Sistemi Operativi I

Corso di Laurea in Informatica

2024-2025

SAPIENZA
UNIVERSITÀ DI ROMA

Gabriele Tolomei

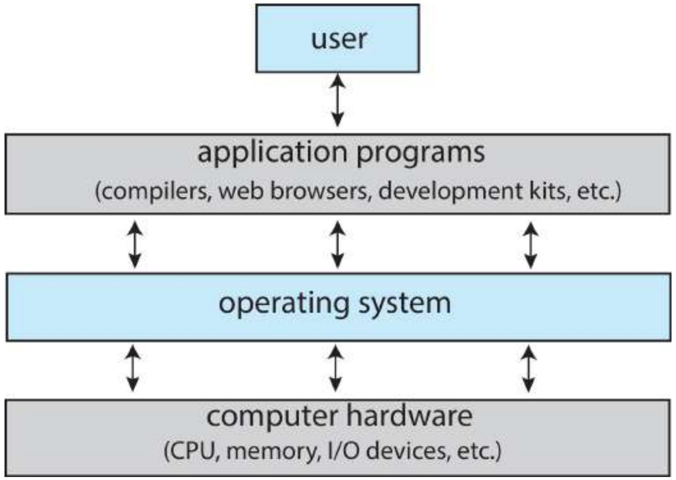
Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

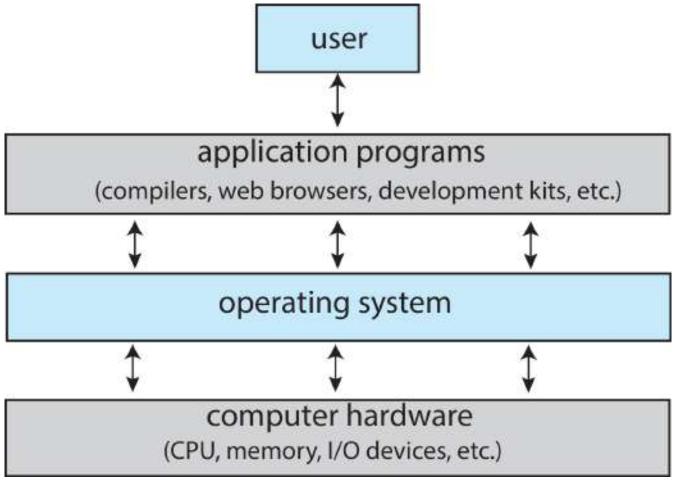
Recap from Last Lecture

- Operating System is a complex system which plays several roles:
  - resource manager
  - virtual machine
  - HW/SW interface



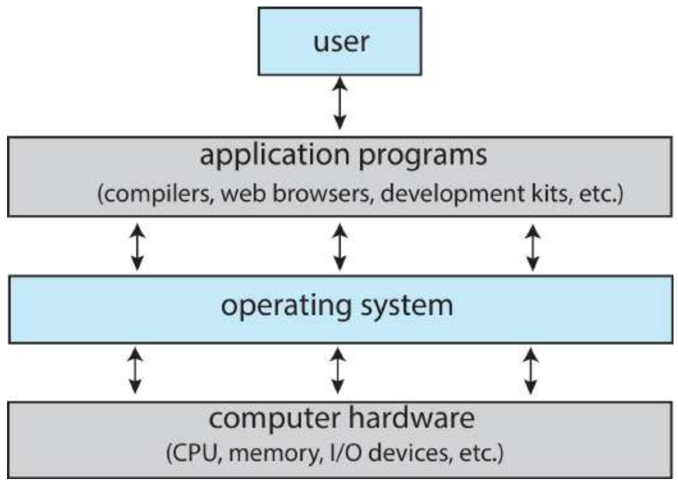
Recap from Last Lecture

- Operating System is a complex system which plays several roles:
  - resource manager
  - virtual machine
  - HW/SW interface
- Exposes services to users/applications (SW) leveraging the physical machine (HW)



Recap from Last Lecture

- Operating System is a complex system which plays several roles:
  - resource manager
  - virtual machine
  - HW/SW interface
- Exposes services to users/applications (SW) leveraging the physical machine (HW)
- Changes in HW may affect OS design



Outline of this Lecture

1. HW support for OS functionalities and services
2. OS design and implementation

HW Support for OS

OS and Computer Architecture

- Basic OS functionalities (enabled by architectural features)

OS and Computer Architecture

- Basic OS functionalities (enabled by architectural features)
- What the OS can do is partially dictated by the underlying architecture

OS and Computer Architecture

- Basic OS functionalities (enabled by architectural features)
- What the OS can do is partially dictated by the underlying architecture
- Architectural support may significantly simplify or complicate the OS design

Architectural Features Enabling OS Services

OS Service	HW Support
Protection and Security	Kernel/user mode, protected instructions, base/limit registers
System calls	Trap instructions and interrupt vectors
Exception handling	Trap instructions and interrupt vectors
I/O operations	Trap instructions, interrupt vectors, and memory mapping
Scheduling	Timer
Synchronization	Atomic instructions
Virtual memory	Translation Look-aside Buffer (TLB)

Architectural Features Enabling OS Services

OS Service	HW Support
Protection and Security	Kernel/user mode, protected instructions, base/limit registers
System calls	Trap instructions and interrupt vectors
Exception handling	Trap instructions and interrupt vectors
I/O operations	Trap instructions, interrupt vectors, and memory mapping
Scheduling	Timer
Synchronization	Atomic instructions
Virtual memory	Translation Look-aside Buffer (TLB)

Privileged Instructions

- Some CPU instructions are more sensitive than others
  - `MOV %eax, %ebx` → move the content of the register `ebx` into `eax`
  - `MOV %eax, [%ebx]` → move the content of memory indexed by register `ebx` to `eax`
  - `HLT` → halt the system
  - `INT X` → generate interrupt `X`

Privileged Instructions

- Some CPU instructions are more sensitive than others
  - `MOV %eax, %ebx` → move the content of the register `ebx` into `eax`
  - `MOV %eax, [%ebx]` → move the content of memory indexed by register `ebx` to `eax`
  - `HLT` → halt the system
  - `INT X` → generate interrupt `X`

Privileged Instructions

- Some CPU instructions are more sensitive than others
  - `MOV %eax, %ebx` → move the content of the register `ebx` into `eax`
  - `MOV %eax, [%ebx]` → move the content of memory indexed by register `ebx` to `eax`
  - `HLT` → halt the system
  - `INT X` → generate interrupt `X`

Idea:
privileged instructions can be executed **only** by the OS

Kernel vs. User Mode

- 2 different "states" while executing CPU instructions

Kernel vs. User Mode

- 2 different "states" while executing CPU instructions
- **Kernel mode** is unrestricted:
  - The OS can perform any instruction (including privileged ones)

Kernel vs. User Mode

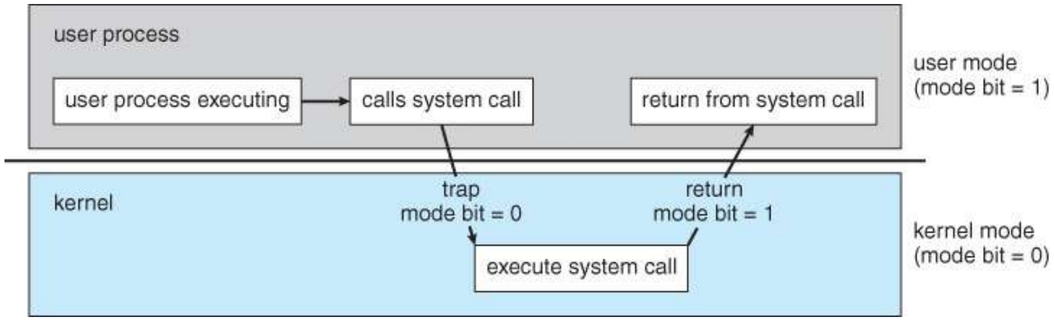
- 2 different "states" while executing CPU instructions
- **Kernel mode** is unrestricted:
  - The OS can perform any instruction (including privileged ones)
- **User mode** is restricted so that the user is **not** able to:
  - Address I/O (directly)
  - Manipulate the content of main memory
  - Halt the machine
  - Switch to kernel mode
  - etc.

Kernel vs. User Mode

- 2 different "states" while executing CPU instructions
- **Kernel mode** is unrestricted:
  - The OS can perform any instruction (including privileged ones)
- **User mode** is restricted so that the user is **not** able to:
  - Address I/O (directly)
  - Manipulate the content of main memory
  - Halt the machine
  - Switch to kernel mode
  - etc.

Implemented in HW!
A status bit stored in a protected CPU register (`0`=kernel, `1`=user)

Kernel vs. User Mode



System calls are just **one** way to "trap into" kernel mode

We'll soon talk about system calls

The idea is the same!
The control is transferred from the user to the system

Beyond Kernel vs. User Mode

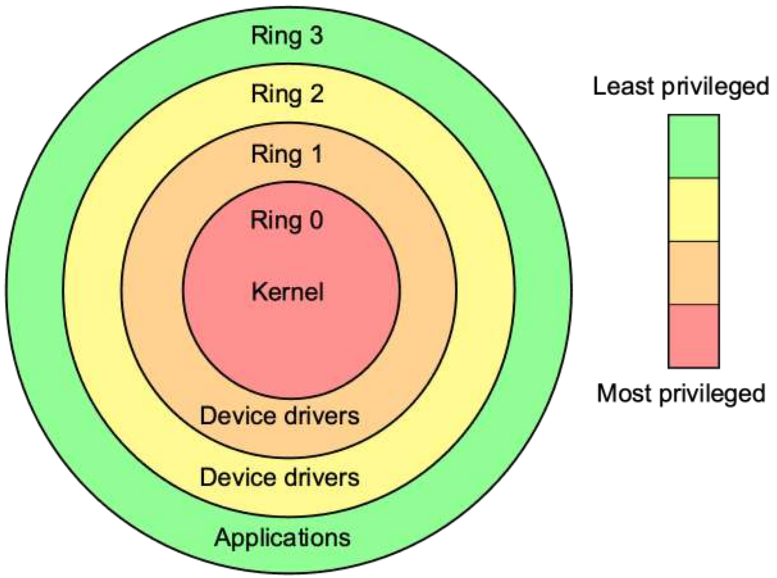
- The underlying HW must support at least kernel and user mode

Beyond Kernel vs. User Mode

- The underlying HW must support at least kernel and user mode
- More fine-grained solutions are also possible

