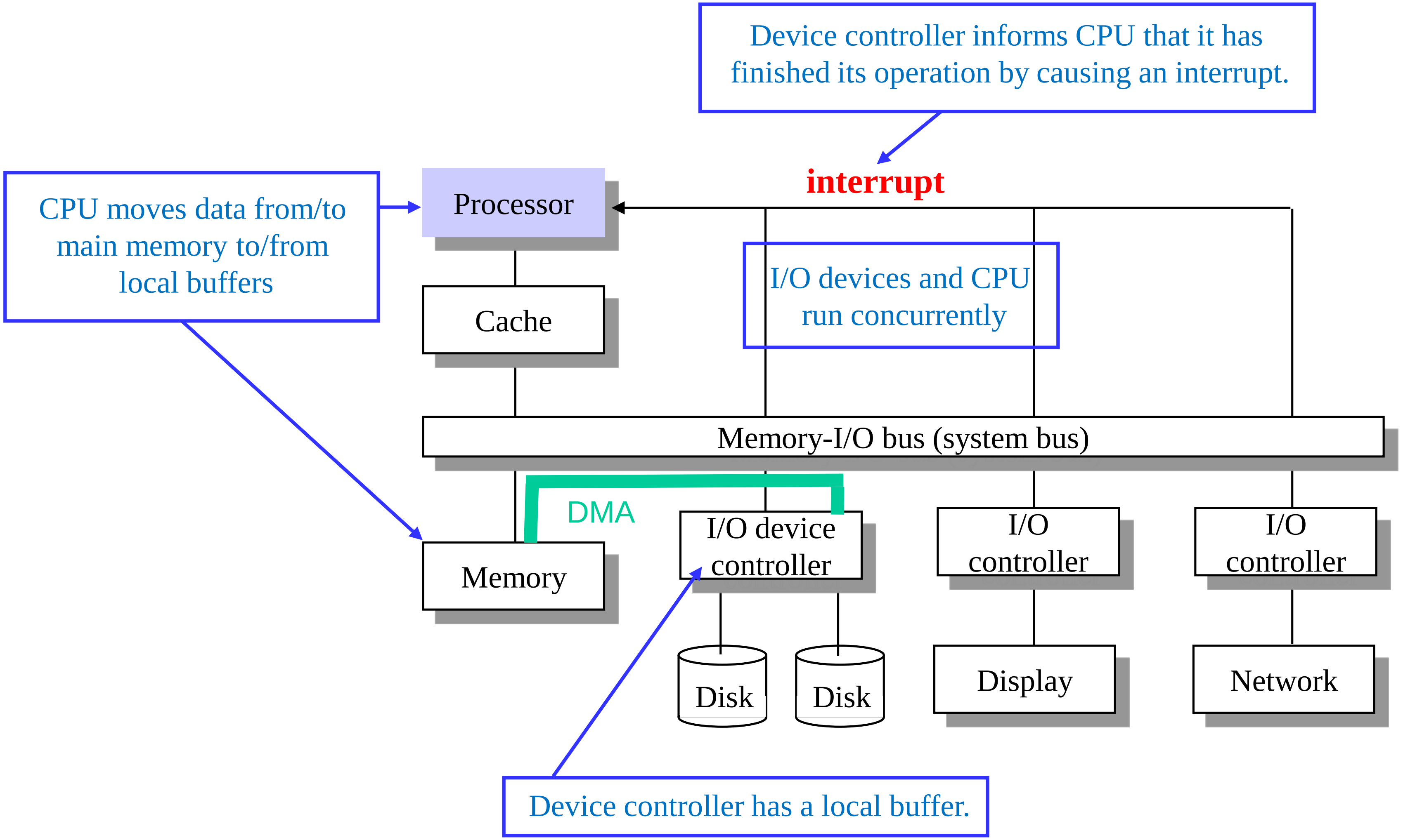
Lecture 2: Operating System - Background Knowledge

