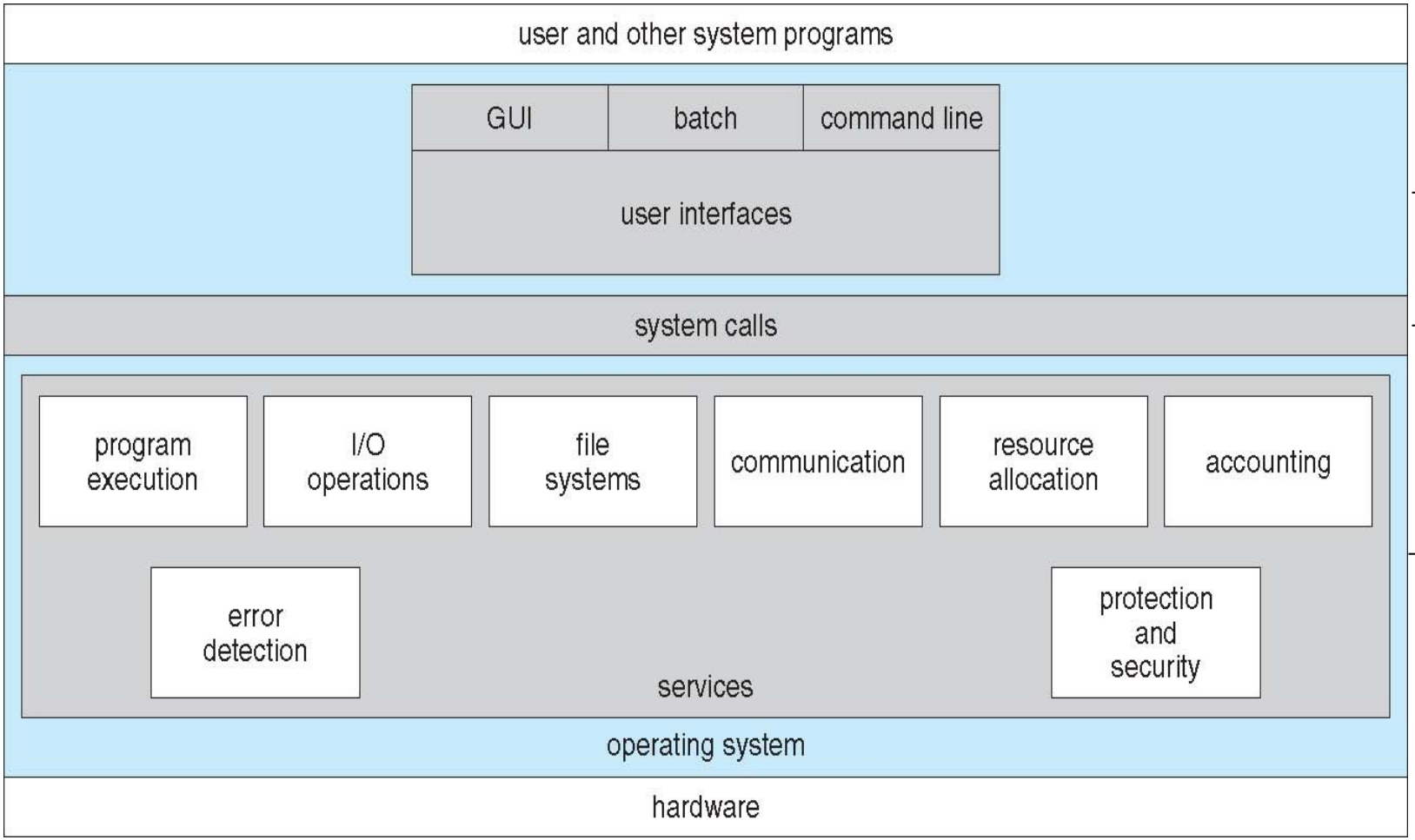
Course
TÖL401G: Stýrikerfi / Operating Systems
2. Operating System Structures

Chapter Objectives

- Identify services provided by an operating system.
- Illustrate how system calls are used to provide operating system services.
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems.
- Illustrate the process for booting an operating system.
- Learn tools for monitoring operating systems.

Contents

Operating System Services, User Interfaces, and System Calls: Overview



2.1 Operating System Services: Services for the user/application (1)

- Operating system provides services that are helpful to the user and application programmer:
  - User interface:
    - More on this in section 2.2.
  - Program execution:
    - The system must be able to load a program into memory and to run that program, to end execution either normally or abnormally (indicating an error).
  - I/O (Input/Output) operations:
    - A running program may require I/O, which may involve a file (see below) or a particular I/O device.
  - File-system manipulation:
    - Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.

Operating System Services: Services for the user/application (2)

- Further helpful operating system services:
  - Communications:
    - Processes may exchange information, on the same computer or between computers over a network.
    - Communications may be via shared memory or through message passing (packets moved by the OS).
  - Error detection:
    - OS needs to be constantly aware of possible errors:
      - May occur in CPU & memory hardware, in I/O devices or user program.
      - For each type of error, OS should take the appropriate action to ensure correct and consistent computing.
      - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system.

Operating System Services: Services for the operation of the system itself (1)

- OS services for ensuring the efficient operation (=efficient sharing of resources) of the system itself:
  - Resource allocation:
    - Resources need to be allocated to users or jobs running concurrently.
    - Allocation strategies (and corresponding implementation) dependent on type of resources:
      - Allocation strategy may be specific to resource, e.g. CPU scheduler, memory management, file storage.
      - Generic allocation strategies for other resources, e.g. all I/O devices may be allocated and released in the same way.
  - Accounting:
    - To keep track of which users use how much and what kinds of computer resources (e.g. for billing or for identifying bottlenecks).

Operating System Services: Services for the operation of the system itself (2)

- Further OS services for ensuring the operation of the system itself:
  - Protection and security:
    - Access of information stored in a multiuser or networked computer system must be controllable.
    - Concurrent processes should not interfere with each other.
    - Protection: involves ensuring that all access to system resources from system users is controlled.
    - Security: prevent access from outsiders. Requires user authentication, extends to defending external I/O devices from invalid access attempts.

2.2 User Interface of Operating Systems

- Almost all OS have a user interface (UI).
  - Varies between
    - Interactive UI, e.g.:
      - Command-Line Interface (CLI), see next slide.
      - Graphical User Interface (GUI), see slide after next slide.
    - Non-interactive UI, e.g.:
      - Batch interface:
        - User specifies all details of a batch job in advance to the actual batch processing. (Using some job submission language that is similar to the command-line interface.)
        - No prompt for further input after the processing has started.
        - User receives the output when all the processing is done.

Command-Line Interface (CLI)

- CLI allows direct command entry via keyboard.
  - Output as simple characters on the monitor.
- Sometimes CLI implemented in kernel, sometimes by a separate system program (shell).
  - Sometimes multiple flavors of shells implemented.
- Primarily fetches a user textual command and executes it.
  - Sometimes commands built-in into shell,
  - Sometimes commands are just names of external programs that will be executed in a new process (eg.:from a system library).
    - Adding new features doesn't require shell modification.

Graphical User Interface (GUI)

- User-friendly graphical interface based on desktop metaphor (invented at Xerox PARC in the early 1970ies).
  - Usually mouse, keyboard, and monitor.
  - Icons represent files, programs, actions, etc.
  - Various mouse buttons in objects interface cause various actions.
  - Mobile systems lack a mouse: touch interface with gestures.

- Many systems now include both CLI and GUI interfaces:
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X as "Aqua" GUI interface with UNIX kernel underneath and shells available.
  - GNU/Linux is CLI with optional GUI interfaces (Gnome, KDE, etc.)
    - Android is Linux kernel with touch GUI and frameworks from Google.

2.3 System Calls

- Include program interface to services provided by the OS.
  - Technically realised as software interrupts (traps) callable via special assembly-language trap instruction.
    - Application Binary Interface (ABI).
  - Mostly accessed by programs via an assembly-language independent library providing high-level Application Programming Interface (API) rather than direct system call use.
    - API may be operating system specific or/and programming language specific (see next slide).
  - (Note that system-call names used in this course are often POSIX names ($\rightarrow$next slide), but sometimes, we abstract from a certain operating system and use rather generic system-call names.)

System Calls: APIs

- Most common OS-specific APIs:
  - Windows API for Microsoft Windows.
    - Corresponding libraries available for all major MS programming languages.
  - POSIX API for POSIX-based systems, i.e. virtually all commercial versions of UNIX, e.g. Mac OS X.
    - (POSIX is a standard for the API of UNIX-like operating systems.)
    - C library that maps directly to system calls of POSIX-based systems.
    - For non-POSIX systems, a POSIX API library may try to emulate POSIX functionality (e.g. CYGWIN for MS Windows or MS Windows Subsystem for Linux).
- Many programming languages come with operating system-independent APIs for accessing common OS services (for specific OS services, still the OS API needs to be used). E.g.
  - Standard C library,
  - Java API for the Java virtual machine (JVM).
  - Internally, these make then system calls using OS-specific API.