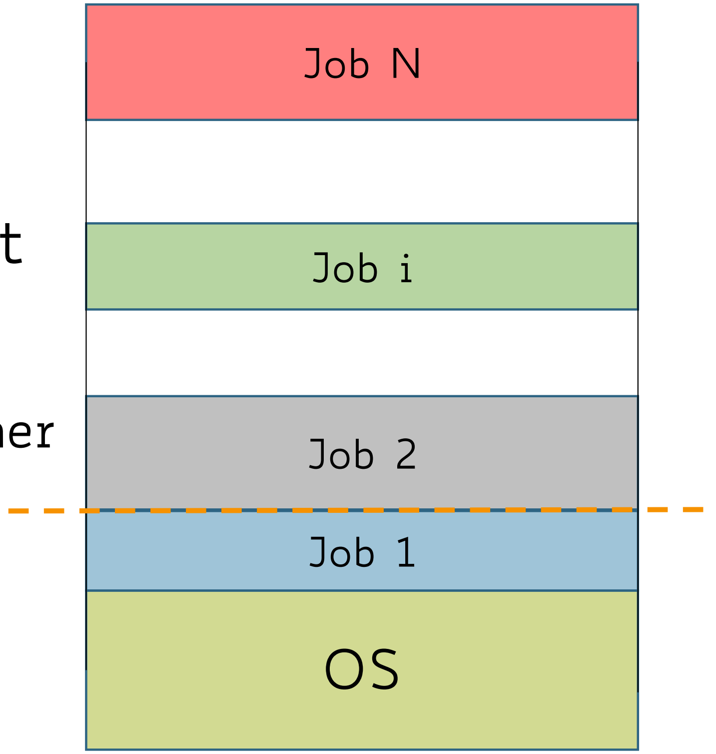
Beyond Kernel vs. User Mode

- The underlying HW must support at least kernel and user mode
- More fine-grained solutions are also possible
- **Protection Rings**
  - 4 different privilege levels {0, …, 3}
  - Still implementable in HW (2 bits)



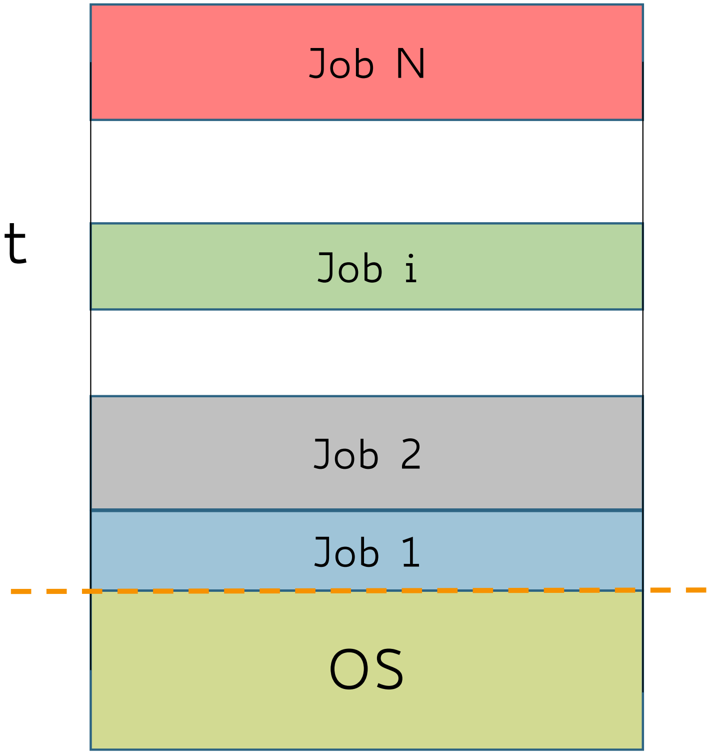
Memory Protection

- Architecture must provide support for the OS to:
  - Protect user programs from each other



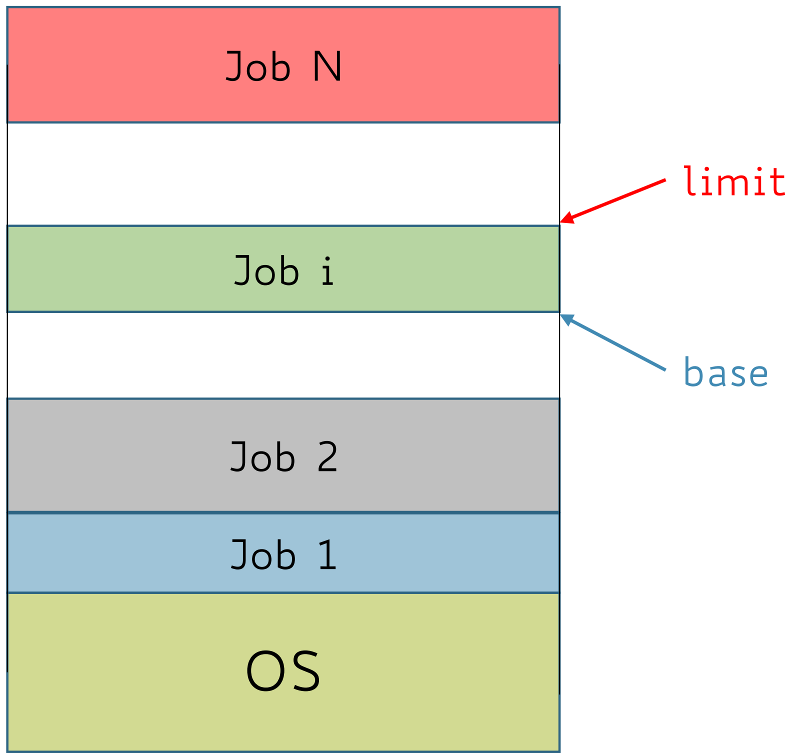
Memory Protection

- Architecture must provide support for the OS to:
  - Protect the OS from user programs



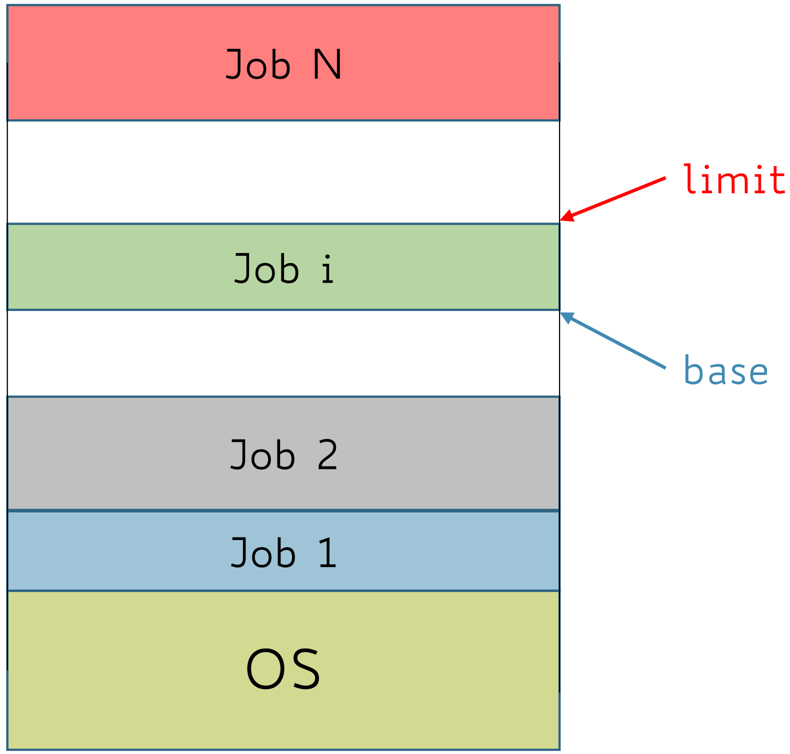
Memory Protection

- The simplest technique is to have **2 dedicated registers**
  - base → contains the starting valid memory address
  - limit → contains the last valid memory address



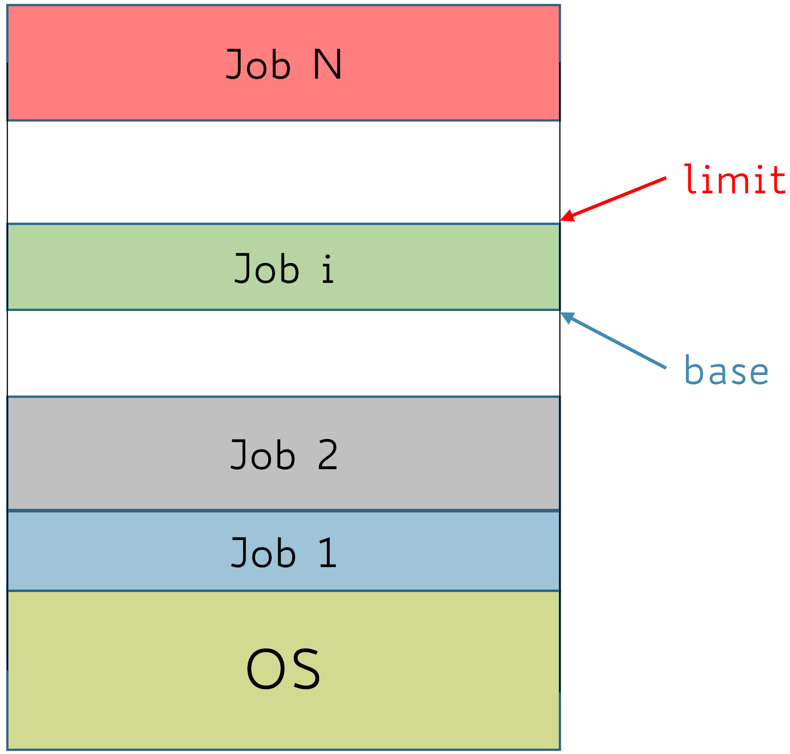
Memory Protection

- The simplest technique is to have **2 dedicated registers**
  - base → contains the starting valid memory address
  - limit → contains the last valid memory address
- The OS loads the base and limit registers upon program startup



Memory Protection

- The simplest technique is to have **2 dedicated registers**
  - base → contains the starting valid memory address
  - limit → contains the last valid memory address
- The OS loads the base and limit registers upon program startup
- The CPU checks every memory address referenced by user program falls between base and limit values



Architectural Features Enabling OS Services

OS Service	HW Support
Protection and Security	Kernel/user mode, protected instructions, base/limit registers
System calls	Trap instructions and interrupt vectors
Exception handling	Trap instructions and interrupt vectors
I/O operations	Trap instructions, interrupt vectors, and memory mapping
Scheduling	Timer
Synchronization	Atomic instructions
Virtual memory	Translation Look-aside Buffer (TLB)

Preserving Usability: System Calls

- Privileged instructions cannot be executed in user mode **directly**

Preserving Usability: System Calls

- Privileged instructions cannot be executed in user mode **directly**
- But programs running in user mode can ask the OS to perform some restricted operations on their behalf (in kernel mode)

Preserving Usability: System Calls

- Privileged instructions cannot be executed in user mode **directly**
- But programs running in user mode can ask the OS to perform some restricted operations on their behalf (in kernel mode)
- For example, a user program may require to:
  - write data to a file stored on disk
  - send data over the network interface
  - etc.

