Operating System Concepts
Ch. 2: Operating System Structures

Silberschatz, Galvin & Gagne

Universiteit Leiden
The Netherlands

Content

- This chapter goes into more detail on the structure of Operating Systems.
  - Organization of the different components.
  - Different User Interfaces
  - Types of system calls and how system calls are invoked
  - Ways of structuring Operating Systems

Services

- An operating system provides an environment in which programs can be executed.
- As part of this environment it provides several services to programs (and its users).
- The exact services different among operating systems. But several common ones can be identified.

Services (2)

This first set consists of services that are mainly helpful/convenient to the user:

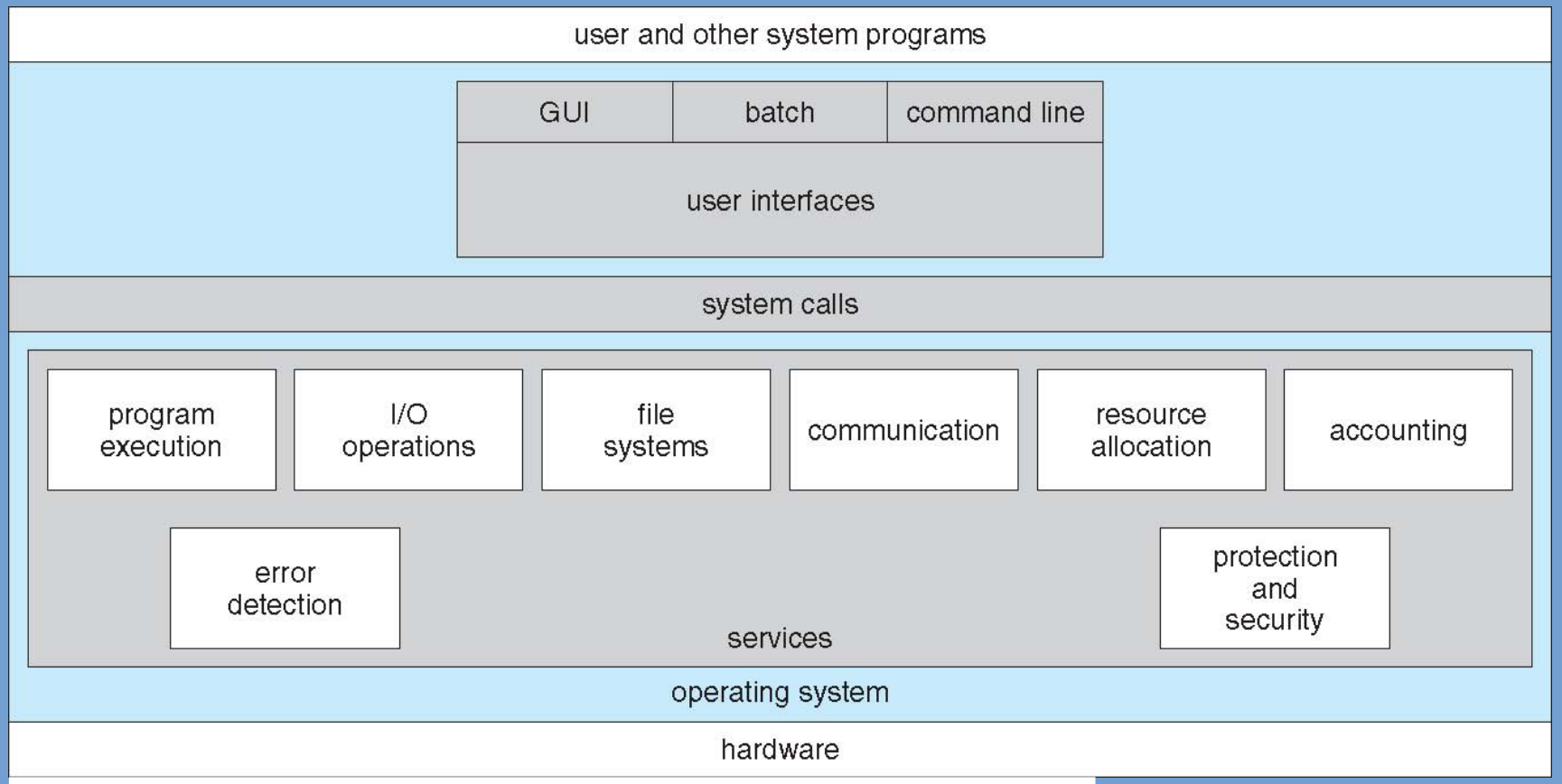
- Provide a **User Interface (UI)**
  - Typically a system program running in user-mode.
  - Distinguish between Command Line (CLI), Graphical (GUI) or batch system.
- Services to allow **program loading & execution**.
- **I/O operations**: access (special) devices. Users cannot access I/O devices directly.
- **File System Manipulation**: read/write files and directories.
- **Interprocess communication**: e.g. shared memory or message passing.
- **Error detection**: detect and correct errors, such as disk I/O errors, network errors, printer errors, faults in user programs.

Services (3)

A second set concerns efficient operation of the system, in particular for multi-user systems:

- **Resource allocation**: allocation algorithms for different resources/devices, how to deal with conflicting requests? How to maximize use of the available resources?
- **Accounting**: keep track of CPU cycles, disk space, etc.
- **Protection & Security**: control access to all system resources, file system ACL, user authentication.

Services (4)

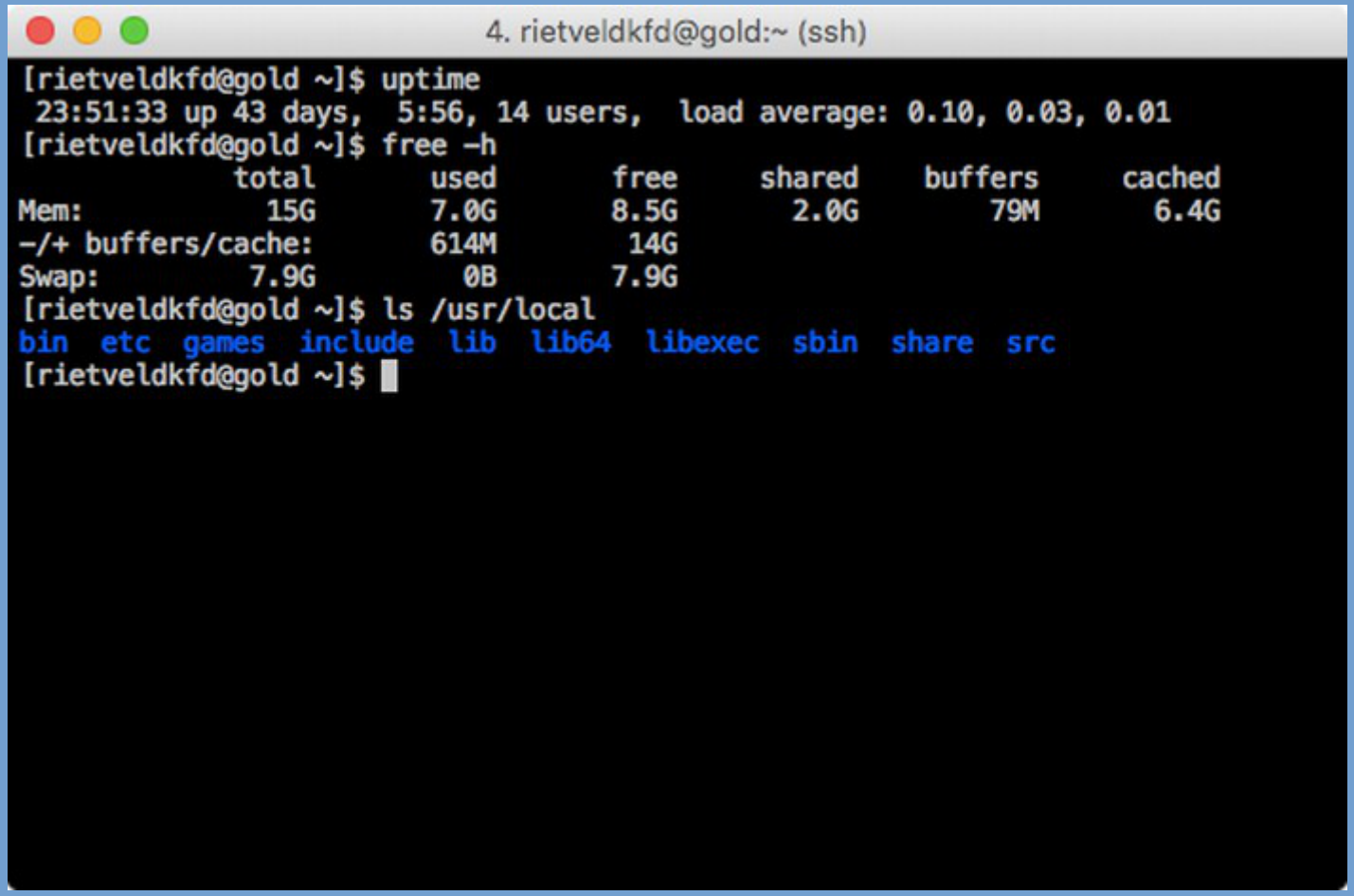


Source: Silberschatz et al., Operating System Concepts, 9th Edition

User Interfaces

- CLI: Command Line Interpreter
  - A Command Interpreter, known as a ***shell***, is provided.
  - Users can enter commands that the system will execute.
    - On UNIX, these commands often correspond with simple system programs installed on the system in /bin or /usr/bin.
    - In this case: additional “commands” can be added by installing programs into the appropriate directories.
    - Otherwise systems rely on commands built into the shell.
  - Through system calls, the kernel is told of programs that should be executed.
  - Many different shells exist and easy to write your own.

