CS 3113 Introduction to Operating Systems

Topic #2. Operating-System Structures

Outline

- 2.1 Operating System Services
- 2.2 User and Operating-System Interface
- 2.3 System Calls
- 2.4 System Services
- 2.7 Operating System Design and Implementation
- 2.8 Operating System Structure
- 2.10 Operating System Debugging

Operating System Services

- ✓ Benefit the users:
  - ➢ **User interface (UI)** - Almost all OSes have a UI
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
  - ➢ **Program Execution** - The purpose of a computer systems is to allow the user to execute programs. So, the OS provides an environment where the user can conveniently run programs.
    - The user does not have to worry about the memory allocation or multitasking or anything. These things are taken care of by the OS.

Operating System Services (Cont’d)

- **I/O Operations** - Each program requires an input and produces output. This involves the use of I/O.
  - The OS hides the details of underlying hardware for the I/O. All the user sees is that the I/O has been performed without any details.
  - By providing I/O, the OS makes it convenient for the users to run programs.
  - For efficiency and protection, users cannot control I/O so this service cannot be provided by user-level programs.

Operating System Services (Cont’d)

- **File-system manipulation** - The file system is of particular interest. Programs need to read/write files/directories, create/delete/search them, list file Information, permission management.
- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)

Operating System Services (Cont’d)

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

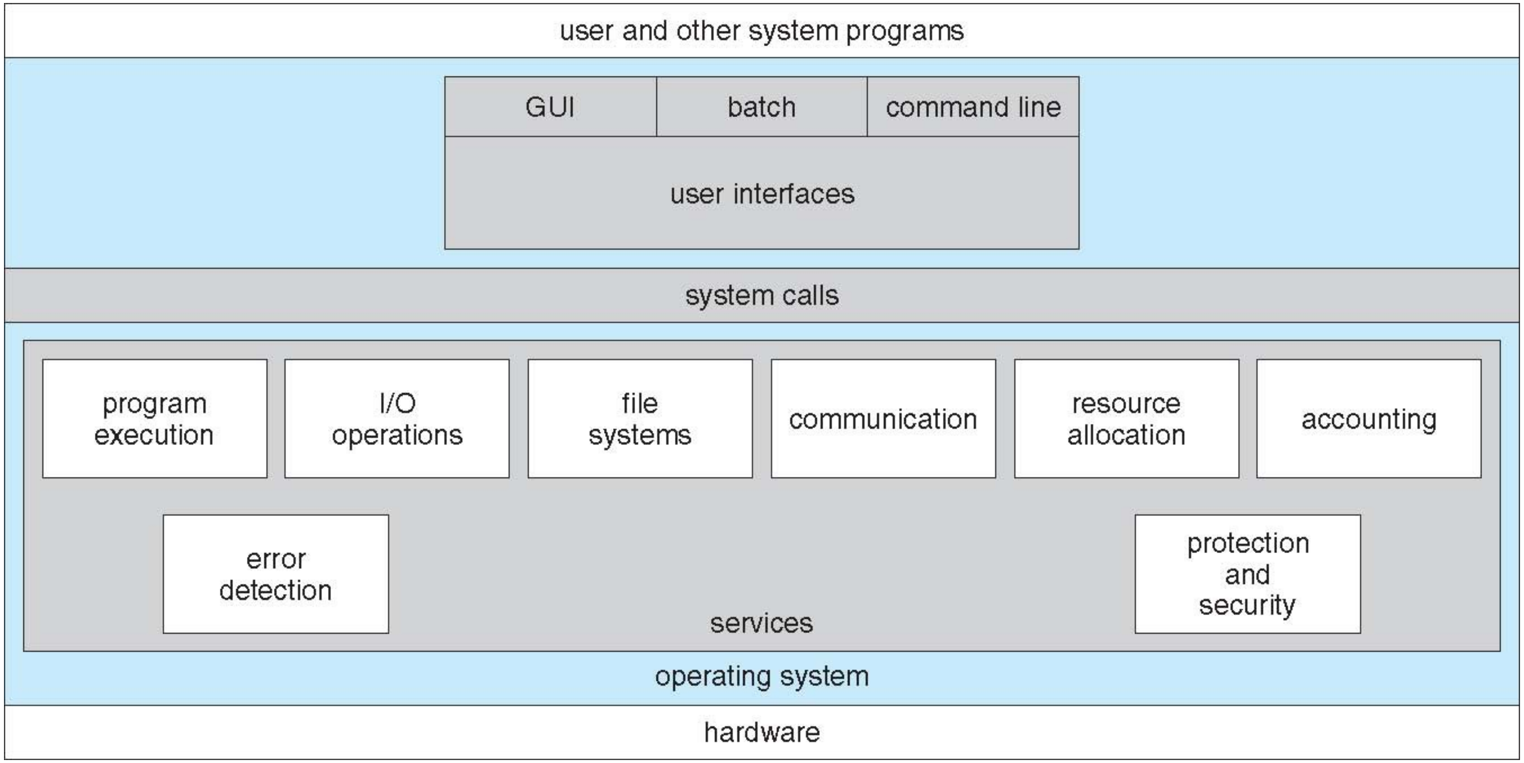
Operating System Services (Cont’d)