Preserving Usability: System Calls

- Privileged instructions cannot be executed in user mode **directly**
- But programs running in user mode can ask the OS to perform some restricted operations on their behalf (in kernel mode)
- For example, a user program may require to:
  - write data to a file stored on disk
  - send data over the network interface
  - etc.

Crossing protection boundaries using **system calls**

Exceptions and Interrupts

- Exceptions
  - software-generated
  - e.g., program error like division by 0

Exceptions and Interrupts

- Exceptions
    - software-generated
    - e.g., program error like division by 0
- Interrupts
    - hardware-generated (by external devices)
    - e.g., I/O completion or timer interrupt on a multi-tasking system

A Quick Note on Terminology

TRAP

A Quick Note on Terminology

TRAP

A Quick Note on Terminology

TRAP

A Quick Note on Terminology

TRAP

We will refer to **trap** as any event that causes switch to OS kernel mode

A Quick Note on Terminology

TRAP

SYSTEM CALL
(software trap)

synchronous
SW-initiated
request an OS
service
e.g., `SYSCALL`, `INT`
on the x86

A Quick Note on Terminology

TRAP

SYSTEM CALL
(software trap)

synchronous
SW-initiated
request an OS
service
e.g., `SYSCALL`, `INT`
on the x86

EXCEPTION
(fault)

synchronous
SW-initiated
response to an
exceptional event
e.g., illegal instruction,
protection fault, divide-
by-0, page fault, etc.

A Quick Note on Terminology

TRAP

SYSTEM CALL (software trap)

synchronous
SW-initiated
request an OS service
e.g., `SYSCALL`, `INT` on the x86

EXCEPTION (fault)

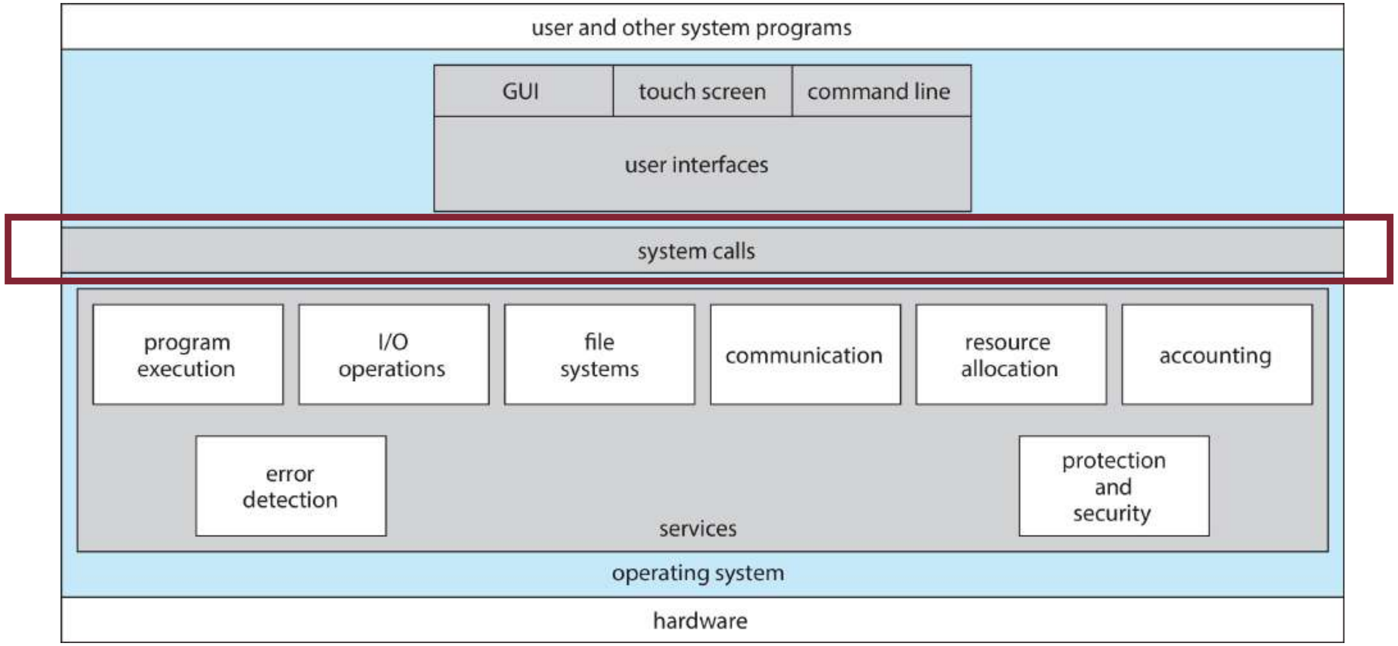
synchronous
SW-initiated
response to an exceptional event
e.g., illegal instruction, protection fault, divide-by-0, page fault, etc.

INTERRUPT

asynchronous
HW-initiated

e.g., network packet arrived, I/O completion, timer

User Programs-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

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

System Calls: Process Control

- create/load/execute/abort/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

- create/load/execute/abort/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

System Calls: Categories

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

System Calls: File Management

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

System Calls: File Management

- create/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

- create/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

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

System Calls: Device Management

- request/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

- request/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

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

System Calls: Information Maintenance

- 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

- 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

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

System Calls: Communication

- 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 (either blocking or non-blocking)
  - 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 (either blocking or non-blocking)
  - Delete the connection when no longer needed

Easier (particularly for inter-computer communications) and 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

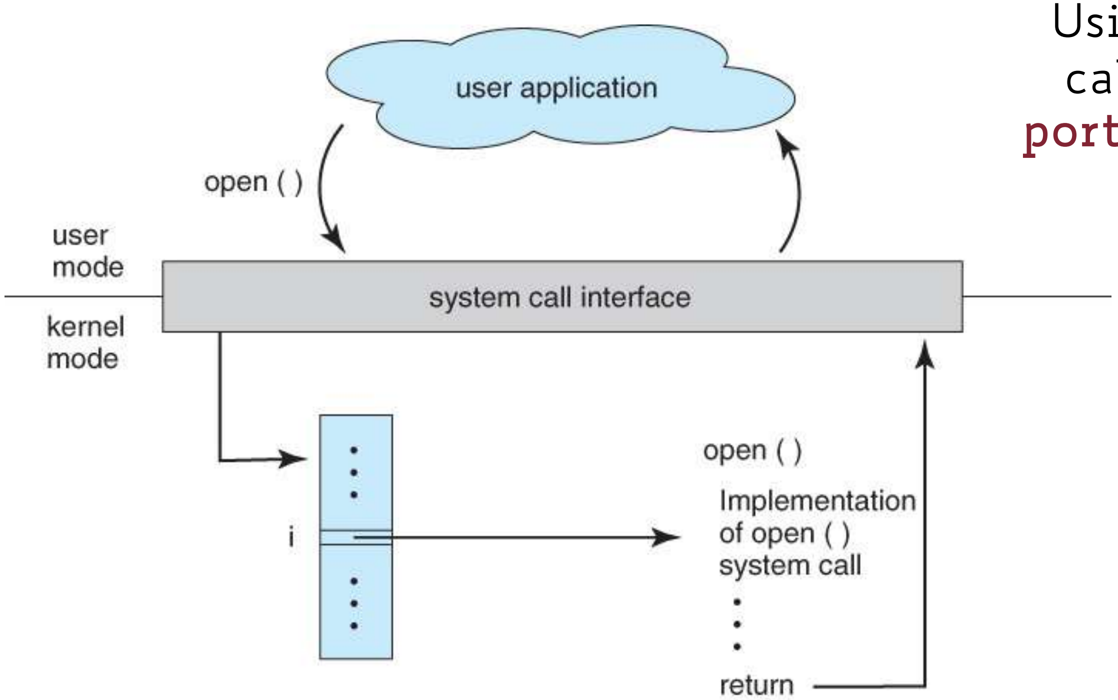
System Calls: Protection

- 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

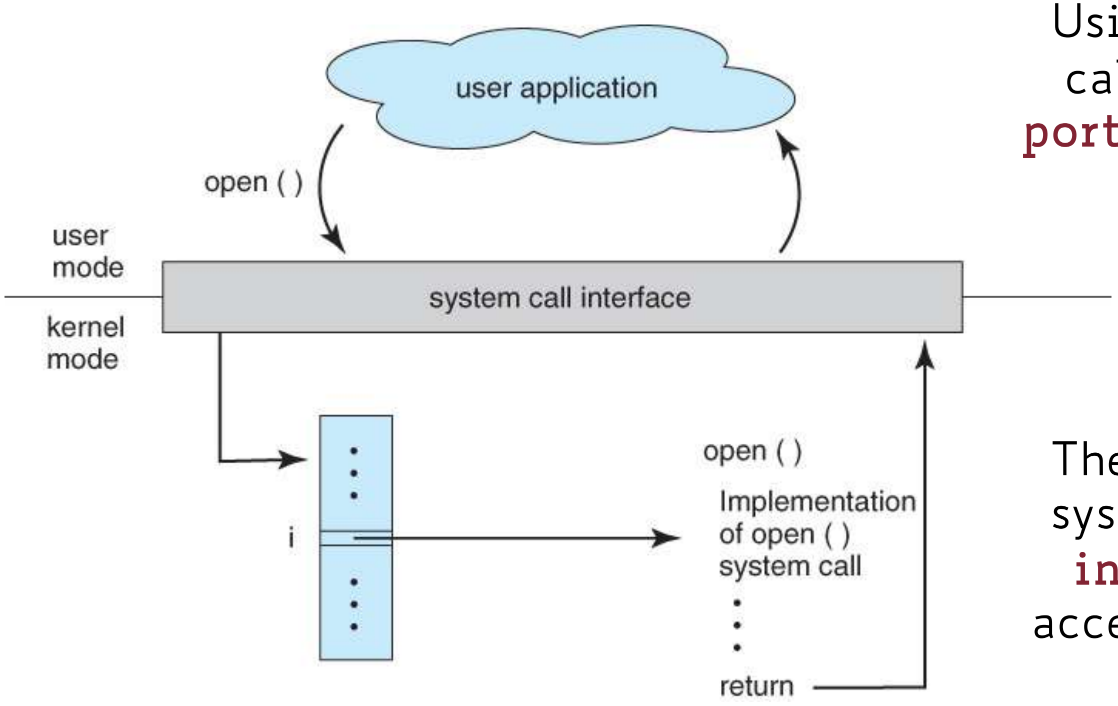
- Provides mechanisms for controlling which users/processes have access to which system resources
- System calls allow the access mechanisms to be adjusted as needed
- Non-priveleged users may temporarily be granted elevated access permissions under specific circumstances
- Crucial in the age of ubiquitous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

