OPERATING SYSTEMS-1
CS 241, SPRING 2020
The kernel Abstraction: Lec 3

Ass. Prof. Ghada Ahmed
ghada_khoriba@fci.helwan.edu.eg

Edited slides,
main reference

Operating systems: principles and practice,
Tom Anderson, 2nd ed

"THERE IS NO SUCH THINGS AS STUPID QUESTION…"

Let us recap what we should know till now

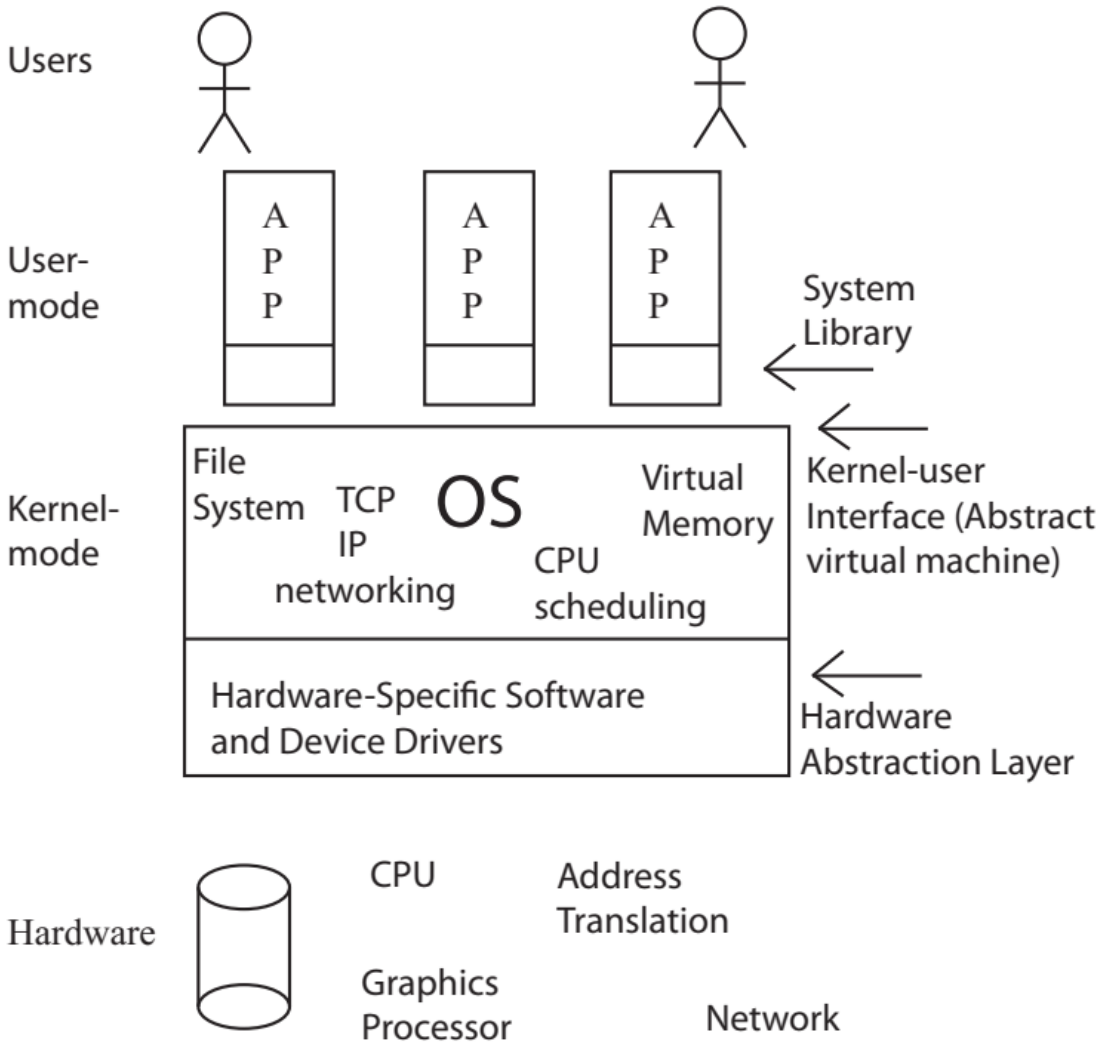
Recap

Recap

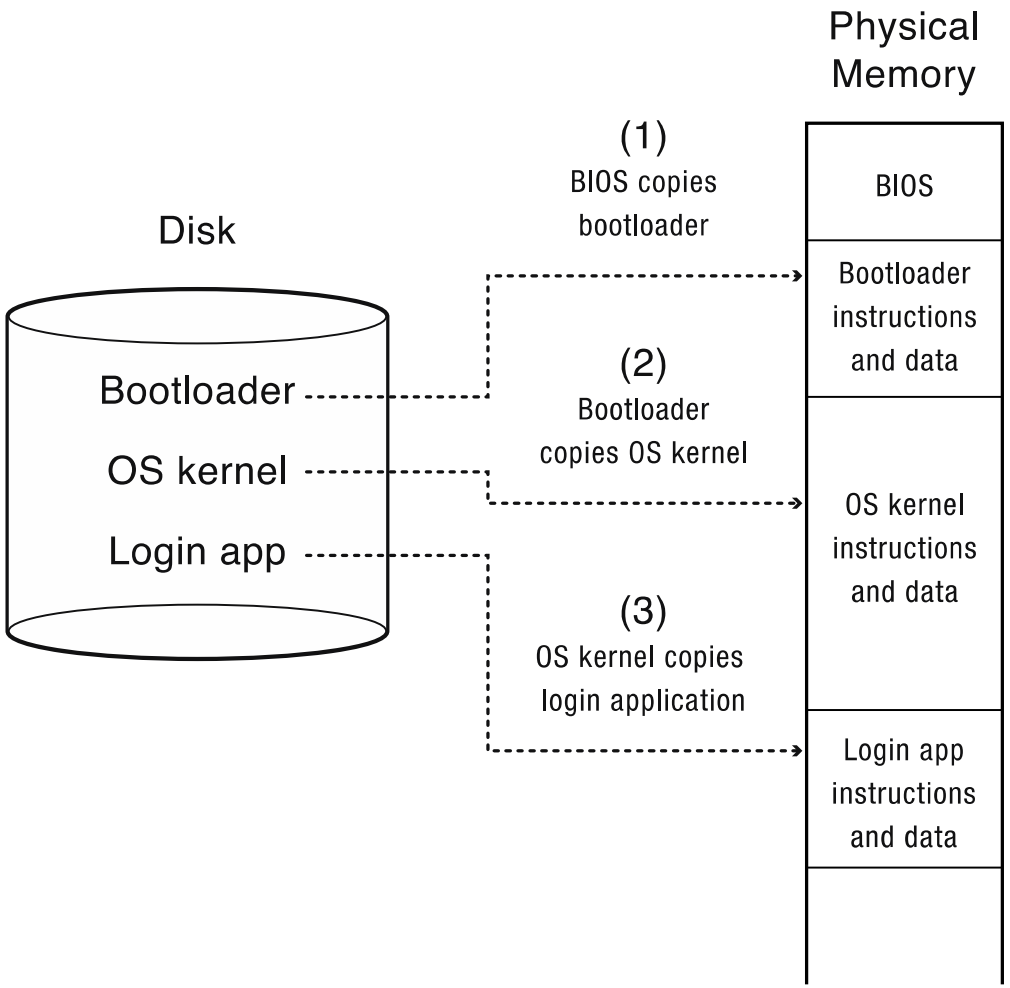
- What is an OS?

