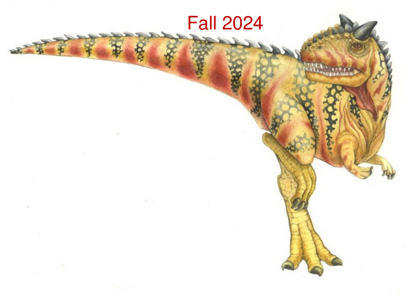
Lecture 2: Operating-System Structures

Hossein Asadi (asadi@sharif.edu)

Rasool Jalili (jalili@sharif.edu)

Fall 2024



Lecture 2: Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines

Objectives

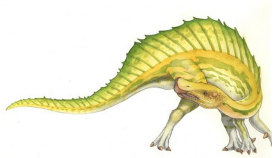
- To Describe Services an OS Provides to Users, Processes, and Other Systems
- To Discuss Various Ways (or Different Goals) of Structuring an OS

Operating System Services

- OSs Provide an Environment for Execution of Programs and Services to Programs and Users
- Set of OS Services Provides Functions that are Helpful to Users for Convenience of Programmers
  - User interface
  - Program execution
  - I/O operations
  - File-system manipulation
  - Communications
  - Error detection

Operating System Services (cont.)

- **User Interface**
  - Almost all OSes have a user interface (**UI**)
    - **Command-Line (CLI)**
    - **Graphics User Interface (GUI):** window/mouse
    - **Batch interface:** input commands from file
- **Program Execution**
  - System must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)



Operating System Services (cont.)

- **I/O Operations**
  - A running program may require I/O, which may involve a file or an I/O device
  - Users cannot directly control I/O devices
    - For sake of protection and efficiency
- **File-System Manipulation**
  - Programs need to read and write files/dir
  - Create and delete files/dir
  - Search files/dir
  - List file information
  - Permission management

Operating System Services (cont.)

- **Communications**
  - Processes may exchange information
    - On same computer
    - Or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by OS)

Operating System Services (cont.)

- **Error Detection**
  - OS needs to be constantly aware of possible errors
  - May occur in CPU and memory HW, in I/O devices, in user program
  - For each type of error, OS should take appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance user’s and programmer’s abilities to efficiently use system

Operating System Services (cont.)

- Another Set of OS Functions Exist for Ensuring Efficient Operation of System itself via Resource Sharing
  - Resource allocation
  - Accounting
  - Protection and security

Operating System Services (cont.)

Resource Allocation

- When multiple users/jobs running concurrently, resources must be allocated to each of them
- Many types of resources:
  - CPU cycles
  - Main memory
  - File storage
  - I/O devices

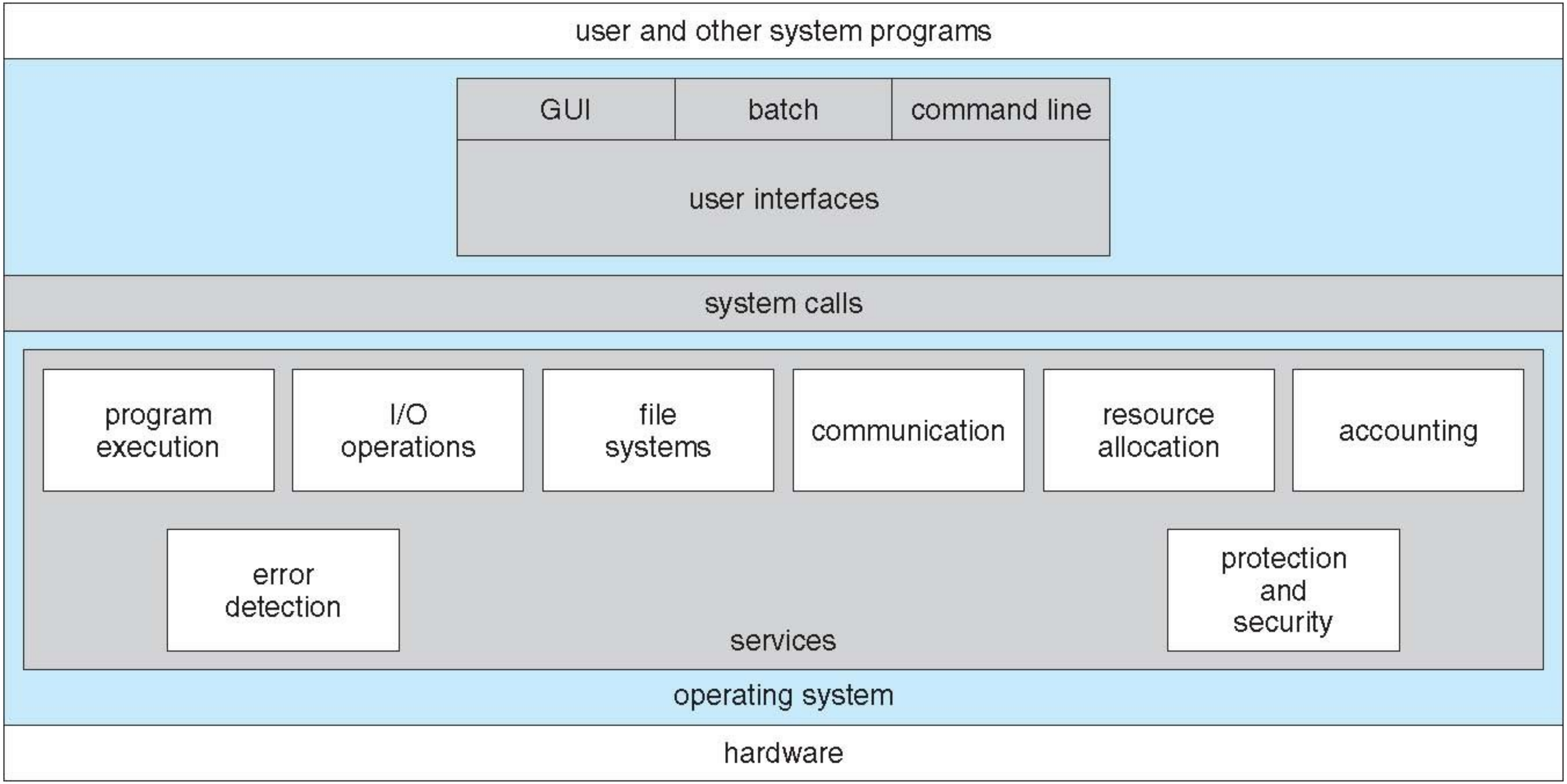
Operating System Services (cont.)

- **Accounting**
  - To keep track of which users use how much and what kinds of computer resources
  - Accounting info can be used for billing or just statistics
  - Such statistics info helps researchers/admins to reconfigure system to improve computing efficiency and services

Operating System Services (cont.)

- **Protection and Security**
  - Owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
- **Protection**
  - Involves ensuring that all accesses to system resources are controlled
- **Security**
  - Security of system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

OS Services: an Overview



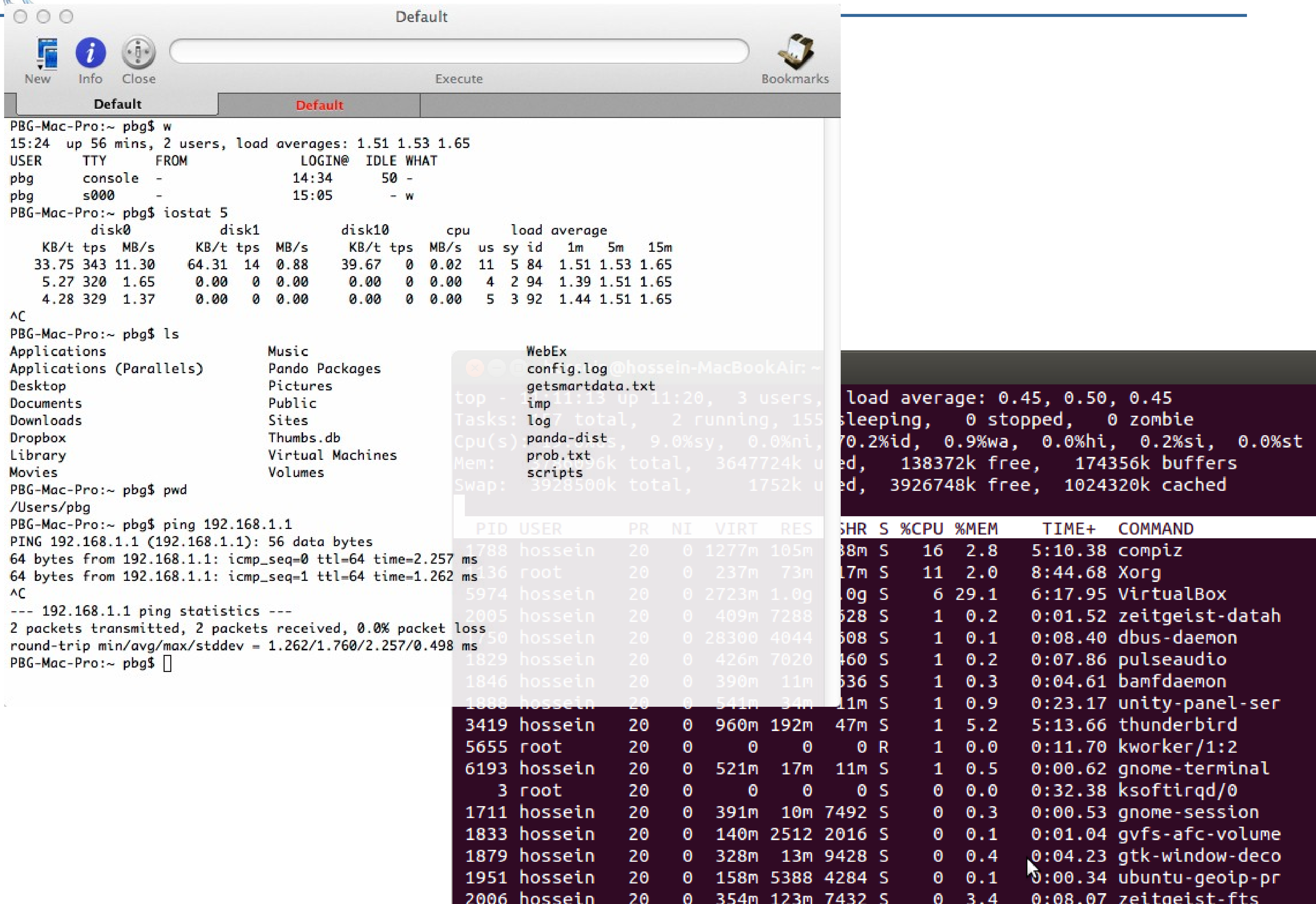
User OS Interface - CLI

- CLI or Command Interpreter
  - Allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented: **shells**
  - Primarily fetches a command from user and executes it

User OS Interface – CLI (cont.)

- Two Approaches in Implementing Shells
  - Commands built-in
    - For each command, there is a relevant code in Shell
  - Just names of programs
    - Used in UNIX
    - Does not understand input command
      - Just searches for a file/program with given name of command
    - E.g.: rm file.txt
      - Would search for a file called “rm” ➔ loads file into memory ➔ execute it with the parameter “file.txt”
    - Adding new features doesn’t require shell modification

Bourne Shell Command Interpreter



User Operating System Interface - GUI

- User-Friendly **Desktop** Metaphor Interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC

User Operating System Interface – GUI (cont.)