System Call Implementation

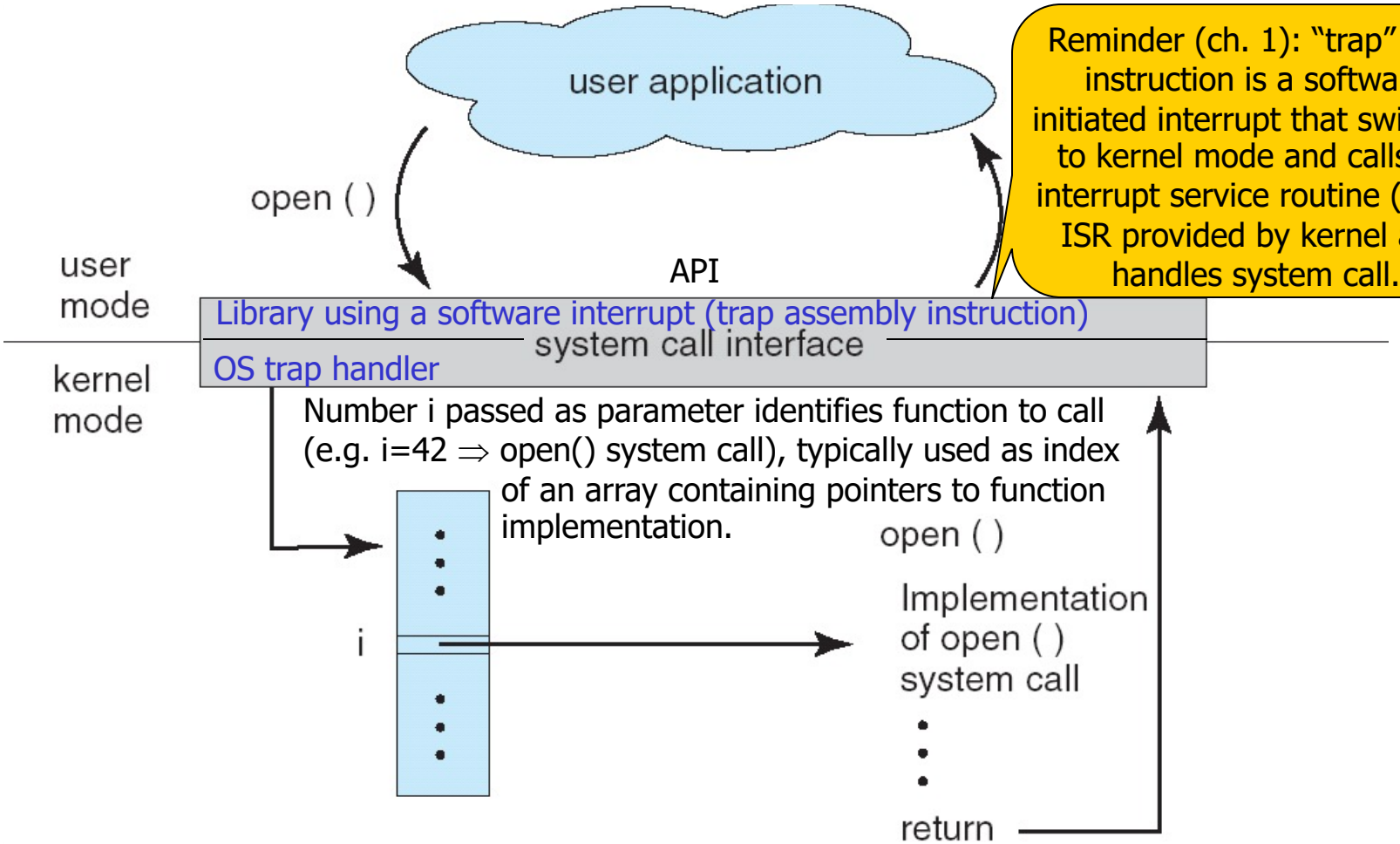
- Typically, a number is associated with each system call.
  - Number identifies associated routine in OS kernel.
  - Number is used as an index of a table that contains addresses of functions that implement the corresponding system call.

- The system call API invokes intended system call in OS kernel by passing number (and additional parameters) using a trap assembly instruction. Result of the system call is retrieved by API and returned to caller.

- Caller needs not knowing how system call is implemented
  Just needs to obey API and understand what OS will do as result.
  - Most details of OS interface hidden from programmer by API .

API – System Call – OS Relationship

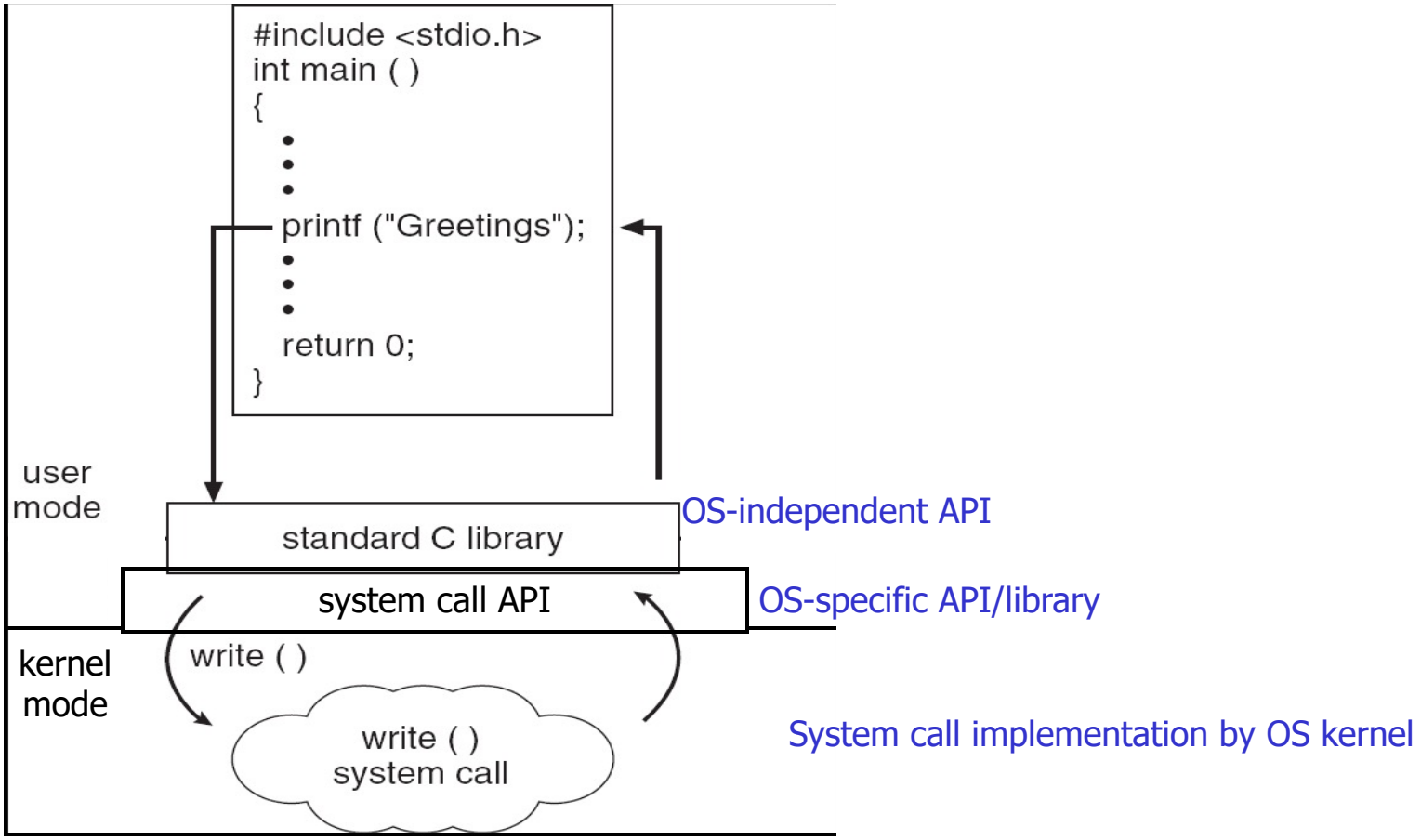
- User application invoking open() call offered by API, which calls the open() system call:



Reminder (ch. 1): "trap" CPU instruction is a software initiated interrupt that switches to kernel mode and calls an interrupt service routine (ISR). ISR provided by kernel and handles system call.