The Anatomy of a System Call



Using APIs instead of direct system calls provides for **greater program portability** between different systems

The Anatomy of a System Call



Using APIs instead of direct system calls provides for **greater program portability** between different systems

The API then makes the appropriate system calls through the **system call interface**, using a table lookup to access specific numbered system calls

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

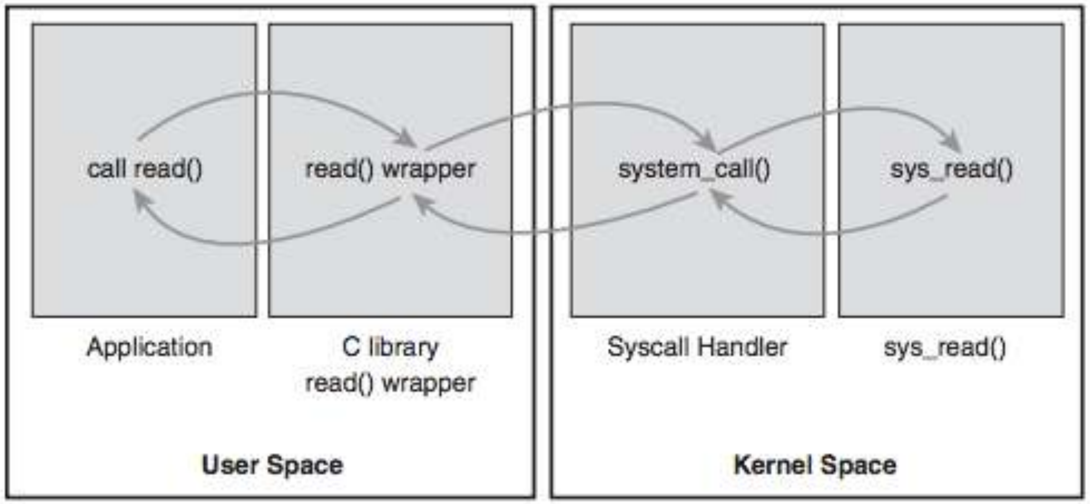
return value | function name | parameters

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

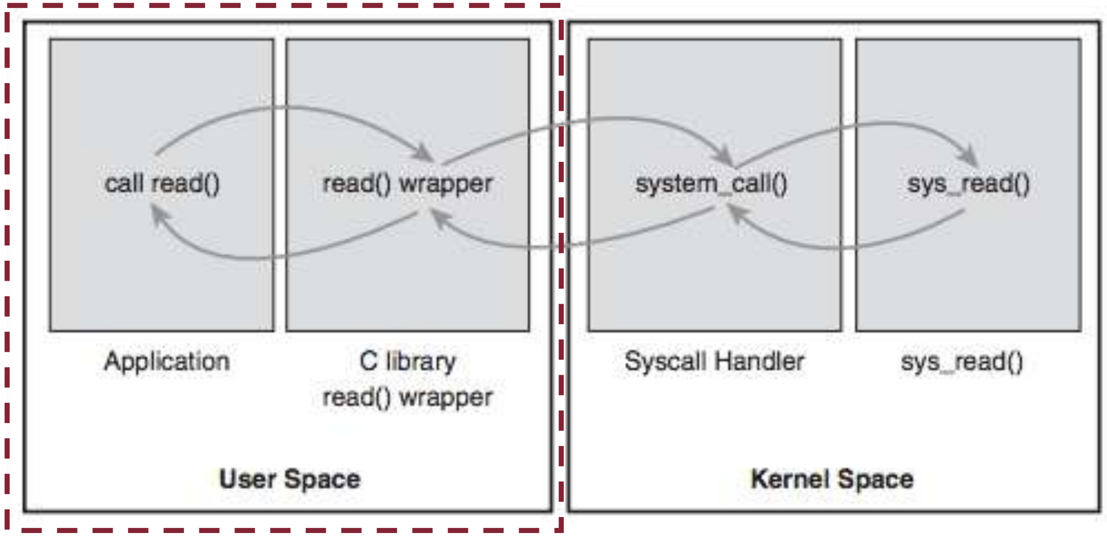
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

System Call: Flow



System Call: Flow

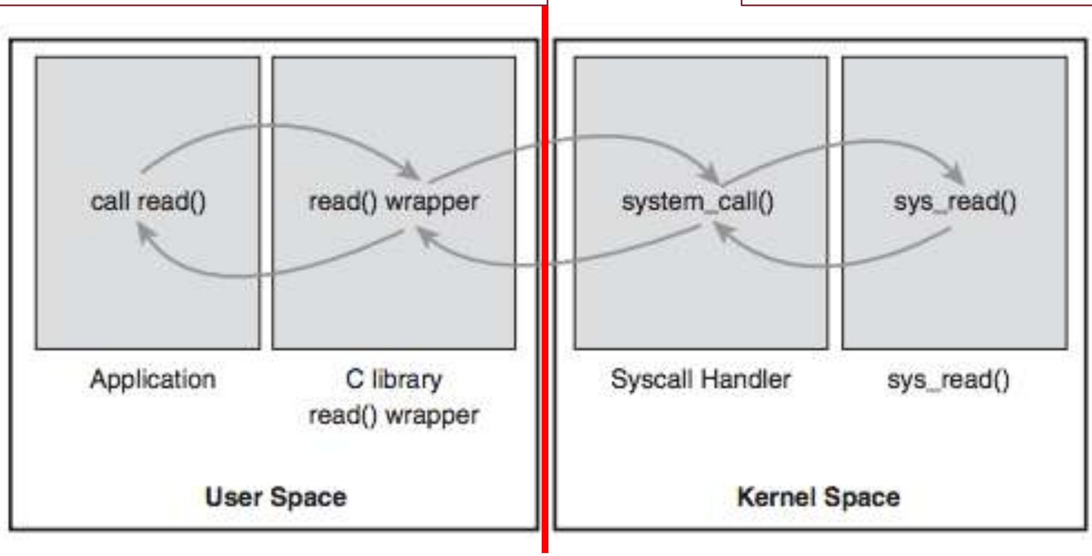
The caller (user program) doesn't have to know how the system call is implemented



System Call: Flow

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

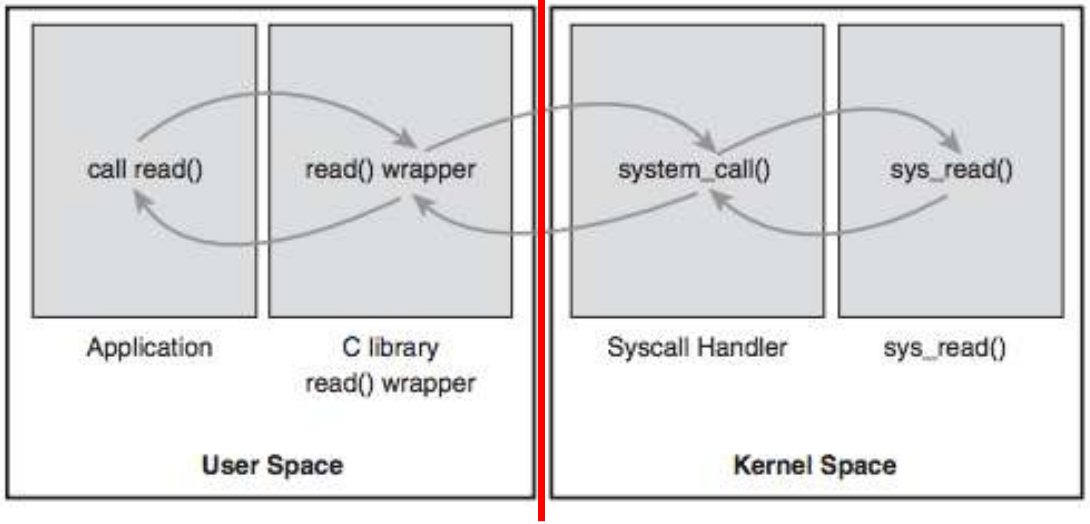
Most of the details are hidden by the API



