Sistemi Operativi

Corso di Laurea in Informatica

a.a. 2020-2021

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

Recap from Last Lecture

- Operating System exposes a set of services to users/applications
  - The (efficient) implementation of those services often depends on the underlying HW architecture

Recap from Last Lecture

- Operating System exposes a set of services to users/applications
  - The (efficient) implementation of those services often depends on the underlying HW architecture
- In particular, we have have talked about:
  - Protection → kernel vs. user mode, base and limit registers

Recap from Last Lecture

- Operating System exposes a set of services to users/applications
  - The (efficient) implementation of those services often depends on the underlying HW architecture
- In particular, we have have talked about:
  - Protection → kernel vs. user mode, base and limit registers
  - Delegation → system calls to allow applications executing privileged tasks

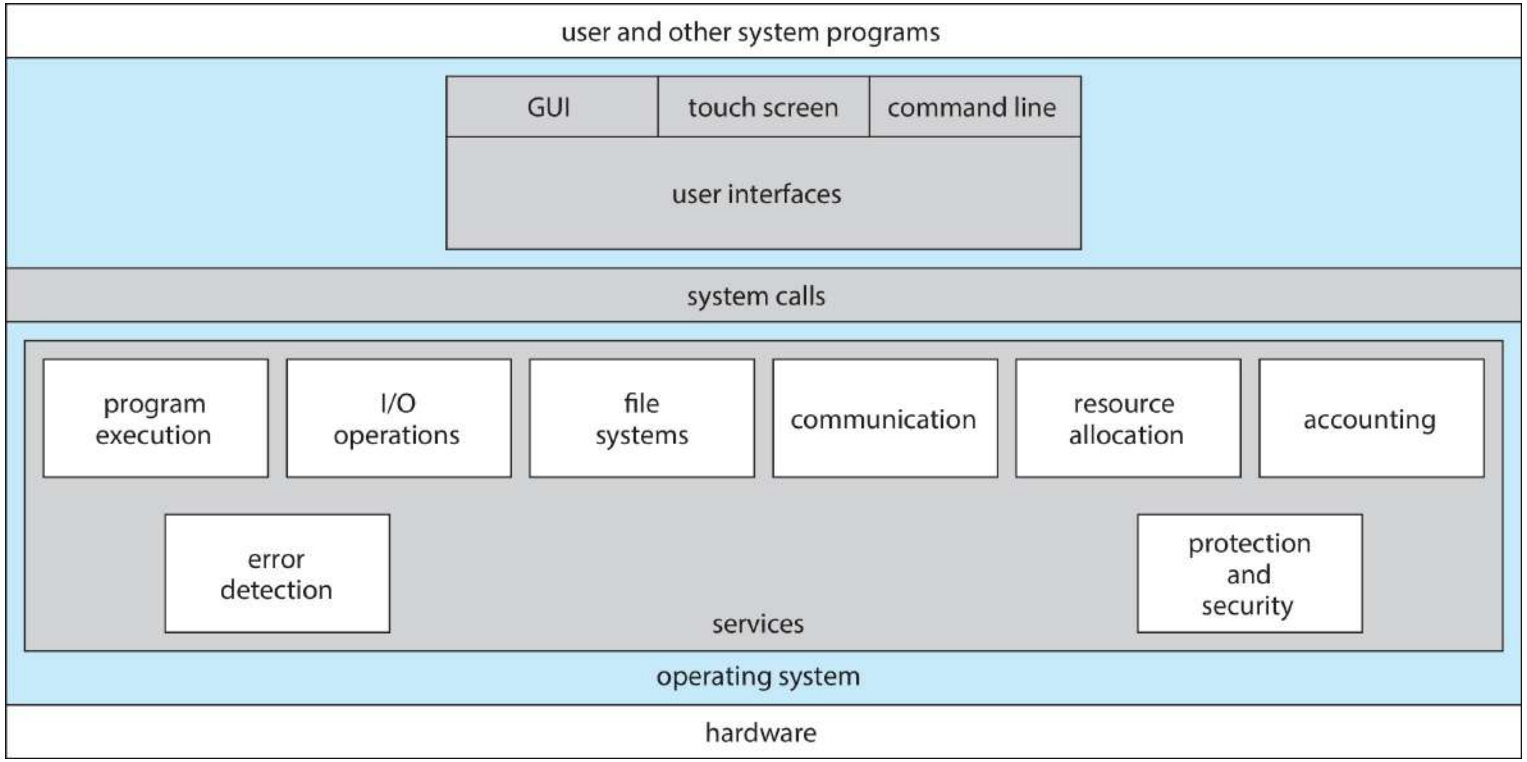
Recap from Last Lecture

- Operating System exposes a set of services to users/applications
  - The (efficient) implementation of those services often depends on the underlying HW architecture
- In particular, we have have talked about:
  - Protection → kernel vs. user mode, base and limit registers
  - Delegation → system calls to allow applications executing privileged tasks
  - Synchronization/Scheduling → traps and interrupt vector table

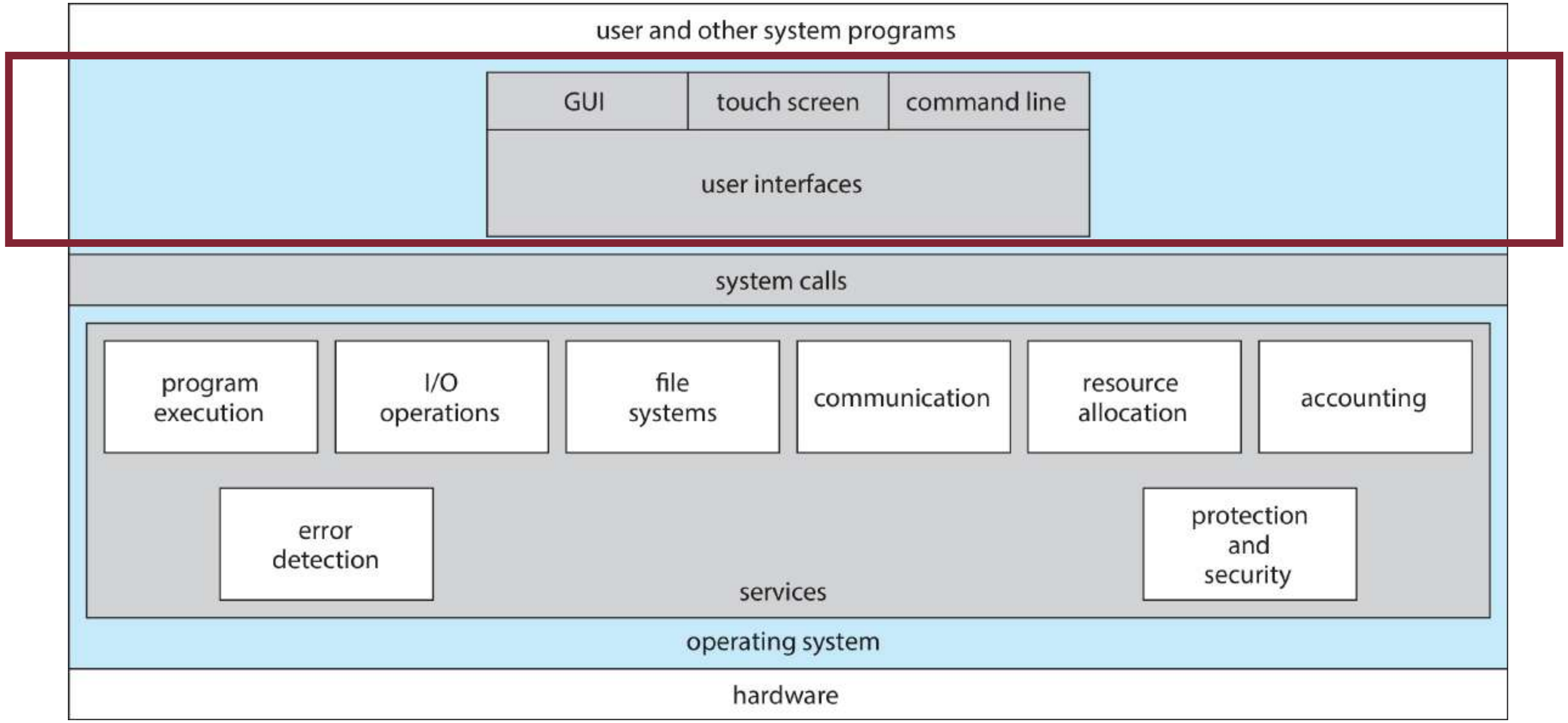
Recap from Last Lecture

- Operating System exposes a set of services to users/applications
    - The (efficient) implementation of those services often depends on the underlying HW architecture
- In particular, we have have talked about:
    - Protection → kernel vs. user mode, base and limit registers
    - Delegation → system calls to allow applications executing privileged tasks
    - Synchronization/Scheduling → traps and interrupt vector table
    - Virtualization → virtual memory and virtual address space

Modern OS Functionalities



User to OS Interface



User-OS Interface: CLI

- Command Line Interpreter (CLI) allows direct command entry
  - Sometimes implemented in kernel, sometimes by system programs
  - Multiple implementation: e.g., Bourne shell (`sh/bash`), Z shell (`zsh`), etc.

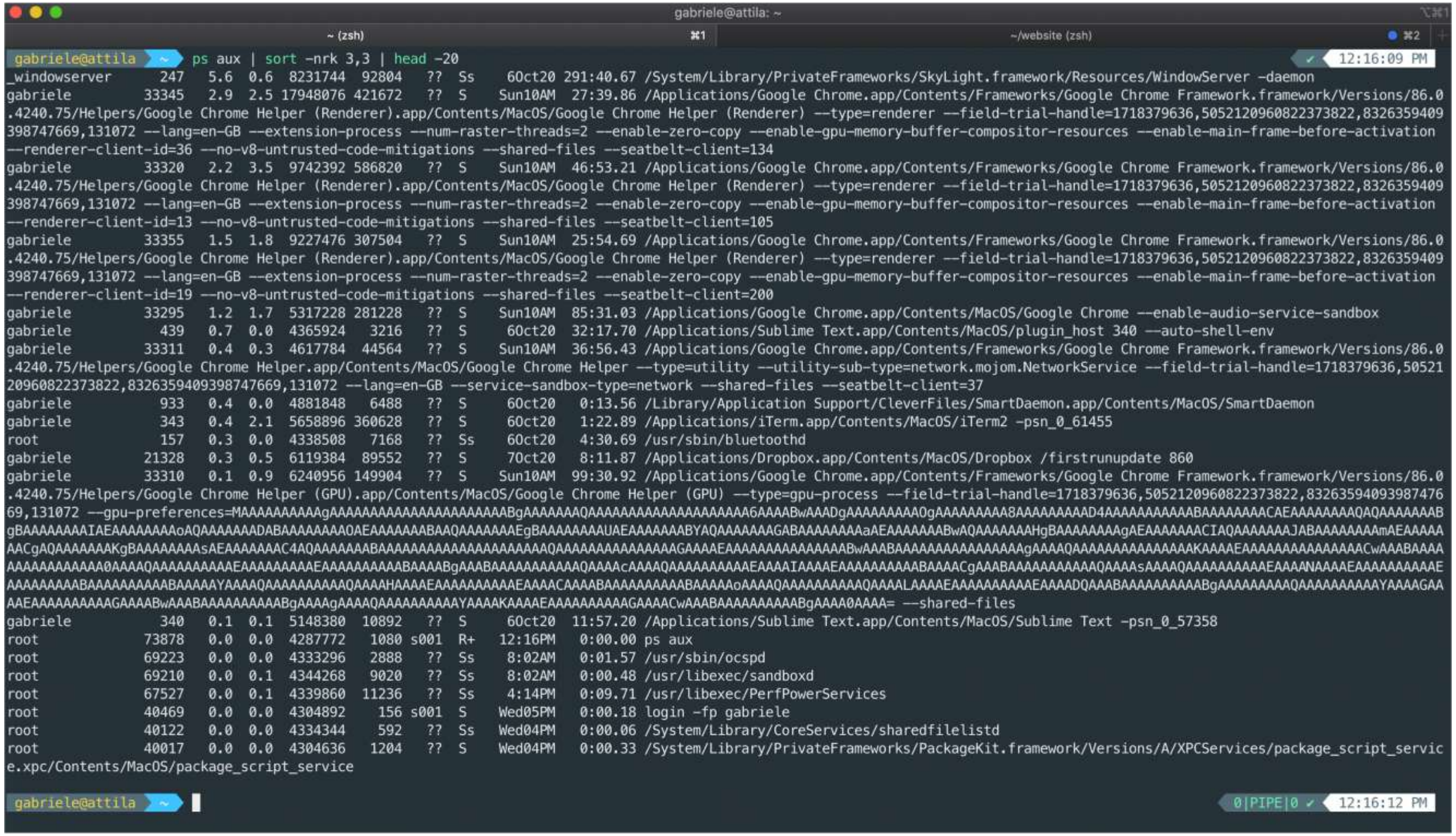
User-OS Interface: CLI

- Command Line Interpreter (CLI) allows direct command entry
  - Sometimes implemented in kernel, sometimes by system programs
  - Multiple implementation: e.g., Bourne shell (`sh/bash`), Z shell (`zsh`), etc.
- Fetches a command from user prompt and executes it

User-OS Interface: CLI

- Command Line Interpreter (CLI) allows direct command entry
  - Sometimes implemented in kernel, sometimes by system programs
  - Multiple implementation: e.g., Bourne shell (`sh`/`bash`), Z shell (`zsh`), etc.
- Fetches a command from user prompt and executes it
- Some commands are built-in, some others are just names of programs

