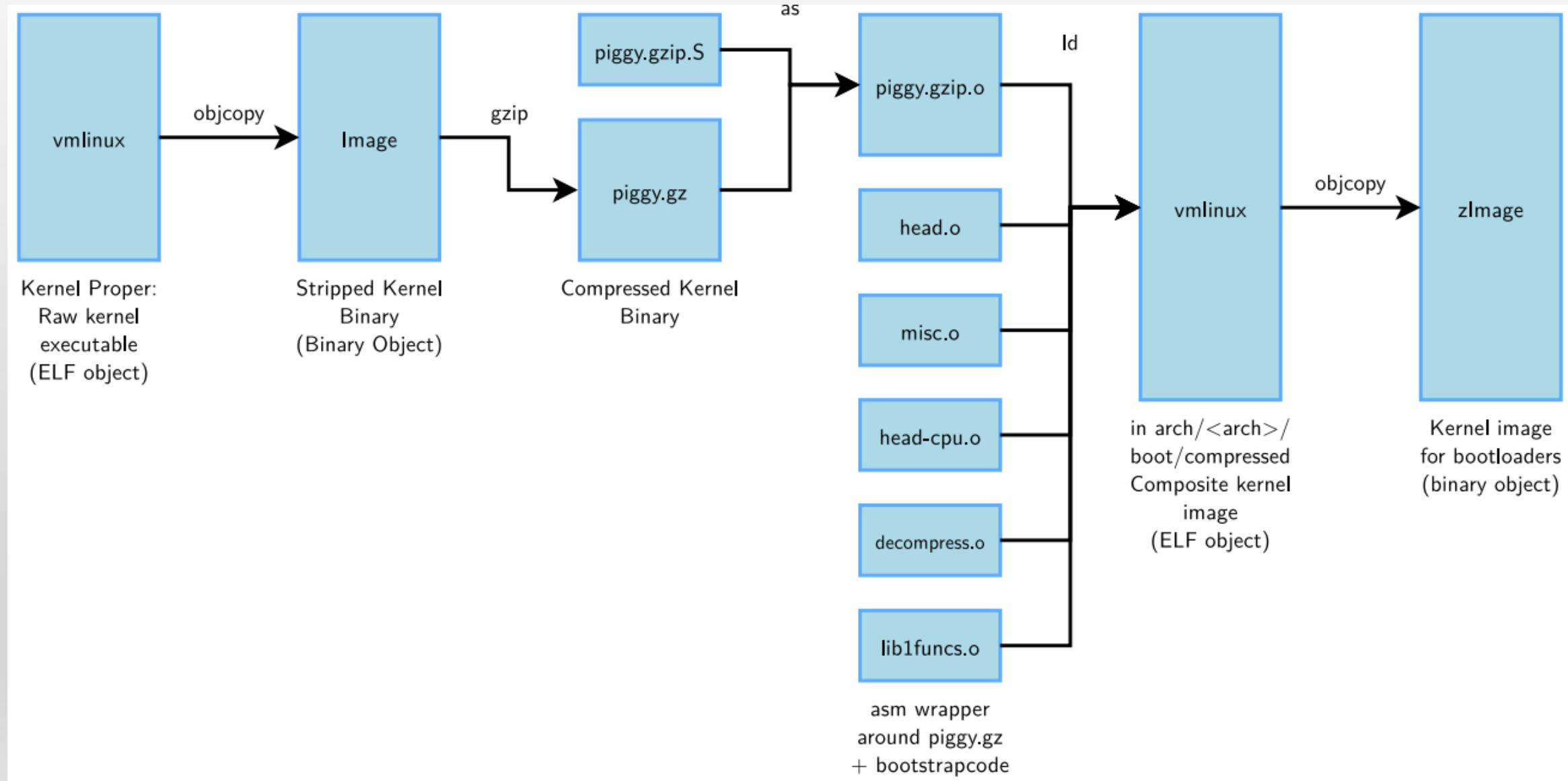
- Upon power-on, the bootloader in an embedded system is the first software to get processor control for low-level hardware initialization.
- Then, the control is passed to the Linux kernel

Kernel Bootstrap

- How the kernel bootstraps itself appears in kernel building. Example on ARM (pxa cpu) in Linux 2.6.36:
 - `make ARCH=arm CROSS_COMPILE=xscale_be- zImage`

```
... < many build steps omitted for clarity>
LD      vmlinux
SYSMAP System.map
SYSMAP .tmp_System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
AS      arch/arm/boot/compressed/head.o
GZIP    arch/arm/boot/compressed/piggy.gz
AS      arch/arm/boot/compressed/piggy.o
CC      arch/arm/boot/compressed/misc.o
AS      arch/arm/boot/compressed/head-xscale.o
AS      arch/arm/boot/compressed/big-endian.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```

Kernel Bootstrap



Bootstrap Code for Compressed Kernels

- Located in `arch/<arch>/boot/compressed`
 - `head.o`
 - Architecture specific initialization code.
 - This is what is executed by the bootloader
 - `head-cpu.o (here head-xscale.o)`
 - CPU specific initialization code
 - `decompress.o, misc.o`
 - Decompression code
 - `piggy.o`
 - The kernel itself
 - Responsible for uncompressing the kernel itself and jumping to its entry point.

Architecture-specific Initialization Code

- The uncompression code jumps into the main kernel entry point, typically located in [arch/<arch>/kernel/head.S](#), whose job is to:
 - Check the architecture, processor and machine type.
 - Configure the MMU, create page table entries and enable virtual memory.
 - Calls the `start_kernel` function in `init/main.c`.
 - Same code for all architectures.
 - Anybody interested in kernel startup should study this file!

start_kernel Main Actions

- Calls `setup_arch(&command_line)`
 - Function defined in `arch/<arch>/kernel/setup.c`
 - Copying the command line from where the bootloader left it.
 - On arm, this function calls `setup_processor` (in which CPU information is displayed) and `setup_machine`(locating the machine in the list of supported machines).
- Initializes the console as early as possible (to get error messages)
- Initializes many subsystems (see the code)
- Eventually calls `rest_init`.

rest_init: Starting the Init Process

```
static __attribute__((noinline)) void __init_refok rest_init(void)
    __releases(kernel_lock)
{
    int pid;

    rcu_scheduler_starting();
    /*
     * We need to spawn init first so that it obtains pid 1, however
     * the init task will end up wanting to create kthreads, which, if
     * we schedule it before we create kthreadd, will OOPS.
     */
    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    numa_default_policy();
    pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
    rcu_read_lock();
    kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
    rcu_read_unlock();
    complete(&kthreadd_done);

    /*
     * The boot idle thread must execute schedule()
     * at least once to get things moving:
     */
    init_idle_bootup_task(current);
    preempt_enable_no_resched();
    schedule();
    preempt_disable();

    /* Call into cpu_idle with preempt disabled */
    cpu_idle();
}
```

kernel_init

- `kernel_init` does two main things:
 - Call `do_basic_setup`

```
static void __init do_basic_setup(void)
{
    cpuset_init_smp();
    usermodehelper_init();
    init_tmpfs();
    driver_init();
    init_irq_proc();
    do_ctors();
    do_initcalls();
}
```

- Once kernel services are ready, start device initialization (Linux 2.6.36 code excerpt):
- Call `init_post`

do_initcalls

- Calls pluggable hooks registered with the macros below. Advantage: the generic code doesn't have to know about them

```
/*
 * A "pure" initcall has no dependencies on anything else, and purely
 * initializes variables that couldn't be statically initialized.
 *
 * This only exists for built-in code, not for modules.
 */
#define pure_initcall(fn)           __define_initcall("0",fn,1)

#define core_initcall(fn)           __define_initcall("1",fn,1)
#define core_initcall_sync(fn)      __define_initcall("1s",fn,1s)
#define postcore_initcall(fn)       __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn)  __define_initcall("2s",fn,2s)
#define arch_initcall(fn)           __define_initcall("3",fn,3)
#define arch_initcall_sync(fn)      __define_initcall("3s",fn,3s)
#define subsys_initcall(fn)          __define_initcall("4",fn,4)
#define subsys_initcall_sync(fn)    __define_initcall("4s",fn,4s)
#define fs_initcall(fn)              __define_initcall("5",fn,5)
#define fs_initcall_sync(fn)        __define_initcall("5s",fn,5s)
#define rootfs_initcall(fn)         __define_initcall("rootfs",fn,rootfs)
#define device_initcall(fn)          __define_initcall("6",fn,6)
#define device_initcall_sync(fn)    __define_initcall("6s",fn,6s)
#define late_initcall(fn)             __define_initcall("7",fn,7)
#define late_initcall_sync(fn)      __define_initcall("7s",fn,7s)
```

Defined in include/linux/init.h

initcall example

From arch/arm/mach-pxa/lpd270.c (Linux 2.6.36)

```
static int __init lpd270_irq_device_init(void)
{
    int ret = -ENODEV;
    if (machine_is_logicpd_pxa270()) {
        ret = sysdev_class_register(&lpd270_irq_sysclass);
        if (ret == 0)
            ret = sysdev_register(&lpd270_irq_device);
    }
    return ret;
}

device_initcall(lpd270_irq_device_init);
```

init_post

- The last step of Linux booting
 - First tries to open a console
 - Then tries to run the init process, effectively turning the current kernel thread into the user space init process

init_post Code: init/main.c

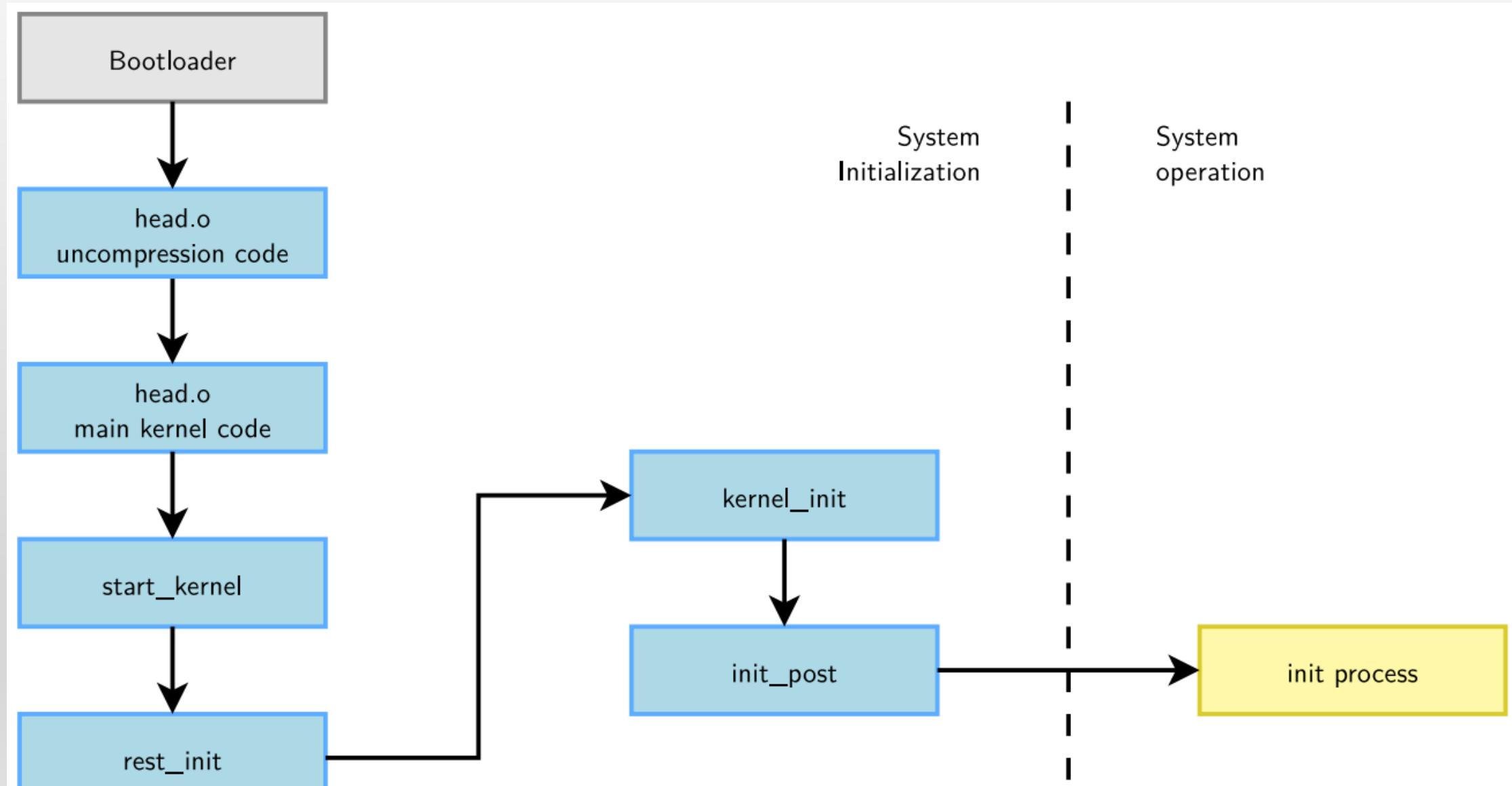
```
static noinline int init_post(void) __releases(kernel_lock) {
    /* need to finish all async __init code before freeing the memory */
    async_synchronize_full();
    free_initmem();
    mark_rodata_ro();
    system_state = SYSTEM_RUNNING;
    numa_default_policy();

    current->signal->flags |= SIGNAL_UNKILLABLE;
    if (ramdisk_execute_command) {
        run_init_process(ramdisk_execute_command);
        printk(KERN_WARNING "Failed to execute %s\n", ramdisk_execute_command);
    }

    /* We try each of these until one succeeds.
     * The Bourne shell can be used instead of init if we are
     * trying to recover a really broken machine. */
    if (execute_command) {
        run_init_process(execute_command);
        printk(KERN_WARNING "Failed to execute %s. Attempting defaults...\n", execute_command);
    }
    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");

    panic("No init found. Try passing init= option to kernel. See Linux Documentation/init.txt");
}
```

Kernel Initialization Graph



Kernel Initialization - Summary

- The bootloader executes bootstrap code.
- Bootstrap code initializes the processor and board, and uncompresses the kernel code to RAM, and calls the kernel's `start_kernel` function.
- Copies the command line from the bootloader.
- Identifies the processor and machine.
- Initializes the console.
- Initializes kernel services (memory allocation, scheduling, file cache...)
- Creates a new kernel thread (future init process) and continues in the idle loop.
- Initializes devices and execute initcalls

That's all for today.

Embedded System

Lecture 07: Driver Allocation, Input Subsystem & Memory

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

Hong-Yue Technology Research Building 334
R 09:10 - 12:00

Course Schedule Update

Full Online Course: To practice taking online course,
Required by the School (due to Covid-19)

W	Date	Lecture	Notes	Homework
1	Sept. 17	Lec01: Introduction		HW00
2	Sept. 24	Lec02: The Big Picture		
3	Oct. 1	Moon Festival	No Class	
4	Oct. 8	Lec04: Linux and Real time		
5	Oct. 15	Lec05: The Linux Kernel		
6	Oct. 22	Lec06: Kernel Arch for Device Drivers & Kernel Initialization		
7	Oct. 29	Lec07: Driver Allocation, Input Subsystem & Memory		
8	Nov. 5	Lec08: Direct Memory Access & misc & File System	Full Online Course	
9	Nov. 12	School Midterm Week	No Class	
10	Nov. 19	Lec09: Flash Storage	Full Online Course	
11	Nov. 26	Midterm on Nov. 26	3 Hour Paper-and-pencil Test (Slides & Notes Allowed)	
12	Dec. 3	Lec10: MTD System		
13	Dec. 10	Lec11: Busy box		
14	Dec. 17	Lec12: Device Driver Basics		
15	Dec. 24	Lec13: USB		
16	Dec. 31	Lec14: udev		
17	Jan. 7	Final Presentation	Full Online Course	
18	Jan. 14	Final Presentation	Full Online Course	

Device Managed Allocations

- The `probe()` function is typically responsible for allocating a significant number of resources: memory, mapping I/O registers, registering interrupt handlers, etc.
- These resource allocations have to be properly freed:
 - In the `probe()` function, in case of failure
 - In the `remove()` function
- This required a lot of failure handling code that was rarely tested
- To solve this problem, device managed allocations have been introduced.
- The idea is to associate resource allocation with the `struct device`, and automatically release those resources
 - When the device disappears
 - When the device is unbound from the driver
- Functions prefixed by `devm_`
- See Documentation/driver-model/devres.txt for details

Device Managed Allocations

- Memory allocation example
 - Normally done with `kmalloc(size_t, gfp_t)`, released with `kfree(void *)`
 - Device managed with `devm_kmalloc(struct device *, size_t, gfp_t)`

Without devm functions

```
int foo_probe(struct platform_device *pdev)
{
    foo_t *foo;

    foo = kmalloc(sizeof(struct foo_t),
                  GFP_KERNEL);
    ...
    if (failure) {
        kfree(foo);
        return -EBUSY;
    }
    ...
    platform_set_drvdata(pdev, foo);
    pm_runtime_enable(&pdev->dev);
    pm_runtime_get_sync(&pdev->dev);
    return 0;
}

void foo_remove(struct platform_device *pdev)
{
    pm_runtime_disable(&pdev->dev);
    foo_t *foo = platform_get_drvdata(pdev);
    kfree(foo);
}
```

With devm functions

```
int foo_probe(struct platform_device *pdev)
{
    foo_t *foo;

    foo = devm_kmalloc(&pdev->dev,
                      sizeof(struct foo_t),
                      GFP_KERNEL);
    ...
    if (failure)
        return -EBUSY;
    ...
    pm_runtime_enable(&pdev->dev);
    pm_runtime_get_sync(&pdev->dev);
    return 0;
}

void foo_remove(struct platform_device *pdev)
{
    pm_runtime_disable(&pdev->dev);
}
```

Driver Data Structures and Links

- Each framework defines a structure that a device driver must register to be recognized as a device in this framework
 - `struct uart_port` for serial ports, `struct net_device` for network devices, `struct fb_info` for framebuffers, etc.
- In addition to this structure, the driver usually needs to store additional information about each device
- This is typically done
 - By subclassing the appropriate framework structure
 - By storing a reference to the appropriate framework structure
 - Or by including your information in the framework structure

Driver-Specific Data Structure Examples

- i.MX serial driver: `struct imx_port` is a subclass of `struct uart_port`

```
struct imx_port {  
    struct uart_port port;  
    struct timer_list timer;  
    unsigned int old_status;  
    int txirq, rxirq, rtsirq;  
    unsigned int have_rtscts:1;  
    [...]  
};
```

- ds1305 RTC driver: `struct ds1305` has a reference to `struct rtc_device`

```
struct ds1305 {  
    struct spi_device *spi;  
    struct rtc_device *rtc;  
    [...]  
};
```

Driver-Specific Data Structure Examples

- rtl8150 network driver: `struct rtl8150` has a reference to `struct net_device` and is allocated within that framework structure.

```
struct rtl8150 {  
    unsigned long flags;  
    struct usb_device *udev;  
    struct tasklet_struct tl;  
    struct net_device *netdev;  
    [...]  
};
```

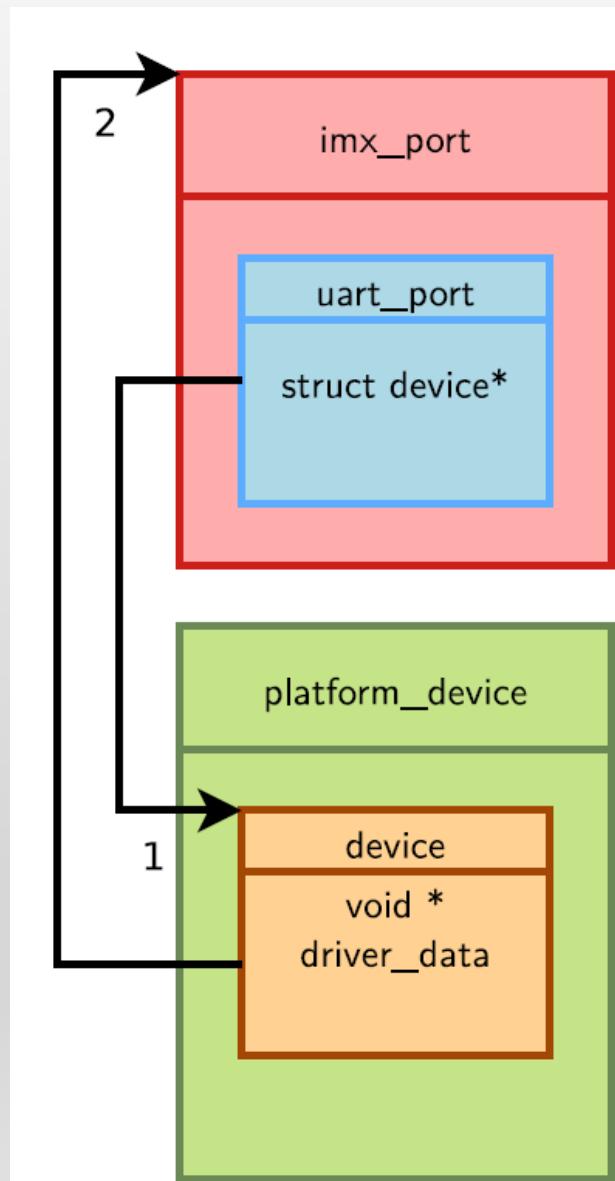
Links between Structures

- The framework structure typically contains a `struct device *` pointer that the driver must point to the corresponding `struct device`
 - It's the relation between the logical device (for example a network interface) and the physical device (for example the USB network adapter)
- The device structure also contains a `void *` pointer that the driver can freely use.
 - It's often used to link back the device to the higher-level structure from the framework.
 - It allows, for example, from the `struct platform_device` structure, to find the structure describing the logical device

Links between Structures

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imx_port *sport;
    [...]
    sport = devm_kzalloc(&pdev->dev, sizeof(*sport), GFP_KERNEL);
    [...]
    /* setup the link between uart_port and the struct
     * device inside the platform_device */
    sport->port.dev = &pdev->dev;                                // Arrow 1
    [...]
    /* setup the link between the struct device inside
     * the platform device to the imx_port structure */
    platform_set_drvdata(pdev, sport);                            // Arrow 2
    [...]
    uart_add_one_port(&imx_reg, &sport->port);
}

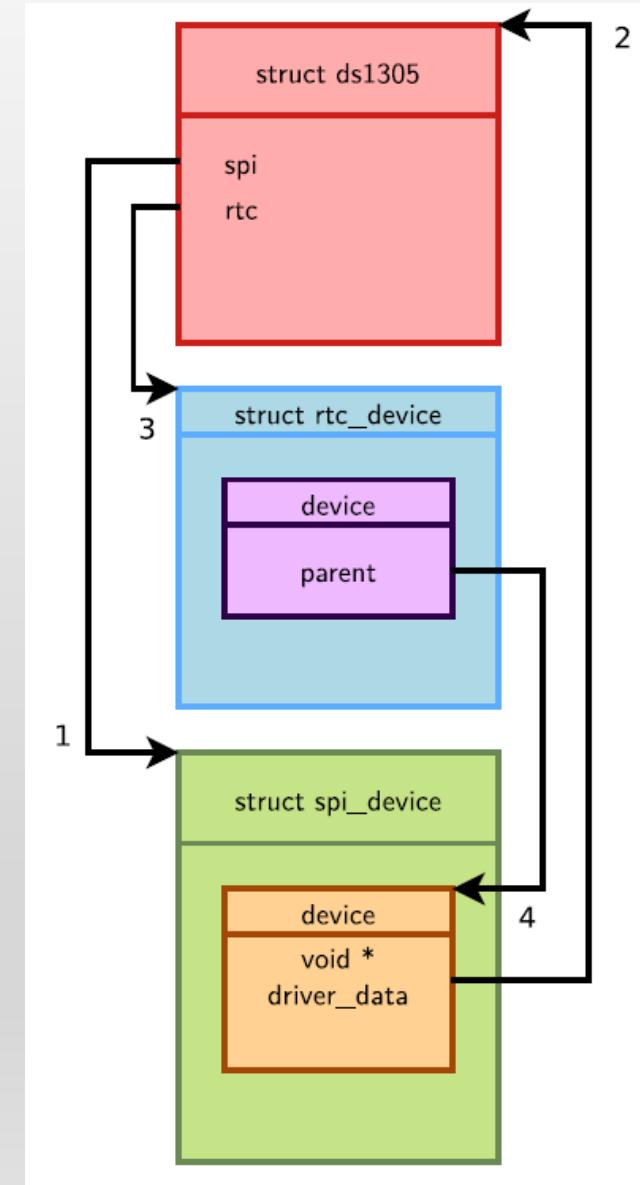
static int serial_imx_remove(struct platform_device *pdev)
{
    /* retrieve the imx_port from the platform_device */
    struct imx_port *sport = platform_get_drvdata(pdev);
    [...]
    uart_remove_one_port(&imx_reg, &sport->port);
    [...]
}
```



Links between Structures

```
static int ds1305_probe(struct spi_device *spi)
{
    struct ds1305             *ds1305;
    [...]
    /* set up driver data */
    ds1305 = devm_kzalloc(&spi->dev, sizeof(*ds1305), GFP_KERNEL);
    if (!ds1305)
        return -ENOMEM;
    ds1305->spi = spi;                                // Arrow 1
    spi_set_drvdata(spi, ds1305);                      // Arrow 2
    [...]
    ds1305->rtc = devm_rtc_allocate_device(&spi->dev); // Arrows 3 and 4
    [...]
}

static int ds1305_remove(struct spi_device *spi)
{
    struct ds1305 *ds1305 = spi_get_drvdata(spi);
    [...]
}
```



Links between Structures

```
static int rtl8150_probe(struct usb_interface *intf,
    const struct usb_device_id *id)
{
    struct usb_device *udev = interface_to_usbdev(intf);
    rtl8150_t *dev;
    struct net_device *netdev;

    netdev = alloc_etherdev(sizeof(rtl8150_t));
    dev = netdev_priv(netdev);

    [...]

    dev->udev = udev;      // Arrow 1
    dev->netdev = netdev;  // Arrow 2

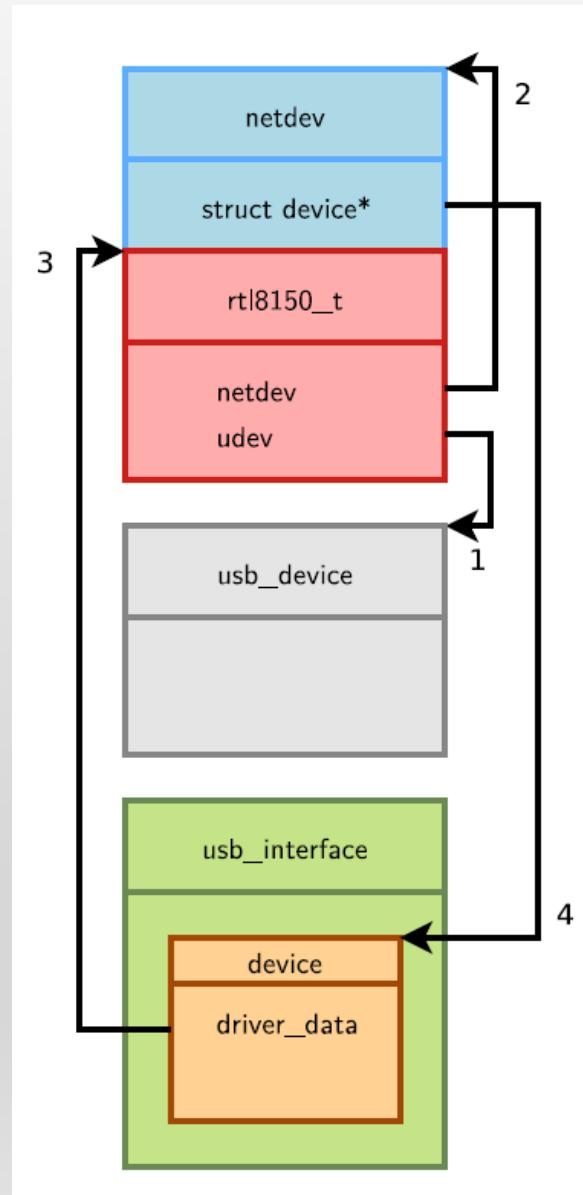
    [...]

    usb_set_intfdata(intf, dev);          // Arrow 3
    SET_NETDEV_DEV(netdev, &intf->dev); // Arrow 4

    [...]
}

static void rtl8150_disconnect(struct usb_interface *intf)
{
    rtl8150_t *dev = usb_get_intfdata(intf);

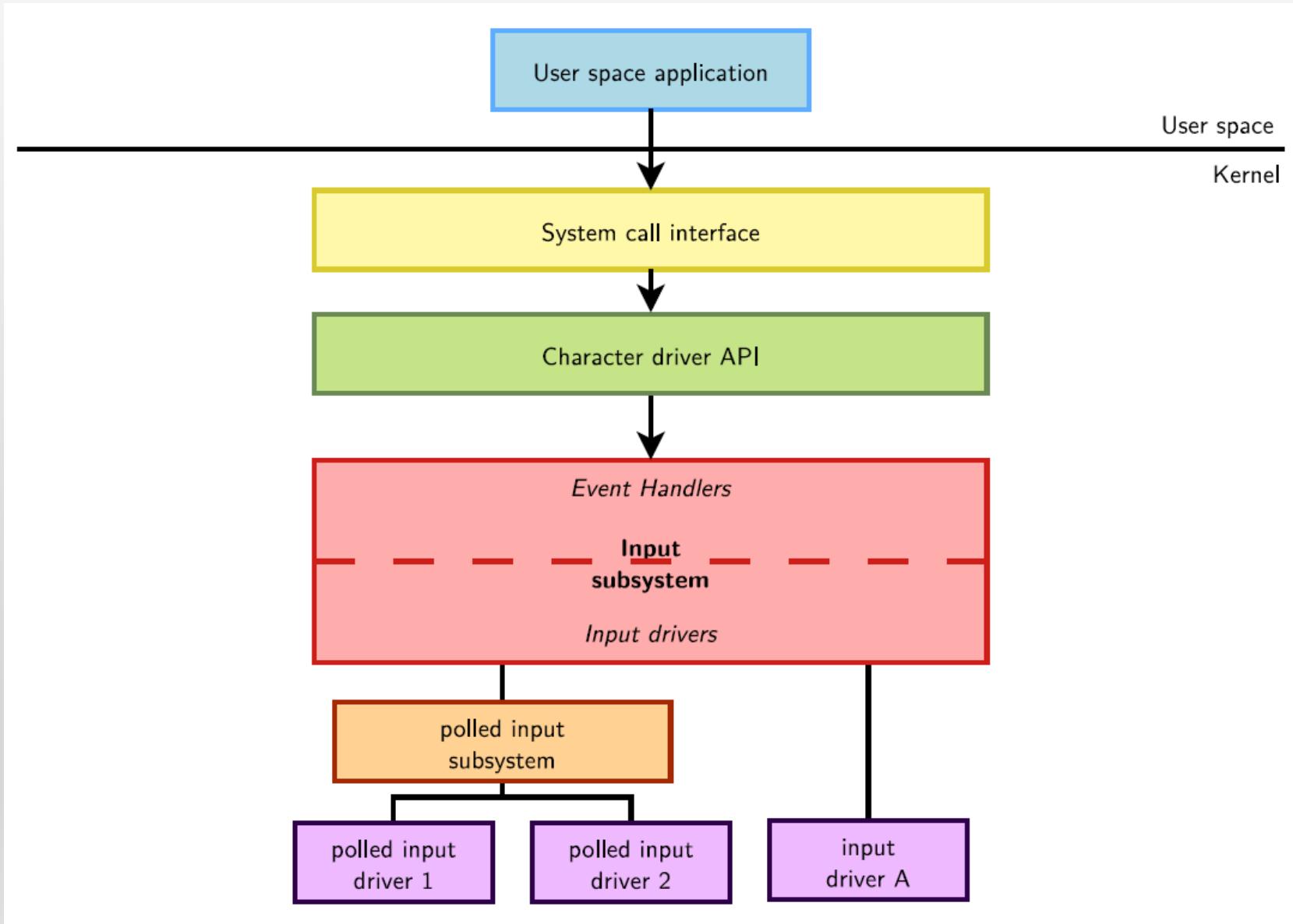
    [...]
}
```



What is the Input Subsystem?

- The input subsystem takes care of all the input events coming from the human user.
- Initially written to support the USB HID (Human Interface Device) devices, it quickly grew up to handle all kind of inputs (using USB or not): keyboards, mice, joysticks, touchscreens, etc.
- The input subsystem is split in two parts:
 - **Device drivers**: they talk to the hardware (for example via USB), and provide events (keystrokes, mouse movements, touchscreen coordinates) to the input core
 - **Event handlers**: they get events from drivers and pass them where needed via various interfaces (most of the time through evdev)
- In user space it is usually used by the graphic stack such as X.Org, Wayland or Android's InputManager.

Input Subsystem Diagram



Input Subsystem API

- Kernel option [CONFIG_INPUT](#)
 - menuconfig INPUT
 - tristate "Generic input layer (needed for keyboard, mouse, ...)"
- Implemented in [drivers/input/](#)
 - `input.c`, `input-polldev.c`, `evbug.c`
- Implements a single character driver and defines the user/kernel API
 - [include/uapi/linux/input.h](#)
- Defines the set of operations a input driver must implement and helper functions for the drivers
 - [struct input_dev](#) for the device driver part
 - [struct input_handler](#) for the event handler part
 - [include/linux/input.h](#)

Input Subsystem API

- An input device is described by a very long `struct input_dev` structure, an excerpt is:

```
struct input_dev {  
    const char *name;  
    [...]  
    struct input_id id;  
    [...]  
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];  
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];  
    [...]  
    int (*getkeycode)(struct input_dev *dev,  
                      struct input_keymap_entry *ke);  
    [...]  
    int (*open)(struct input_dev *dev);  
    [...]  
    int (*event)(struct input_dev *dev, unsigned int type,  
                 unsigned int code, int value);  
    [...]  
};
```

- Before being used it, this structure must be allocated and initialized, typically with:

```
struct input_dev *devm_input_allocate_device(struct device *dev);
```

Input Subsystem API

- Depending on the type of events that will be generated, the input bit fields `evbit` and `keybit` must be configured:
- For example, for a button we only generate `EV_KEY` type events, and from these only `BTN_0` events code:

```
set_bit(EV_KEY, myinput_dev.evbit);
set_bit(BTN_0, myinput_dev.keybit);
```

- `set_bit()` is an atomic operation allowing to set a particular bit to 1
- Once the *input device* is allocated and filled, the function to register it is:

```
int input_register_device(struct input_dev *);
```

Input Subsystem API

- Event types are groupings of codes under a logical input construct. Each type has a set of applicable codes to be used in generating events. See the Codes section for details on valid codes for each type.
- **EV_SYN** - Used as markers to separate events. Events may be separated in time or in space, such as with the multitouch protocol.
- **EV_KEY** - Used to describe state changes of keyboards, buttons, or other key-like devices.
- **EV_REL** - Used to describe relative axis value changes, e.g. moving the mouse 5 units to the left.
- **EV_ABS** - Used to describe absolute axis value changes, e.g. describing the coordinates of a touch on a touchscreen.
- **EV_MSC** - Used to describe miscellaneous input data that do not fit into other types.

...

Input Subsystem API

- The events are sent by the driver to the event handler using `input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value);`
 - The event types are documented in Documentation/input/event-codes.txt
 - An event is composed by one or several input data changes (packet of input data changes) such as the button state, the relative or absolute position along an axis, etc..
 - After submitting potentially multiple events, the input core must be notified by calling: `void input_sync(struct input_dev *dev);`
 - The input subsystem provides other wrappers such as `input_report_key()`, `input_report_abs()`, ...