Standard C Library Example

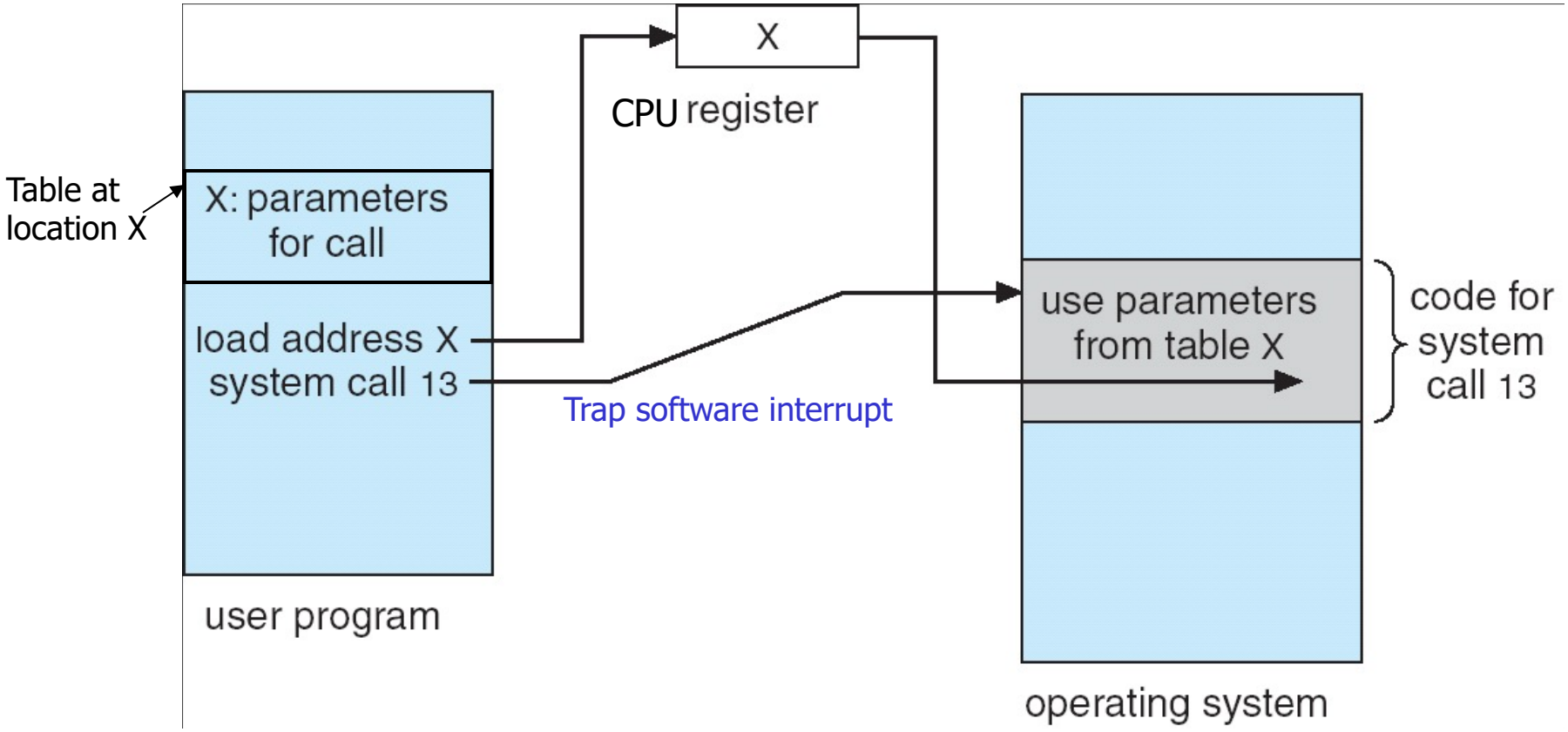
- C program invoking printf() Standard C library call, which calls write() system call:



System Call Parameter Passing

- Three possible approaches passing parameters to the OS?
  - Pass the parameters in CPU registers. Simple, but not sufficient
    - If more parameters than registers shall be passed.
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register (see next slide).
    - This approach taken by Linux and Solaris.
  - Parameters placed, or pushed, onto the stack (pointed to by the Stack Pointer (SP) CPU register) by the program and popped off the stack by the operating system.
  - Block and stack methods do not limit the number or length of parameters being passed.

Parameter Passing via Table



Difference API ↔ ABI

- Application Programming Interface (API) is the interface used by a high-level programming language program.
  - If API changes (e.g. new function/method name or change of parameters) then the whole application program source needs to be adjusted to be able to re-compiled.
- Application Binary Interface (ABI) is the interface used on assembly/machine code level.
  - ABI change: different number used for an existing system call or different order or format of parameters (=caller & callee assume different parameter passing tables).
    - All compiled programs (=compiled into binary code and making use of the ABI) will fail when trying to call this system call.
    - As long as the API did not change, it is enough to recompile the source code of the application program to deal with ABI changes: the library that implements the API will then internally use the new ABI. (Of course library, needs to updated to the new ABI, first.)
- ABI changes not only a problem at OS level, but also for Java, C++ etc:
  - E.g. method parameter (=API & ABI change) change in Java class B
    ⇒ Java class A calling method from B, but compiled using old API contains byte code that passes now wrong parameters (=relying on old ABI).
- Both, API and ABI changes should be avoided (rather add additional calls).

Documentation of POSIX System Calls

If man pages for system calls are not installed on your Linux system, install packages: **`manpages-dev`** **`manpages-posix-dev`** (your Linux distribution may use different names).

- On most POSIX systems (Linux, Mac OS X etc.), a manual page ("man page") can be called from command line to obtain further information of how to use a system call from within a C program:
  - **`man -S s x`** provides documentation for system call ***`x`***:
    - Required C include files,
    - Parameters and return values,
    - Short description,
    - Related system calls,
    - And many further useful information.
  - ***`s`*** specifies section of manual, e.g.:
    - 1 Shell command or system program,
    - 2 System calls (Platform specific system calls as provided by operating system),
    - 3 Library calls (Functions provided by C standard library).
    - If **`-S s`** is not used as parameter, the first section having an entry for ***`x`*** is used.
- Example:
  - **`man kill`** : description of shell command kill (kill first found in section 1),
  - **`man -S 2 kill`** : description of system call kill.

How to read man page of an API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

2.4 Types of System Calls

- Roughly six major categories of system calls (most of them refer to related OS services):
  - Process control,
  - File management,
  - Device management,
  - Information maintenance,
  - Communications,
  - Protection.
- (Most of these will be covered in detail in later chapters. On the following slides, only a brief overview is given.)

Process Control System Calls

- End, abort (=abnormal halt) process.
- Load, execute process.
- Create process, terminate process.
- Get/set process attributes.
- Wait for a certain amount of time.
- Signal event, wait for event.
- Allocate memory, free memory.

2.5 System Programs/ System Services

- System programs are delivered together with an OS and provide a convenient environment for program development and execution.
  - Some of them are simply user interfaces to system calls; others are considerably more complex.
  - Most users' view of the operation system is defined by system programs, not the actual system calls.
- In the past, these were mainly command line tools.
  - (e.g. **`format`** to create a new file system, **`cp`** to copy a file.)
- Nowadays, these are provided as a system service via a GUI.

System Programs

- System programs can be divided into (details on the next slides…):
  - File management,
  - Status information,
  - File modification,
  - Programming language support,
  - Program loading and execution,
  - Communications,
  - Application programs,
  - Background programs.

Categories of System Programs (1)

- File management:
  - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- Status information:
  - Some ask the system for info:
    - date, time, amount of available memory, disk space, number of users.
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems (e.g. MS Windows) implement a registry:
    - Used to store and retrieve configuration information.
- File modification:
  - Text editors to create and modify files.
  - Special commands to search contents of files or perform transformations of the text.
- Programming-language support:
  - Compilers, assemblers, debuggers and interpreters.

Categories of System Programs (2)

- Program loading and execution:
  - Command interpreter (shell).
  - Linker, tracing of system calls.
- Communications:
  - Provide the mechanism for creating virtual connections among processes, users, and computer systems.
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another.
- Application programs:
  - Additional programs for solving application specific problems (Word processor, graphic editor, games, etc.)
- Background programs ("daemons"):
  - Print spooler for buffering print jobs while printer is busy, syslog for logging events, execution of commands at scheduled times ("cron jobs"), etc.

2.6 Operating System Design and Implementation

- There is no standard design for operating systems.
  - Internal structure of different operating systems can vary widely.
- Proven approach:
  - Start by defining goals and specifications:
    - Choice of hardware, type of OS (batch, multitasking, etc.).
    - Identify User goals and System goals:
      - User goals: operating system should be convenient to use, easy to learn, reliable, safe, and fast.
      - System goals: operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient.

Operating System Design and Implementation: Mechanisms and Policies

- Separate policy from mechanism:
  - A fixed mechanism is responsible for executing a variable policy.
  - Example:
    - A scheduling mechanism assigns CPU time to processes.
    - The scheduling policy specifies the criteria to use for scheduling, e.g.
      - Multitasking scheduling policy: switch process every 20ms.
      - Batch scheduling policy: switch process just when a process is waiting for I/O to finish.
  - Policies are likely to change (just like in politics).
  - A policy-independent mechanism may support different policies without needing to re-implement the mechanism.
- Separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are likely to change.

Operating System Design and Implementation: Implementation

- Initially, operating systems implemented using assembly language.
- Nowadays, operating systems usually implemented using higher-level languages, in particular C.
  - Small parts using assembly language, e.g. code responsible for saving and restoring CPU registers (context switch, e.g. as part of interrupt handler).
  - Advantage of using higher-level programming languages:
    - OS is easer to port to other CPUs.
  - Disadvantage of using higher-level programming languages:
    - Reduced speed of operating system.
  - Compromise: implement time critical parts in assembly language, e.g. scheduler.