- ✓ Ensure the efficient operation of the system itself:
  - ➢ **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - ➢ **Accounting** - To keep track of which users use how much and what kinds of computer resources

Operating System Services (Cont’d)

- **Protection and security** - The 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 access to system resources is controlled
  - *Security* of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

A View of Operating System Services

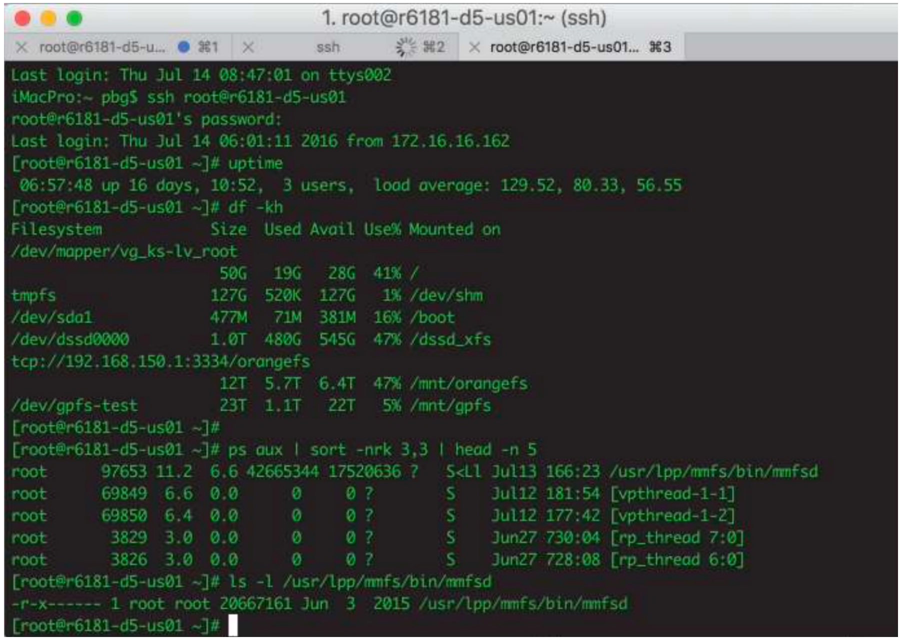


Outline

- 2.1 Operating System Services
- 2.2 User and Operating-System Interface
- 2.3 System Calls
- 2.4 System Services
- 2.7 Operating System Design and Implementation
- 2.8 Operating System Structure
- 2.10 Operating System Debugging