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
…
```

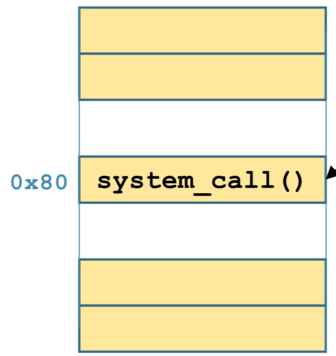
store the number which uniquely identifies the system call requested

System Call Example: Reading from File

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

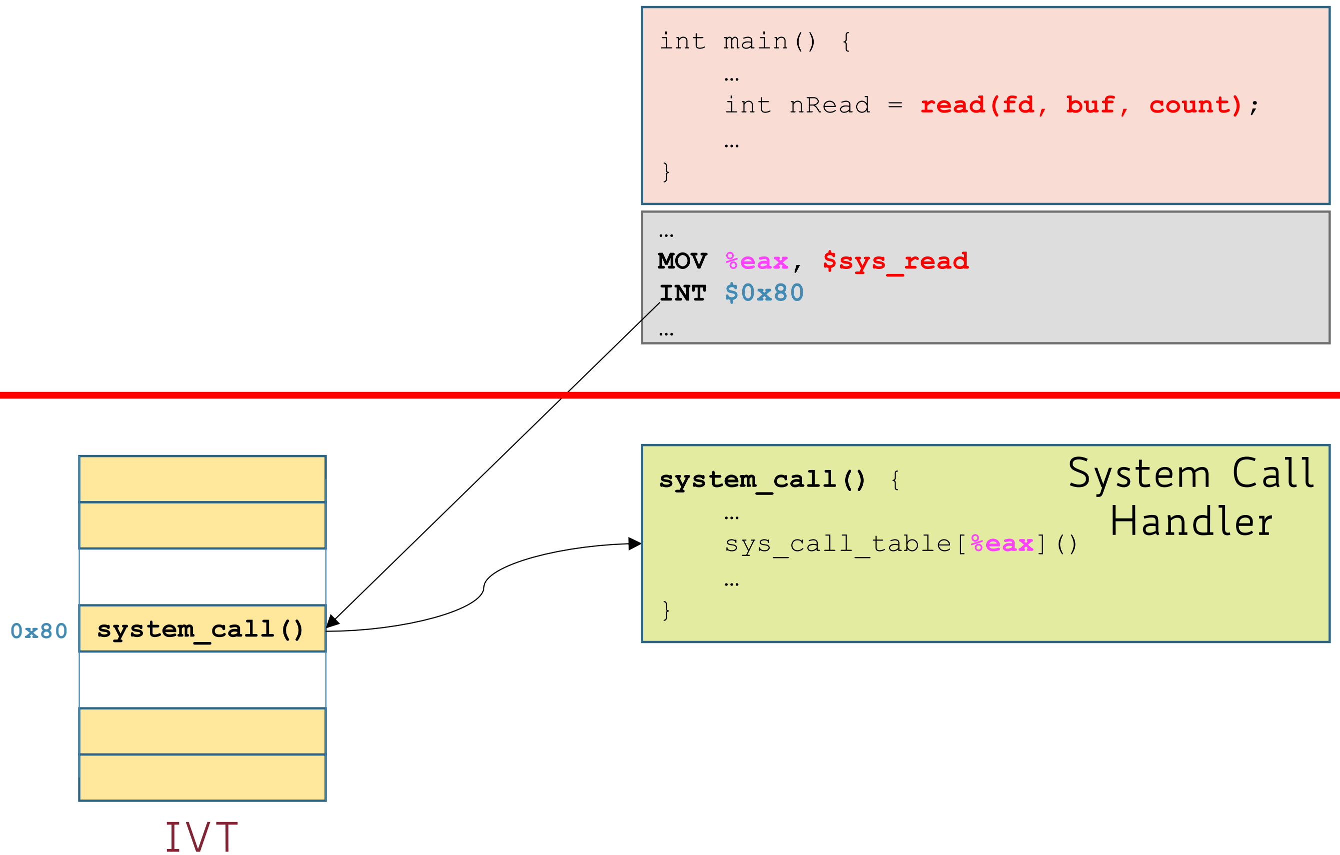
```
…
MOV %eax, $sys_read
INT $0x80
…
```

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

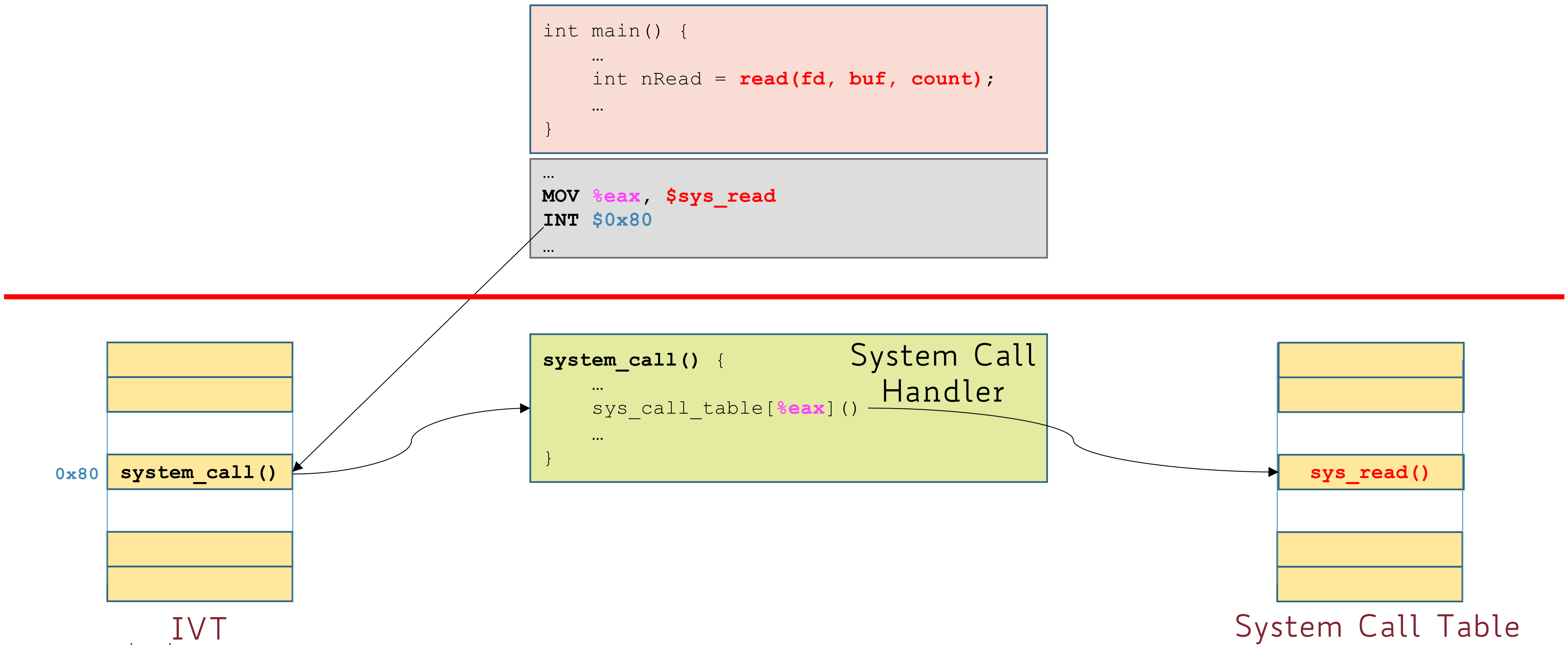


IVT

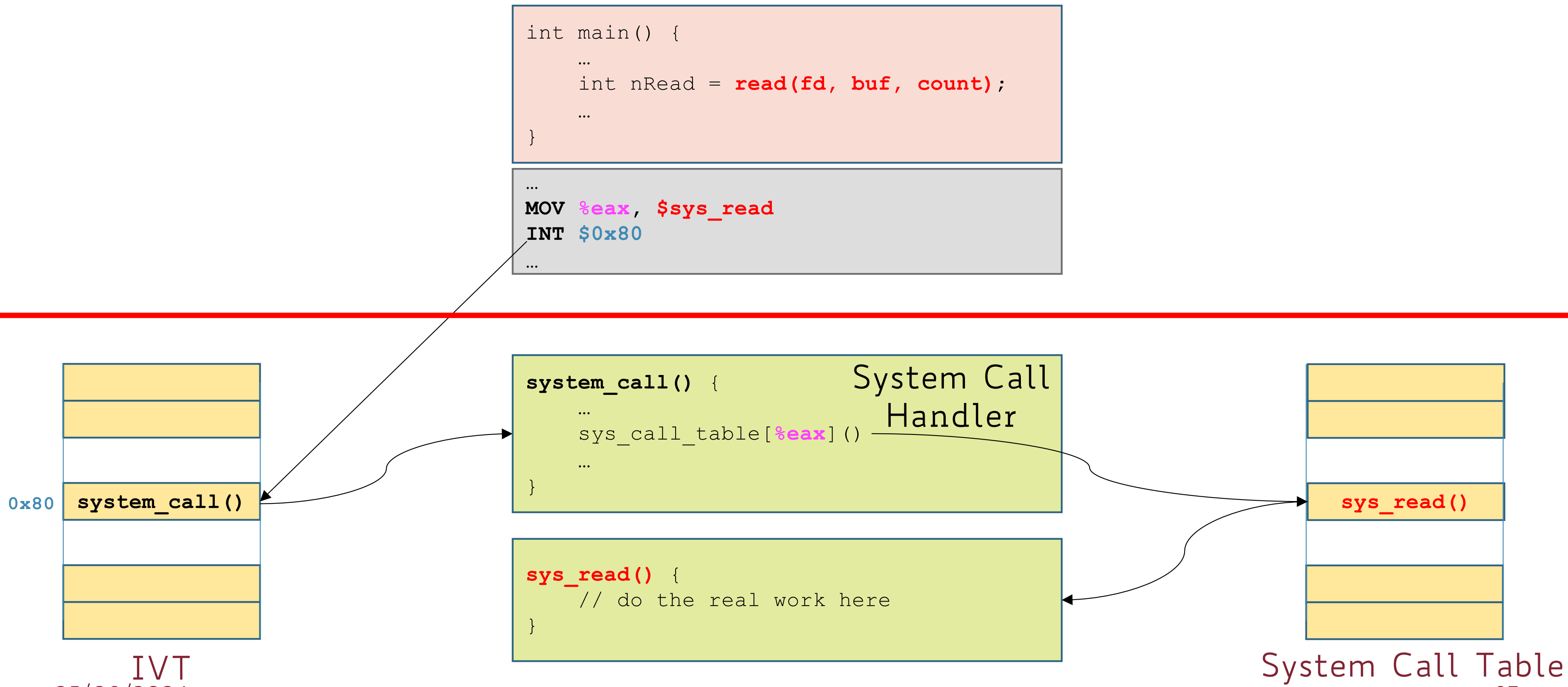
System Call Example: Reading from File



System Call Example: Reading from File



System Call Example: Reading from File



System Call Handler

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

System Call Handler

- The trap caused by system call invocation 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

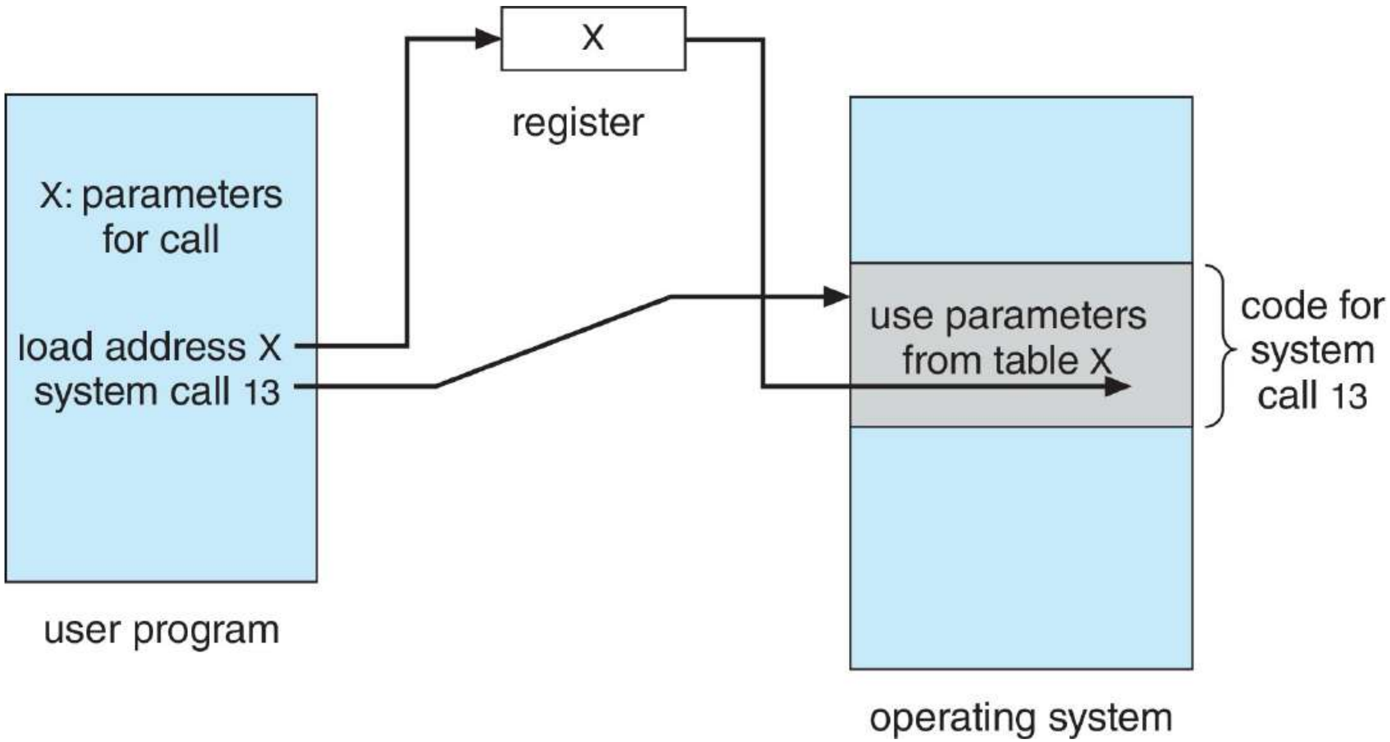
- Often, more information is required than simply the identifier of the desired system call
- **3 methods** used to pass parameters to the OS
  - Store parameters in **registers** (may be more parameters than registers)
  - Store parameters in a **block** or **table** in a dedicated area of memory, and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

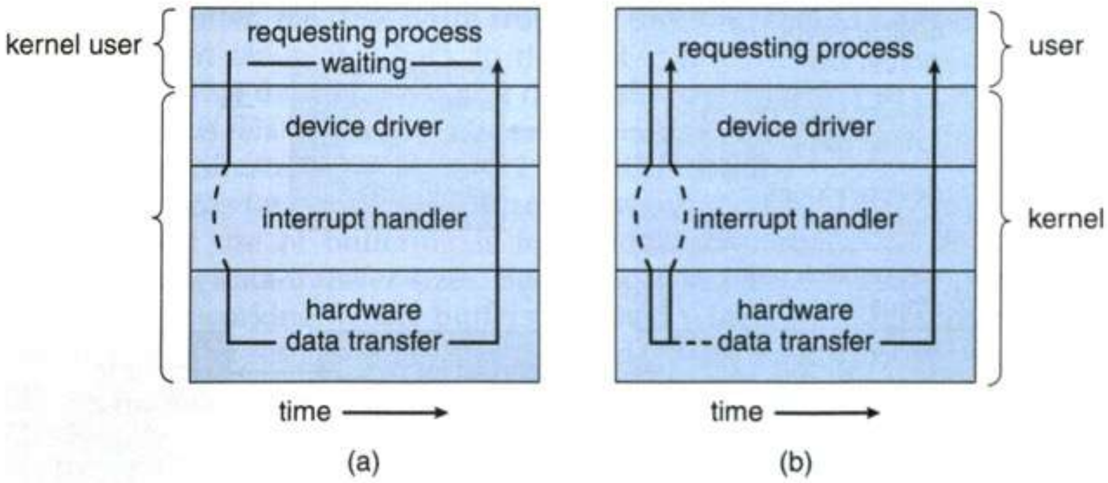
Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table

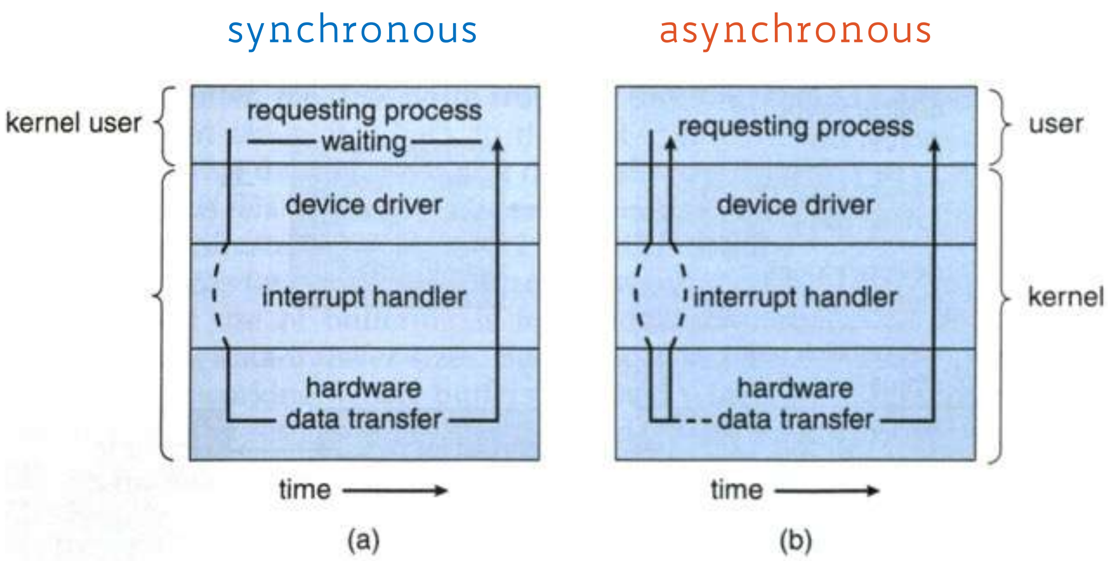


Blocking vs. Non-Blocking I/O

synchronous asynchronous



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()

Architectural Features Enabling OS Services

OS Service	HW Support
Protection and Security	Kernel/user mode, protected instructions, base/limit registers
System calls	Trap instructions and interrupt vectors
Exception handling	Trap instructions and interrupt vectors
I/O operations	Trap instructions, interrupt vectors, and memory mapping
Scheduling	Timer
Synchronization	Atomic instructions
Virtual memory	Translation Look-aside Buffer (TLB)

Timer

- Hardware facility to enable CPU scheduling

Timer

- Hardware facility to enable CPU scheduling
- It is just a clock which marks the time of the day

Timer

- Hardware facility to enable CPU scheduling
- It is just a clock which marks the time of the day
- In multi-tasking systems, allows the CPU not to be monopolized by "selfish" processes

Timer

- Hardware facility to enable CPU scheduling
- It is just a clock which marks the time of the day
- In multi-tasking systems, allows the CPU not to be monopolized by "selfish" processes