Z Shell (**zsh**)


gabriele@attila: ~
~ (zsh)
⌘1
~/website (zsh)
⌘2
gabriele@attila ~ ps aux | sort -nrk 3,3 | head -20
12:16:09 PM
_windowserver 247 5.6 0.6 8231744 92804 ?? Ss 6Oct20 291:40.67 /System/Library/PrivateFrameworks/SkyLight.framework/Resources/WindowServer -daemon
gabriele 33345 2.9 2.5 17948076 421672 ?? S Sun10AM 27:39.86 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/86.0.4240.75/Helpers/Google Chrome Helper (Renderer).app/Contents/MacOS/Google Chrome Helper (Renderer) --type=renderer --field-trial-handle=1718379636,5052120960822373822,8326359409398747669,131072 --lang=en-GB --extension-process --num-raster-threads=2 --enable-zero-copy --enable-gpu-memory-buffer-compositor-resources --enable-main-frame-before-activation --renderer-client-id=36 --no-v8-untrusted-code-mitigations --shared-files --seatbelt-client=134
gabriele 33320 2.2 3.5 9742392 586820 ?? S Sun10AM 46:53.21 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/86.0.4240.75/Helpers/Google Chrome Helper (Renderer).app/Contents/MacOS/Google Chrome Helper (Renderer) --type=renderer --field-trial-handle=1718379636,5052120960822373822,8326359409398747669,131072 --lang=en-GB --extension-process --num-raster-threads=2 --enable-zero-copy --enable-gpu-memory-buffer-compositor-resources --enable-main-frame-before-activation --renderer-client-id=13 --no-v8-untrusted-code-mitigations --shared-files --seatbelt-client=105
gabriele 33355 1.5 1.8 9227476 307504 ?? S Sun10AM 25:54.69 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/86.0.4240.75/Helpers/Google Chrome Helper (Renderer).app/Contents/MacOS/Google Chrome Helper (Renderer) --type=renderer --field-trial-handle=1718379636,5052120960822373822,8326359409398747669,131072 --lang=en-GB --extension-process --num-raster-threads=2 --enable-zero-copy --enable-gpu-memory-buffer-compositor-resources --enable-main-frame-before-activation --renderer-client-id=19 --no-v8-untrusted-code-mitigations --shared-files --seatbelt-client=200
gabriele 33295 1.2 1.7 5317228 281228 ?? S Sun10AM 85:31.03 /Applications/Google Chrome.app/Contents/MacOS/Google Chrome --enable-audio-service-sandbox
gabriele 439 0.7 0.0 4365924 3216 ?? S 6Oct20 32:17.70 /Applications/Sublime Text.app/Contents/MacOS/plugin_host 340 --auto-shell-env
gabriele 33311 0.4 0.3 4617784 44564 ?? S Sun10AM 36:56.43 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/86.0.4240.75/Helpers/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --type=utility --utility-sub-type=network.mojom.NetworkService --field-trial-handle=1718379636,5052120960822373822,8326359409398747669,131072 --lang=en-GB --service-sandbox-type=network --shared-files --seatbelt-client=37
gabriele 933 0.4 0.0 4881848 6488 ?? S 6Oct20 0:13.56 /Library/Application Support/CleverFiles/SmartDaemon.app/Contents/MacOS/SmartDaemon
gabriele 343 0.4 2.1 5658896 360628 ?? S 6Oct20 1:22.89 /Applications/iTerm.app/Contents/MacOS/iTerm2 -psn_0_61455
root 157 0.3 0.0 4338508 7168 ?? Ss 6Oct20 4:30.69 /usr/sbin/bluetoothd
gabriele 21328 0.3 0.5 6119384 89552 ?? S 7Oct20 8:11.87 /Applications/Dropbox.app/Contents/MacOS/Dropbox /firstrunupdate 860
gabriele 33310 0.1 0.9 6240956 149904 ?? S Sun10AM 99:30.92 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/86.0.4240.75/Helpers/Google Chrome Helper (GPU).app/Contents/MacOS/Google Chrome Helper (GPU) --type=gpu-process --field-trial-handle=1718379636,5052120960822373822,8326359409398747669,131072 --gpu-preferences=MAAAAAAAAAAgAAAAAAAAAAAAAAAAAAAAABgAAAAAAQAAAAAAAAAAAAAAAAAAA6AAAABwAAADgAAAAAAAAAOgAAAAAAAAA8AAAAAAAAAD4AAAAAAAAAAABAAAAAAAACAEAAAAAAAAQAQAAAAAAABgBAAAAAAAAIAEAAAAAAAAoAQAAAAAAADABAAAAAAAAOAEAAAAAAABAAQAAAAAAAEgBAAAAAAAAUAEAAAAAAABYAQAAAAAAAGABAAAAAAAAaAEAAAAAAABwAQAAAAAAAHgBAAAAAAAAgAEAAAAAAACIAQAAAAAAAJABAAAAAAAAmAEAAAAAAACgAQAAAAAAAKgBAAAAAAAAsAEAAAAAAAC4AQAAAAAAABAAAAAAAAAAAAAAAAAAAAAQAAAAAAAAAAAAAAAGAAAAEAAAAAAAAAAAAAAABwAAABAAAAAAAAAAAAAAAAgAAAAQAAAAAAAAAAAAAAAKAAAAEAAAAAAAAAAAAAAACwAAABAAAAAAAAAAAAAA0AAAAQAAAAAAAAAAEAAAAAAAAAAEAAAAAAAAAAABAAAABgAAABAAAAAAAAAAAQAAAAcAAAAQAAAAAAAAAAEAAAAIAAAAEAAAAAAAAAAABAAAACgAAABAAAAAAAAAAAQAAAAsAAAAQAAAAAAAAAAEAAAANAAAAEAAAAAAAAAAEAAAAAAAAAABAAAAAAAAAABAAAAAYAAAAQAAAAAAAAAAQAAAAHAAAAEAAAAAAAAAAEAAAACAAAABAAAAAAAAAAABAAAAAoAAAAQAAAAAAAAAAQAAAALAAAAEAAAAAAAAAAEAAAADQAAABAAAAAAAAAABgAAAAAAAAQAAAAAAAAAAAYAAAAGAAAEAAAAAAAAAAAGAAAABwAAABAAAAAAAAAABgAAAAgAAAAQAAAAAAAAAAYAAAAKAAAAEAAAAAAAAAAAGAAAACwAAABAAAAAAAAAABgAAAA0AAAA= --shared-files
gabriele 340 0.1 0.1 5148380 10892 ?? S 6Oct20 11:57.20 /Applications/Sublime Text.app/Contents/MacOS/Sublime Text -psn_0_57358
root 73878 0.0 0.0 4287772 1080 s001 R+ 12:16PM 0:00.00 ps aux
root 69223 0.0 0.0 4333296 2888 ?? Ss 8:02AM 0:01.57 /usr/sbin/ocspd
root 69210 0.0 0.1 4344268 9020 ?? Ss 8:02AM 0:00.48 /usr/libexec/sandboxd
root 67527 0.0 0.1 4339860 11236 ?? Ss 4:14PM 0:09.71 /usr/libexec/PerfPowerServices
root 40469 0.0 0.0 4304892 156 s001 S Wed05PM 0:00.18 login -fp gabriele
root 40122 0.0 0.0 4334344 592 ?? Ss Wed04PM 0:00.06 /System/Library/CoreServices/sharedfilelistd
root 40017 0.0 0.0 4304636 1204 ?? S Wed04PM 0:00.33 /System/Library/PrivateFrameworks/PackageKit.framework/Versions/A/XPCServices/package_script_service.xpc/Contents/MacOS/package_script_service
gabriele@attila ~
0|PIPE|0
12:16:12 PM

User-OS Interface: GUI

- Graphical User Interface (GUI) desktop metaphore
  - Invented at Xerox PARC in the early 1970's
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc.

User-OS Interface: GUI

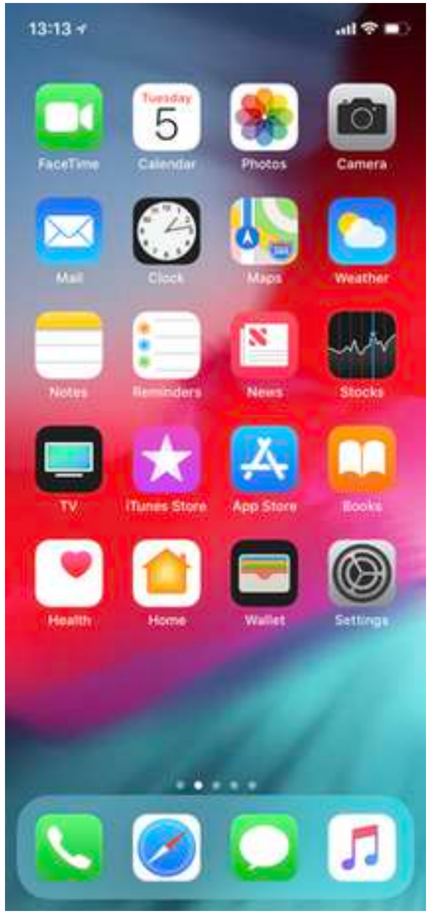
- Graphical User Interface (GUI) desktop metaphore
  - Invented at Xerox PARC in the early 1970's
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc.
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple macOS has "Aqua" GUI interface with UNIX kernel underneath and shells available
  - UNIX and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

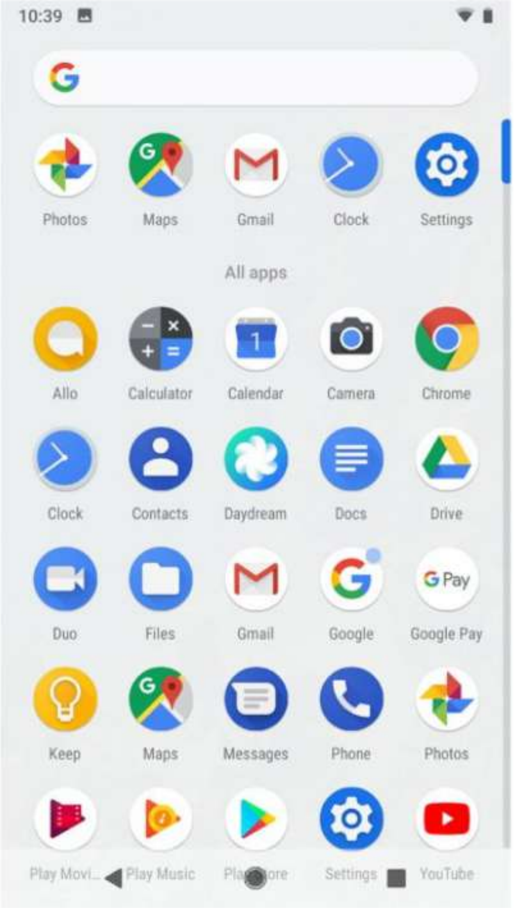
macOS Mojave "Aqua" GUI

User-OS Interface: Touchscreen

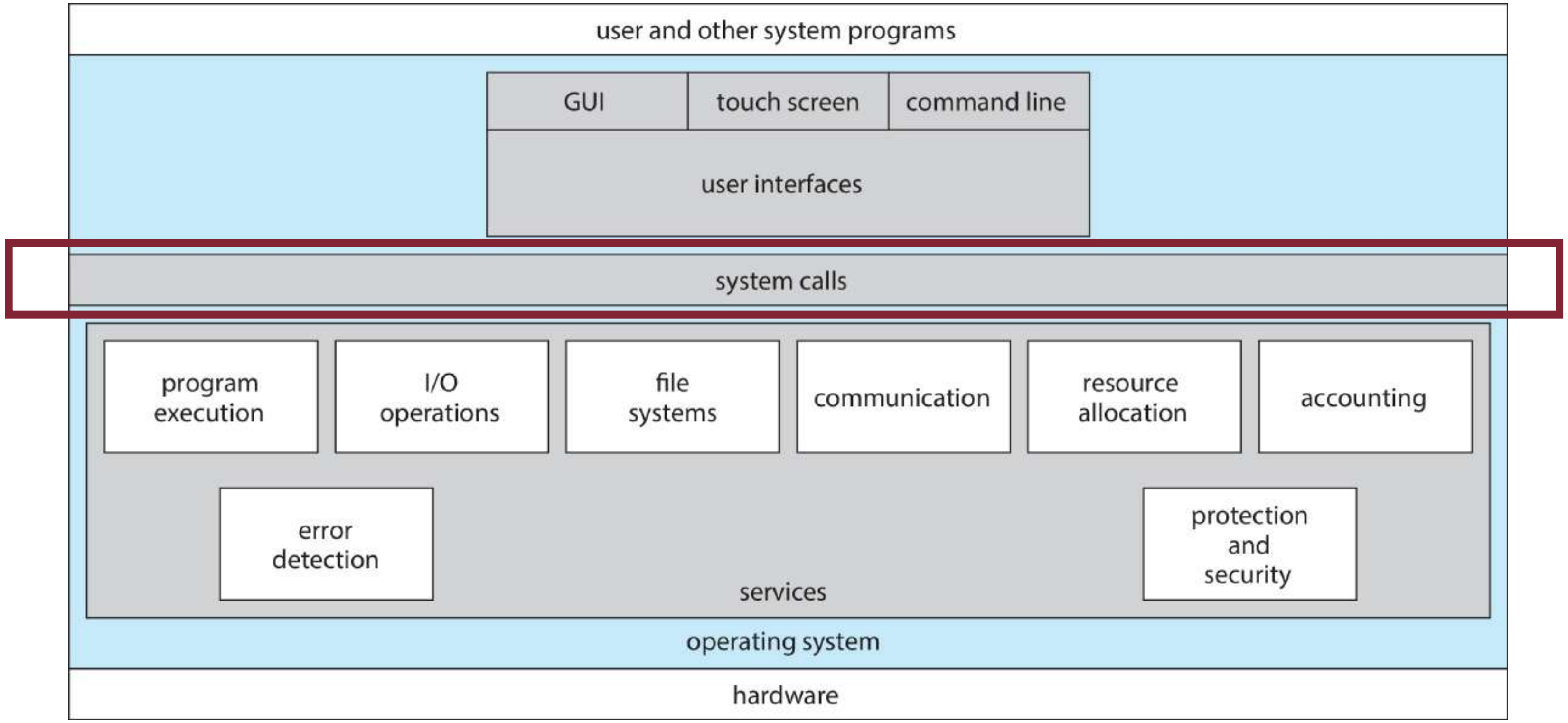
- Used by mobile smartphones and tablets
- Main features:
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
  - Voice commands

Touchscreen: iOS vs. Android





User-Programs to OS Interface



User Programs-OS Interface: System Calls

- OS procedures that execute privileged instructions (e.g., I/O)

User Programs-OS Interface: System Calls

- OS procedures that execute privileged instructions (e.g., I/O)
- Programming interface to the services provided by the OS

User Programs-OS Interface: System Calls

- OS procedures that execute privileged instructions (e.g., I/O)
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)

User Programs-OS Interface: System Calls

- OS procedures that execute privileged instructions (e.g., I/O)
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call
  - GNU C Library (POSIX-based systems like UNIX, Linux, macOS)
  - Win32 API (Windows systems)
  - Java API (JVM)

System Calls: Categories

- 6 main categories of system calls:
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communications

System Calls: Categories

- 6 main categories of system calls:
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communications