Users and Operating-System Interface

- **Command Line (Command interpreter)** allows direct command entry
  - Reads commands from the user or from a file of commands and executes them, usually by turning them into one or more system calls.



```
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                       50G   19G   28G  41% /
tmpfs                 127G  520K  127G   1% /dev/shm
/dev/sda1             477M   71M  381M  16% /boot
/dev/dssd0000         1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                       12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test         23T  1.1T   22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653 11.2  6.6 42665344 17520636 ?   S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root      69849  6.6  0.0      0     0 ?        S    Jul12 181:54 [vpthread-1-1]
root      69850  6.4  0.0      0     0 ?        S    Jul12 177:42 [vpthread-1-2]
root       3829  3.0  0.0      0     0 ?        S    Jun27 730:04 [rp_thread 7:0]
root       3826  3.0  0.0      0     0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x------ 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```

Users and Operating-System Interface

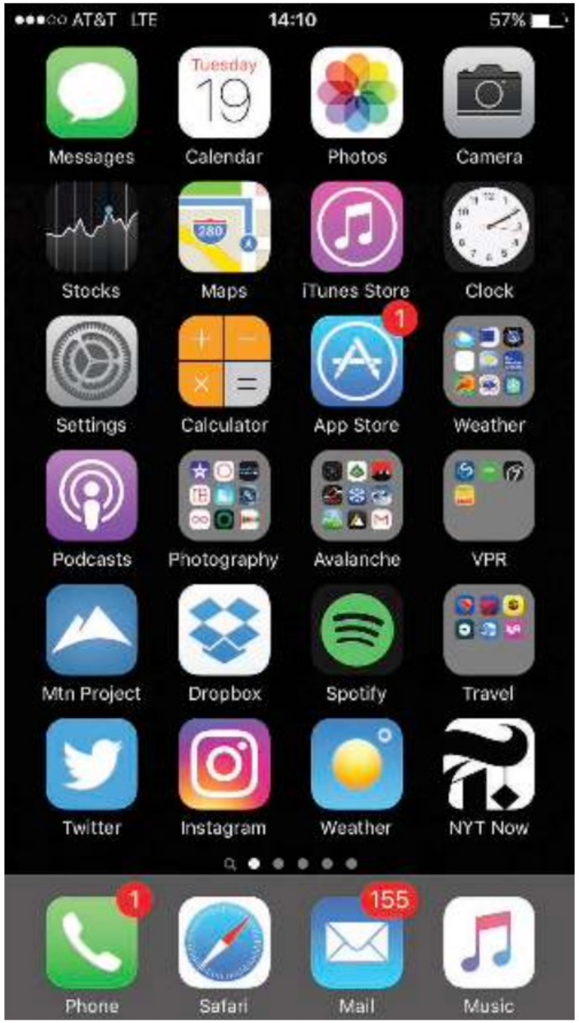
- **Command Line (Command interpreter)** allows direct command entry
  - It is usually not part of the kernel since the command interpreter is subject to changes.
  - A user should be able to develop a new command interpreter using the system-call interface provided by the operating system.
  - The command interpreter allows a user to create and manage processes and also determine ways by which they communicate (such as through pipes and files). In some systems the CI may be incorporated directly into the kernel.

Graphical User Interface (GUI)

- Generally implemented as a desktop metaphor, with file folders, trash cans, and resource icons.
- First developed in the early 1970's at Xerox PARC research facility.
- Mac has traditionally provided ONLY the GUI interface. With the advent of OSX (based partially on UNIX), a command line interface has also become available.

Touch-Screen Interface

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands



Outline

- 2.1 Operating System Services
- 2.2 User and Operating-System Interface
- 2.3 System Calls
- 2.4 System Services
- 2.7 Operating System Design and Implementation
- 2.8 Operating System Structure
- 2.10 Operating System Debugging

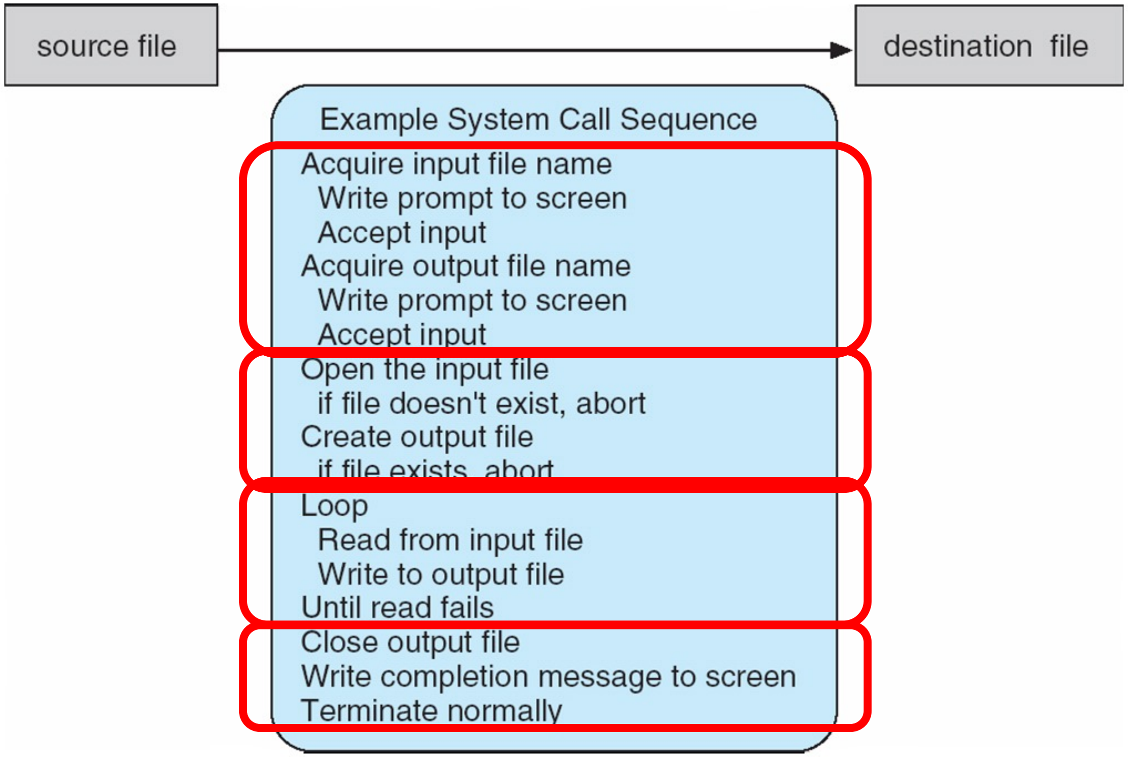
System Calls

- Systems calls provide an 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)

Example of System Calls

- System call sequence to copy the contents of one file to another file