- The timer generates an interrupt every, say, 100 microseconds

Timer

- Hardware facility to enable CPU scheduling
- It is just a clock which marks the time of the day
- In multi-tasking systems, allows the CPU not to be monopolized by "selfish" processes
- The timer generates an interrupt every, say, 100 microseconds
- At each timer interrupt, the CPU scheduler takes over and decides which process to execute next

Atomic Instructions

- Interrupts may occur at any time and interfere with running processes

Atomic Instructions

- Interrupts may occur at any time and interfere with running processes
- OS must be able to synchronize the activities of cooperating, concurrent processes

Atomic Instructions

- Interrupts may occur at any time and interfere with running processes
- OS must be able to synchronize the activities of cooperating, concurrent processes
- Hardware must ensure that short sequences of instructions (e.g., read-modify-write) are executed **atomically** by either:
  - Disabling interrupts before the sequence and re-enable them afterwards

  or
  - Special instructions that are natively executed atomically

Architectural Features Enabling OS Services

OS Service	HW Support
Protection and Security	Kernel/user mode, protected instructions, base/limit registers
System calls	Trap instructions and interrupt vectors
Exception handling	Trap instructions and interrupt vectors
I/O operations	Trap instructions, interrupt vectors, and memory mapping
Scheduling	Timer
Synchronization	Atomic instructions
Virtual memory	Translation Look-aside Buffer (TLB)

What is Virtual Memory?

- An `abstraction` (of the actual, physical main memory)

What is Virtual Memory?

- An **abstraction** (of the actual, physical main memory)
- It gives each process the illusion that physical memory is just a contiguous address space (virtual address space)

What is Virtual Memory?

- An **abstraction** (of the actual, physical main memory)
- It gives each process the illusion that physical memory is just a contiguous address space (virtual address space)
- It allows to run programs without them being entirely loaded in main memory
  - They are entirely loaded in virtual memory, though!

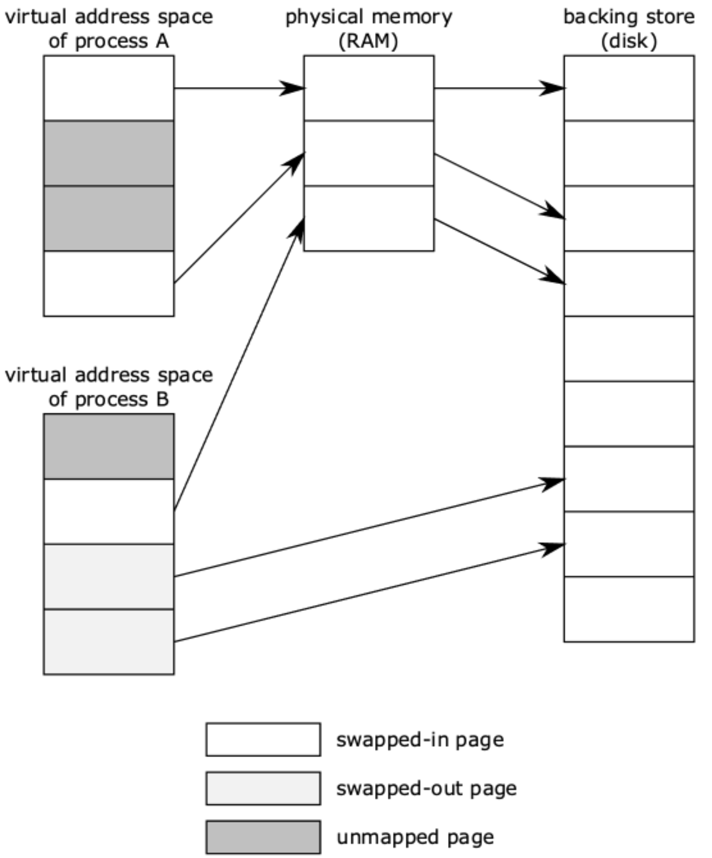
What is Virtual Memory?

- An **abstraction** (of the actual, physical main memory)
- It gives each process the illusion that physical memory is just a contiguous address space (virtual address space)