Spring 2017

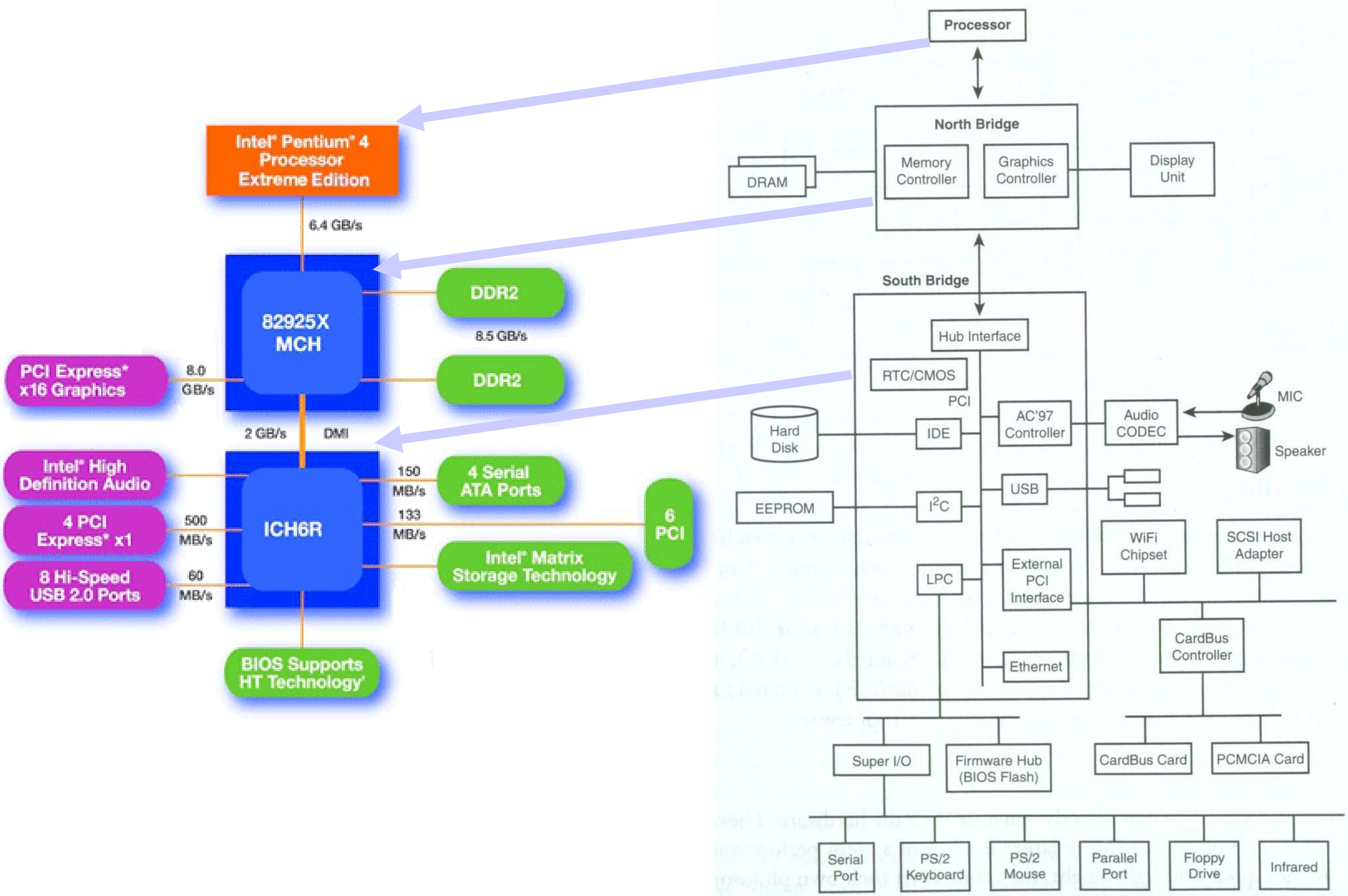
차호정

연세대학교 컴퓨터과학과

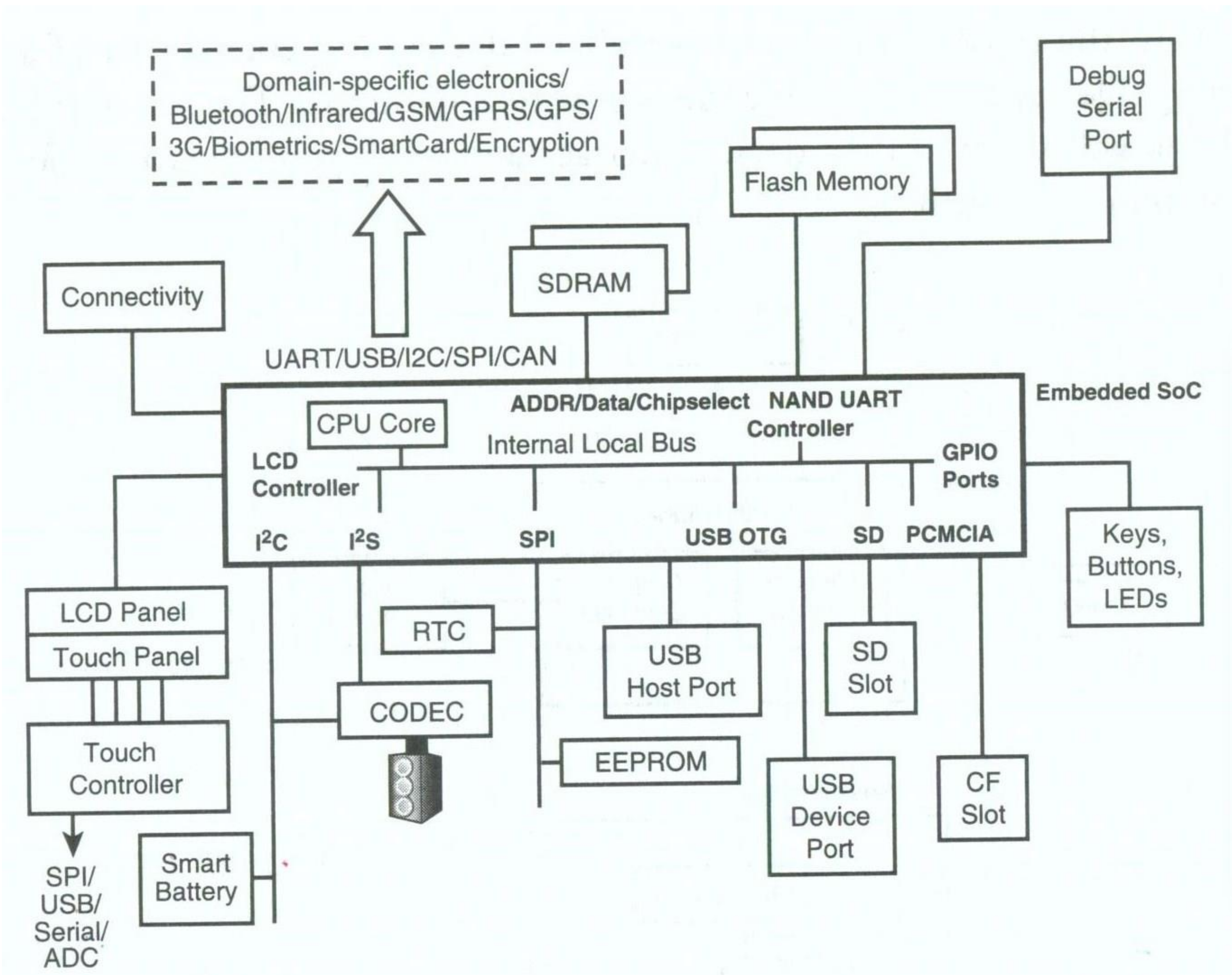
Computer System Architecture: Abstract View



Real World: a Large Pentium (PC) System

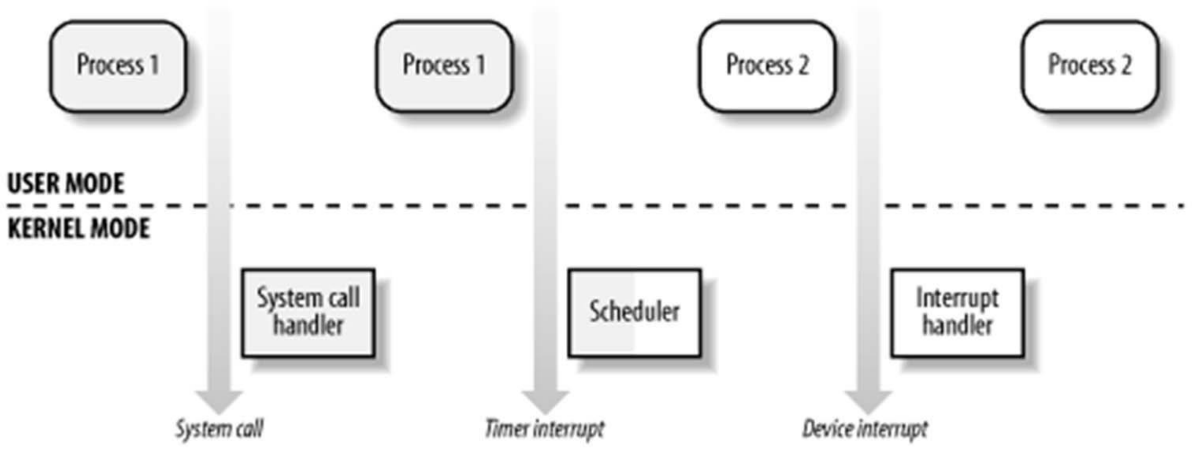


Real World: Embedded System



Hardware Protection for OS

- Dual mode operation
  - A properly designed OS must ensure that an incorrect program cannot cause other programs to execute incorrectly.
  - OS has a dual mode mechanism (user mode and kernel mode) to protect the system from many types of application faults
  - Processor provides the protection mechanism



Dual-Mode (User/Kernel Mode) Operation (1)

- Provide *hardware support* to differentiate between at least two modes of operations
  - User mode: execution done on behalf of a user
  - Kernel mode (also "*supervisor mode"* or "*system mode"* or *"monitor mode"*): execution done on behalf of operating system
  - Mode is set by a status bit in a protected processor register.
  - Example:
    - Intel 80x86 has four different execution states (i.e. the **cs** segmentation register includes 2-bit field that specifies the CPL (Current Privilege Level) of the CPU.
    - All standard UNIX kernels make use of only User Mode and Kernel Mode.
  - Some machine instructions are designed as *privileged (protected) instructions* and they can be issued only in kernel mode.
    - Most of the I/O instructions

Dual-Mode (User/Kernel Mode) Operation (2)

- Crossing protection boundaries
  - User programs must call on OS to do something privileged (i.e., invoking privileged instructions)
  - Pass control to a kernel service routine running in kernel mode.
  - The kernel verifies that the parameters are correct and legal, executes the request.