Example from drivers/hid/usbhid/usbmouse.c

```
static void usb_mouse_irq(struct urb *urb)
{
    struct usb_mouse *mouse = urb->context;
    signed char *data = mouse->data;
    struct input_dev *dev = mouse->dev;
    ...

    input_report_key(dev, BTN_LEFT,    data[0] & 0x01);
    input_report_key(dev, BTN_RIGHT,   data[0] & 0x02);
    input_report_key(dev, BTN_MIDDLE,  data[0] & 0x04);
    input_report_key(dev, BTN_SIDE,   data[0] & 0x08);
    input_report_key(dev, BTN_EXTRA,  data[0] & 0x10);

    input_report_rel(dev, REL_X,      data[1]);
    input_report_rel(dev, REL_Y,      data[2]);
    input_report_rel(dev, REL_WHEEL,  data[3]);

    input_sync(dev);
    ...
}
```

Polled Input Subclass

- The input subsystem provides a subclass supporting simple input devices that do not raise interrupts but have to be periodically scanned or polled to detect changes in their state.
- A polled input device is described by a `struct input_polled_dev` structure:

```
struct input_polled_dev {  
    void *private;  
    void (*open)(struct input_polled_dev *dev);  
    void (*close)(struct input_polled_dev *dev);  
    void (*poll)(struct input_polled_dev *dev);  
    unsigned int poll_interval; /* msec */  
    unsigned int poll_interval_max; /* msec */  
    unsigned int poll_interval_min; /* msec */  
    struct input_dev *input;  
/* private:  
    struct delayed_work work;  
}
```

Polled Input Subsystem API

- Allocating the `struct input_polled_dev` structure is done using `devm_input_allocate_polled_device()`
- Among the handlers of the `struct input_polled_dev` only the `poll()` method is mandatory, this function polls the device and posts input events.
- The fields `id`, `name`, `evbit` and `keybit` of the `struct input` structure must be initialized too.
- If none of the `poll_interval` fields are filled then the default poll interval is 500ms.
- The device registration/unregistration is done with:
 - `input_register_polled_device(struct input_polled_dev *dev)`.
 - Unregistration is automatic after using `devm_input_allocate_polled_device()`

evdev User Space Interface

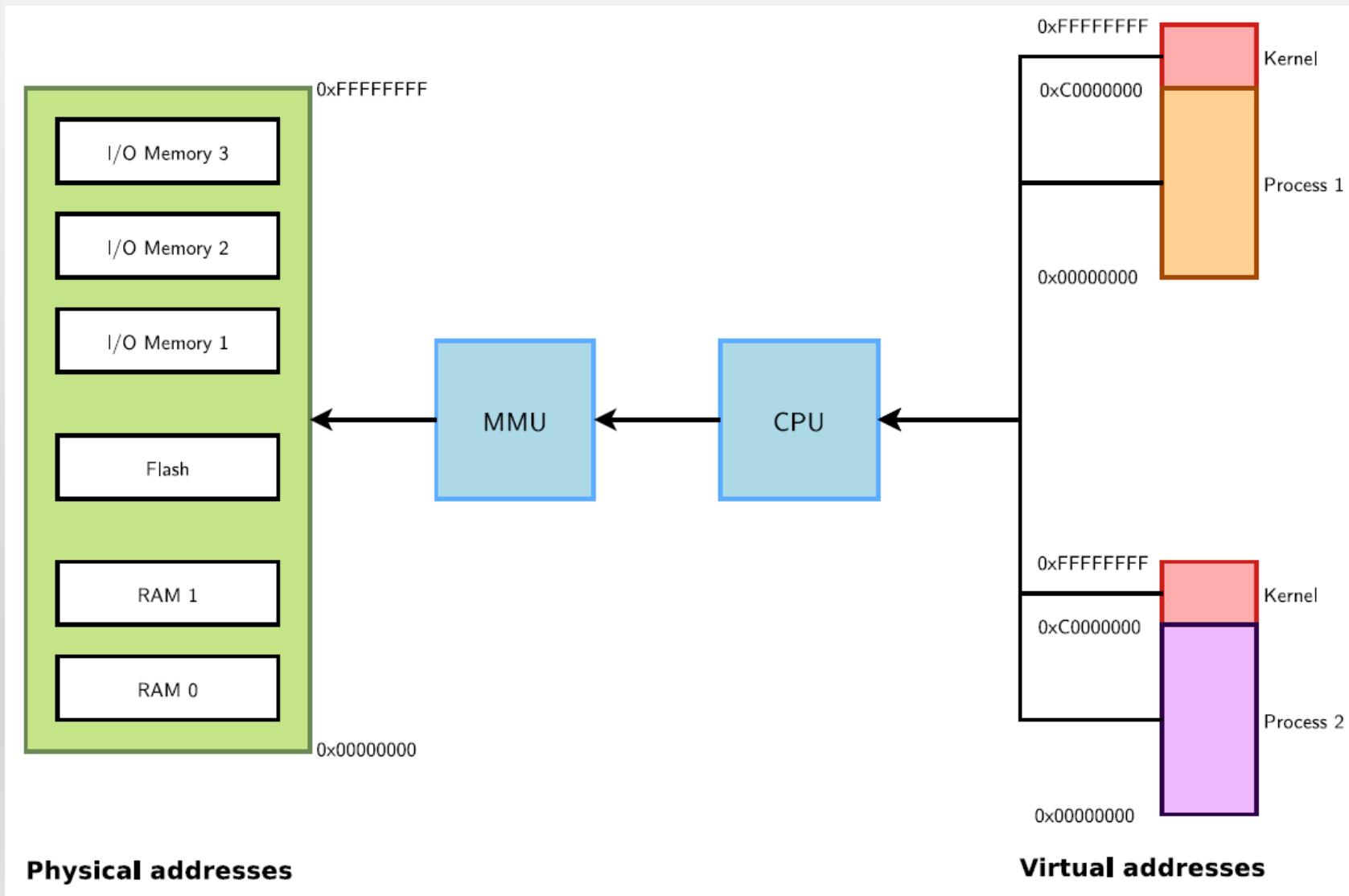
- The main user space interface to input devices is the event interface
- Each input device is represented as a `/dev/input/event<X>` character device
- A user space application can use blocking and non-blocking reads, but also `select()` (to get notified of events) after opening this device.
- Each read will return `struct input_event` structures of the following format:

```
struct input_event {  
    struct timeval time;  
    unsigned short type;  
    unsigned short code;  
    unsigned int value;  
};
```

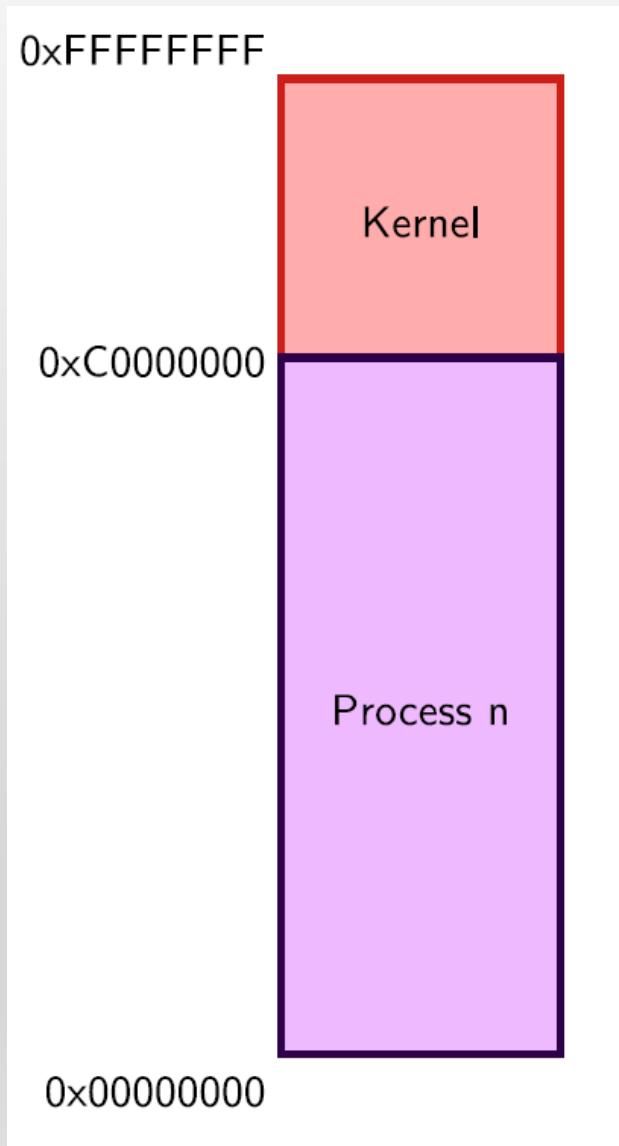
- A very useful application for *input device* testing is `evtest`, from
<https://cgit.freedesktop.org/evtest/>

Memory Management

- Physical and Virtual Memory

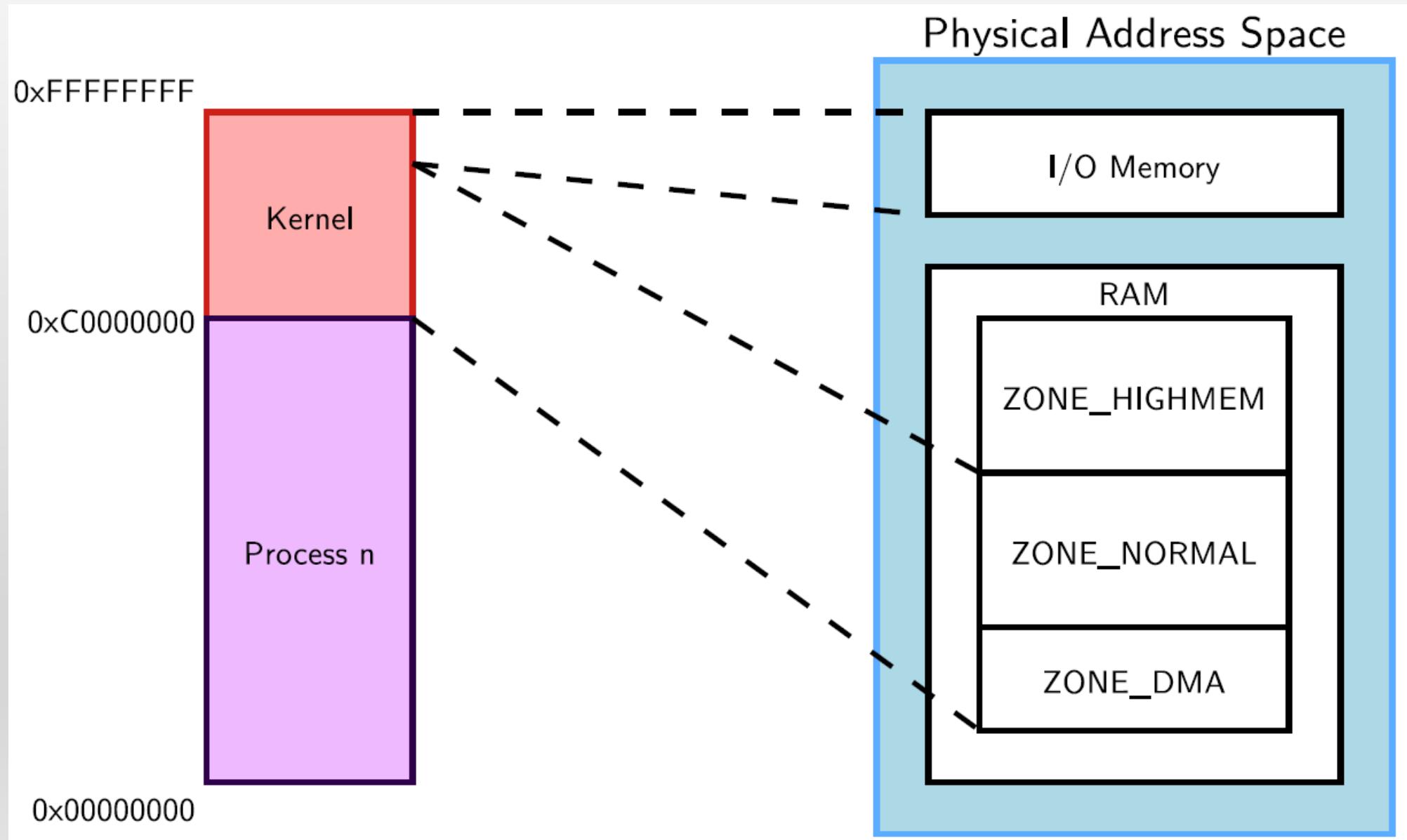


Virtual Memory Organization



- 1GB reserved for kernel-space
 - Contains kernel code and core data structures, identical in all address spaces
 - Most memory can be a direct mapping of physical memory at a fixed offset
- Complete 3GB exclusive mapping available for each user space process
 - Process code and data (program, stack, ...)
 - Memory-mapped files
 - Not necessarily mapped to physical memory (demand fault paging used for dynamic mapping to physical memory pages)
 - Differs from one address space to another

Physical / Virtual Memory Mapping



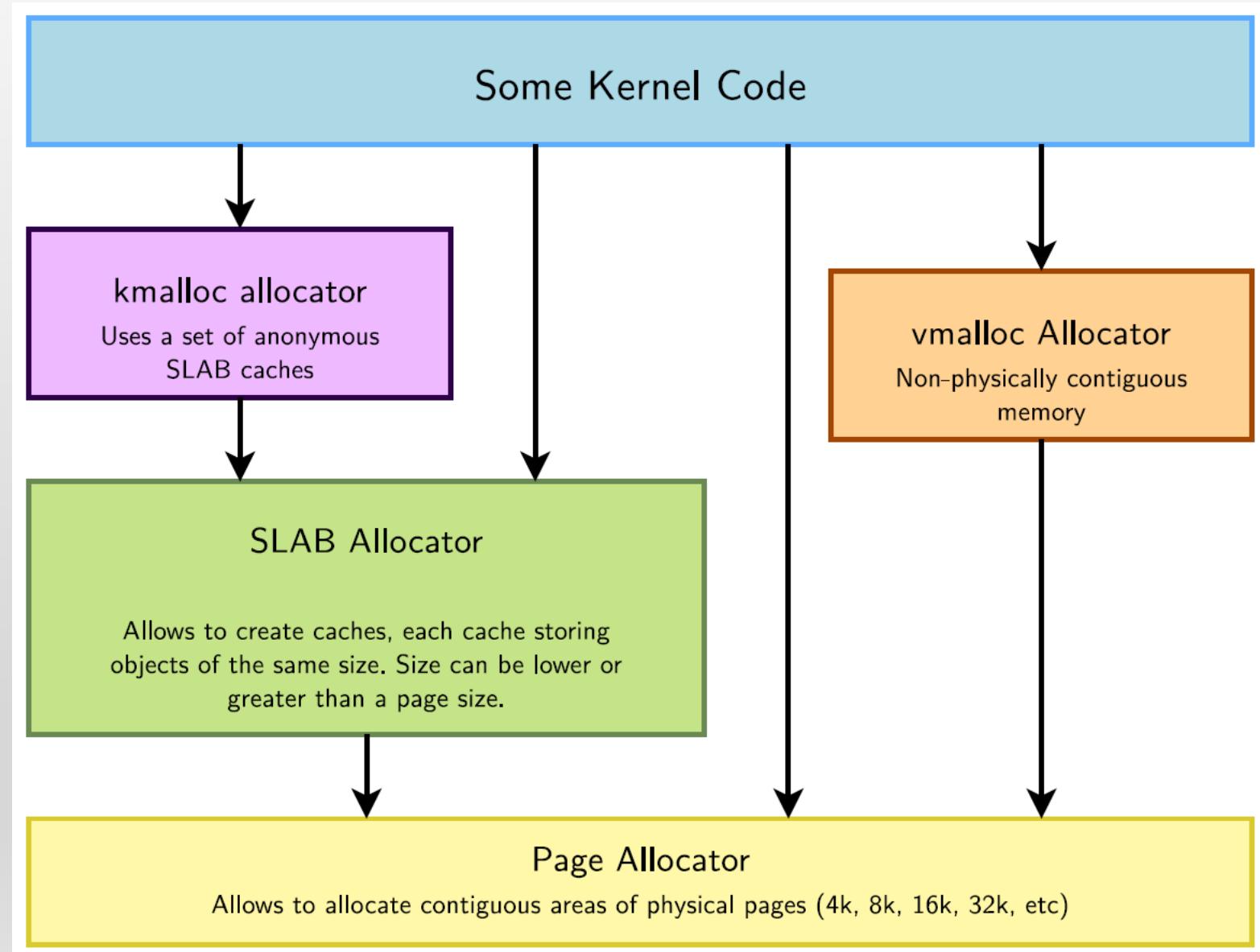
Accessing More Physical Memory

- Only less than 1GB memory addressable directly through kernel virtual address space
- If more physical memory is present on the platform, part of the memory will not be accessible by kernel space, but can be used by user space
- To allow the kernel to access more physical memory:
 - Change the 3GB/1GB memory split to 2GB/2GB or 1GB/3GB ([CONFIG_VMSPLIT_2G](#) or [CONFIG_VMSPLIT_1G](#)) -> reduce total user memory available for each process
 - Change for a 64 bit architecture ;-) See Documentation/x86/x86_64/mm.txt for an example.
 - Activate highmem support if available for your architecture:
 - Allows kernel to map parts of its non-directly accessible memory
 - Mapping must be requested explicitly
 - Limited addresses ranges reserved for this usage
- See <https://lwn.net/Articles/75174/> for useful explanations

Notes on User Space Memory

- New user space memory is allocated either from the already allocated process memory, or using the mmap system call
- Note that memory allocated may not be physically allocated:
 - Kernel uses **demand fault paging** to allocate the physical page (the physical page is allocated when access to the virtual address generates a page fault)
 - ... or may have been swapped out, which also induces a page fault
- User space memory allocation is allowed to over-commit memory (more than available physical memory) -> can lead to out of memory
- OOM killer kicks in and selects a process to kill to retrieve some memory. That's better than letting the system freeze.

Allocators in the Kernel



Page Allocator

- Appropriate for medium-size allocations
- A page is usually 4K, but can be made greater in some architectures (sh, mips: 4, 8, 16 or 64 KB, but not configurable in x86 or arm).
- Buddy allocator strategy, so only allocations of power of two number of pages are possible: 1 page, 2 pages, 4 pages, 8 pages, 16 pages, etc.
- Typical maximum size is 8192 KB, but it might depend on the kernel configuration.
- The allocated area is virtually contiguous (of course), but also physically contiguous. It is allocated in the identity-mapped part of the kernel memory space.
 - This means that large areas may not be available or hard to retrieve due to physical memory fragmentation.

Page Allocator API: Get free pages

- `unsigned long get_zeroed_page(int flags)`
 - Returns the virtual address of a free page, initialized to zero
 - flags: see the next pages for details.
- `unsigned long __get_free_page(int flags)`
 - Same, but doesn't initialize the contents
- `unsigned long __get_free_pages(int flags, unsigned int order)`
 - Returns the starting virtual address of an area of several contiguous pages in physical RAM, with order being $\log_2(\text{number_of_pages})$. Can be computed from the size with the `get_order()` function.

Page Allocator API: Free Pages

- `void free_page(unsigned long addr)`
 - Frees one page.
- `void free_pages(unsigned long addr, unsigned int order)`
 - Frees multiple pages. Need to use the same order as in allocation.

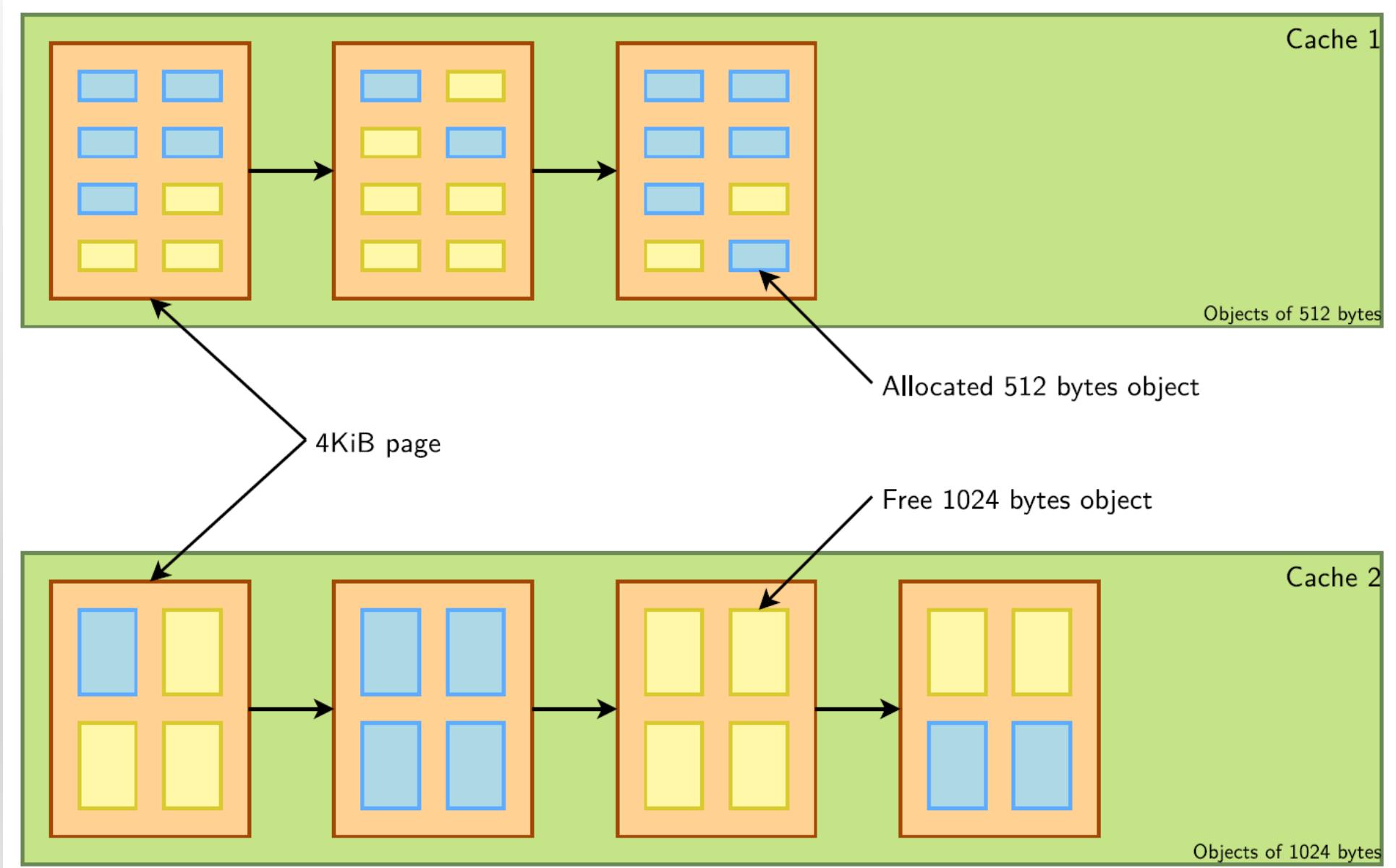
Page Allocator Flags

- The most common ones are:
- **GFP_KERNEL**
 - Standard kernel memory allocation. The allocation may block in order to find enough available memory. Fine for most needs, except in interrupt handler context.
- **GFP_ATOMIC**
 - RAM allocated from code which is not allowed to block (interrupt handlers or critical sections). Never blocks, allows to access emergency pools, but can fail if no free memory is readily available.
- **GFP_DMA**
 - Allocates memory in an area of the physical memory usable for DMA transfers. See our DMA chapter.
- Others are defined in [include/linux/gfp.h](#)

SLAB Allocator

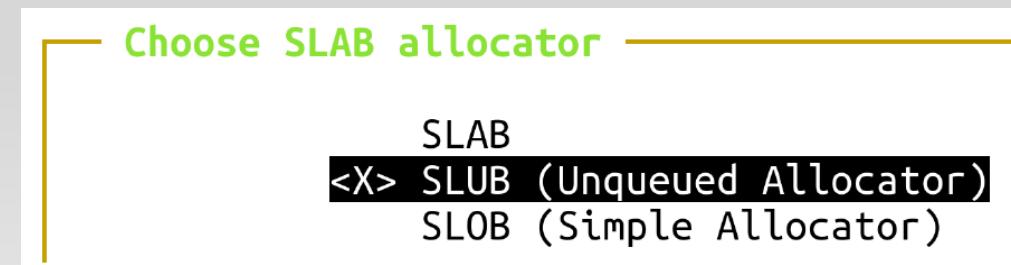
- The SLAB allocator allows to create caches, which contain a set of objects of the same size
- The object size can be smaller or greater than the page size
- The SLAB allocator takes care of growing or reducing the size of the cache as needed, depending on the number of allocated objects. It uses the page allocator to allocate and free pages.
- SLAB caches are used for data structures that are present in many many instances in the kernel: directory entries, file objects, network packet descriptors, process descriptors, etc.
- See `/proc/slabinfo`
- They are rarely used for individual drivers.
- See [`include/linux/slab.h`](#) for the API

SLAB Allocator



Different SLAB Allocators

- There are three different, but API compatible, implementations of a SLAB allocator in the Linux kernel. A particular implementation is chosen at configuration time.
- **SLAB**: legacy, well proven allocator.
 - Linux 4.20 on ARM: used in 48 defconfig files
- **SLOB**: much simpler. More space efficient but doesn't scale well. Can save space in small systems (depends on CONFIG_EXPERT).
 - Linux 4.20 on ARM: used in 7 defconfig files
- **SLUB**: more recent and simpler than SLAB, scaling much better (in particular for huge systems) and creating less fragmentation.
 - Linux 4.20 on ARM: used in 0 defconfig files



kmalloc Allocator

- The kmalloc allocator is the general purpose memory allocator in the Linux kernel
- For small sizes, it relies on generic SLAB caches, named kmalloc-XXX in /proc/slabinfo
- For larger sizes, it relies on the page allocator
- The allocated area is guaranteed to be physically contiguous
- The allocated area size is rounded up to the size of the smallest SLAB cache in which it can fit (while using the SLAB allocator directly allows to have more flexibility)
- It uses the same flags as the page allocator ([GFP_KERNEL](#), [GFP_ATOMIC](#), [GFP_DMA](#), etc.) with the same semantics.
- Maximum sizes, on x86 and arm (see <https://j.mp/YIGq6W>):
 - - Per allocation: 4 MB
 - - Total allocations: 128 MB
- Should be used as the primary allocator unless there is a strong reason to use another

kmalloc API

- `#include <linux/slab.h>`
- `void *kmalloc(size_t size, int flags);`
 - Allocate size bytes, and return a pointer to the area (virtual address)
 - size: number of bytes to allocate
 - flags: same flags as the page allocator
- `void kfree(const void *objp);`
 - Free an allocated area
- Example: (drivers/infiniband/core/cache.c)
 - `struct ib_update_work *work;`
 - `work = kmalloc(sizeof *work, GFP_ATOMIC);`
 - `...`
 - `kfree(work);`

kmalloc API

- `void *kzalloc(size_t size, gfp_t flags);`
 - Allocates a zero-initialized buffer
- `void *kcalloc(size_t n, size_t size, gfp_t flags);`
 - Allocates memory for an array of n elements of size size, and zeroes its contents.
- `void *krealloc(const void *p, size_t new_size, gfp_t flags);`
 - Changes the size of the buffer pointed by p to new_size, by reallocating a new buffer and copying the data, unless new_size fits within the alignment of the existing buffer.

devm_kmalloc Functions

- Allocations with automatic freeing when the corresponding device or module is unprobed.
 - `void *devm_kmalloc(struct device *dev, size_t size, int flags);`
 - `void *devm_kzalloc(struct device *dev, size_t size, int flags);`
 - `void *devm_kcalloc(struct device *dev, size_t n, size_t size, gfp_t flags);`
 - `void *devm_kfree(struct device *dev, void *p);`
- Useful to immediately free an allocated buffer
- For use in probe() functions.

vmalloc Allocator

- The `vmalloc()` allocator can be used to obtain virtually contiguous memory zones, but not physically contiguous. The requested memory size is rounded up to the next page.
- The allocated area is in the kernel space part of the address space, but outside of the identically-mapped area
- Allocations of fairly large areas is possible (almost as big as total available memory), since physical memory fragmentation is not an issue, but areas cannot be used for DMA, as DMA usually requires physically contiguous buffers.
- Example use: to allocate kernel buffers to load module code.
- API in `include/linux/vmalloc.h`
 - `void *vmalloc(unsigned long size);`
 - Returns a virtual address
 - `vfree(void *addr);`

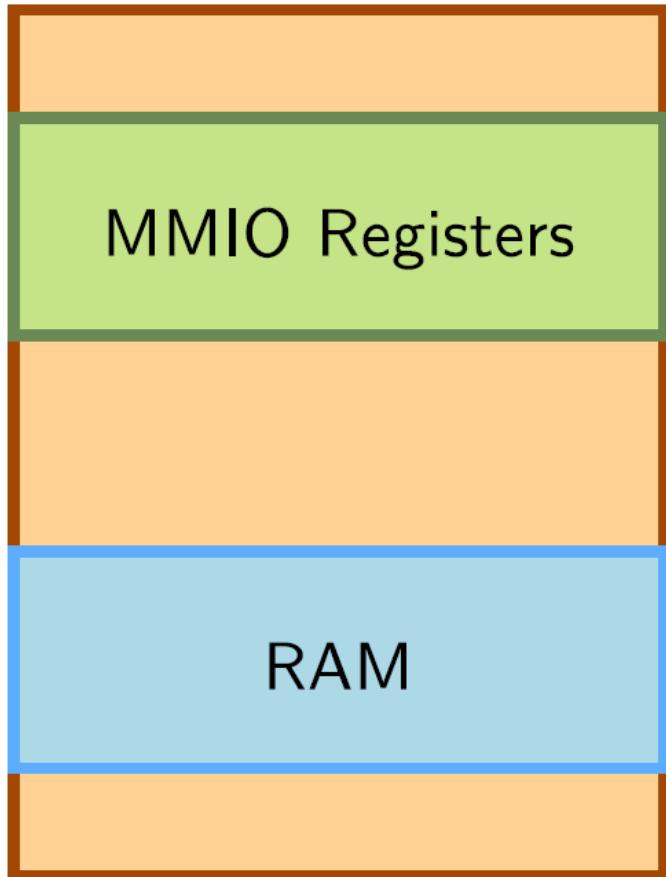
Kernel Memory Debugging

- **KASAN** (Kernel Address Sanitizer)
 - Dynamic memory error detector, to find use-after-free and out-of-bounds bugs.
 - Only available on x86_64, arm64, s390 and xtensa so far (Linux 4.20 status), but will help to improve architecture independent code anyway.
 - See dev-tools/kasan for details.
- **Kmemleak**
 - Dynamic checker for memory leaks
 - This feature is available for all architectures.
 - See dev-tools/kmemleak for details.
- Both have a significant overhead. Only use them in development!
- -

I/O Memory and Port I/O

- MMIO
 - Same address bus to address memory and I/O devices
 - Access to the I/O devices using regular instructions
 - Most widely used I/O method across the different architectures supported by Linux
- PIO
 - Different address spaces for memory and I/O devices
 - Uses a special class of CPU instructions to access I/O devices
 - Example on x86: IN and OUT instructions

MMIO vs PIO



Physical Memory
address space, accessed with
normal load/store instructions



Separate I/O address space,
accessed with specific instructions

Requesting I/O ports

- Tells the kernel which driver is using which I/O ports
- Allows to prevent other drivers from using the same I/O ports, but is purely voluntary.

```
struct resource *request_region(  
    unsigned long start,  
    unsigned long len,  
    char *name);
```

- Tries to reserve the given region and returns NULL if unsuccessful.

```
request_region(0x0170, 8, "ide1");  
  
void release_region(  
    unsigned long start,  
    unsigned long len);
```

/proc/ioports example (x86)

```
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
0376-0376 : ide1
03f6-03f6 : ide0
03f8-03ff : serial
0800-087f : 0000:00:1f.0
...
```

Accessing I/O ports

- Functions to read/write bytes (**b**), word (**w**) and longs (**l**) to I/O ports:
 - `unsigned in[bwl](unsigned long port)`
 - `void out[bwl](value, unsigned long port)`
- And the strings variants: often more efficient than the corresponding C loop, if the processor supports such operations!
 - `void ins[bwl](unsigned port, void *addr, unsigned long count)`
 - `void outs[bwl](unsigned port, void *addr, unsigned long count)`
- Examples
 - read 8 bits
 - `oldlcr = inb(baseio + UART_LCR)`
 - write 8 bits
 - `outb(MOXA_MUST_ENTER_ENCHANCE, baseio + UART_LCR)`

Requesting I/O memory

- Functions equivalent to `request_region()` and `release_region()`, but for I/O memory.

```
struct resource *request_mem_region(  
    unsigned long start,  
    unsigned long len,  
    char *name);  
  
void release_mem_region(  
    unsigned long start,  
    unsigned long len);
```

/proc/iomem example - ARM (Raspberry Pi, Linux 4.19)

```
00000000-3b3fffff : System RAM
 00008000-00bfffff : Kernel code
 00d00000-00e6ad0f : Kernel data
3f006000-3f006fff : dwc_otg
3f007000-3f007eff : dma@7e007000|
3f00a000-3f00a023 : watchdog@7e100000
3f00b840-3f00b87b : mailbox@7e00b840
3f00b880-3f00b8bf : mailbox@7e00b880
3f100000-3f100113 : watchdog@7e100000
3f101000-3f102fff : cprman@7e101000
3f104000-3f10400f : rng@7e104000
3f200000-3f2000b3 : gpio@7e200000
3f201000-3f2011ff : serial@7e201000
 3f201000-3f2011ff : serial@7e201000
3f202000-3f2020ff : mmc@7e202000
3f212000-3f212007 : thermal@7e212000
3f215000-3f215007 : aux@7e215000
3f980000-3f98ffff : dwc_otg
```

Mapping I/O Memory in Virtual Memory

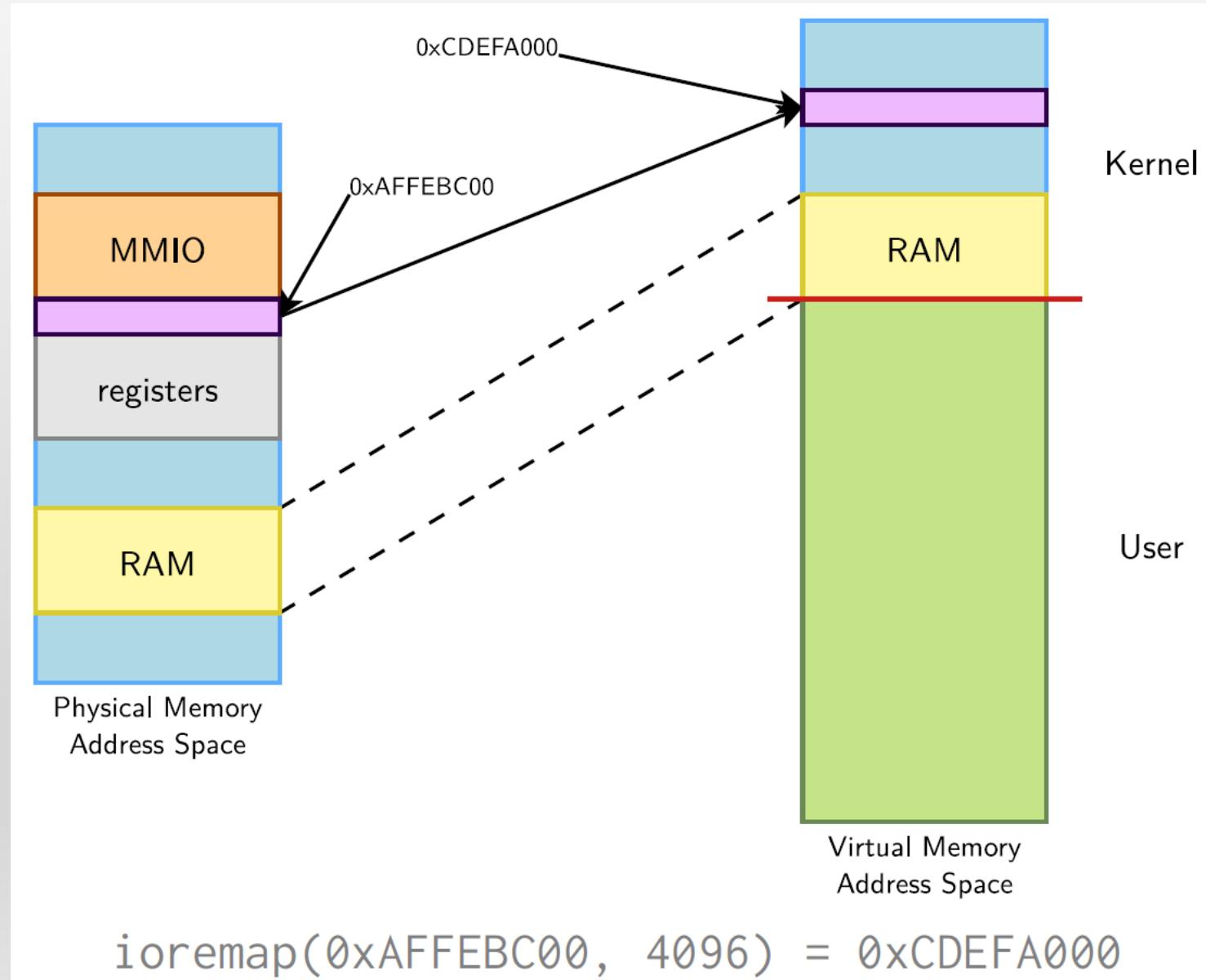
- Load/store instructions work with virtual addresses
- To access I/O memory, drivers need to have a virtual address that the processor can handle, because I/O memory is not mapped by default in virtual memory
- The `ioremap` function satisfies this need:

```
#include <asm/io.h>

void __iomem *ioremap(phys_addr_t phys_addr,
                      unsigned long size);
void iounmap(void __iomem *addr);
```

- Caution: check that `ioremap()` doesn't return a NULL address!

ioremap()



Managed API

- Using `request_mem_region()` and `ioremap()` in device drivers is now deprecated. You should use the below "managed" functions instead, which simplify driver coding and error handling:
 - `devm_ioremap()`
 - `devm_iounmap()`
 - `devm_ioremap_resource()`
 - Takes care of both the request and remapping operations
 - Example (drivers/crypto/atmel-sha.c):

```
sha_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
...
sha_dd->io_base = devm_ioremap_resource(&pdev->dev, sha_res);
```

- `devm_platform_ioremap_resource()`

Accessing MMIO devices

- Directly reading from or writing to addresses returned by ioremap() (pointer dereferencing) may not work on some architectures.
- To do PCI-style, little-endian accesses, conversion being done automatically
 - `unsigned read[bwl](void *addr);`
 - `void write[bwl](unsigned val, void *addr);`
- To do raw access, without endianness conversion
 - `unsigned __raw_read[bwl](void *addr);`
 - `void __raw_write[bwl](unsigned val, void *addr);`
- Example
 - 32 bits write
 - `__raw_writel(1 << KS8695_IRQ_UART_TX, membase + KS8695_INTST);`

Avoiding I/O Access Issues

- Caching on I/O ports or memory already disabled
- Use the `writel()/readl()` macros, they do the right thing for your architecture
- The compiler and/or CPU can reorder memory accesses, which might cause troubles for your devices if they expect one register to be read/written before another one.
 - Memory barriers are available to prevent this reordering
 - `rmb()` is a read memory barrier, prevents reads to cross the barrier
 - `wmb()` is a write memory barrier
 - `mb()` is a read-write memory barrier
- Starts to be a problem with CPUs that reorder instructions and SMP.
- See Documentation/memory-barriers.txt for details

/dev/mem

- Used to provide user space applications with direct access to physical addresses.
- Usage: open /dev/mem and read or write at given offset. What you read or write is the value at the corresponding physical address.
- Used by applications such as the X server to write directly to device memory.
- On x86, arm, arm64, powerpc, s390 and unicore32: **CONFIG_STRICT_DEVMEM** option to restrict /dev/mem to non-RAM addresses, for security reasons (Linux 4.20 status).