User Interfaces (2)



User Interfaces (3)

- Graphical User Interfaces typically based on “Desktop” metaphor.
  - Mouse is used to manipulate objects (icons) on the screen, laid out on a desktop.
  - Objects can represent files, programs, directories, links, etc.
  - Pioneered at Xerox PARC in the '70s. Steve Jobs came to visit. Macintosh user interface was based on this idea, released in '80s. Did not go unnoticed by Microsoft who developed and released Windows.
  - Modern macOS: Open Source XNU kernel with closed-source User Interface (formerly Aqua) on top.
  - For Linux many Desktop Environments exist: GNOME, KDE, Xfce, Mate, Enlightenment, TWM, FVWM, ...

Remote access

- Consider that remote access is possible both text-based and graphically.
  - ssh: “secure shell”, log in on a remote system to get access to a CLI running on that remote system.
  - RDP protocol: get a “remote desktop” on a remote Windows system. Microsoft Remote Desktop Client.

User Interfaces (4)

- Touchscreen interfaces are a further development of classic Graphical User Interfaces.
  - Instead of a mouse, a touchscreen is used for input.
  - Interactions with the system are different as a result (gestures).
  - Physical keyboard typically not present, on-screen keyboard is used.
  - More emphasis on appearance of system: icons, animations, design, etc.
  - New paradigms: augmented reality (AR), Face ID.

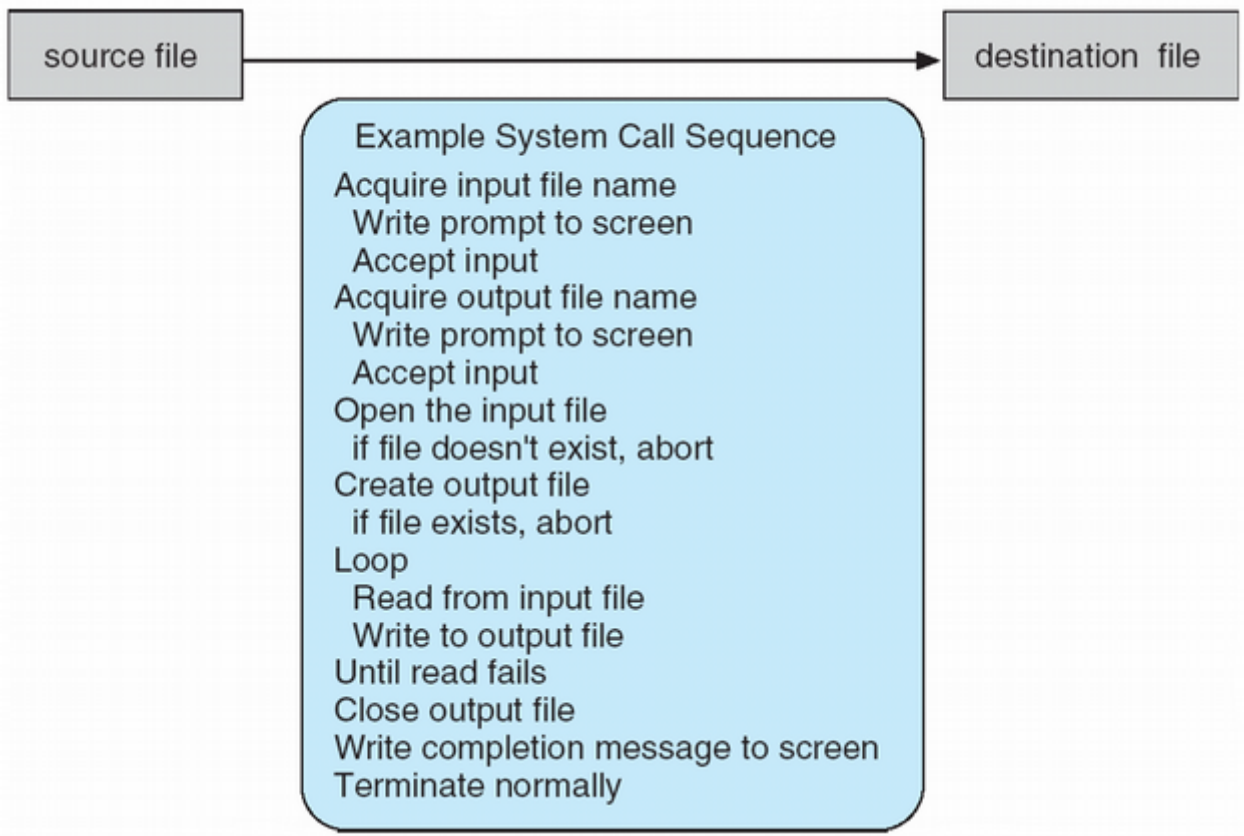
Source: Silberschatz et al., Operating System Concepts, 9th Edition

System Calls

- The system call interface forms the main proramming interface to the services provided by the Operating System kernel.
  - Barrier between user- and kernel-space.
- System calls are invoked using traps.
  - The trap mechanism differs per hardware platform.
  - Therefore, system calls are wrapped in, for instance, C-functions that are provided by the C library.
  - The function signatures of these system call functions in the library constitute the API (Application Programming Interface) of the OS.
- Common System Call APIs:
  - POSIX, Win32, Java VM (JVM).

System Calls (2)

- To get an idea of the granularity of system calls, consider the following example that copies a source file to a destination file:



Source: Silberschatz et al., Operating System Concepts, 9th Edition

System Calls (3)

- The system call API is thoroughly documented. On UNIX systems also available as “man pages”.

EXAMPLE OF STANDARD 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.

Source: Silberschatz et al., Operating System Concepts, 9th Edition

Implementing System Calls

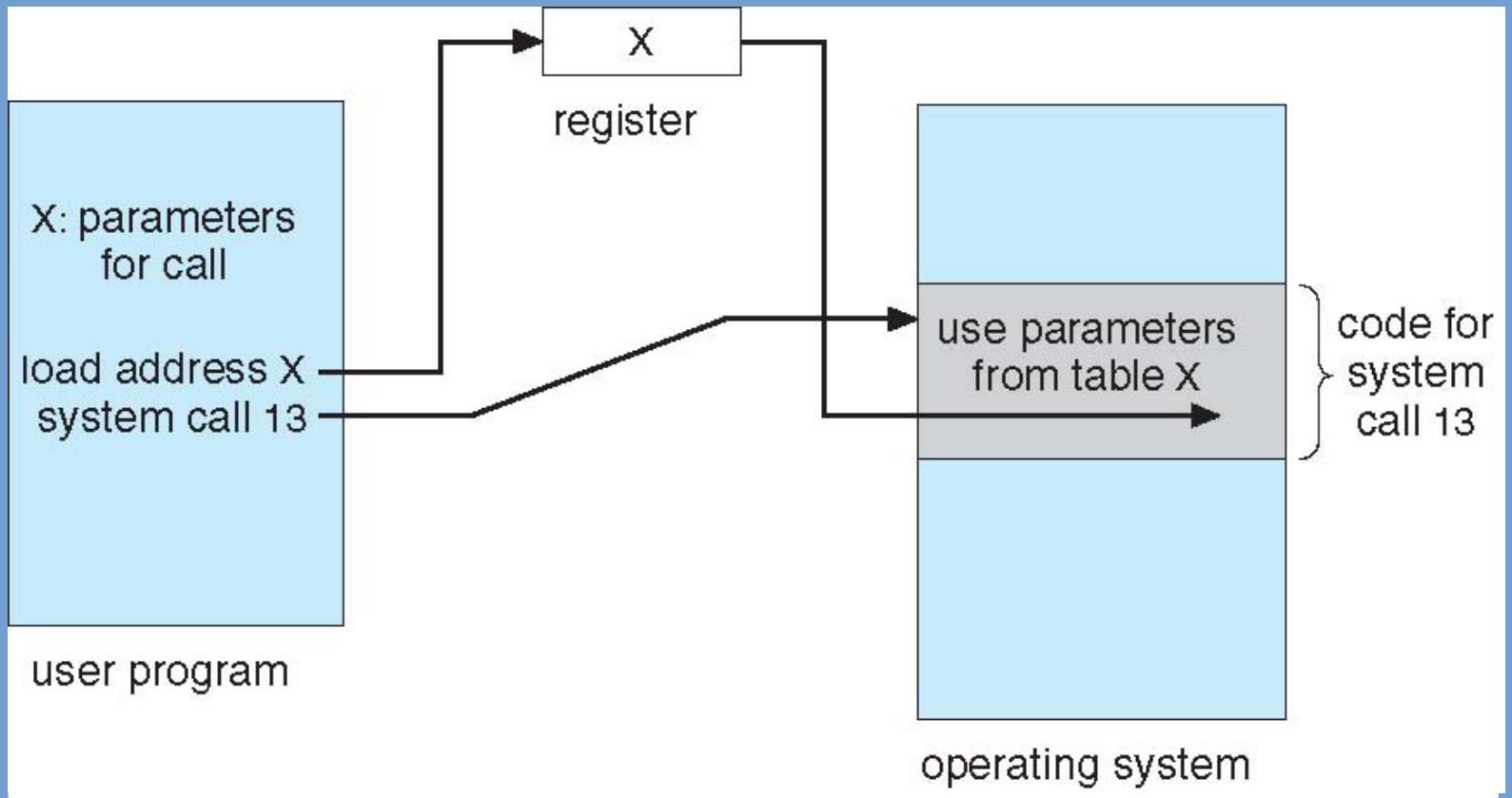
- To implement system calls, an interface needs to be defined that specifies how system calls are invoked in the kernel:
  - A trap mechanism is needed to trigger a mode-switch and handing over control to the kernel.
  - The requested call must be communicated. This is typically done by numbering system calls and passing a number.
    - This number is used to subscript an array of function pointers in the kernel (after checking this number is within bounds :)).
  - Arguments need to be passed to the system call function.
  - The return value of the system call should be communicated after completion.