Software to manage a computer's resources for its users and applications

- What are the two main responsibilities of OS?
  - Manage hardware resources
  - Provide a clean set of interface to programs

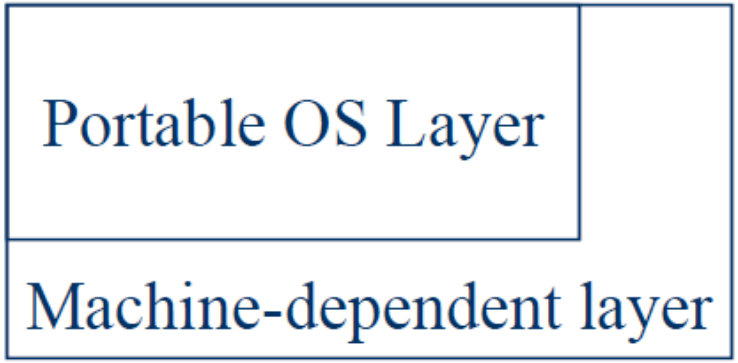


BOOTING

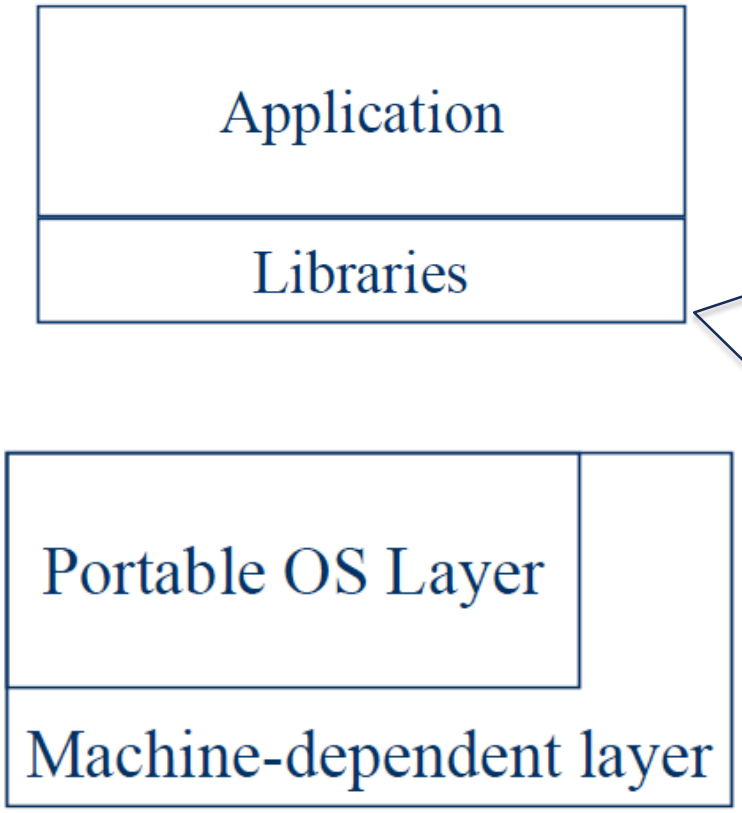

Physical Memory
(1)
BIOS copies bootloader
BIOS
Disk
Bootloader instructions and data
Bootloader
(2)
Bootloader copies OS kernel
OS kernel
OS kernel instructions and data
Login app
(3)
OS kernel copies login application
Login app instructions and data

A PEEK INTO UNIX STRUCTURE

Written by programmer
Compiled by programmer
Uses library calls (e.g., printf)