That's all for today.

Embedded System

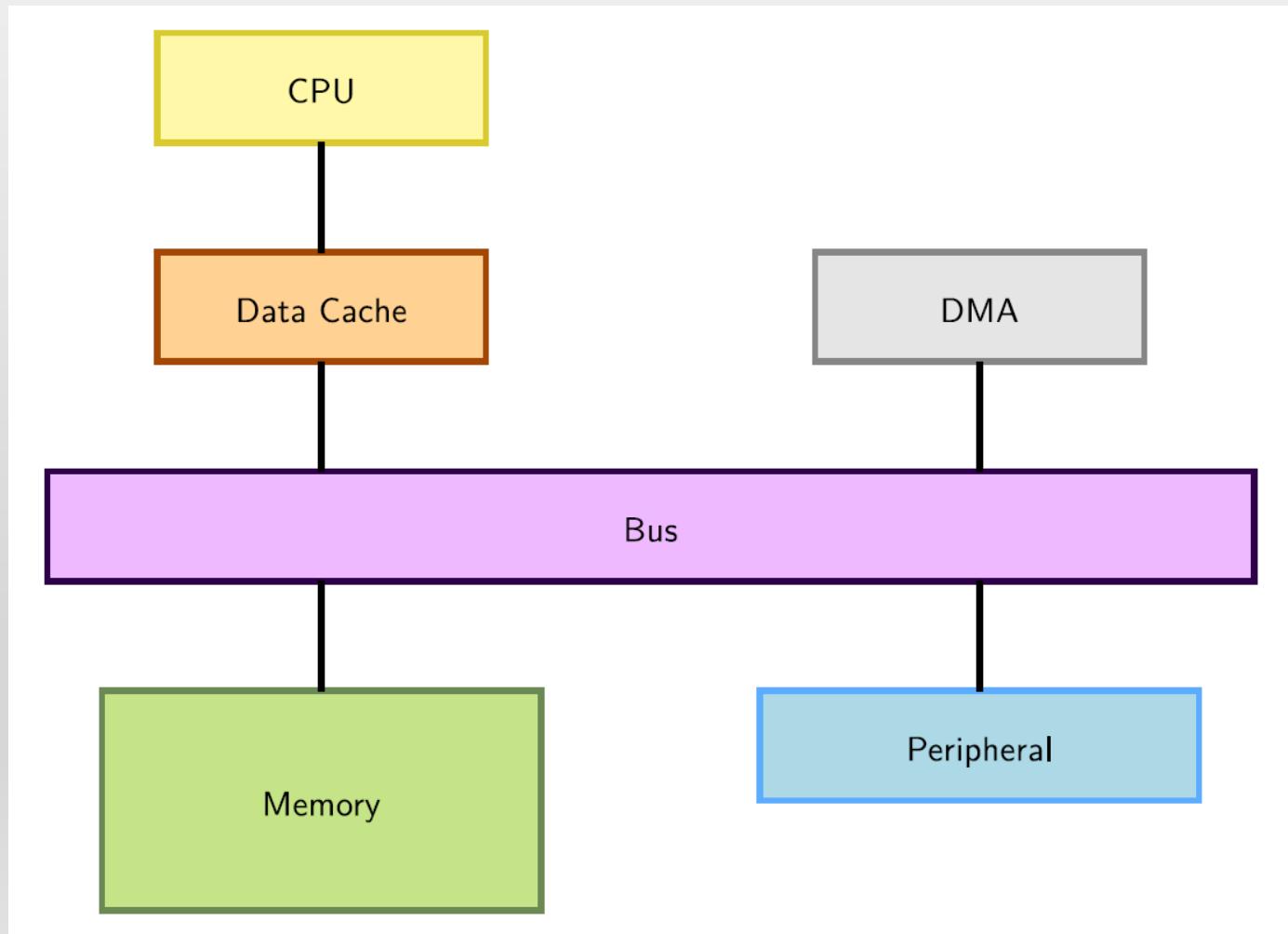
Lecture 08: Direct Memory Access & misc & File System

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

Hong-Yue Technology Research Building 334
R 09:10 - 12:00

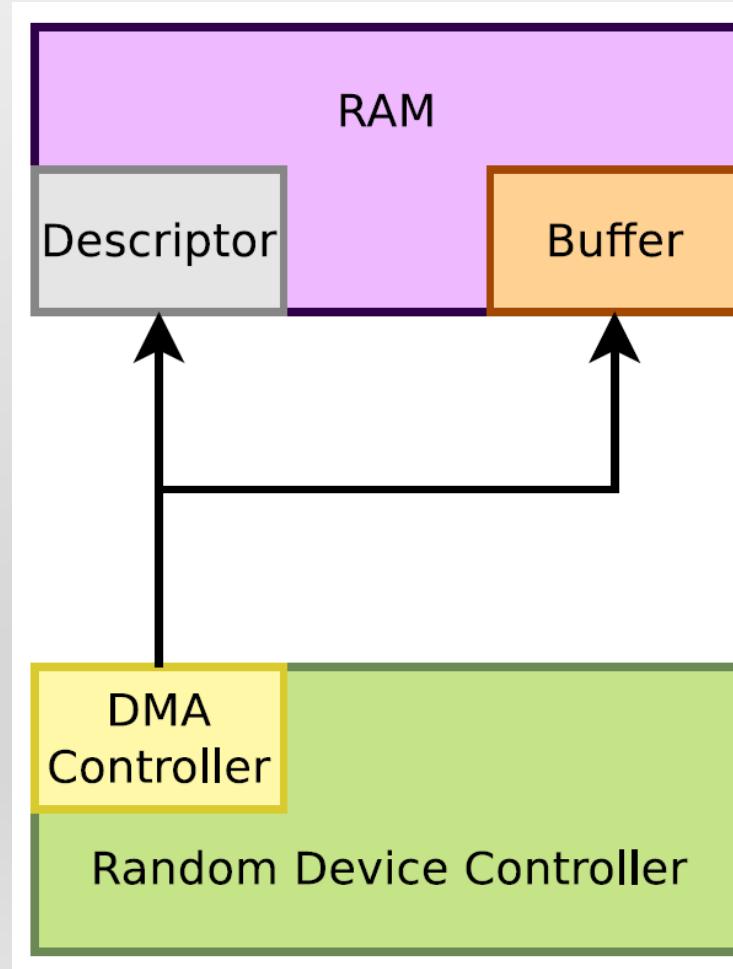
DMA Integration

- DMA (Direct Memory Access) is used to copy data directly between devices and RAM, without going through the CPU.



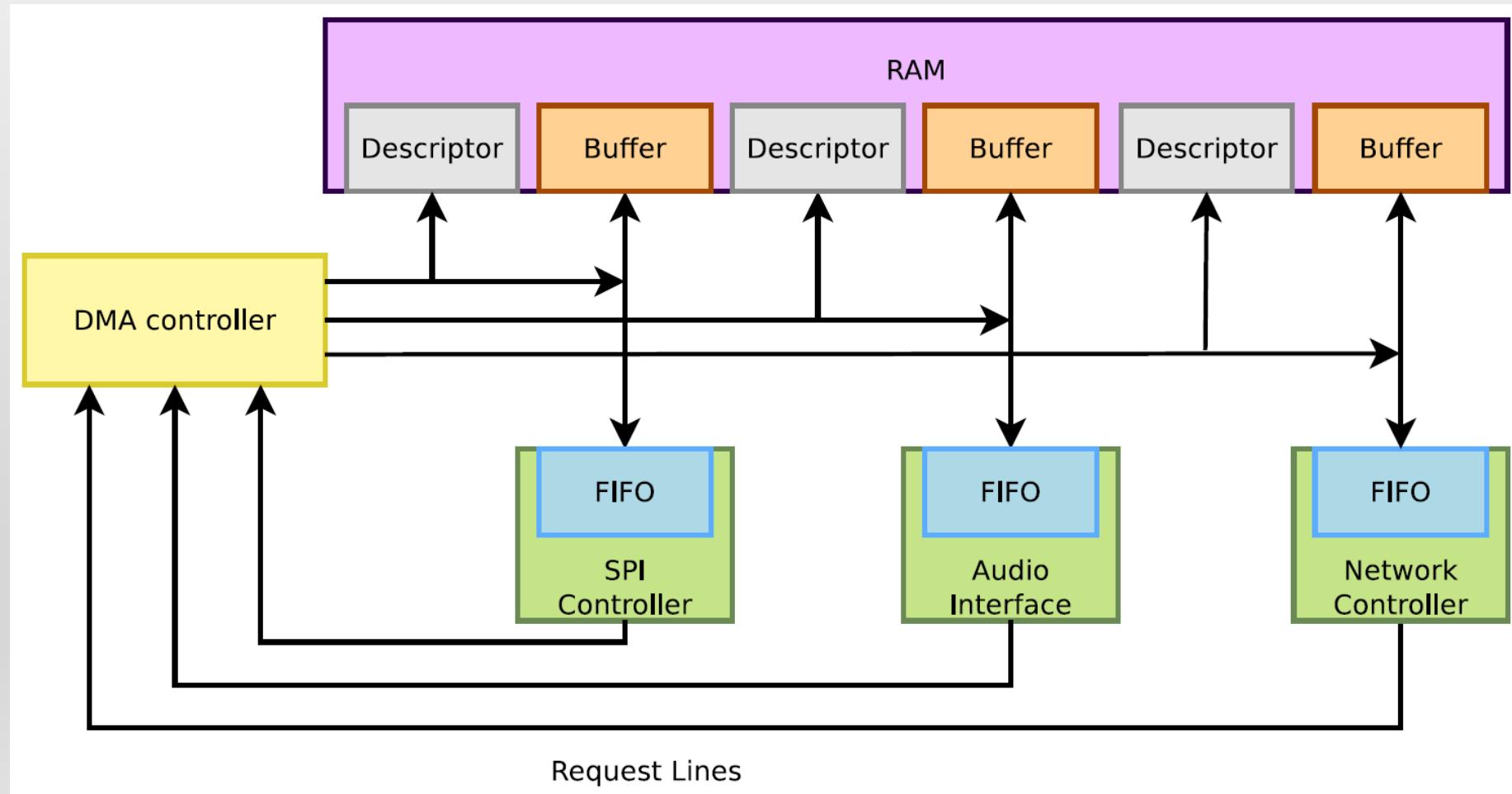
Peripheral DMA

- Some device controllers embedded their own DMA controller and therefore can do DMA on their own.



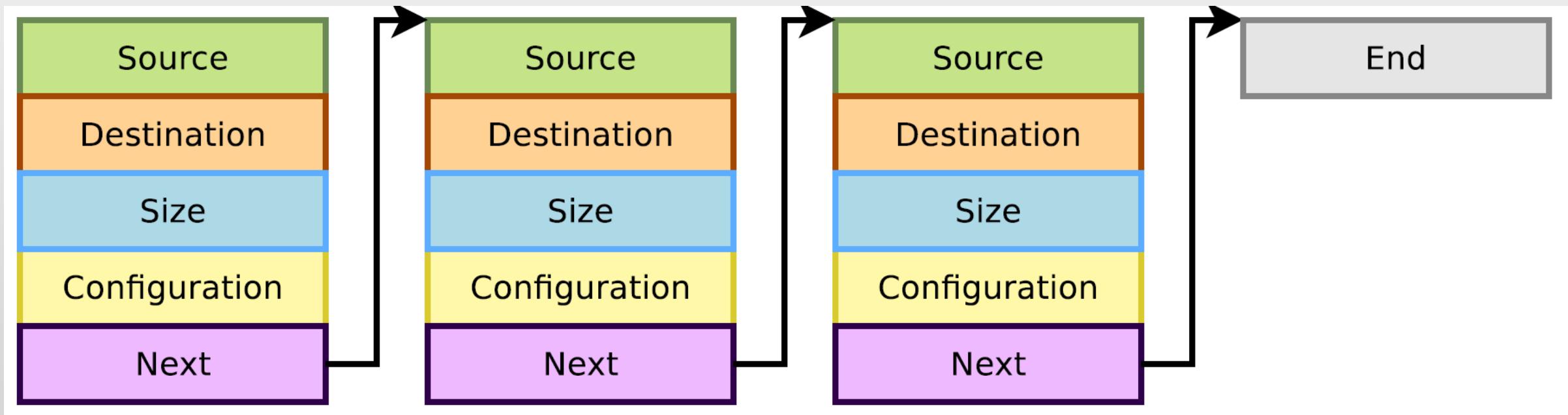
DMA Controllers

- Other device controllers rely on an external DMA controller (on the SoC). Their drivers need to submit DMA descriptors to this controller.



DMA Descriptors

- DMA descriptors describe the various attributes of a DMA transfer, and are chained.



Constraints with a DMA

- A DMA deals with physical addresses, so:
 - Programming a DMA requires retrieving a physical address at some point (virtual addresses are usually used)
- The memory accessed by the DMA shall be physically contiguous
- The CPU can access memory through a data cache
 - Using the cache can be more efficient (faster accesses to the cache than the bus)
 - But the DMA does not access the CPU cache, so one needs to take care of cache coherency (cache content vs. memory content)
 - Either clean (write to memory) or invalidate the cache lines corresponding to the buffer accessed by DMA and processor at the right times

DMA Memory Constraints

- Need to use contiguous memory in physical space.
- Can use any memory allocated by `kmalloc()` (up to 128 KB) or `__get_free_pages()` (up to 8MB).
- Can use block I/O and networking buffers, designed to support DMA.
- Can not use `vmalloc()` memory (would have to setup DMA on each individual physical page).

Memory synchronization issues

- Memory caching could interfere with DMA
- Before DMA to device
 - Need to make sure that all writes to the DMA buffer are done (corresponding cache lines cleaned)
- After DMA from device
 - Before drivers read from a DMA buffer, need to make sure that the corresponding cache lines are invalidated.
- Bidirectional DMA
 - Need to do both of the above operations

Linux DMA API

- The kernel DMA utilities can take care of:
 - Either allocating a buffer in a cache coherent area,
 - Or making sure caches are handled when required,
 - Managing the DMA mappings and IOMMU (if any).
- See Documentation/DMA-API.txt for details about the Linux DMA generic API.
- Most subsystems (such as PCI or USB) supply their own DMA API, derived from the generic one. May be sufficient for most needs.

Coherent or Streaming DMA Mappings

- Coherent mappings
 - The kernel allocates a suitable buffer and sets the mapping for the driver.
 - Can simultaneously be accessed by the CPU and device.
 - So, has to be in a cache coherent memory area.
 - Usually allocated for the whole time the module is loaded.
 - Can be expensive to setup and use on some platforms.
- Streaming mappings
 - The kernel just sets the mapping for a buffer provided by the driver.
 - Use a buffer already allocated by the driver.
 - Mapping set up for each transfer. Keeps DMA registers free on the hardware.
 - The recommended solution.

Allocating Coherent Mappings

- The kernel takes care of both buffer allocation and mapping

```
#include <asm/dma-mapping.h>

void *dma_alloc_coherent( /* Output: buffer address */
    struct device *dev, /* device structure */
    size_t size, /* Needed buffer size in bytes */
    dma_addr_t *handle, /* Output: DMA bus address */
    gfp_t gfp /* Standard GFP flags */
);

void dma_free_coherent(struct device *dev,
    size_t size, void *cpu_addr, dma_addr_t handle);
```

Setting up Streaming Mappings

- Works on buffers already allocated by the driver

```
#include <linux/dmapool.h>

dma_addr_t dma_map_single(
    struct device *,          /* device structure */
    void *,                  /* input: buffer to use */
    size_t,                  /* buffer size */
    enum dma_data_direction /* Either DMA_BIDIRECTIONAL,
                           * DMA_TO_DEVICE or
                           * DMA_FROM_DEVICE */
);
void dma_unmap_single(struct device *dev, dma_addr_t handle,
                      size_t size, enum dma_data_direction dir);
```

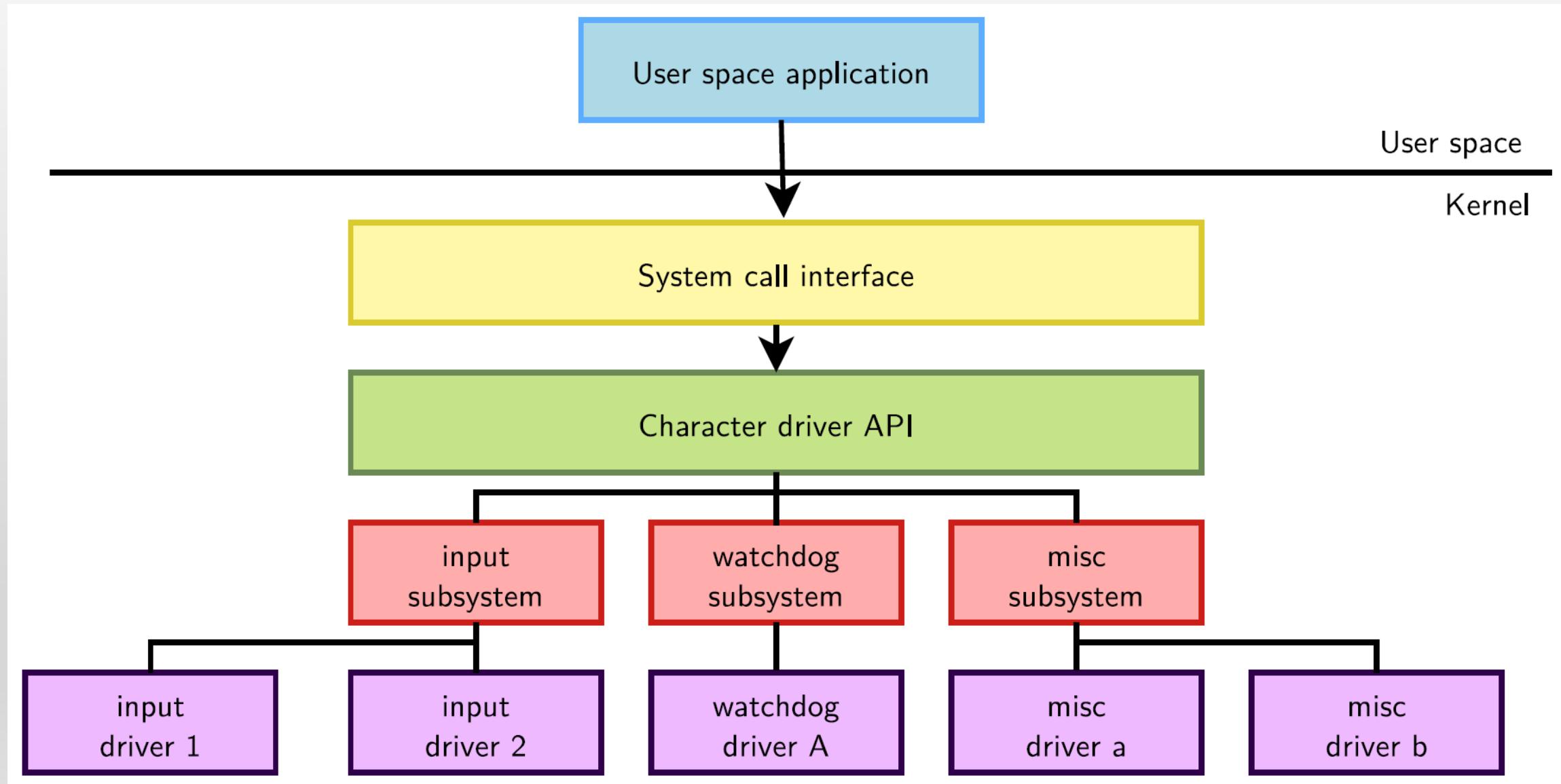
DMA Streaming Mapping Notes

- When the mapping is active: only the device should access the buffer (potential cache issues otherwise).
- The CPU can access the buffer only after unmapping!
- Another reason: if required, this API can create an intermediate bounce buffer (used if the given buffer is not usable for DMA).
- The Linux API also supports scatter / gather DMA streaming mappings.

Why a misc subsystem?

- The kernel offers a large number of **frameworks** covering a wide range of device types: input, network, video, audio, etc.
 - These frameworks allow to factorize common functionality between drivers and offer a consistent API to user space applications.
- However, there are some devices that **really do not fit in any of the existing frameworks**.
 - Highly customized devices implemented in a FPGA, or other weird devices for which implementing a complete framework is not useful.
- The drivers for such devices could be implemented directly as raw character drivers (with `cdev_init()` and `cdev_add()`).
- But there is a subsystem that makes this work a little bit easier: the **misc subsystem**.
 - It is really only a **thin layer** above the character driver API.
 - Another advantage is that devices are integrated in the Device Model (device files appearing in devtmpfs, which you don't have with raw character devices).

misc Subsystem Diagram



Misc Subsystem API

- The misc subsystem API mainly provides two functions, to register and unregister a single misc device:
 - `int misc_register(struct miscdevice * misc);`
 - `void misc_deregister(struct miscdevice *misc);`
- A misc device is described by a `struct miscdevice` structure:

```
struct miscdevice {  
    int minor;  
    const char *name;  
    const struct file_operations *fops;  
    struct list_head list;  
    struct device *parent;  
    struct device *this_device;  
    const char *nodename;  
    umode_t mode;  
};
```

Misc Subsystem API

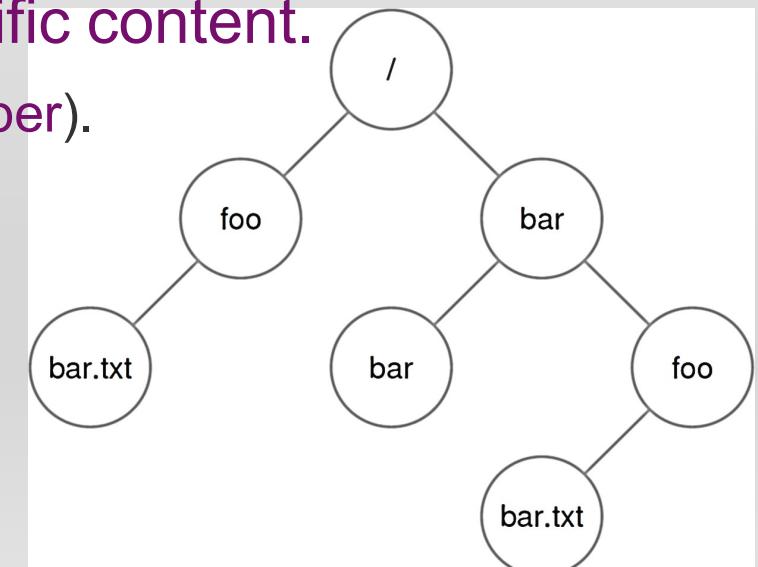
- The main fields to be filled in `struct miscdevice` are:
 - `minor`, the minor number for the device, or `MISC_DYNAMIC_MINOR` to get a minor number automatically assigned.
 - `name`, name of the device, which will be used to create the device node if devtmpfs is used.
 - `fops`, pointer to the same `struct file_operations` structure that is used for raw character drivers, describing which functions implement the read, write, ioctl, etc. operations.
 - `parent`, pointer to the `struct device` of the underlying “physical” device (platform device, I2C device, etc.).

User Space API for misc Devices

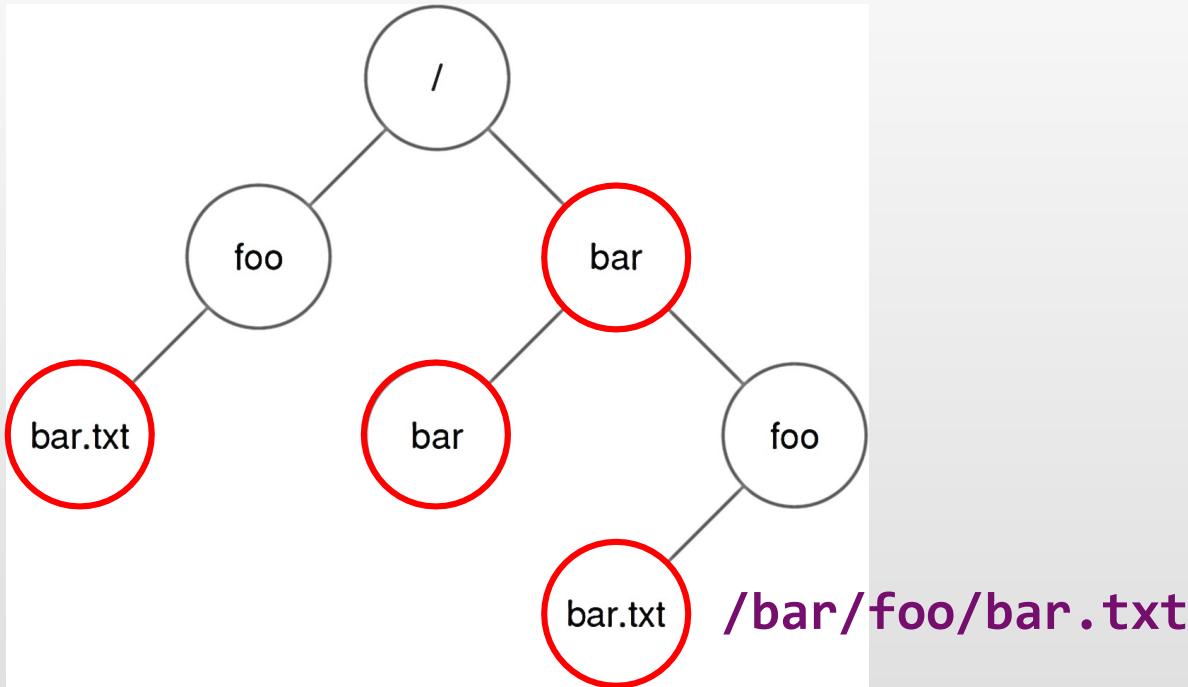
- misc devices are regular character devices
- The operations they support in user space depends on the operations the kernel driver implements:
 - The `open()` and `close()` system calls to open/close the device.
 - The `read()` and `write()` system calls to read/write to/from the device.
 - The `ioctl()` system call to call some driver-specific operations.

File Systems Basics

- **File:** a linear array of bytes that can be read/written
 - Each file has a **low-level name** (or **inode number**) that uniquely identifies itself in the file system.
 - Often, the user is **not aware** of this name.
- **Directory:** a list of **entries**
 - A directory has an **inode number** as well.
 - A directory is just a **special type of file** with specific content.
 - Each entry is a pair of (**user-readable name**, **inode number**).
 - Each entry refers to either files or other directories.
- **A directory tree is formed**
 - Leaf node: **file**
 - Non-leaf node: **directory**



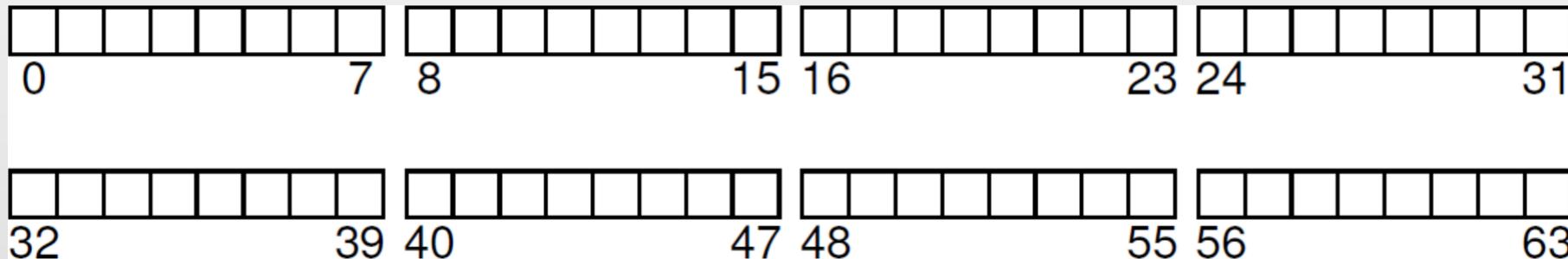
Directory Tree



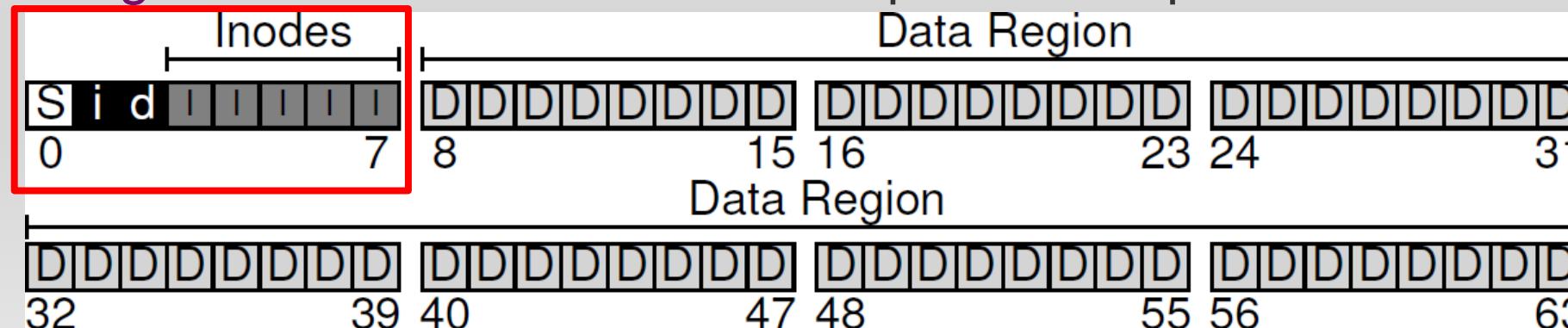
- **/ is the root directory.**
 - / is also used as a “**separator**” to name subsequent sub- directories and files.
- A file/directory is referred by the **absolute pathname**.
 - Directories and files can have the **same name**.
 - If they are in different locations/directories (e.g., **/bar/foo/bar.txt**).
 - The **file extension** is to indicate the type of a file (e.g., **.txt**).

Overall Organization

- **On-Disk Organization**
 - A series of **blocks** (e.g., 4 KB) is addressed from 0 to N – 1.



- **File System Organization**
 - **Metadata Region**: tracks data and file system information.
 - **Data Region**: stores user data and occupies most space.



File System Metadata

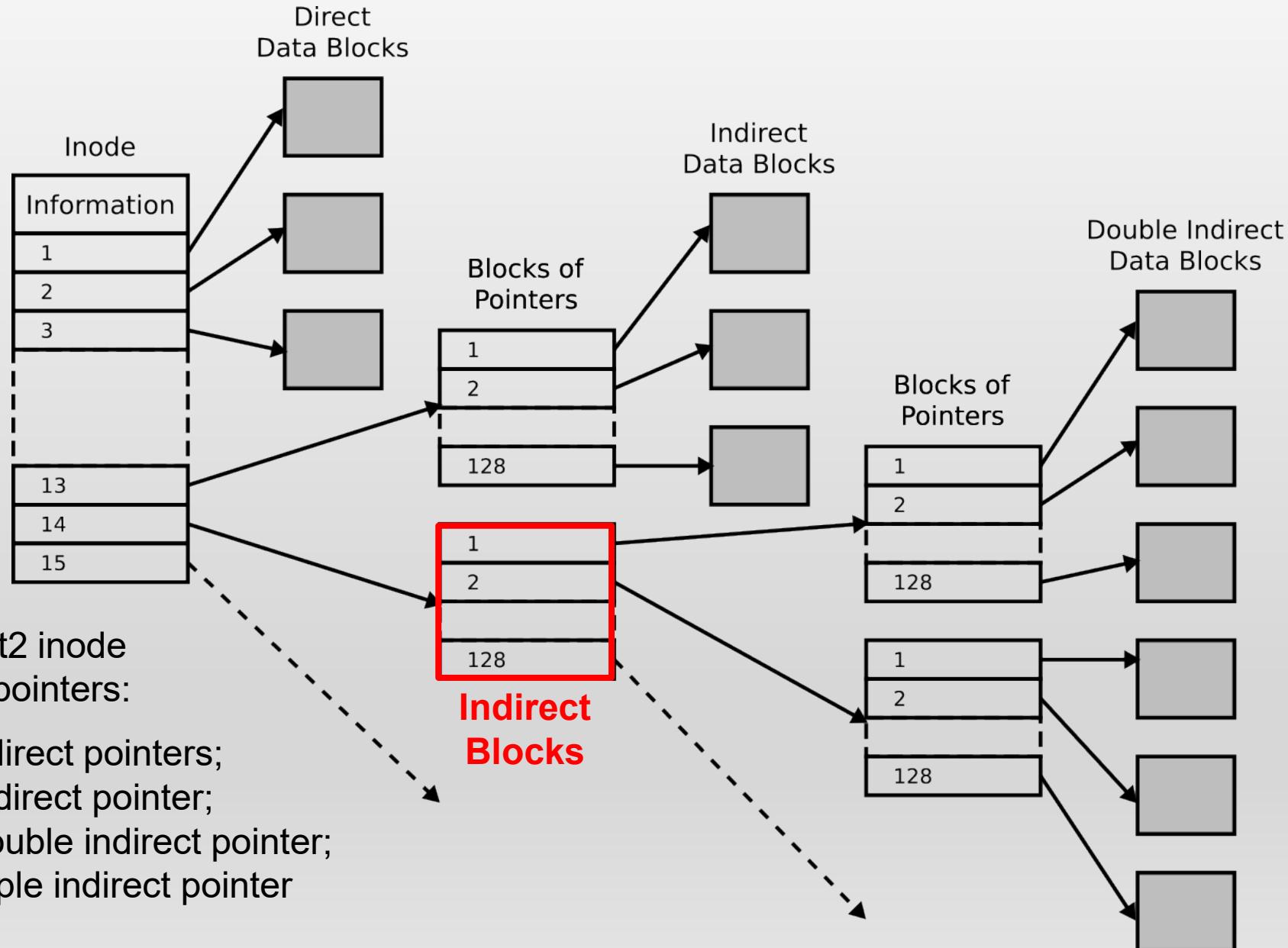
- **Inode (I)**: tracks “**everything**” about a **file / directory**.
 - Each inode is referenced by an **i-number (low-level name)**.
 - Given an i-numbers, the inode can be located.
 - An inode keeps which data block(s) are used for a file / dir.
 - **Inode Table**: the **collection** of all inodes.
- **i-bmap**: tracks which **inode** is **allocated**.
- **d-bitmap**: tracks which **data block** is **allocated**.
- **Superblock (S)**: tracks a **file system**.