- Many Systems now Include both CLI and GUI Interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

Touchscreen Interfaces

Touchscreen Devices Require New Interfaces

- Mouse not possible or not desired
- Actions and selection based on gestures
- Virtual keyboard for text entry
- Voice commands

Mac OS X GUI

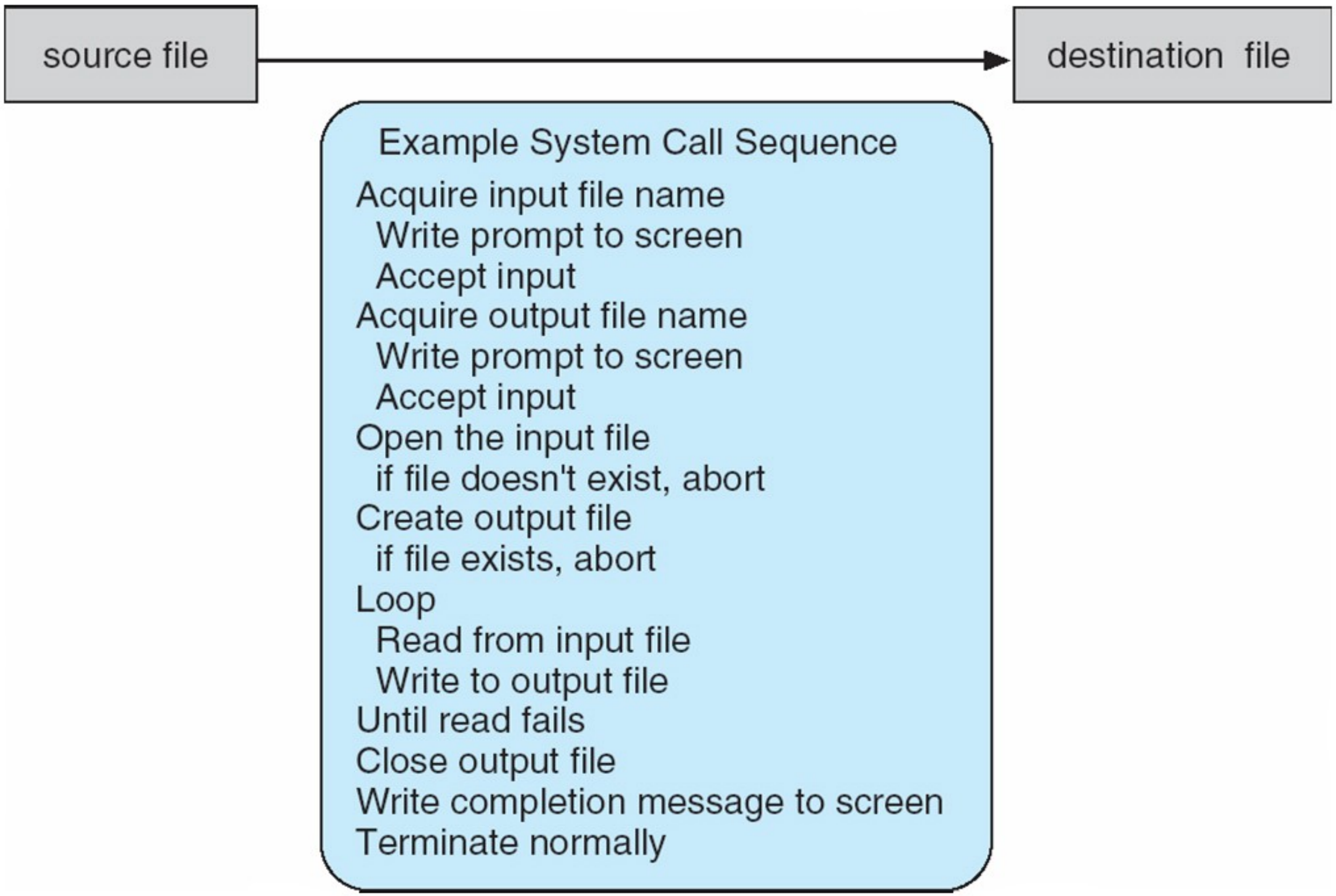

Grab File Edit Capture Window Help
15:06 EDT Mon 2 Jul
fig-dir
Address Book
Apple Computer Inc.
main 1-800-MY-APPLE
other 800-275-2273
home page http://www.apple.com
work 1 Infinite Loop
Cupertino CA 95014
United States
Dictionary and Thesaurus
operating system
Dictionary
op•er•at•ing sys•tem
noun
the software that supports a computer's basic functions, such as scheduling tasks, executing applications, and controlling peripherals.
PCalc
TIFF
Trash

System Calls

- Programming Interface to Services Provided by OS
- Typically Written in a High-Level Language (C/C++)
  - Some low-level and most-frequent tasks written in Assembly
- Examples
  - A system call to control a process
    - CreateProcess()
    - ExitProcess()
  - A system call to manipulate a file
    - ReadFile()
    - CreateFile()

Example of System Calls

- System Call Sequence to Copy Contents of one File to another File



System Calls (cont.)

- Mostly Accessed by Programs via API
  - Rather than direct system call use
- **Application Programming Interface (API)**
  - Specifies a set of functions that are available to an application programmer
    - Parameters passed to each function
    - Return values programmer can expect
- Three Most Common APIs
  - Win32 API for Windows
  - POSIX API for POSIX-based systems
    - Including virtually all versions of UNIX, Linux, and Mac OS X
  - Java API for Java Virtual Machine (JVM)

System Calls (cont.)

- APIs Invoke Actual System Calls
  - On behalf of application programmer
- Examples:
  - Win32 function CreateProcess()
    - Calls NTCreateProcess() in Windows kernel
- Why to Use APIs?
  - Portability
    - Programmer can expect his/her program to be compiled on any system that supports the same API
    - Some system calls are very detailed and hard to work
  - More coarse-grained component in programming
- Mostly a One-to-One Mapping
  - Between APIs and system calls

Example of Standard API

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:

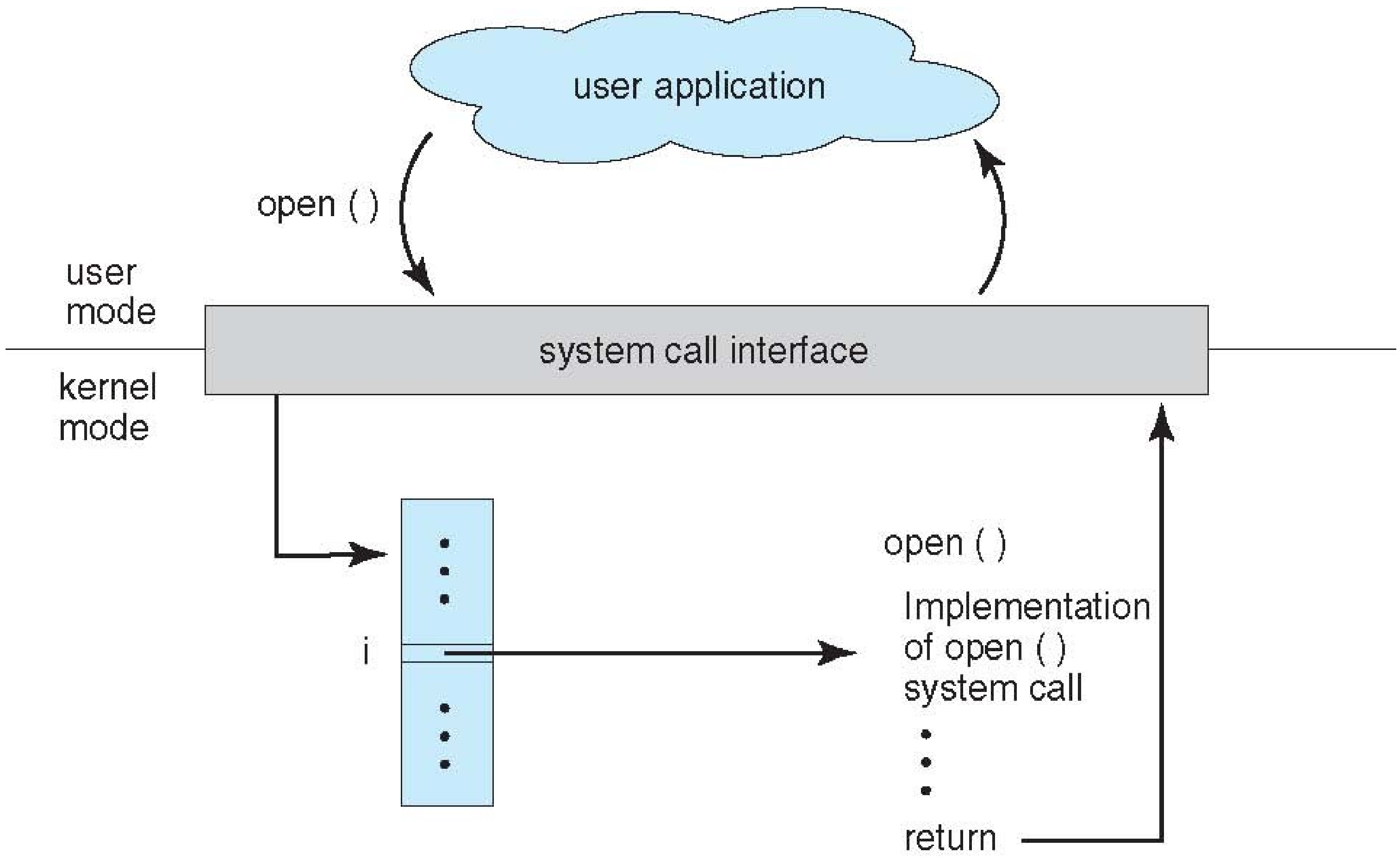
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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

System Call Implementation

- **System-Call Interface**
  - Invokes intended system call in OS Kernel
  - Then returns status of system call and any return values
- Caller Needs Know Nothing about how 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

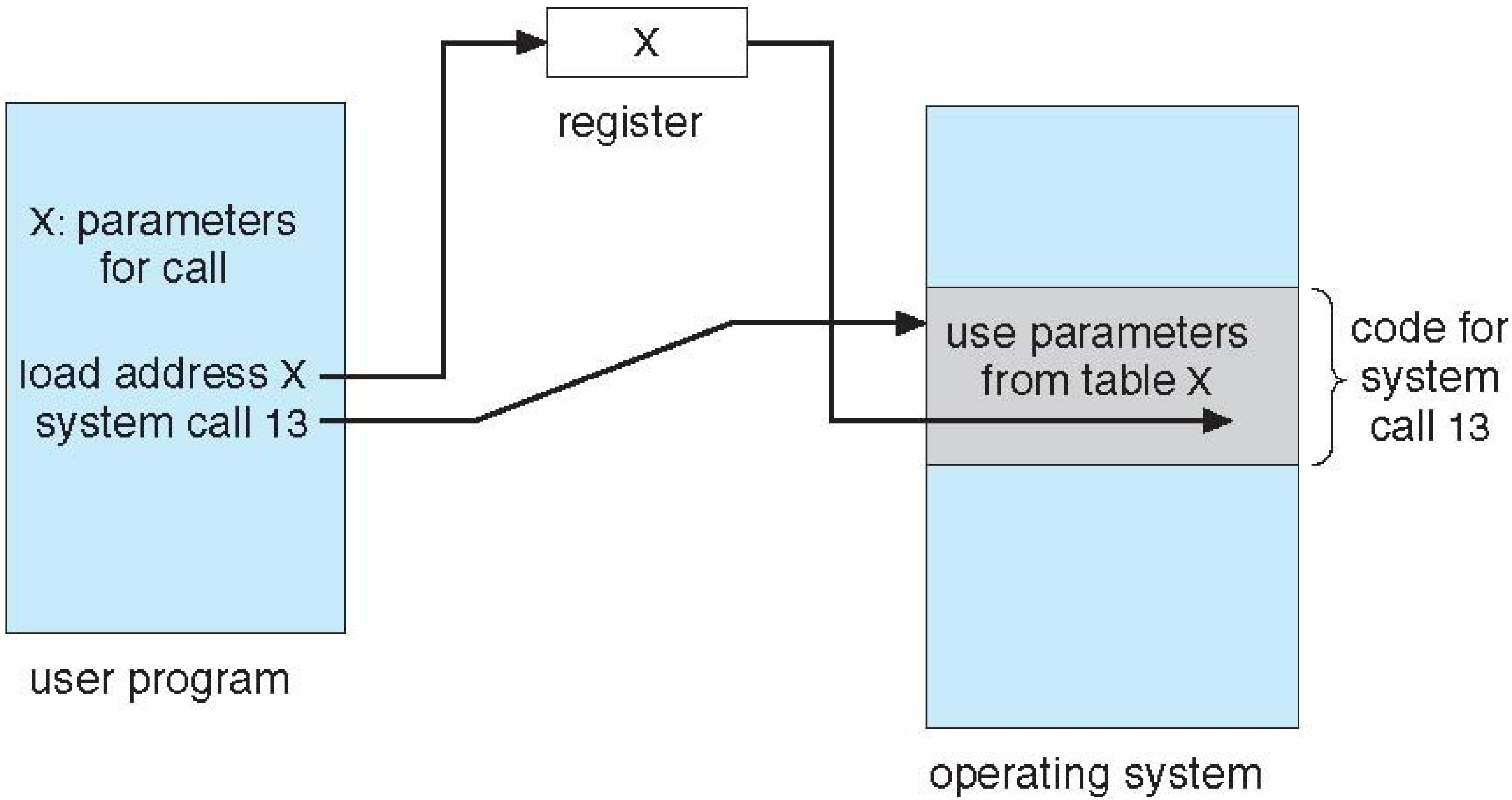
API – System Call – OS Relationship



System Call Parameter Passing

- Often, More Information Required than Simply Identity of Desired System Call
  - Exact type and amount of information vary according to OS and call
- Three Methods Used to Pass Parameters to OS
  - Simplest: pass 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 **stack** by program and **popped** off the stack by OS
  - Block and stack methods do not limit number or length of parameters being passed

Parameter Passing via Table

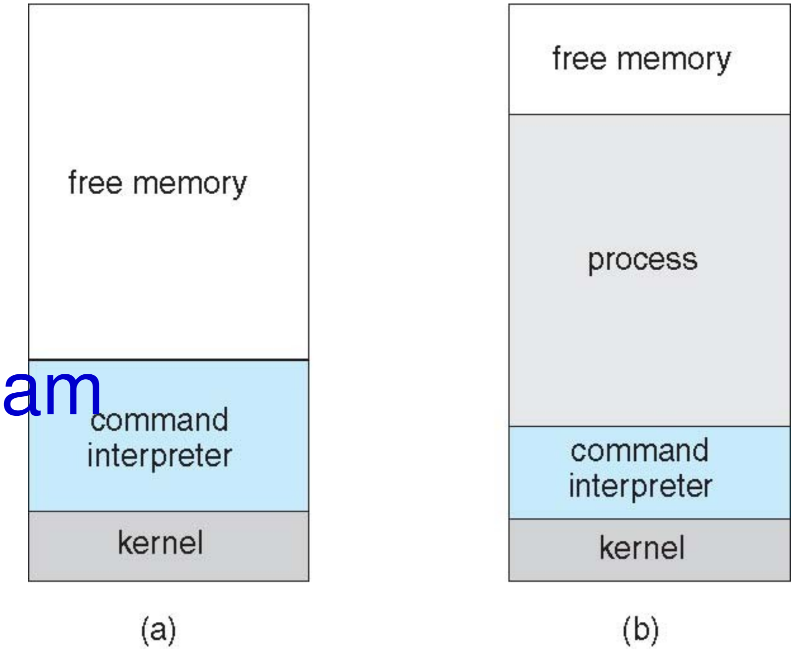


Types of System Calls

- **Process Control**
  - End, abort
  - Create process, terminate process, e.g., fork()
  - Load, execute
  - Get process attributes, set process attributes: e.g., getpid
  - Wait for time: e.g. wait()
  - Wait event, signal event
  - Allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs**, **single step** execution
  - **Locks** for managing access to shared data between processes or release lock