- Three cases :

  (1) Hardware Interrupt

  (2) Software interrupt (exception)

  (3) System call

Case Study: x86 *Real* Mode

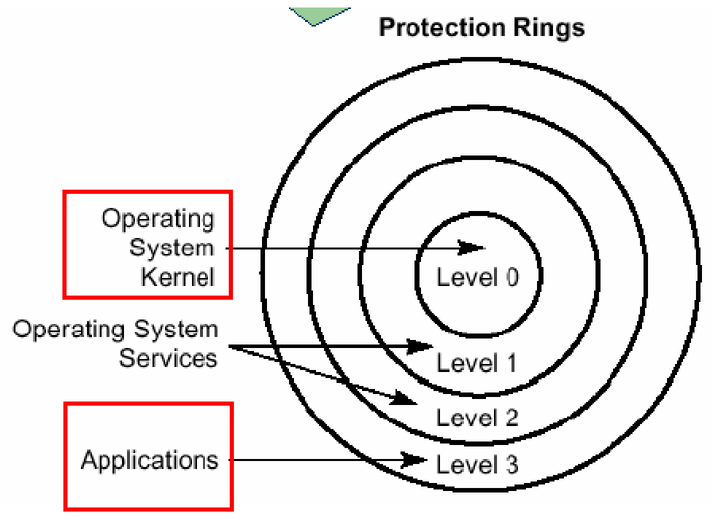
- Real mode refers to compatibility with 16 bit Intel CPUs (8086, 80286)
- All x86 CPUs start in “real mode”
  - The system BIOS only works in real mode.
  - So, the boot code has to work in real mode.
- Segmented memory
  - All segments are restricted to 64KB in size when in real mode
    - IP (instruction pointer), segment registers are all 16 bits!
  - Awkward to work with objects larger than a segment (64KB)
  - No paging or memory protection

Case Study: x86 *Protected* Mode

- The 386 and higher CPUs have a *protected mode*
  - 32bit memory address
  - Memory protection
  - Virtual memory paging
  - IO protection
  - Privilege levels
  - Task switching
  - Interrupt handling
- Entering protected mode
  - Construct valid code, data, and stack segments (GDT, LDT)
  - Set PE bit in CR0 register
  - Jump to a valid code address in a code segment

x86 Protected Mode: Privilege Level

- Privilege checking
  - Ensure that the currently-executing program cannot access areas of memory unless permitted to do so.



- 0 is most privileged, 3 is least
- Most OS uses 0 for kernel code, 3 for user code
- Privilege levels 1 and 2 could be used for more fine-grained protection (e.g., device drivers)

  - Three components are involved in the privilege checking:
    - CPL (current privilege level) of the current program
    - RPL (requestor privilege level) in the segment register
    - DPL (descriptor privilege level) of the target segment

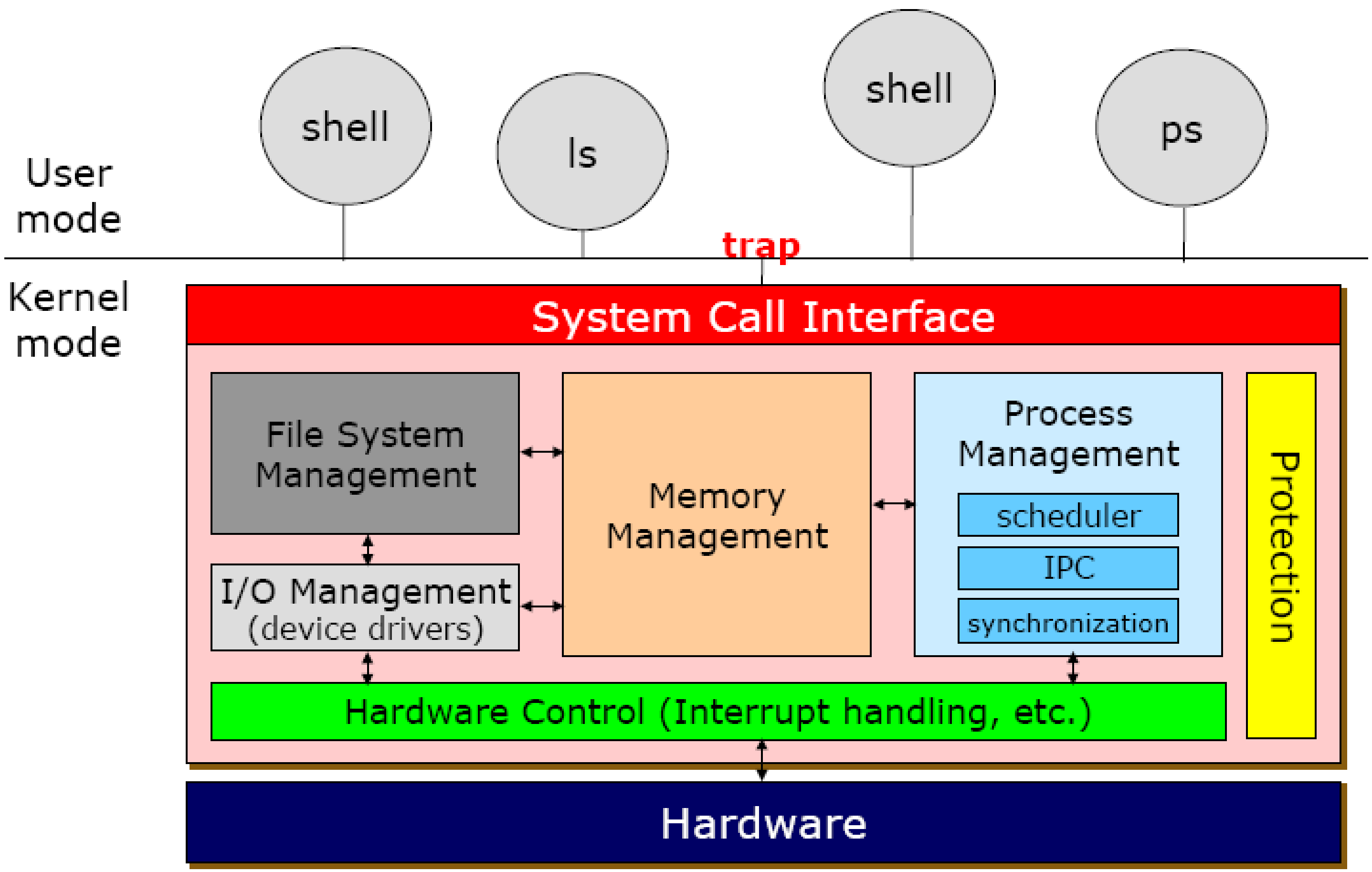
Operating System Services (1)

- User Services
  - Program execution
    - System capability to load a program into memory and run it.
  - I/O operations
    - Since user programs cannot execute I/O operations directly, the operating system must provide some means to perform I/O.
  - File-system manipulation
    - Program capability to read, write, create, and delete files.
  - Communications
    - Exchange of information between processes executing either on the same computer or on different systems tied together by a network.
  - Error detection
    - Ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user programs.