The Inode Table (Closeup)																	
				iblock 0	iblock 1	iblock 2	iblock 3	iblock 4									
Super	i-bmap	d-bmap		0 1 2 3 16 17 18 19 32 33 34 35 48 49 50 51 64 65 66 67	4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 68 69 70 71	8 9 10 11 24 25 26 27 40 41 42 43 56 57 58 59 72 73 74 75	12 13 14 15 28 29 30 31 44 45 46 47 60 61 62 63 76 77 78 79										
0KB	4KB	8KB	12KB	16KB	20KB	24KB	28KB	32KB									

File Organization: Inode (1/3)

- The most important design of the inode:
 - *How it refers to where data blocks are.*
- One simple approach would be to have one or more
 - **direct pointers** (each refers to one data block).
 - Challenge: Hard to support files of **big** sizes.
- **Multi-Level Index**
 - **Direct Pointer**: points to a **data block** explicitly.
 - **Indirect Pointer**: points to an **indirect block** that holds (multiple) pointers to **data blocks**.
 - **Double Indirect Pointer**: points to pointers to indirect blocks.
 - **Triple Indirect Pointer**: points to pointers to pointers to indirect blocks.

File Organization: Inode (2/3)



File Organization: Inode (3/3)

- An inode tracks everything **except** “file name” (**why?**).

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Directory Organization

- A **directory** is a **special type of file**.
 - Each directory is also associated with an **inode number**.
 - A directory contains a list of (**file name, inode number**) pairs in its corresponding data block(s).

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a.pretty_long_name

. = current directory

.. = parent directory

strlen = length of the file name (including '\0')

reclen = actual space for an entry (used when deletion)

Free Space Management

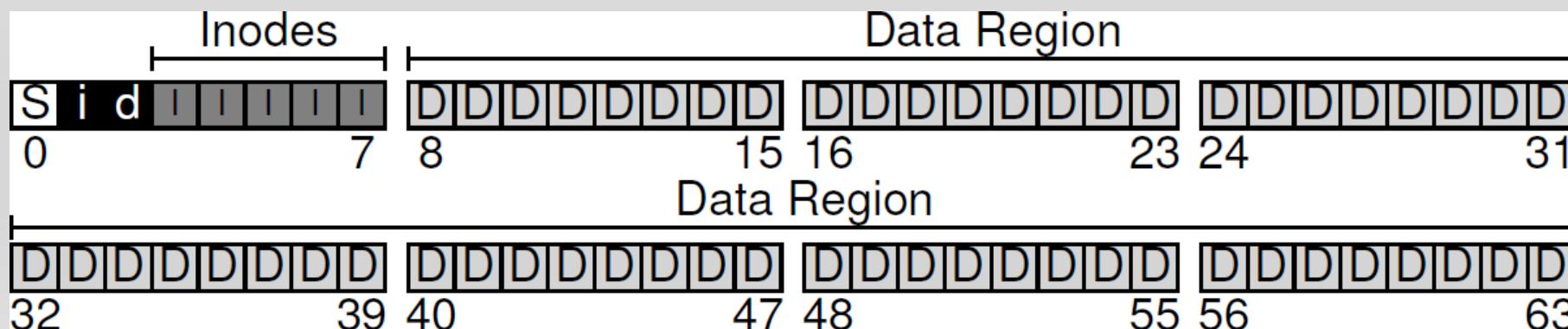
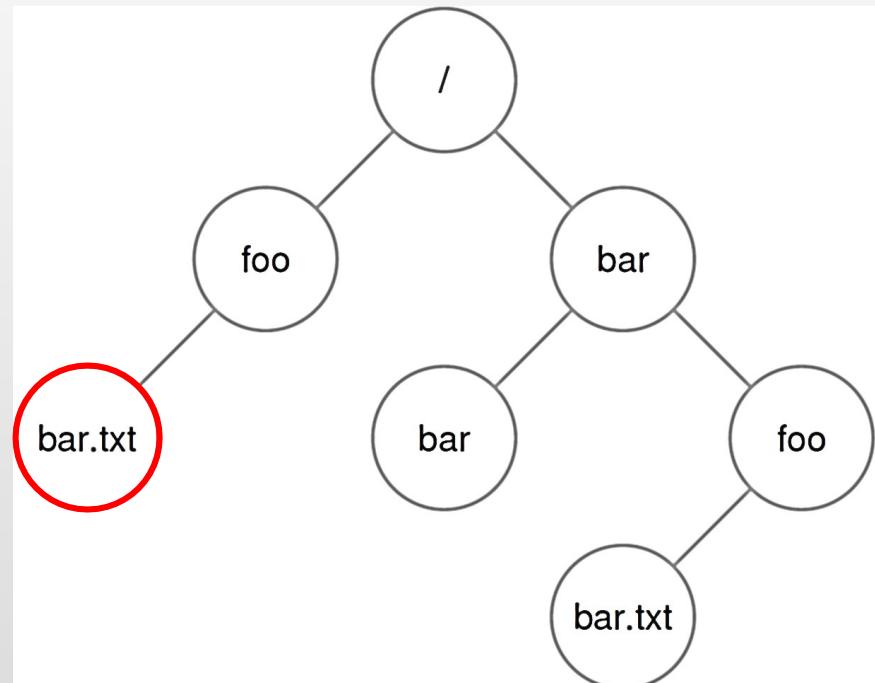
- A file system must track which inodes and data blocks are **allocated** or not:
 - **Bitmap** is one way for free space management.
 - 0: free; 1: used
 - Other structures, e.g., **free list** and **B-tree**, are feasible.
 - There is always a **trade-off** between time and space.
 - **Pre-allocation** may also be used.
 - Strategy: Always looking for a sequence of free blocks (say 8).
 - – A portion of the file will be contiguous on the disk (better performance).

			The Inode Table (Closeup)																			
			iblock 0				iblock 1				iblock 2				iblock 3				iblock 4			
Super	i-bmap	d-bmap	0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67
			4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71
			8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75
			12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79

Exercise

- Question: Can you locate the data block(s) for a file?
 - The root inode number must be “well known” (e.g., `inode #2`).

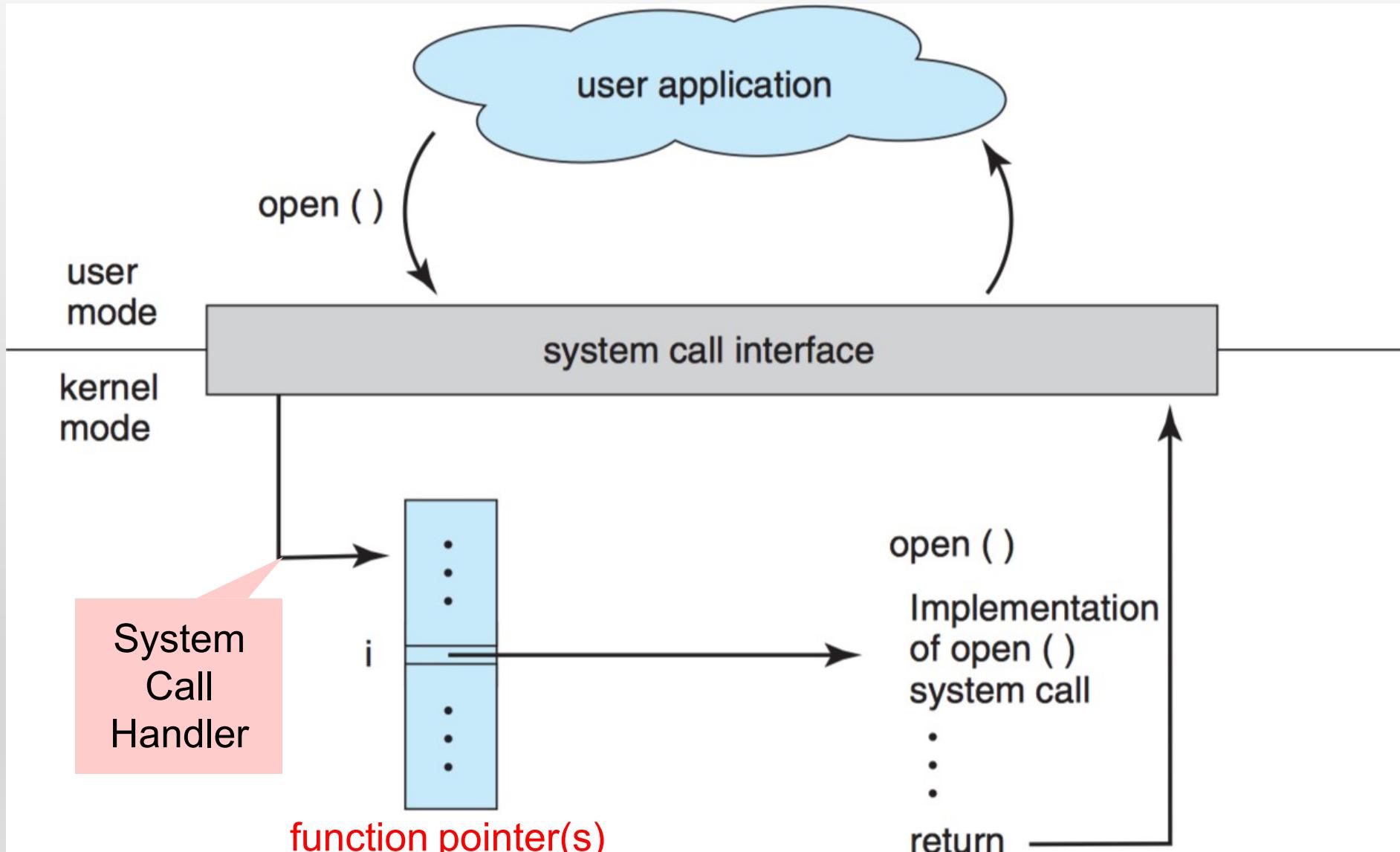
/foo/bar.txt



File System Interface

- File system interface includes:
 - Creating files;
 - Reading/writing files;
 - Renaming files;
 - Getting information about files;
 - Removing files;
 - Managing directories;
 - Linking files/directories;
 - Mounting/unmounting a file system.
- The file system interface uses (or wraps) the OS **system calls** for file/directory management.
 - We focus on UNIX.

System Calls



Creating Files (1/2)

- The system call `open()` is to **create** or **open** a file:
 - 1st argument: file name (absolute or relative pathname)
 - 2nd argument:
 - `O_CREAT`: creates a file;
 - `O_WRONLY`: only write is allowed;
 - `O_TRUNC`: truncate to zero size if a file exists.
 - 3rd argument: specifies permissions (readable or writable).

```
int fd = open( "foo", O_CREAT | O_WRONLY | O_TRUNC,  
              S_IRUSR|S_IWUSR);
```

- On success, a **file descriptor** is returned
 - A pointer for **subsequent accesses** (function calls) to a file.
 - In UNIX, it's just an integer.

Creating Files (2/2)

- **File descriptors** are managed on a **per-process basis** by the operating system .
- For example, the UNIX systems (xv6 kernel) must keep some kind of structure in the struct proc:
 - A **simple array** (with a maximum of NOFILE open files) tracks which files are opened on a **per-process** basis.
 - Each entry of the array is just a **pointer** to a struct file, which tracks the information of the “**open file**” being used.

```
struct proc {  
    ...  
    struct file *ofile[NOFILE]; // Open files  
    ...  
};
```

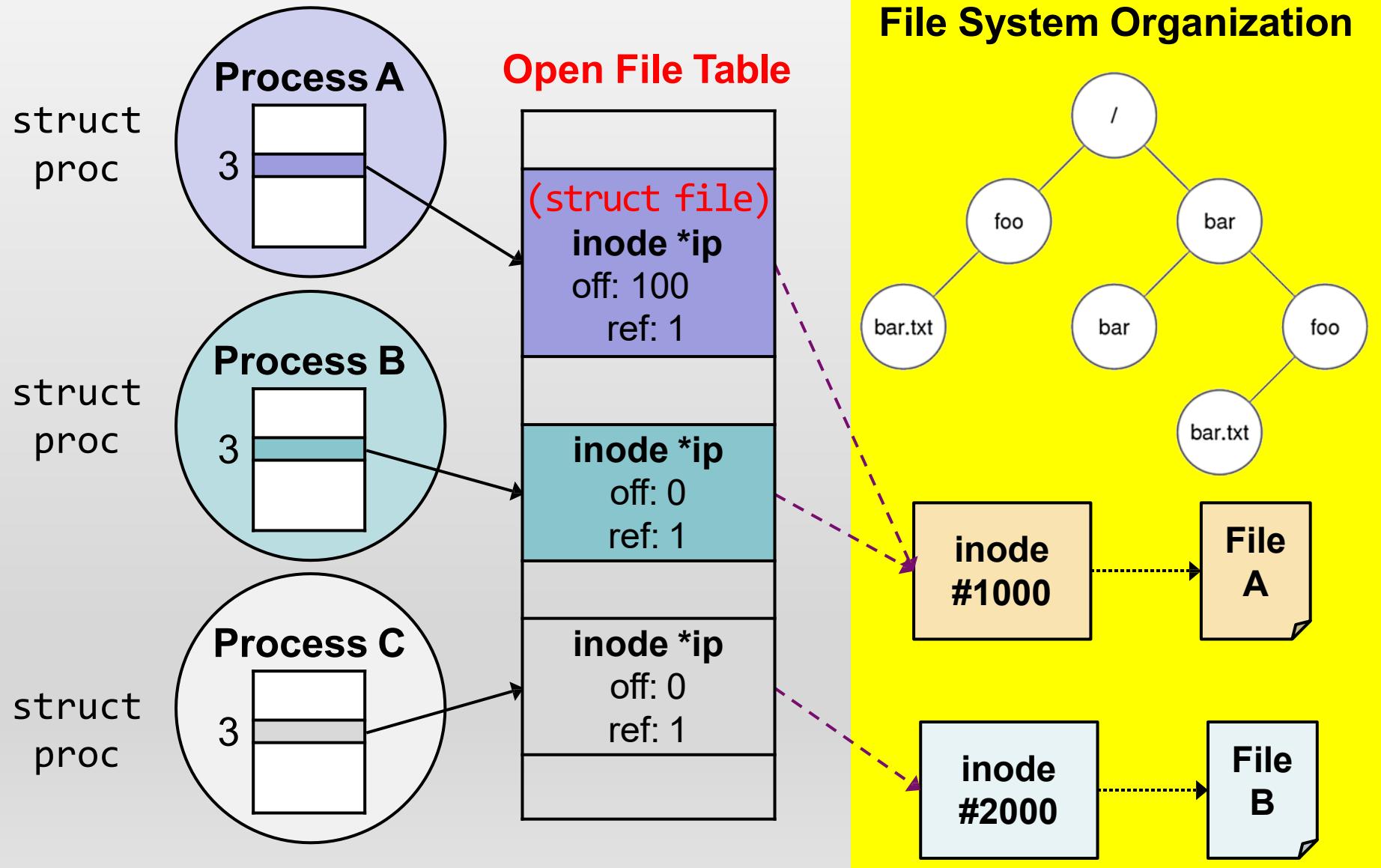
Management of Open Files (1/2)

- The `struct file` represents an **open file**:
 - The `struct file` (an open file) is referenced by a **process**.
 - The `readable/writable` specifies **read/write permissions**.
 - The `off` keeps the “current” offset, where the next read/write should take place, for this open file.
 - The **actual file** is referenced by the `struct inode`.
- All open files are kept in an **open file table** by OS.

```
struct proc {  
    ...  
    struct file *ofile[N];  
    // open files  
    ...  
};
```

```
struct file {  
    struct inode *ip;  
    char readable;  
    char writable;  
    uint off;  
    int ref;  
};
```

Management of Open Files (2/2)



Reading and Writing Files (1/4)

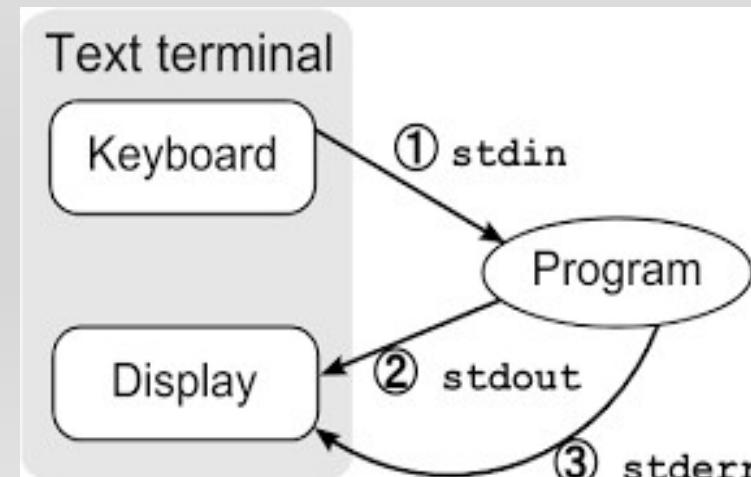
- How does a process actually read or write a file?
- Exercise: Let's use the **strace** tool to trace every system call made by reading (cat) the file foo:

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE) = 3
read(3, "hello\n", 4096)          = 6
write(1, "hello\n", 6)            = 6
hello
read(3, "", 4096)                = 0
close(3)                         = 0
...
```

Reading and Writing Files (2/4)

```
open("foo", O_RDONLY|O_LARGEFILE) = 3
```

- First, the **open()** system call opens a file for reading:
 - `O_RDONLY`: read only (writing is not allowed)
 - `O_LARGEFILE`: 64-bit offset is used.
- **open() returns** a file descriptor of **3**.
 - Each running process already has three “open files”:
 - Standard Input: 0
 - Standard Output: 1
 - Standard Error: 2



Reading and Writing Files (3/4)

```
read(3, "hello\n", 4096) = 6
write(1, "hello\n", 6)    = 6
hello
read(3, "", 4096)        = 0
close(3)                 = 0
```

- `read()` are called for **many times** to read the file:
 - 1st argument: file descriptor
 - 2nd argument: buffer where the results are stored
 - 3rd argument: size of the buffer
- `write()` is called to display output on screen (**fd=1**).
- `close()` is called when reaching the EOF.
- Writing a file? `open() → write() → close()`

Reading and Writing Files (4/4)

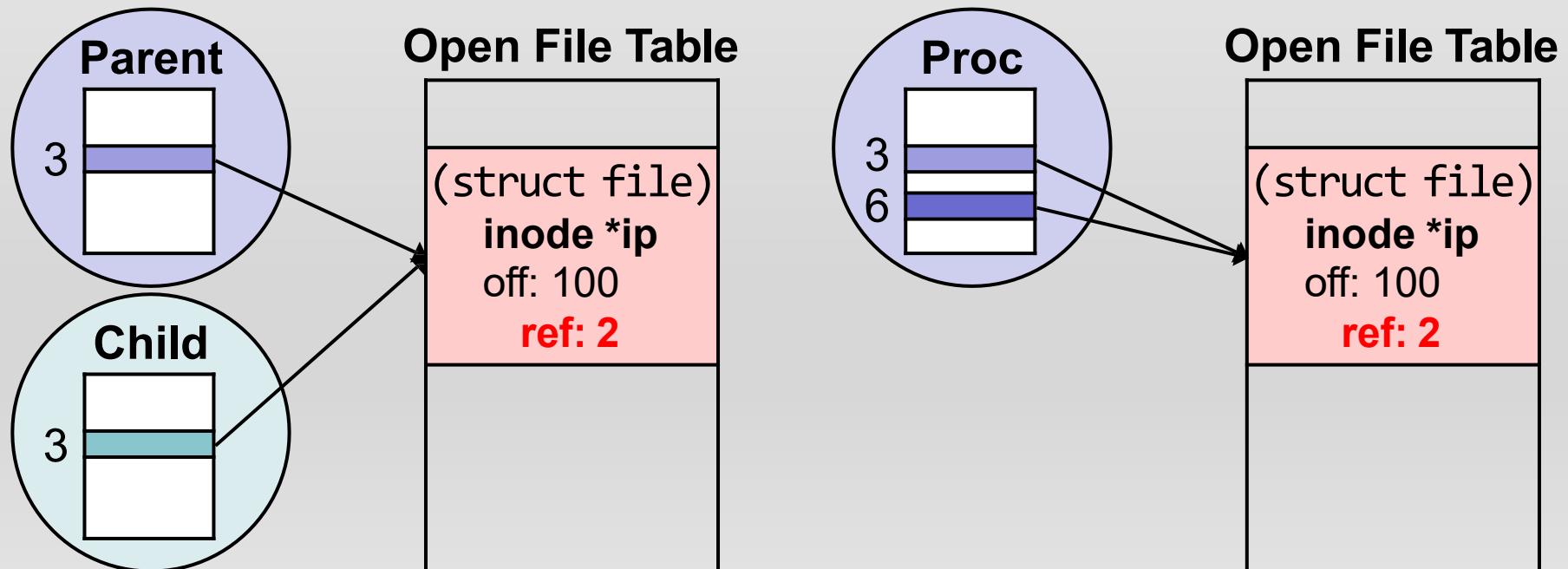
- lseek() is to read or write to a **specific offset** within a file (*rather than from the beginning to the end*).

```
off_t lseek(int fd, off_t offset, int whence);
```

- 1st argument: file descriptor
- 2nd argument: positions the **offset** to a particular location within a file (for subsequent reads/writes).
 - lseek() has **nothing** to do with a disk seek!
- 3rd argument: specifies how lseek() is performed.
 - SEEK_SET: set to offset bytes from the **beginning**
 - SEEK_CUR: set to **current location** plus offset bytes
 - SEEK_END: set to offset bytes from the **end**

Shared File Table Entries

- In many cases, the mapping of **file descriptor** to an **entry in the open file table** is a **one-to-one** mapping.
- An entry in the open file table can be **shared** when
 - A parent process creates a child process with `fork()`;
 - A process creates a few file descriptors that refers to the same file with `dup()` or its cousins `dup2()` and `dup3()`.



Forcing Writes

- For performance, the file system **buffers writes** in memory (e.g., for 5 sec or 30 sec).
- The `fsync()` system call **forces** all dirty (i.e., not yet written) data to the disk.

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,  
              S_IRUSR|S_IWUSR);  
assert(fd > -1);  
int rc = write(fd, buffer, size);  
assert(rc == size);  
rc = fsync(fd);  
assert(rc == 0);
```

Information of Files

- The file system keeps a fair amount of **information** about each **file** it is storing.
 - `stat()` or `fstat()` calls can be used to see the metadata.

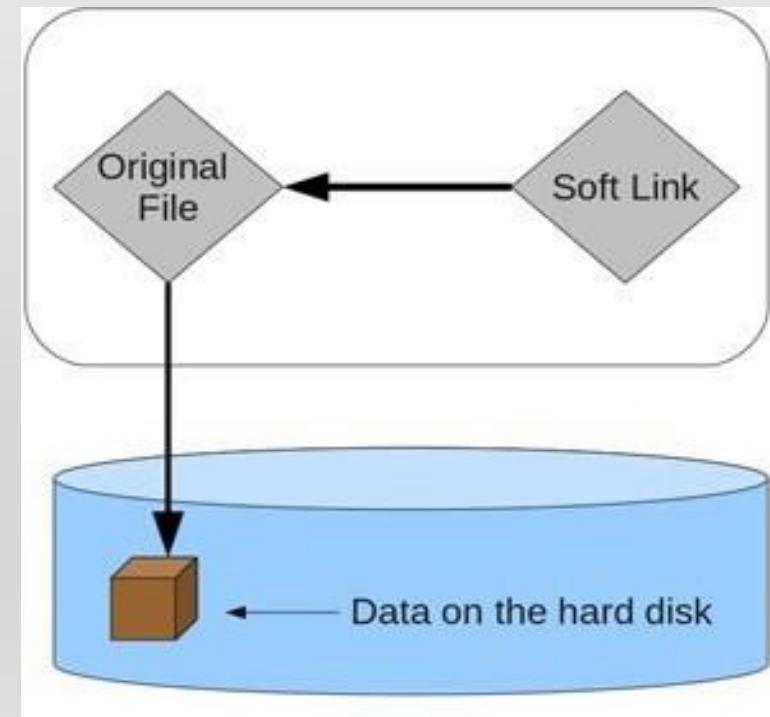
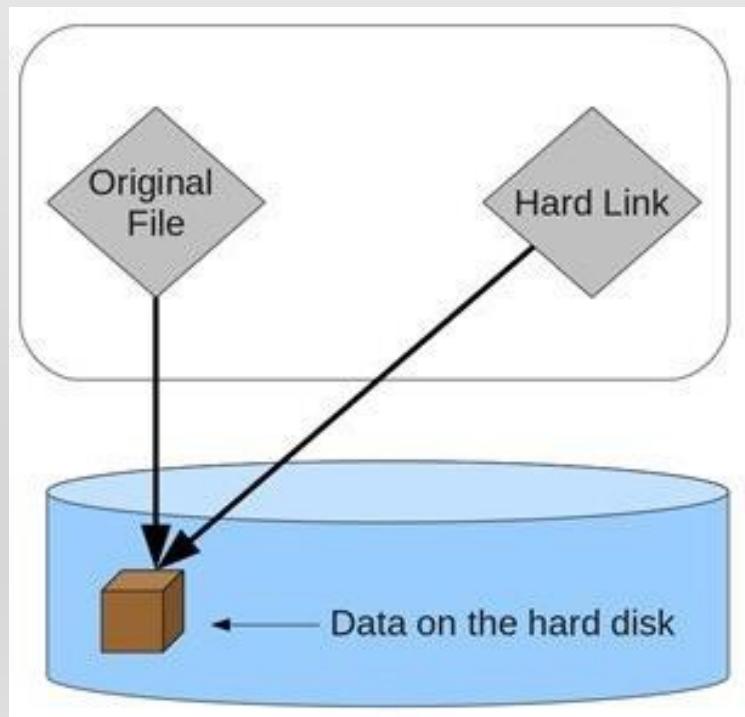
```
struct stat {  
    dev_t st_dev; /* ID of device containing file */  ino_t st_ino;  
                                /* inode number */  
    mode_t st_mode; /* protection */  
    nlink_t st_nlink; /* number of hard links */  
    uid_t st_uid; /* user ID of owner */  gid_t st_gid;  
                                /* group ID of owner*/  
    dev_t st_rdev; /* deviceID (if special file) */  off_t  
    st_size;        /* total size, in bytes */  
    blksize_t st_blksize; /* blocksize for filesystem I/O */  blkcnt_t  
    st_blocks;       /* numberofblocks allocated */  time_t st_atime;  
                                /* time of last access */  
    time_t st_mtime; /* time of last modification */  
    time_t st_ctime; /* time of last status change */};
```

Summary of File System Operations

File System Operations	System Calls
Creating a file	<code>open()</code>
Reading a file	<code>read()</code>
Writing a file	<code>write()</code> , <code>fsync()</code>
Seeking to an offset	<code>lseek()</code>
Renaming a file	<code>rename()</code> (often an atomic call)
Getting file information	<code>stat()</code> or <code>fstat()</code>
Removing a file	<code>unlink()</code>
Making a directory	<code>mkdir()</code>
Reading a directory	<code>opendir()</code> , <code>readdir()</code> , <code>closedir()</code>
Removing a directory	<code>rmdir()</code> (must be empty)

Links

- File systems allow **links** to create multiple names (aliases) for the same file
 - **Hard Link**: holds the **inode number** of a file.
 - **Symbolic/Soft Link**: holds the **pathname** to a file.



Recall: Directory Organization

- A **directory** is a **special** type of **file**.
 - Each directory is also associated with an **inode number**.
 - A directory contains a list of (**file name**, **inode number**) pairs in its corresponding data block(s).

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a.pretty_lon

. = current directory

.. = parent directory

strlen = length of the file name (including '\0')

reclen = actual space for an entry (used when deletion)

Hard Link

- Hard link (`ln`) creates **a new entry in the directory** to refer to the same inode number of the original file.

```
prompt> echo hello > file  
prompt> cat file  
hello  
prompt> ln file hard_link  
prompt> cat hard_link  
hello
```

```
prompt>  
ls -i file hard_link  
67158084 file  
67158084 hard_link
```

```
prompt> rm file  
removed ‘file’  
prompt> cat hard_link  
hello
```

Create a Hard Link Show inode Numbers Remove a File

- The inode has a **reference count** that keeps track of how many hard links refer to it.
 - Only when the reference count is **zero**, the file system frees the inode and related data blocks.
 - This explains why **unlink()** is called when **removing** a file.

Recall: File Organization: Inode

- An inode tracks everything **except** “file name” (**why?**).

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

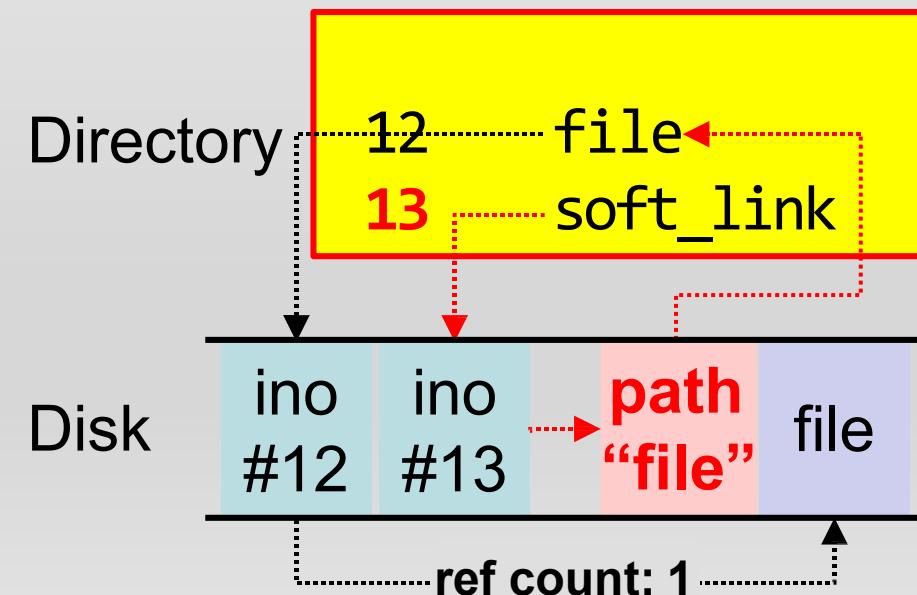
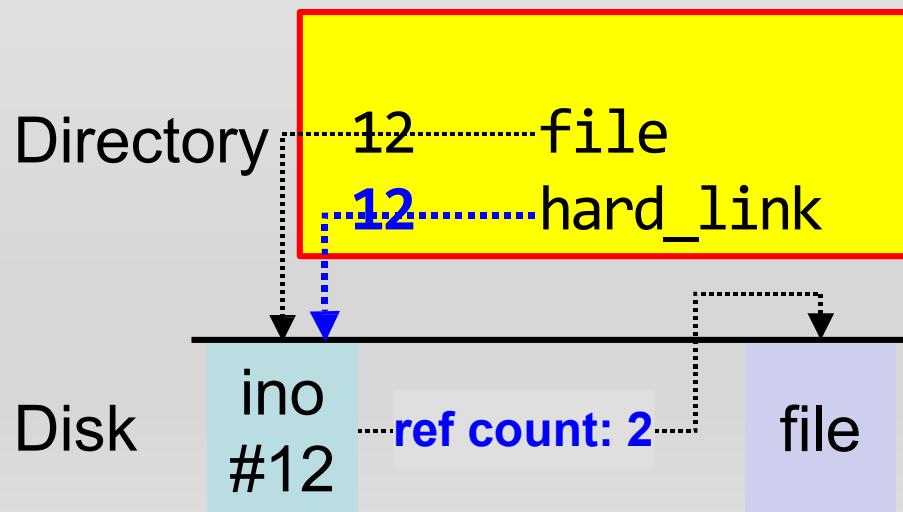
Symbolic/Soft Link

- Hard links are **limited**:
 - Cannot link to a **directory** (to avoid creating a cycle).
 - Cannot link to a **different partition** (only within a file system).
- **Symbolic Link** (`ln -s`)
 - **It is a special file** with its own inode number.
 - It holds a **pointer** to link but may cause **dangling reference**.

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln    file soft_link
prompt> cat soft_link
hello
prompt> rm file
prompt> cat soft_link
cat: soft_link: No such file or directory
```

Hard Link vs. Soft Link

- File systems allow **links** to create multiple names (aliases) for the same file
 - Hard Link:** holds the **inode number** of a file
 - By only creating a **new directory entry**.
 - Symbolic/Soft Link:** holds the **pathname** to a file
 - By creating a **new file of special type**.
 - Three types of file: 1) Data File; 2) Directory File; 3) Soft Link File.



Mounting a File System

- Final Step: Set up a file system to make it run.
- **Mounting** (mount) a file system:
 - Create a **mountpoint**
 - Paste a **file system** onto the directory tree at **that point**

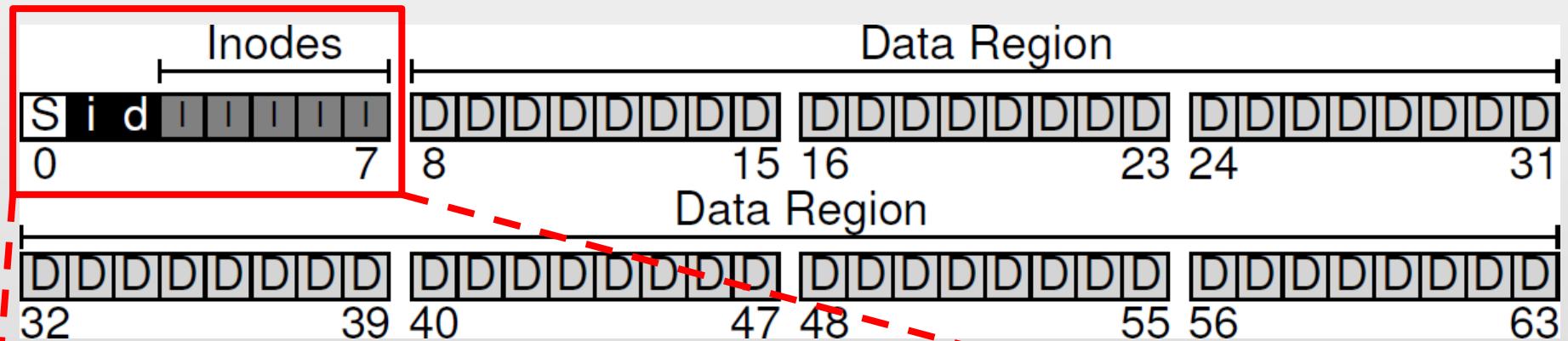
```
prompt> mount -t ext3 /dev/sda1 /home/users
```

- You can have **multiple file systems** on the same machine, and mounts all file systems into one tree!

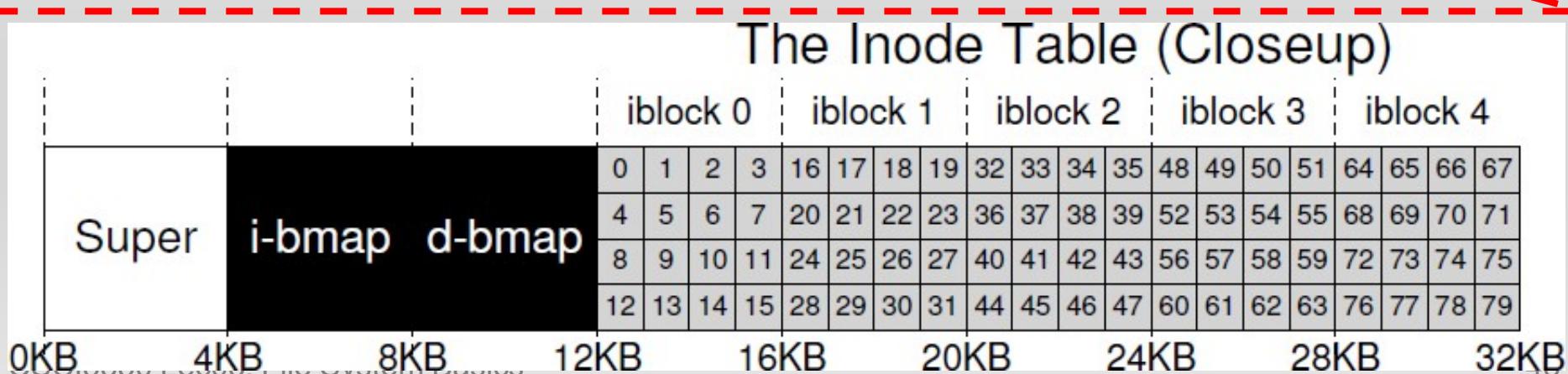
```
/home/users/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```

UNIX File System

- The organization we have learnt is a simplified version of a typical **UNIX file system**:



- **Metadata Region:** tracks data and file system information.
- **Data Region:** stores user data and occupies most space.