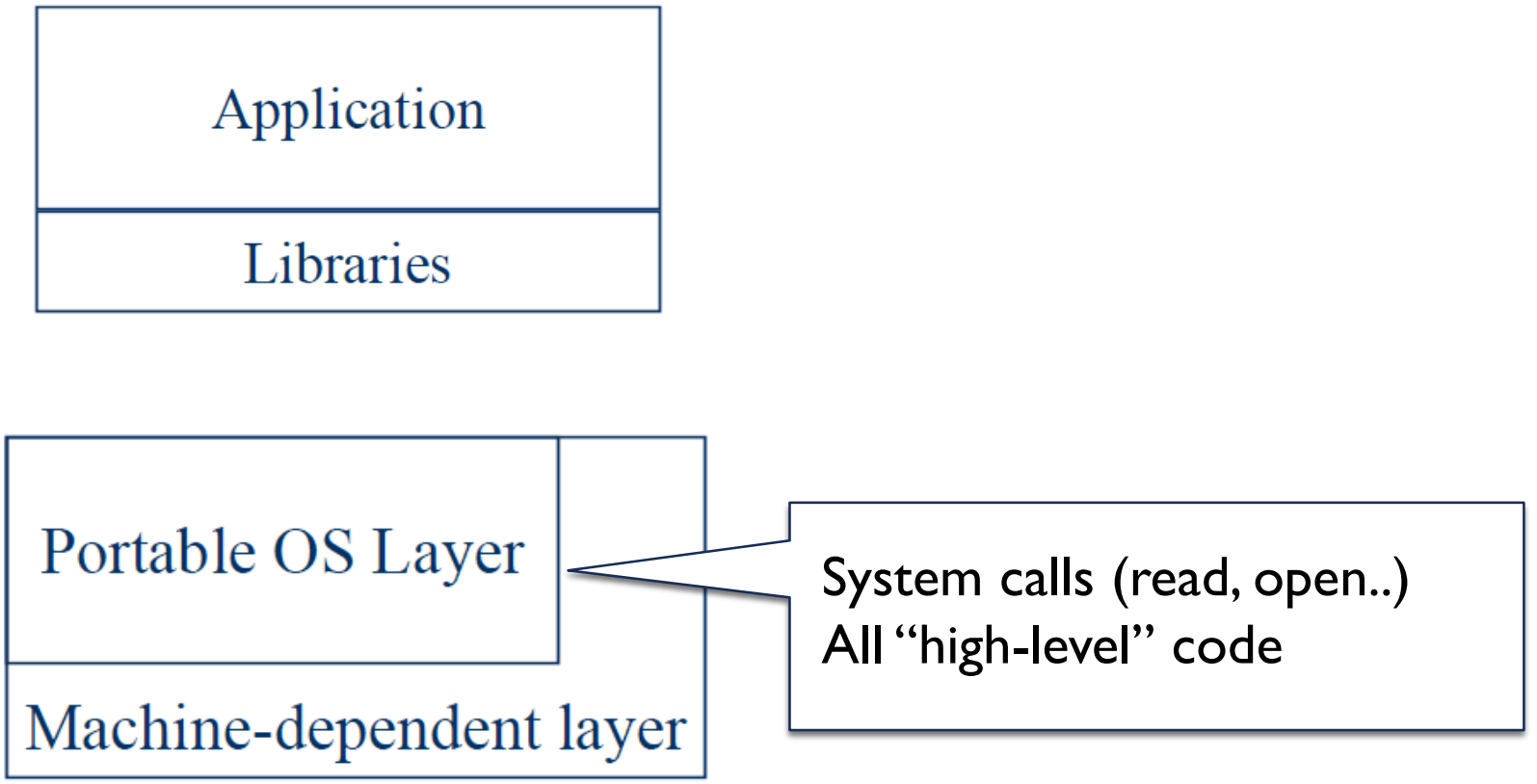


A PEEK INTO UNIX STRUCTURE

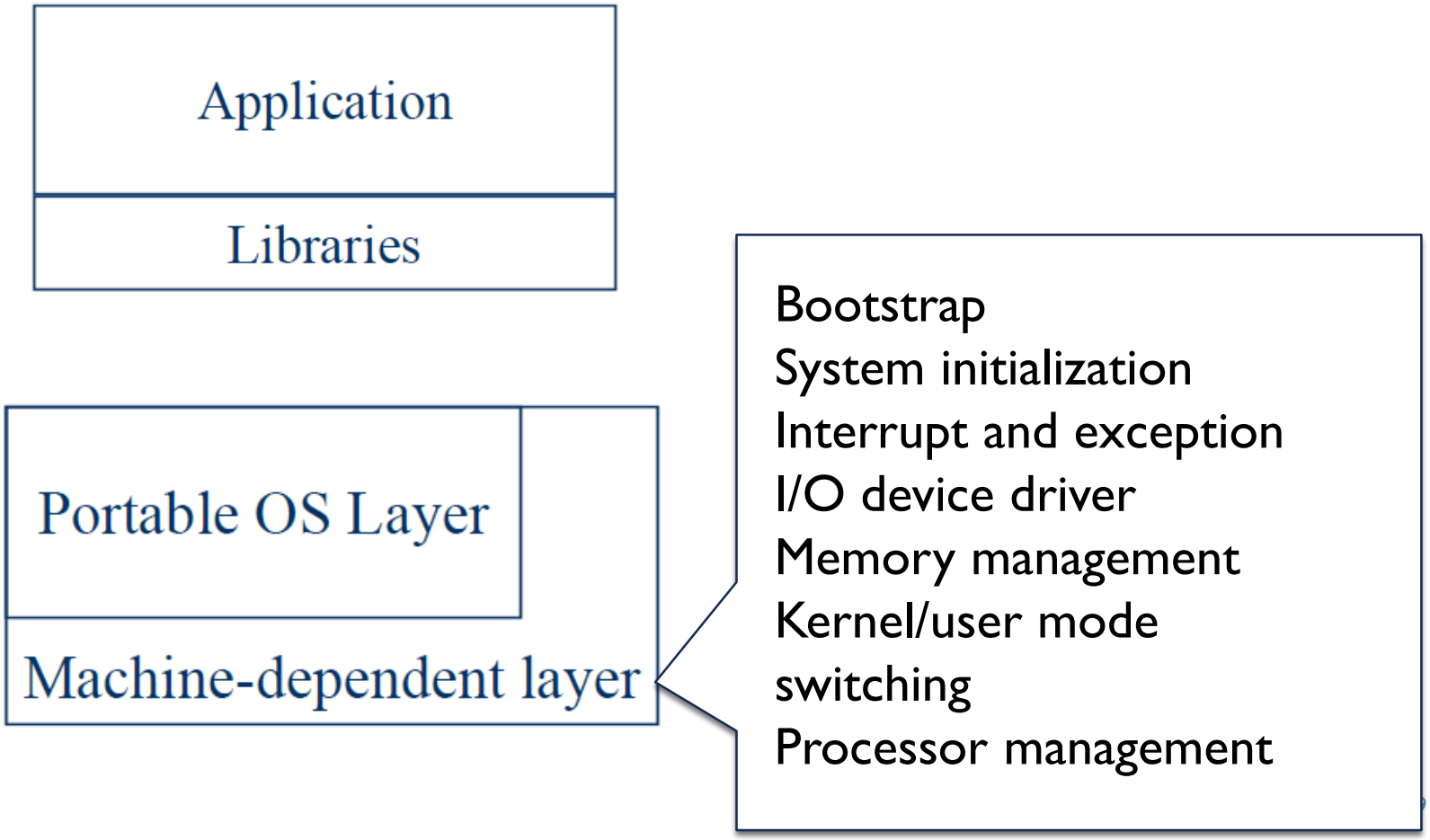


Example: stdio.h
Written by elves
Uses *system calls*
Defined in headers
Input to linker (compiler)
Invoked like functions
May be "resolved" when program is loaded.