Implementing System Calls (2)

- Argument passing can be done in several ways:
  - Through CPU registers
  - By placing these on the stack and popping when the system call returns.
  - By passing a pointer to a block of memory (or structure) containing the necessary parameters.
  - Or a combination thereof: for instance passing the first 4 parameters through registers and if the call has more parameters passing the remainder using the stack.

Implementing System Calls (3)

- Example when parameters are passed using a pointer to a block/table/structure:



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Implementing System Calls (3)

- This kernel interface to invoke system calls differs per hardware platform and OS implementation.
- Having everybody program against this interface directly is cumbersome for the programmers and requires rewrites for different OS platforms.
- Therefore, these details are typically hidden behind a system call API.
  - Programmers program against this API.
  - This API is implemented by a system library, such as the C library for POSIX.
  - To run the same code on a different platform, only a recompile needed against the different system library (which exposes the same API).

Implementing System Calls (4)

Examples: `syscall` function from uclibc-ng:

```
.globl syscall
.type   syscall,%function
syscall:
        movq %rdi, %rax         /* Syscall number -> rax.  */
        movq %rsi, %rdi         /* shift arg1 - arg5.  */
        movq %rdx, %rsi
        movq %rcx, %rdx
        movq %r8, %r10
        movq %r9, %r8
        movq 8(%rsp),%r9        /* arg6 is on the stack.  */
        syscall                 /* Do the system call.  */
        cmpq $-4095, %rax       /* Check %rax for error.  */
        jae __syscall_error             /* Branch forward if it failed.  */
        ret                     /* Return to caller.  */
```

```
.global syscall
.type   syscall,%function
syscall:
        pushl %ebp
        pushl %edi
        pushl %esi
        pushl %ebx

        movl  44(%esp),%ebp     /* Load the 6 syscall argument registers
        movl  40(%esp),%edi
        movl  36(%esp),%esi
        movl  32(%esp),%edx
        movl  28(%esp),%ecx
        movl  24(%esp),%ebx
        movl  20(%esp),%eax     /* Load syscall number into %eax.  */
        int $0x80

        popl %ebx
        popl %esi
        popl %edi
        popl %ebp

        cmpl $-4095,%eax
        jae  __syscall_error
        ret                     /* Return to caller.  */
```

```
long syscall(long sysnum, long a, long b, long c, long d, long e, long f)
{
#if !defined(__thumb__)
        register long _r0 __asm__("r0")=(long)(sysnum);
        register long _r6 __asm__("r6")=(long)(f);
        register long _r5 __asm__("r5")=(long)(e);
        register long _r4 __asm__("r4")=(long)(d);
        register long _r3 __asm__("r3")=(long)(c);
        register long _r2 __asm__("r2")=(long)(b);
        register long _r1 __asm__("r1")=(long)(a);
        __asm__ __volatile__(
                        "swi %1"
                        : "=r"(_r0)
                        : "i"(__NR_syscall), "r"(_r0), "r"(_r1),
                        "r"(_r2), "r"(_r3), "r"(_r4), "r"(_r5),
                        "r"(_r6)
                        : "memory");
```

Implementing System Calls (4)

Examples: `syscall` function from uclibc-ng:

x86_64

```
.globl syscall
.type   syscall,%function
syscall:
        movq %rdi, %rax         /* Syscall number -> rax.  */
        movq %rsi, %rdi         /* shift arg1 - arg5.  */
        movq %rdx, %rsi
        movq %rcx, %rdx
        movq %r8, %r10
        movq %r9, %r8
        movq 8(%rsp),%r9        /* arg6 is on the stack.  */
        syscall                 /* Do the system call.  */
        cmpq $-4095, %rax       /* Check %rax for error.  */
        jae __syscall_error             /* Branch forward if it failed.  */
        ret                     /* Return to caller.  */
```

i386

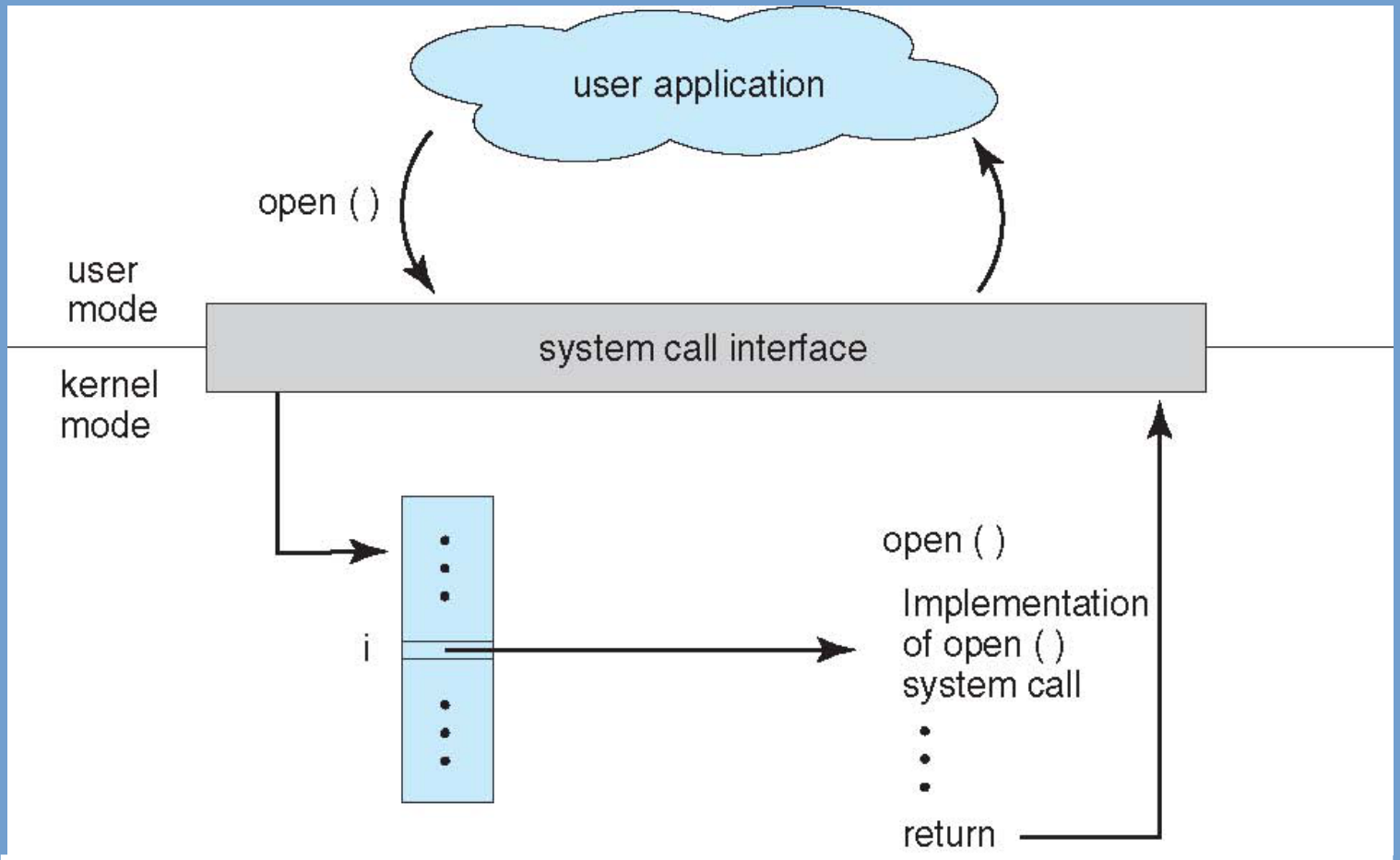
```
.global syscall
.type   syscall,%function
syscall:
        pushl %ebp
        pushl %edi
        pushl %esi
        pushl %ebx

        movl  44(%esp),%ebp     /* Load the 6 syscall argument registers
        movl  40(%esp),%edi
        movl  36(%esp),%esi
        movl  32(%esp),%edx
        movl  28(%esp),%ecx
        movl  24(%esp),%ebx
        movl  20(%esp),%eax     /* Load syscall number into %eax.  */
        int $0x80

        popl %ebx
        popl %esi
        popl %edi
        popl %ebp

        cmpl $-4095,%eax
        jae  __syscall_error
        ret                     /* Return to caller.  */
```