```
cp in.txt out.txt
```



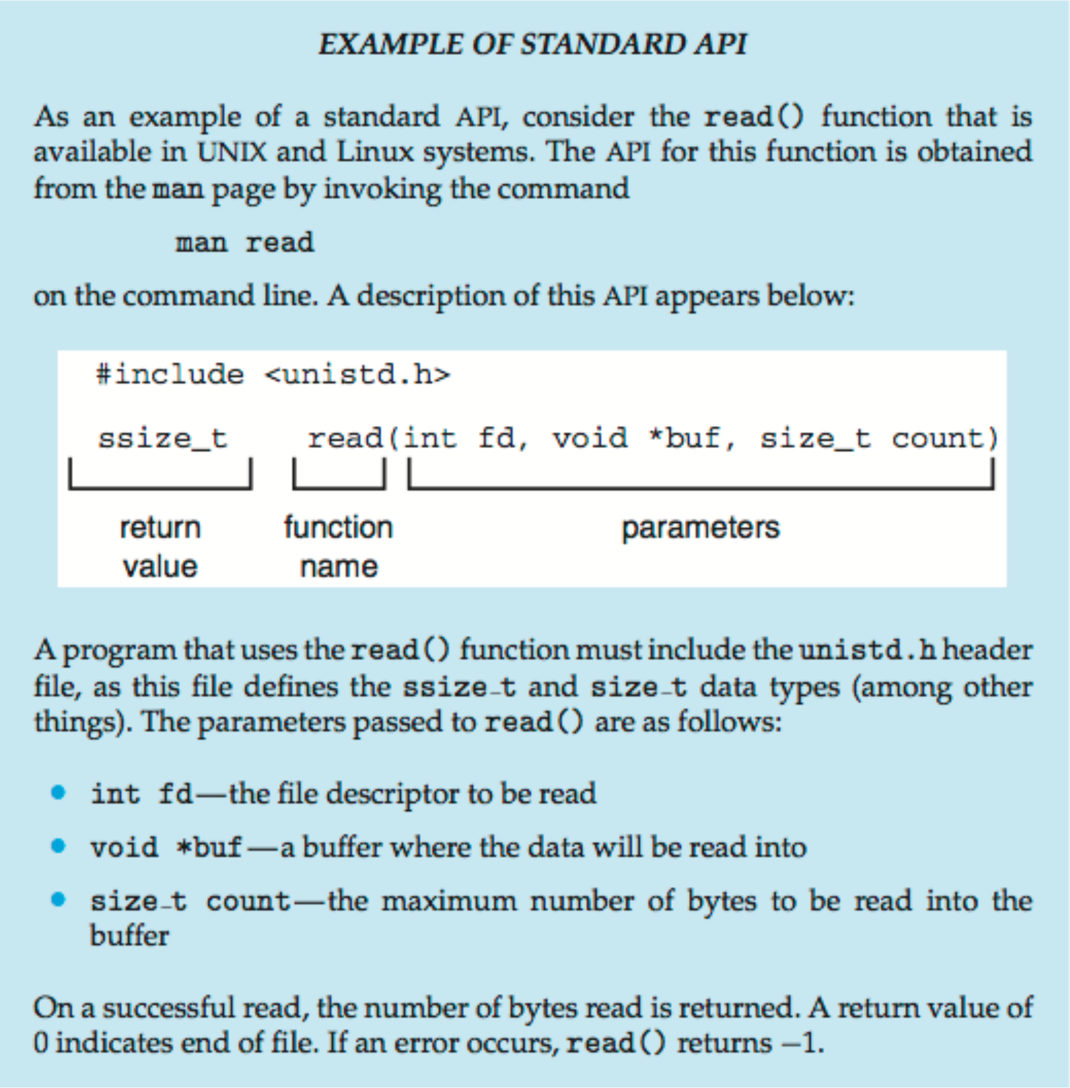
source file

destination file

Example System Call Sequence

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

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

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:

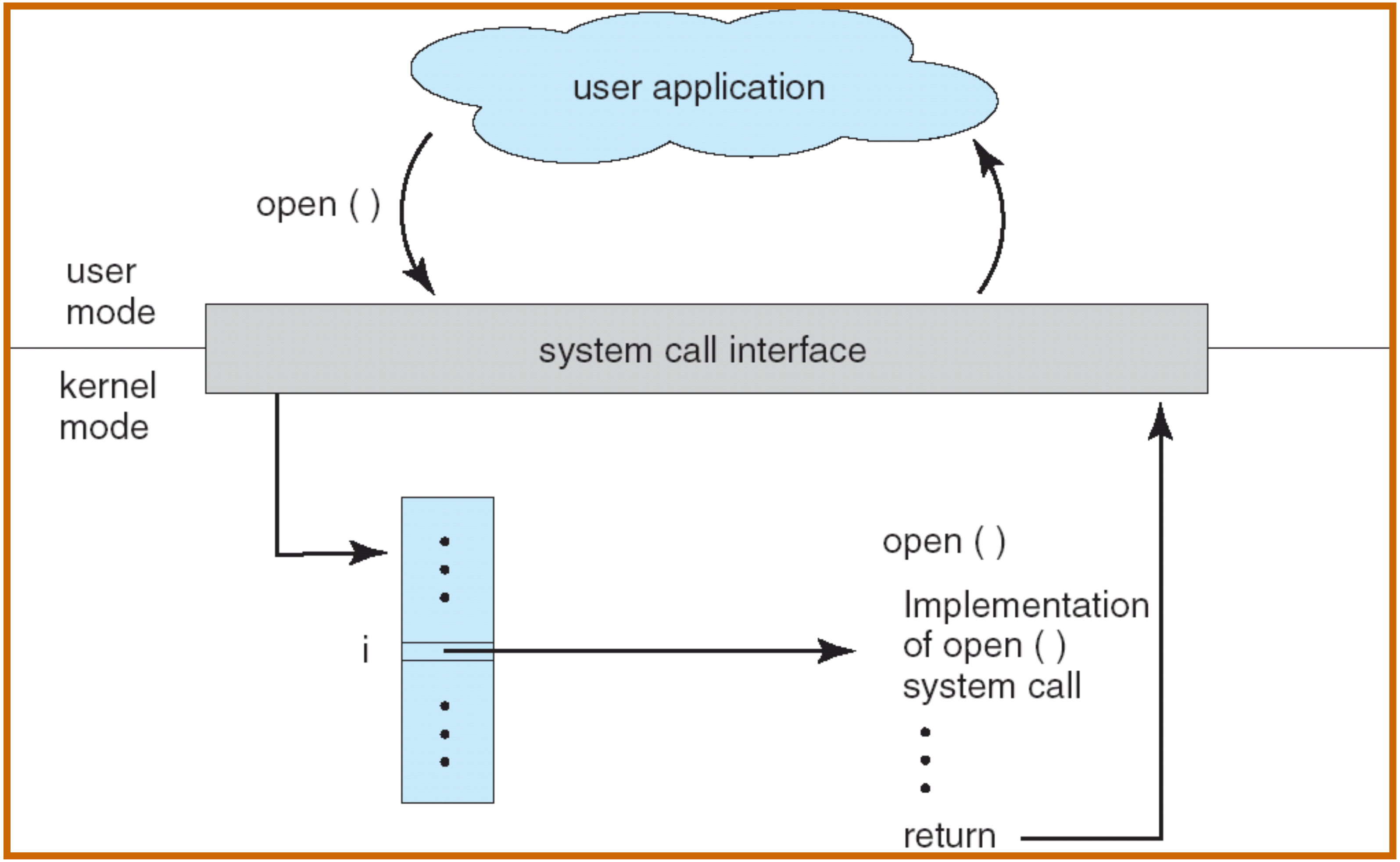
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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

- 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 a compiler)
- The use of APIs instead of direct system calls provides for greater program portability between different systems.
- The API makes the appropriate system calls through the **system call interface**, using a table lookup to access specific numbered system calls

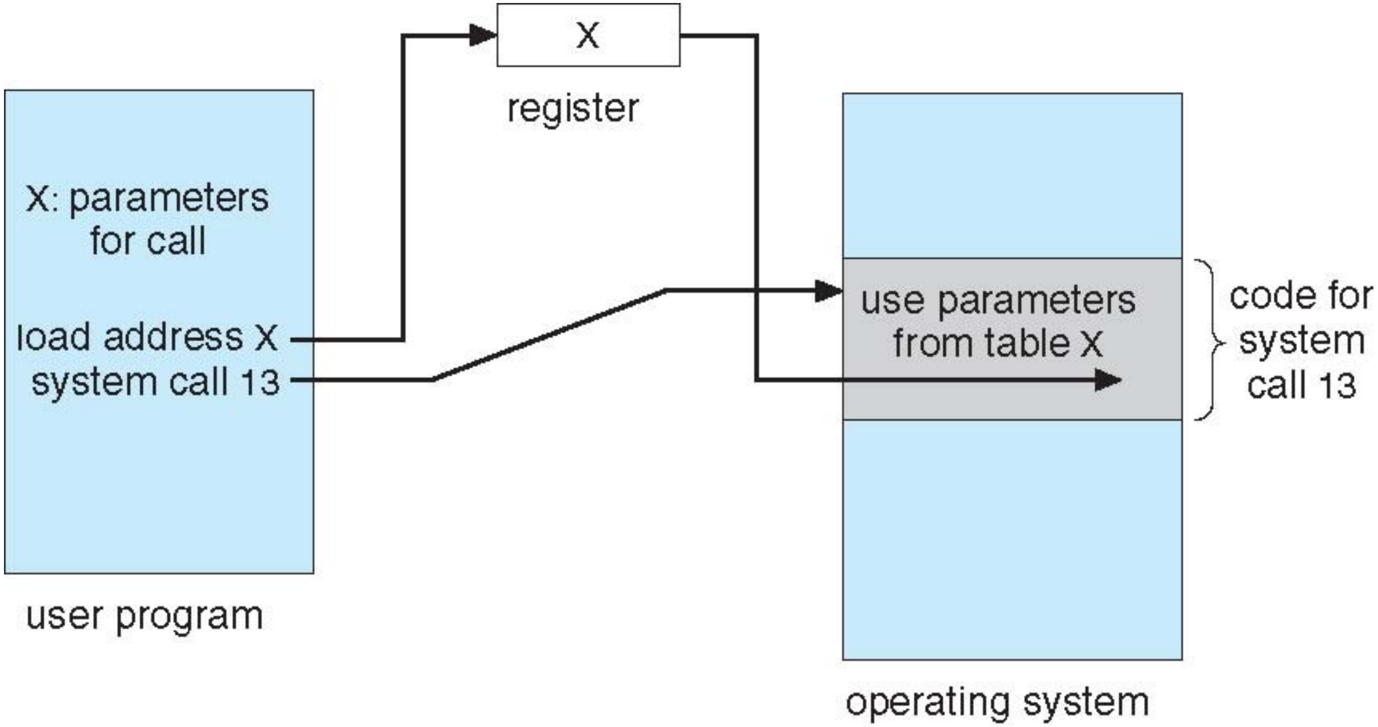
API – System Call Interface – OS Relationship