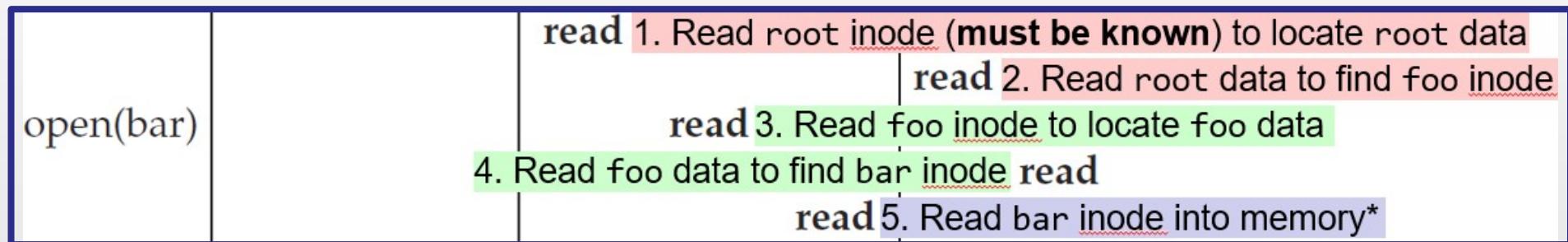


Access Path: Read (1/2)

- Example: Read a file /foo/bar
 - Traverse the pathname to locate the requested inode:
 - root → foo → bar

	data inode bitmap bitmap	root foo bar inode inode inode	root foo bar bar bar data data data[0] data[1] data[2]
open(bar)		read 1. Read root inode (must be known) to locate root data	read 2. Read root data to find foo inode
		read 3. Read foo inode to locate foo data	read 4. Read foo data to find bar inode
read()			read 5. Read bar inode into memory*
read()	6. Read bar inode to locate data 7. Read data block of bar 8. Update timestamp of bar inode	read write read write	read read
read()		read write	read

Access Path: Read (2/2)



- **Note 1:** The amount of I/O generated by the `open()` is **proportional** to the length of the pathname.
 - **Large directories** would make this worse. (Why? Step 4)
- **Note 2:** The following work is also needed but not listed:
 - **Step 5** also needs to check permissions; allocate a file descriptor for this process; create an entry in the open-file table; return the file descriptor to the user.

6. Read bar inode to locate data

- **Note 3:** The read will further update the **in-memory open file table** to maintain the **file offset** for this file descriptor.
 - Such that the next read will read the subsequent file block.

Access Path: Write (1/2)

- Example: Create (write) a new file /foo/bar

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
10 I/Os			read		read	read	read			
	create (/foo/bar)			read (find a free inode) write (mark it allocated) (link the file name and the inode in dir)			read (initialize)			
				read (update)	write	write				
5 I/Os	write()		read (find a free data block) write (update data bitmap)			read				
5 I/Os	write()	read write				write				
5 I/Os	write()	read write				write				
						read				
							write			
								write		
									write	

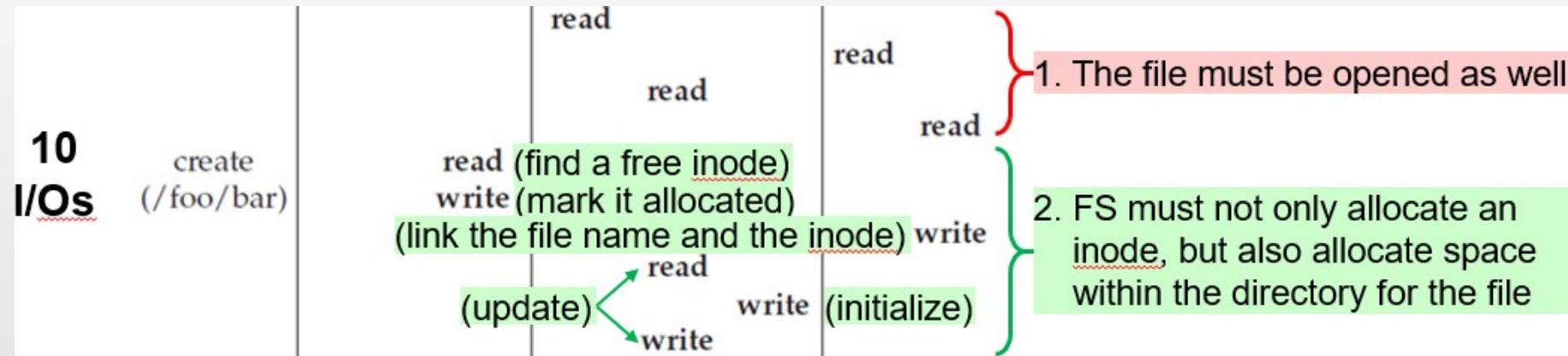
Annotations:

- 1. The file must be opened as well
- 2. FS must not only allocate an inode, but also allocate space within the directory for the file

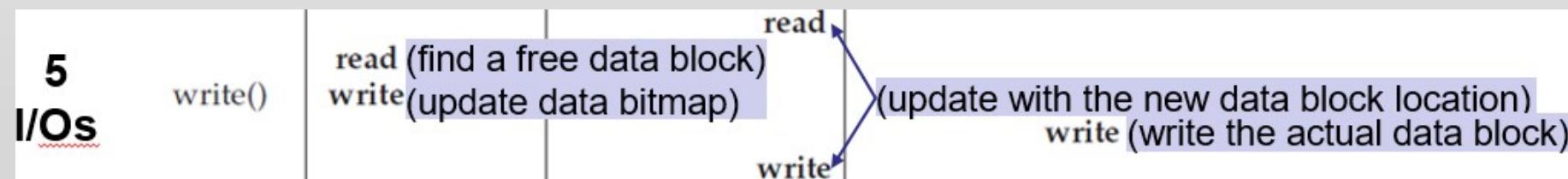
Legend:

- read
- write
- read (find a free inode)
- write (mark it allocated)
- (link the file name and the inode in dir)
- read (update)
- write (initialize)
- read (find a free data block)
- write (update data bitmap)
- (update with the new data block location)
- write (write the actual data block)

Access Path: Write (2/2)



- **10 I/Os** are needed to walk the pathname and create file.
 - If the directory needs to grow, **additional I/Os** are needed.
 - – i.e., to the data bitmap, and the new directory block.



- Each data block write logically generates 5 I/Os.
 - If `write()` involves indirect pointers, **more I/Os** are needed as well.

Caching and Buffering

- Like in UNIX file system, reading and writing files can be **expensive**, incurring many I/Os to the (slow) disk.
- Most file systems leverage the **system memory** to:
 - **Cache** some important or popular blocks
 - To **avoid repeated reads** to the same blocks
 - To **avoid performing hundreds of reads** to open a file with long pathname (e.g., `/1/2/3/.../100/file.txt`).
 - **Buffer** a number of writes (for 5~30 seconds)
 - To **allow writes** to the same location (**in memory**)
 - To **batch updates** into a smaller set of I/Os
 - To **allow rescheduling** of I/Os
 - **Cache/buffer trades reliability for performance!**
 - But **not** everyone likes it; some applications (e.g., databases) require frequent `fsync()` to **avoid losing data** kept in the write buffer.

Revisit File System Organization (1/2)

- “Old UNIX File System” by Ken Thompson:



- The **super block (S)** contained the file system information:
 - How big the volume is, how many inodes there are, a pointer to the head of a free list of blocks, and so forth.
- The inode region contained all **inodes** for the file system.
- Most of the disk space was taken up by **data blocks**.
- **Problem 1: Poor Performance**
 - The file system was delivering only 2% of disk bandwidth, because of **expensive disk positioning costs**.
 - The data blocks of a file were often **very far away** from its inode.
 - An expensive seek was induced whenever one first read the inode, and then read the file system block.

Revisit File System Organization (2/2)

- “Old UNIX File System” by Ken Thompson:



- **Problem 2: Fragmentation**

- **External Fragmentation**

- Free block space is **not contiguous**.
 - A large file may have blocks **scattered** across disk.
 - Disk defragmentation tools may help by **reorganization**.



After writing file E of four blocks



After writing file F of 1/2 block

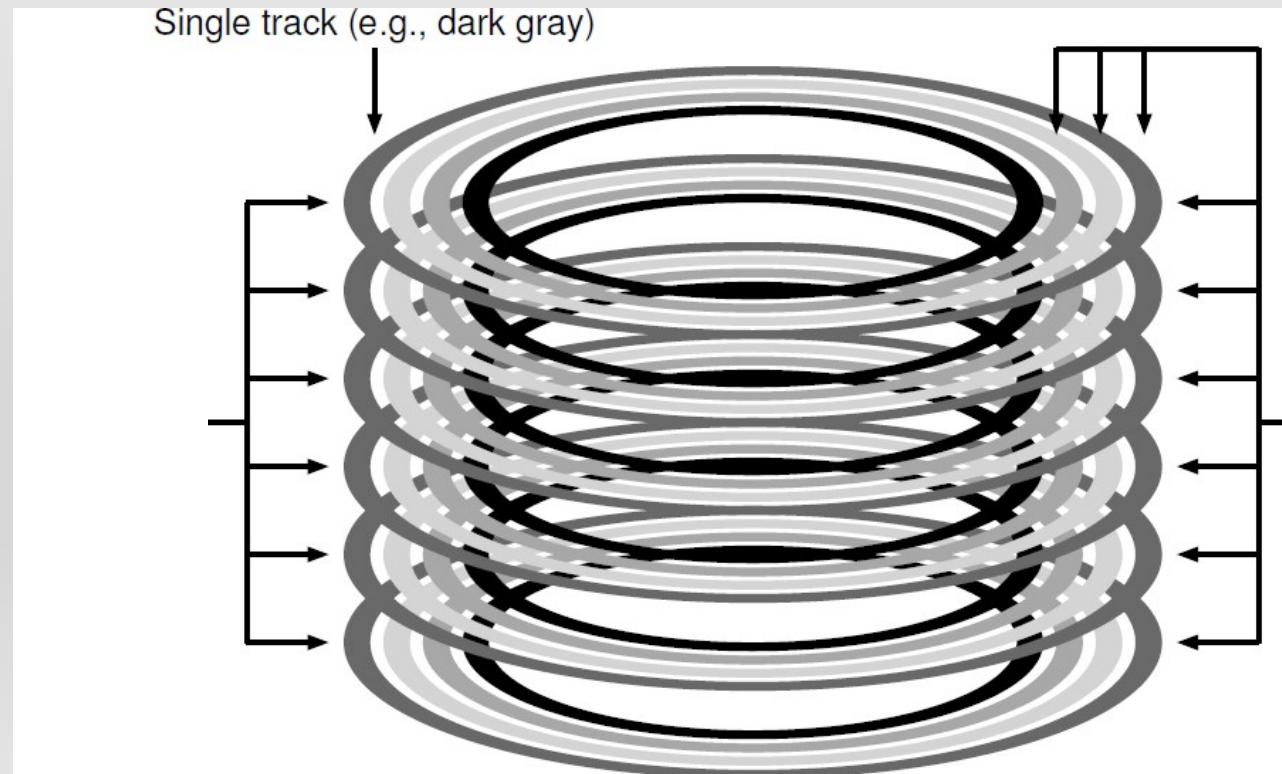
- **Internal Fragmentation**

- Reads/writes are in units of blocks.
 - If a small file cannot cover a block, block space is **wasted**.
 - Smaller blocks may have more **positioning overhead**.

Fast File System (FFS) by Berkeley

- **Goal:** Make the file system structures and allocation policies to be “disk-aware” to improve performance.
 - – By keeping the same file system **interface** (i.e., system calls) but changing the **internal implementation**.
- **Key:** FFS divides disk into **cylinder groups**.

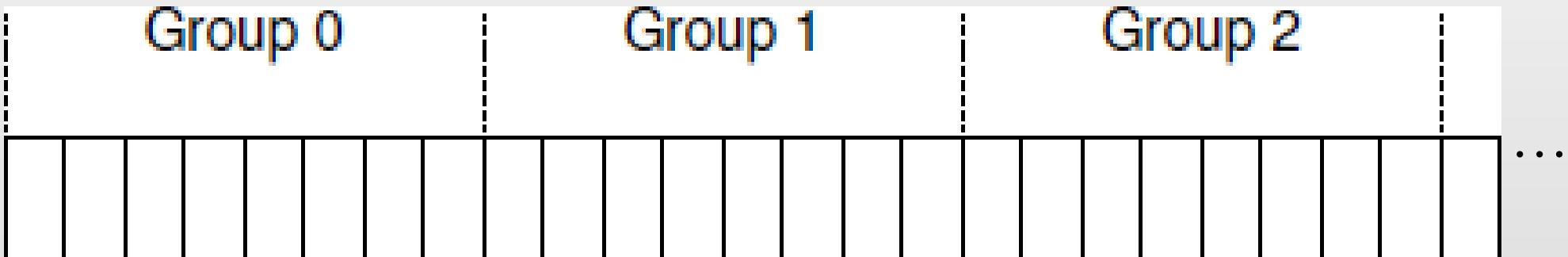
Cylinder:
Tracks at same
distance from
center across
different surfaces
(same color)



Cylinder Group:
A set of N
consecutive
cylinders
(different color)

Fast File System (FFS) (2/3)

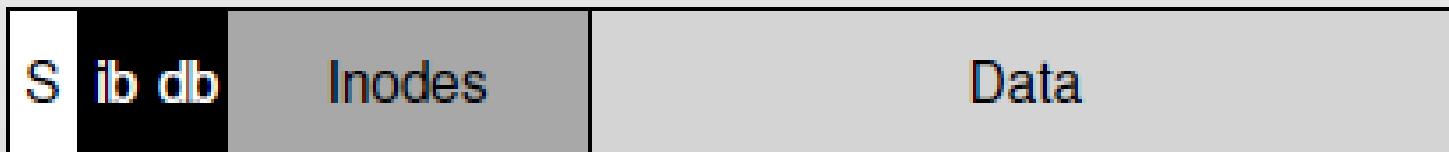
- FFS aggregates N consecutive cylinders into a group, and the disk is of a collection of **cylinder groups**.



- Modern disks do **not** export **cylinder information** for the file system to explore.
- Modern FSs instead organize disk into **block groups**.
 - Each block group is of the **consecutive block addresses** (rather than **consecutive cylinders**).

Fast File System (FFS) (3/3)

- FFS maintains **similar structures for each group**:
 - A copy of superblock (S)
 - Per-group inode bitmap (ib) and data bitmap (db)
 - Per-group inode and data block regions.



- FFS further explores the **data locality** to place files, directories, and associated metadata on disk:
 - *keep related stuff together, keep unrelated stuff far apart*
 - ① Allocate **data blocks** of a file in the same group as its inode
 - ② Place **files** of the same directory in the same group
 - ③ Balance **directories** across groups

That's all for today.

Embedded System

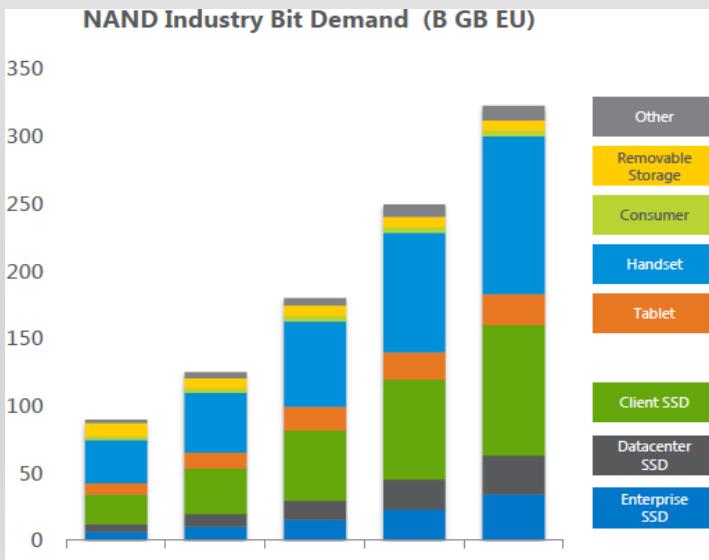
Lecture 09: Flash Memory

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

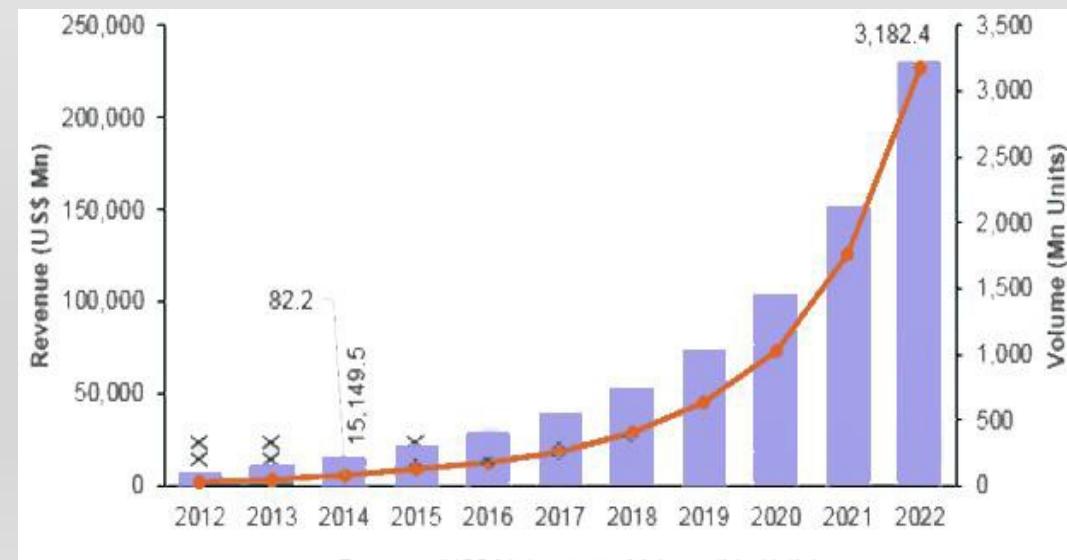
Hong-Yue Technology Research Building 334
R 09:10 - 12:00

Why Flash Memory

- Flash memory is a widely used **memory/storage technology** in today's products.
 - It is a type of **non-volatile memory**.
 - Data can be **persisted** under power loss.
 - **Revenue Growth:** 40% every year!
 - **Volume Price:** ~40% annual reduction!



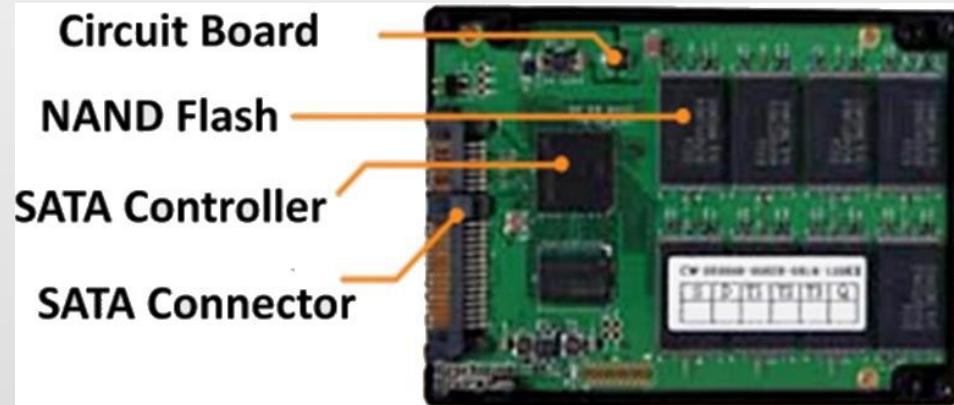
Sources : Micron, Intel And 3D NAND Post [\[link\]](#)



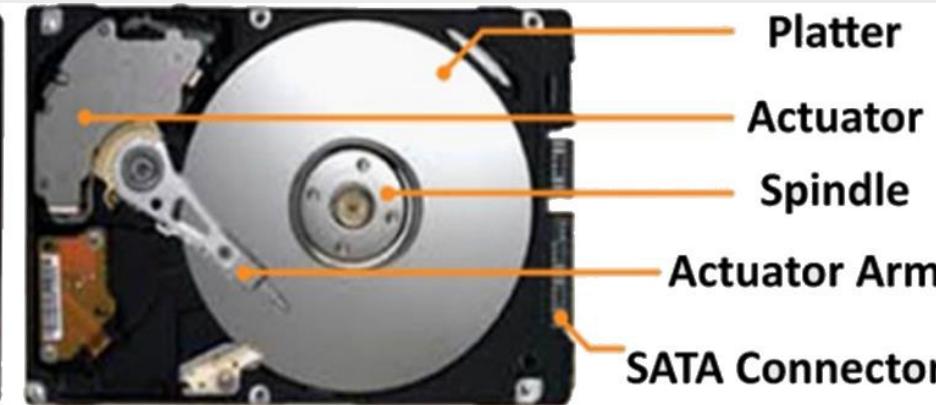
Sources : Global SSD Market to Post CAGR of 41% Until 2022, OriginStorage

Solid-State Drive vs. Hard Disk Drive

Solid-State Drive (SSD)



Hard Disk Drive (HDD)

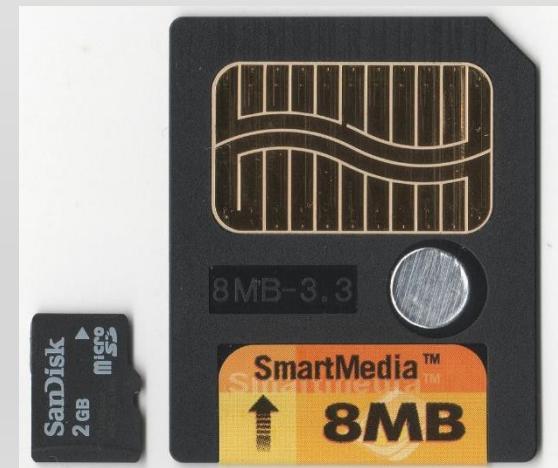


- ✓ Faster performance
- ✓ No vibrations or noise
- ✓ Shock resistance
- ✓ More energy efficient
- ✓ Lighter and smaller

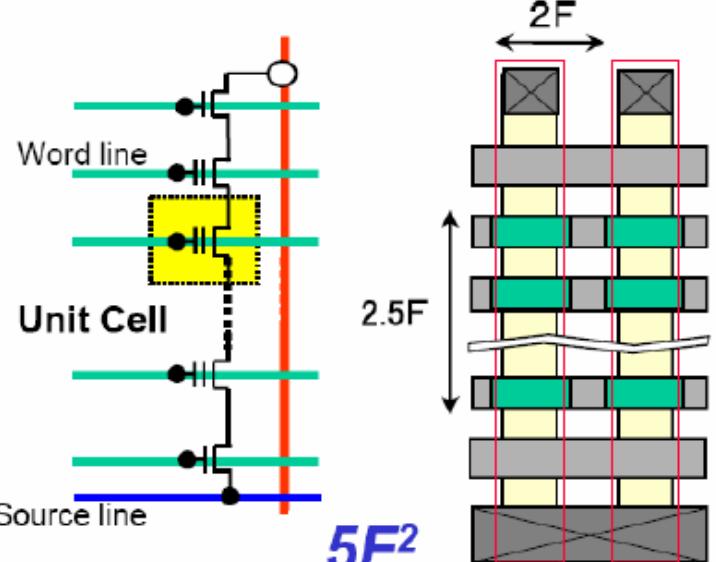
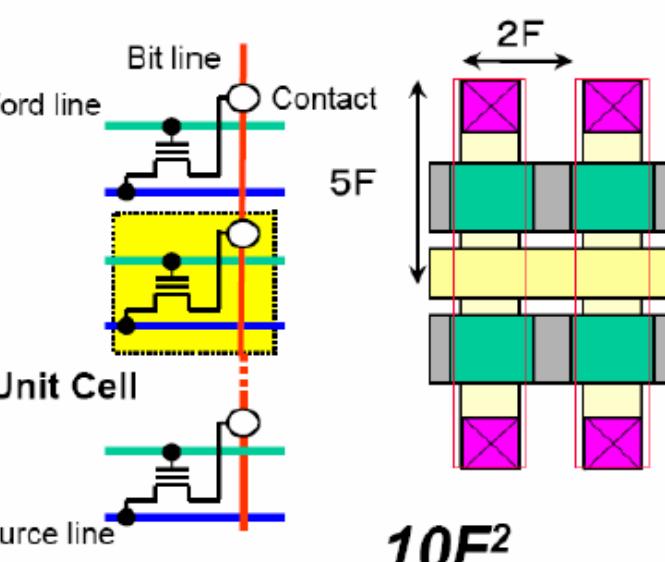
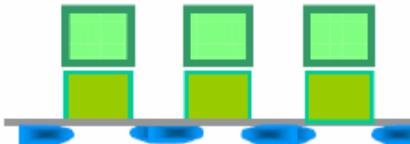
- ✓ Cheaper per GB
- ✓ Reliability

The Greatest Invention in 1990s

- Invented by **Dr. Fujio Masuoka** (舛岡 富士雄) born in 1943, while working for Toshiba around 1980.
- First presented in *IEEE International Electron Devices Meeting*, 1984.
- First **NAND flash** first introduced as
 - **SmartMedia storage** in 1995.
- First **NOR flash** commercialized by Intel in 1998 to replace the read-only memory (ROM) to store **BIOS**

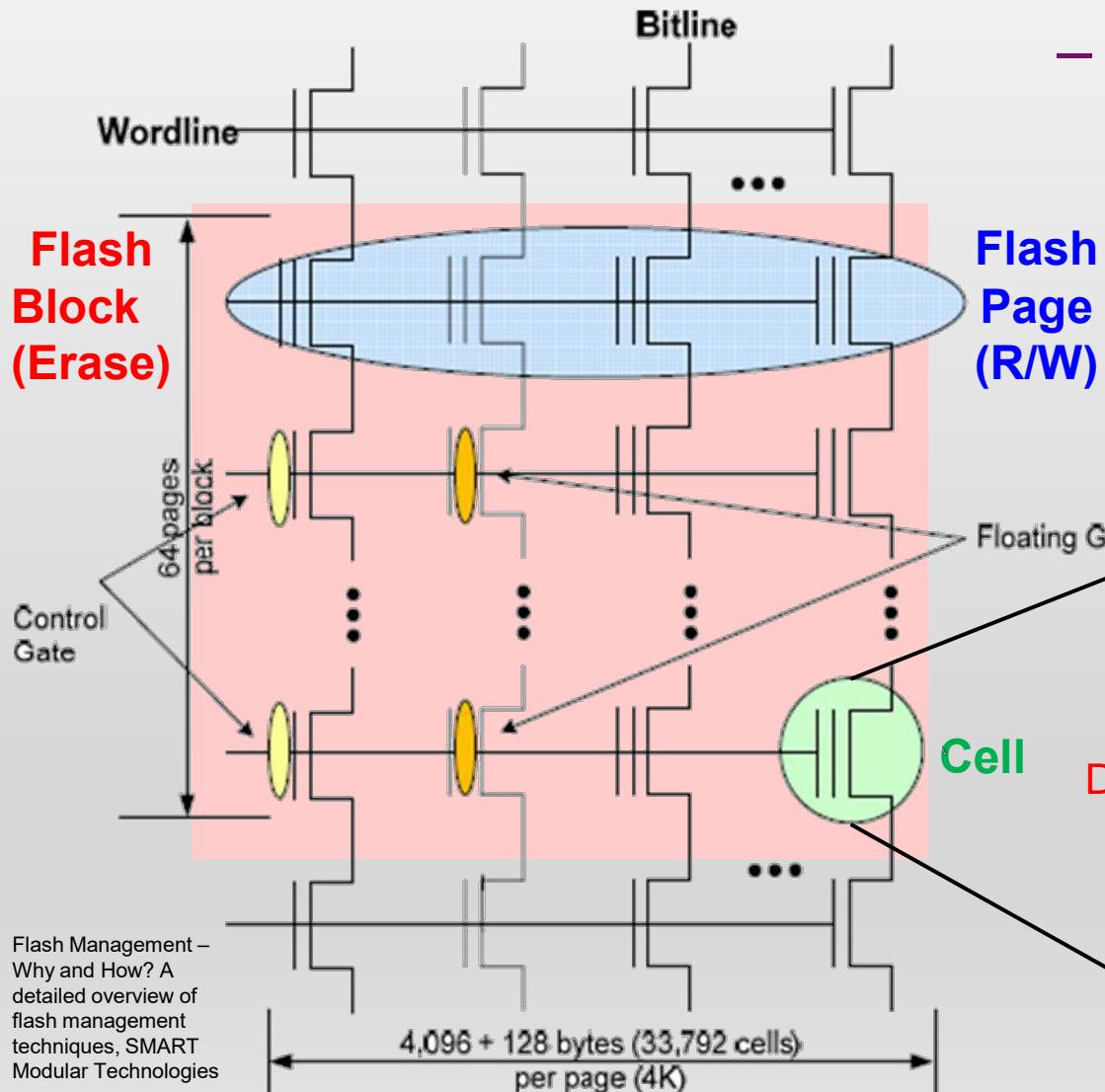


NAND Flash vs. NOR Flash

	NAND	NOR
Cell Array & Size	 <p>Unit Cell Word line Source line</p> <p>2F</p> <p>2.5F</p> <p>$5F^2$</p>	 <p>Bit line Word line Contact Unit Cell Source line</p> <p>2F</p> <p>5F</p> <p>$10F^2$</p>
Cross-section		
Features	<p>Small Cell Size, High Density Low Power & Good Endurance → Mass Storage</p>	<p>Large Cell Current, Fast Random Access → Code Storage</p>

NAND Flash Technology

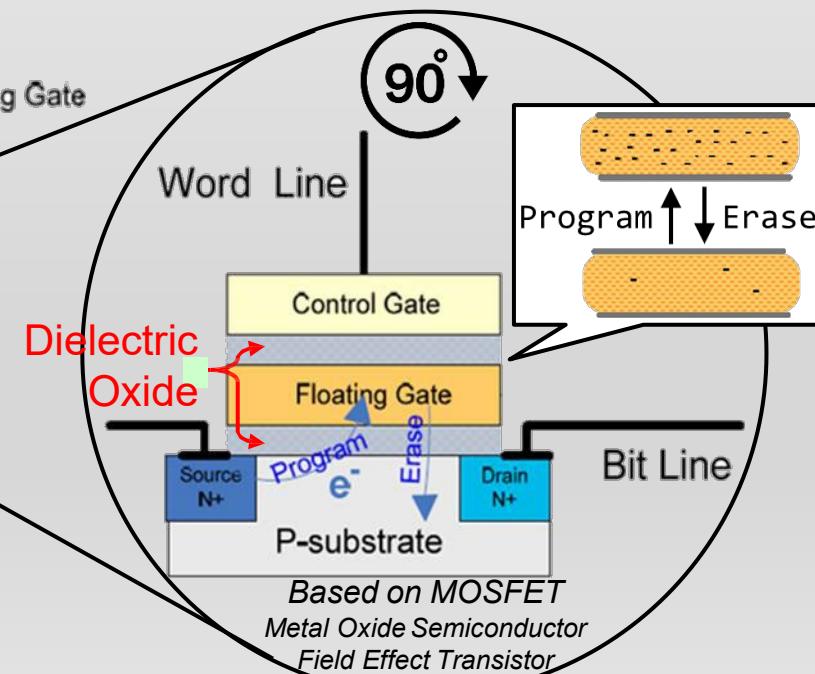
- NAND Flash Array



- NAND Flash Cell

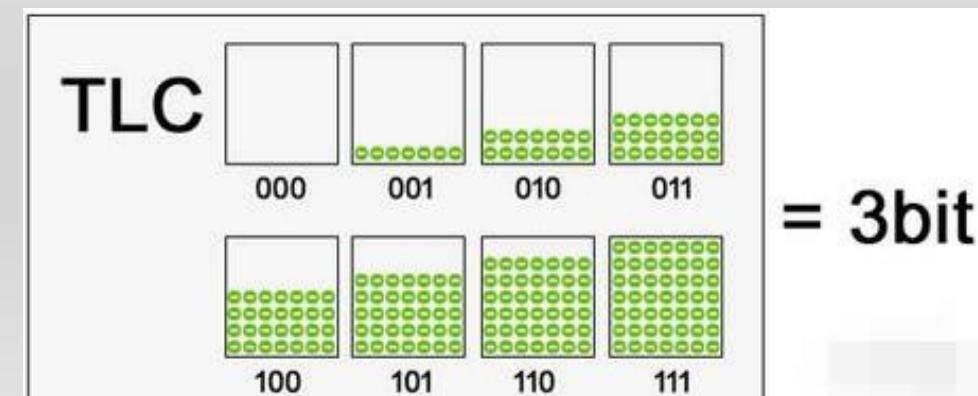
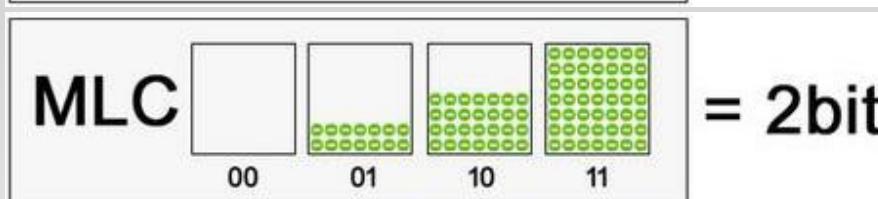
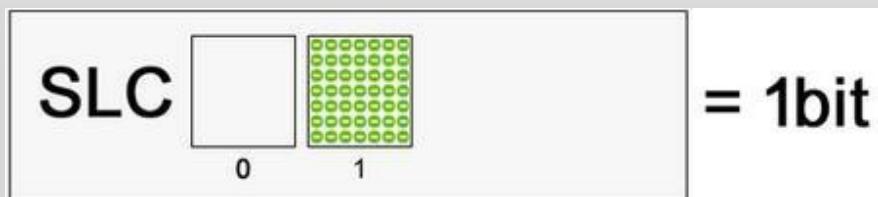
- Floating-Gate Transistor

- **Program:** Inject electrons into FG to raise voltage
 - **Erase:** Remove electrons from FG to lower voltage
 - **Read:** Sense voltage of FG

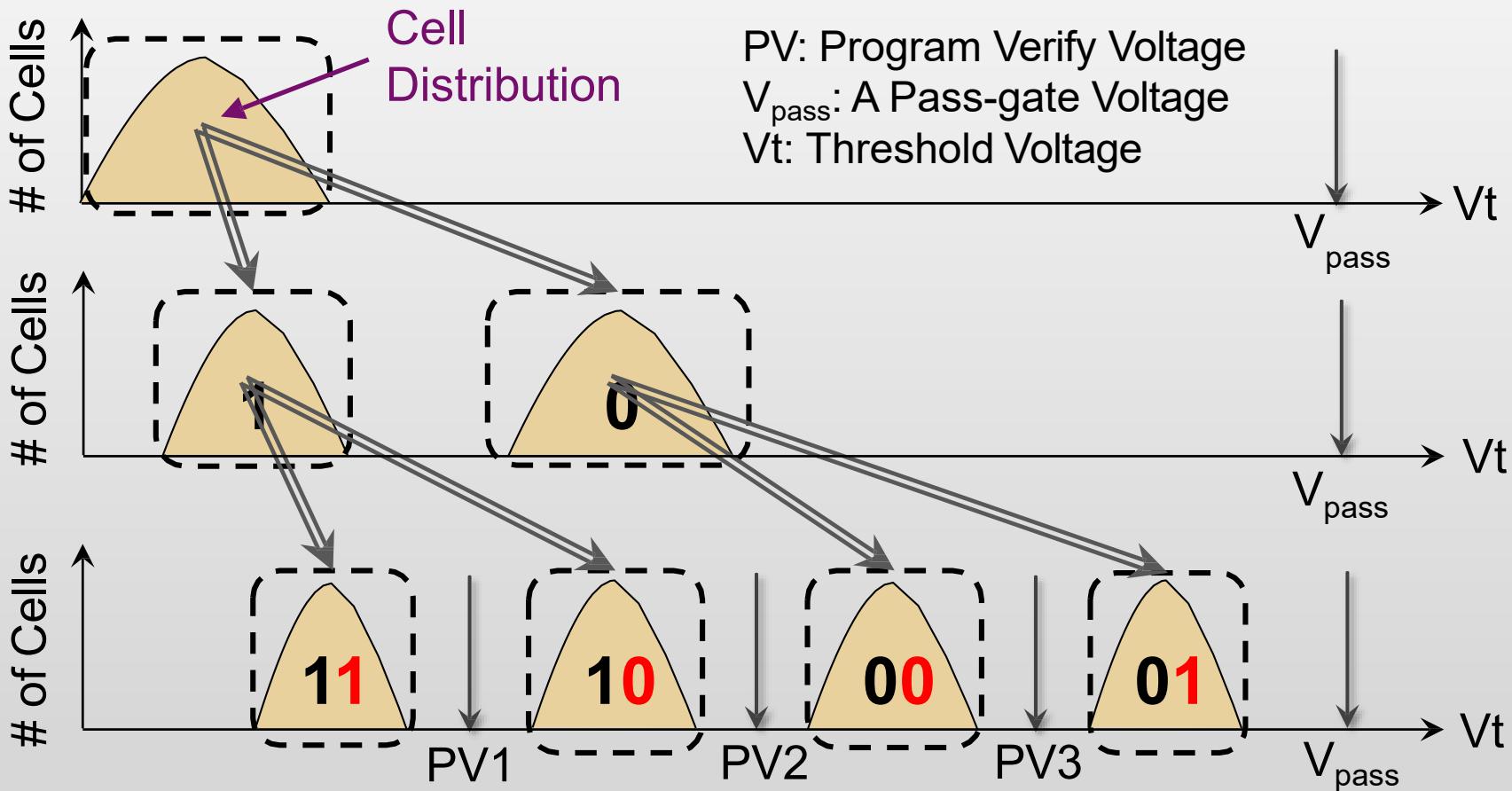


Single-Level Cell & Multi-Level Cell (1/2)

- **Single-Level Cell (SLC):** one bit per cell
 - SLC provides faster read/write speed, lower error rate and longer endurance with higher cost.
- **Multi-Level Cell (MLC):** multiple bits per cell
 - MLC allows each memory cell to store multiple bits of information with degraded performance and reliability.
 - Multi-Level Cell (MLC_{x2}): 2 bits per cell
 - Triple-Level Cell (TLC): 3 bits per cell
 - Quad-Level Cell (QLC): 4 bits per cell



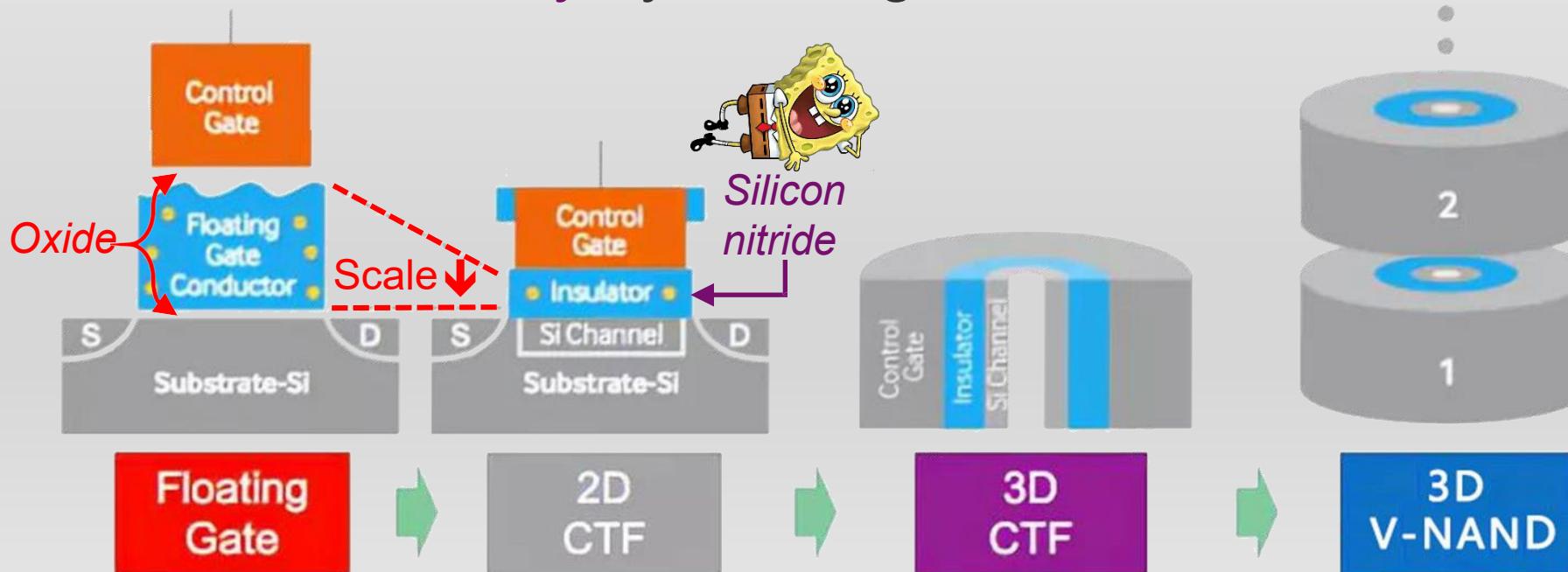
Single-Level Cell & Multi-Level Cell (2/2)



- Low efficiency in programming/verifying low bit(s)
- High bit error rate in low bit(s)

Evolution of NAND Flash

- Scaling down **floating-gate** cell is challenging.
 - The **oxide thickness** must be more than **6-nm**.
- Charge trap flash (CTF)** becomes popular.
 - It uses a **silicon nitride film** to suck electrons.
- 3D flash** further scales down the feature size and increase **areal density** by building tall.