ARM

```
long syscall(long sysnum, long a, long b, long c, long d, long e, long f)
{
#if !defined(__thumb__)
        register long _r0 __asm__("r0")=(long)(sysnum);
        register long _r6 __asm__("r6")=(long)(f);
        register long _r5 __asm__("r5")=(long)(e);
        register long _r4 __asm__("r4")=(long)(d);
        register long _r3 __asm__("r3")=(long)(c);
        register long _r2 __asm__("r2")=(long)(b);
        register long _r1 __asm__("r1")=(long)(a);
        __asm__ __volatile__(
                        "swi %1"
                        : "=r"(_r0)
                        : "i"(__NR_syscall), "r"(_r0), "r"(_r1),
                        "r"(_r2), "r"(_r3), "r"(_r4), "r"(_r5),
                        "r"(_r6)
                        : "memory");
```

Implementing System Calls (5)

- Overview in a figure:



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Examples of System Calls

- We already saw an overview of the different components that are present in an operating system.
- We can distinguish system calls based on by which component they are implemented.
- Process Management
  - Create / terminate process
  - Load executable in memory
  - Suspend / resume
  - Send signals
  - Interprocess Communication (IPC) / Synchronization

Examples of System Calls (2)

- Memory management
  - Enlarge heap/stack within process
  - Allocate memory (anonymous mappings)
  - Map file into memory area
- File management
  - Open, close, read/write files
  - Create, remove, read directory
  - Change file/directory attributes
- Device management
  - Open devices, send special commands (e.g. ioctl).

Examples of System Calls (3)

- System information
  - Get / set time of day, system up time
  - Request system reboot / shutdown
  - Read kernel log buffer (dmesg)
- Communication
  - Between local processes, e.g. pipes.
  - Network communication (socket I/O)

Windows vs. UNIX system calls

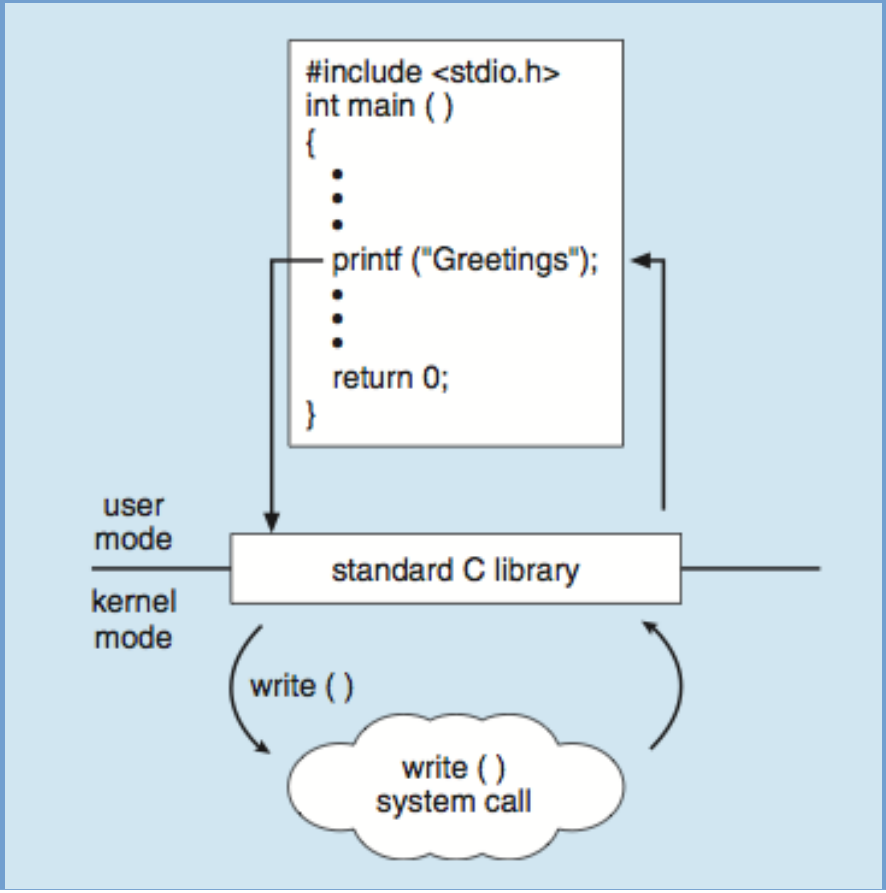
Same concept, different API

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Source: Silberschatz et al., Operating System Concepts, 9th Edition

Library calls vs. system calls

`printf` is a C library function, that calls a system call in its implementation.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

System Programs

- System programs vs. Application programs
  - A kernel is typically supported by a set of system programs.
  - A bare kernel is often not too useful.
  - Again: the exact boundary between system and application programs is not always clear.
- Some typical examples:
  - Tools for file system management: ls, mv, rm mkdir, touch, df, du
    - In several cases these tools are simple wrappers around the corresponding system call (e.g. mkdir).
  - Tools to acquire system information: ps, uptime, listing devices
  - Programs for system initialization and management (init, launchd, systemd), system settings & configuration.
  - Required background services (or subsystems, daemons) such as system logging, monitoring hotplug devices (e.g. USB).

System Boot

- System boot often relies on system programs.
- What happens on power on?
  - The CPU starts executing instructions from a pre-defined address
  - At this location some firmware is typically present, stored in a ROM or EEPROM.
  - Low-level initialization is performed and a boot loader is loaded from a specific location on secondary storage.
  - Depending on the system sometimes a more versatile second-stage boot loader is loaded. GRUB is the de-facto standard for x86 Linux systems.
  - The boot loader loads the selected kernel from secondary storage into memory and jumps to it.
  - Kernel performs necessary system initialization.
  - Once done, it starts an initial user-space program. This is usually a system program that performs further initialization of the system and launches other services.
  - The system is now running and waits for commands.

System Operation

- The kernel and many background services generate log files
  - Log of access to a service (access granted / denied)
  - Log of errors, from file not found to hard disk I/O errors
  - Etc…
- Sometimes software exhibits problems
  - When user processes crash, they can leave a ***core dump***. The current state of the process is saved to a file which can later be loaded into a debugger.
  - When the kernel crashes, some systems support the creation of a ***crash dump***.
- Typically tools are provided to analyze the performance of the system
  - Profiling of user applications
  - Monitoring I/O devices or collecting kernel trace points.

Operating System Implementation

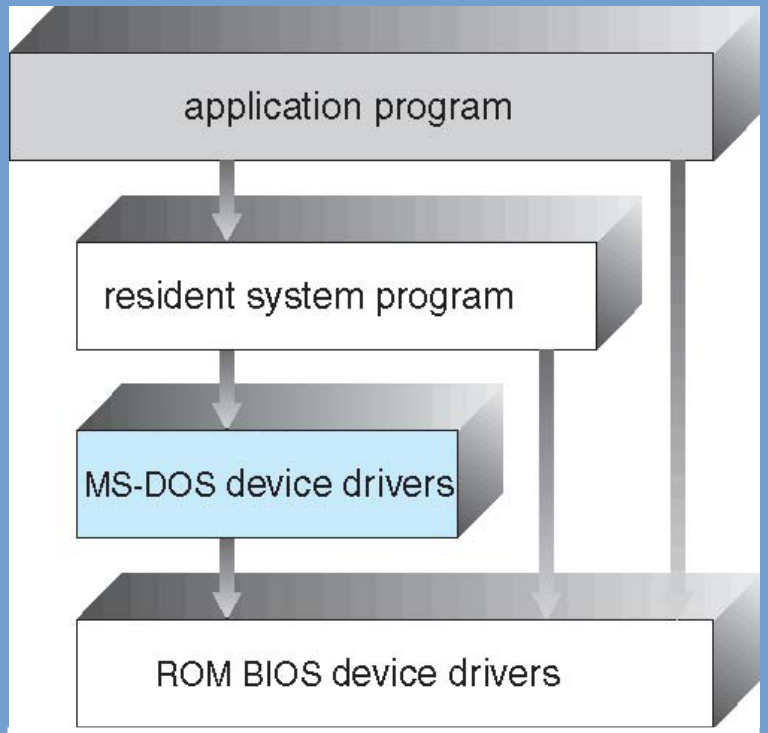
- So we want to write an OS, what programming language do we use?
  - Often a low-level language, because we need to interface with hardware and devices.
  - Some assembly required, depending on architecture.
    - Past: everything would need to be written in assembly.
  - An OS can also be written in a subset of C++, it is low-level enough.
  - System programs: any language from which system calls can be invoked. Typically C, C++, Perl, Python.
- Important for choice of language: the more is written in a higher level language, the easier it is to port the OS to another hardware platform / architecture.
  - 100% assembly code required a 100% rewrite.