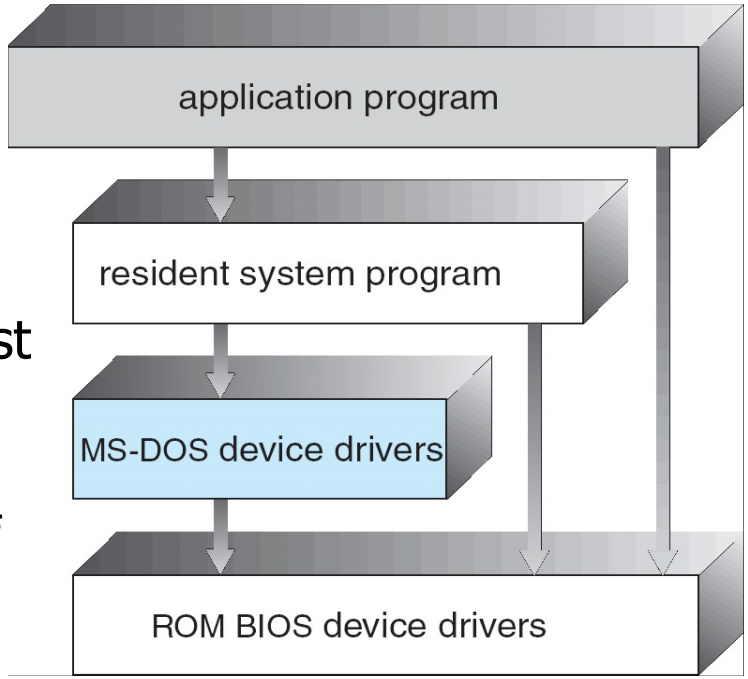
2.7 Operating System Structure

- Different design of operating systems results in operating system structures
  (i.e. how components of an OS are arranged and interconnected):
  - Simple structure: Monolithic system,
  - Layered system,
  - Microkernel system,
  - Module-based system.

    (We will have a closer look at each of them on the next slides…)

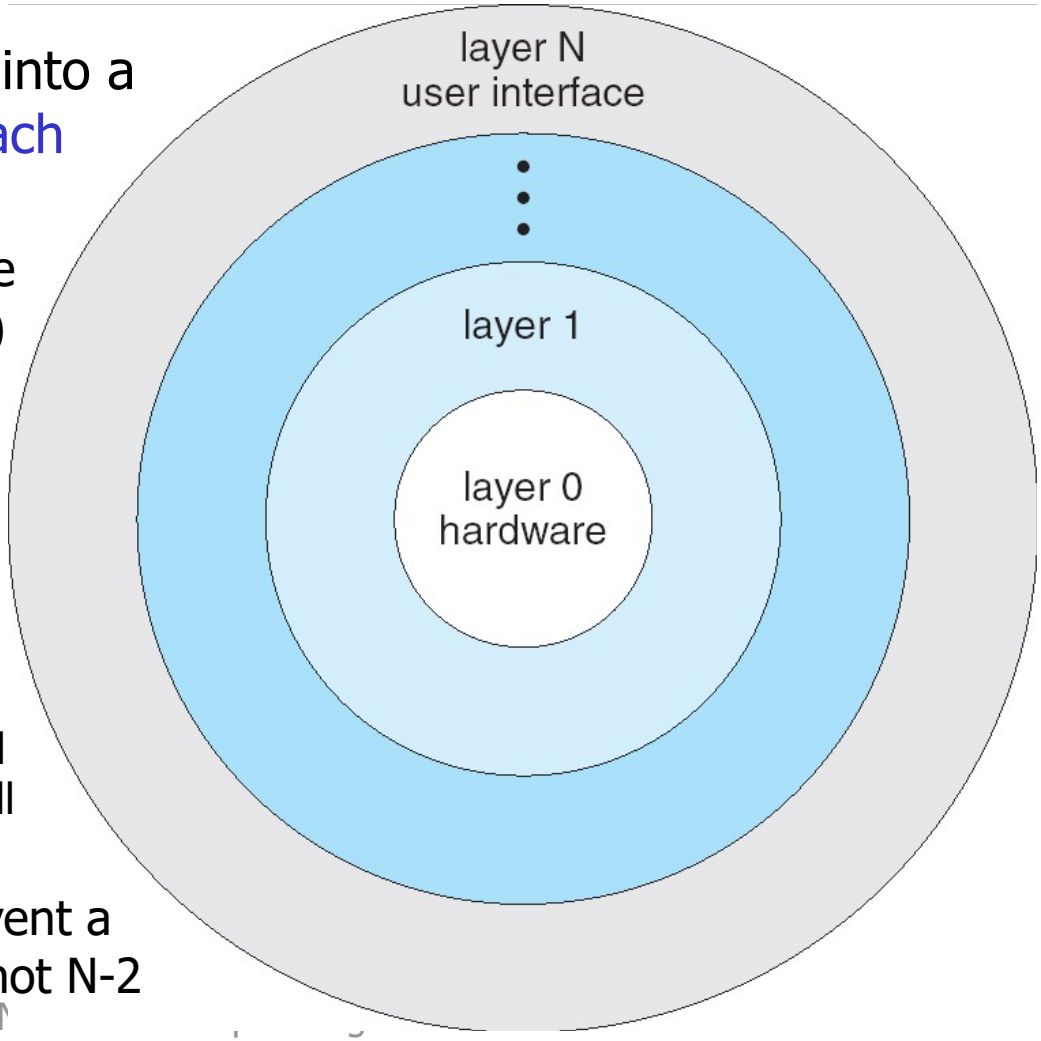
Simple Structure: Monolithic System

- Operating system not divided into separated modules or layers.
  - Layers may exist, but an application may circumvent layers and thus directly access hardware, making the system vulnerable.
- Example:
  - MS-DOS: written to provide the most functionality in the least space.
    - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.
    - Direct access to hardware possible.



Layered Approach

- Operating system is divided into a number of layers (levels), each built on top of lower layers.
  - Bottom layer (layer 0) is the hardware; highest (layer N) is the user interface.
  - Layers are designed such that each layer uses functions (operations) and services of only lower-level layers.
    - E.g.: While layer 2 may call layer 1, layer 1 may not call layer 2.
  - It is not possible to circumvent a layer: N can only call N-1, not N-2 (only via N-1).



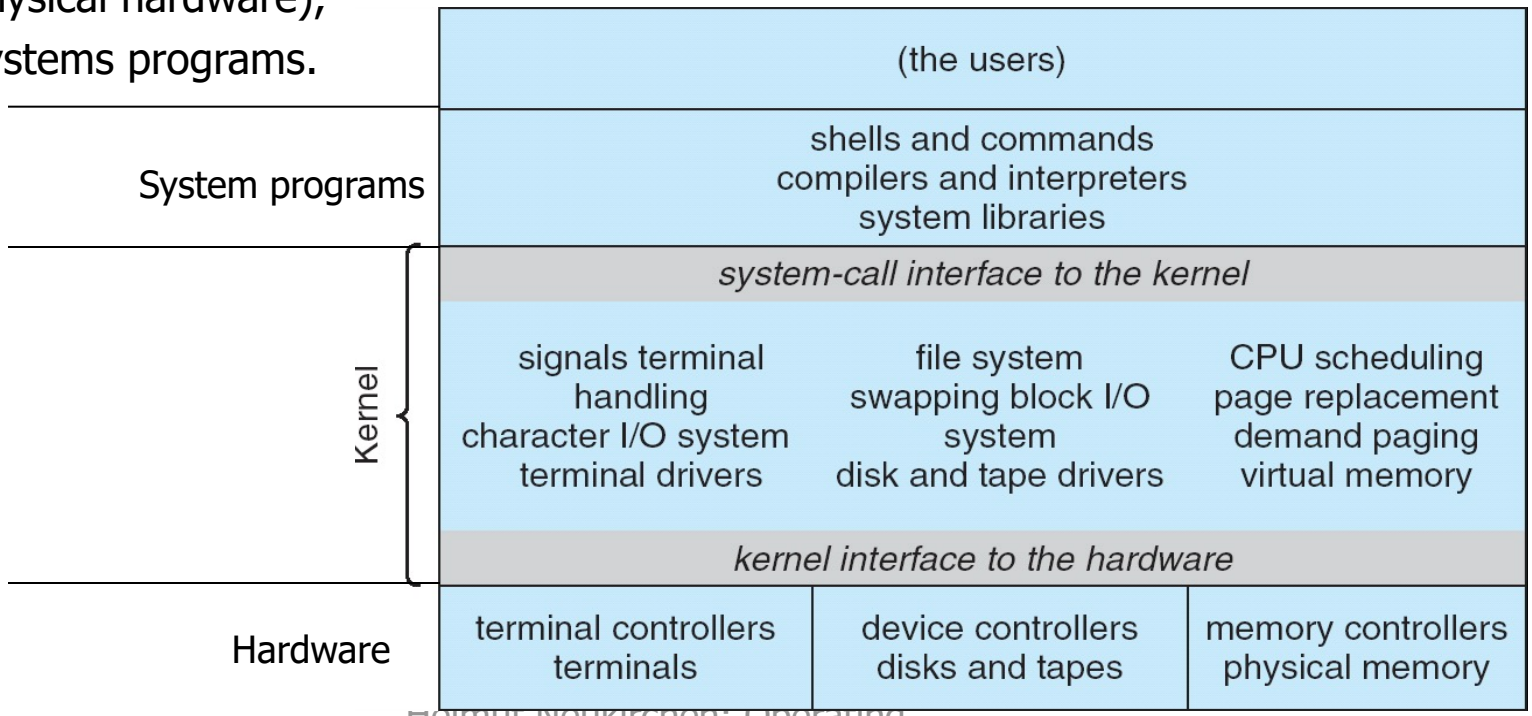
Layered Approach: Advantages and Disadvantages

- Advantages:
  - Layers hide complexity.
    - Development is eased.
  - Layers are easy to exchange.
    - E.g. replacing the layer directly above the hardware may be sufficient to port an operating system to another hardware.