Operating System Services (2)

- Resource allocation
  - Allocating resources to multiple users or multiple jobs running at the same time.
- Accounting
  - Keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics.
- Protection
  - Ensuring that all access to system resources is controlled

Operating System Structure



System Call Interface

- System Calls
  - Provide the interface between a running program and the operating system.
  - Generally available as function calls.

Process Management	fork	CreateProcess	Create a new process
	waitpid	WaitForSingleObject	Wait for a process to exit
	execve	(none)	CreateProcess = fork + execve
	exit	ExitProcess	Terminate execution
	kill	(none)	Send a signal
File Management	open	CreateFile	Create a file or open an existing file
	close	CloseHandle	Close a file
	read	ReadFile	Read data from a file
	write	WriteFile	Write data to a file
	lseek	SetFilePointer	Move the file pointer
	stat	GetFileAttributesEx	Get various file attributes
	chmod	(none)	Change the file access permission
File System Management	mkdir	CreateDirectory	Create a new directory
	rmdir	RemoveDirectory	Remove an empty directory
	link	(none)	Make a link to a file
	unlink	DeleteFile	Destroy an existing file
	mount	(none)	Mount a file system
	umount	(none)	Unmount a file system
	chdir	SetCurrentDirectory	Change the curent working directory

System Call Principles

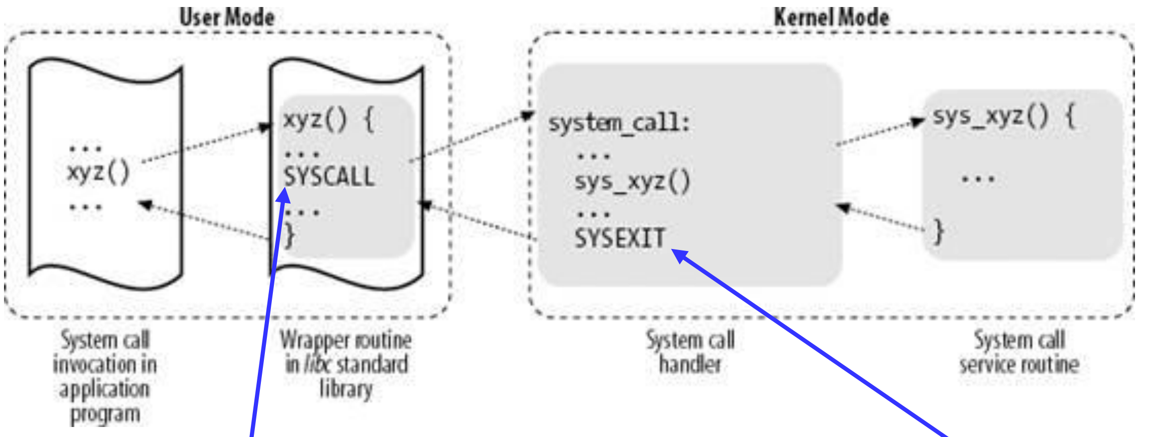
- Putting an extra layer between the applications and hardware
  - Advantages
    - Easy to program: freeing user from aware low-level programming characteristics of hardware devices
    - Increasing system security: the kernel can check the correctness of the request at the interface level
    - Increase program portability
- System calls
  - UNIX systems implement most interfaces between User Mode processes and hardware devices by means of system calls issued to the kernel.
    - Interfaces between User Mode processes and hardware devices
    - To request the kernel services

POSIX APIs and System Calls

- API (Application Programming Interface)
  - A function definition that specifies how to obtain a given service
    - For example, POSIX APIs - malloc(), calloc(), free() - are implemented in the ***libc***, which uses the brk() system calls.
    - "POSIX-compliant" if a system offers the proper set of APIs to the applications, no matter how the corresponding functions are implemented.
  - Programmers point of view: User Mode libraries
- System call
  - An explicit request to the kernel made via a software interrupt.
    - Kernel designer's pointer of view: belongs to the kernel
  - Some system calls takes one or more arguments.
  - Returns an integer value
    - If failed: return –1 and set errno (see include/asm-i386/errno.h)
  - Implementation: *a system call is implemented as a wrapper function in* ***libc*** *in user space.*

System Call Handling in x86/Linux (1)

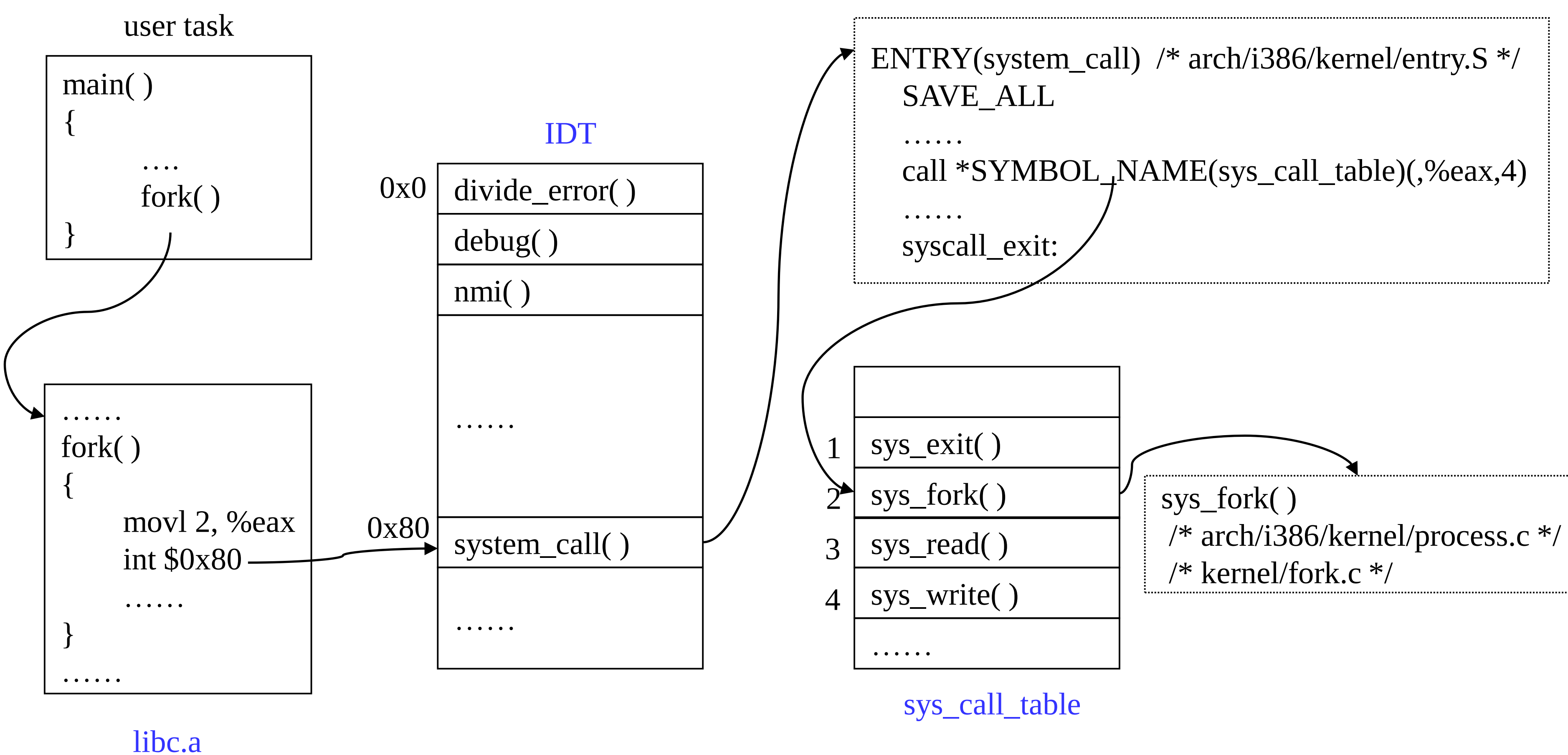
- Overview of system call handling
  - (1) Saves the contents of most registers in the Kernel Mode stack
  - (2) Handles the system call by invoking the system call service routine
  - (3) Exits from the handler: restore the registers and switches back to User Mode.