Operating System Implementation (2)

- Another important decision is how to structure the implementation.
- In the past this was not of much importance.
- Todays complex systems required a clear structure to be followed.
  - Systems otherwise hard to maintain / debug.
  - Hard to decouple responsibilities.
  - Hard to extend in the future.
- We now review a number of such structures.

DOS: Simple structure

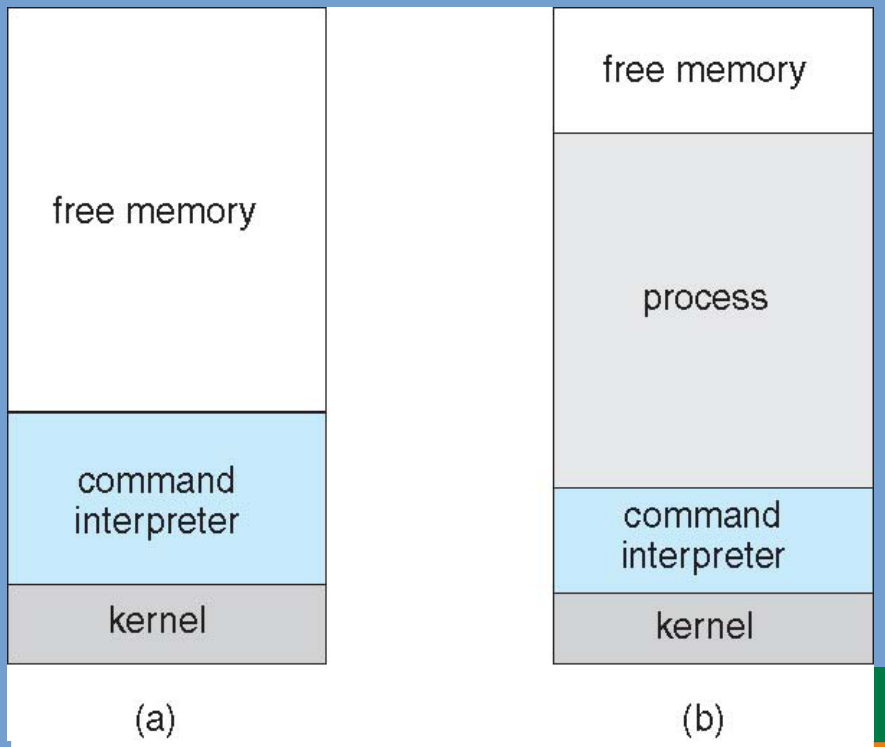
- An early OS, designed for personal computers (PCs) without much available resources.
- Minimal kernel in memory, extended with drivers and extension routines.
  - Extension routines installed themselves in the background; TSR: Terminate and Stay Resident.
  - Programming interface with software interrupts.
  - Ralf Brown Interrupt List.
- No real multitasking. Could not take advantage of modern architectures.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

DOS: Simple structure (2)

- COMMAND.COM: main command interpreter.
- Only a single program could run at a time. On program startup, COMMAND.COM was for a large part unloaded.
- Needed to be reloaded from disk on program exit. Screeching floppy sound when exiting a game (for example).
- No process isolation or memory protection, there was a single memory space that everybody could access.



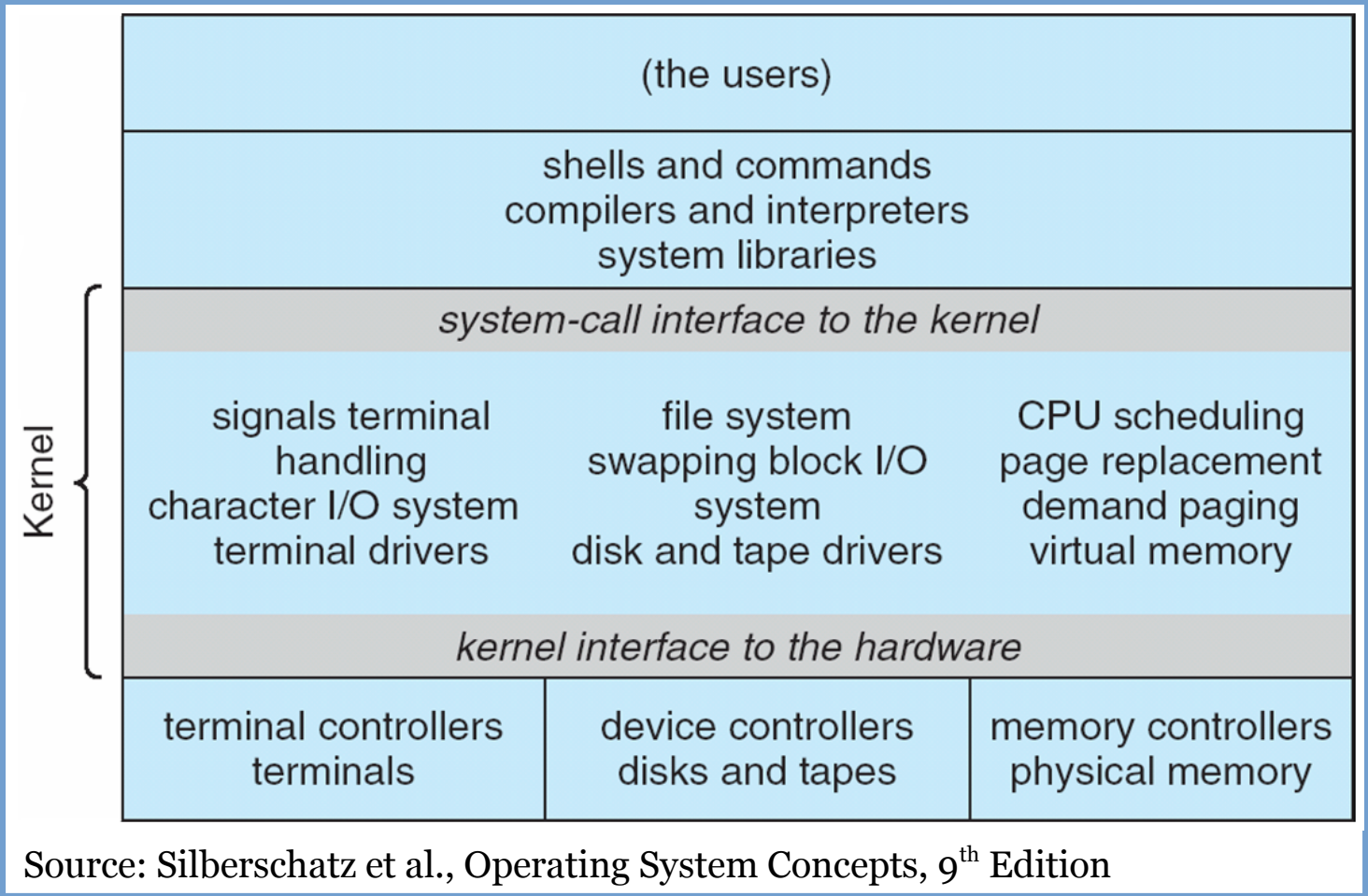
Source: Silberschatz et al., Operating System Concepts, 9th Edition

Monolithic kernels

- Most UNIX systems (but certainly not all) are based on a monolithic kernel.
  - A single binary image that contains all kernel functionality. This is always present in memory.
  - Runs in a single address space. Different components within this image are not isolated from each other.
  - Much clearer system call interface compared to DOS.
  - With good programming practices, the source code for this single binary image can be well structured, with clear interfaces between different components.
    - It is then up to the programmers to not violate these interfaces … The system does not protect against this.
- A UNIX kernel relies heavily on system programs to control the system and make the system convenient to use.