A PEEK INTO UNIX STRUCTURE



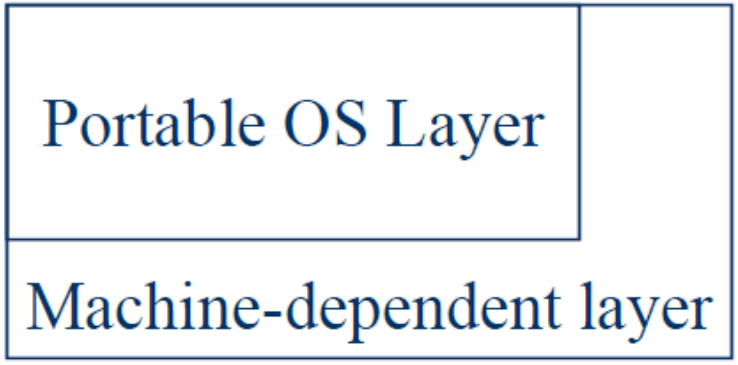
A PEEK INTO UNIX STRUCTURE



A PEEK INTO UNIX STRUCTURE

Cannot execute "protected_instruction", e.g., directly access I/O device

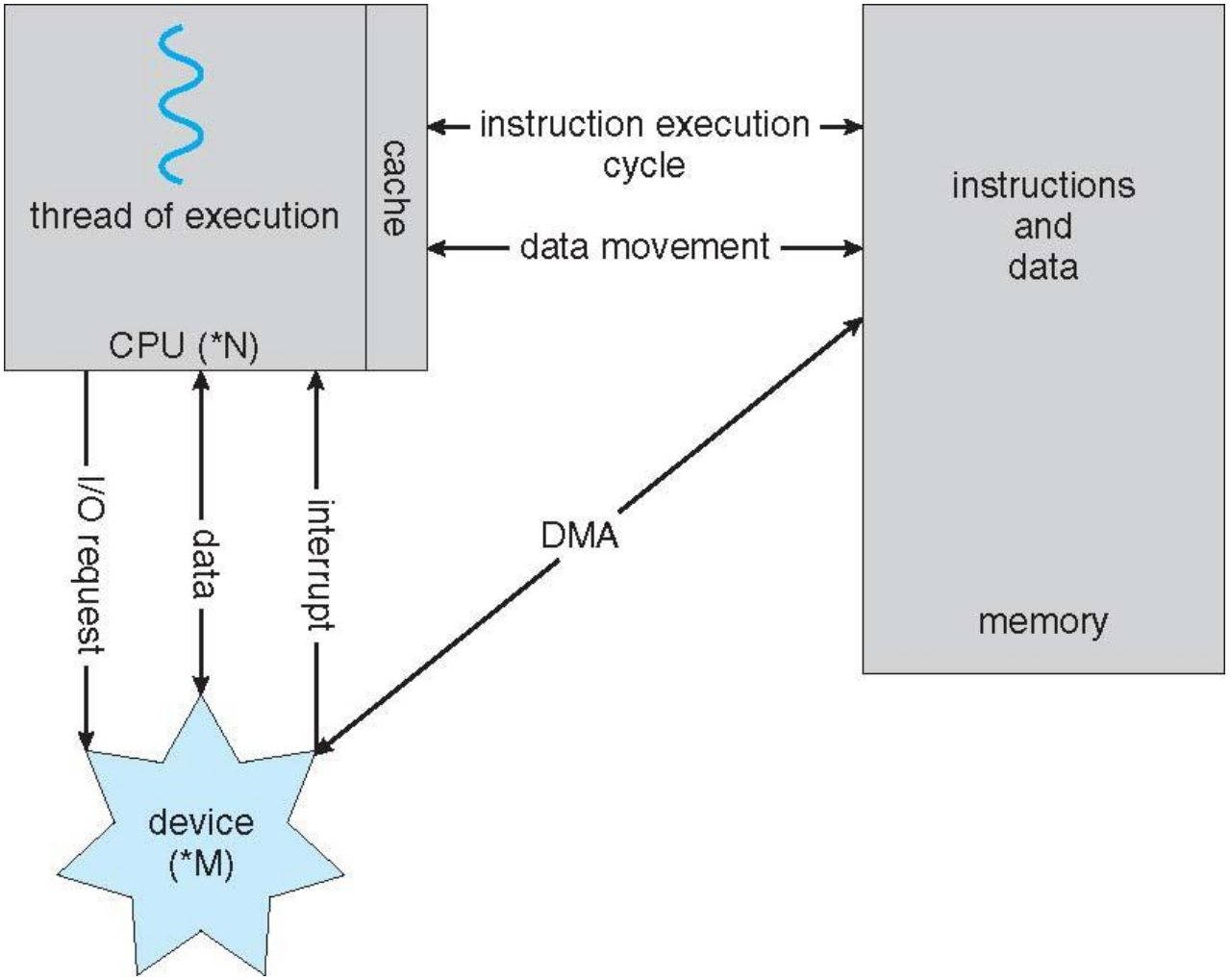
User mode



Kernel mode

- Some systems do not have clear user-kernel boundary
- User/kernel mode is supported by hardware (why?)

DMA: TO ACCESS I/O DEVICE



WHY HARDWARE HAS TO SUPPORT USER/KERNEL MODE?

Imaginary OS code (software-only solution)

```
if ([PC] != protected_instruction)
   execute(PC);
else
   switch_to_kernel_mode();
```

WHY HARDWARE HAS TO SUPPORT USER/KERNEL MODE?

Application's code:

```
lw      $t0, 4($gp)
mult    $t0, $t0, $t0
lw      $t1, 4($gp)
ori     $t2, $zero, 3
mult    $t1, $t1, $t2
add     $t2, $t0, $t1
sw      $t2, 0($gp)
```


OS: check if next instruction is protected instruction.

WHY HARDWARE HAS TO SUPPORT USER/KERNEL MODE?

Application's code:

```
lw      $t0, 4($gp)
mult    $t0, $t0, $t0
lw      $t1, 4($gp)
ori     $t2, $zero, 3
mult    $t1, $t1, $t2
add     $t2, $t0, $t1
sw      $t2, 0($gp)
```

OS: check if next instruction is protected instruction.

- Performance overhead is too big: OS needs to check every instruction of the application!
  - *Simulators*

WHY HARDWARE HAS TO SUPPORT USER/KERNEL MODE?

OS: set-up the environment; load the application

Application's code:

```
lw      $t0, 4($gp)
mult    $t0, $t0, $t0
lw      $t1, 4($gp)
ori     $t2, $zero, 3
mult    $t1, $t1, $t2
add     $t2, $t0, $t1
sw      $t2, 0($gp)
```

- Instead, what we really want is to give the CPU entirely to the application

Return to OS after termination; OS: schedule next application to execute..

- *Any problems?*
- *How can OS check if application executes protected instruction?*
  - *How can OS know it will **ever** run again?*