(1) “int 0x80” or (2) “sysenter”

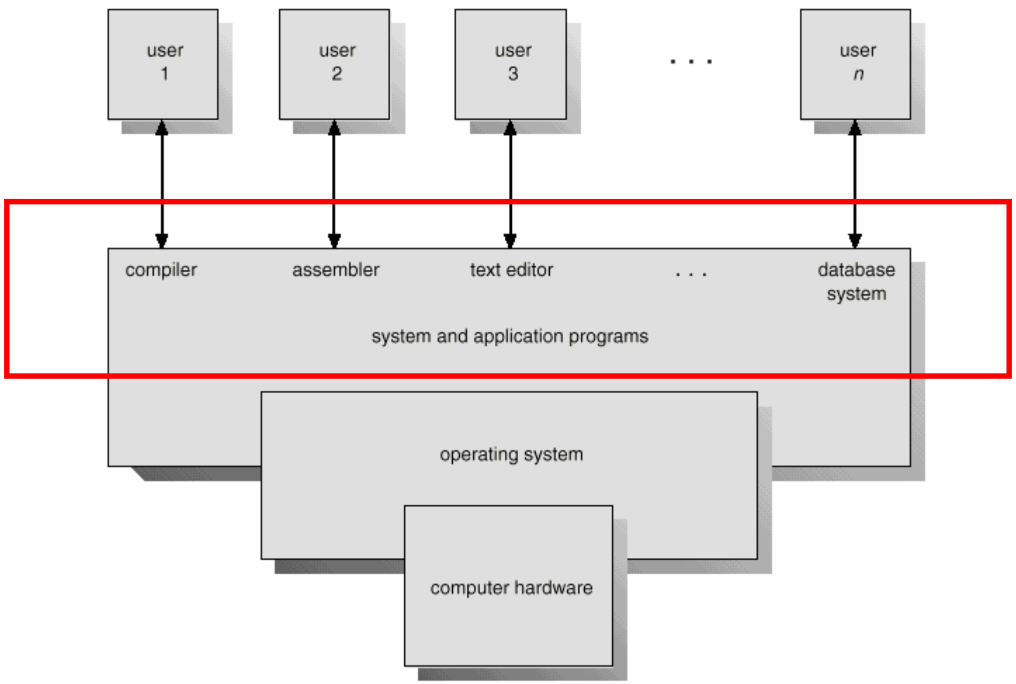
(1) “syscall_exit“ or (2) “sysexit”

System Call Handling in x86/Linux (2)



OS & System Programs (1)

- System Programs
  - Provide a convenient environment for program development and execution.
  - Most users' view of the operating system is defined by system programs, not the actual system calls.



OS & System Programs (2)

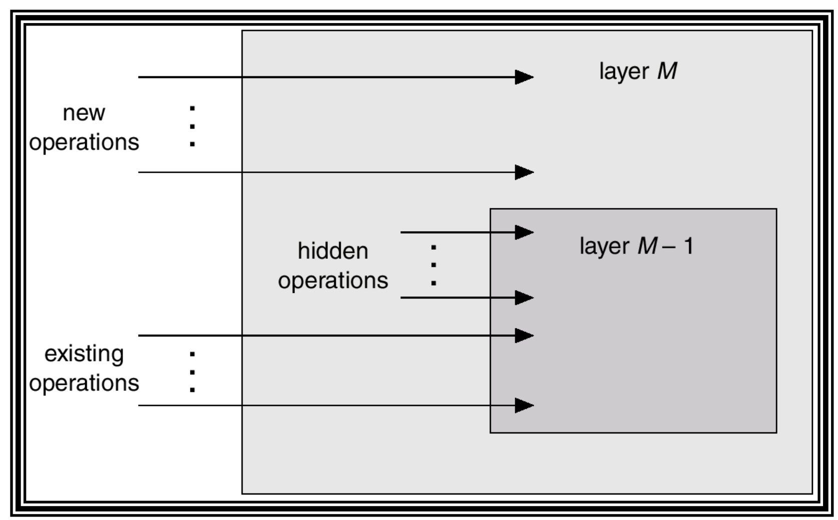
- They can be divided into:
  - File manipulation
    - create, delete, copy, rename, print, dump, list, …
  - Status information
    - date, df, du, top, …
  - File modification
    - Editor
  - Programming language support
    - Compiler, assembler, interpreter
  - Program loading and execution
    - Loader, linkage editor, debugger
  - Communications
    - Socket, stream, …
  - Application programs
    - DBMS, Web browsers, word processors, games, plotting tools, …

Operating System Design (1)

- System Design Goals
  - User Goals
    - Operating system should be convenient to use, easy to learn, reliable, safe, and fast.
  - System Goals
    - Operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient.
- Mechanisms and Policies
  - *Mechanisms* determine how to do something, *policies* decide what will be done.
  - Example
    - Mechanism: Time slicing in time-sharing system
    - Policy: time quantum
  - <u>The separation of policy from mechanism is an important principle</u>.
    - Allow maximum flexibility if policy decisions are to be changed later.