- Disadvantages:
  - Good separation of layers may be difficult.
    - If component A needs to call component B and component B needs to call component A, they must reside in the same layer.
  - Low speed as each layer involves an additional function call adding some overhead.

Layered Approach: Example: UNIX

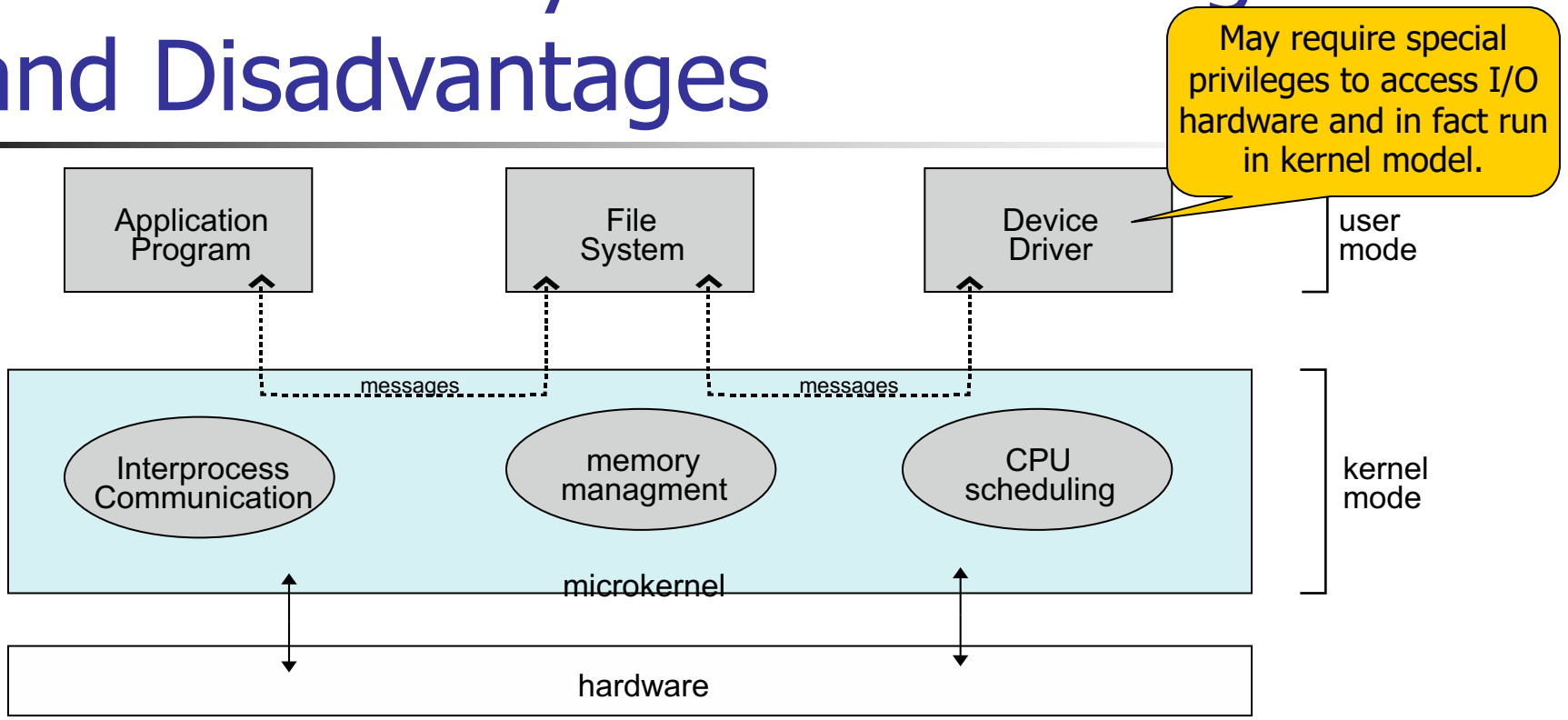
- The original UNIX operating system had limited structuring: Only two separable layers (hence, you could consider it also as a simple structure, quite monolithic system):
  - The kernel (everything below the system-call interface and above the physical hardware),
  - Systems programs.



Microkernel System

- Moves as much from the kernel into "user" space (i.e. ordinary processes running in user mode instead of kernel mode).
  - Microkernel (running in kernel mode) consists only of minimal functionality for process and memory management and communications.
  - All additional functionality is provided by user space modules.
  - Communication takes place between user space modules using message passing (via microkernel system calls for inter-process communication).
    - E.g. instead of accessing an internal file system layer via a system call, a user process that provides file system functionality needs to be contacted by sending a message to it that requests file access.
      - File system process may in turn send a message to a device driver process.

Microkernel System: Advantages and Disadvantages



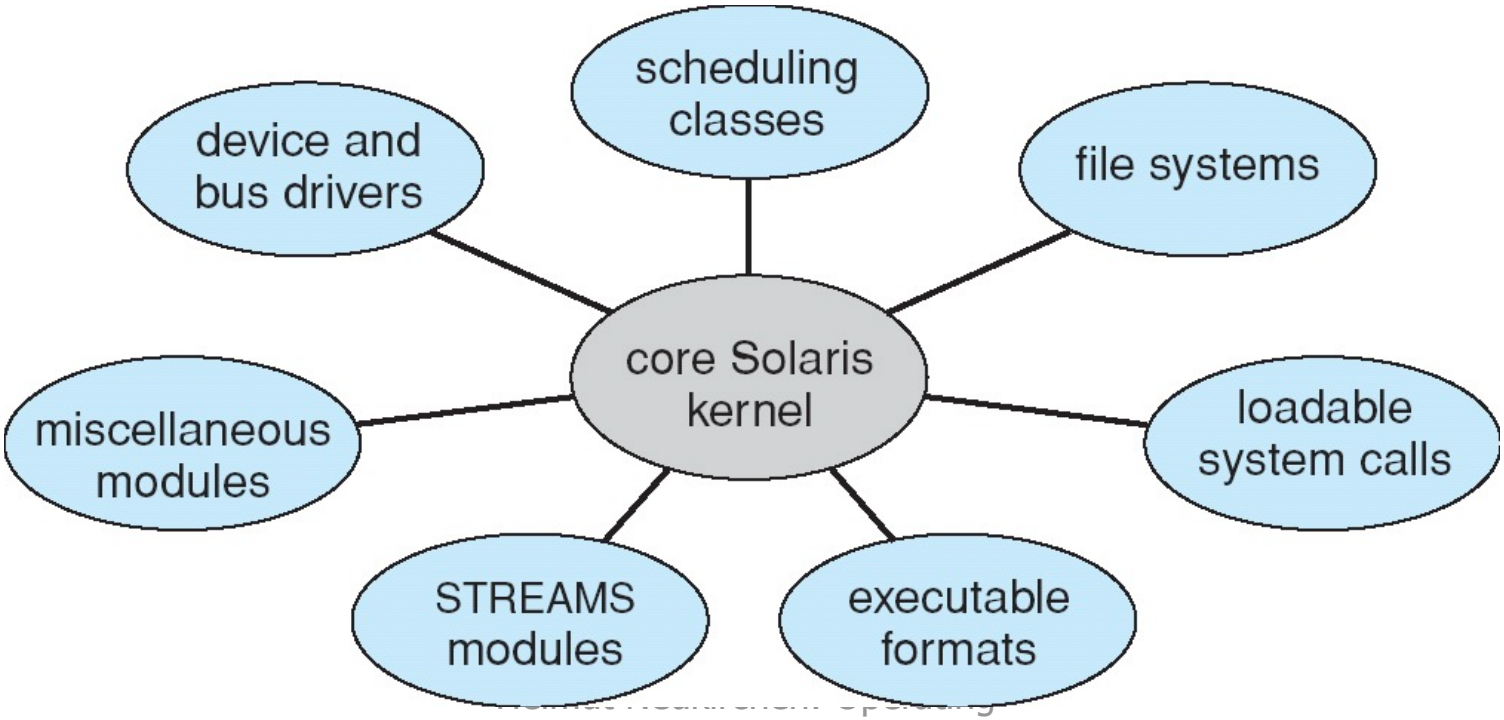
- Advantages:
  - Easier to extend a microkernel system: just add user space module.
  - Easier to port the operating system to new architectures: just port kernel.
  - More secure & more reliable (less code is running in kernel mode).
- Disadvantages:
  - Huge performance overhead due to user space to kernel space (and back) communication.