INTERRUPTS

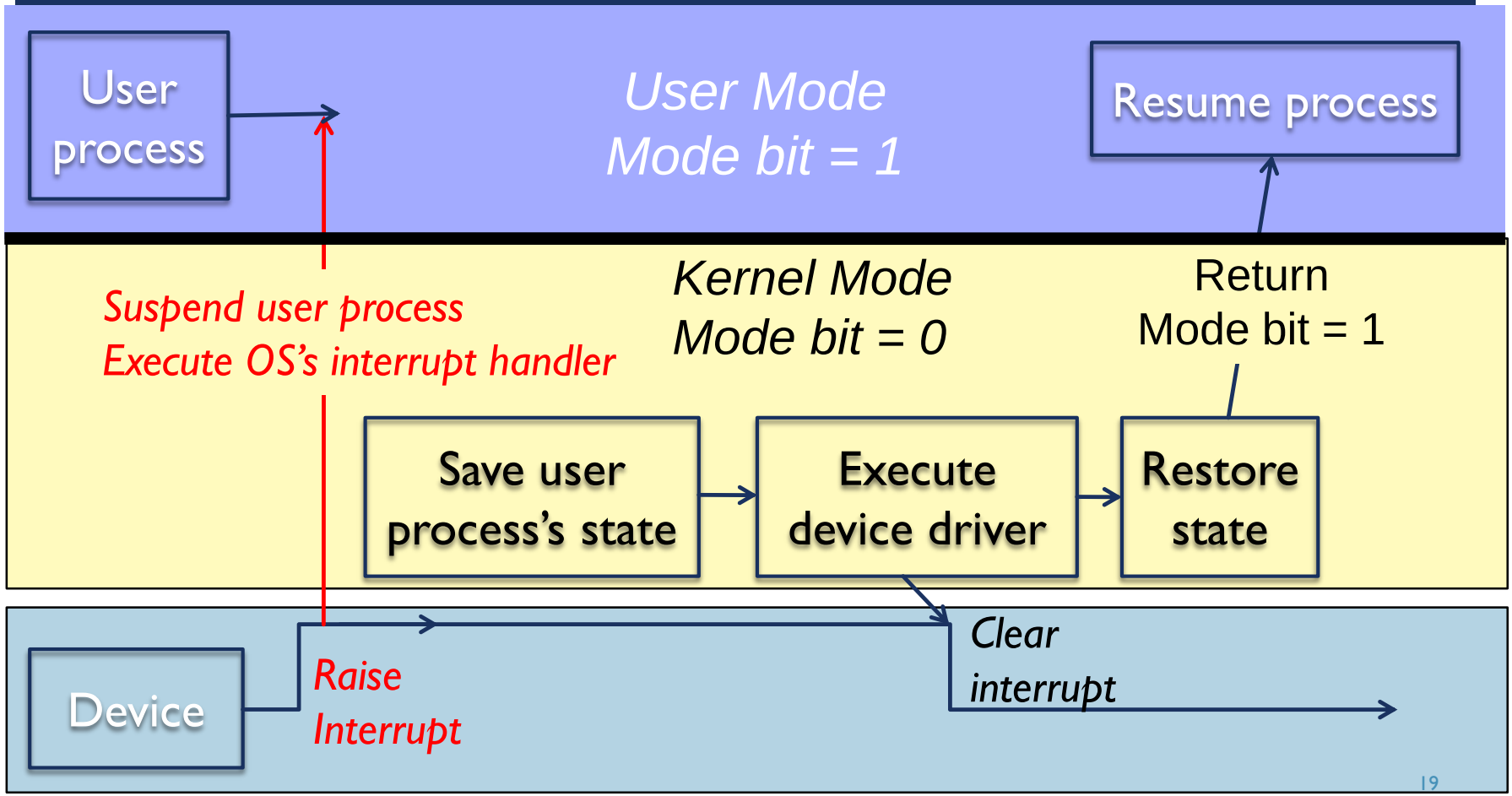
COMPUTER-SYSTEM OPERATION

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an interrupt

INTERRUPTS

- Interrupts signal asynchronous events
  - I/O hardware interrupts
  - Hardware timers

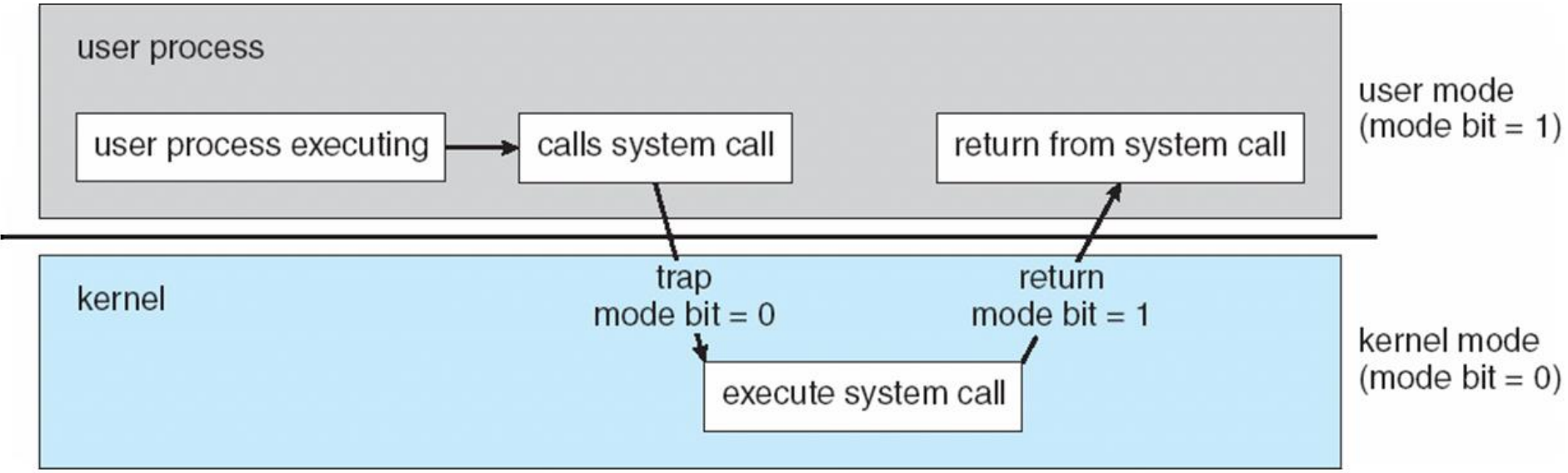
INTERRUPT ILLUSTRATED



COMMON FUNCTIONS OF INTERRUPTS

- Interrupt transfers control to the **interrupt service routine** generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*
- A ***trap*** is a software-generated interrupt caused either by an error or a user request
- An operating system is **interrupt driven**

Trap


user process
user process executing
calls system call
return from system call
user mode
(mode bit = 1)
kernel
trap
mode bit = 0
return
mode bit = 1
execute system call
kernel mode
(mode bit = 0)

INTERRUPT HANDLING

- The operating system preserves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
  - **polling**
  - **vectored** interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

ALTERNATIVE APPROACH

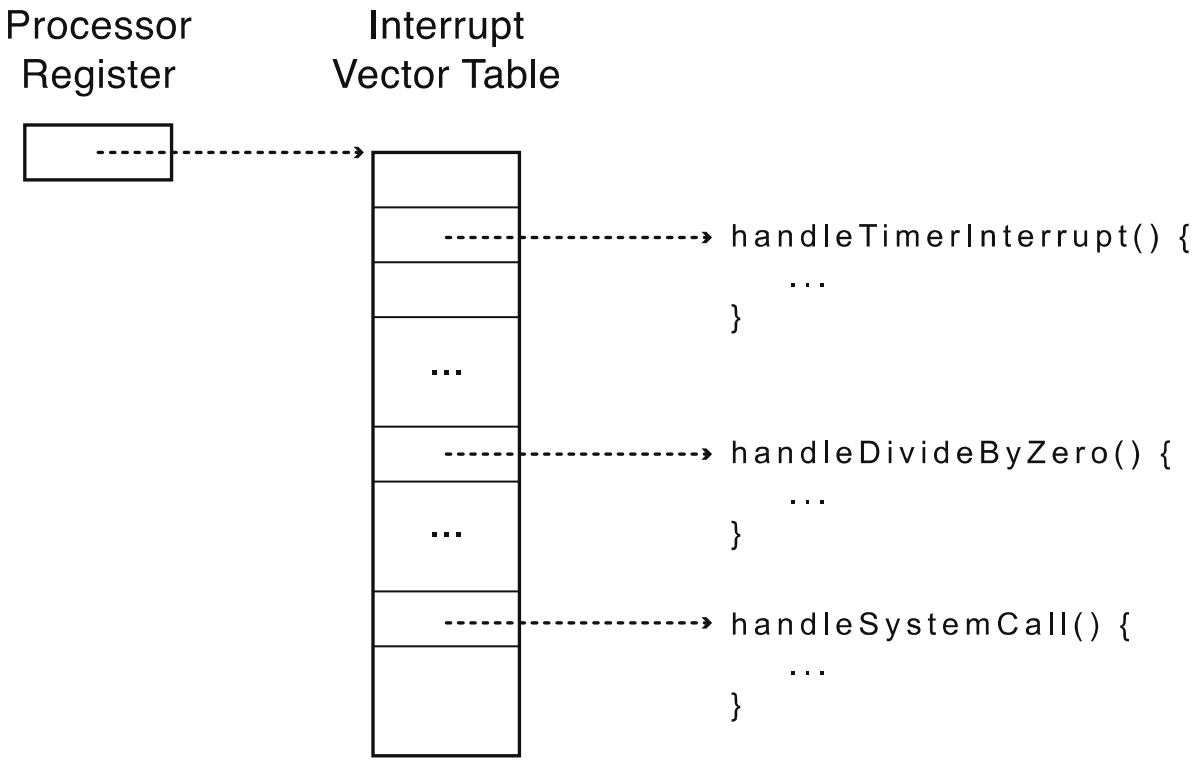
- Polling

```
while (Ethernet_card_queue_is_empty)
        ;
// Ethernet card received packets.
handle_packets();
```

- Problems?
- Analogy:
  - Polling: keeps checking the email every 30 seconds
  - Interrupt: when email arrives, give me a ring

INTERRUPT VECTOR

- Table set up by OS kernel; pointers to code to run on different events



QUESTION: HOW DO WE TAKE INTERRUPTS SAFELY?