Monolithic kernels (2)

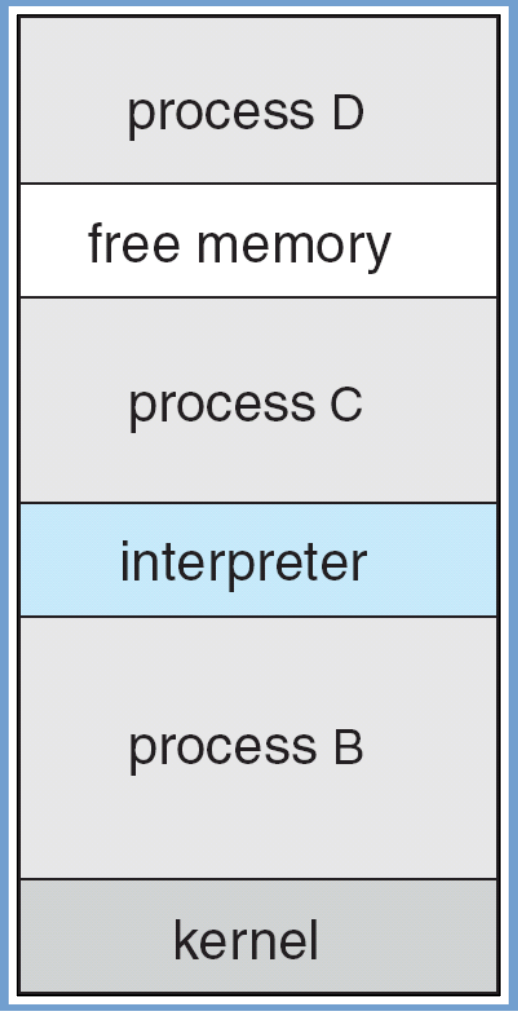
- Example of UNIX system structure:



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Monolithic kernels (3)

- Example memory layout on the right.
- Multi-tasking system.
- A shell is an ordinary user process run on behalf of the users.
- System calls present to create new processes (fork) and load new executable images (execv).
  - Used by the shell to start programs requested by the user.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Monolithic kernels (4)

- Such a single binary image does not seem extensible.
- How to add new drivers, file system implementations, schedulers, etc?
  - Recompile the entire kernel and reboot.
  - Cumbersome. Therefore many modern monolithic kernels support ***loadable kernel modules***.
- Kernel modules …
  - … are written against interfaces exposed by the kernel
  - … can be loaded and unloaded at run-time (without reboots). In fact, the single binary image is “extended” to contain this module.

Layered approach

- The monolithic approach that we have just seen can also be strictly layered.
  - In some cases requiring a trap instruction to call functions across layers.
- Each layer is constructed based on the layer below it.
- First system built in this way: THE system, TH Eindhoven, 1968.

Source: Operating Systems: Design and implementation, 2nd Edition. Tanenbaum and Woodhull. Sect. 1.5.2

Layered approach (2)

- Example layer structure:

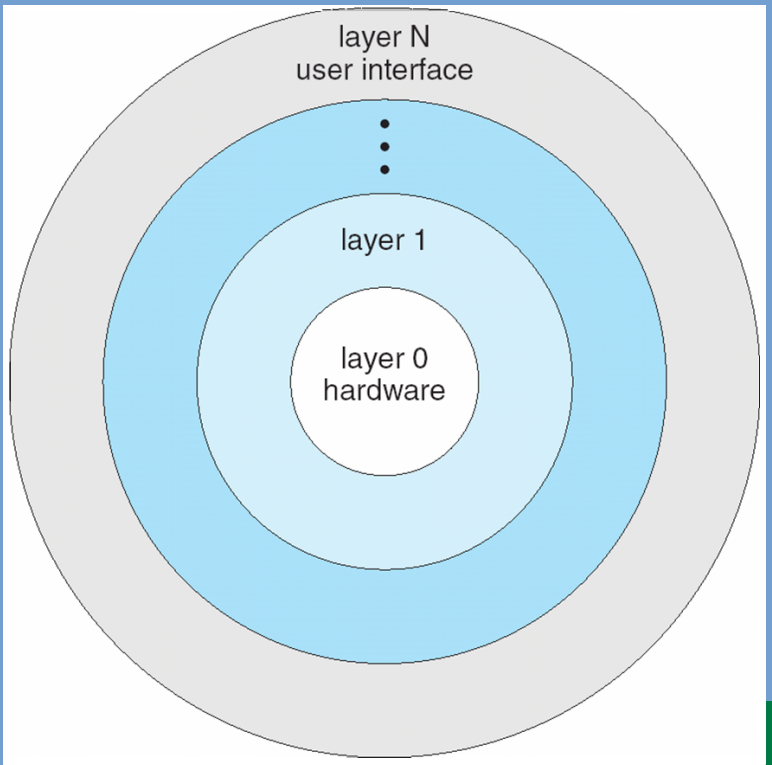
Layer	Function
5	The Operator
4	User programs
3	I/O management
2	Operator-process communication
1	Memory management
0	Processor allocation and scheduling

- Finally linked into a single object, so only a design aid!

Source: Operating Systems: Design and implementation, 2nd Edition. Tanenbaum and Woodhull. Sect. 1.5.2

Layered approach (3)

- In MULTICS concentric rings were used instead of layers.
- Inner rings were more privileged & protected
- Only accessible using a trap instruction, so protection enforced by hardware.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

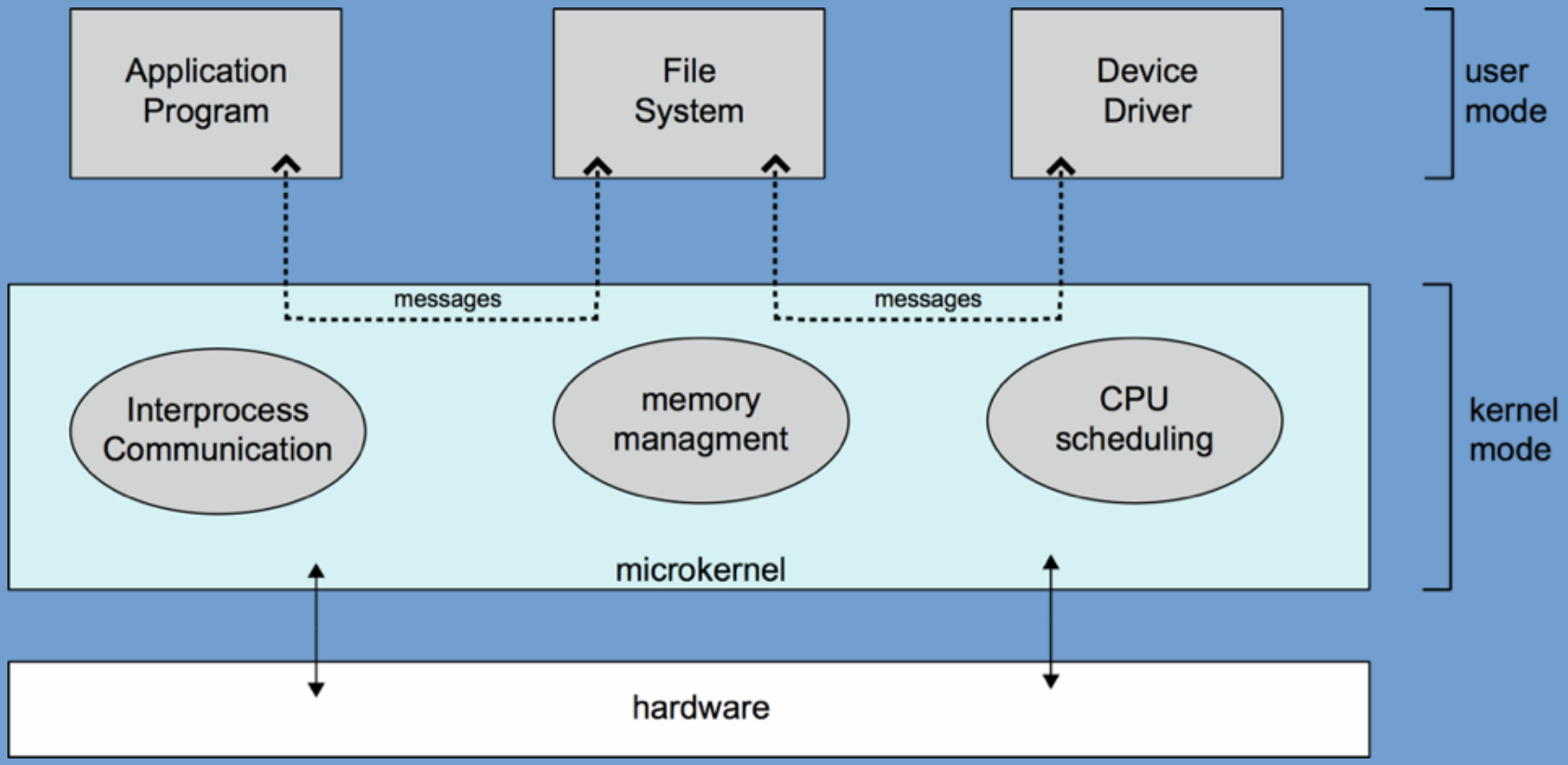
Source: Operating Systems: Design and implementation, 2nd Edition. Tanenbaum and Woodhull. Sect. 1.5.2

Micro kernels

- The main idea behind a micro kernel is to make the kernel as small as possible and move as much components as possible to user-space.
  - So only these parts that really must run in kernel-space are present in kernel space.
- The micro kernel supports for instance:
  - Communication between the various components, often through message passing.
  - Setting up CPU registers required for enforcing protection & isolation.
  - Performing context switches.
  - Receiving interrupts and forwarding these to the responsible component.
- Components that would be implemented in user space include these for process management, memory management, file management and of course device drivers.