- It allows to run programs without them being entirely loaded in main memory
  - They are entirely loaded in virtual memory, though!
- Implemented both in HW (MMU) and SW (OS)
  - MMU is responsible for translating virtual addresses into physical ones
  - OS is responsible for managing virtual address spaces

Virtual vs. Physical Address Space

- On a 64 bit system the CPU is able to address 2^{64} bytes = 16 exbibytes (EiB)
- Virtual address space ranges from 0 to $2^{64} - 1$
- This is about a billion times more than main memory capacity currently available!
- Virtual address space is typically divided into contiguous blocks of the same size (e.g., 4 KiB), called **pages**
- Pages that are not loaded in main memory are stored on disk

Virtual vs. Physical Address Space



Memory Management Unit (MMU)

- Maps virtual addresses to physical ones through a **page table** managed by the OS

Memory Management Unit (MMU)

- Maps virtual addresses to physical ones through a **page table** managed by the OS
- Uses a cache called **Translation Look-aside Buffer** (TLB) with "recent mappings" for quicker lookups

Memory Management Unit (MMU)

- Maps virtual addresses to physical ones through a **page table** managed by the OS
- Uses a cache called **Translation Look-aside Buffer** (TLB) with "recent mappings" for quicker lookups
- The OS must be aware of which pages are loaded in main memory and which ones are on disk

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* is to be done
  - **mechanism** → *how* to do it

Policy vs. Mechanism

- Decoupling policy logic from the underlying mechanism is a general design principle, as it improves system's:
  - **flexibility** → addition and modification of policies can be easily supported

Policy vs. Mechanism

- Decoupling policy logic from the underlying mechanism is a general design principle, 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

Policy vs. Mechanism

- Decoupling policy logic from the underlying mechanism is a general design principle, 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, 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 (entirely) 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

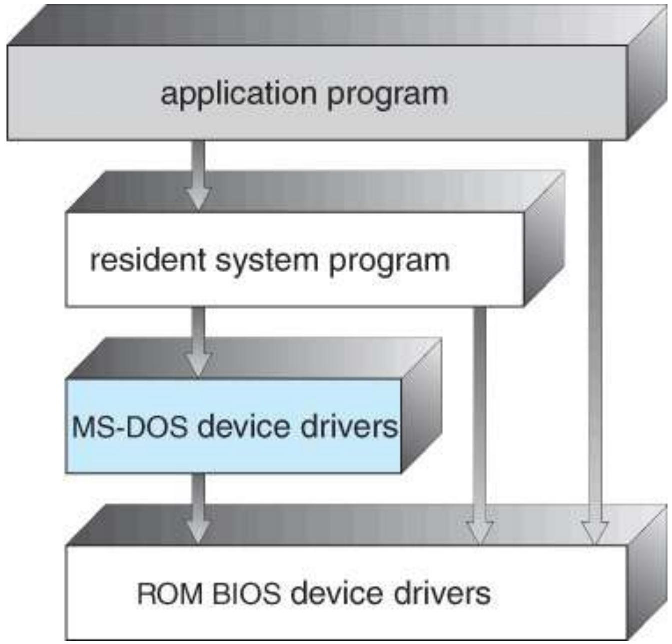
OS Structure

- 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

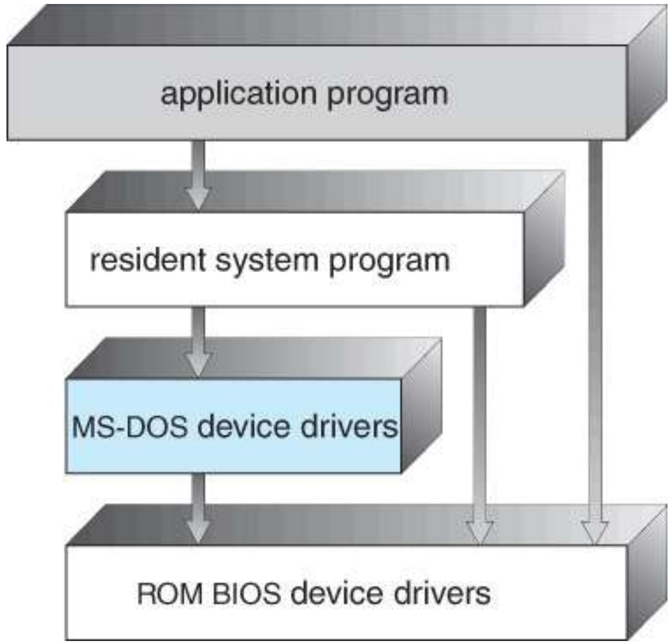