Example: MS-DOS

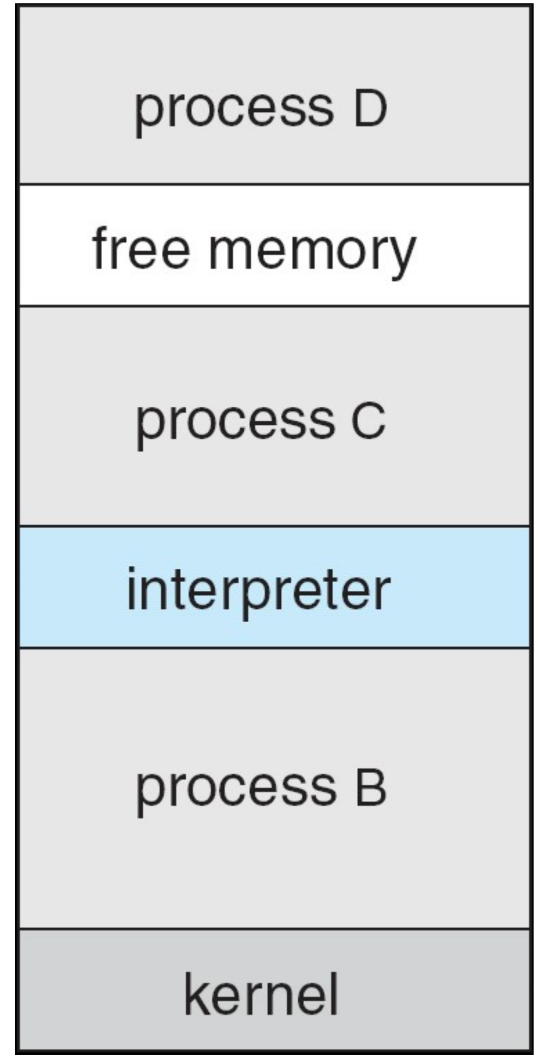
- Single-Tasking
- Shell Invoked when System Booted
- Simple Method to Run Program
  - No process created
- Single Memory Space
- Loads Program into memory, overwriting all but kernel
- Program exit ➔ shell reloaded



At system startup   running a program

Example: FreeBSD

- Unix Variant
- Multi-Tasking
- Shell Executes fork() System Call to Create Process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process Exits with:
  - (code = 0): no error
  - (code > 0): error code