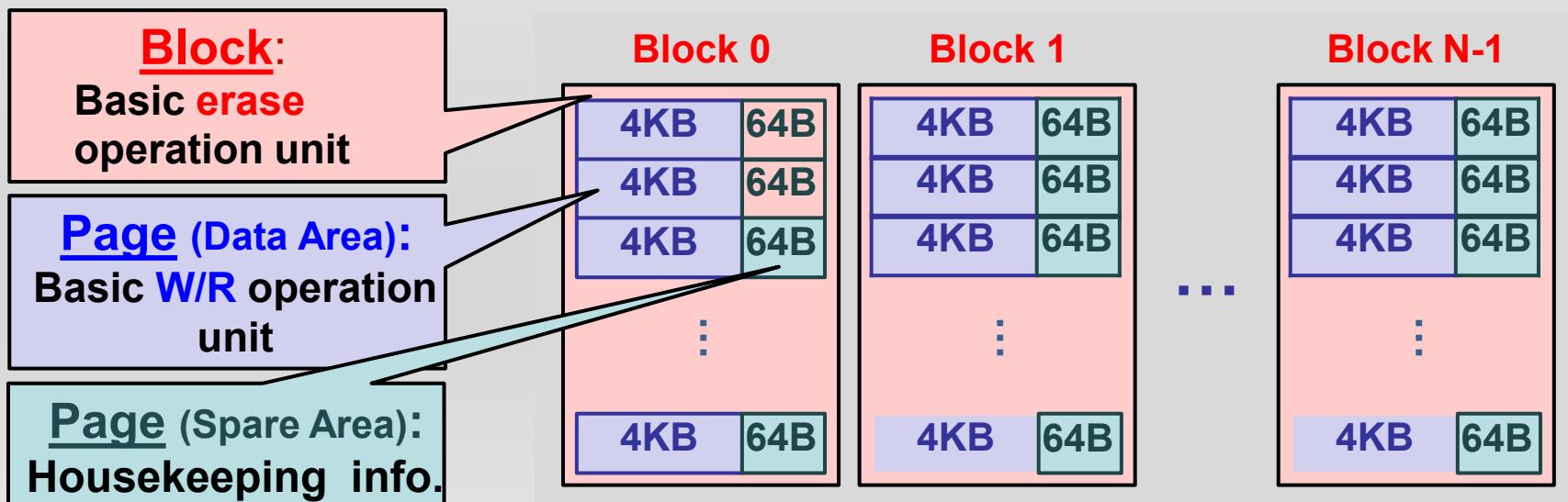


Inherent Challenges

- Common challenges of NAND flash:
 - ① Asymmetric operation units between read/write and erase
 - ② Erase before writing (a.k.a., write-once property)
 - ③ Limited endurance
 - ④ Data errors caused by write and read disturb
 - ⑤ Data retention errors
- Sophisticated management techniques are needed to make flash become better.

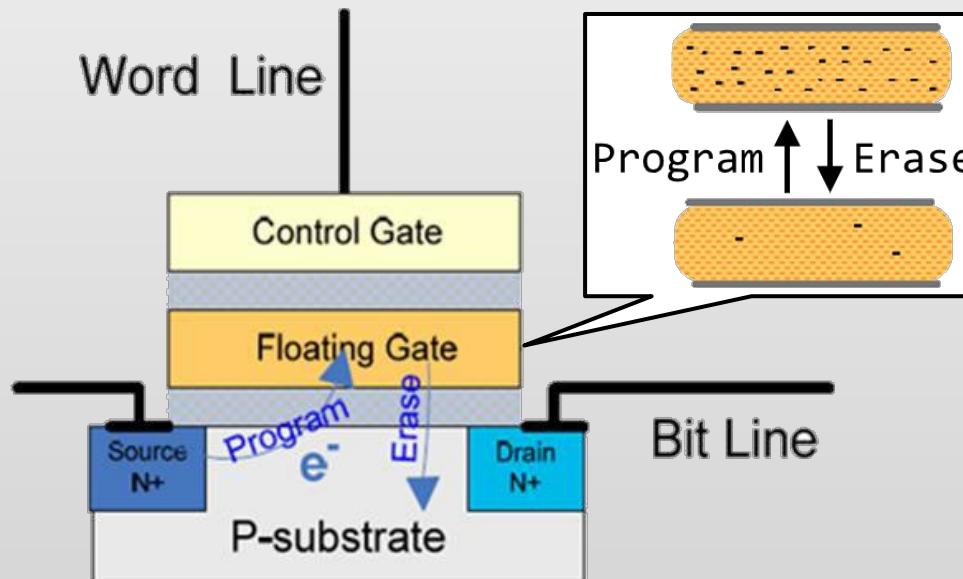
1. Asymmetric Operation Units

- Flash cells are organized into **pages**, and hundreds of pages are grouped into a **block**.
 - A page is further divided into data area and spare area.
 - The spare area keeps redundancy for error correction or metadata.
- **Asymmetric Operation Units:** Flash cells can only be read or written in the unit of **a page**; while all pages of a **block** need to be erased at a time.



2. Erase before Write (Write-once)

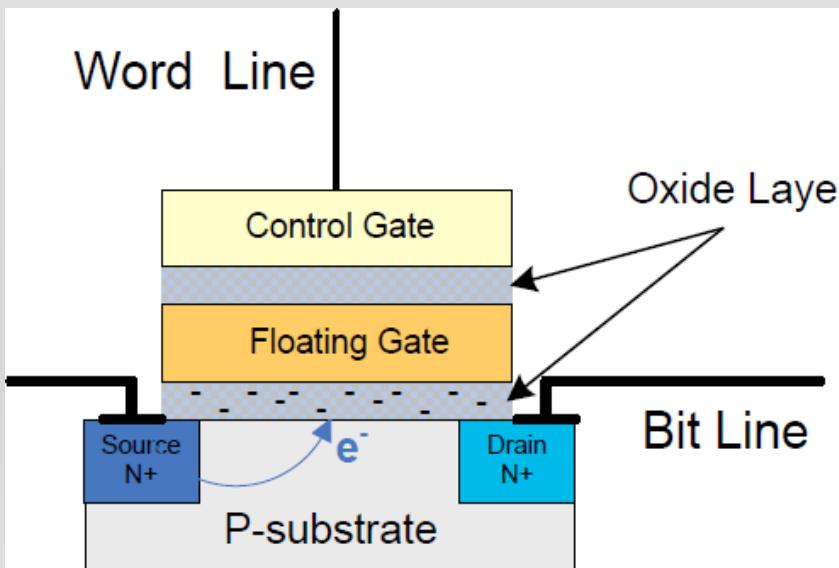
- **Erase before Write:** Once written to “0”, the only way to reset a flash cell to “1” is by **erasing**.
 - The **erase** operation sets all bits in a **block** to “1”’s at a time.



- **Write-once Property:** A flash **page** cannot be overwritten until the residing **block** is erased first.

3. Limited Endurance

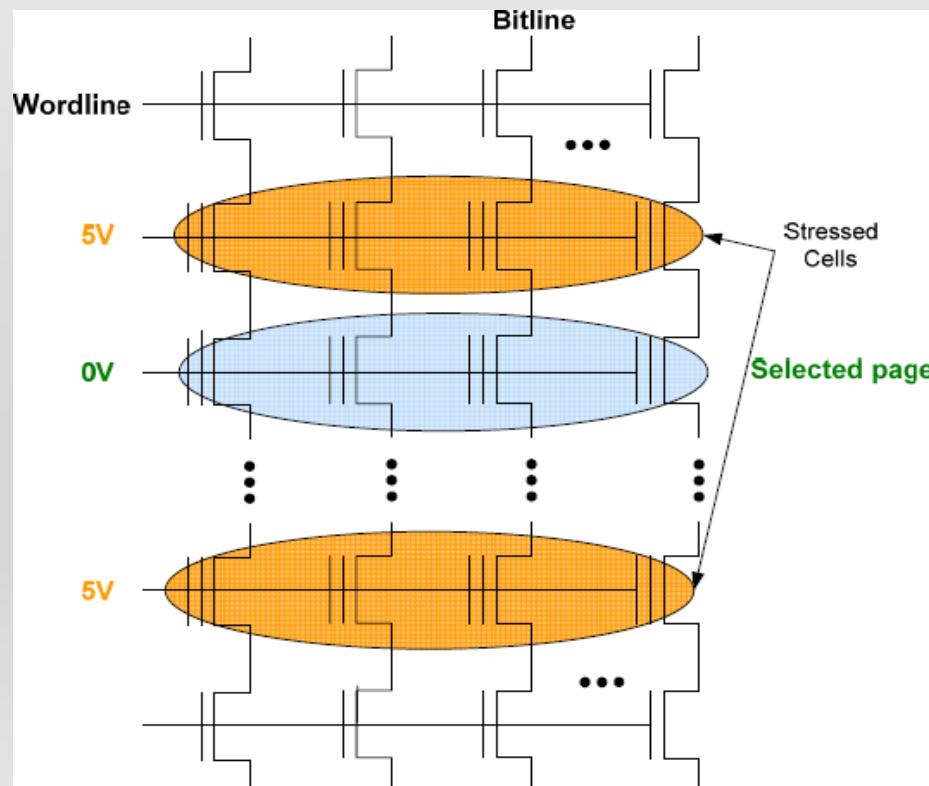
- **Limited Endurance:** A flash block can only endure a **limited number of program/erase (P/E) cycles**.
 - SLC: $60K \sim 100K$ P/E cycles
 - MLC: $1K \sim 10K$ P/E cycles
 - TLC: $< 1K$ P/E cycles
- Reason: The oxide layer may be “**worn out**”.



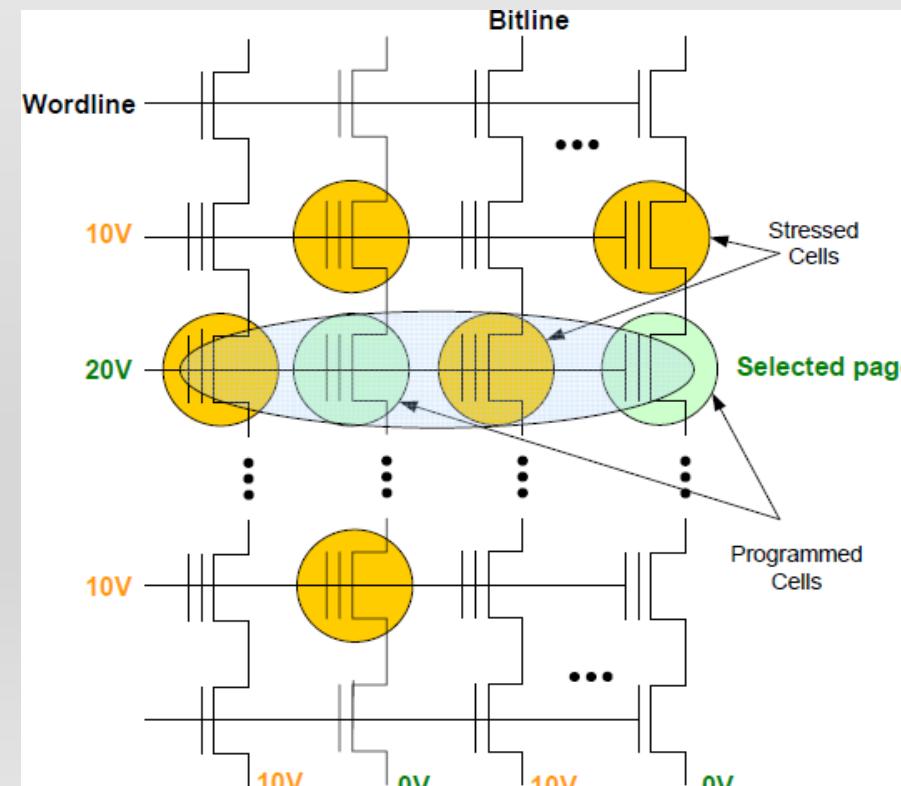
4. Read/Write Disturb

- **Read/Write Disturb:** Reading or writing a page may result in the “**weak programming**” on its neighbors.
 - **Solution to Write Disturb:** Programming pages of a block “in a sequential order” (a.k.a., **sequential write constraint**).

Read Disturb

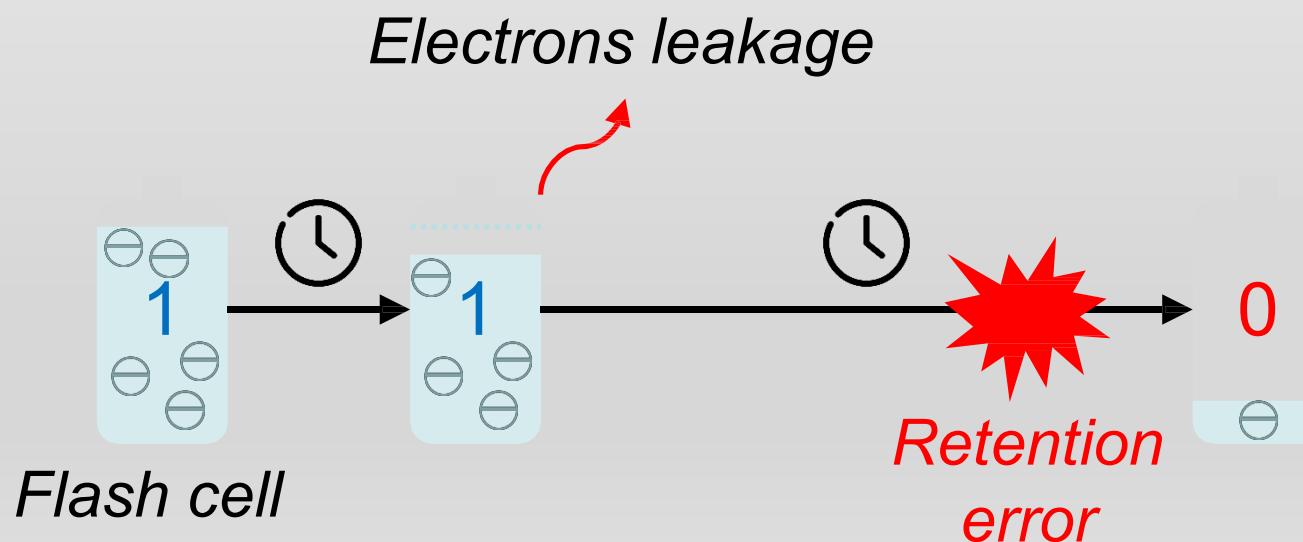


Write Disturb



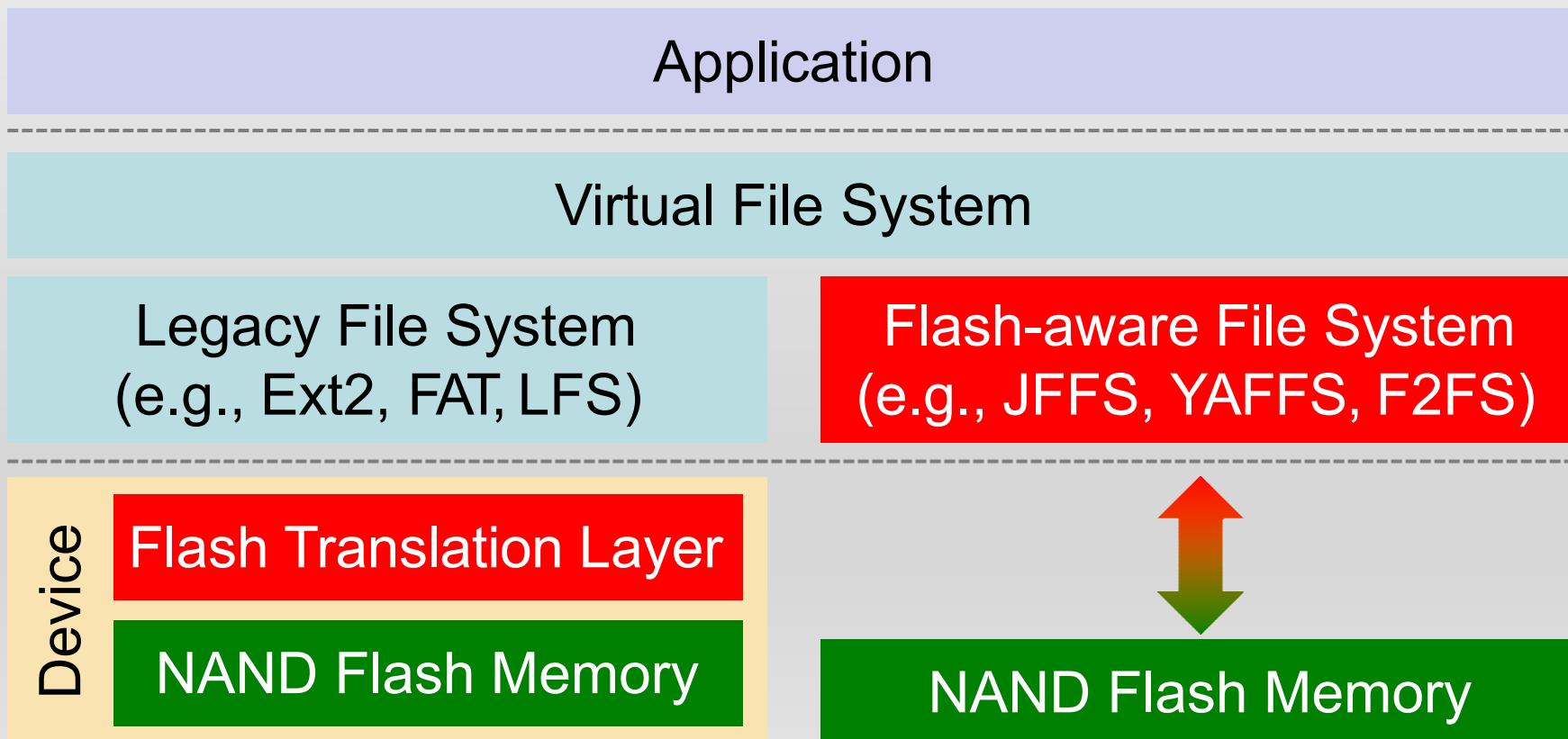
5. Data Retention Errors

- **Data retention time** defines how long the written data remains valid within a flash cell.
 - It is inversely related to the number of P/E cycles.
- Electrons **leak** over time and result in **retention errors**.
 - Solution: “Correct-and-refresh pages” from time to time.
 - This solution can also reset the **read/write disturb**.



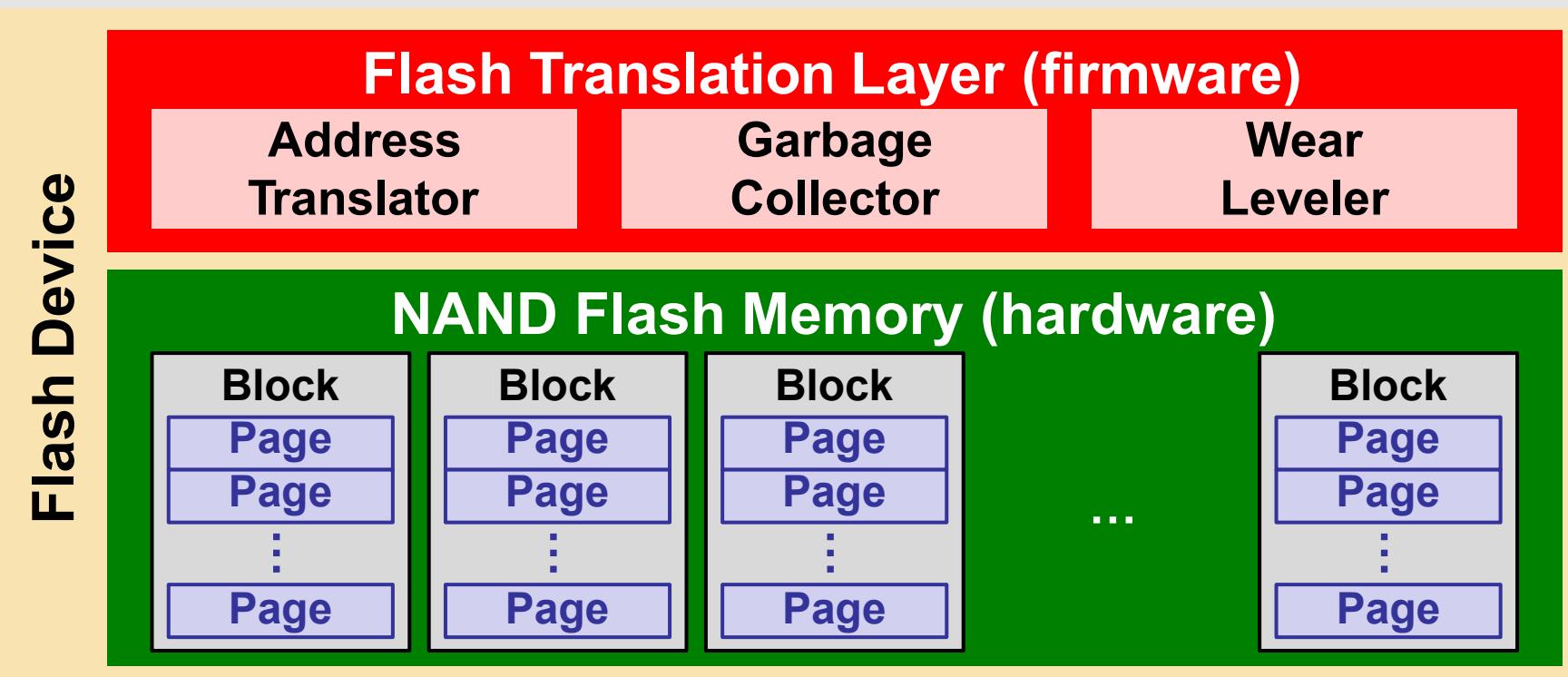
System Architecture

- There are two typical ways to address the inherent challenges of flash memory:
 - Implementing a Flash Translation Layer in the device.
 - Designing a Flash-aware File System in the host.



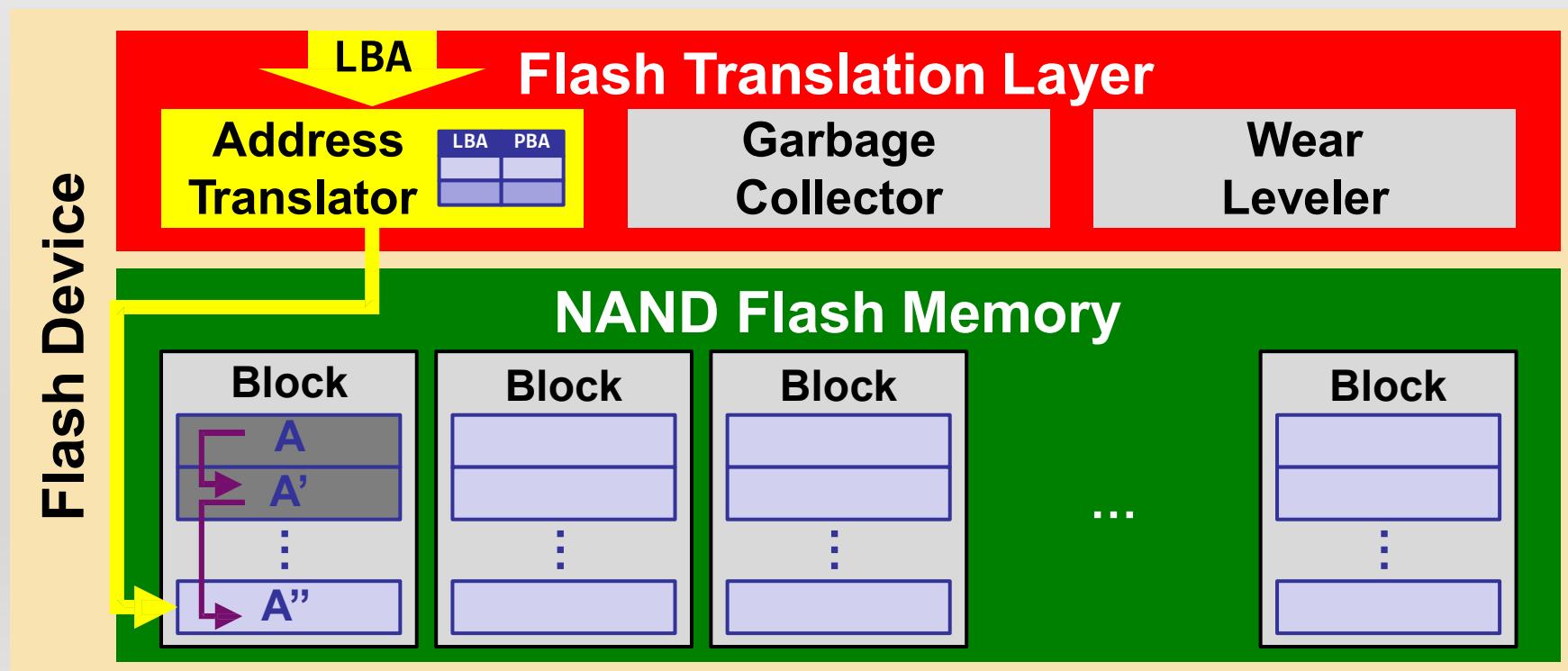
Flash Translation Layer

- **Flash Translation Layer (FTL)** is a firmware inside the flash device to make the NAND flash memory “appear as” a **block device** to the host.
 - It consists of three major components: ① address translator, ② garbage collector, and ③ wear leveler.



1. Address Translator

- Due to the **write-once property**, the **out-place update** is adopted to write the updated data to free pages.
 - Address translator **maps** logical block addresses (LBAs) from the host to physical page addresses (PPAs) in flash.
 - The mapping table is kept in the **memory space** of the flash device.

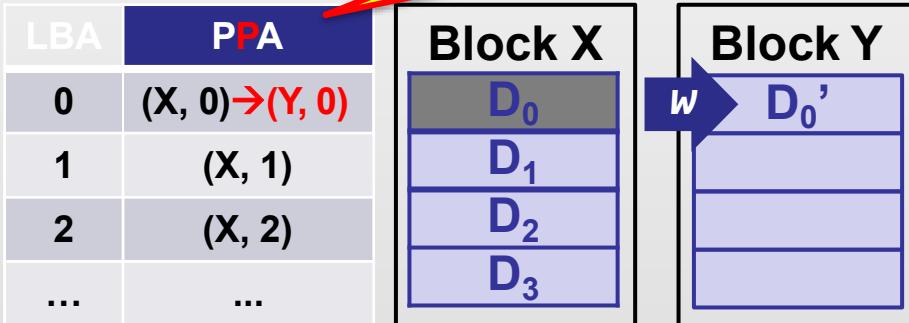


Page-Level, Block-Level, and Hybrid

- Page-Level (PL)

1-1 Page-Level
Mapping Table

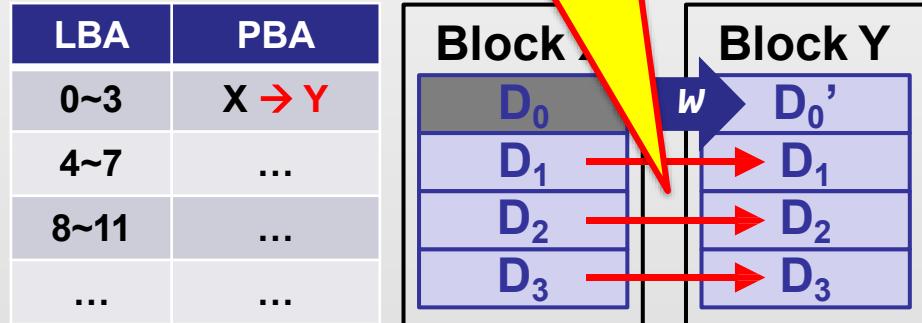
Big in table size!



- Block-level (BL)

1-1 Block-Level
Mapping Table

Slow in write!



- Block-Level (BAST)

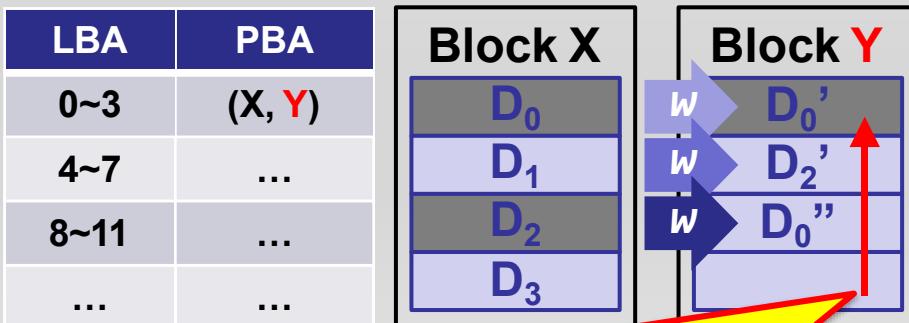
1-2 Block-Level
Mapping Table

(Like BL)

Primary

(Logging)

Secondary



Slow in read (must check spare area)!

- Hybrid (FAST)

Good R/W?