Micro kernels (2)

- What this looks like schematically:



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Micro kernels (3)

- Advantages:
  - Easier to extend, the micro kernel does typically not need modification and a user-space module can be written and started.
  - More reliable, a crash in a user-space kernel component does not bring down the entire system.
    - Compare with monolithic kernel that runs in a single address space.
  - Easier to port to other architectures, since this only has to be done for the micro kernel and device drivers.
- Disadvantages:
  - Performance: overhead due to message passing between different components.
  - Overhead due to context switches between different components to service a system call.

Micro kernels (4)

- Examples of systems based on the micro kernel design:
  - Minix
  - macOS, its XNU kernel is based on Mach (from CMU).
  - L4
  - QNX
  - Windows NT in its early days (next slide)

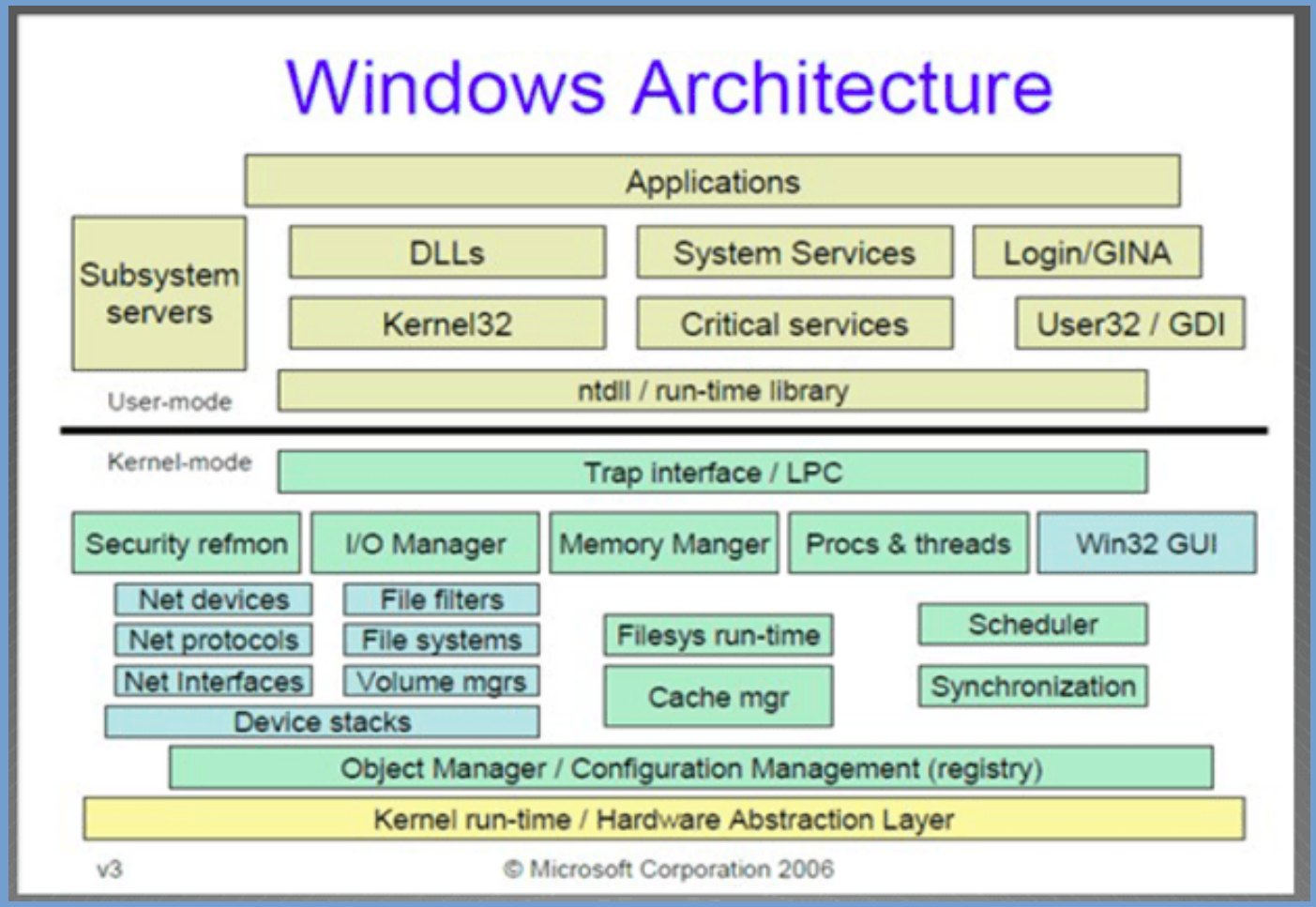
Hybrid Systems

- Systems do not always strictly adhere to a single model as just discussed.
  - For instance, due to changing requirements during the development process, to address performance or security concerns.
- Example: Windows NT was originally designed as a pure micro kernel.
  - However, performance shortfalls caused MicroSoft to move many of the subsystem services from user-space to kernel-space.
- Linux, Solaris, FreeBSD are all monolithic kernels, but can also be considered modular because they support loadable kernel modules.

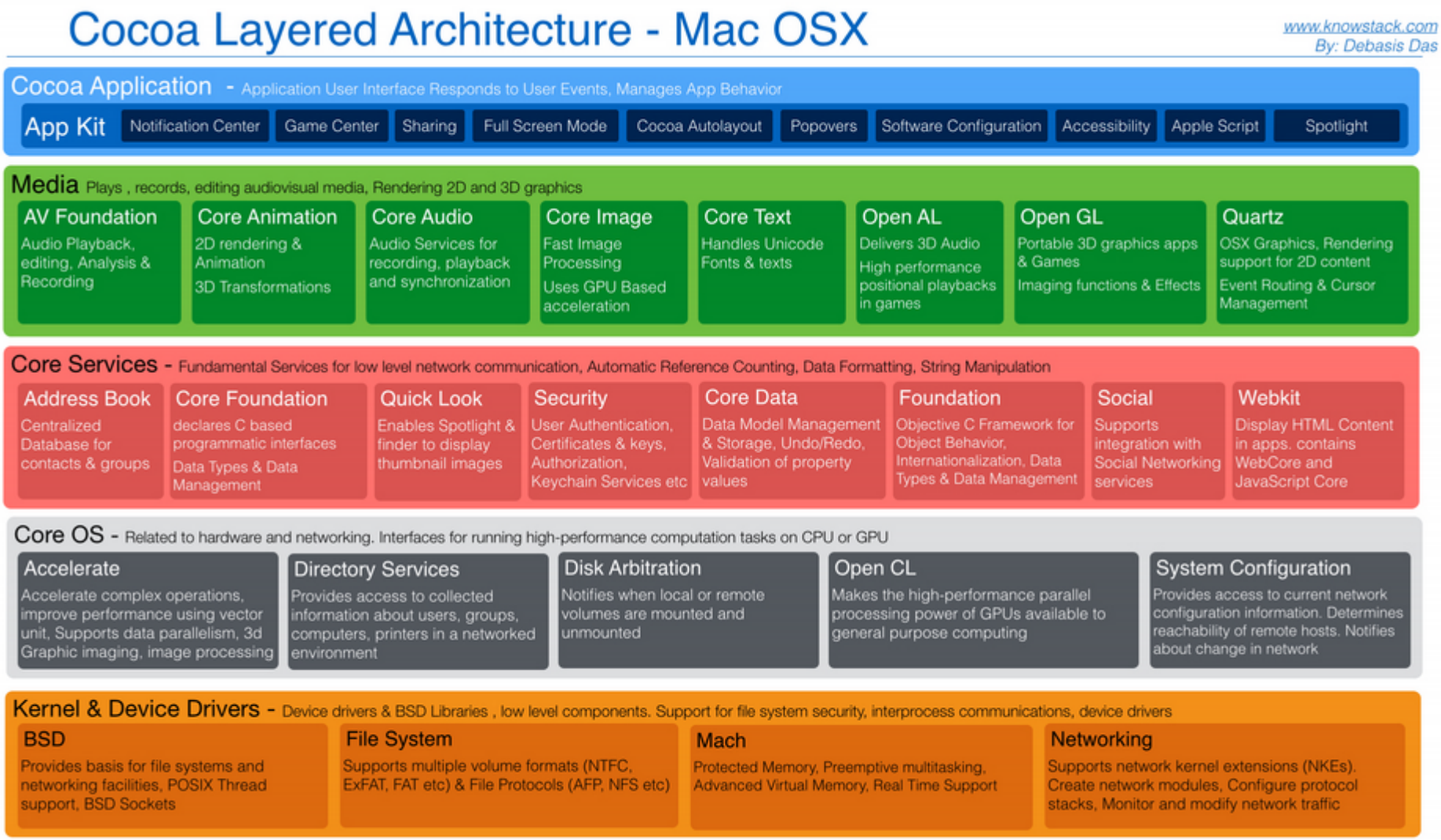
Full System Structures

- Modern Operating Systems thus consist of:
  - An OS kernel with system call interface.
  - A system library around this system call interface.
  - Various support libraries for the system, for instance to provide graphical user interfaces (GUIs).
  - System programs.
  - Sometimes a graphical shell.
- To conclude the chapter, let's have a quick look at some examples of full system structures.