Sample Code to Call System Call

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    fork(); //make a child process of same type
    sleep(2);
    printf("Fork testing code\n");
    return 0;
}
```

Types of System Calls (cont.)

- File Management
  - Create file, delete file
    - create()
  - Open, close file
    - open(), close()
  - Read, write, reposition
  - Get and set file attributes
    - File attributes: file name, file type, protection codes, accounting info, etc.
  - Copy or move a file
    - rename(), …

Types of System Calls (cont.)

- Device Management
- Various Resources Controlled by OS can be Thought as **Devices**
  - Physical devices: main memory, disk drives
  - Virtual devices: files
  - Many OSes such as UNIX do not distinguish between physical and virtual devices
    - ➔ Use a set of similar system calls

Types of System Calls (cont.)

- Device Management
  - Request device, release device
  - Read, write, reposition
  - Get device attributes, set device attributes
  - Logically attach or detach devices

Types of System Calls (cont.)

- Communications
  - Create, delete communication connection
  - Send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
    - **get hostid(), get processid()**
    - **open() & accept() to establish a connection**
    - **close() to terminate the communication**
    - Message passing more suitable for small requests
  - **Shared-memory model**
    - Create and gain access to memory regions
    - Form of data in shared memory: under process control
    - Maximum speed and convenience of communication

Types of System Calls (Cont.)

- Information Maintenance
  - Get time or date, set time or date
  - Get system data, set system data
  - Number of current users
  - OS version
  - Get and set process, file, or device attributes
  - Debugging info
    - E.g., trace program (where lists each system call as it is executed)
    - Single step run provided by microprocessors

Types of System Calls (Cont.)

- Protection
  - Control access to resources
  - Get and set permissions
    - get permission()
    - set permission()
  - Allow and deny user access

Examples of Windows and Unix System Calls

	Windows	Unix

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

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

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

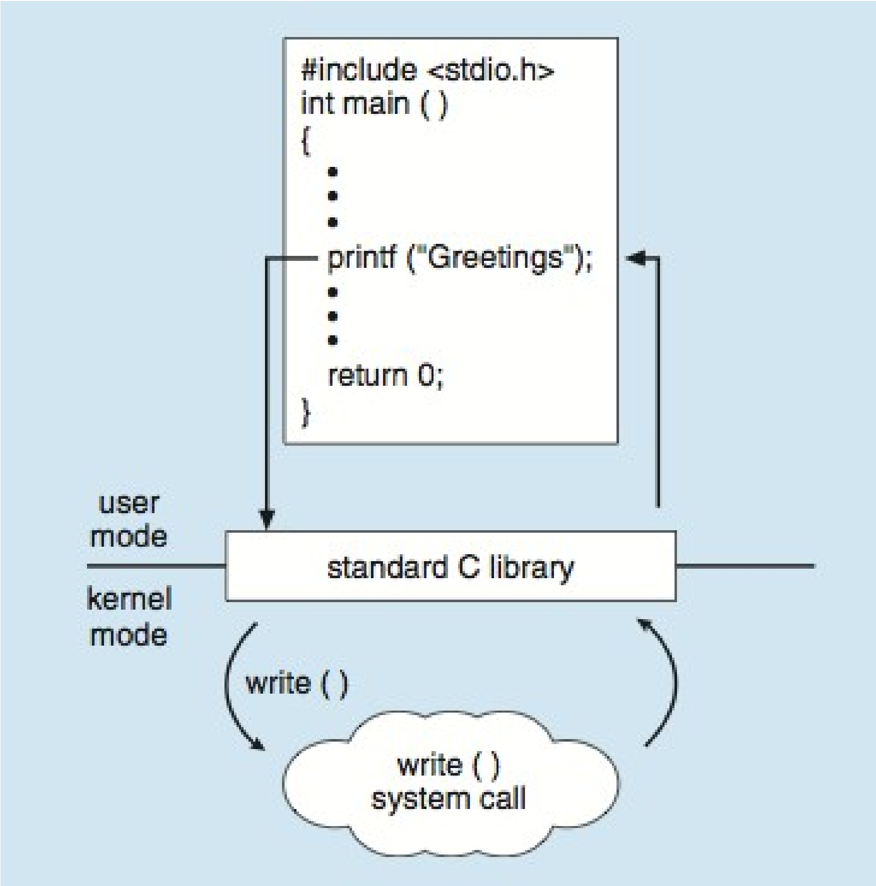
Information Maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communication	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shmget()`
		`mmap()`

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

Standard C Library Example

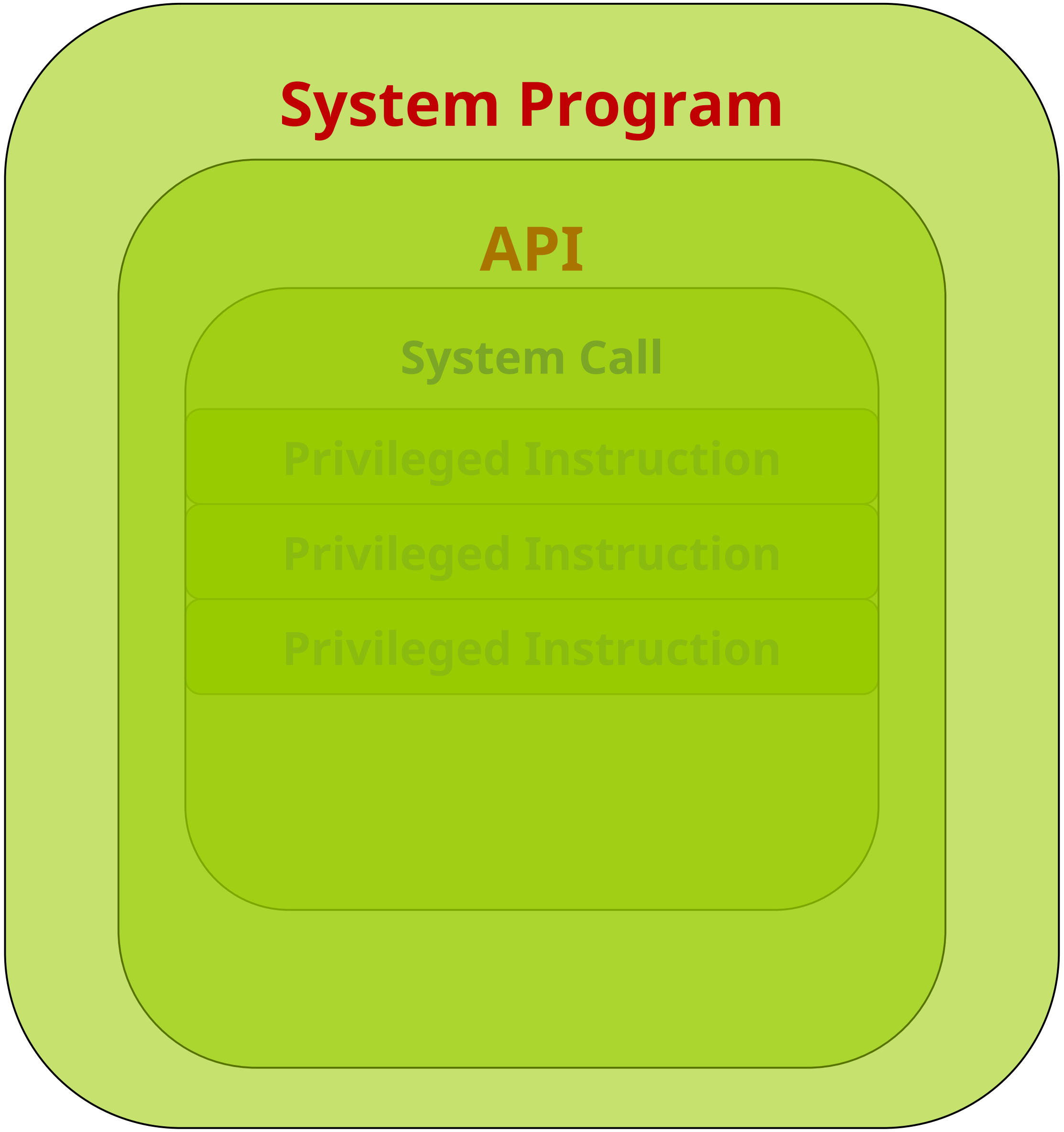
- C Program Invoking printf() Library Call, which Calls write() System Call



System Programs

- Provide a Convenient Environment for Program Development and Execution
  - Aka, **system utilities**
- Most Users' View of OS defined by System Programs, **not Actual System Calls**
- Some are Simply User Interfaces to System Calls
  - Others are considerably more complex

Privileged Instruction to System Program



System Programs (cont.)

- Can be Divided into:
  - File manipulation
  - Status information
  - File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs

System Programs (cont.)

- **File Management**
  - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status Information**
  - Some ask system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print output to terminal or other output devices
  - Some systems implement a **registry** - used to store and retrieve configuration information

System Programs (Cont.)