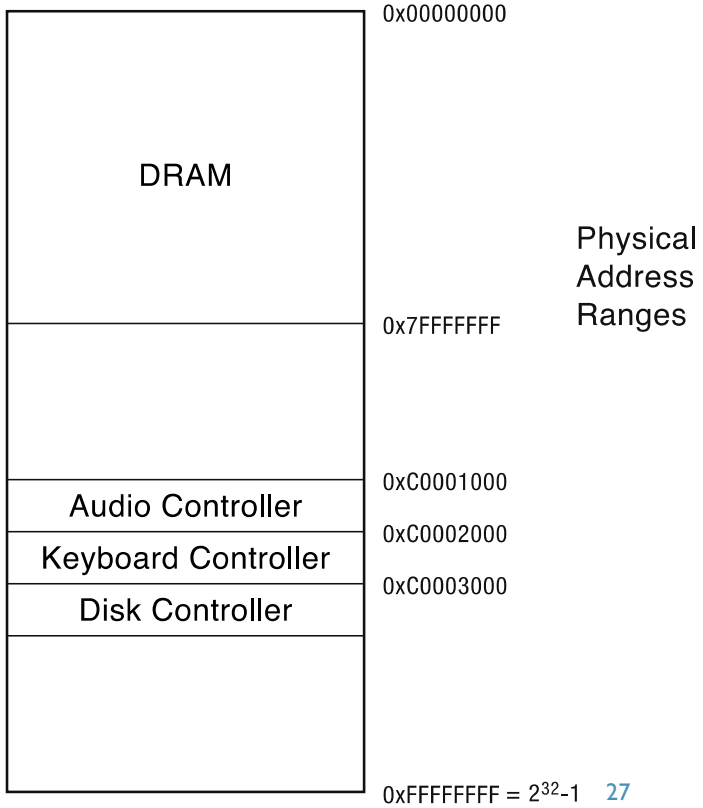
- Interrupt vector
  - Limited number of entry points into kernel
- Atomic transfer of control
  - Single instruction to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent re-startable execution
  - User program does not know interrupt occurred

INTERRUPT HANDLERS

- Non-blocking, run to completion
  - Minimum necessary to allow device to take next interrupt
  - Any waiting must be limited duration
  - Wake up other threads to do any real work
    - Linux: semaphore
- Rest of device driver runs as a kernel thread

QUESTION

- For a "Hello world" program, the kernel must copy the string from the user program memory into the screen memory.
- Why not allow the application to write directly to the screen's buffer memory?

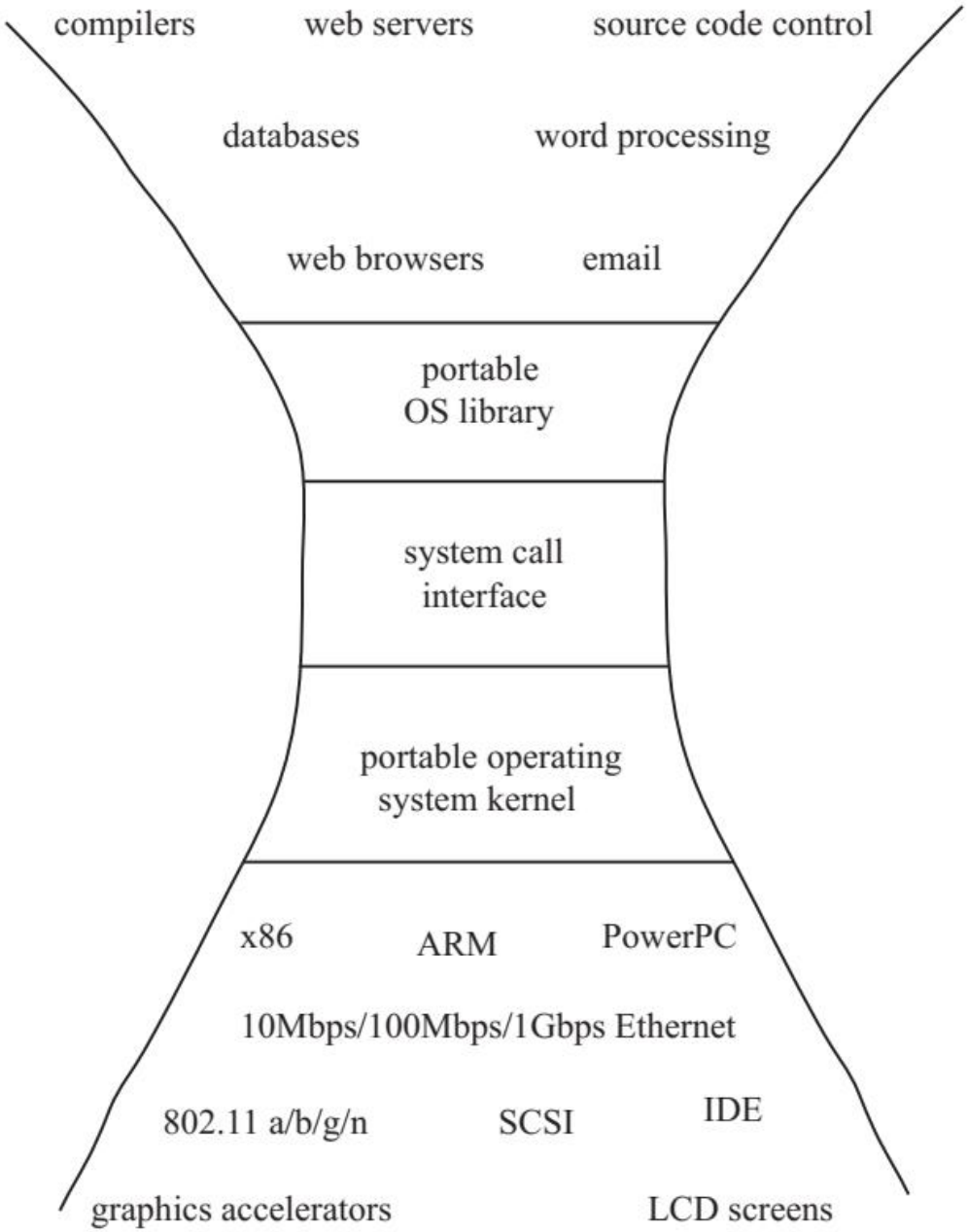


HARDWARE TIMER

- Hardware device that periodically interrupts the processor
  - Returns control to the kernel handler
  - Interrupt frequency set by the kernel
    - Not by user code!
  - Interrupts can be temporarily deferred
    - Not by user code!
    - Interrupt deferral crucial for implementing mutual exclusion