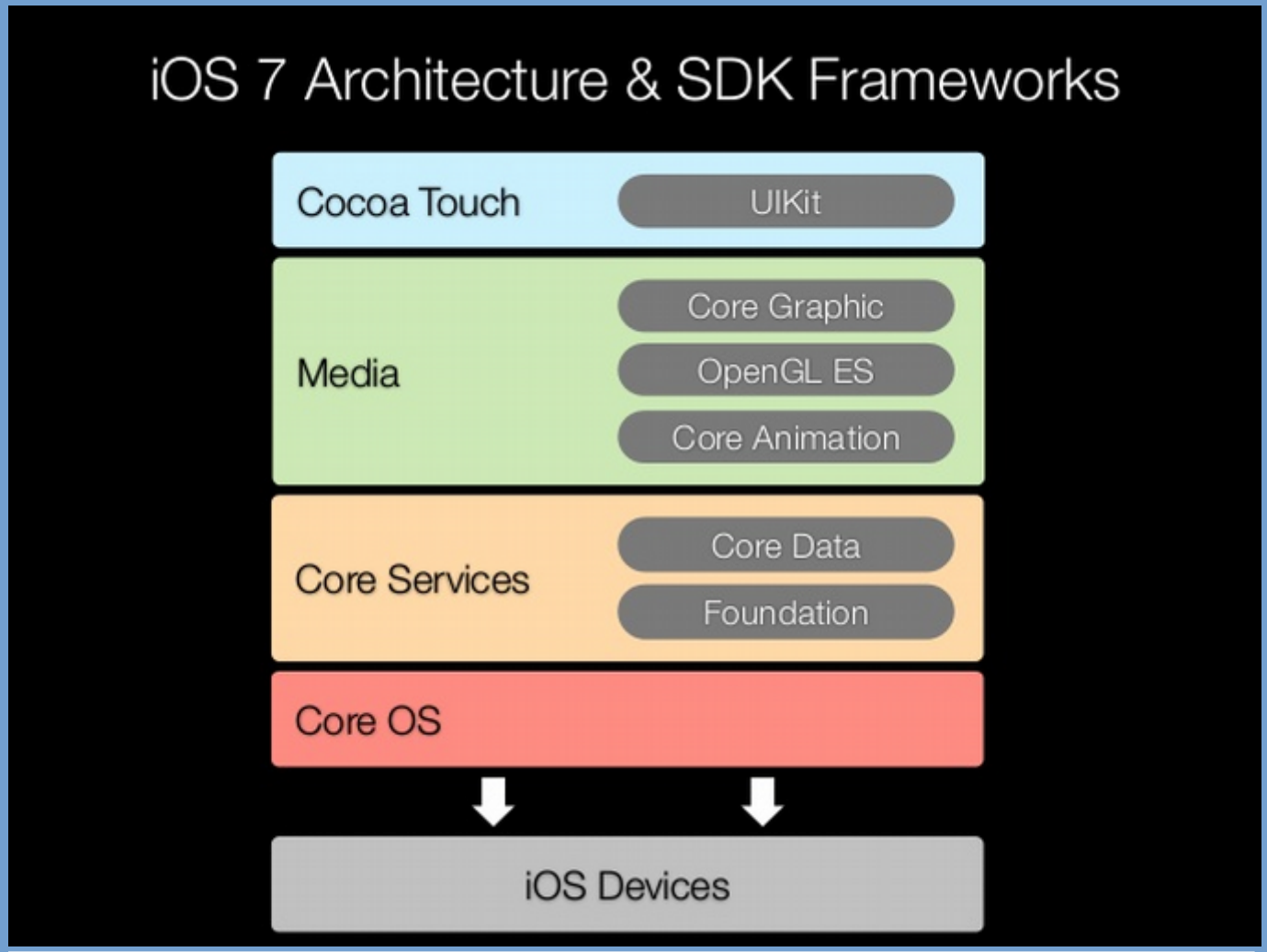
Windows NT architecture



macOS Architecture

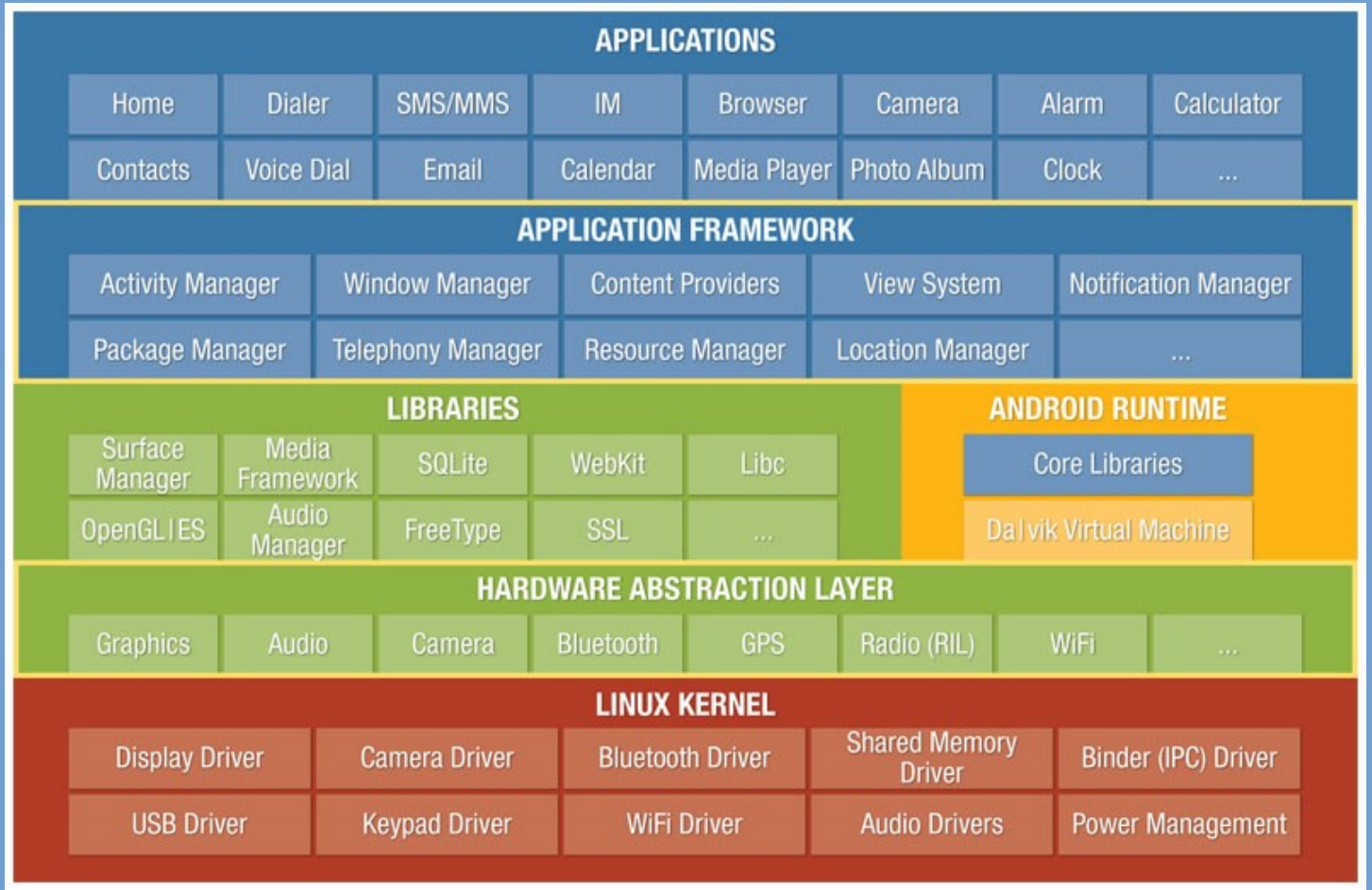


iOS architecture



Source: https://www.slideshare.net/vutlam9083/session-1-introduction-to-i-os-7-and-sdk

Android Architecture



Source: Manifest Security (https://manifestsecurity.com/android-application-security-part-2/)

End of Chapter 2.