- **File Modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-Language Support**
  - Compilers
  - Assemblers
  - Debuggers and interpreters
  - Such system programs provided for C, C++, Java, Visual Basic, Perl, etc.

System Programs (Cont.)

- **Program Loading and Execution**
  - Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications**
  - Provide mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

System Programs (Cont.)

- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Known as **services**, **subsystems**, **daemons**
- **Application Programs** (such as web browsers, spreadsheets, database systems, games)
  - Don’t pertain to system
  - Run by users
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke

OS Design and Implementation

- Design and Implementation of OS
  - Internal structure of different OSs can vary widely
  - Start design by defining goals and specifications
    - Throughput, energy efficiency, response time, real-time, user-friendly
  - Also affected by:
    - Choice of HW
    - Type of system: single user, multiuser, real-time, distributed, general purpose
    - E.g. 1: VxWorks: real-time OS for embedded systems
    - E.g. 2: MVS: a large multiuser, multi-access OS (IBM)

OS Design and Implementation (cont.)

- **Important Q: User goals and System goals**
  - User goals – OS should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – OS should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

OS Design and Implementation (Cont.)

- Important Principle to Separate: Policy vs. Mechanisms
  - Policies: decide what will be done
  - Mechanisms determine how to do sth
- Example: Timer
  - Timer construct: a mechanism used for ensuring CPU protection
  - Policies
    - How long a process can acquire a CPU?
    - Timer is given to user processes only for 10ms
- **Separation** of Policy from Mechanism
  - Allows maximum flexibility if policy decisions are to be changed later

Implementation

- Much Variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Usually a Mix of Languages
  - Lowest levels in assembly while main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts

Implementation (cont.)

- High-Level Languages
  - Faster code development ☺
  - Compact
  - Easy to understand and debug
  - Improvements in compiler further optimizes code
  - Easier to **Port** to other HW
  - Slower ☹
  - Increased storage/memory ☹
- Assembly Languages
  - Faster ☺
  - Only can be run on target ISA ☹

Implementation (cont.)

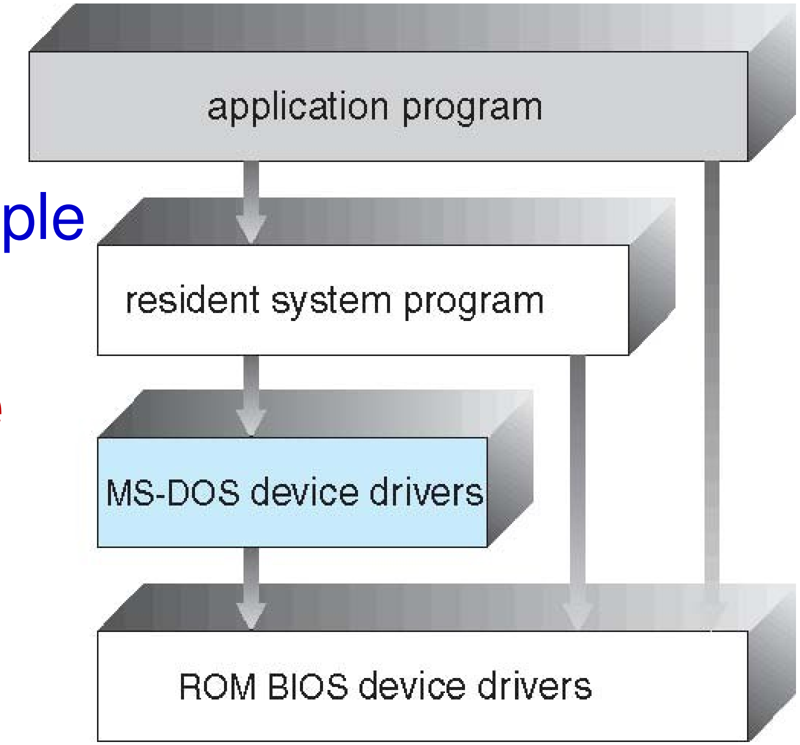
- Hybrid Solution
  - OS is large
  - But small fraction of code is performance bottleneck
- Critical Sections of OS
  - CPU scheduler → Assembly
  - Memory manager → Assembly
  - Other Parts → C/C++

Operating System Structure

- General-Purpose OS is Very Large Program
- Various Ways to Structure Ones
  - Simple structure – MS-DOS
  - More complex – UNIX
  - Layered – an abstraction
  - Microkernel – Mach

Simple Structure: MS-DOS

- History
  - Originally designed and implemented by very few people
- Written to Provide Most Functionality in Least space
- Not Well-Defined Structure
  - Not divided into modules
- Interfaces and Levels of Functionality Not Well Separated
  - Although it has some structure
- Very Vulnerable to Errant or Malicious Programs

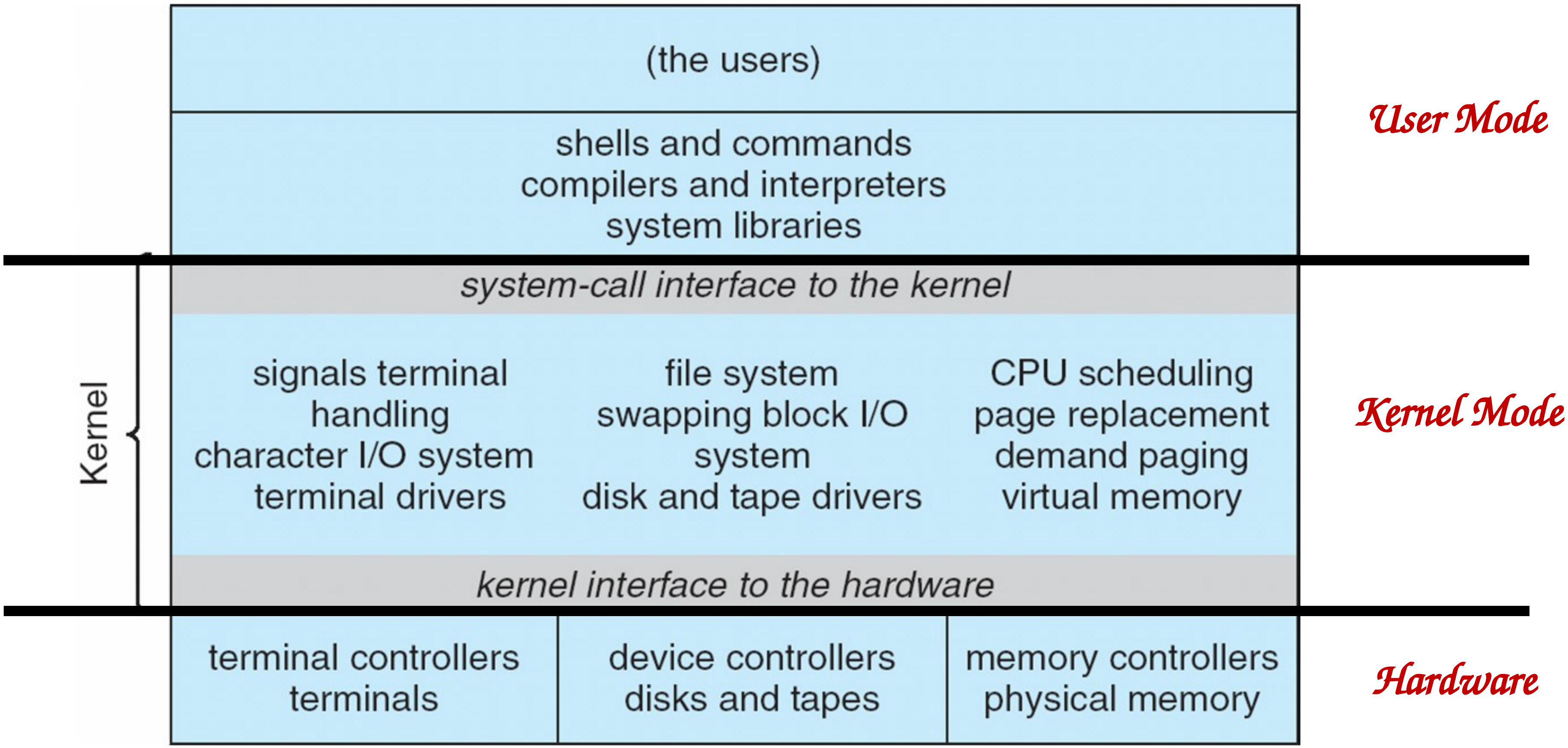


Non-Simple Structure – UNIX

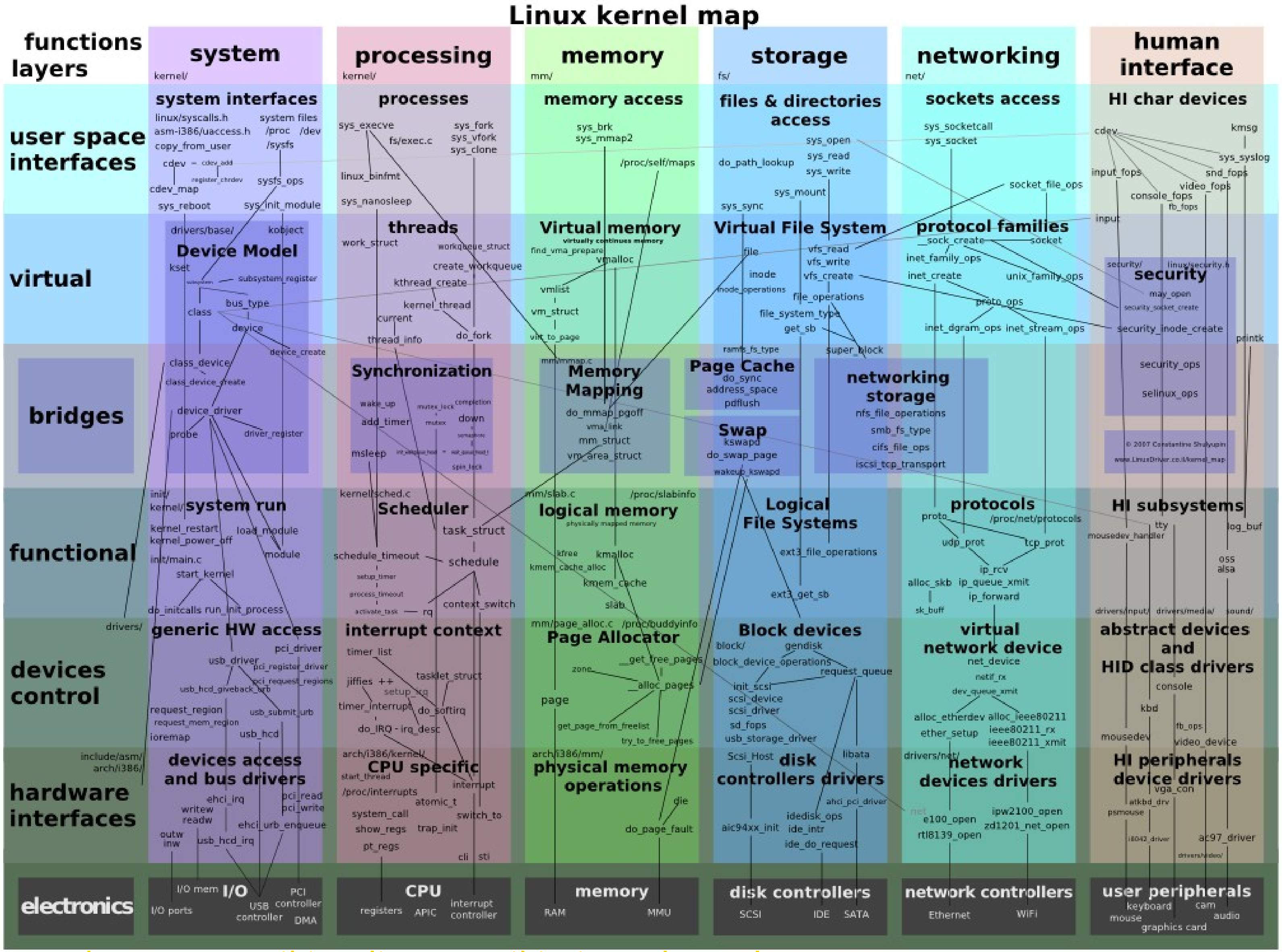
- Limited by HW Functionality
  - Original UNIX OS had limited structuring
- Consists of Two Separable Parts
  - Systems programs
  - Kernel
    - Consists of everything below system-call interface and above physical HW
    - Provides file system, CPU scheduling, memory management, and other OS functions