System Calls: Process Control

- Include end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory

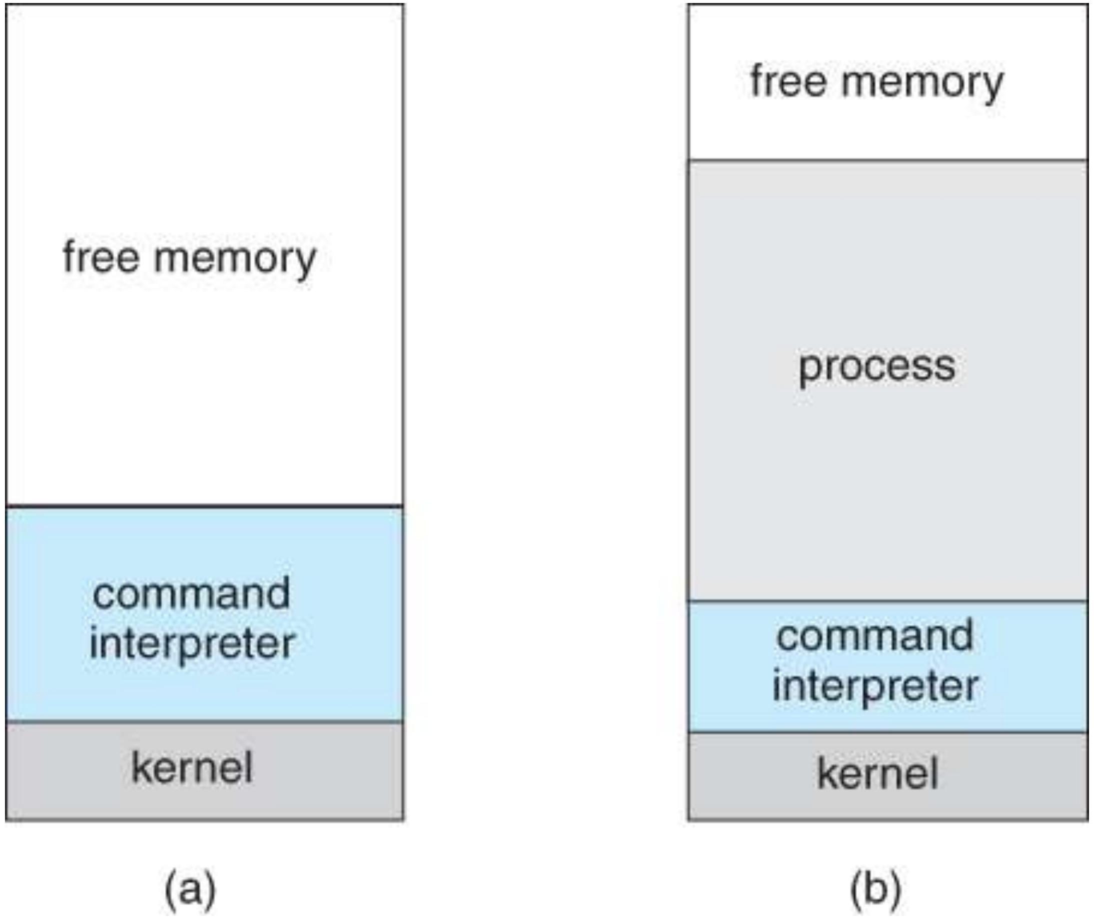
System Calls: Process Control

- Include end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory
- When one process pauses or stops, then another must be launched or resumed

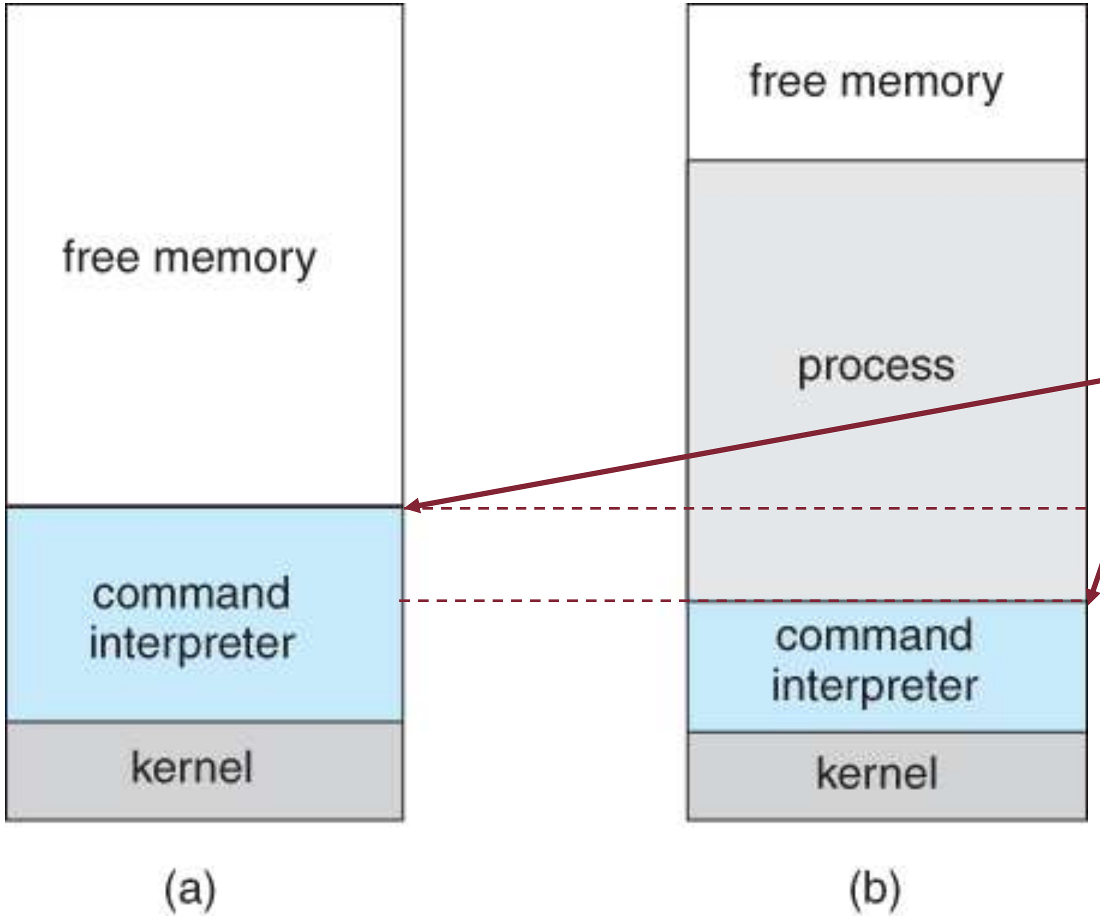
System Calls: Process Control

- Include end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory
- When one process pauses or stops, then another must be launched or resumed
- When processes stop abnormally it may be necessary to provide core dumps and/or other diagnostic or recovery tools

Process Control: MS-DOS (single-tasking)

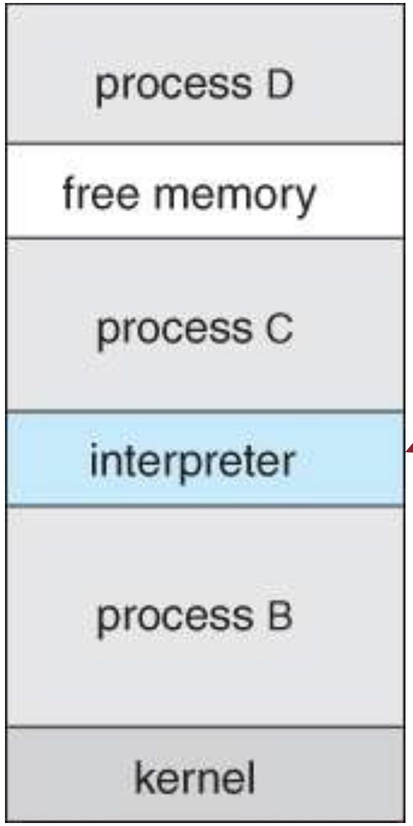


Process Control: MS-DOS (single-tasking)



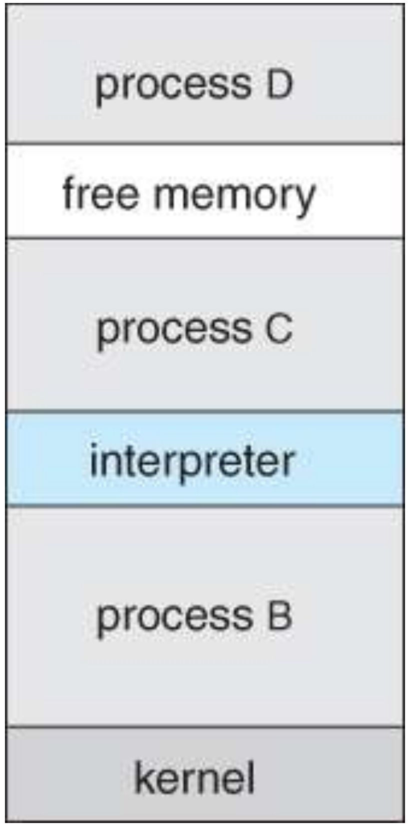
The command interpreter firstly unloads itself as much as it can to free up memory

Process Control: UNIX (multi-tasking)



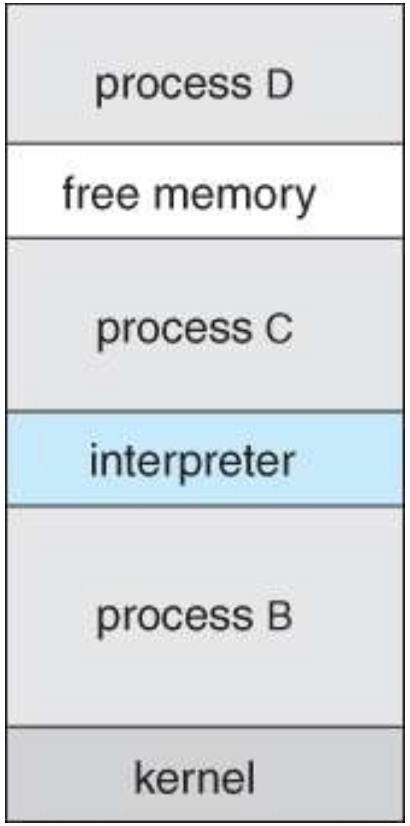
The command interpreter remains completely resident when executing a process

Process Control: UNIX (multi-tasking)



The user can switch back to the command interpreter at any time, and can place the running process in the background even if it was not originally launched as a background process

Process Control: UNIX (multi-tasking)



The user can switch back to the command interpreter at any time, and can place the running process in the background even if it was not originally launched as a background process

The command interpreter achieves that by issuing **`fork`**/**`exec`** system calls

System Calls: Categories

- 6 main categories of system calls:
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communications

System Calls: File Management

- Include create file, delete file, open, close, read, write, reposition, get file attributes, and set file attributes

System Calls: File Management

- Include create file, delete file, open, close, read, write, reposition, get file attributes, and set file attributes
- These operations may also be supported for directories as well as ordinary files

System Calls: File Management

- Include create file, delete file, open, close, read, write, reposition, get file attributes, and set file attributes
- These operations may also be supported for directories as well as ordinary files
- The actual directory structure may be implemented using ordinary files on the file system, or through other means (more on this later)

System Calls: Categories

- 6 main categories of system calls:
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communications

System Calls: Device Management

- Include request device, release device, read, write, reposition, get/set device attributes, and logically attach or detach devices

System Calls: Device Management

- Include request device, release device, read, write, reposition, get/set device attributes, and logically attach or detach devices
- Devices may be physical (e.g., disk drives), or virtual/abstract (e.g., files, partitions, and RAM disks)

System Calls: Device Management

- Include request device, release device, read, write, reposition, get/set device attributes, and logically attach or detach devices
- Devices may be physical (e.g., disk drives), or virtual/abstract (e.g., files, partitions, and RAM disks)
- Some systems represent devices as special files in the file system, so that accessing the "file" calls upon the appropriate OS device driver
  - e.g., the `/dev` directory on any UNIX system

System Calls: Categories

- 6 main categories of system calls:
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communications

System Calls: Information Maintenance

- Include calls to get/set the time, date, system data, and process, file, or device attributes

System Calls: Information Maintenance

- Include calls to get/set the time, date, system data, and process, file, or device attributes
- Systems may also provide the ability to dump memory at any time

System Calls: Information Maintenance

- Include calls to get/set the time, date, system data, and process, file, or device attributes
- Systems may also provide the ability to dump memory at any time
- Single step programs pausing execution after each instruction, and tracing the operation of programs (debugging)

System Calls: Categories

- 6 main categories of system calls:
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communications

System Calls: Communication

- Include create/delete communication connection, send/receive messages, transfer status information, and attach/detach remote devices

System Calls: Communication

- Include create/delete communication connection, send/receive messages, transfer status information, and attach/detach remote devices
- 2 models of communication:
  - message passing
  - shared memory

Communication: Message Passing

- The message passing model must support calls to:
  - Identify a remote process and/or host with which communicate to
  - Establish a connection between the two processes
  - Open and close the connection as needed
  - Transmit messages along the connection
  - Wait for incoming messages, in either a blocking or non-blocking state
  - Delete the connection when no longer needed

Communication: Message Passing

- The message passing model must support calls to:
  - Identify a remote process and/or host with which communicate to
  - Establish a connection between the two processes
  - Open and close the connection as needed
  - Transmit messages along the connection
  - Wait for incoming messages, in either a blocking or non-blocking state
  - Delete the connection when no longer needed

Simpler and easier (particularly for inter-computer communications) and generally appropriate for small amounts of data

Communication: Shared Memory

- The shared memory model must support calls to:
  - Create and access memory that is shared amongst processes (and threads)
  - Provide locking mechanisms restricting simultaneous access
  - Free up shared memory and/or dynamically allocate it as needed

Communication: Shared Memory

- The shared memory model must support calls to:
  - Create and access memory that is shared amongst processes (and threads)
  - Provide locking mechanisms restricting simultaneous access
  - Free up shared memory and/or dynamically allocate it as needed

Faster and generally the better approach where large amounts of data are to be shared

Ideal when most processes need to read data rather than write

System Calls: Protection

- Provides mechanisms for controlling which users/processes have access to which system resources

System Calls: Protection

- Provides mechanisms for controlling which users/processes have access to which system resources
- System calls allow the access mechanisms to be adjusted as needed

System Calls: Protection

- Provides mechanisms for controlling which users/processes have access to which system resources
- System calls allow the access mechanisms to be adjusted as needed
- Non-priveleged users may temporarily be granted elevated access permissions under specific circumstances

System Calls: Protection

- Provides mechanisms for controlling which users/processes have access to which system resources
- System calls allow the access mechanisms to be adjusted as needed
- Non-priveleged users may temporarily be granted elevated access permissions under specific circumstances
- Crucial in the age of ubiquitous network connectivity

The Anatomy of a System Call

System Call: **`read`** (C Library)