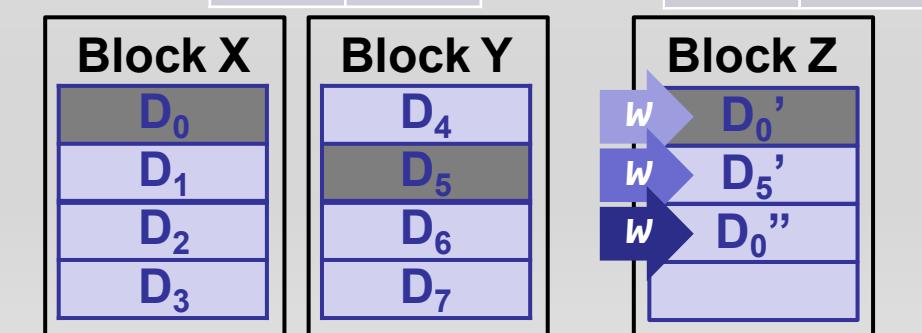
1-1
BL Map

LBA | PBA

+

PL
Map

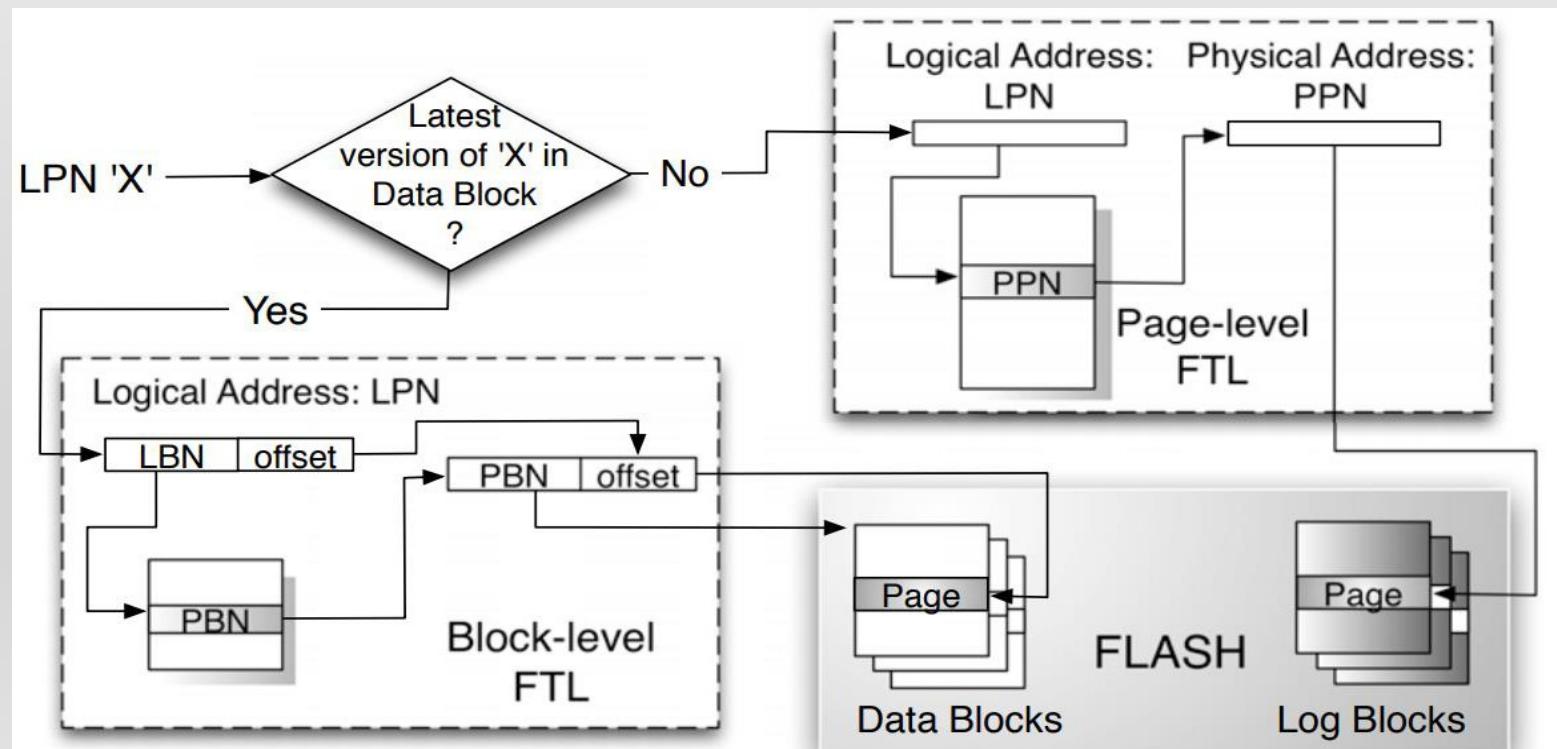
LBA | PPA



Data Block(s) share Log Block(s)

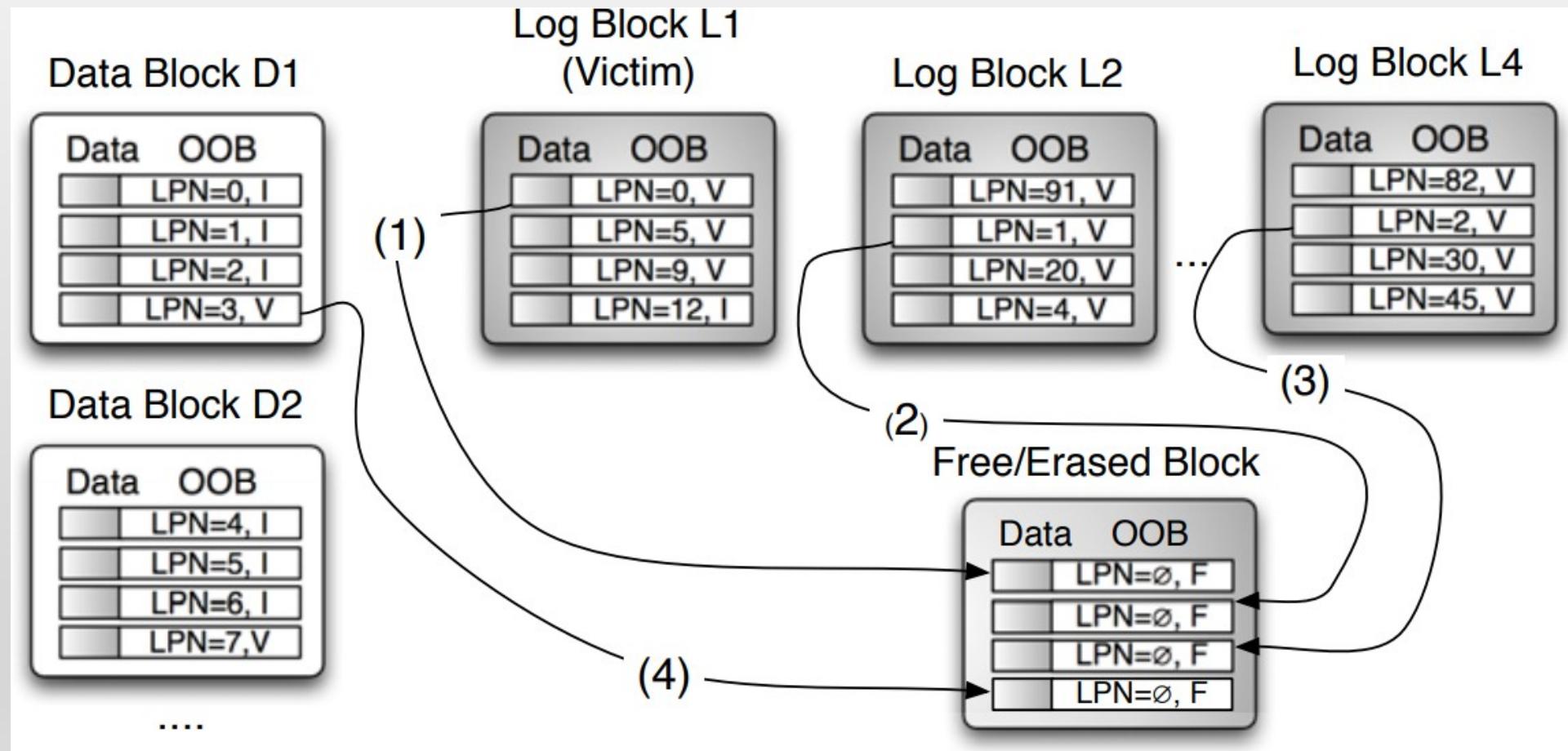
Hybrid FTL

- Hybrid FTLs logically partition blocks into two groups:
 - **Data Blocks** are mapped via the block-level mapping.
 - **Log/Update Blocks** are mapped via a page-level mapping.
 - Any update on data blocks are performed by writes to the log blocks.
 - Few log blocks are **shared** by all data blocks.



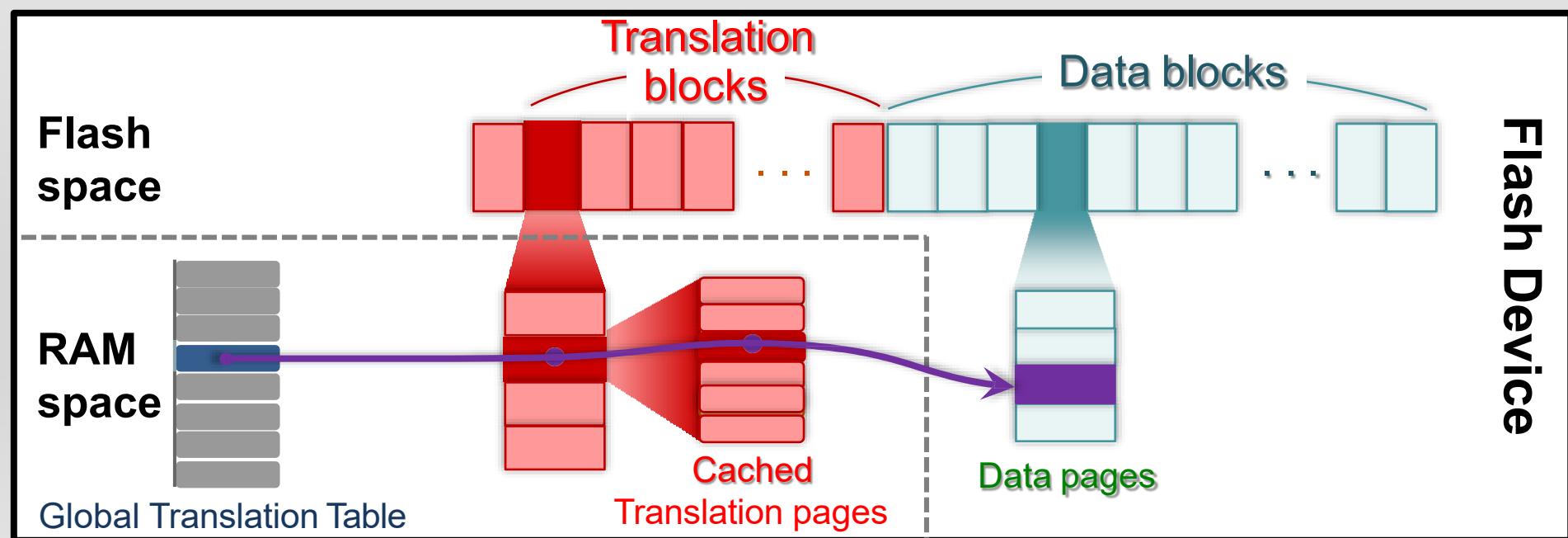
Expensive Merge of Hybrid FTL

- Hybrid FTLs induce **costly garbage collection**.

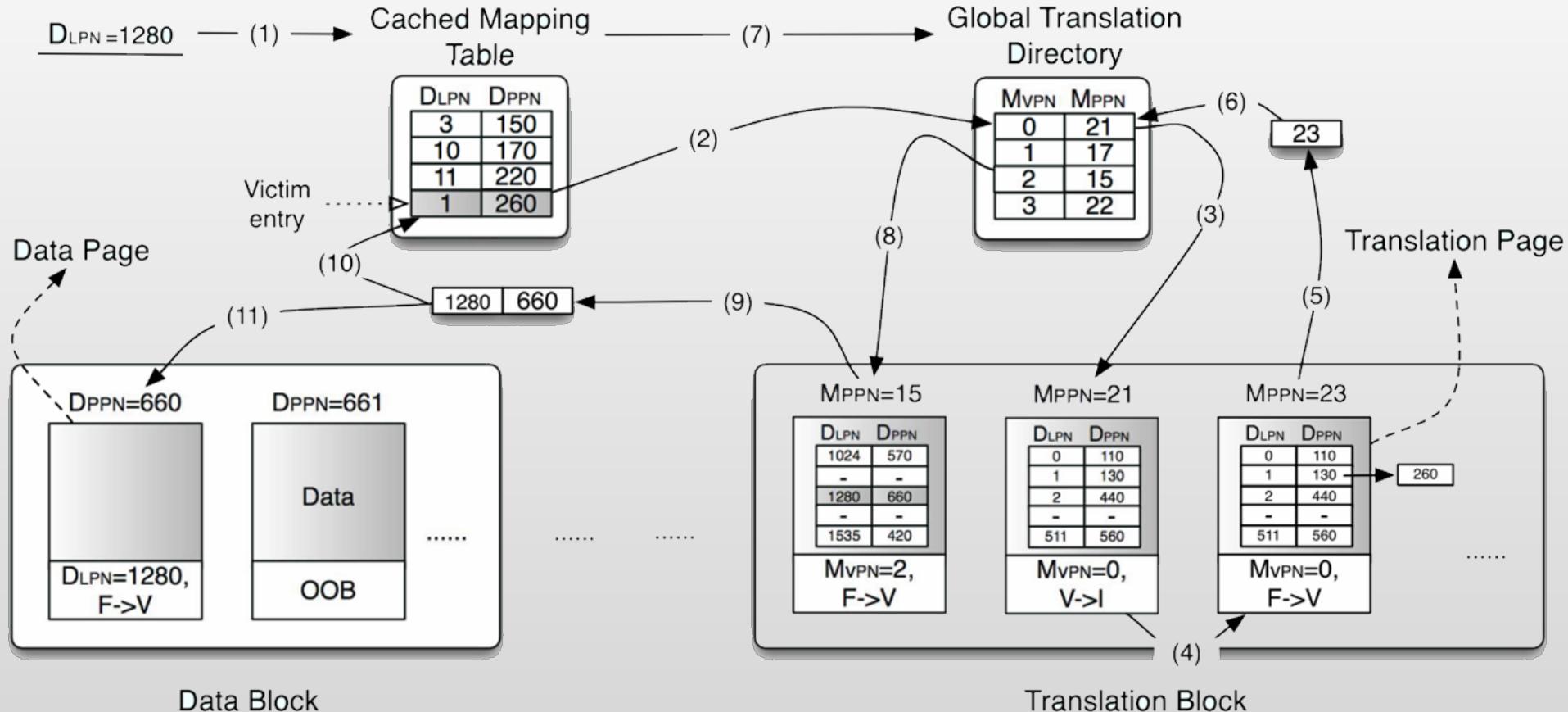


Demand-based Address Translation

- Keeping all mapping tables in RAM is **ineffective**.
- **Page-level** translation for all data with **limited RAM**.
 - Map of **data pages** are stored in **translation pages** on **flash**.
 - **Translation pages** are cached in **RAM** on demand.
 - Map of **translation pages** (i.e., global translation table) are kept in **RAM** for efficient lookup.



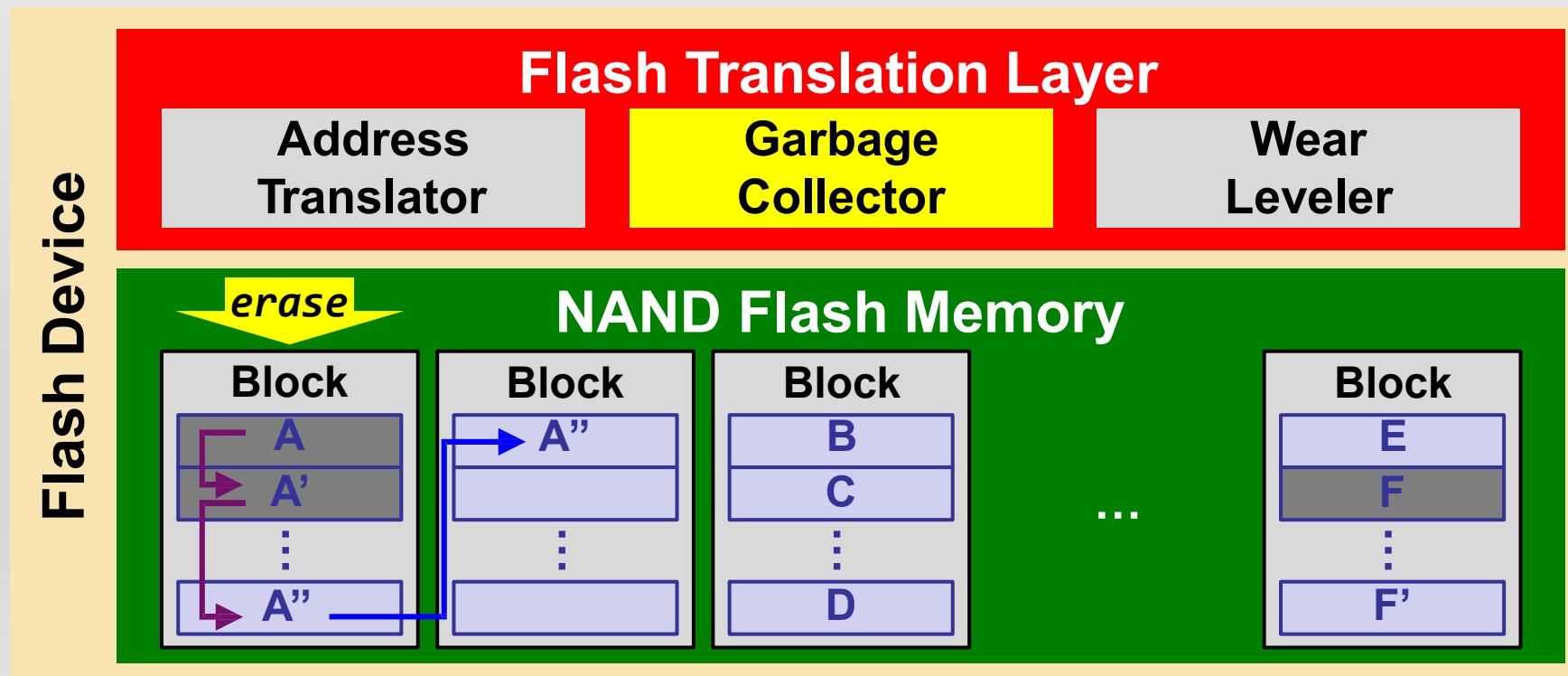
DFTL: An Working Example



(1) Request to DLP N 1280 incurs a **miss** in Cached Mapping Table (CMT), **(2)** **Victim entry** DLP N 1 is selected, its corresponding translation page MPPN 21 is located using Global Translation Directory (GTD), **(3)-(4)** MPPN 21 is **read**, **updated** (DPPN 130 → DPPN 260) and **written** to a free translation page (MPPN 23), **(5)-(6)** GTD is **updated** (MPPN 21 → MPPN 23) and DLP N 1 entry is erased from CMT. **(7)-(11)** The original request's (DLP N 1280) **translation page** is located on flash (MPPN 15). The mapping entry is **loaded** into CMT and the request is **serviced**. Note that each GTD entry maps 512 logically consecutive mappings.

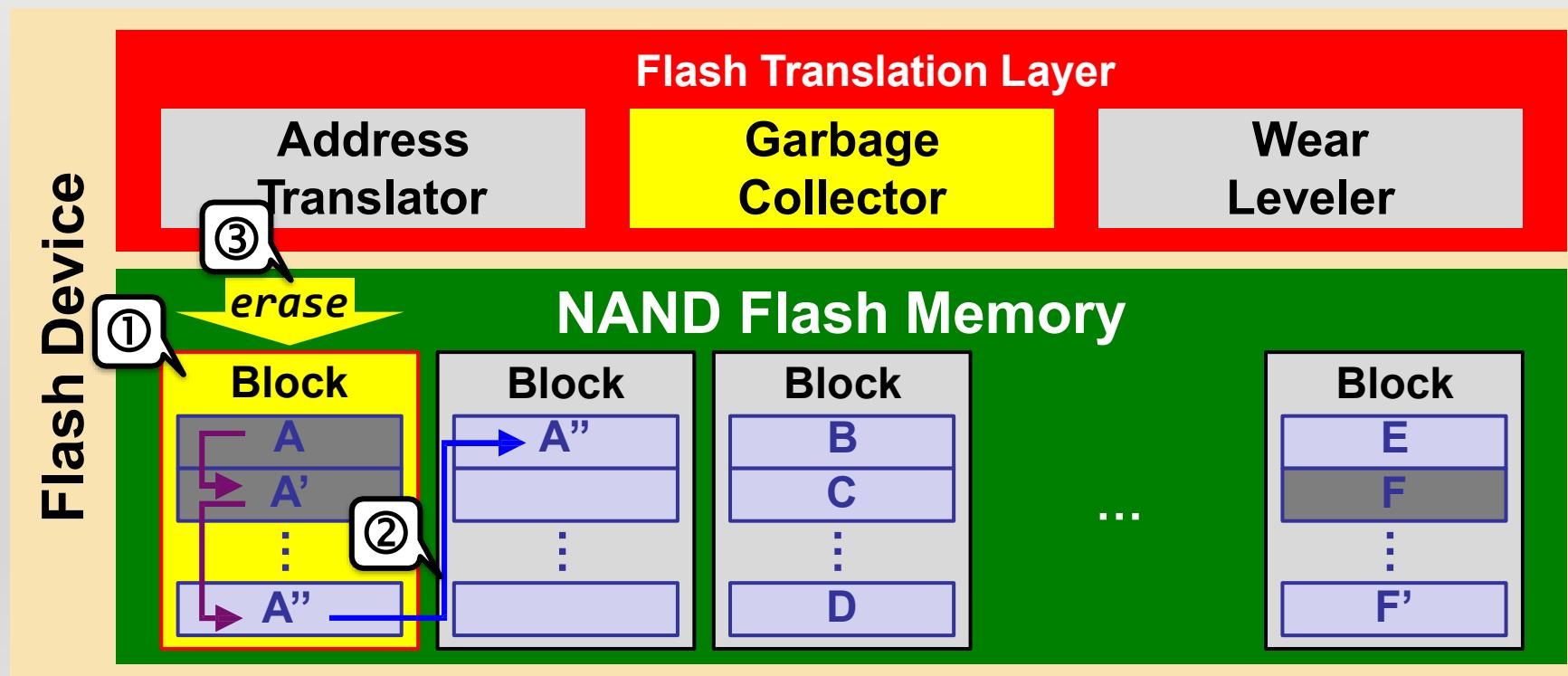
2. Garbage Collector

- Since the **out-place update** leaves **multiple versions** of data, the **garbage collector** is to reclaim pages occupied by stale data by **erasing** its residing blocks.
 - The **live-page copying** is needed to migrate pages of the latest versions of data before the erase operation.



Typical GC Procedure

1. **Victim Block Selection:** Pick one (or more) block that is “most worthy” to be reclaimed
2. **Live-Page Copying:** Migrate all live pages out
3. **Victim Block Erasing:** Reclaim the space occupied by dead pages



Typical Victim Block Selection Policies

- **Random** selects the victim block in a uniformly-random manner to yield a **long-term average use**.
- **FIFO** (or **RR**) cleans blocks in a round-robin manner for **minimized P/E cycle difference**.
- **Greedy** cleans the block with the largest number of dead pages for **minimized live-page copying**.
- **Cost-Benefit (CB)** (and its variants) cleans the block with the largest benefit-cost for **wear leveling**:

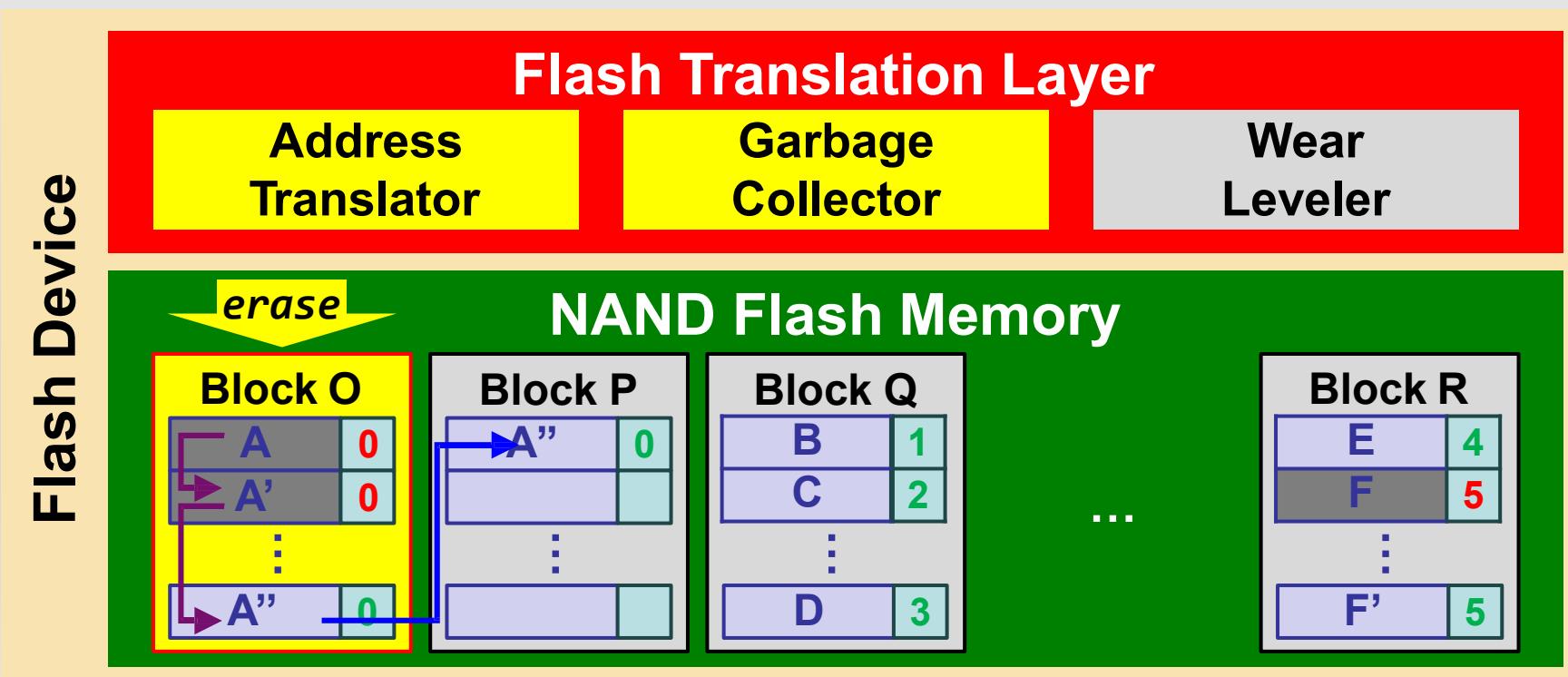
$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{age} \times (1 - u)}{2u}$$

- **age**: invalidated time period = current time – the time when a page of the block was lastly invalidated
- **u**: percentage of live pages of the block

Page Liveness

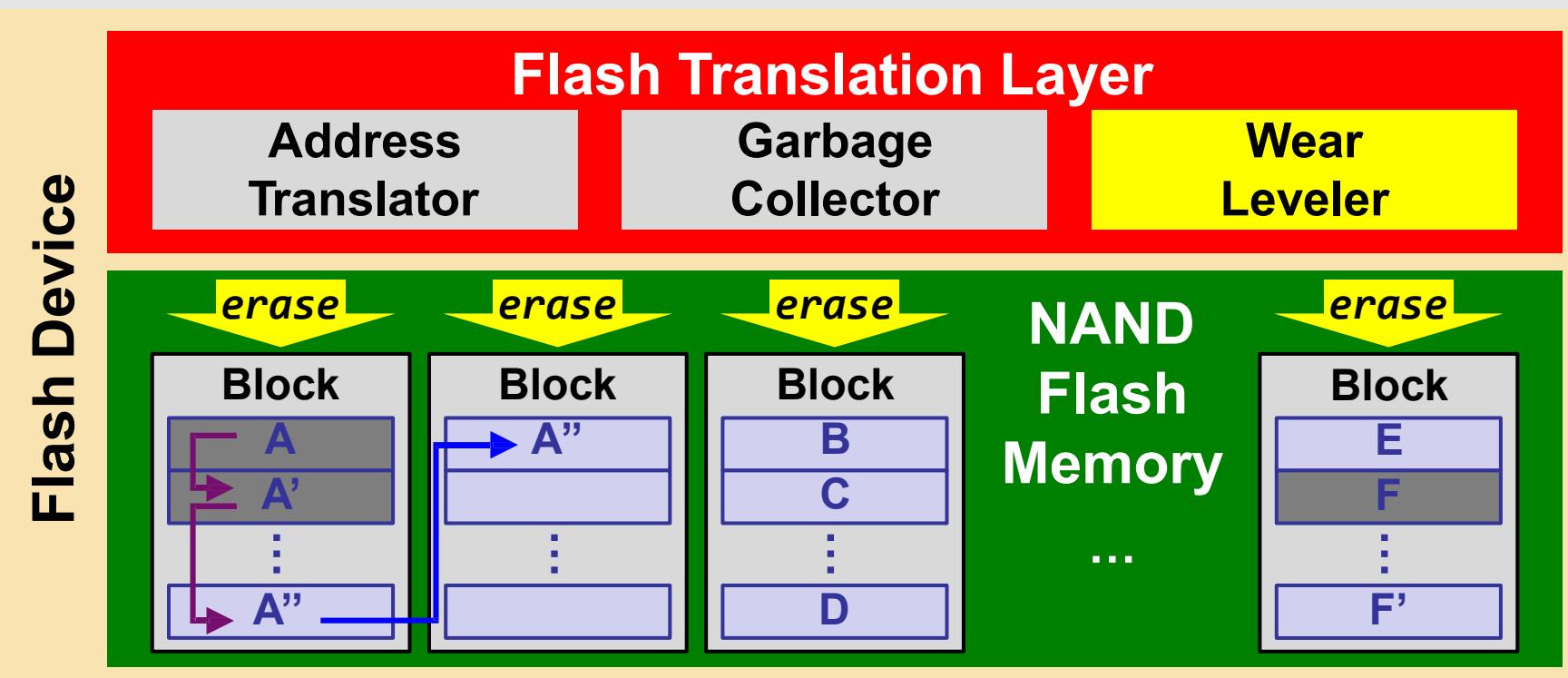
- Keep the **LBA** into the **spare area**.
- Check both the spare area of pages and the mapping table during GC.
 - Live: LBA **matches** the mapping.
 - Dead: LBA **mismatches** the mapping.

LBA	PPA
0	(O, 3)
1	(Q, 0)
2	(Q, 1)
3	(Q, 3)
...	...



3. Wear Leveler

- Since each block can only endure a **limited number of P/E cycles**, the **wear leveler** is to prolong the overall lifetime by **evenly distributing erases**.
 - The main objective is to prevent any flash block from being worn out “prematurely” than others.

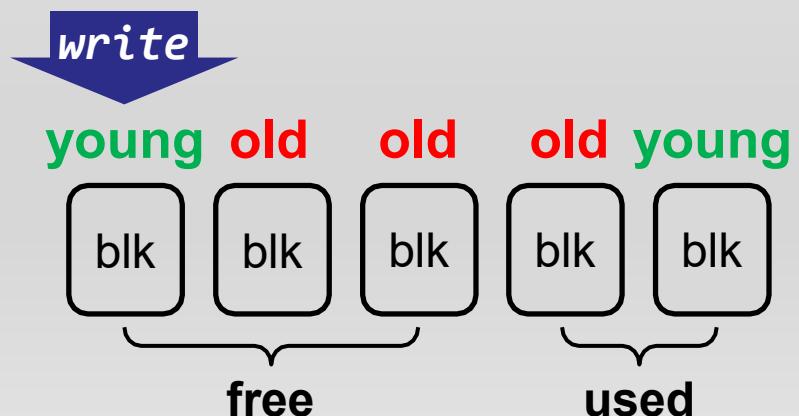


Dynamic vs. Static Wear Leveling

- Wear leveling is classified into **static** or **dynamic** based on the type of blocks involved in WL:

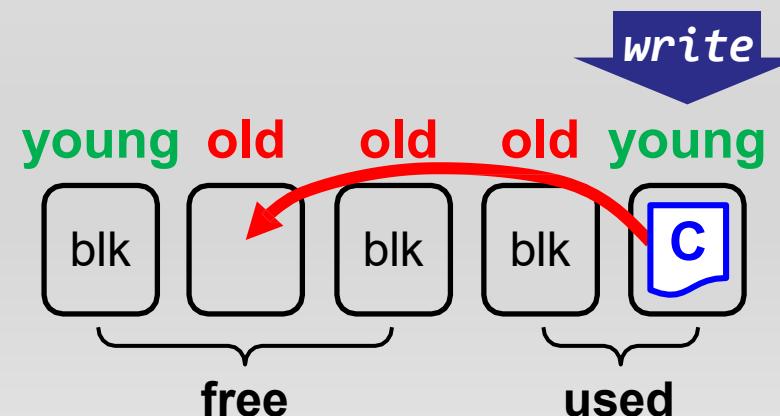
– Dynamic Wear Leveling

- Let **free blocks** have closer erase counts
- How? Simply use free block with lower erase count (i.e., **young** block) to service writes



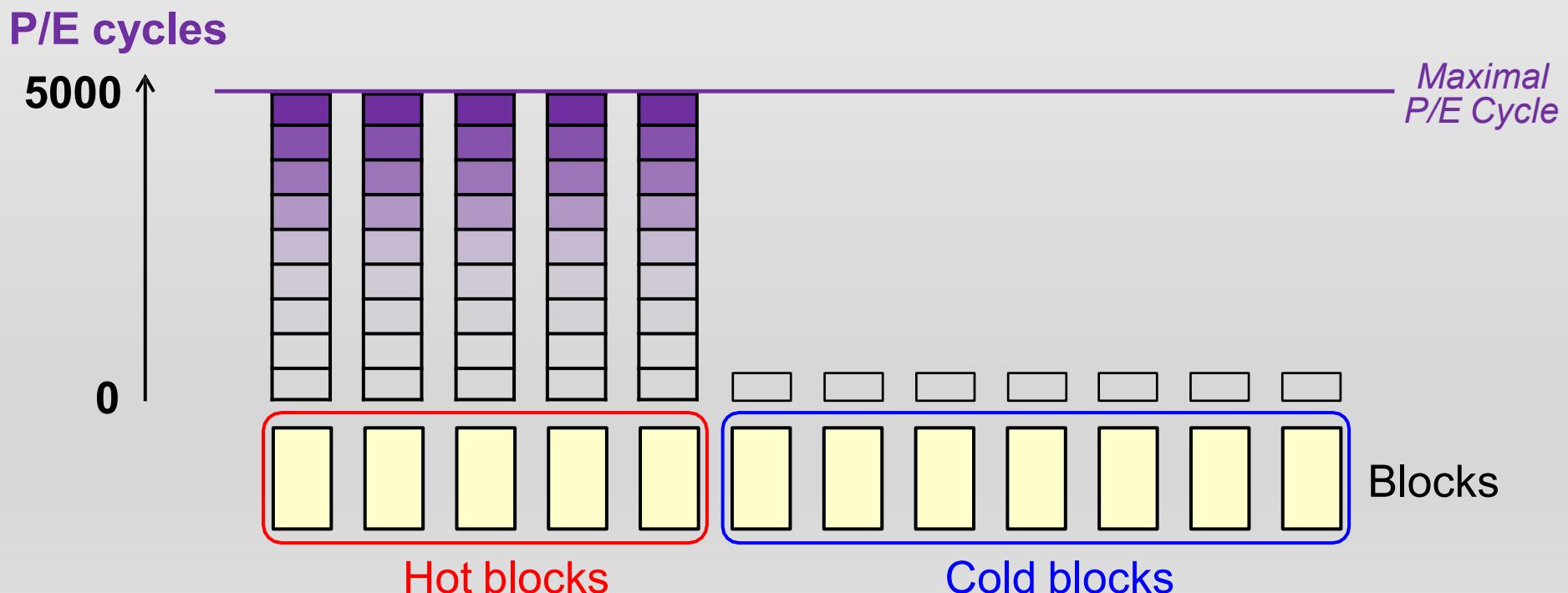
– Static Wear Leveling

- Let **all blocks (free + used)** have closer erase counts
- ① Actively move cold data in **young** block to **elder** block
- ② Then use this (previously-used) **young** block to service writes



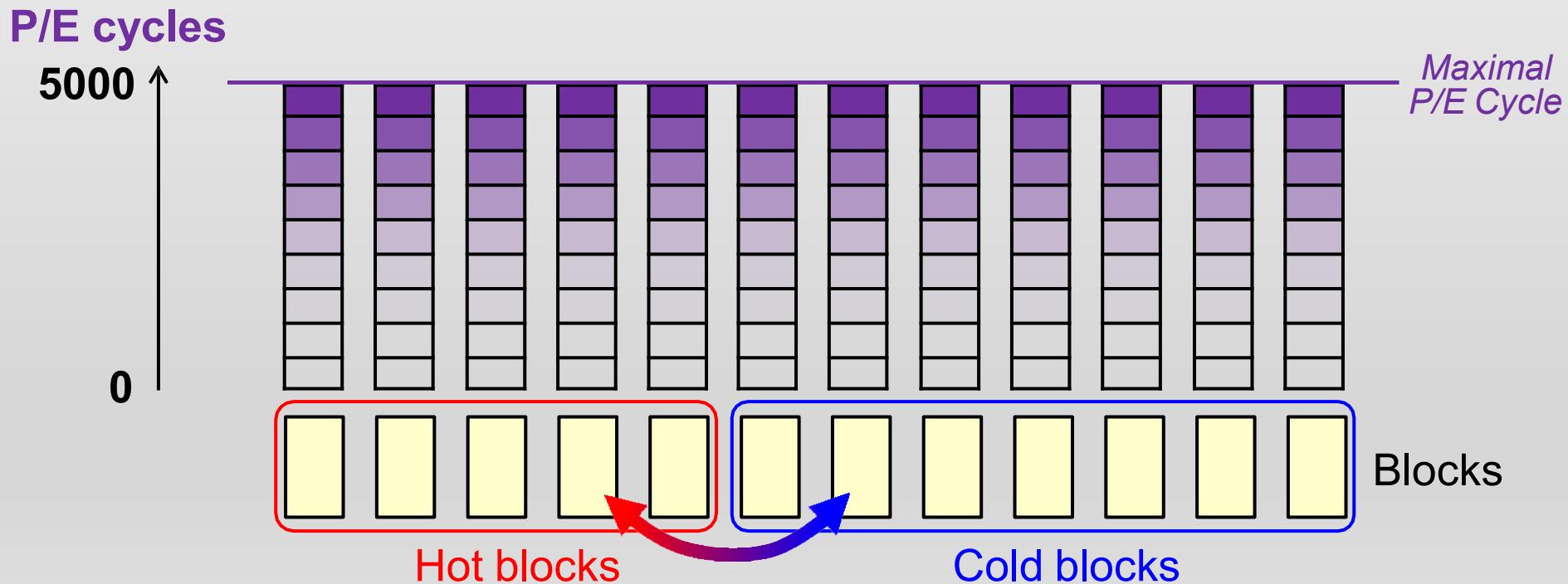
Dynamic Wear Leveling (DWL)

- DWL achieves wear leveling **only for hot blocks**.
 - **Hot Block**: a block mainly containing **hot** data
 - **Cold Block**: a block mainly containing **cold** data
- Hot blocks will be **worn out earlier** than cold blocks.