    - A large number of functions for one level

Traditional UNIX System Structure

Beyond simple but not fully layered

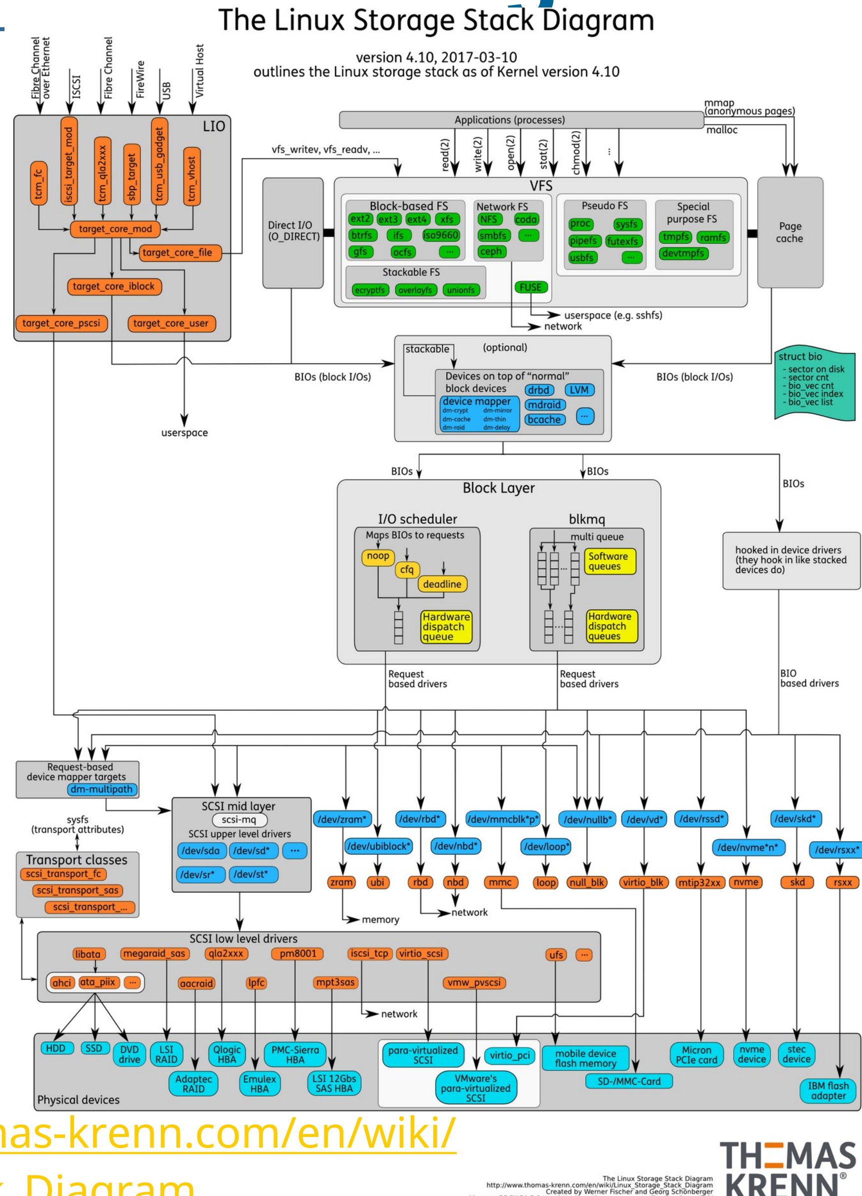


Linux Structure



© https://en.wikipedia.org/wiki/Linux_kernel

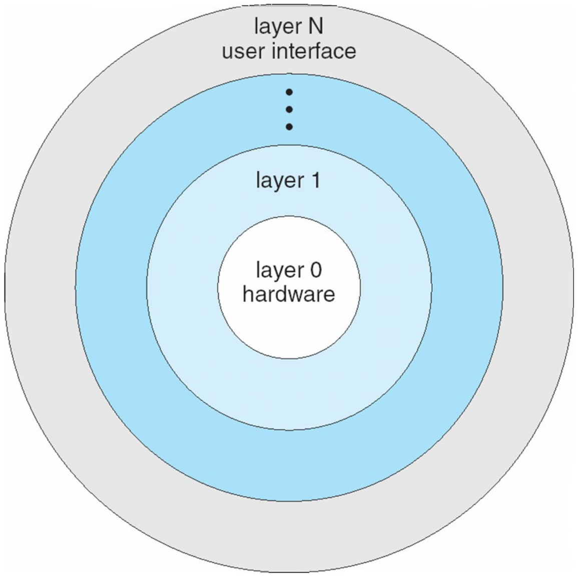
Linux Storage Stack



[© https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram](https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram)

Layered Approach

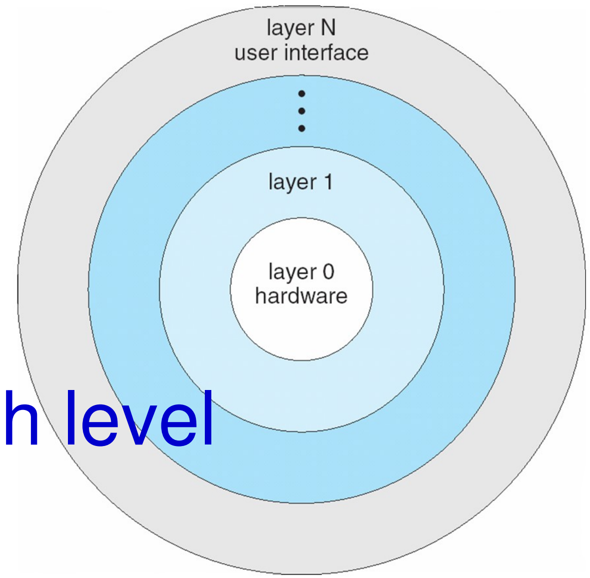
- OS Divided into a Number of Layers (Levels)
  - Each built on top of lower layers
  - Bottom layer (layer 0): HW
  - Highest (layer N): user interface



- Modularity
  - Layers are selected such that each uses functions (operations) and services of only lower-level layers

Layered Approach (cont.)

- **Pros**
  - Simplicity of construction
  - Simplified verification scheme
  - Ease of debugging
  - Can easily update or extend each level



- **Cons**
  - Difficulty to define layers
  - Layers may need to interact with various layers
  - Too many layers ➔ performance degradation

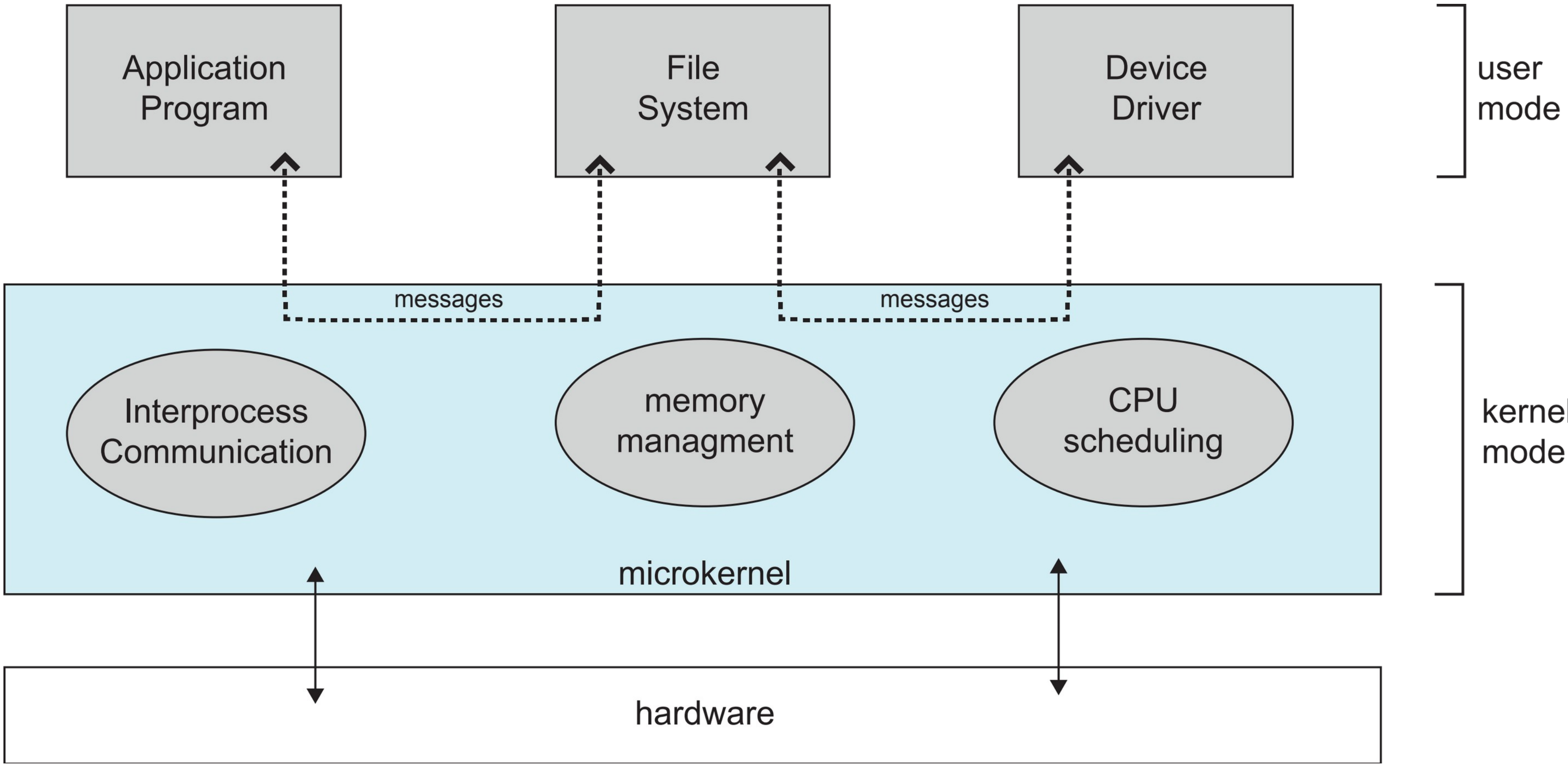
Microkernel System Structure

- Moves as Much from Kernel into User Space
  - Non-essential components implemented as system and user-level programs
  - ➔ Results in much smaller kernel called microkernel
- Microkernel
  - Provide minimal process and memory management and communication facility
- **Mach** Examples of **Microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
  - QNX: a real-time OS
  - True64 UNIX (or formerly Digital UNIX)

Microkernel System Structure (cont.)

- Communication Takes Place between User Modules using **Message Passing**
- Benefits
  - Easier to extend a microkernel
  - Easier to port OS to new architectures
    - Fewer modification needed
  - More reliable (less code running in kernel mode)
  - More secure
- Detriments
  - Performance overhead of user space to kernel space communication (e.g., WinNT vs. Win 95)

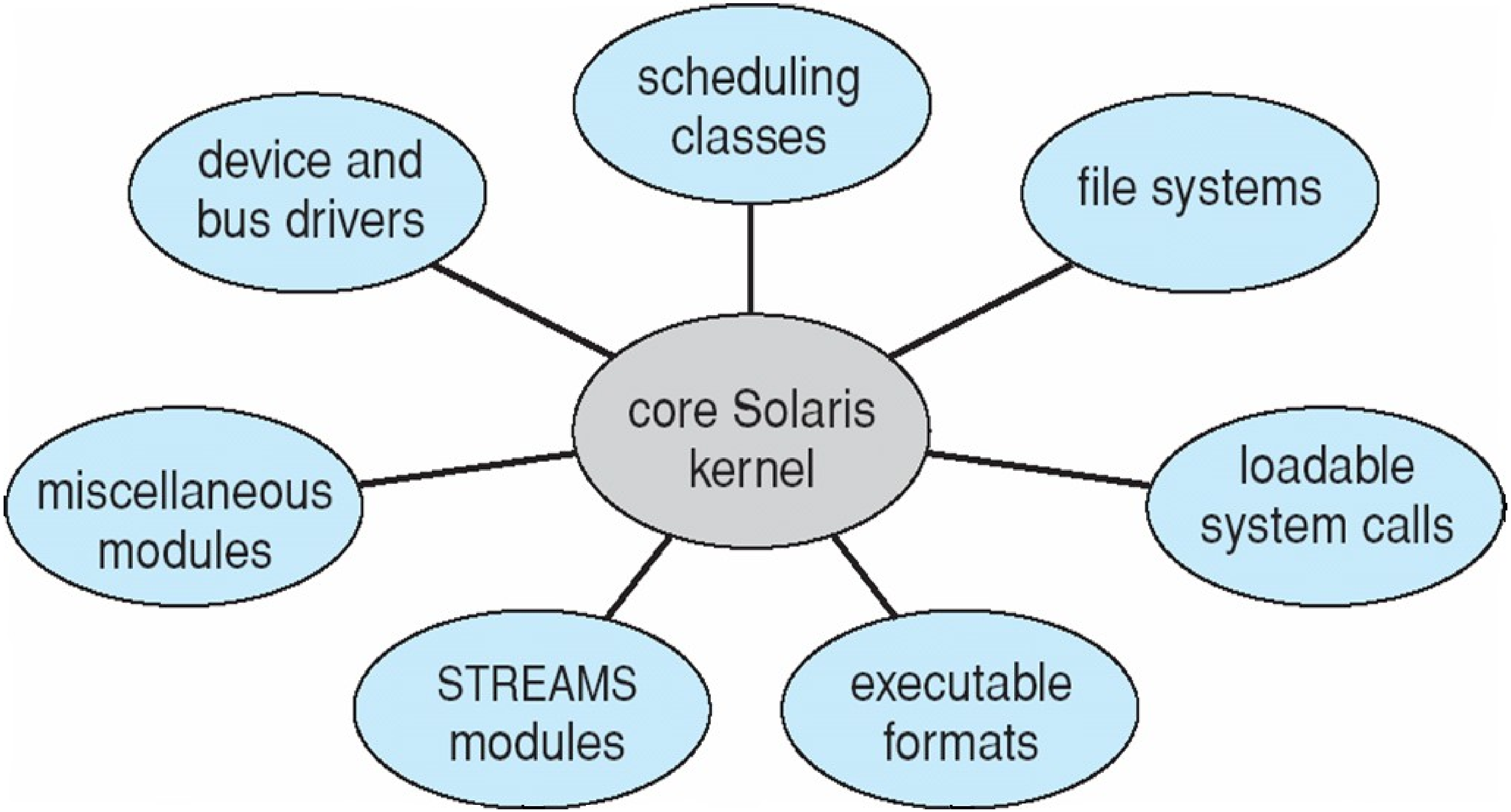
Microkernel System Structure (cont.)



Modules

- **Loadable Kernel Modules** (Used in Adv. OS)
  - Components can be loaded either at boot time or during run time
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to others over known interfaces
  - Each is loadable as needed within kernel
- Similar to Layers but with More Flexibility
  - Linux, Solaris, etc
- Similar to Microkernels but More Efficient
  - Modules do not need to invoke message passing to communicate

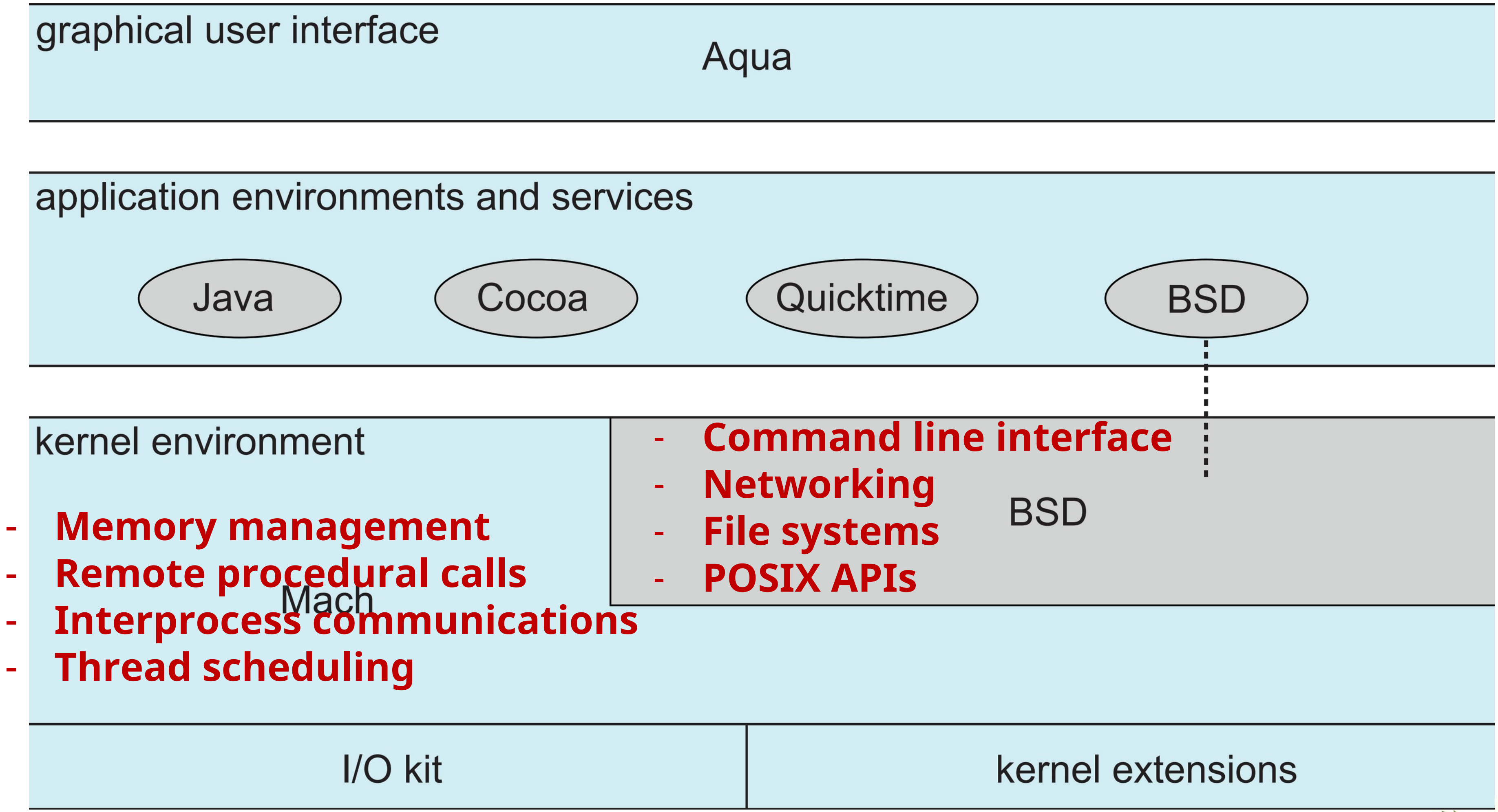
Solaris Modular Approach



Hybrid Systems

- **Most Modern OSes actually not One Pure Model**
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem ***personalities***
- Apple Mac OS X hybrid, Layered
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

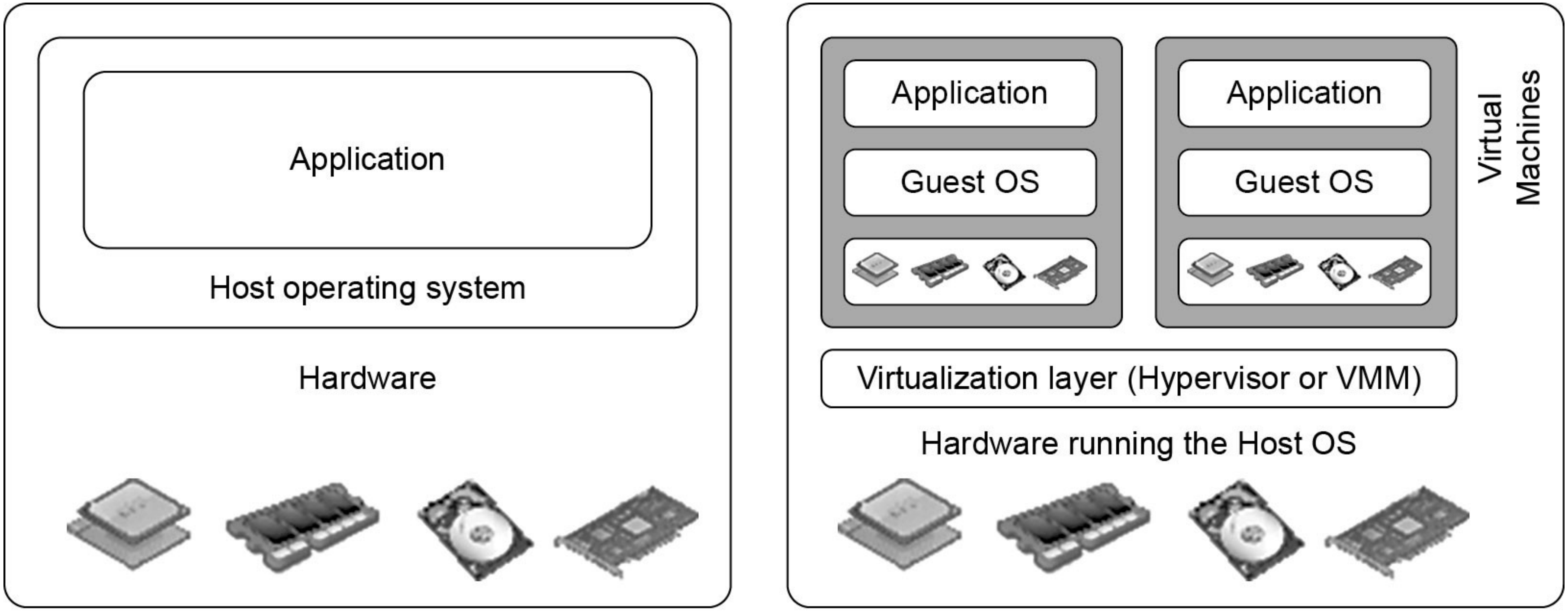
Mac OS X Structure



Virtual Machines

- **Features**
  - To abstract HW of a single computer into several different execution environments
  - Creates an illusion that each environment running its own private computer
  - Allows OSes to run applications within other OSes
    - Vast and growing industry
  - Host OS runs several guest OSes as processes
  - First appeared commercially on IBM mainframes
    - As VM operating system in 1972

Virtual Machines (cont.)



(a) Traditional computer

(b) After virtualization

(Courtesy of VMWare, 2008)

Virtual Machines (cont.)

- **Hypervisor (HV): Virtual Machine Monitor**
  - SW/HW that creates and runs virtual machines
- **Type 1**
  - HV runs directly on host's HW to control HW and to manage guest OSes
  - A guest OS thus runs on another level above HV
- **Type 2**