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)
```

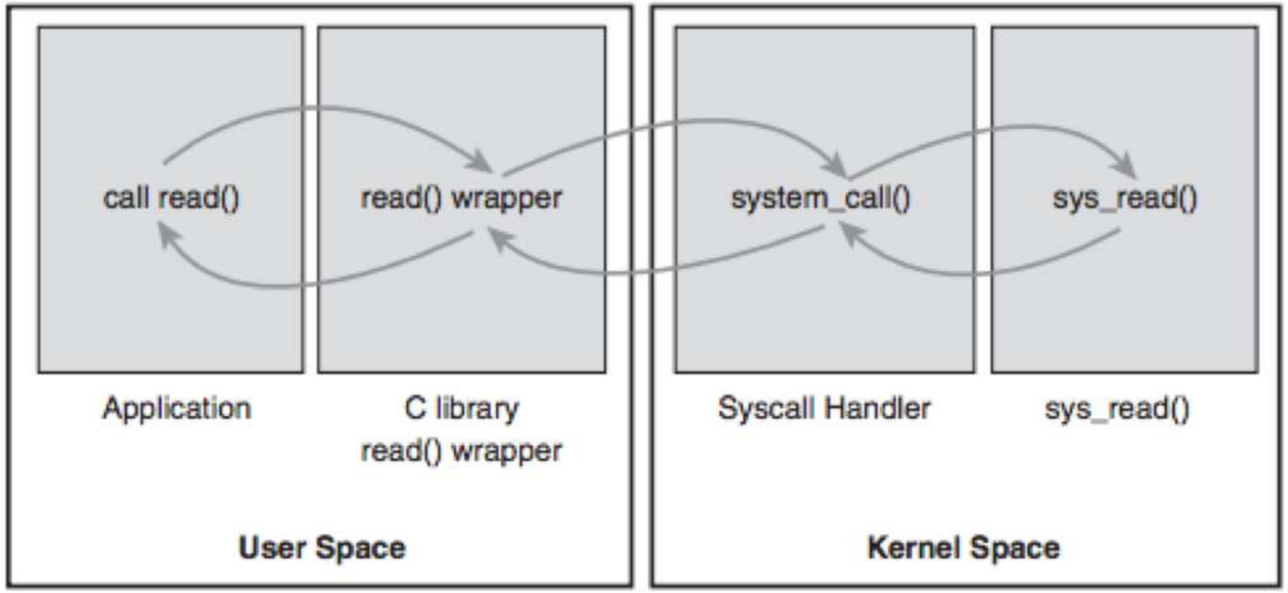
return value	function name	parameters
`ssize_t`	`read`	`(int fd, void *buf, size_t count)`

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 into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

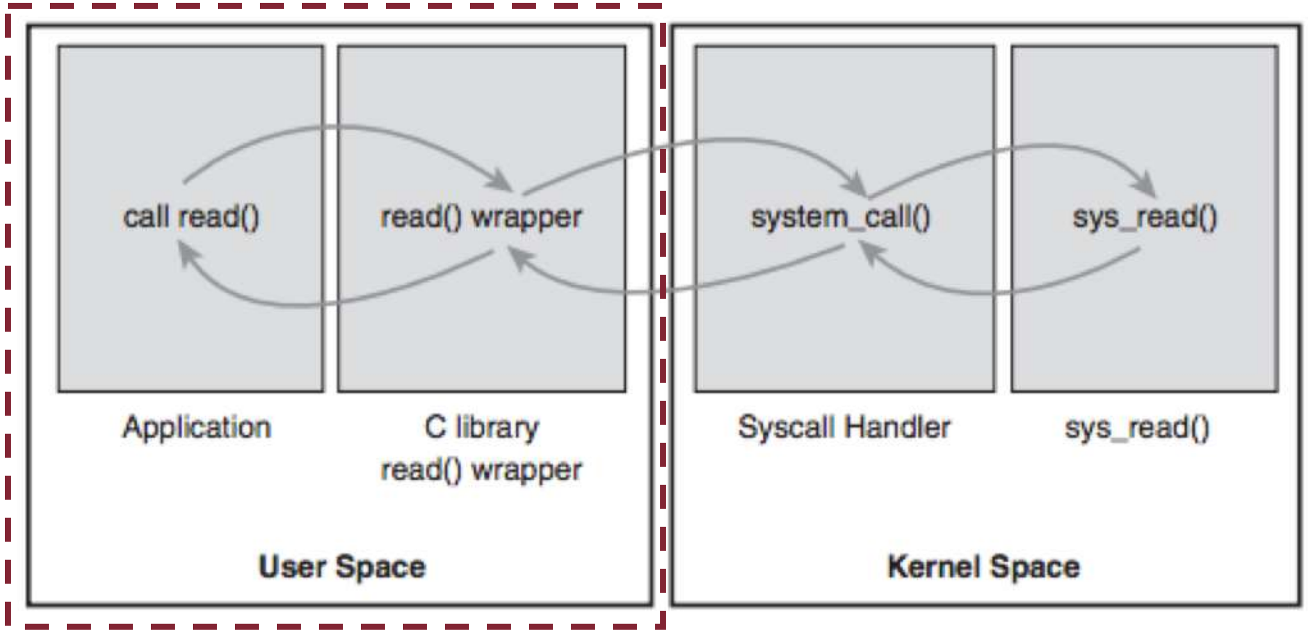
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

System Call: Flow



System Call: Flow

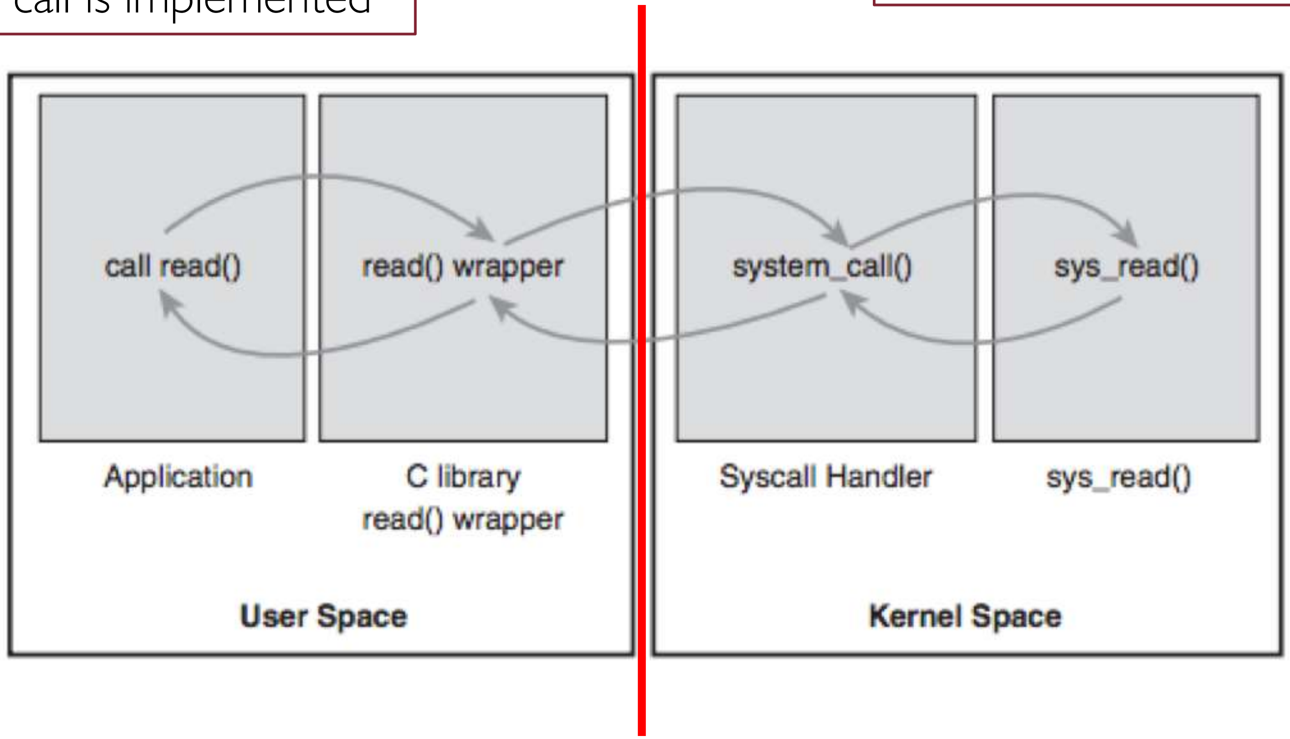
The caller (user program) doesn't have to know how the system call is implemented



System Call: Flow

The caller (user program) doesn't have to know how the system call is implemented

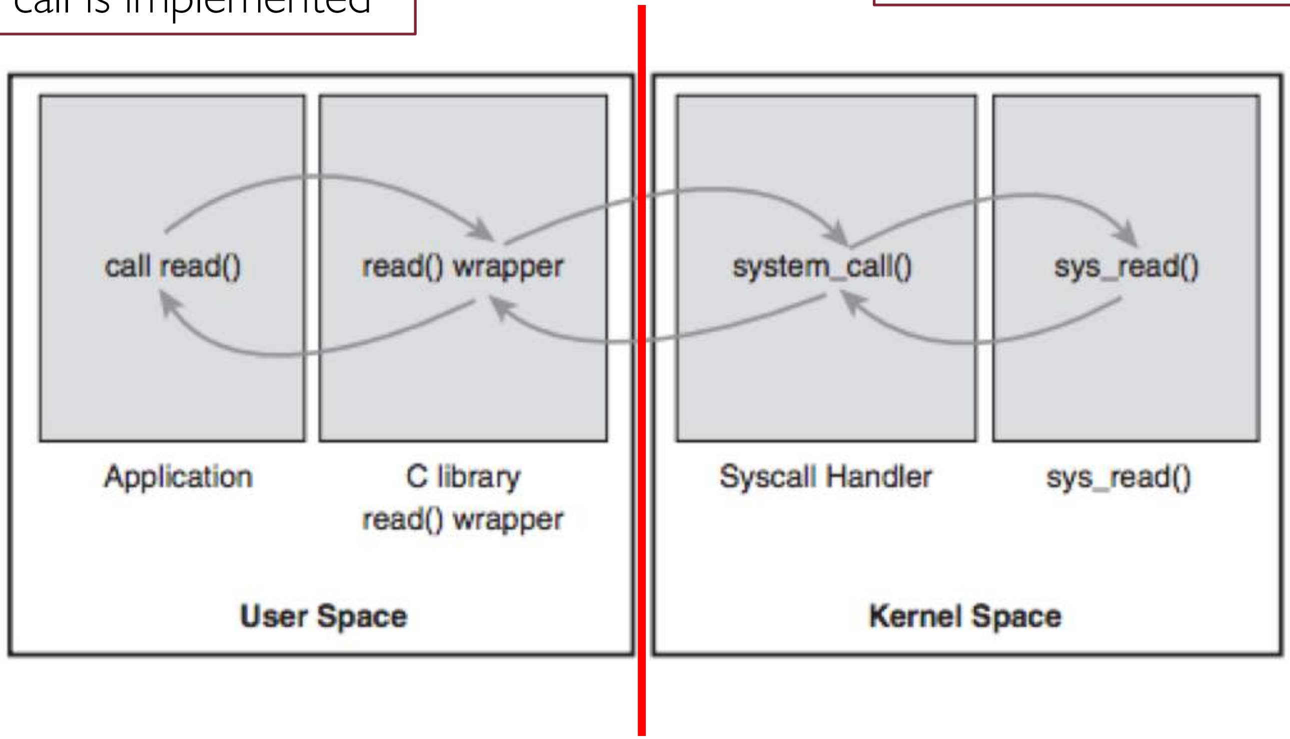
Most of the details are hidden by the API



System Call: Flow

The caller (user program) doesn't have to know how the system call is implemented

Most of the details are hidden by the API



The caller must only obey to the API
(know the input arguments and the expected output from the OS)

System Call Example: Reading from File

```
int main() {
    …
    int nRead = read(fd, buf, count);
    …
}
```

C library's **read** function call

System Call Example: Reading from File

```
int main() {
    …
    int nRead = read(fd, buf, count);
    …
}
```

C library's **read** function call

```
…
MOV %eax, $sys_read
INT $0x80
…
```

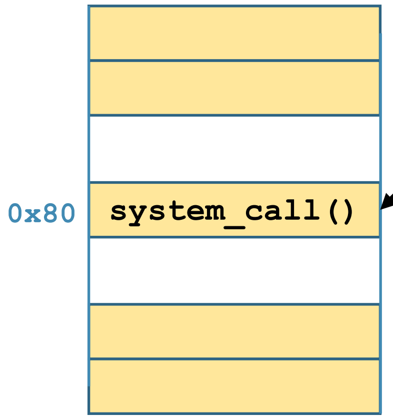
store the number which uniquely identifies the system call requested

System Call Example: Reading from File

```
int main() {
    …
    int nRead = read(fd, buf, count);
    …
}
```

```
…
MOV %eax, $sys_read
INT $0x80
…
```

A trap jumps to the interrupt vector table (IVT) in the OS kernel

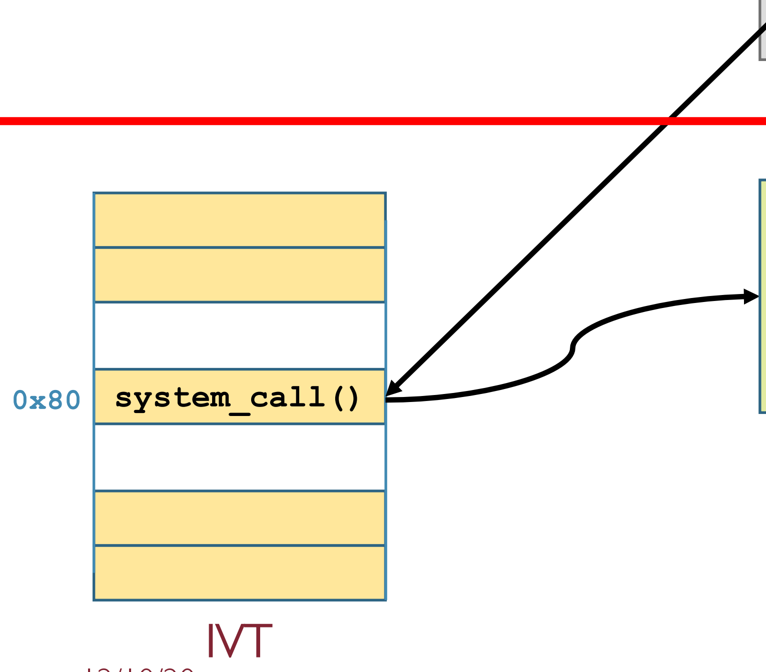


IVT

System Call Example: Reading from File