Operating System Design (2)

- Layering Approach
  - The operating system is divided into a number of layers (levels), each built on top of lower layers.
  - Advantages
    - Modularity: layers are selected such that each uses functions (operations) and services of only lower-level layers.
    - Simplify debugging and system verification
  - Difficulties
    - Careful definition of layers
    - Performance



Operating System Design (3)

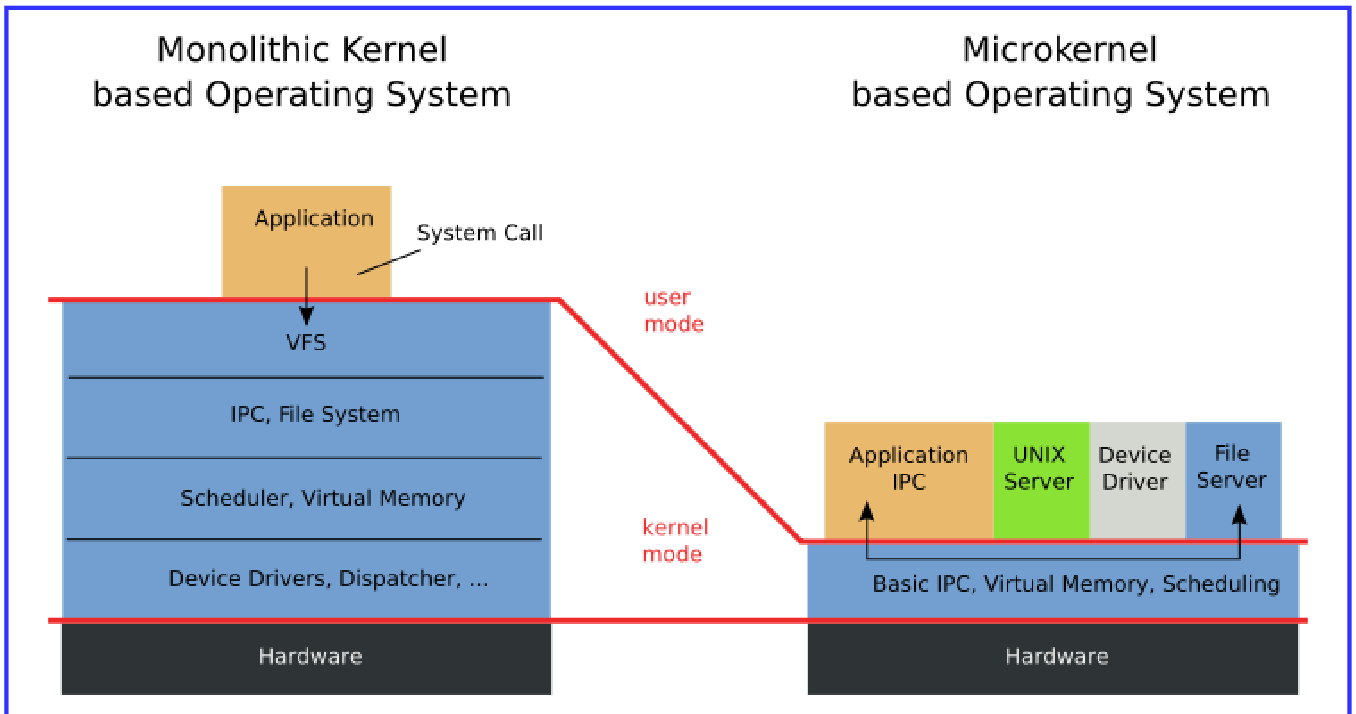
- System Implementation
  - Traditionally written in assembly language, operating systems can now be written in higher-level languages.
- Code written in a high-level language:
  - Can be written faster.
  - Easier to understand and debug.
  - Far easier to *port !*

Microkernels (1)

- Monolithic Kernel
  - All operating system services implemented in one big monolithic kernel
  - Virtually any procedure calls any other procedure.
- Microkernel
  - Only the *essential* core OS functions should be in the kernel.
  - Less essential services and applications are built on the microkernel and execute *in user mode.*
    - The operating system services are structured as a collection of independent processes
  - Communication takes place between user modules using message passing.
  - Example: CMU' Mach, QNX, L4 kernel, …

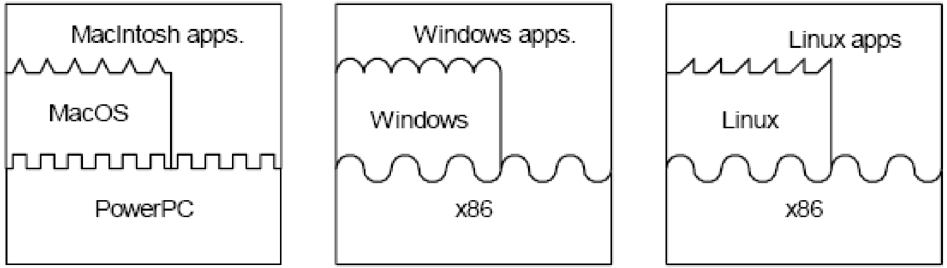
Microkernels (2)

- Advantages
  - Extensibility, Modularity (Maintainability)
  - Small kernel makes it easy to debug and be more efficient
- Disadvantages: performance
  - Invoking services involve mode/process switches
  - Essential system services executing in user model

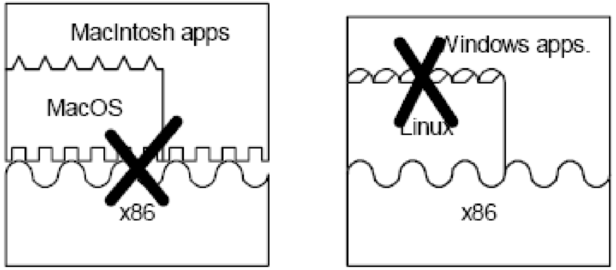


Virtual Machine (1)

- What is a Virtual Machine (VM)?
  - Software for cross-platform compatibility
  - Traditionally, an application program is bound to a specific platform which is ISA(instruction set architecture) + OS.
  - VM eliminates this real-platform constraint for higher degree of portability and flexibility



(a)



(b)

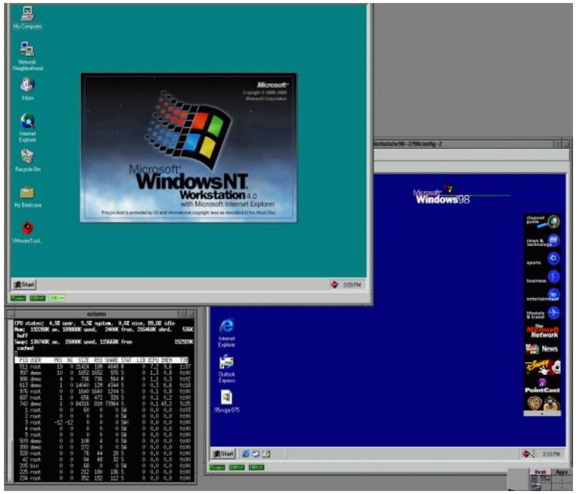
Virtual Machine (2)

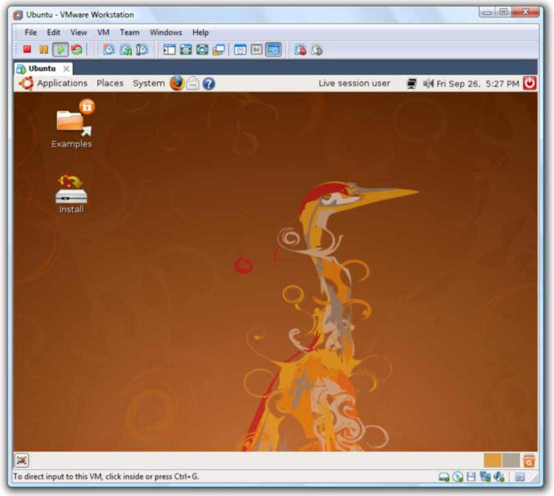
- Why do we need VM?
  - Portability is essential in networked computing
    - Especially useful for mobile, wireless-download platforms where we can achieve consistent execution environment on diverse CPU/OS/hardware devices: e.g., Java VM, GVM, Brew, WIPI, …
  - CPU innovations are often limited by old interfaces
    - New powerful CPUs that cannot run x86 binaries are not viable on the market
    - Solution: run x86 binaries on high-performance, low-power CPUs
  - Single OS on a H/W may open a security hole
    - E.g., a server shared by different groups of users who want to be assured of a secure environment
  - Sandbox an OS that is not trusted, possibly because it is a system under development.
    - Virtual machines have other advantages for OS development, including better debugging access and faster reboots