MODE SWITCH

- From user mode to kernel mode
  - Interrupts
    - Triggered by timer and I/O devices
  - Exceptions
    - Triggered by unexpected program behavior
    - Or malicious behavior!
  - System calls (aka protected procedure call)
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points

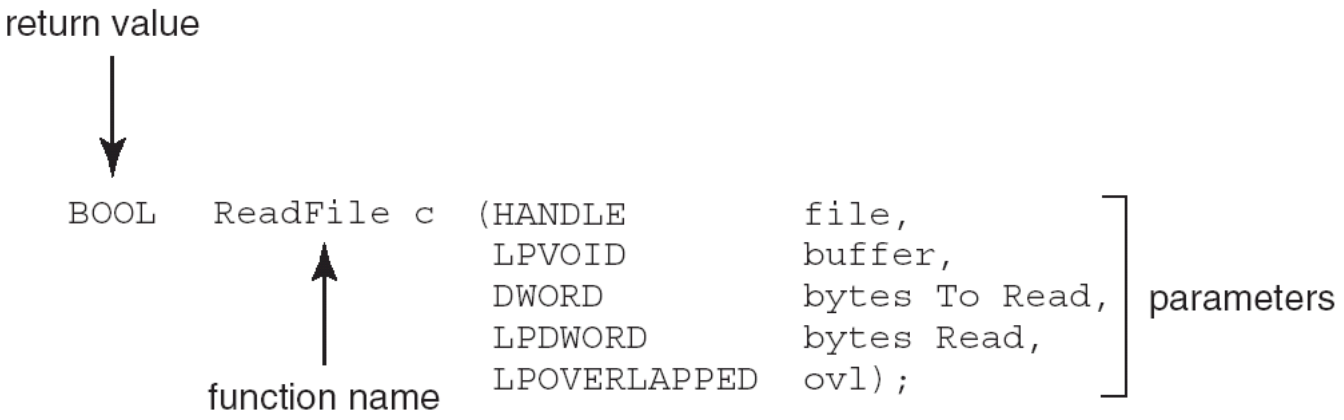

compilers
web servers
source code control
databases
word processing
web browsers
email
portable
OS library
system call
interface
portable operating
system kernel
x86
ARM
PowerPC
10Mbps/100Mbps/1Gbps Ethernet
802.11 a/b/g/n
SCSI
IDE
graphics accelerators
LCD screens

SYSTEM CALLS

- 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 Program Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?

EXAMPLE OF STANDARD API

- **Consider the ReadFile() function in the**
- **Win32 API—a function for reading from a file**