```
int main() {
    …
    int nRead = read(fd, buf, count);
    …
}
```

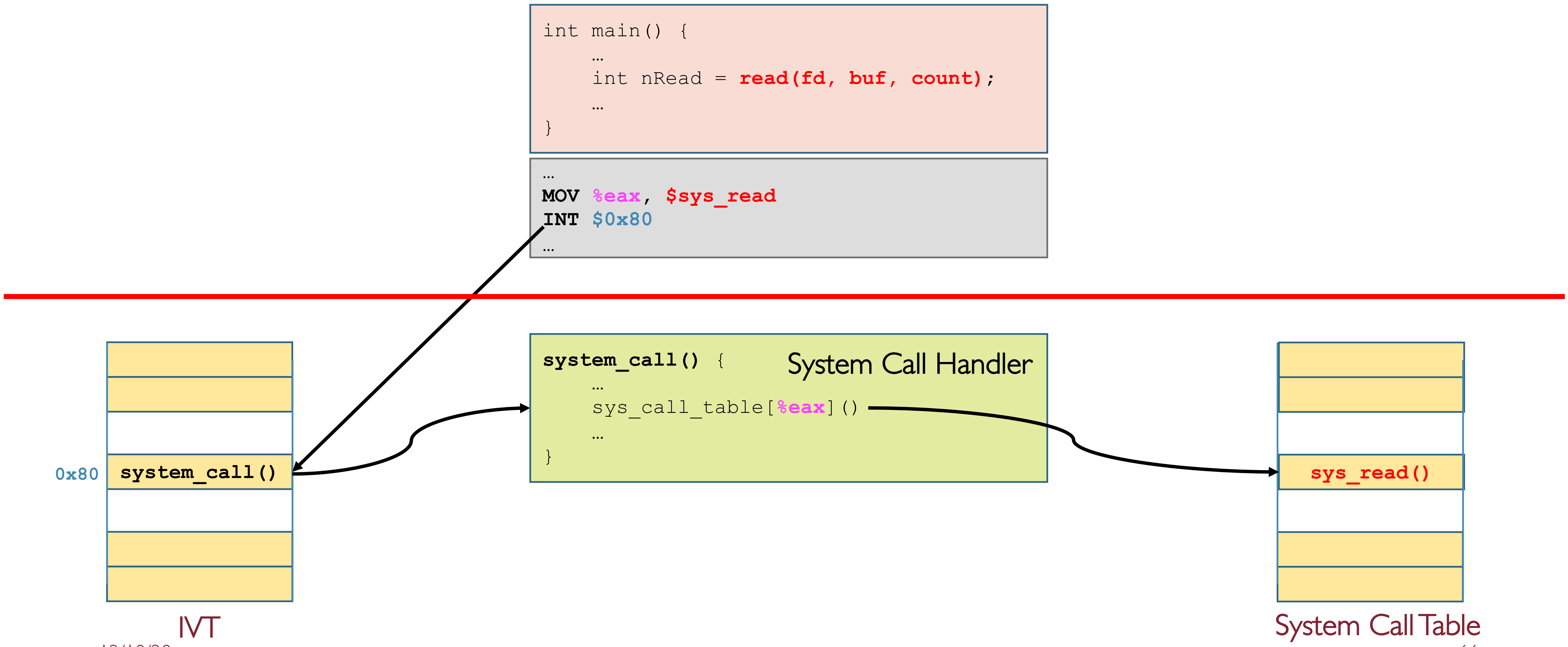
```
…
MOV %eax, $sys_read
INT $0x80
…
```



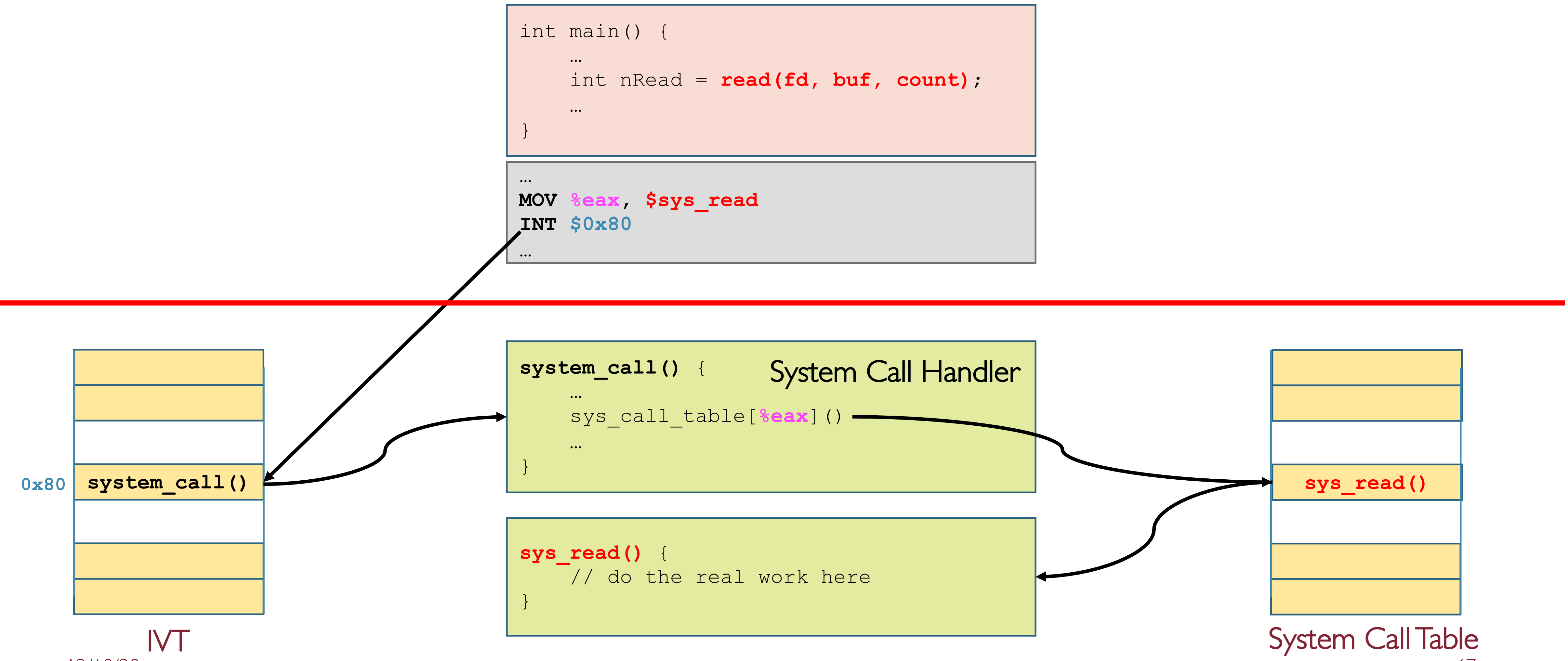
System Call Handler