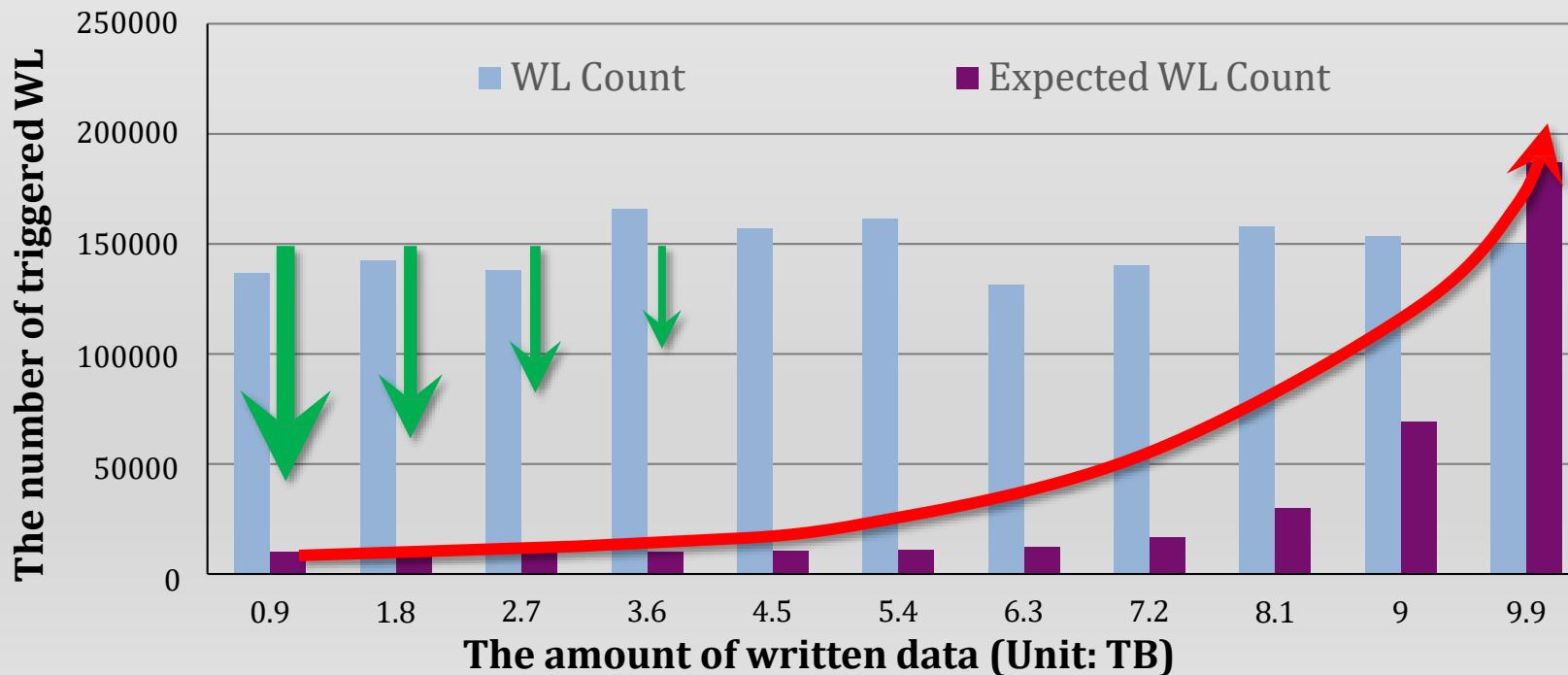
Static Wear Leveling (SWL)

- SWL achieves wear leveling **for all blocks** by **pro- actively moving** cold data to young blocks.
 - Extra data migrations are introduced.
 - Performance is traded for lifetime/endurance.



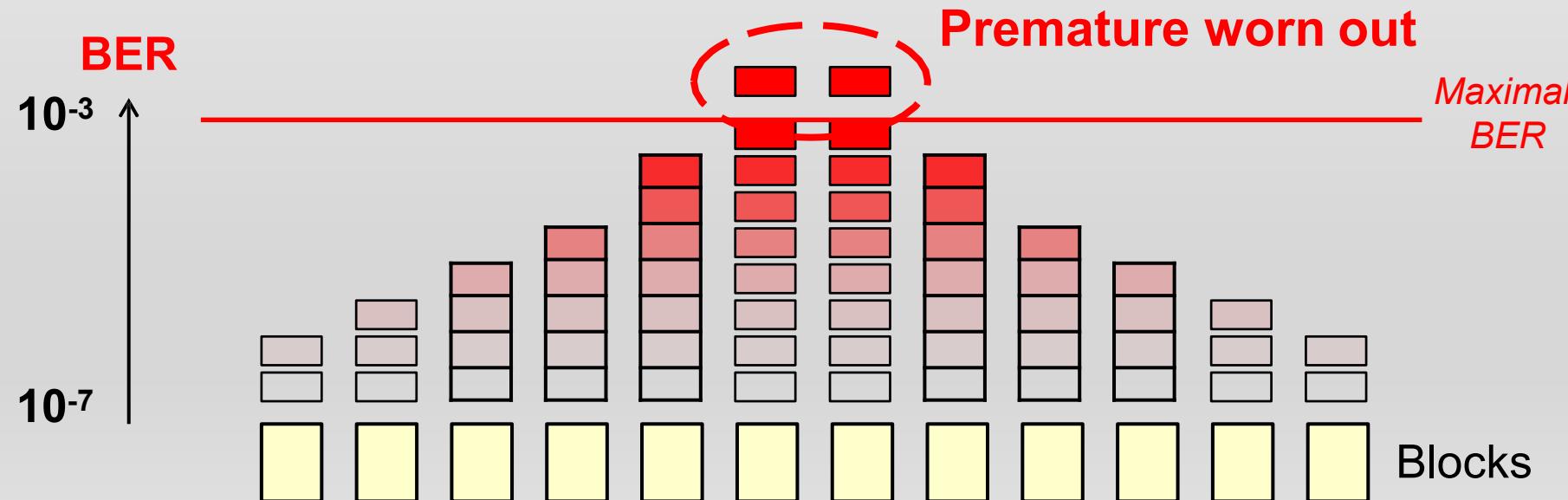
Progressive Wear Leveling (PWL)

- When, and how often WL should be performed?
- WL should be performed in a **progressive way**:
 - Prevent WL in the early stages for **better performance**.
 - **Progressively trigger WL to prolong lifetime.**
 - More and more WLs are performed **over time**.



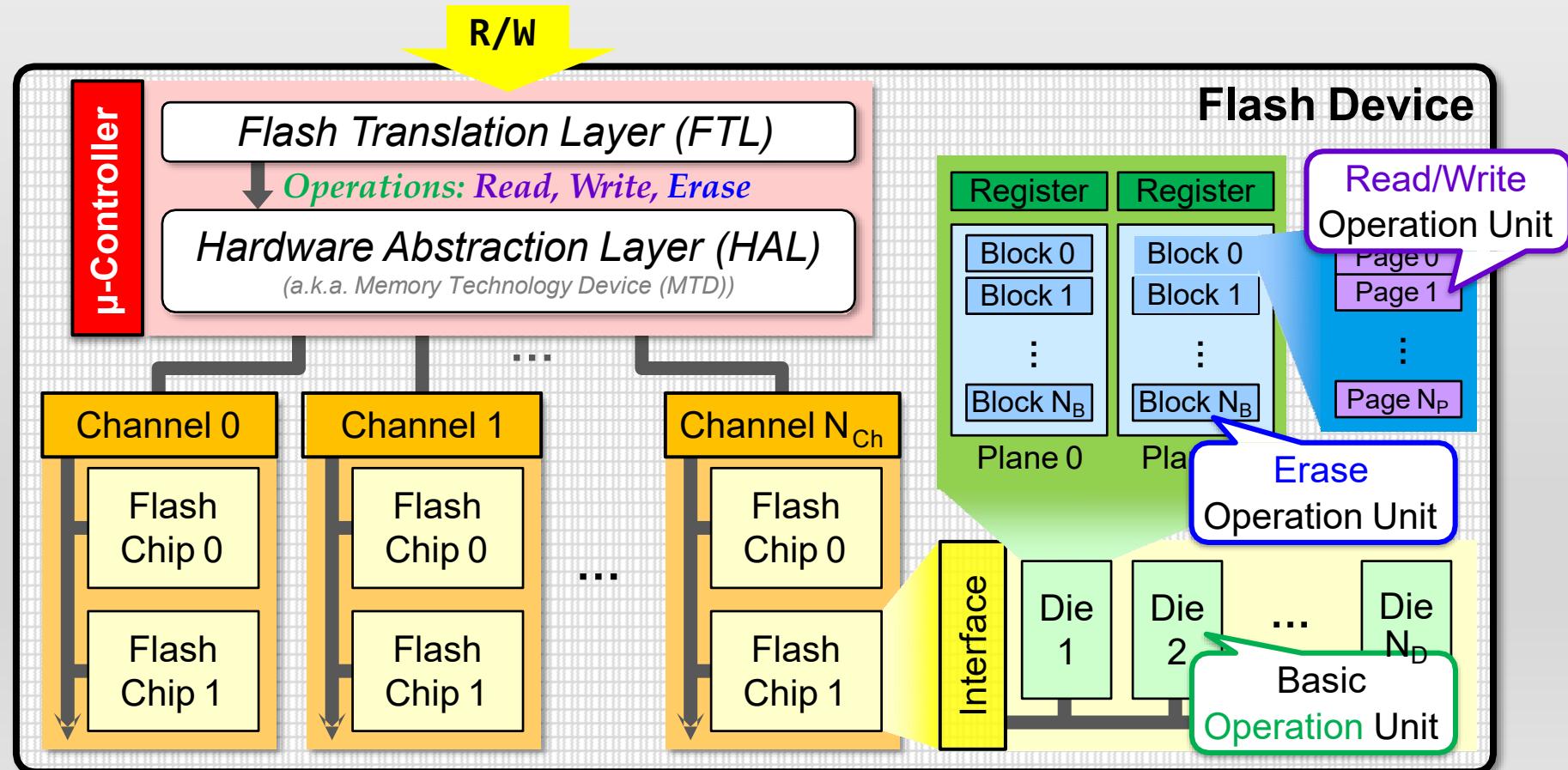
Error-Rate-Aware Wear Leveling

- Key Observations: Flash is not perfect!
 - P/E cycles might **inaccurately** reflect the reliability of flash.
 - Blocks might have different **bit-error rates (BERs)** when enduring the same P/E cycles due to **process variation**.
- Idea: **BER** could be a **better metric** to WL designs.
 - The error correction hardware can report BER to FTL.



Multilevel I/O Parallelism (1/2)

- The internal of flash devices is **highly hierarchical**:
 - Channel → Chip → Die → Plane → Block → Page**
- Multiple I/O operations can be performed **concurrently**.

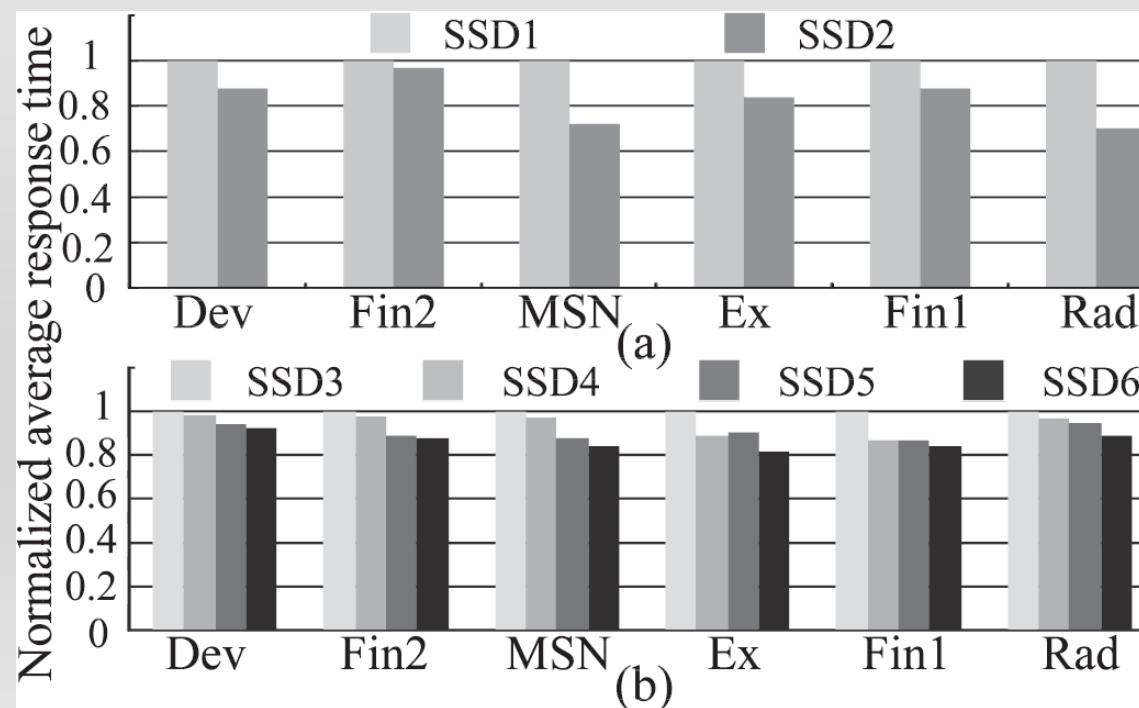


Multilevel I/O Parallelism (2/2)

- The optimal priority order of parallelism should be:

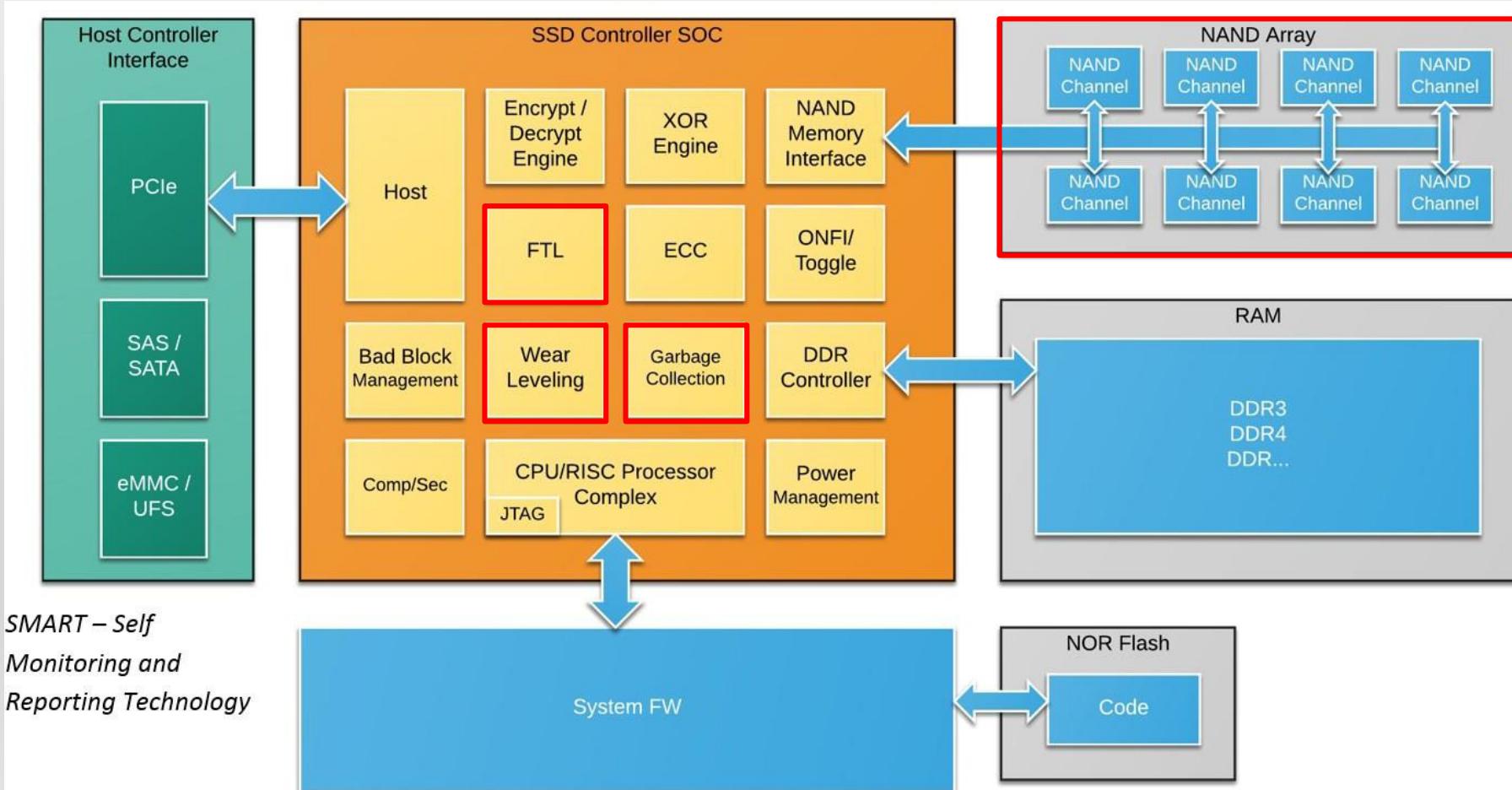
- ① Channel-level
- ② Die-level
- ③ Plane-level
- ④ Chip-level

SSD	Cl.-Cp.-D.-P.	A	Page	Priority order
SSD1	8-4-2-2	Yes	2KB	chip>die>plane>channel
SSD2	8-4-2-2	Yes	2KB	channel>chip>die>plane
SSD3	1-4-2-2	Yes	2KB	channel>chip>die>plane
SSD4	1-4-2-2	Yes	2KB	channel>die>chip>plane
SSD5	1-4-2-2	Yes	2KB	channel>plane>die>chip
SSD6	1-4-2-2	Yes	2KB	channel>die>plane>chip



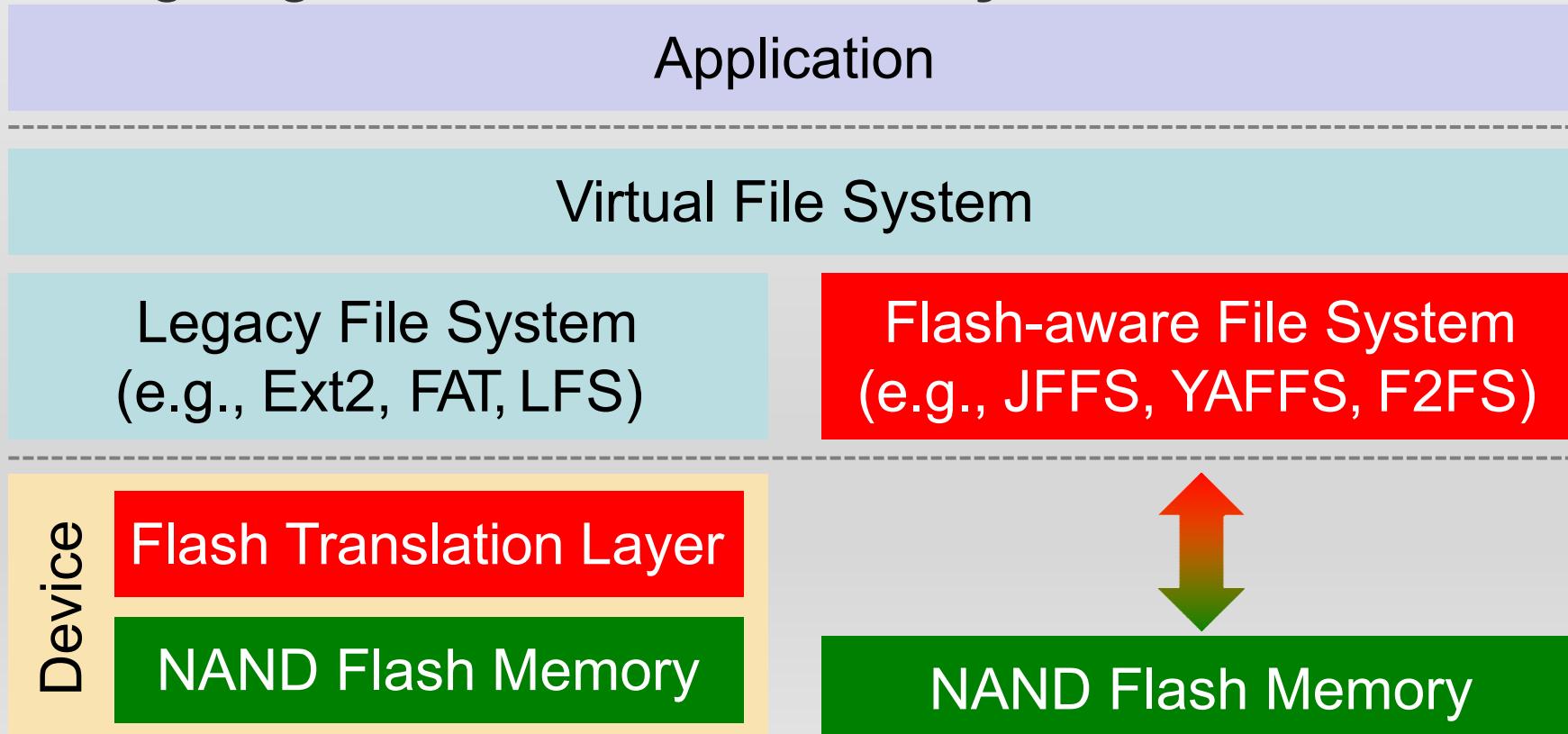
Flash Management is Complex

- The controller of flash memory device is **complex**.
 - It must perform a **myriad of tasks** to receive, monitor and deliver data **efficiently and reliably**.



Recall: System Architecture

- There are two typical ways to address the inherent challenges of flash memory:
 - ① Implementing a **Flash Translation Layer** in the device.
 - ② Designing a **Flash-aware File System** in the host.

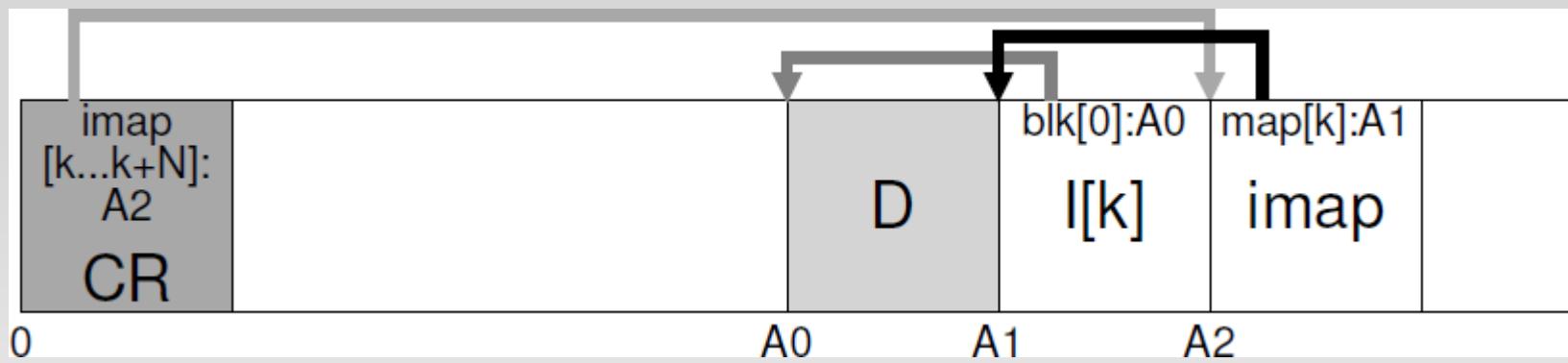


Flash-aware File System

- Random writes are **bad** to flash devices.
 - Free space fragmentation
 - Degraded performance (due to GC)
 - Reduced lifetime (due to GC)
- Writes must be reshaped into **sequential writes**.
 - Same as Log-structured file system (LFS) for HDD!
- Most flash-aware file systems are derived from LFS:
 - Journaling Flash File System (JFFS)
 - Yet Another Flash File System (YAFFS)
 - **Flash-Friendly File System (F2FS)**
 - Publicly available, included in Linux mainline kernel since Linux 3.8.

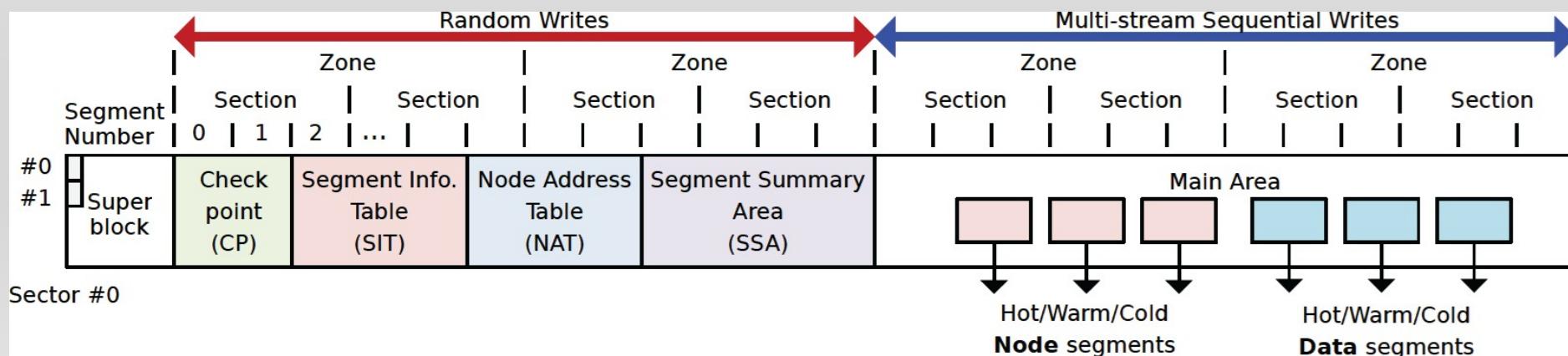
Log-structured File System

- LFS first buffers all writes in an **in-memory segment**
 - and commits the segment to disk **sequentially**.
 - The **Inode Map (imap)**
 - Maps from an inode-number to the disk-address of the most recent version of the inode (i.e., one more mapping!).
 - Updated whenever an inode is written to disk.
 - Placed right next to where data block (D) and inode ($I[k]$) reside.
 - The **Checkpoint Region (CR)**:
 - Records disk pointers to all latest pieces of **imap**.
 - Flushed to disk periodically (e.g., every 30 seconds).



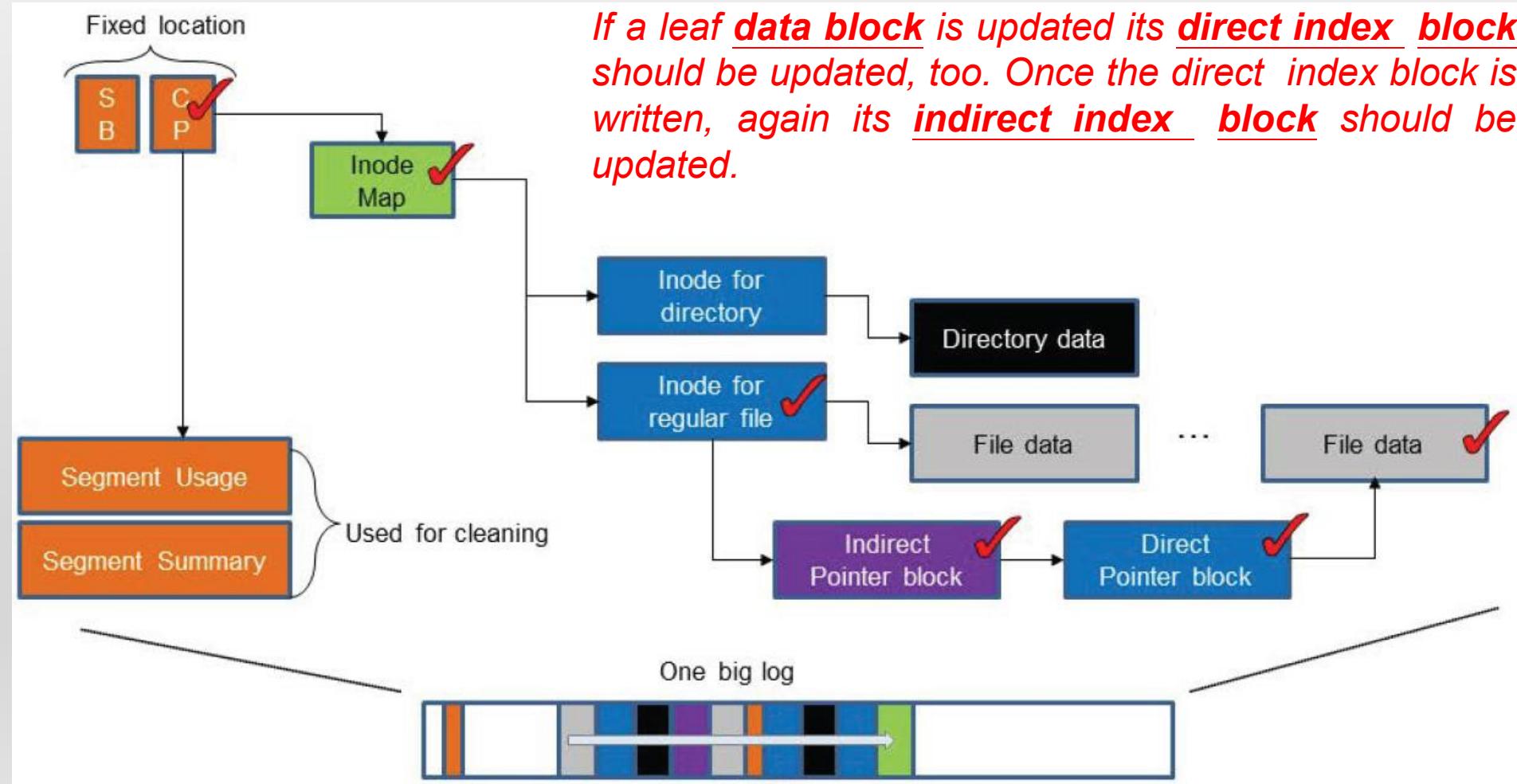
Flash-friendly On-Disk Layout

- Key: There is **no re-position delay** in flash memory!
- Flash Awareness
 - All the FS metadata are **located together** for locality.
 - Start address of main area is **aligned** to the **zone size**.
 - **block=4KB; segment=2MB; section=n segments; zone=m sections.**
 - File system cleaning (i.e., GC) is done in a unit of **section**.
- Cleaning Cost Reduction
 - **Multi-head logging** for hot/cold data separation.



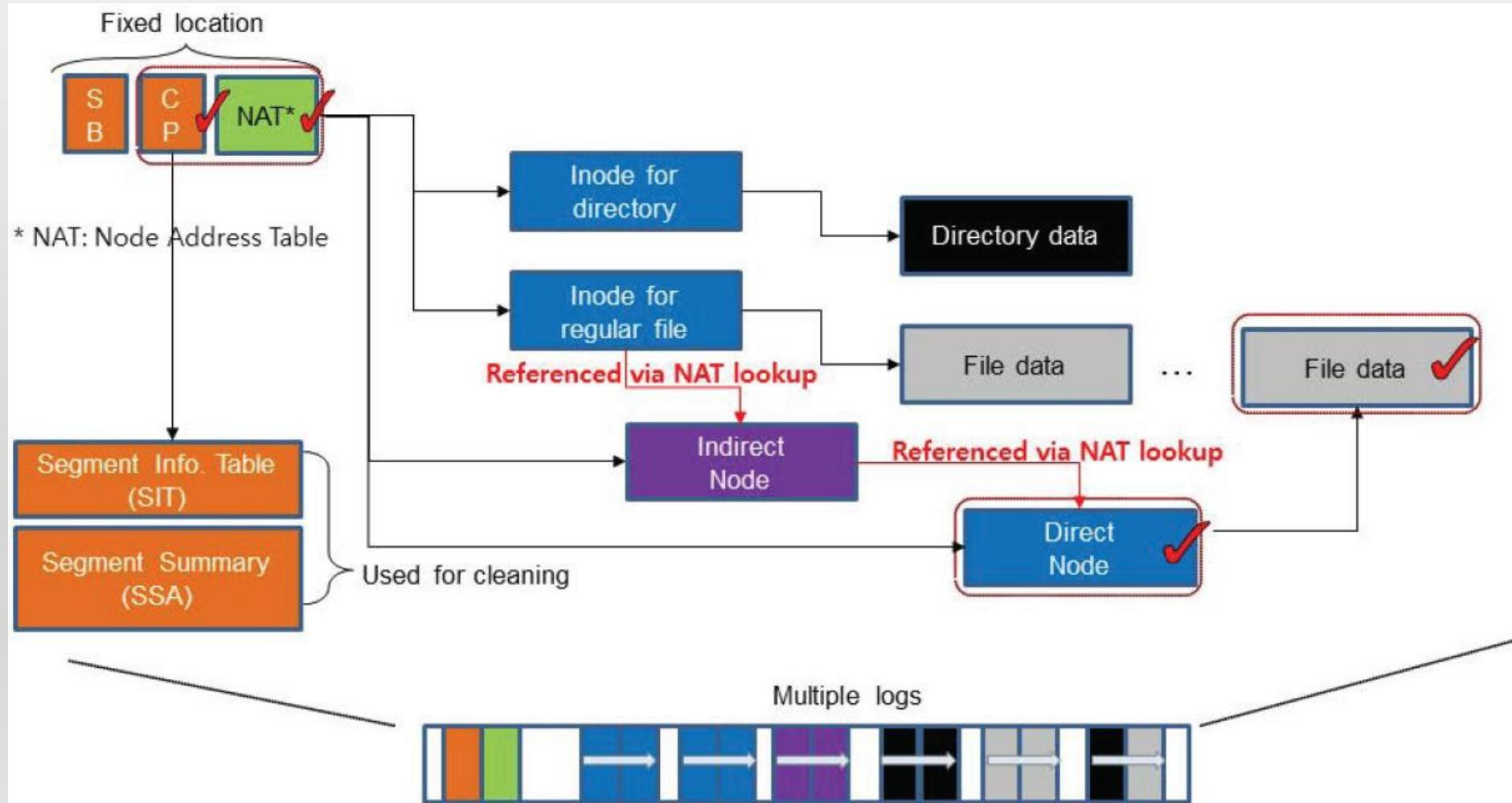
LFS Index Structure

- LFS manages the disk space as one big log.
- LFS has the update propagation problem.



F2FS Index Structure

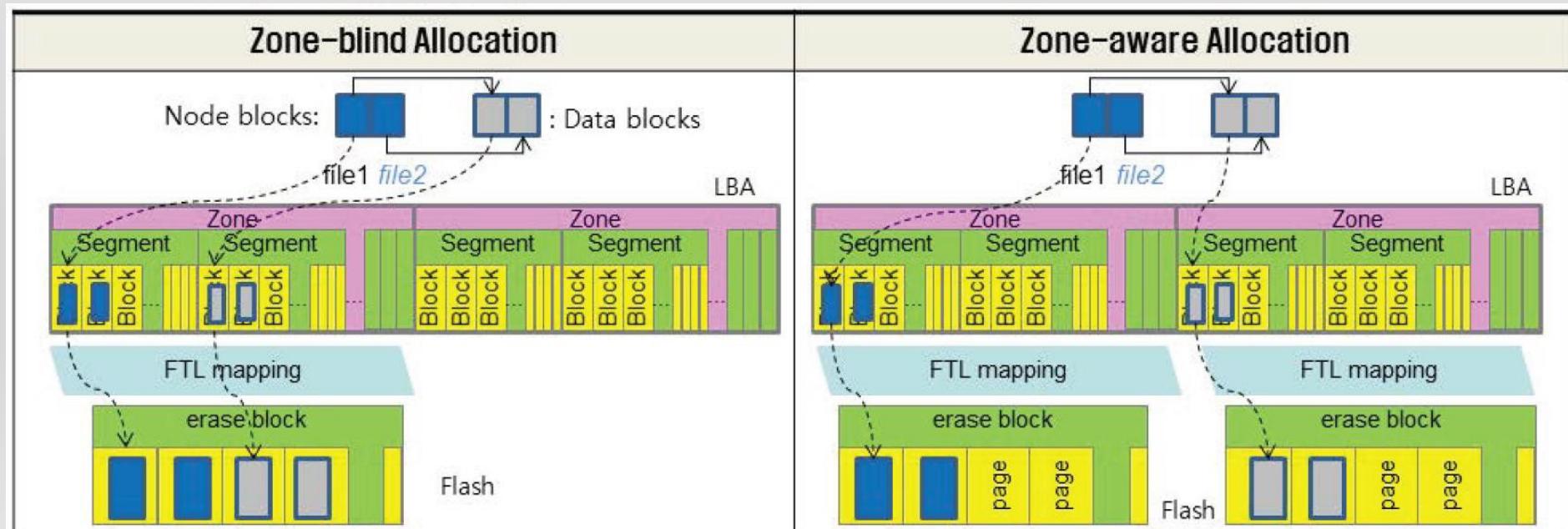
- Restrained update propagation by *node address table*.
- F2FS manages the flash space as *multi-head log*.



Multi-head Logging

- **Data temperature:**
 - Node > Data
 - Direct Node > Indirect Node
 - Directory > User File
- **Separation of multi-head logs in NAND flash**
 - Hot/cold separation reduces the cleaning (GC) overhead.

Type	Temp.	Objects
Node	Hot	Direct node blocks for directories
	Warm	Direct node blocks for regular files
	Cold	Indirect node blocks
Data	Hot	Directory entry blocks
	Warm	Data blocks made by users
	Cold	Data blocks moved by cleaning; Cold data blocks specified by users; Multimedia file data



That's all for today.