Module-based Operating System

- Most modern operating systems (e.g. Linux, Solaris, Mac OS X, even Windows) implement kernel modules:
  - Kernel consists only of core functionality.
  - Functionality can be added by loading kernel modules at run-time.
    - Similar to object-oriented approach:
      - Each class of kernel module (file system, device driver etc.) has a well defined interface.
      - Each kernel module implements this interface.
  - Mixture of Microkernel and Layers, but
    - Faster than microkernel: no message passing required for communication, instead: kernel and kernel modules run in kernel space and may use ordinary function calls to call each other.
    - More flexible than layered approach: each class of kernel module may call functions from another class of kernel module (no strict hierarchical layers), additional kernel modules may be added at run-time.

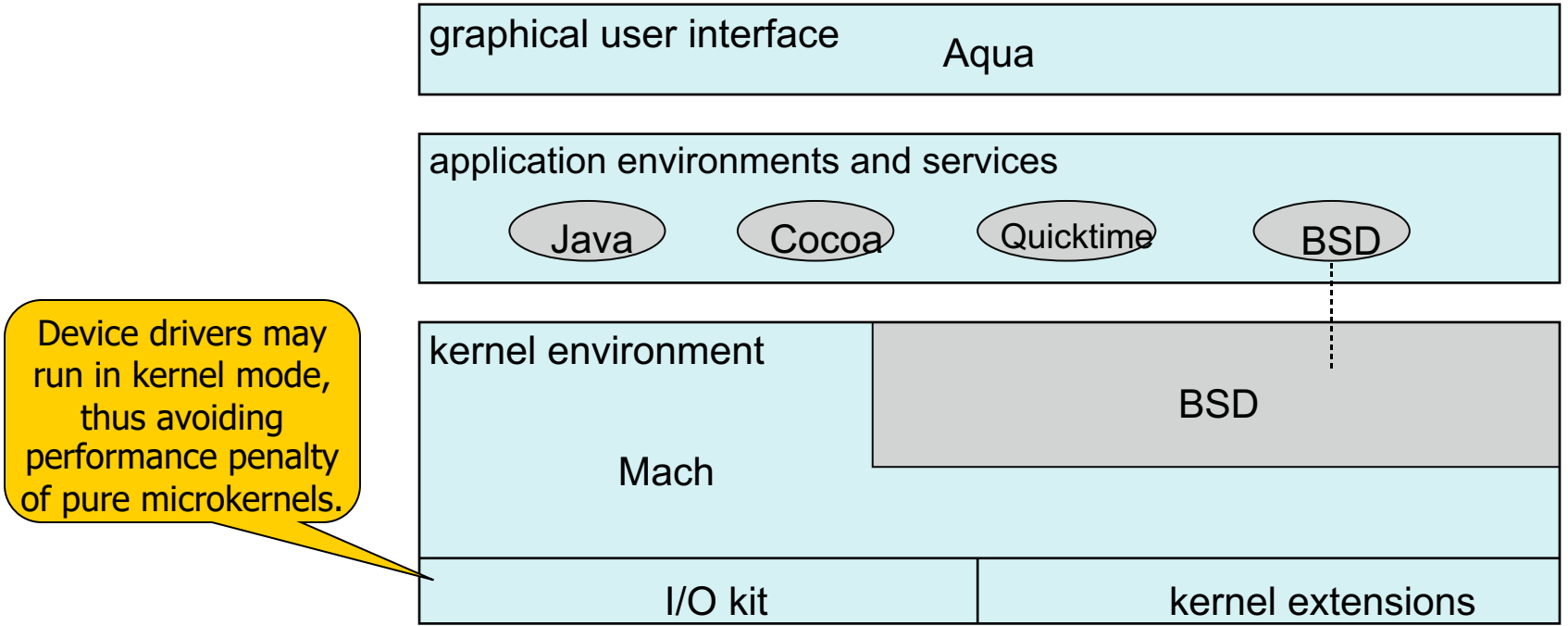
Example: Solaris Modular Approach

- 7 different classes of kernel modules. Allows at run-time to, e.g.,
  - exchange scheduler,
  - add support for devices,
  - add support for further file systems.



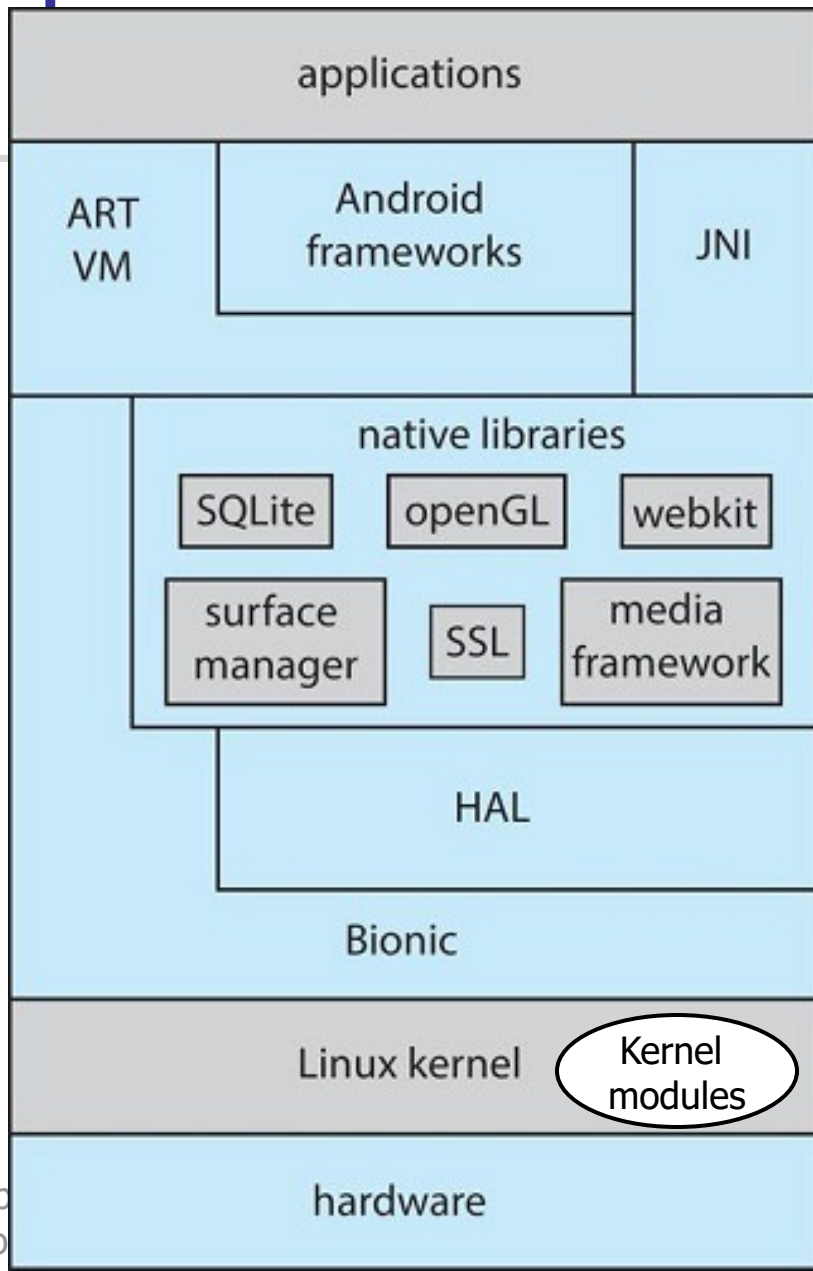
Hybrid Systems example: Apple Mac OS X (iOS is similar)

- Hybrid structure (microkernel and layered):
  - Based on Mach microkernel.
    - Some common services running as user space processes.
  - Monolithic/Layered BSD kernel provides BSD Unix API on top of microkernel (to allow to use the more common BSD Unix API instead of Mach API).
  - Furthermore: support for kernel modules.