```
system_call() {
    …
    sys_call_table[%eax]()
    …
}
```

System Call Example: Reading from File



System Call Example: Reading from File



System Call Handler

- The trap caused by system call invokation makes the CPU switch from user to kernel mode

System Call Handler

- The trap caused by system call invokation makes the CPU switch from user to kernel mode
- The **system call handler** is responsible for:
  - saving the status of user-mode computation on dedicated registers
  - finding and jumping to the correct routine for that trap (e.g., **`sys_read()`**)
  - restoring user-mode program's state upon the service routine is done (e.g., **`IRET`** privileged instruction)

Parameter Passing

- Often, more information is required than simply the identifier of the desired system call

Parameter Passing

- Often, more information is required than simply the identifier of the desired system call
- 3 methods used to pass parameters to the OS
  - Store parameters in registers (may be more parameters than registers)

Parameter Passing

- Often, more information is required than simply the identifier of the desired system call
- 3 methods used to pass parameters to the OS
  - Store parameters in registers (may be more parameters than registers)
  - Store parameters in a block or table in a dedicated area of memory, and address of block passed as a parameter in a register (Linux and Solaris)

Parameter Passing

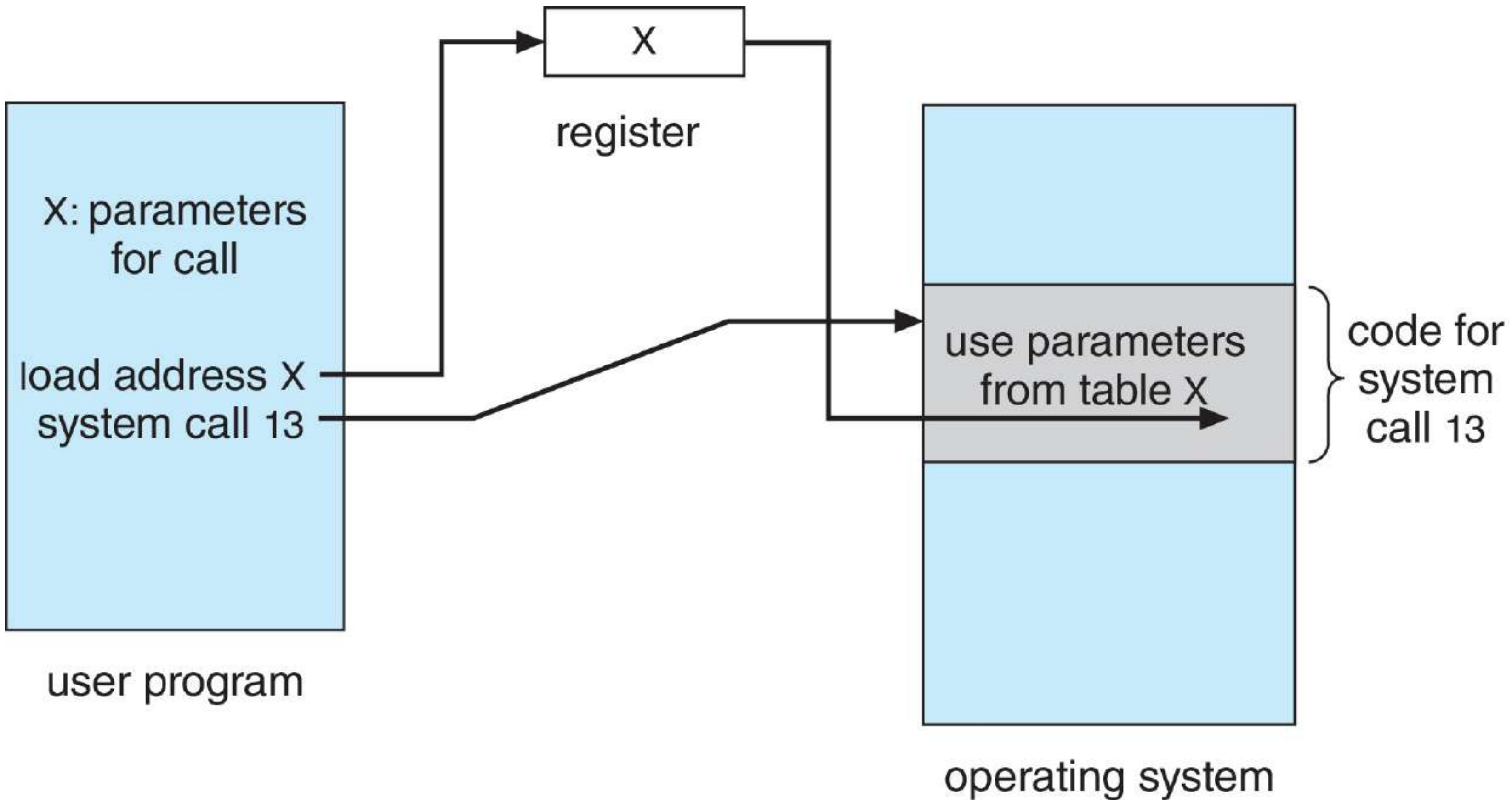
- Often, more information is required than simply the identifier of the desired system call
- 3 methods used to pass parameters to the OS
  - Store parameters in registers (may be more parameters than registers)
  - Store parameters in a block or table in a dedicated area of memory, and address of block passed as a parameter in a register (Linux and Solaris)
  - Parameters pushed onto the stack by the program and popped off the stack by the OS (more complex due to different address spaces!)

Parameter Passing

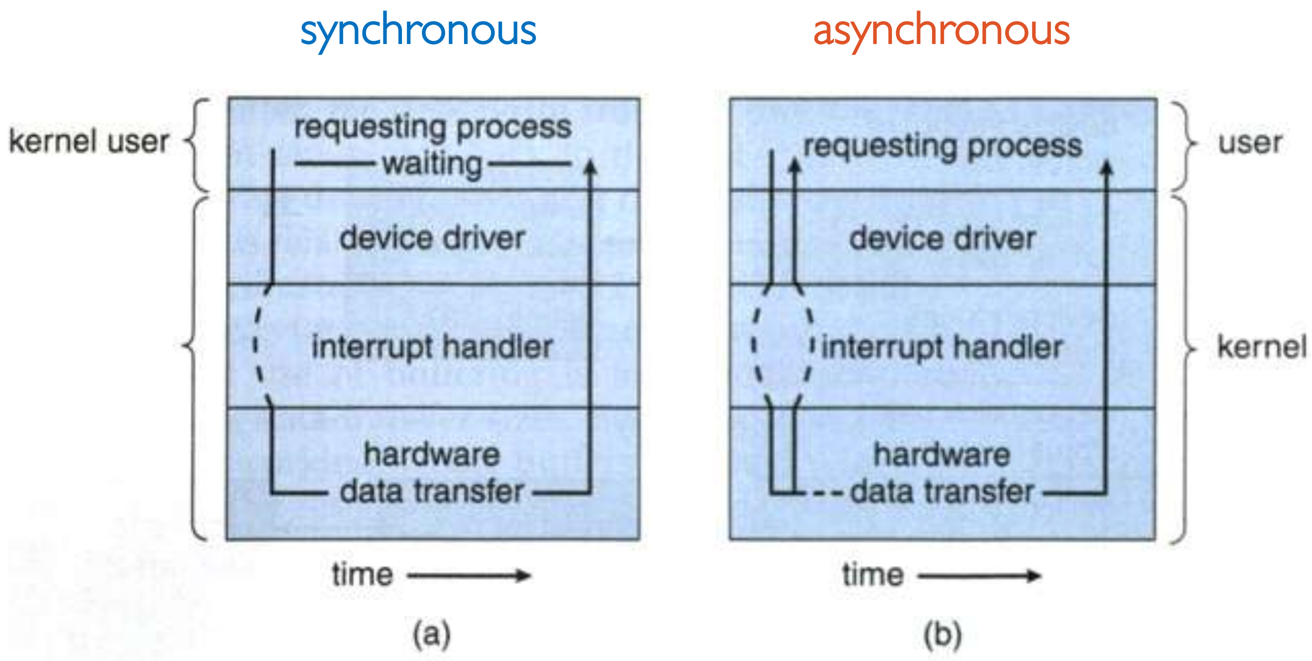
- Often, more information is required than simply the identifier of the desired system call
- 3 methods used to pass parameters to the OS
  - Store parameters in registers (may be more parameters than registers)
  - Store parameters in a block or table in a dedicated area of memory, and address of block passed as a parameter in a register (Linux and Solaris)
  - Parameters pushed onto the stack by the program and popped off the stack by the OS (more complex due to different address spaces!)

Block and stack methods do not limit the number or length of parameters being passed

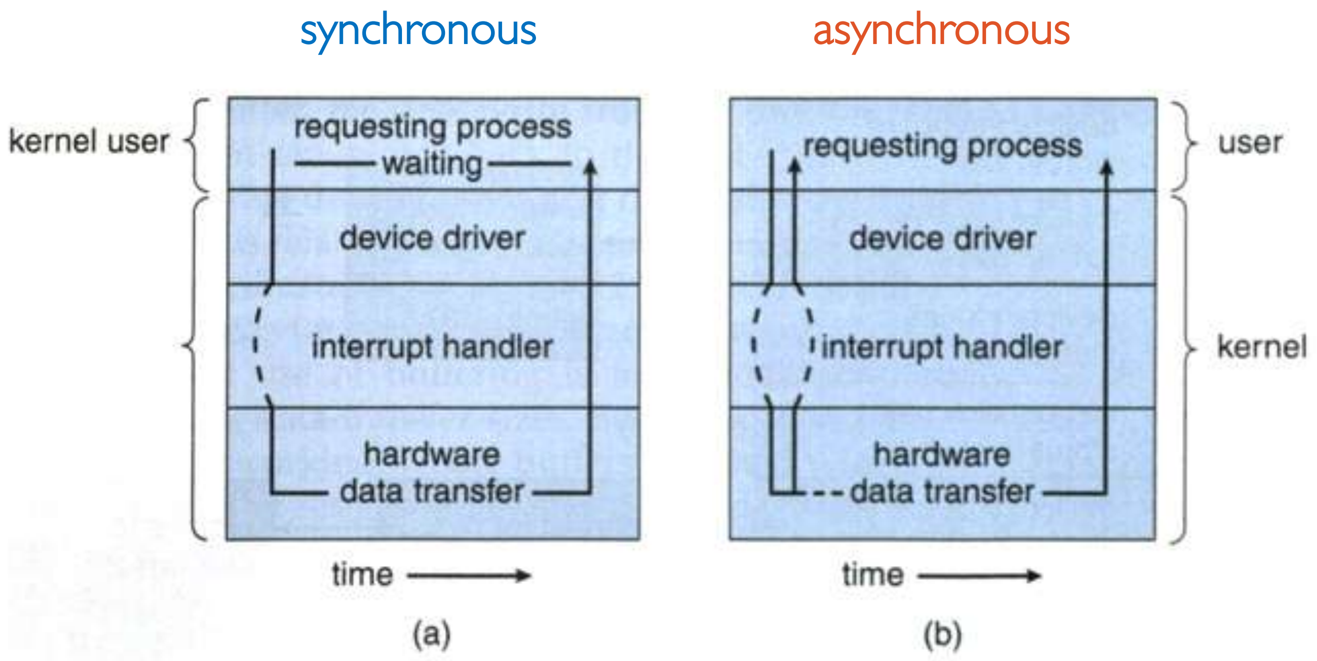
Parameter Passing via Table



Blocking vs. Non-Blocking I/O



Blocking vs. Non-Blocking I/O



NOTE

In a multi-programming and multi-tasking system, blocking I/O will not leave the CPU idle until I/O task is completed!
In fact, the CPU will schedule another (ready) process to take over

System Calls: Windows vs. UNIX APIs

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

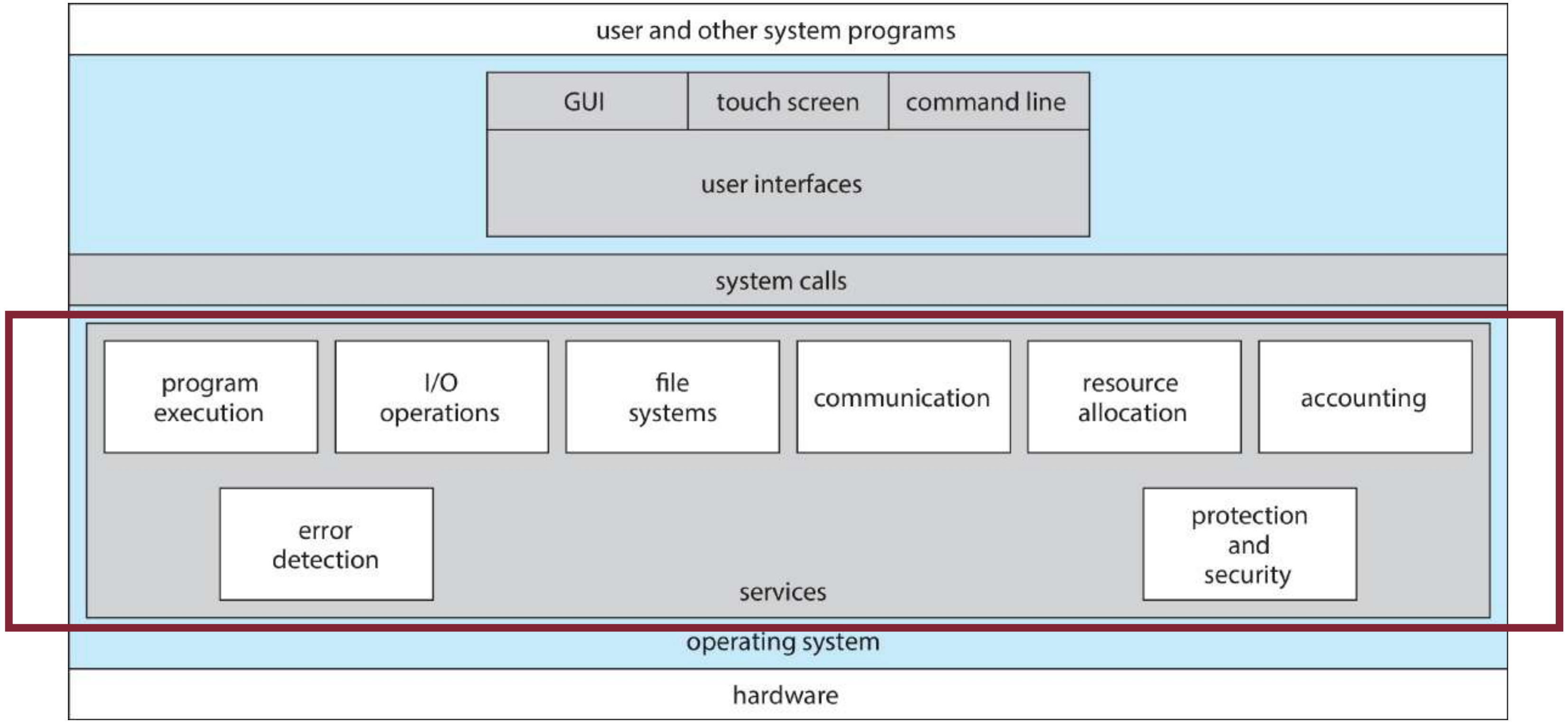
Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

System Programs



System Programs

- Provide OS functionality through separate applications, which are not part of the kernel or command interpreters

System Programs

- Provide OS functionality through separate applications, which are not part of the kernel or command interpreters
- They are also known as system utilities or system applications

System Programs

- Provide OS functionality through separate applications, which are not part of the kernel or command interpreters
- They are also known as system utilities or system applications
- Most systems also ship with useful applications such as calculators and simple editors (e.g., Emacs)

System Programs

- Provide OS functionality through separate applications, which are not part of the kernel or command interpreters
- They are also known as system utilities or system applications
- Most systems also ship with useful applications such as calculators and simple editors (e.g., Emacs)
- Some debate arises as to the border between system and non-system applications

System Programs

- They can be grouped into 7 categories:
  - File management: programs to create, delete, copy, rename, print, list, and manipulate files and directories
  - Status information: utilities to check on the date, time, number of users, processes running, data logging, etc.
  - File modification: text editors and other tools which can change file contents
  - Programming-language support: compilers, linkers, debuggers, profilers, assemblers, library archive management, interpreters for common languages, and support for make