Passing Parameters to 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

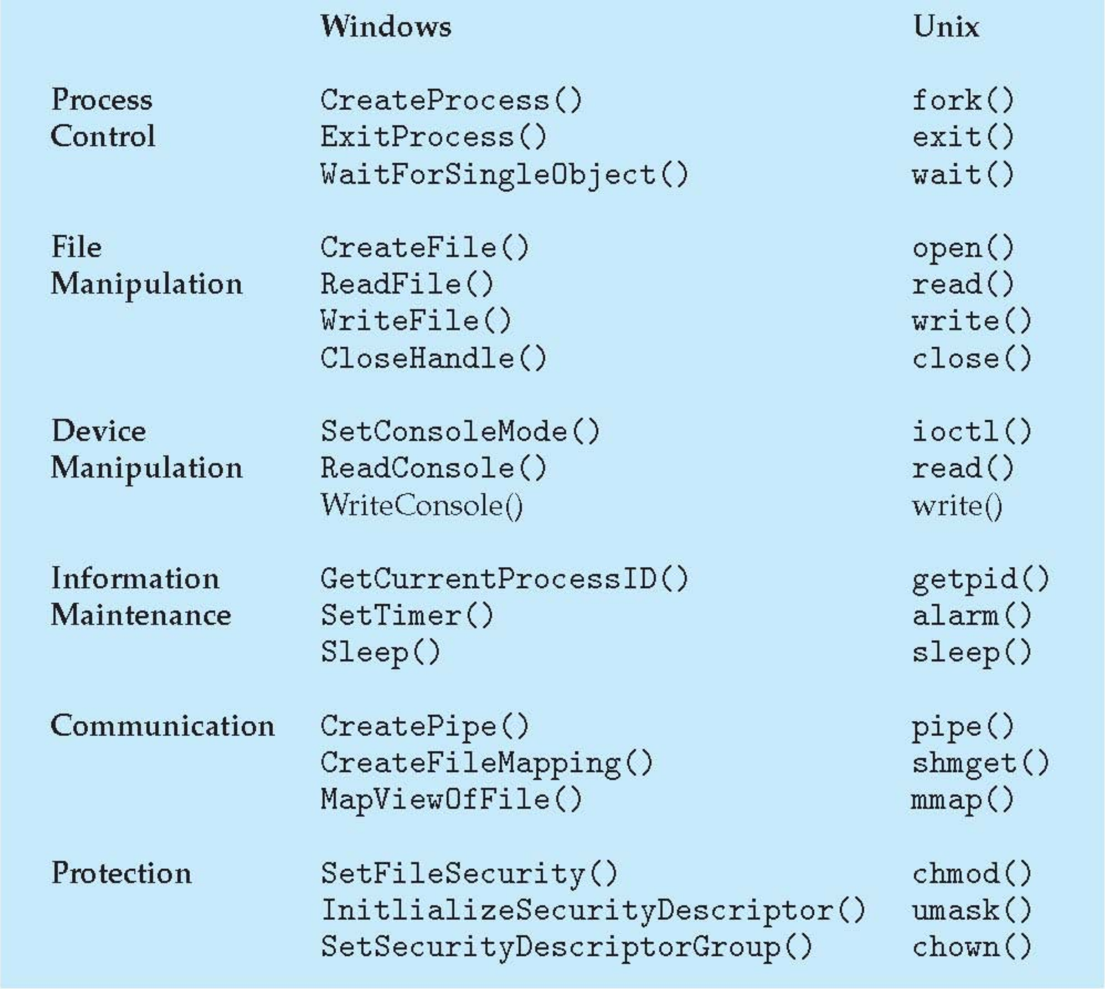
Parameter Passing via Table



Types of System Calls

- Six major categories:
  - Process control
    - Load, execute, end, abort, create process, get/set process attributes, wait for time/signal, allocate/free memory
  - File management (manipulation)
    - Create/delete/open/close/read/write a file, get/set file attributes
  - Device management
    - Request/release device, read/write data, get/set attributes
  - Information maintenance
    - Get/set time or date, get/set system data, get/set attributes for process/file/device
  - Communications
    - Create/delete connection, send/receive messages, attach/detach devices
  - Protection
    - Control access to resources, get and set permissions, allow and deny user access