Hybrid Systems example: Android

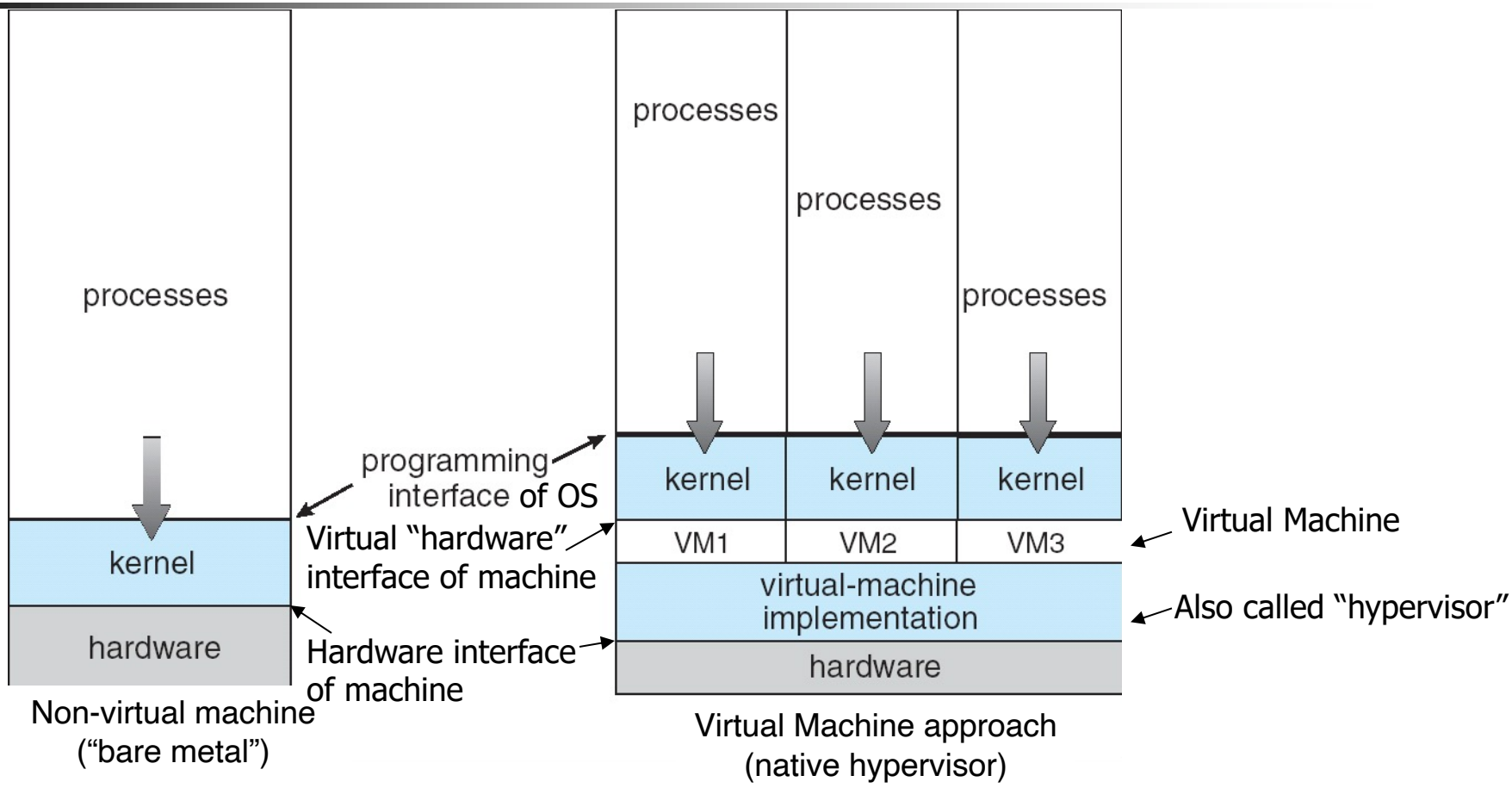
- Module-based Linux kernel with higher layers of libraries and frameworks,
- Android Runtime (ART) virtual machine interprets CPU-independent bytecode (comparable to Java).
  - Also possible to call C code of "native" libraries (=not bytecode, but C code compiled to machine code of the CPU, e.g. built-in SQLite database) via the Java Native Interface (JNI).
  - Hardware Abstraction Layer (HAL) to use hardware in an abstract way, i.e. use a camera without knowing details of it.
  - Bionic is the standard C library used by, e.g. native libraries (that are implemented in C) for doing, e.g., string or math operations that are not part of the C language itself but of a library.



2.8 Virtual Machines (VMs)

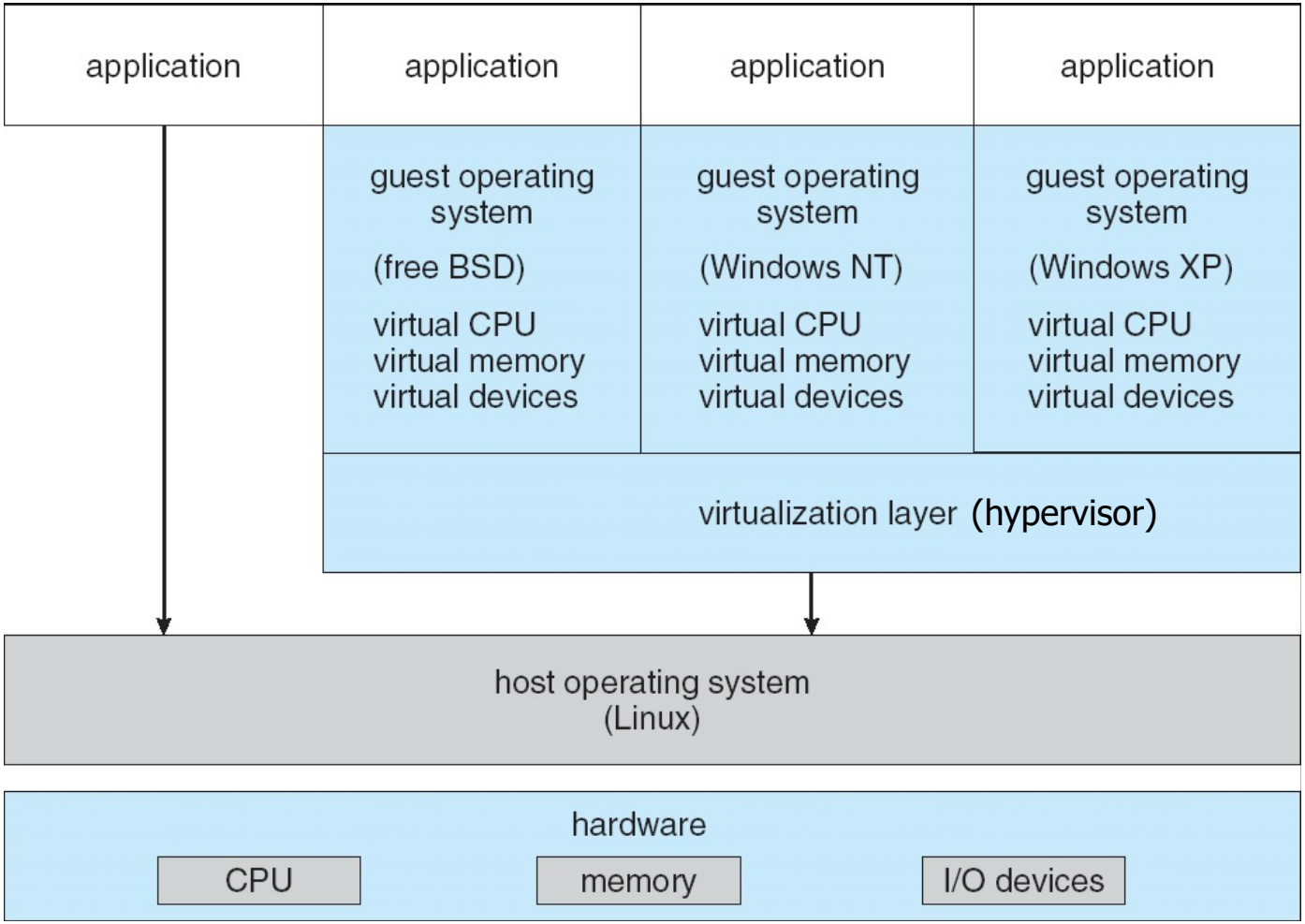
- An operating system (the kernel) runs on top of a machine:
  - CPU, memory, I/O devices, etc. (i.e. the bare hardware).
- A virtual machine manager/monitor (VMM) or hypervisor is a low-level software that provides an interface (=a virtual machine) identical to the underlying bare hardware.
  - Extreme case: a machine code interpreter simulating another CPU.
  - Software running on top of ("inside") the virtual machine has the impression of running directly on the real hardware.
- Using virtual machine technology, multiple operating systems may run in parallel on the same real machine.
  - Just like a timesharing/multi-tasking (→ch. 5) OS that shares the CPU, memory and devices between different processes, a hypervisor shares all the hardware resources between the different operating systems running insider their virtual machine.

Non-Virtual Machine vs. Virtual Machines



- Native hypervisor: Runs on top of bare hardware – all OSes run on top of it.
- Hosted hypervisor: Runs on top of an ordinary OS that runs on top of bare hardware.

Example: Hosted hypervisor (e.g. VMware Workstation Player)



Some device drivers of guest OS need to be exchanged by device drivers that make calls to the underlying hypervisor instead of directly accessing hardware (para-virtualisation).

Support for hypervisor needs to be provided by host OS.

Host OS runs directly on bare hardware.