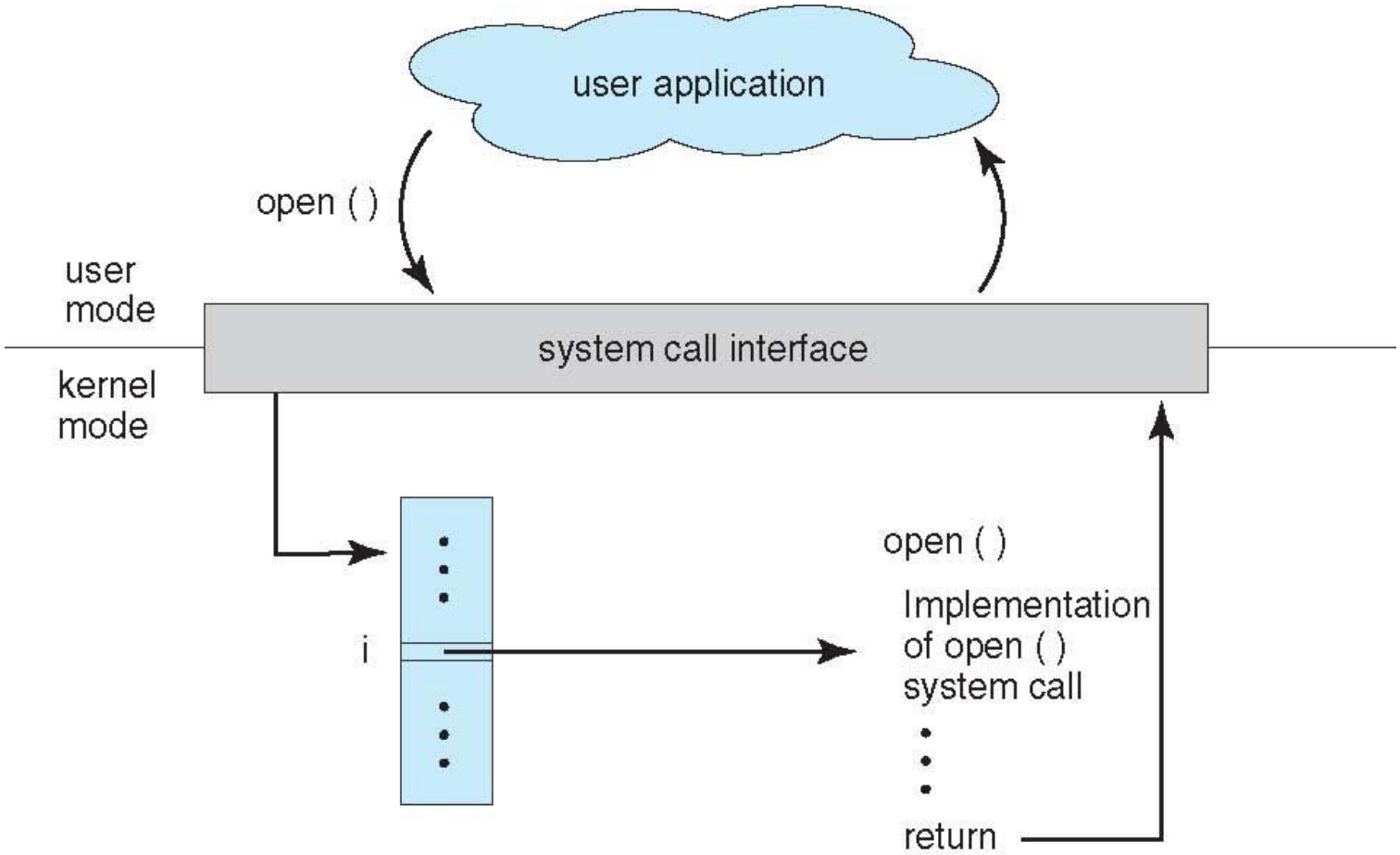


- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

SYSTEM CALL IMPLEMENTATION

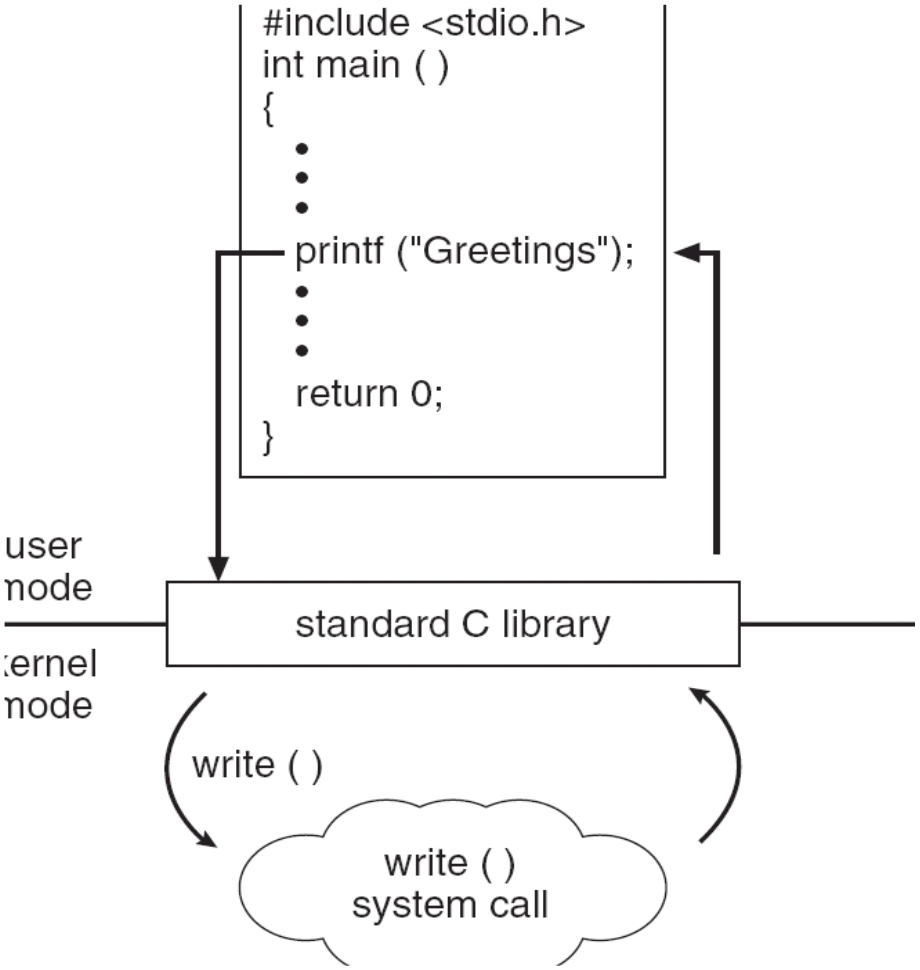
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – SYSTEM CALL – OS RELATIONSHIP



STANDARD C LIBRARY EXAMPLE

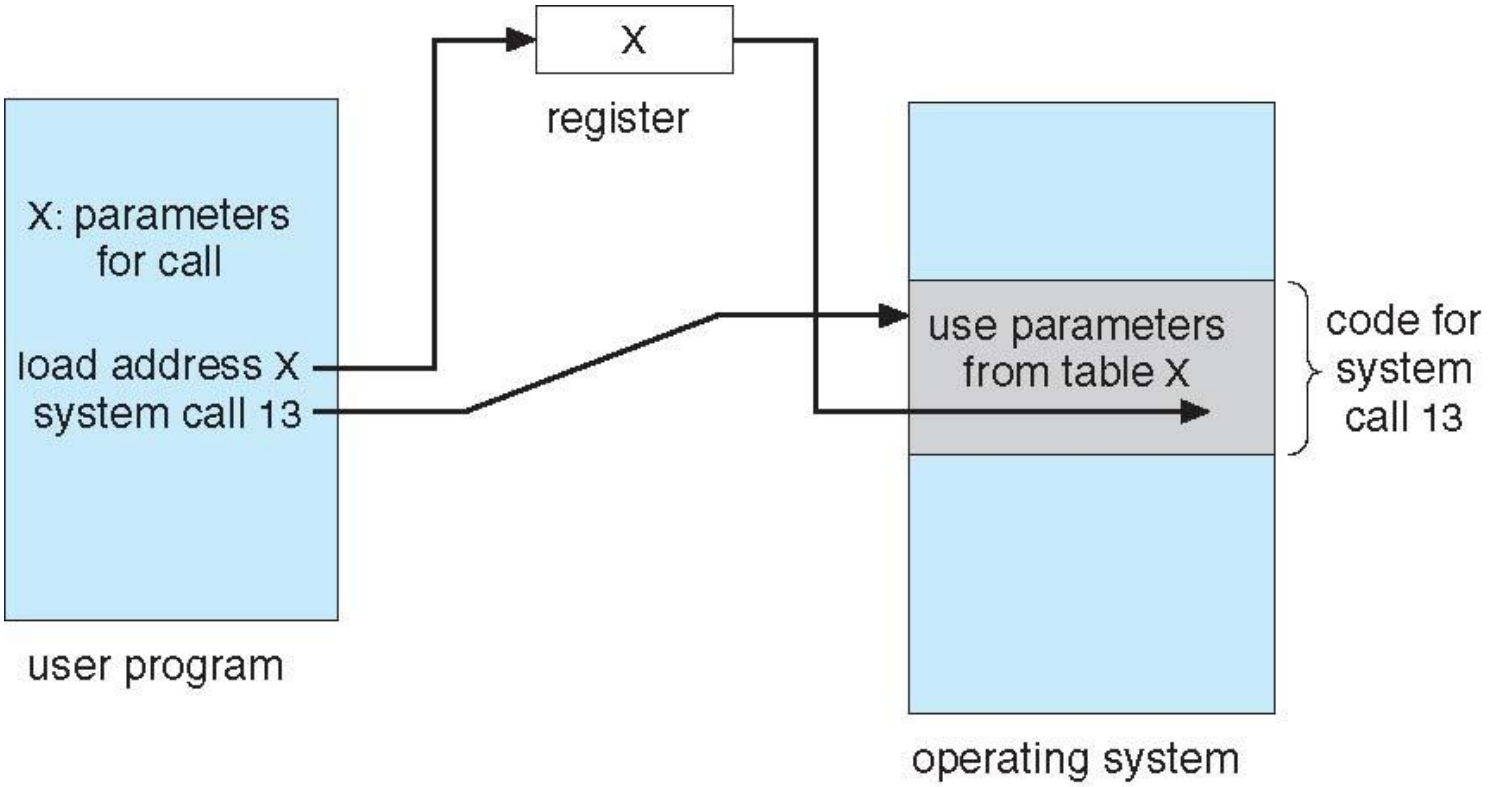
- C program invoking printf() library call, which calls write() system call



SYSTEM CALL PARAMETER PASSING

- Often, more information is required than simply identity of desired system call - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - Parameters stored in a *block*, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

PARAMETER PASSING VIA TABLE



SUMMARY

- Protection
  - User/kernel modes
  - Protected instructions
- System calls
  - Used by user-level processes to access OS functions
  - Access what is "in" the OS
- Exceptions
  - Unexpected event during execution (e.g., divide by zero)
- Interrupts
  - Timer, I/O

SUMMARY (2)

- After the OS has booted, all entry to the kernel happens as the result of an event
  - event immediately stops current execution
  - changes mode to kernel mode, event handler is called
- When the processor receives an event of a given type, it
  - transfers control to handler within the OS
  - handler saves program state (PC, registers, etc.)
  - handler functionality is invoked
  - handler restores program state, returns to program