Examples of Windows and Unix System Calls

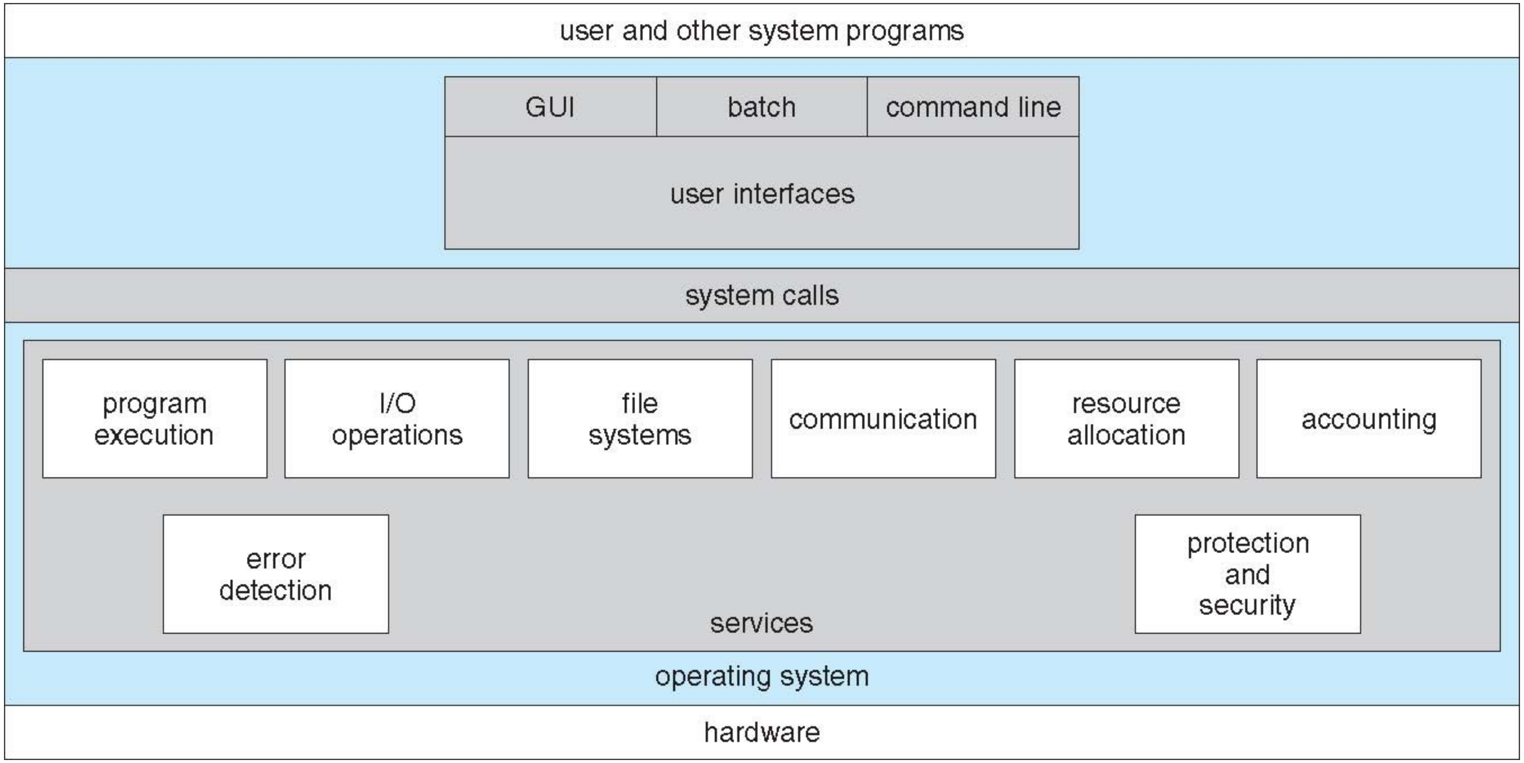


	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Outline

- 2.1 Operating System Services
- 2.2 User and Operating-System Interface
- 2.3 System Calls
- 2.4 System Services
- 2.7 Operating System Design and Implementation
- 2.8 Operating System Structure
- 2.10 Operating System Debugging

Recall: A View of Operating System Services



System Services

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File management (create, delete, copy, rename, print, list, etc.)
  - Status information (date, time, memory, disk space, users, etc.)
  - File modification (text editors, file search)
  - Programming language support (compiler, linkers, interpreters)
  - Program loading and execution (loaders)
  - Communications (virtual links, send/receive messages)
  - Application programs (web browsers, office suites)