Advantages/Disadvantages

- Advantages:
  - Protection between different VMs: problems (e.g. virus) in guest OS does not affect host OS (even not the hard disk: hard disks are only virtual).
  - New/different OS can easily be installed/tested in addition to existing OS and may even run in parallel to it.
    - E.g. best of Windows, Linux, Mac worlds on one machine in parallel.
  - Guest OS (and its processes) may be suspended and resumed again, e.g. for replacing hardware without reboot of guest OS or for migration to another (less busy) server. (Cloud computing is heavily based on virtualisation.)
  - Easier OS development/debugging/experiments:
    - No need to exit development environment: just start new OS in parallel on VM.
    - Debugger may be used on host OS to debug guest OS.
    - No fears that files on host hard disk get destroyed if guest OS may only access virtual disk.
  - If hypervisor implements even a complete CPU in software ("emulation"), machine code for a completely different CPU becomes executable (but slow).
- Disadvantages:
  - Slightly slower (rule of thumb: 80% of the native speed) (reasons: see next slides).
  - Often not all possible hardware devices supported.

Implementation of VMs: CPU virtualisation

- Hypervisor must virtualise all hardware resources:
  - CPU:
    - Hypervisor has CPU scheduler for sharing CPU time between guest operating systems.
      - Similar to CPU scheduler of timesharing/multi-tasking OS.
    - User mode/kernel mode: Guest OS must not be able to access resources directly using kernel mode. (Nevertheless, guest OS assumes that all of its interrupt handlers are running in kernel mode and thus tries to access physical devices instead of virtualised ones.)
      - VM must provide a virtual kernel mode and virtual user mode. However, even in virtual kernel mode, the guest OS is actually in physical user mode.
      - If guest OS performs an action that requires physical kernel mode, this will lead to an interrupt: interrupt handler of hypervisor is called: hypervisor (running in physical kernel mode) will analyse action and perform a corresponding action on behalf of the guest OS and finally resumes execution of guest OS instructions (in virtual kernel mode).

Modern CPUs may provide additional features to support virtualisation, e.g. additional CPU modes ("real kernel mode" for hypervisor, virtualised kernel mode for OS, user mode.)

Implementation of VMs: Memory and device virtualisation

- Hypervisor must virtualise all hardware resources:
  - Memory:
    - Guest OS must not be able to access memory of other operating systems.
      - Virtual memory gives each guest OS the impression of having exclusive access to memory, even though the memory is actually separated.
      - (Based on the same techniques that an OS uses to provide virtual memory to its application. →chapters 8 and 9)
  - Devices:
    - Guest OS must not access physical devices. (Access to devices is only possible in physical kernel mode, thus hypervisor is able to detect this →previous slide.)
    - Depending on the type of device, either a controlled access to the shared physical device is performed or a separate virtual device is simulated for each guest OS. E.g.
      - Keyboard, mouse event are forwarded to guest OS that has focus.
      - Virtualised hard disk of each guest OS is just a file on a physical file system.

Software Container, e.g. Docker

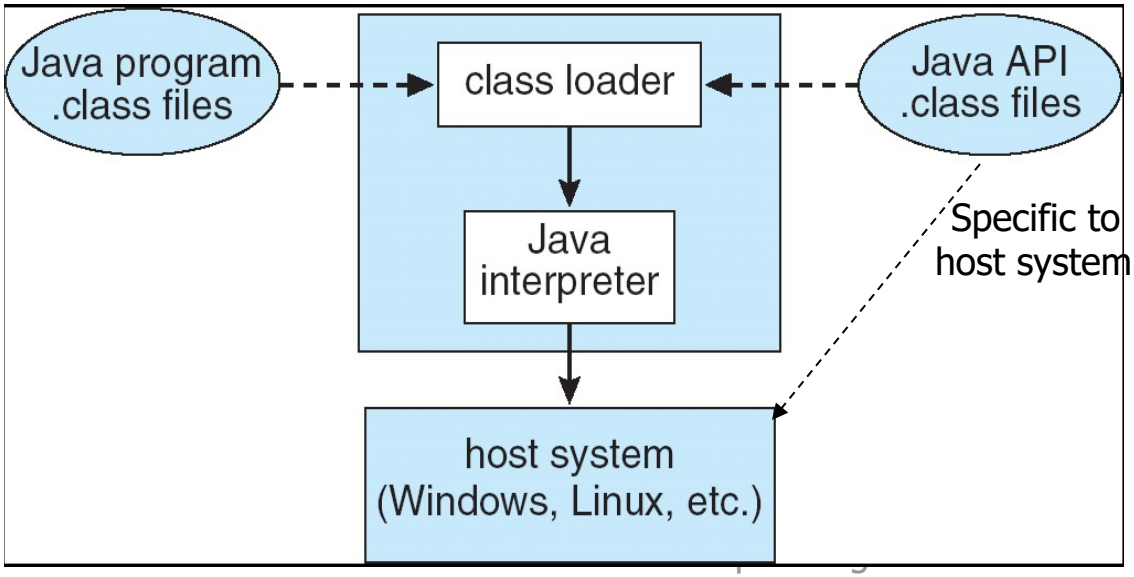
- Assume, both VM1 and VM2 run the same Linux kernel (incl. kernel modules) and system libraries (=you do not want to run different OSes):
  - Waste of disk space and RAM.
    - Store two times identical kernel, modules, libraries on two different virtual files systems, keeping them two times in RAM.
- Software container do the virtualisation at a higher layer (inside kernel):
  - Load only one kernel into memory: the kernel does the virtualisation.
    - Only disk and RAM space for a single kernel needed.
    - Kernel may even support to share identical files (such as common libraries) as well has having own file versions in each container.
      - No need to care about: application 1 runs only with library version 1, application 2 only with library version 2 $\Rightarrow$ Instead, each application has its own container with own libraries and other software it depends on (=eases installation).
    - No overhead for virtualising hardware accesses.:
      - The single kernel runs already on the bare hardware.
      - Kernel needs however to be able to separate container from each other.
        - E.g. processes in Container1 should not be able to modify files from Container 2.

2.9 Java

- Not immediately related to operating systems (or at least to OS kernels), but an interesting application of virtual machines:
  Java technology:
  - Java programming language,
    - Is compiled into bytecode (.class files).
  - Java libraries (API),
    - Provides access to high-level services and operating system-level services via an operating-system independent API.
      - Implementation of that interface (i.e. the Java library), however, is operating-system dependent: OS-level services of Java API need to be mapped on system calls of respective OS.
  - Java virtual machine.
    - Is able to execute bytecode (more on next slide).

Java Virtual Machine

- Java virtual machine (JVM):
  - Does not run directly on hardware, but on top of OS.
  - Available for various platforms (i.e. operating systems and CPU types).
  - Virtual machine concept (together with OS-specific Java API implementation) allows to execute bytecode without modification on all supported platforms:
    - Interpreter for bytecode ("emulation" of Java CPU) (or just-in-time compiler).



2.12 System Boot

- How is an OS started/booted?
  - ("Booting" = starting a computer by loading the kernel.)
- When power initialized on system, CPU starts execution at a fixed memory location (location hard-coded into CPU).
  - CPU runs in kernel mode after power-on.
- At that memory location, a computer hardware has a ROM or EPROM that contains machine code instructions, generally known as firmware.
  - In older PCs: BIOS (Basic Input Output System)
  - More recently: UEFI (Unified Extensible Firmware Interface)
    - + "Compatibility Support Module" (CSM) for compatibility for programs expecting a BIOS.

System Boot: Bootstrap Program/Boot Loader

- Firmware contains bootstrap program / bootstrap loader:
  - Just enough functionality to locate the kernel on mass storage device, loading it into memory, and starting it.
  - Often, a two-step process where the fixed firmware bootstrap program just loads a small boot block from fixed location of mass storage device and then starts the boot code contained in boot block.
    - BIOS was only able to load boot block from on of the firsts blocks of a storage.
    - UEFI is smart enough to locate, load and start bootstrap code everywhere.
  - Contents of boot block can be easily changed (located on mass storage, not in firmware).
    - E.g. GRUB (GRand Unified Bootloader) on Linux systems
  - As CPU runs in kernel mode, boot code can access hardware.

System Boot: Kernel

- Boot code in boot block then loads kernel from a storage device and hands over control to it.
- The first instruction of the operating system kernel is started (=the OS is now running)
  - Kernel initialises hardware.
    - (in addition to basic initialisation done by BIOS/UEFI.)
  - A root filesystem (=the filesystem that contains most important system programs) is mounted, i.e. OS can access it.
  - Kernel starts system processes (e.g. user interface for logging in) and background processes (Unix: "daemons"/ Microsoft: "service") by loading them from the root filesystem.

2.13 Summary

- OS provides number of service to user/application and itself.
  - Services to user accessible via user interface.
    - May be used to execute system programs.
  - Services to applications accessible via system calls.
    - Instead of using the API offered by the OS, often a programming language-specific API (e.g. C standard lib) is used.
      - Application remains portable as long as programming language-specific API is available on other system. However, for OS specific services, OS API needs to be called.
- Several types of OS designs: today, module-based operating systems are dominant.
  - Supporting the exchange of policies keeps OS mechanisms flexible.
- While virtual machines are usually slightly slower, they have many advantages, e.g. running multiple OS in parallel.
- Different stages of booting involve progressively smarter programs.
  - From firmware (in the past via boot block) to kernel.