  - Program loading and execution: loaders, dynamic loaders, overlay loaders, debuggers, etc.
  - Communications: email, web browsers, remote login, file transfer, etc.
  - Background services: network daemons, print servers, process schedulers, and system error monitoring services

System Programs

- Many operating systems today also come with a set of application programs to provide additional services, such as copying files or checking the time and date

System Programs

- Many operating systems today also come with a set of application programs to provide additional services, such as copying files or checking the time and date
- Most users' views of the system is determined by their command interpreter and the application programs

System Programs

- Many operating systems today also come with a set of application programs to provide additional services, such as copying files or checking the time and date
- Most users' views of the system is determined by their command interpreter and the application programs
- Users never make system calls, even through the API, unless they develop programs (e.g., requiring I/O operations)

OS Design and Implementation

Design Goals

- The internal structure of different OSs can vary widely

Design Goals

- The internal structure of different OSs can vary widely
- User vs. System goals
  - easy to use vs. easy to design/implement

Design Goals

- The internal structure of different OSs can vary widely
- User vs. System goals
  - easy to use vs. easy to design/implement
- It is crucial to separate policies from mechanisms
  - policy → *what* will be done
  - mechanism → *how* to do it

Policy vs. Mechanism

- Decoupling policy logic from the underlying mechanism is a general design principle in computer science, as it improves system's:
  - flexibility → addition and modification of policies can be easily supported
  - reusability → existing mechanisms can be reused for implementing new policies
  - stability → adding a new policy doesn't necessarily destabilize the system

Policy vs. Mechanism

- Decoupling policy logic from the underlying mechanism is a general design principle in computer science, as it improves system's:
  - flexibility → addition and modification of policies can be easily supported
  - reusability → existing mechanisms can be reused for implementing new policies
  - stability → adding a new policy doesn't necessarily destabilize the system
- Policy changes can be easily adjusted without re-writing the code

OS Implementation

- Early OSs developed in assembly language,
  - PRO → direct control over the HW (high efficiency)
  - CON → bound to a specific HW (low portability)

OS Implementation

- Early OSs developed in assembly language,
  - PRO → direct control over the HW (high efficiency)
  - CON → bound to a specific HW (low portability)
- Today, a mixture of languages:
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, etc.

OS Structure

- OS should be partitioned into separate subsystems, each with carefully defined tasks, inputs, outputs, and performance characteristics

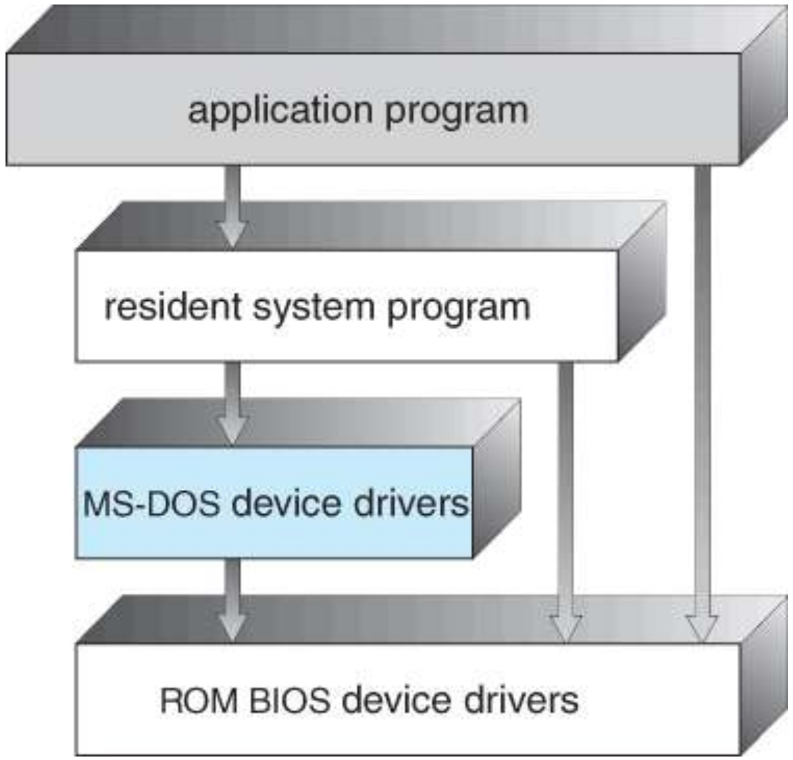
OS Structure

- OS should be partitioned into separate subsystems, each with carefully defined tasks, inputs, outputs, and performance characteristics
- Various ways to structure an operating system:
  - Simple → MS-DOS
  - Complex → UNIX
  - Layered → MULTICS
  - Microkernel → Mach

MS-DOS Structure: Simple Structure

No modular subsystems at all!

No separation between
user and kernel mode



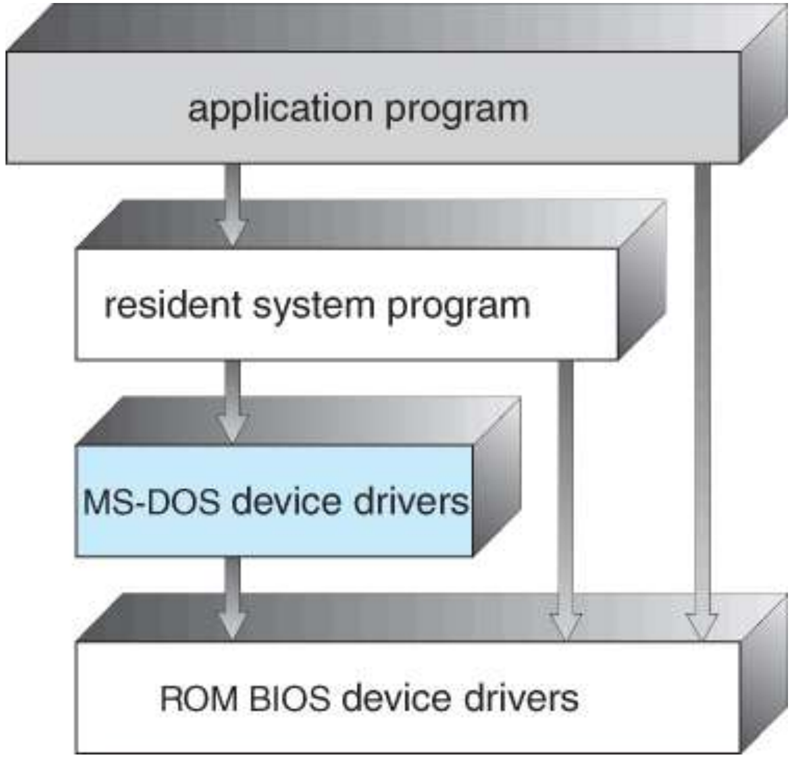
MS-DOS Structure: Simple Structure

No modular subsystems at all!

No separation between
user and kernel mode

PROs: easy to implement

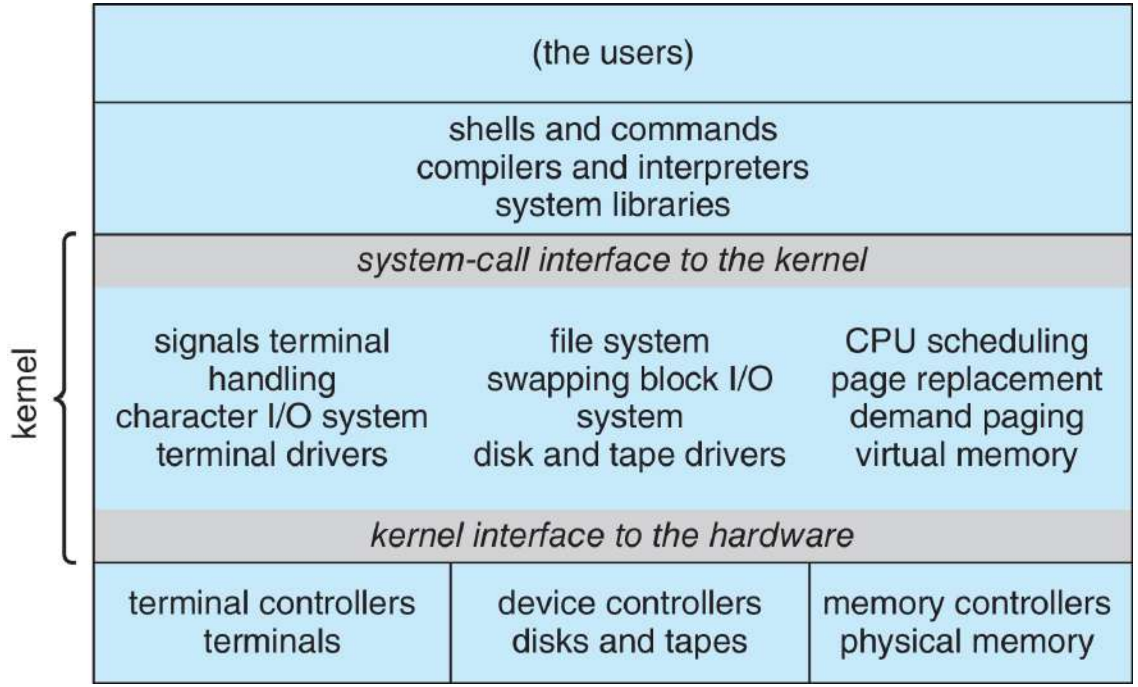
CONs: rigidity, security



UNIX Structure: Traditional Monolithic Kernel

Essentially, one huge piece of software with all services living in the same address space as one big process

Most of modern OSs are variant of this traditional monolithic structure



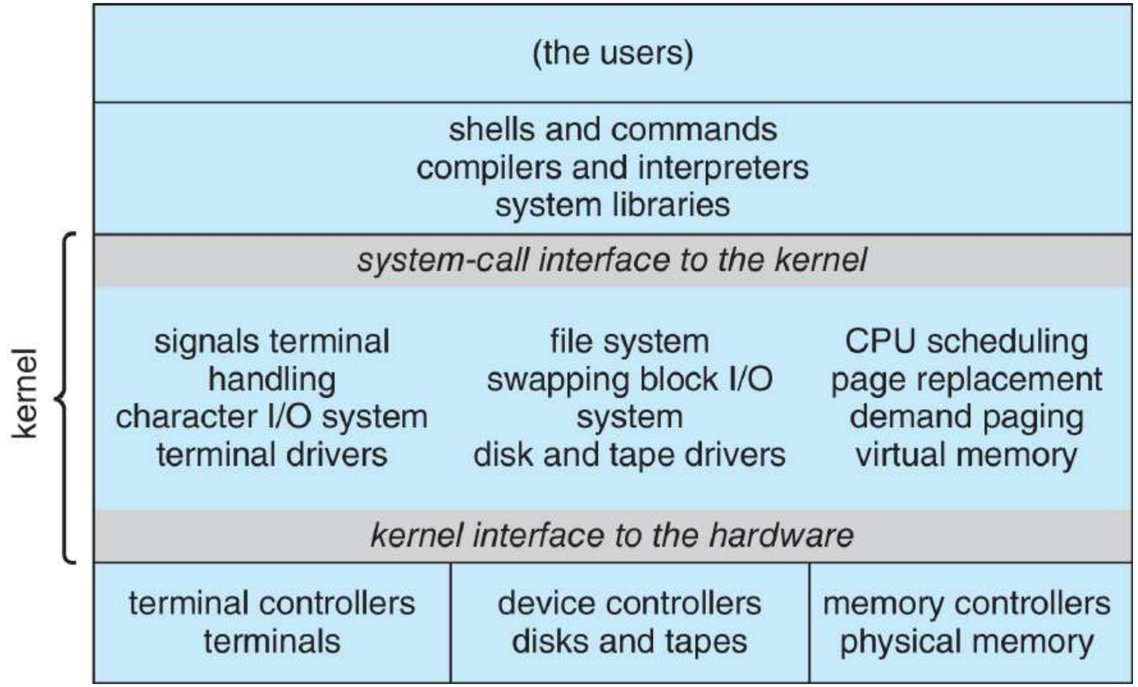
UNIX Structure: Traditional Monolithic Kernel

Essentially, one huge piece of software with all services living in the same address space as one big process

Most of modern OSs are variant of this traditional monolithic structure

PROs: efficiency, easy to implement

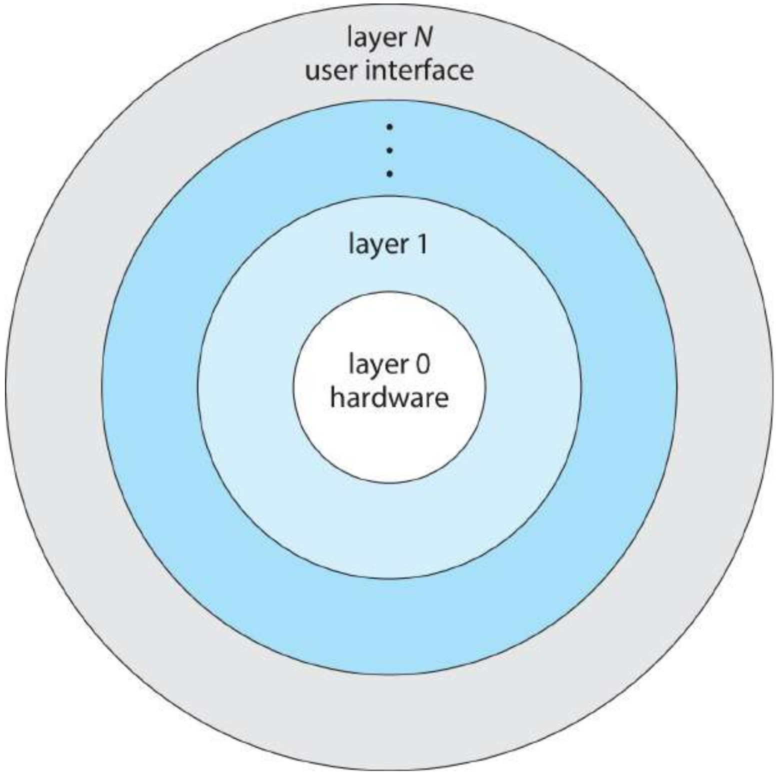
CONs: rigidity, security



Layered Structure

The OS is divided into *N* layers
(HW = layer 0)

Each layer *L* uses the functionalities implemented by the layer *L-1* to expose new functionalities to layer *L+1*



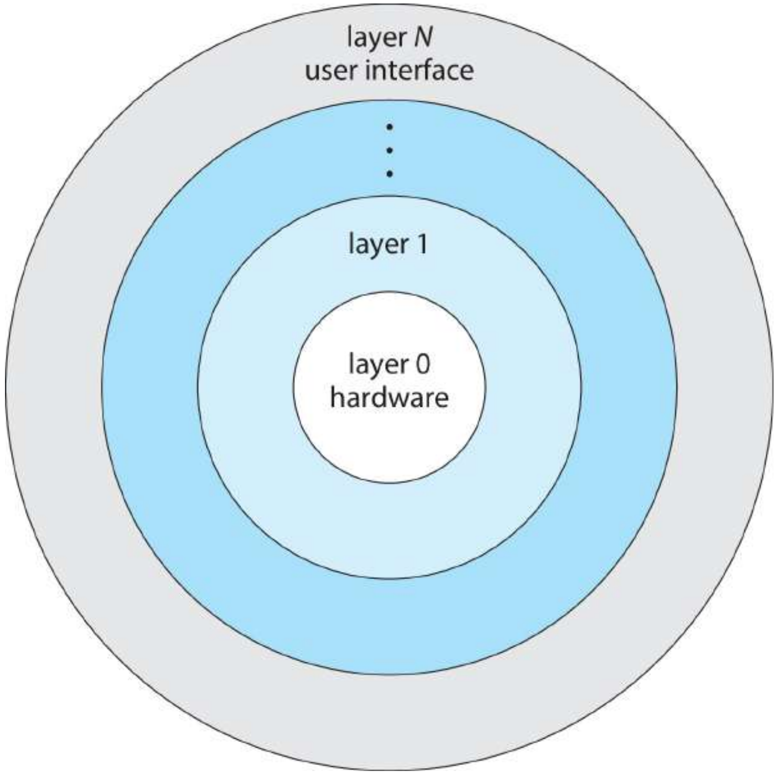
Layered Structure

The OS is divided into *N* layers
(HW = layer 0)

Each layer *L* uses the functionalities implemented by the layer *L-1* to expose new functionalities to layer *L+1*