  - HV runs within a conventional OS environment
  - With HV layer as a distinct second SW level, guest OS run at third level above HW

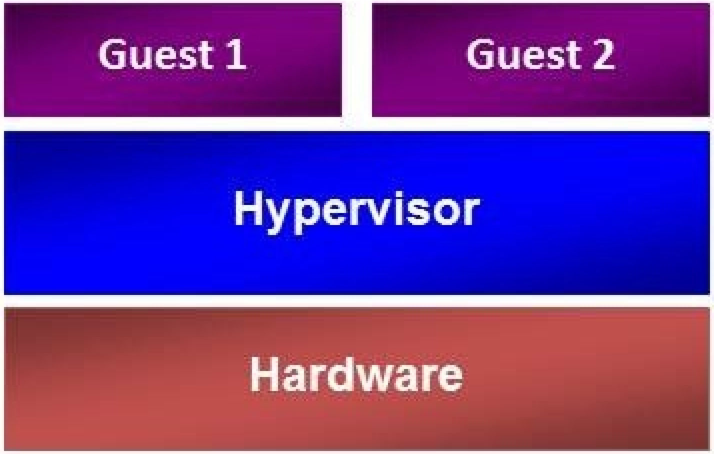
Hypervisor Design:
Two approaches

Type 2 Hypervisor

Examples:

- Virtual PC & Virtual Server
- VMware Workstation
- KVM

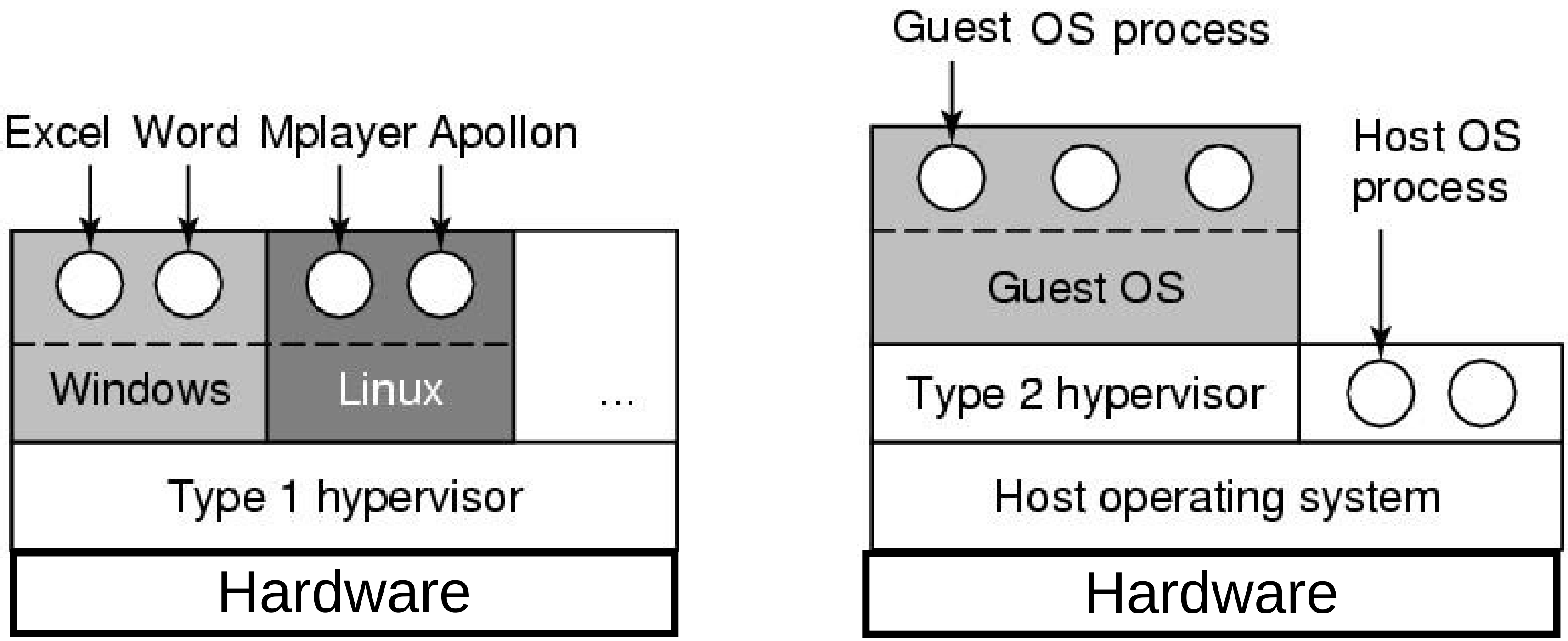
Type 1 Hypervisor



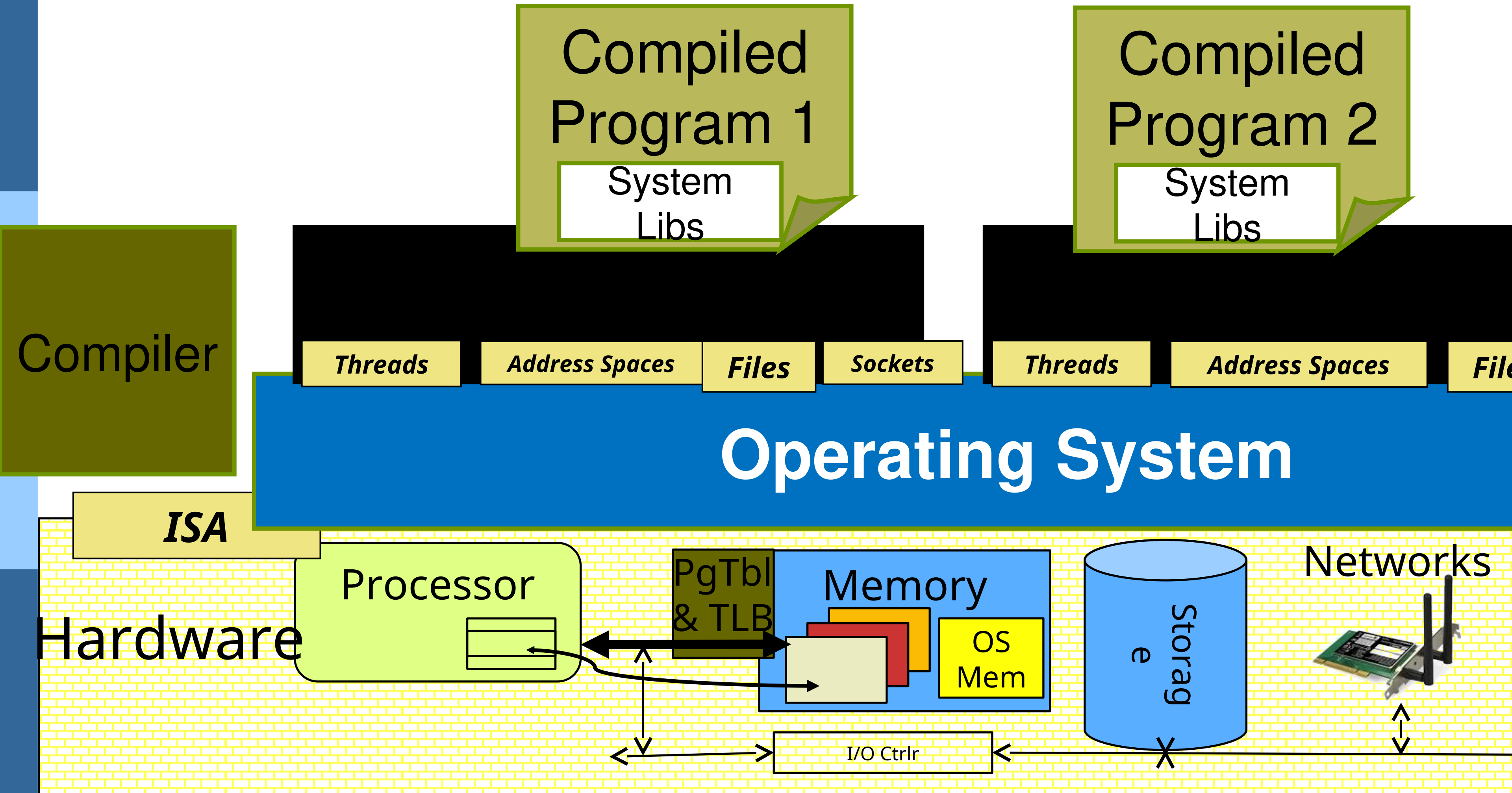
Examples:

- Hyper-V
- Xen
- VMware ESX

Virtualization (cont.)



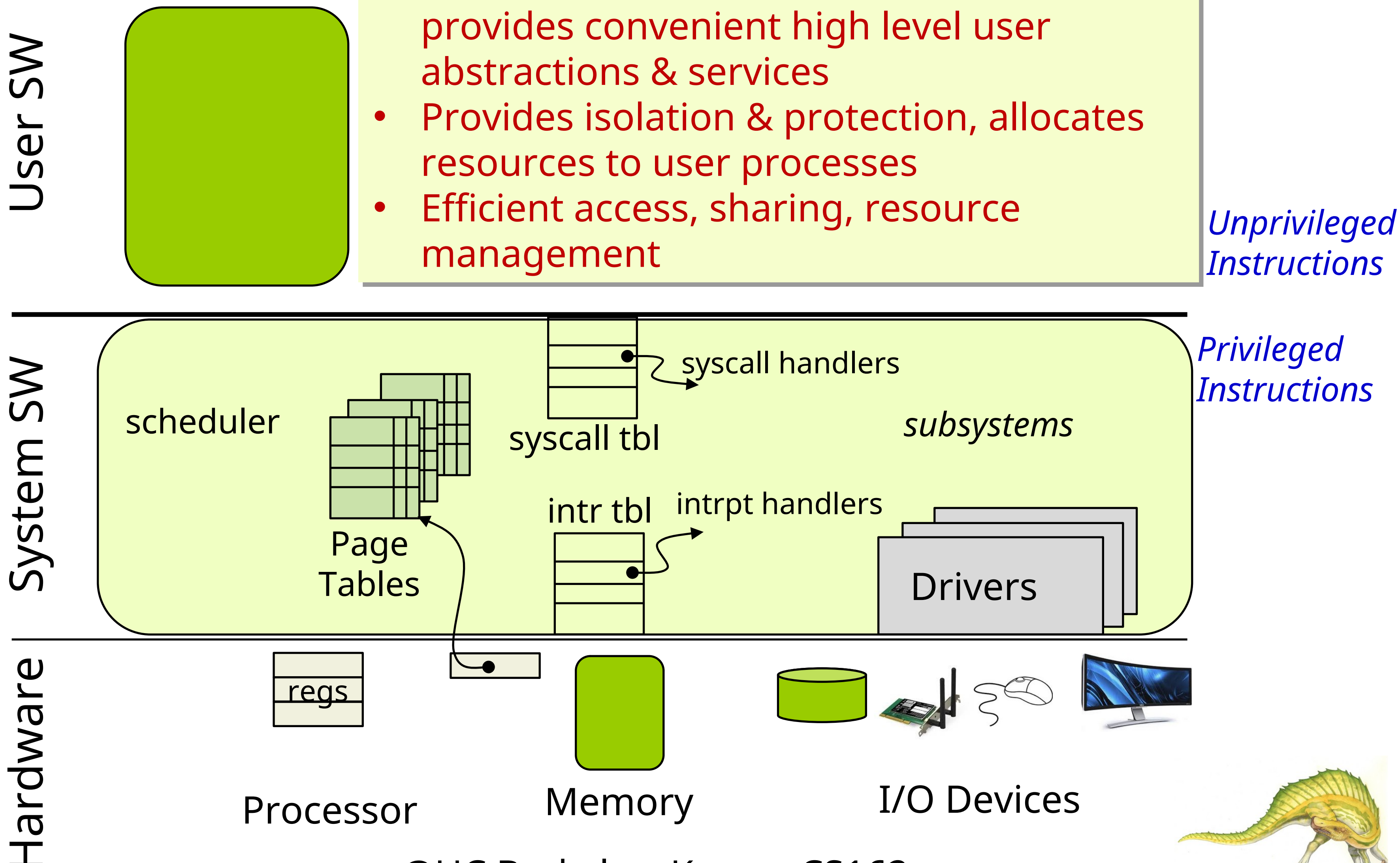
Virtualization (cont.)



@UC Berkeley, Kumar CS162

OS Software Supports User-Level SW

- Virtualizes hardware resources and provides convenient high level user abstractions & services
- Provides isolation & protection, allocates resources to user processes
- Efficient access, sharing, resource management



@UC Berkeley, Kumar CS162

Virtual Machines (cont.)

- **Main Benefits**
  - Guest OSes do not have to be compliant with HW
    - Making it possible to run different OSes on same computer
  - Host system protected from other VMs
    - A virus in a guest OS damage OS itslef but is unlikely to affect the host or other guest OSes
  - Alleviates system development
    - System development is very dangerous on non-virtualized OS
  - A VM can easily be relocated from one physical machine to another as needed

Virtual Machines (cont.)

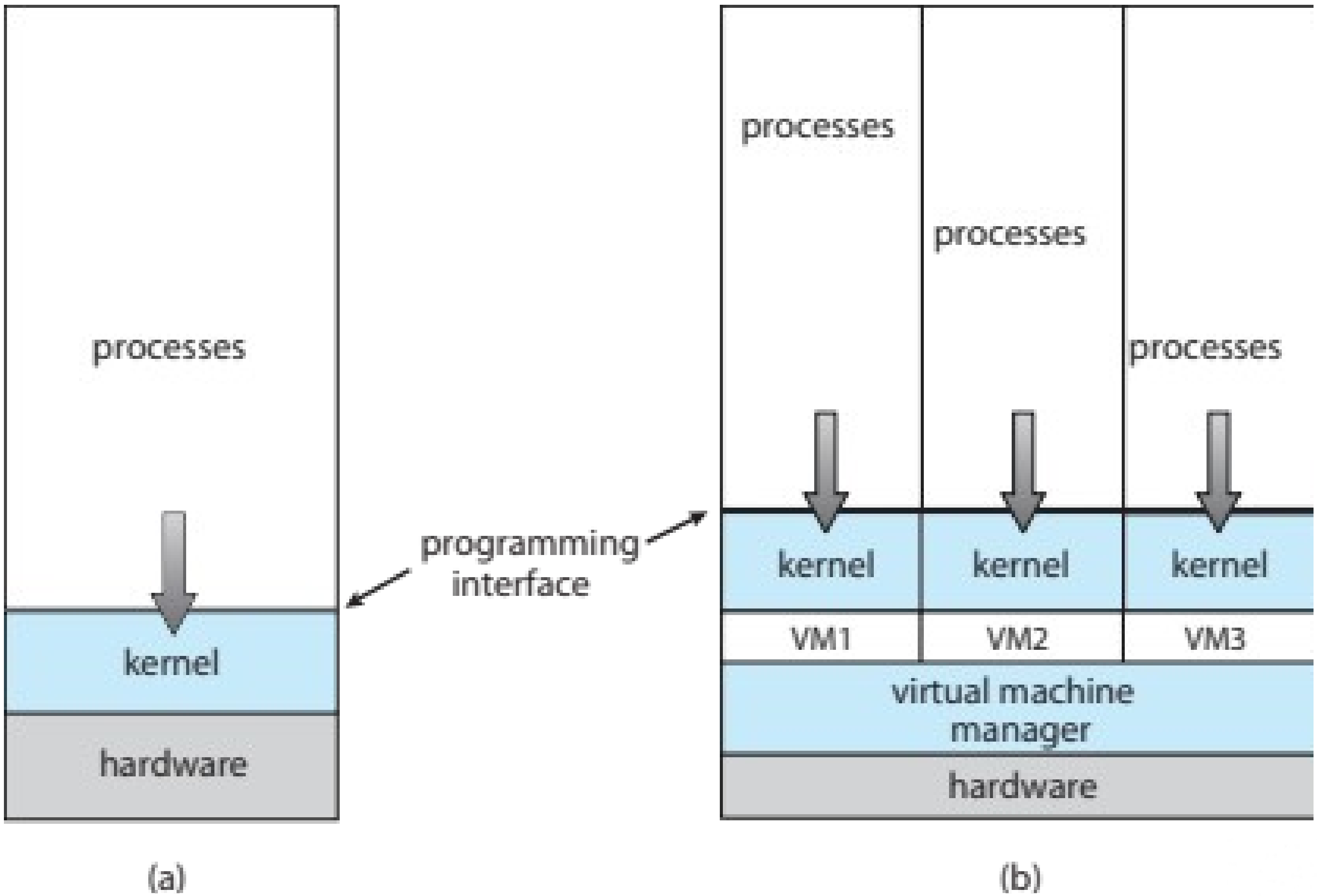
Main Benefits

- Rapid porting and testing of program
  - Quality assurance engineers can test their programs on multiple platforms on a single HW
- Can consolidate multiple systems in a single HW
  ➔ Significant power efficiency in data centers
- Can offer pre-installed applications on VMs
  - Easy to install and config new applications on systems
  - Technical support of applications becomes straightforward (do not need to tune or config for different platforms)

Virtual Machines (cont.)

- **Virtualization**
  - Guest OS & applications run at almost full speed
  - Only system's resources need to be virtualized
    - VMware and Java Virtual Machine (JVM)
    - ISA is not virtualized
- **Emulation**
  - ISA is virtually implemented (or emulated)
  - Used when source CPU type different from target type (i.e. PowerPC to Intel x86)
  - Generally slowest method (by 10X)

Virtual Machines (cont.)

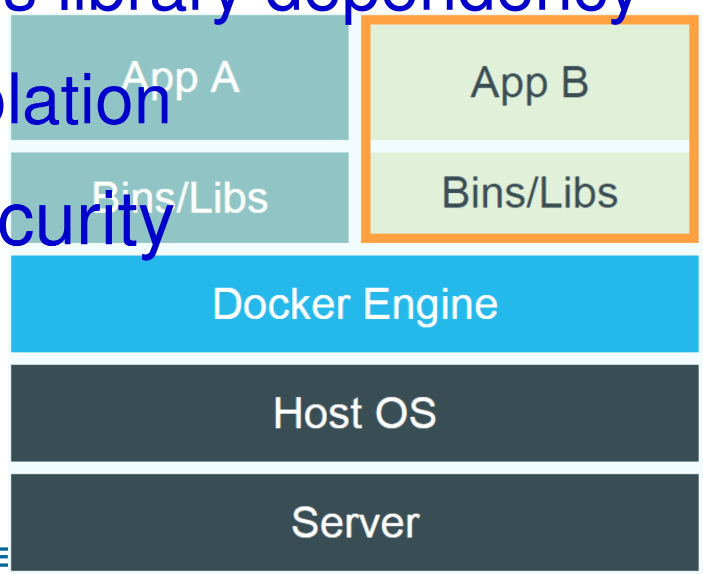


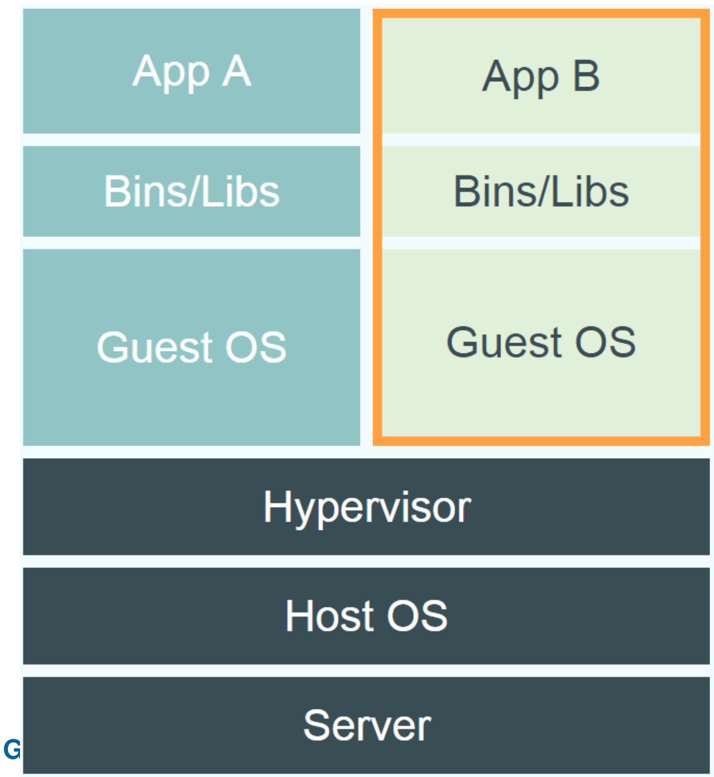
Virtual Machines (cont.)

- **Implementation**
  - Underlying machine: user and kernel mode
  - HV can run in kernel mode
  - VMs can execute only in user mode
  - VM: virtual user mode and virtual kernel mode
    - Both runs in physical user mode
  - ➔ This could adversely affect performance
  - This requires some sort of HW mode support for virtualization: called host mode or HV mode
    - Example: AMD supports host mode vs. guest mode

Docker/Container vs. Virtualization

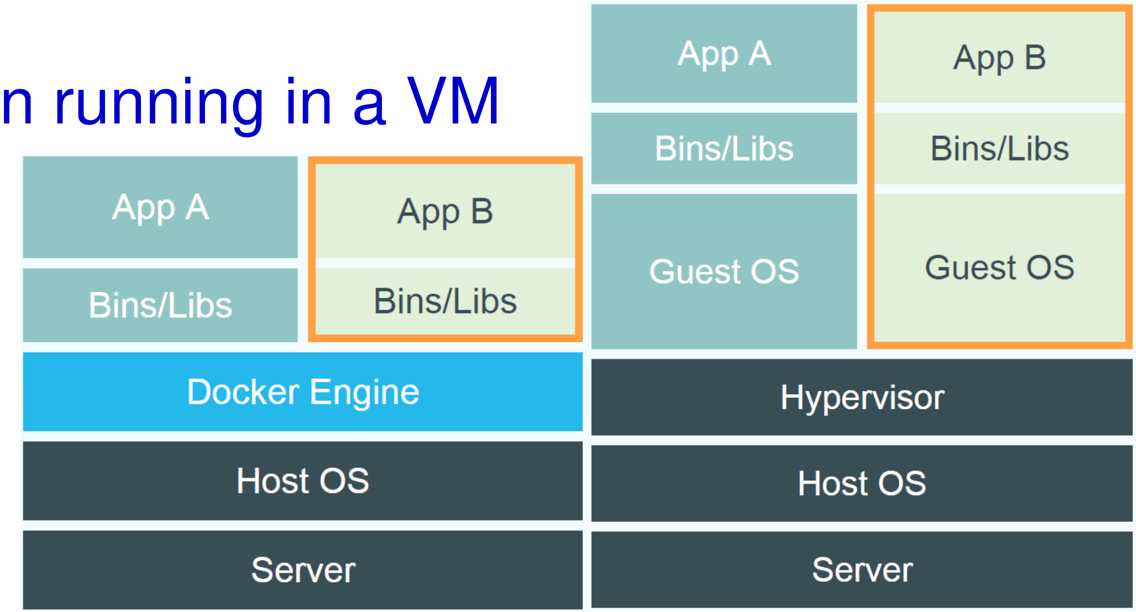
- Container/Docker
  - It does OS virtualization (not HW virtualization)
  - OS-Level virtualization to deliver SW in packages
  - Share one OS (no multiple guest OS)
  - Higher performance
  - Resolves library dependency
  - Less isolation
  - Less security



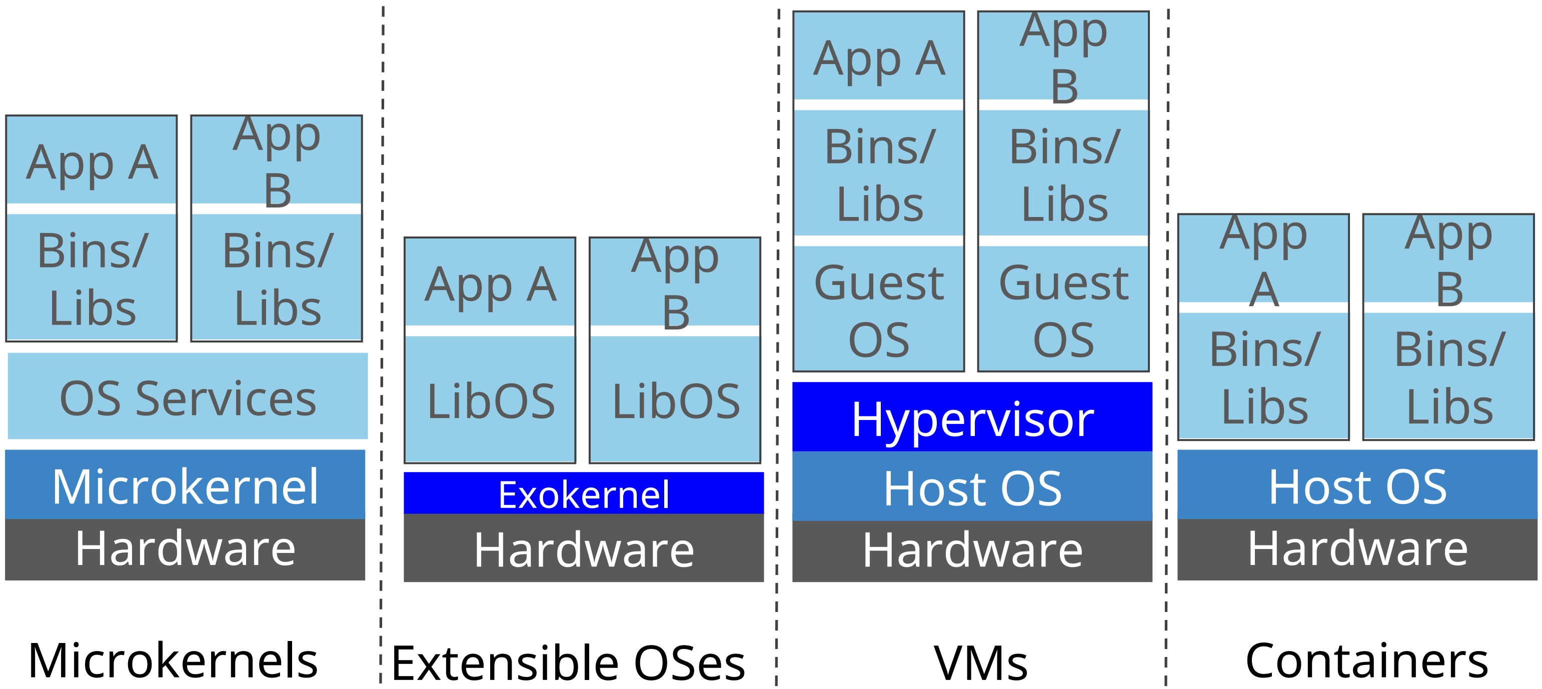


Docker/Container vs. Virtualization

- Container/Docker
  - Each container contains *only* application & its libraries and dependencies
  - Small, fast, and portable
  - A containerized application has more direct access to HW
    - Vs. an application running in a VM



VMs vs. Containers vs. Microkernels



© http://www.eecs.berkeley.edu/~kubitron/cs262

By John Kubiatowicz

Reading Assignment

Reading Assignments

- **OS Debugging**