Virtual Machine (3)

- VM Solution
  - Implementing a layer of Software (VM) for virtualization
  - Mapping a virtual guest system to a real host system
- Two types of VM
  - Process VM: virtualization of individual processes
    - E.g., running x86 applications on Alpha CPU
  - System VM: virtualization of complete systems
    - E.g., running Linux (and its applications) on Windows

Running Windows98 and Windows NT on a Linux Host (www.vmware.com)

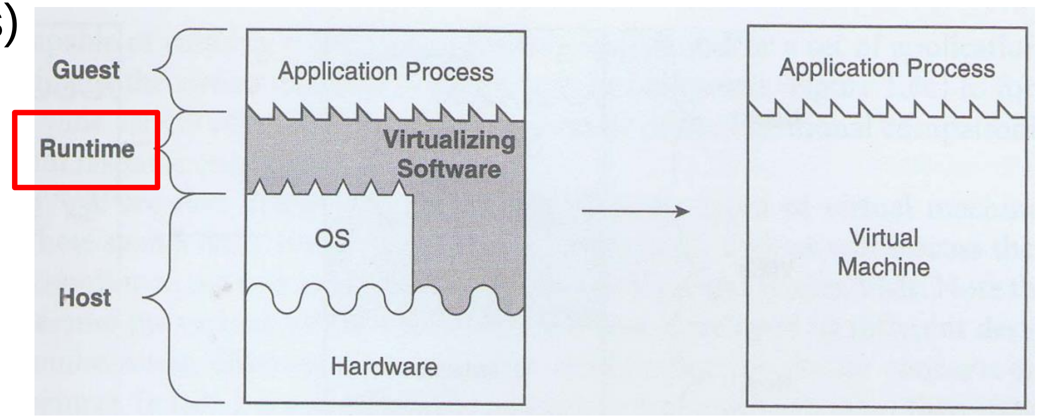




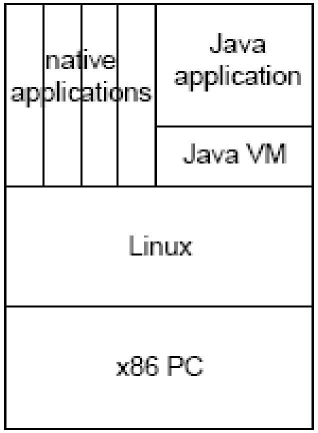
VMware Workstation 6.5.0 on Windows Vista, running Ubuntu 8.04.1

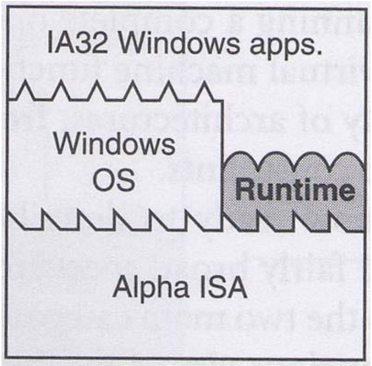
Virtual Machine (4)

- Process VM
  - Run executables of different ISA or OS
  - VM emulates ABI(application binary interface; user-level instructions & system calls)
  - Called "runtime"



Java Virtual Machine:
Run Bytecode binaries on x86

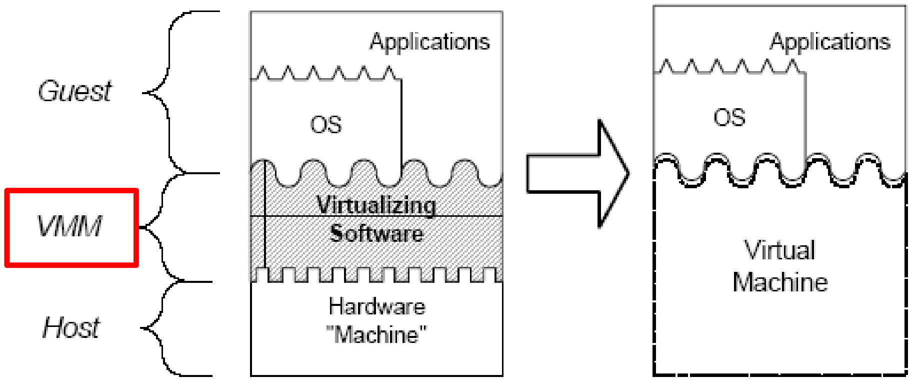




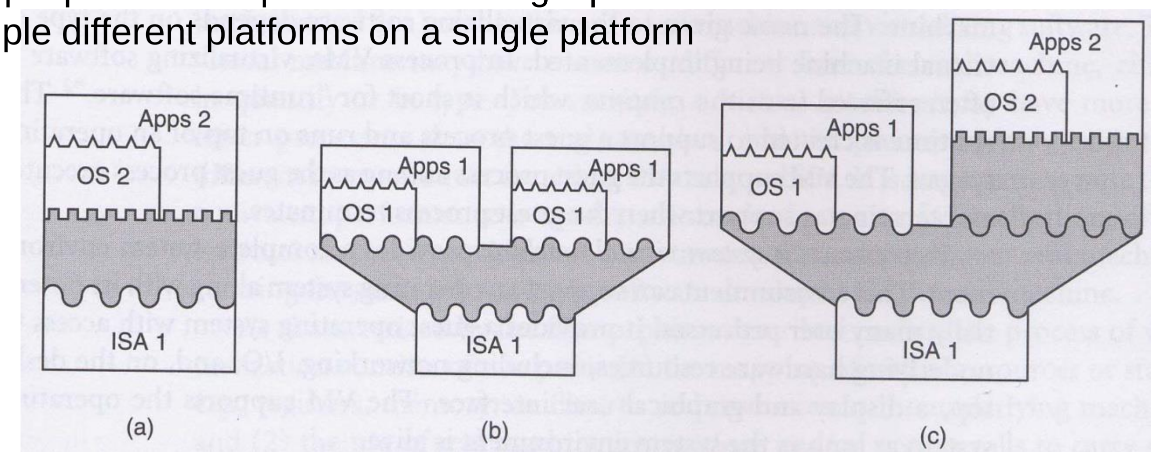
Digital FX!32:
Run x86 binaries on Alpha

Virtual Machine (5)

- System VM
  - Run whole OS(es) & executables
  - VM emulates whole ISA (user-level & system-level)
  - Called “VMM”
    (Virtual Machine Manager)