MS-DOS Structure: Simple Structure

No modular subsystems at all!

No separation between user and kernel mode

NOTE
user vs. kernel mode was not supported by the 8088 chip set anyway, so that really wasn't an option back then



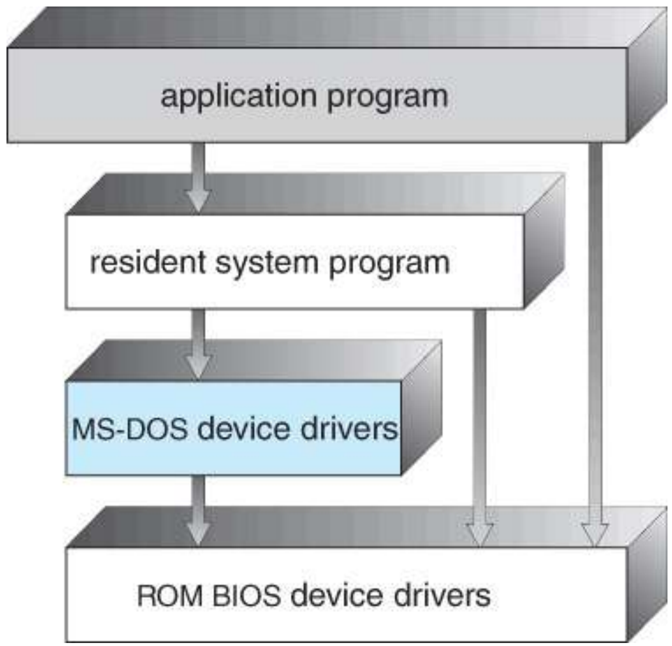
MS-DOS Structure: Simple Structure

No modular subsystems at all!

No separation between user and kernel mode

PROs: easy to implement

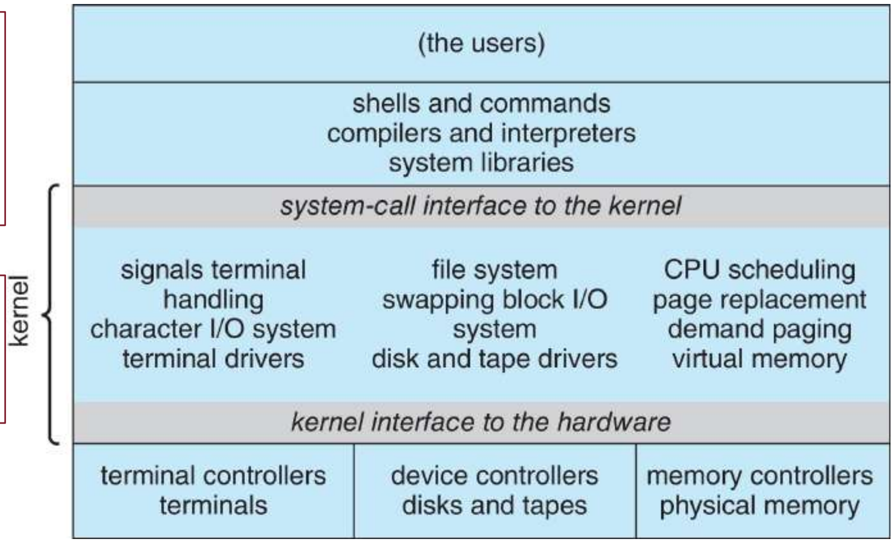
CONs: rigidity, security



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



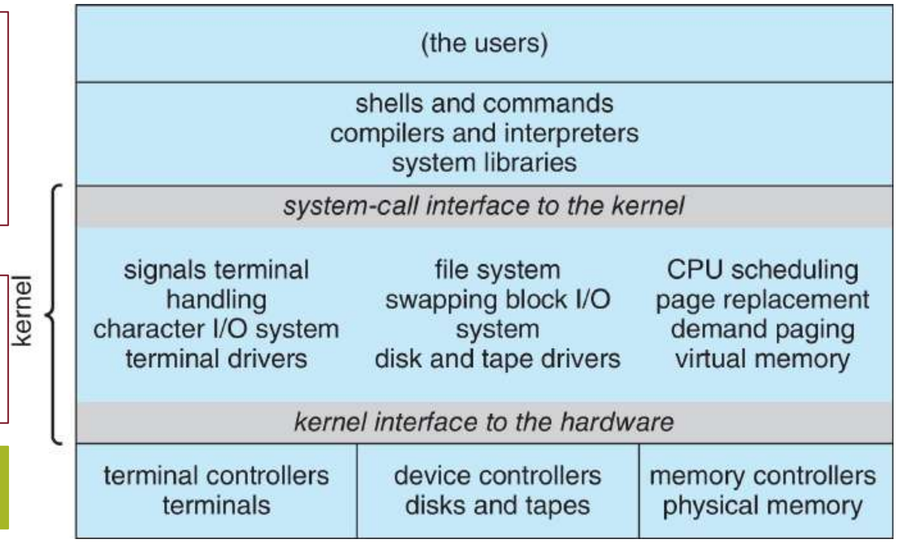
UNIX Structure: Traditional Monolithic Kernel

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

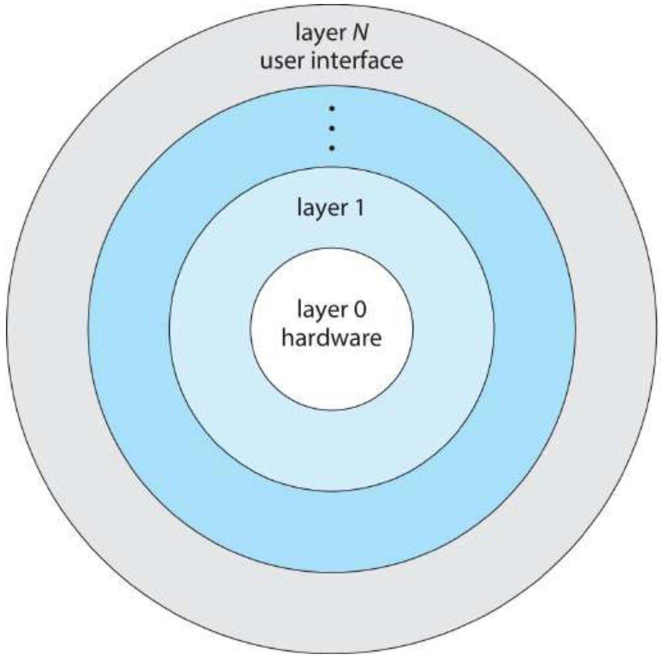
CONs: rigidity, security



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



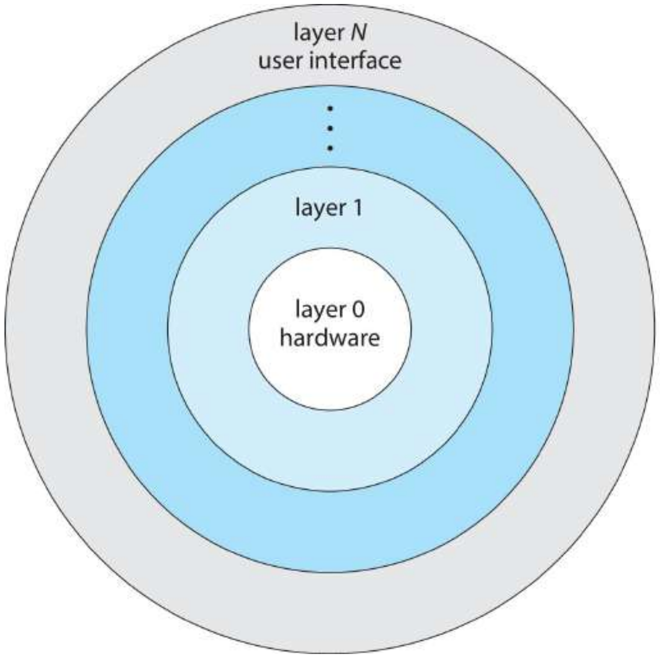
Layered Structure

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

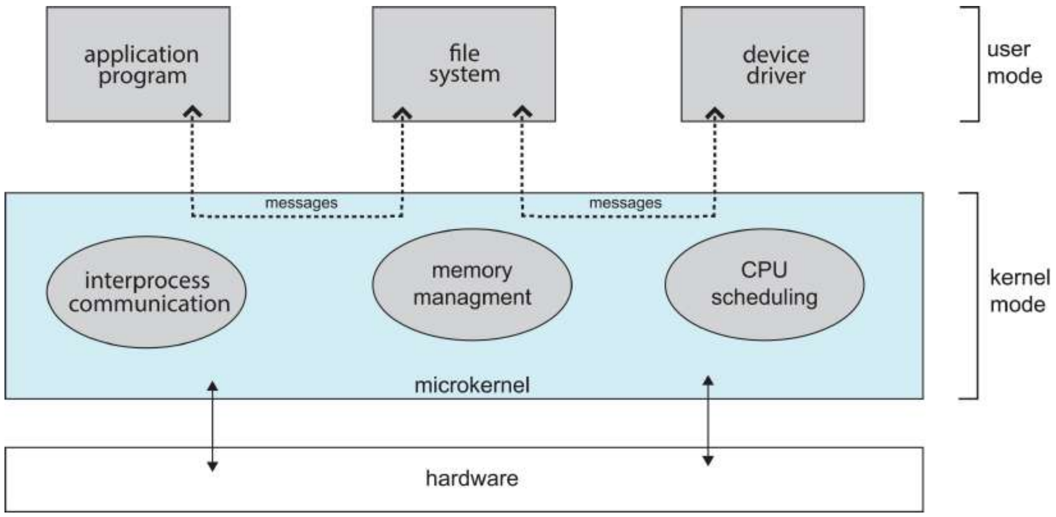
CONs: communication overhead, extra copy



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



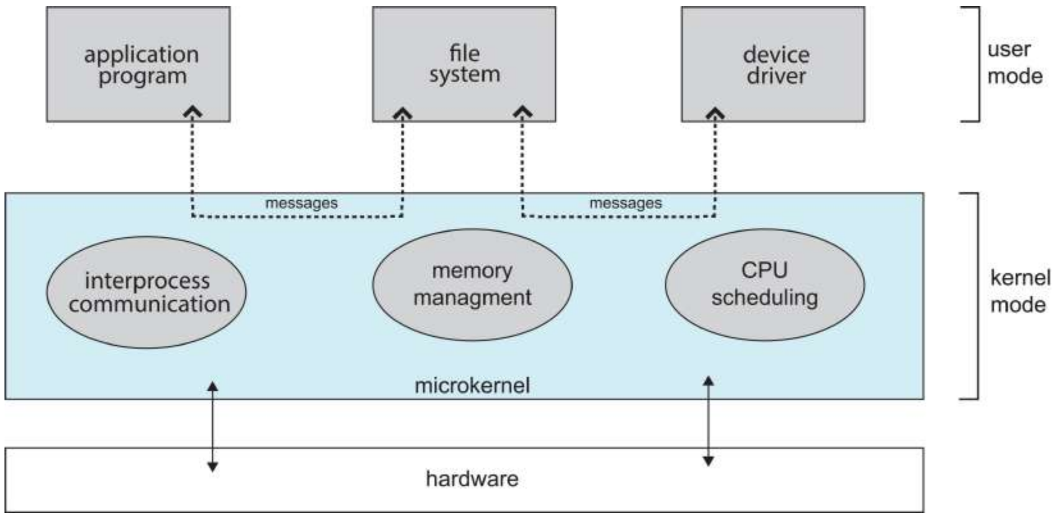
Microkernel Structure

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

PROs: security, reliability, extendibility

CONs: efficiency (message passing)



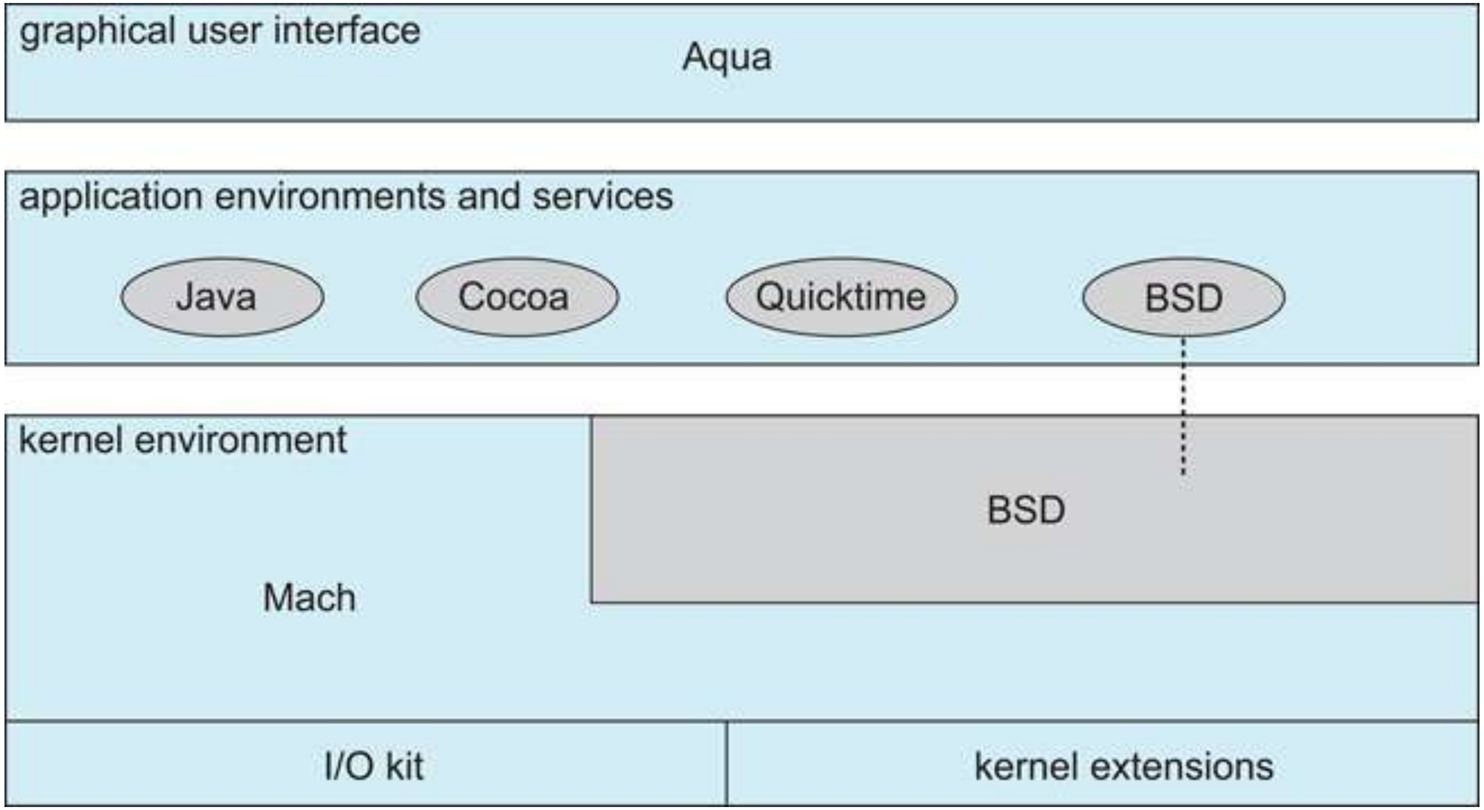
Loadable Kernel Modules (LKMs)

- 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

- 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

Hybrid OS: MacOS X



Summary

- Architecture support is key to OS design

Summary

- Architecture support is key to OS design
- Most of the services provided by the OS to the applications rely on specific HW features

Summary

- Architecture support is key to OS design
- Most of the services provided by the OS to the applications rely on specific HW features
- The OS is tightly coupled to the HW of the host machine

Summary

- Architecture support is key to OS design
- Most of the services provided by the OS to the applications rely on specific HW features
- The OS is tightly coupled to the HW of the host machine
- Several approaches to OS design and implementation

Summary

- Architecture support is key to OS design
- Most of the services provided by the OS to the applications rely on specific HW features
- The OS is tightly coupled to the HW of the host machine
- Several approaches to OS design and implementation
- **Advice:** Keep your Computer Architecture book at hand!