  - **Debugging** is finding and fixing errors, or **Bugs**
  - OS generate **log files** containing error Info
- Performance Monitoring and Tuning
  - Top command
- Dtrace
- Operating System Generation
- Para-Virtualization
- Containers

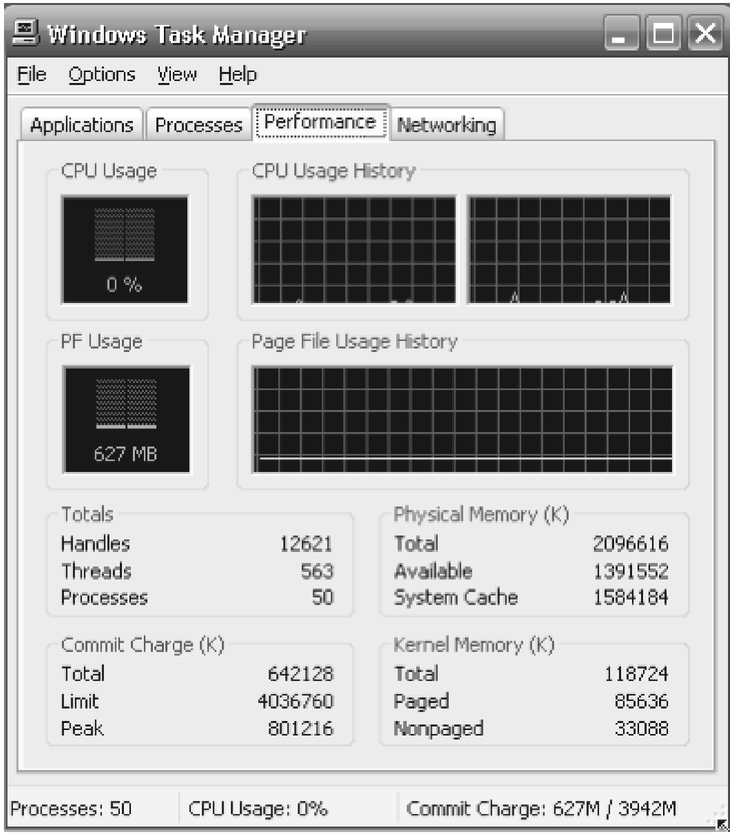
OS Debugging

- **Debugging** is Finding and Fixing errors, or **Bugs**
- OS Generate **Log Files** Containing Error Information
- Failure of an Application can Generate **Core Dump** File Capturing Memory of Process
- OS Failure can Generate **Crash Dump** File Containing Kernel Memory
- Beyond Crashes, Performance Tuning can Optimize System Performance
  - Sometimes using ***trace listings*** of activities, recorded for analysis
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

Kernighan’s Law: “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

Performance Tuning

- Improve Performance by Removing Bottlenecks
- OS must Provide Means of Computing and Displaying Measures of System Behavior
- For example, "top" program or Windows Task Manager



DTrace

- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems
- **Probes** fire when code is executed within a **provider**, capturing state data and sending it to **consumers** of those probes
- Example of following XEventsQueued system call move from libc library to kernel and back

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                       U
  0   -> _XEventsQueued                    U
  0     -> _X11TransBytesReadable          U
  0     <- _X11TransBytesReadable          U
  0     -> _X11TransSocketBytesReadable U
  0     <- _X11TransSocketBytesreadable U
  0     -> ioctl                           U
  0       -> ioctl                         K
  0         -> getf                        K
  0           -> set_active_fd             K
  0           <- set_active_fd             K
  0         <- getf                        K
  0         -> get_udatamodel              K
  0         <- get_udatamodel              K
...
  0         -> releasef                    K
  0           -> clear_active_fd           K
  0           <- clear_active_fd           K
  0           -> cv_broadcast              K
  0           <- cv_broadcast              K
  0         <- releasef                    K
  0       <- ioctl                         K
  0     <- ioctl                           U
  0   <- _XEventsQueued                    U
  0 <- XEventsQueued                       U
```

Dtrace (Cont.)

- DTrace code to record amount of time each process with UserID 101 is in running mode (on CPU) in nanoseconds

```
sched:::on-cpu
uid == 101
{
   self->ts = timestamp;
}

sched:::off-cpu
self->ts
{
   @time[execname] = sum(timestamp - self->ts);
   self->ts = 0;
}
```

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
    gnome-settings-d          142354
    gnome-vfs-daemon          158243
    dsdm                      189804
    wnck-applet               200030
    gnome-panel               277864
    clock-applet              374916
    mapping-daemon            385475
    xscreensaver              514177
    metacity                  539281
    Xorg                     2579646
    gnome-terminal           5007269
    mixer_applet2            7388447
    java                    10769137
```

Figure 2.21 Output of the D code.

Operating System Generation

- OSes Designed to Run on any of a Class of Machines
  - System must be configured for each specific computer site
- **SYSGEN** Program Obtains Info Concerning Specific Configuration of HW system
  - Used to build system-specific compiled kernel or system-tuned
  - Can general more efficient code than one general kernel

End of Lecture 2