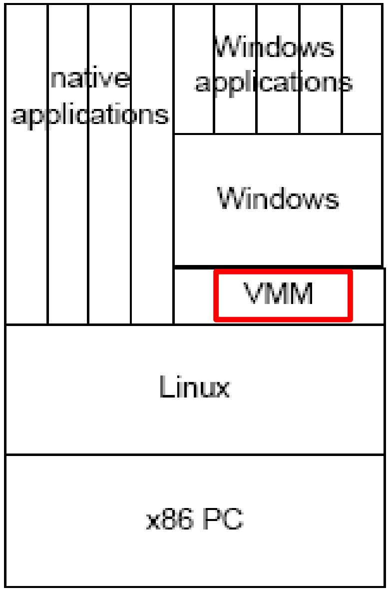


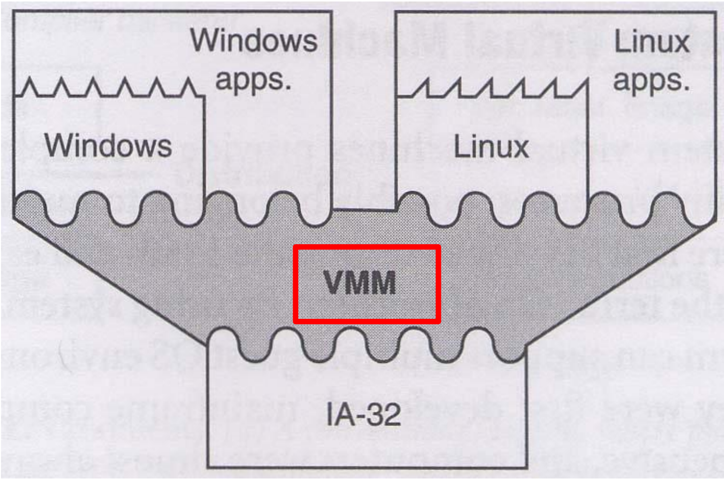
(a) One ISA is emulated by the other
(b) Multiple platforms replicated on a single platform
(c) Multiple different platforms on a single platform



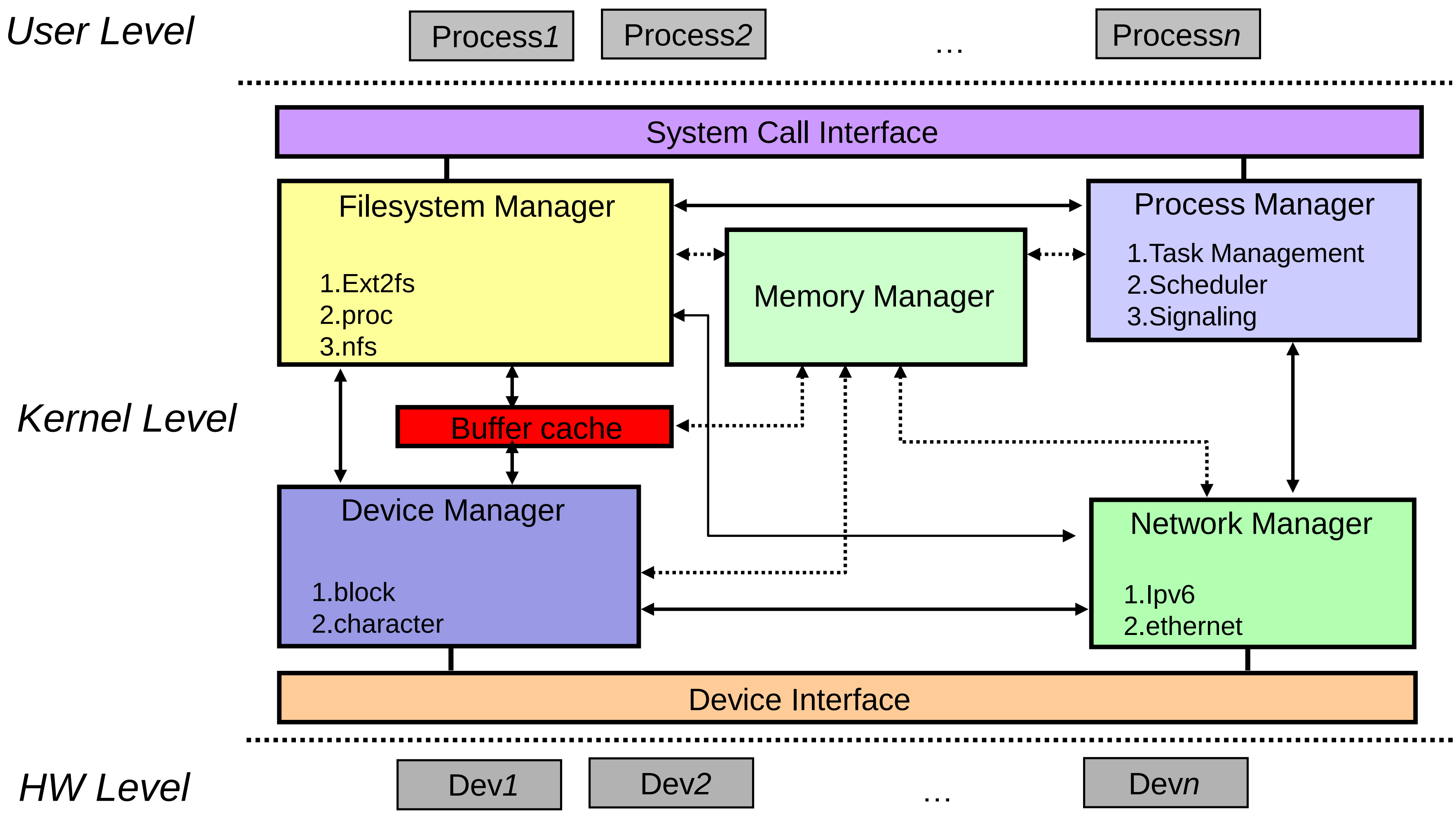
Virtual Machine (6)

- System VM: Implementation issues
  - Hosted or Stand-alone
- Hosted
  - Runs as a process on an existing host OS
  - Rely on host OS for H/W interaction
  - VMWare Workstation, User-Mode Linux, Microsoft Virtual PC/Server, …
- Stand-alone
  - VMM on top of bare hardware
  - All H/W interactions done by VMM itself
  - Highly efficient
  - VMWare ESX, IBM z/VM, …





Linux Kernel Architecture



Key Features of Linux (1)

- Monolithic kernel
  - A large, complex program, composed of several logically different components
  - High performance: low message passing overhead
  - C.f., microkernel approach (modular approach) : CMU's Mach
- Supporting "modules"
  - To dynamically load and unload some portions of the kernel code on demand (typically, device drivers)
  - C.f, traditional Unix kernels: compiled and linked statically
  - Only SVR4.2 kernel has a similar feature.

Key Features of Linux (2)

- Kernel threading
  - "Kernel thread": an execution context that can be independently scheduled on a common address space
  - Linux uses kernel threads in a very limited way to execute a few kernel functions
- Multithreaded application support
  - Linux defines its own version of lightweight process, which is different from those of (kernel thread based) Solaris or SVR4.
  - Lightweight process is the basic execution context.
  - Lightweight process is handled via the nonstandard clone() system call: "copy-on-write"

Key Features of Linux (3)

- Preemptive kernel
  - Starting from 2.6
  - Interleave execution flows while they are in privileged mode.
- Multiprocessor support
  - From Linux 2.2, yet make optimal use of SMP
- File system
  - The standard file system lacks some advanced features such as journaling, but more advanced file systems are available for Linux.