PROs: modularity, portability, easy to debug

CONs: communication overhead, extra copy

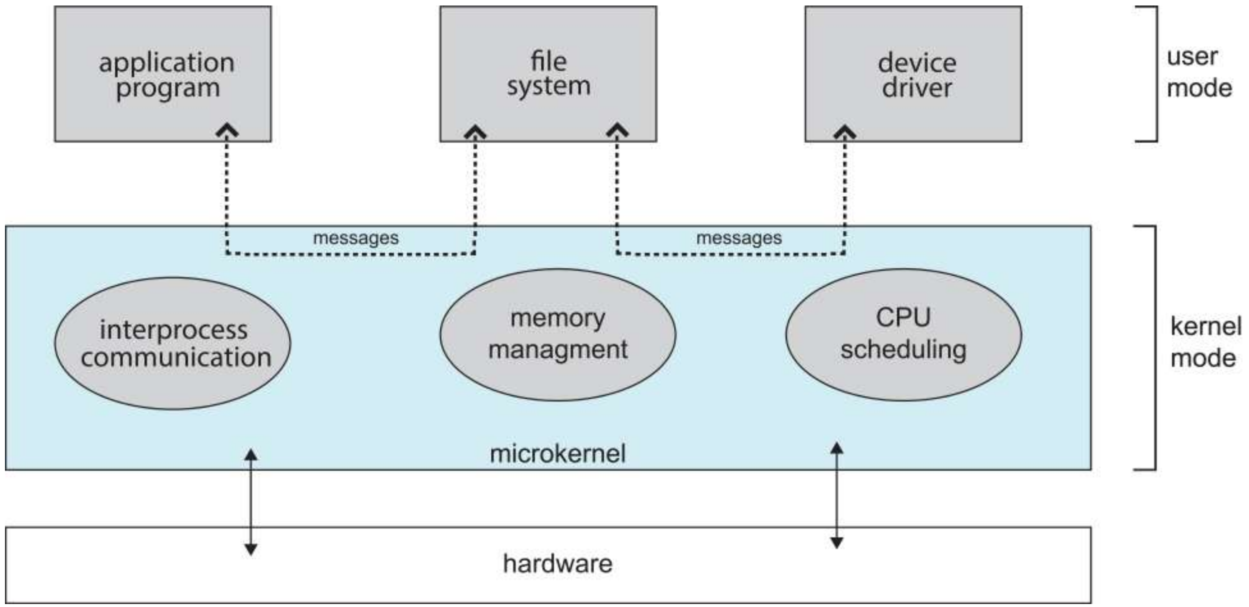


Microkernel Structure

The opposite approach of monolithic

The kernel just contains very basic functionalities, everything else which is still logically part of the OS runs in user mode

Policy (user mode) vs. mechanism (microkernel) separation



Microkernel Structure

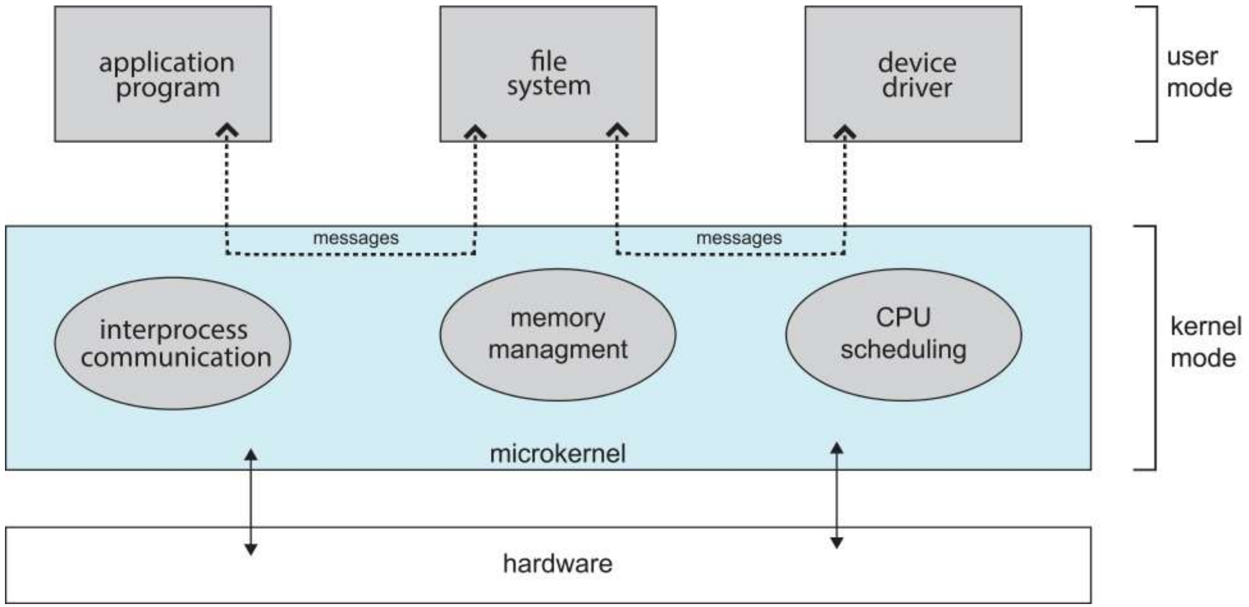
The opposite approach of monolithic

The kernel just contains very basic functionalities, everything else which is still logically part of the OS runs in user mode

Policy (user mode) vs. mechanism (microkernel) separation

PROs: security, reliability, extendibility

CONs: efficiency (message passing)



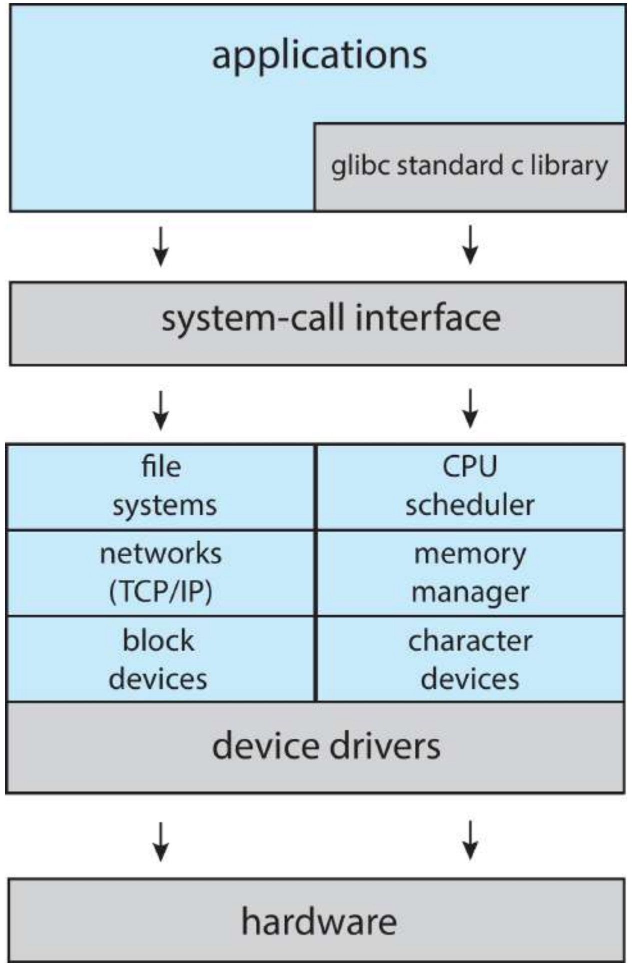
Loadable Kernel Modules (LKMs)

- Many modern OSs use loadable kernel modules (LKMs)
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel (i.e., in kernel space)
- Similar to layered structure but more flexible

Monolithic vs. Microkernel: Hybrid Trade-off

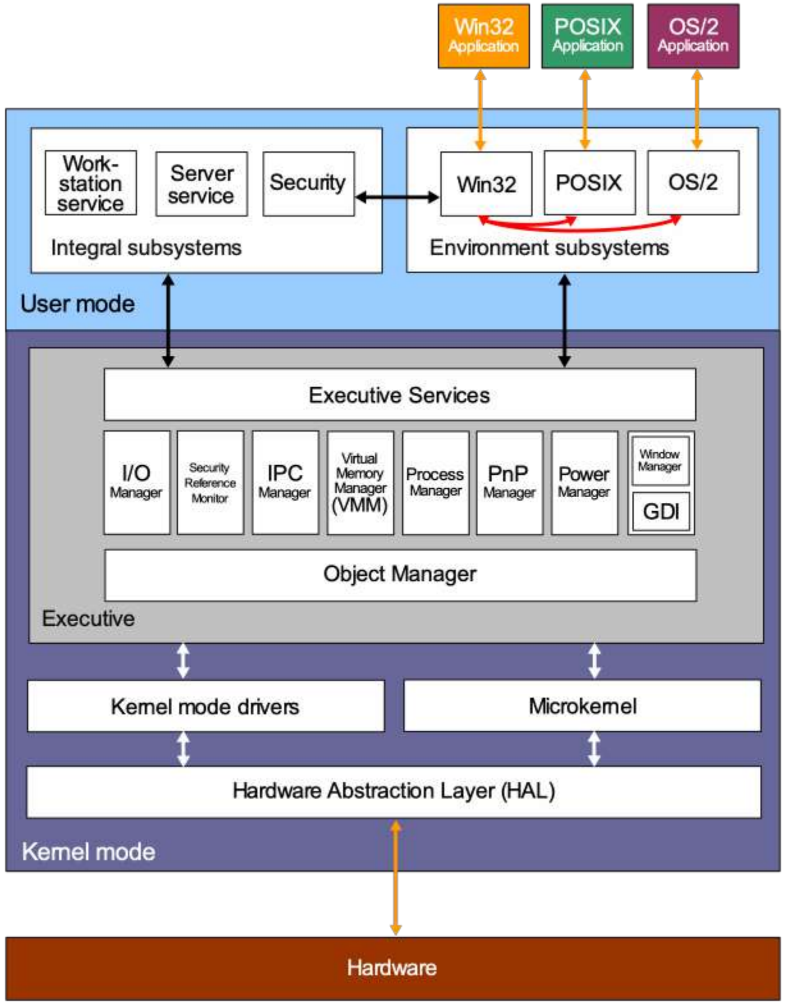
- Try to get the best out of both approaches
  - combining multiple approaches to address performance, security, usability needs
- Linux and Solaris: monolithic + LKMs (i.e., modular monolithic)
- Windows NT: mostly monolithic + microkernel for different subsystems
- Apple Mac OS X: monolithic (BSD UNIX) + microkernel (Mach) + LKMs

Linux

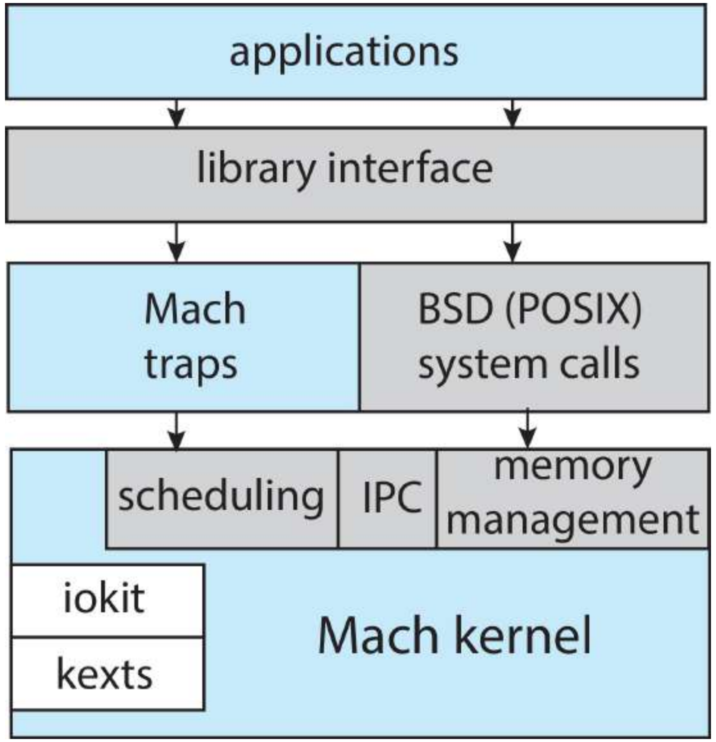


applications
glibc standard c library
system-call interface
file systems
CPU scheduler
networks (TCP/IP)
memory manager
block devices
character devices
device drivers
hardware

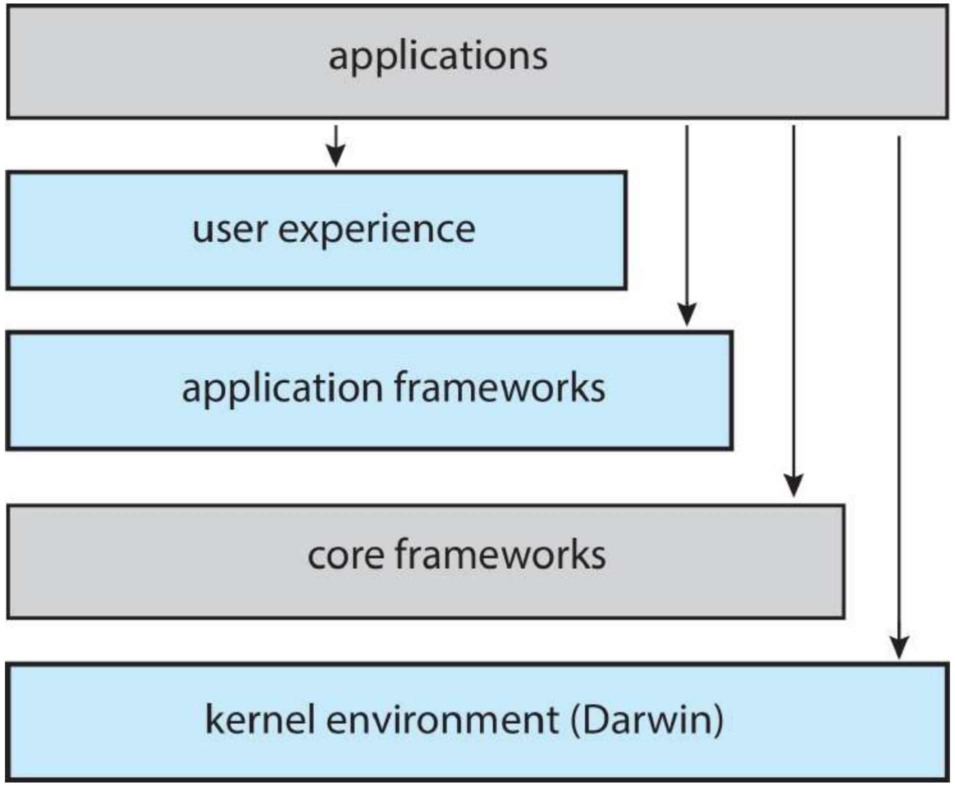
Windows NT Family



Mac OS X (Darwin)

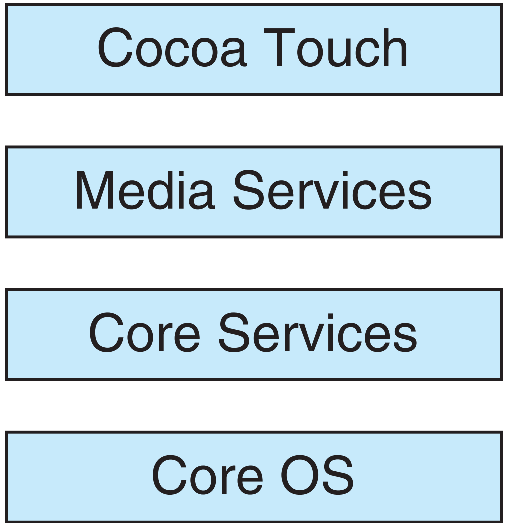


macOS and iOS



iOS

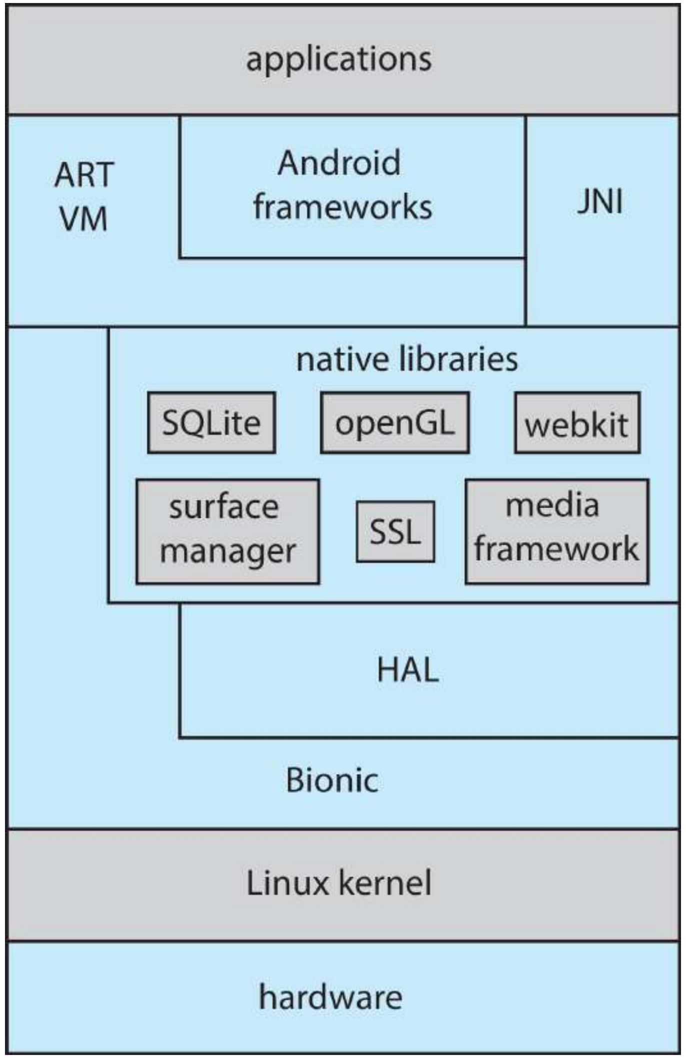
- Apple mobile OS for iPhone and iPad
  - Core operating system based on Mac OS X kernel + added functionalities
  - Does not run OS X applications natively
  - Also runs on different CPU architecture (ARM vs. Intel)
- Cocoa Touch Objective-C API for developing apps
- Media services layer for graphics, audio, video
- Core services provides cloud computing, databases



Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
  - Similar stack to iOS
  - Based on Linux kernel but modified
- Provides process, memory, device-driver management, and power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android


applications
ART
VM
Android
frameworks
JNI
native libraries
SQLite
openGL
webkit
surface
manager
SSL
media
framework
HAL
Bionic
Linux kernel
hardware

Booting: Where Everything Starts

- What happens when we power up our computer?

Booting: Where Everything Starts

- What happens when we power up our computer?
    1. The CPU is reset → kernel mode (0) and jump to a specific location of non-volatile memory (ROM)

Booting: Where Everything Starts

- What happens when we power up our computer?
  1. The CPU is reset → kernel mode (0) and jump to a specific location of non-volatile memory (ROM)
  2. ROM contains the code for executing Basic Input/Output System (BIOS)

Booting: Where Everything Starts

- What happens when we power up our computer?
  1. The CPU is reset → kernel mode (0) and jump to a specific location of non-volatile memory (ROM)
  2. ROM contains the code for executing Basic Input/Output System (BIOS)
  3. BIOS loads the boot(strap) loader from the Master Boot Record (MBR) of a disk into main memory (RAM)

Booting: Where Everything Starts

- What happens when we power up our computer?
  1. The CPU is reset → kernel mode (0) and jump to a specific location of non-volatile memory (ROM)
  2. ROM contains the code for executing Basic Input/Output System (BIOS)
  3. BIOS loads the boot(strap) loader from the Master Boot Record (MBR) of a disk into main memory (RAM)
  4. The boot loader loads the OS kernel, which:
     - Initializes its own data structures (e.g., interrupt vector table)
     - Loads the first process from disk
     - Switch to user mode (1)

Summary

- OS provides a lot of useful services to users/programs for interacting with the system
  - CLI/GUI
  - System calls

Summary

- OS provides a lot of useful services to users/programs for interacting with the system
  - CLI/GUI
  - System calls
- Several ways of designing an OS
  - Tradeoff between usability, reliability, security, etc.

Summary

- OS provides a lot of useful services to users/programs for interacting with the system
  - CLI/GUI
  - System calls
- Several ways of designing an OS
  - Tradeoff between usability, reliability, security, etc.
- There is no "one size fits all" receipe!
  - Each system must be designed on the basis of its purpose