- Most users’ view of the operation system is defined by system programs, not the actual system calls

Outline

- 2.1 Operating System Services
- 2.2 User and Operating-System Interface
- 2.3 System Calls
- 2.4 System Services
- 2.7 Operating System Design and Implementation
- 2.8 Operating System Structure
- 2.10 Operating System Debugging

Operating System Design and Implementation

- ➢ Design and Implementation of OS not “solvable”, but some approaches have proven successful
- ➢ Internal structure of different Operating Systems can vary widely
- ➢ Affected by choice of hardware, type of 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

- Important principle to separate

  Policy: ***What*** will be done?

  Mechanism: ***How*** to do it?

- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)

- Specifying and designing an OS is highly creative task of **software engineering**

Outline

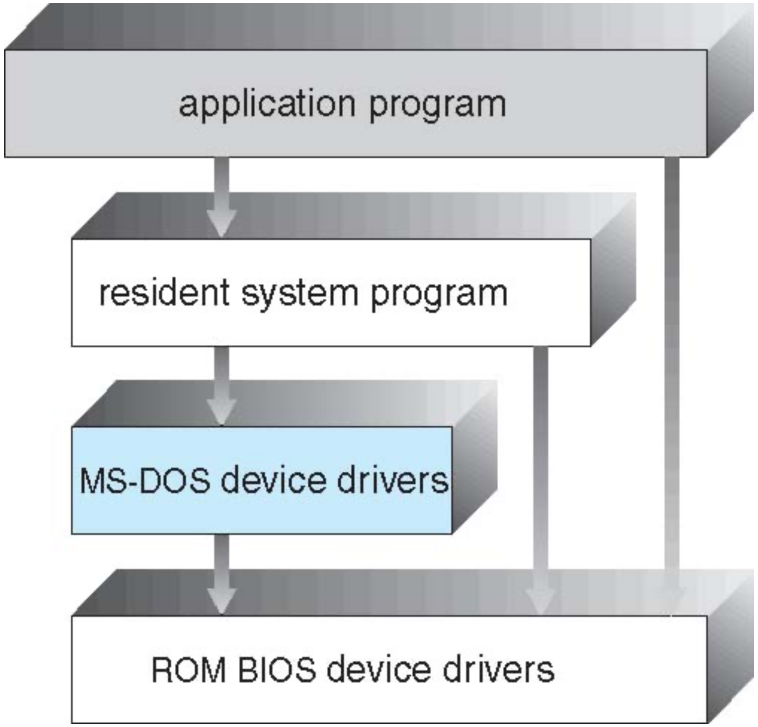
- 2.1 Operating System Services
- 2.2 User and Operating-System Interface
- 2.3 System Calls
- 2.4 System Services
- 2.7 Operating System Design and Implementation
- 2.8 Operating System Structure
- 2.10 Operating System Debugging

Operating System Structure

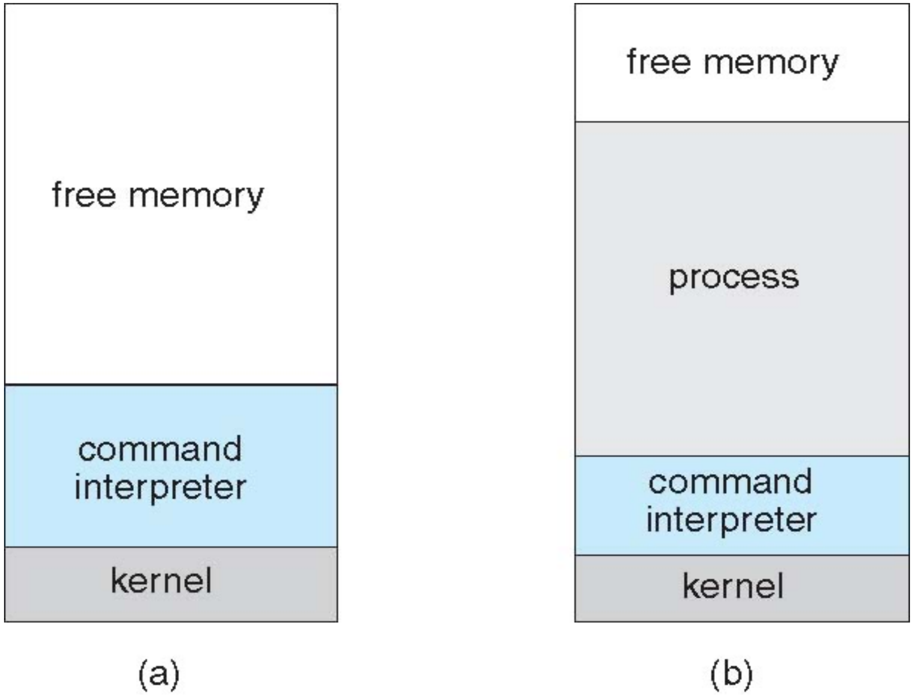
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex -- UNIX
  - Layered – an abstraction
  - Microkernel -Mach

Simple Structure -- MS-DOS

- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
  - Single-tasking systems
  - Shell invoked when system booted
  - Simple method to run program
    - No process created
  - Single memory space
  - Loads program into memory, overwriting all but the kernel
  - Program exit -> shell reloaded



Simple Structure -- MS-DOS



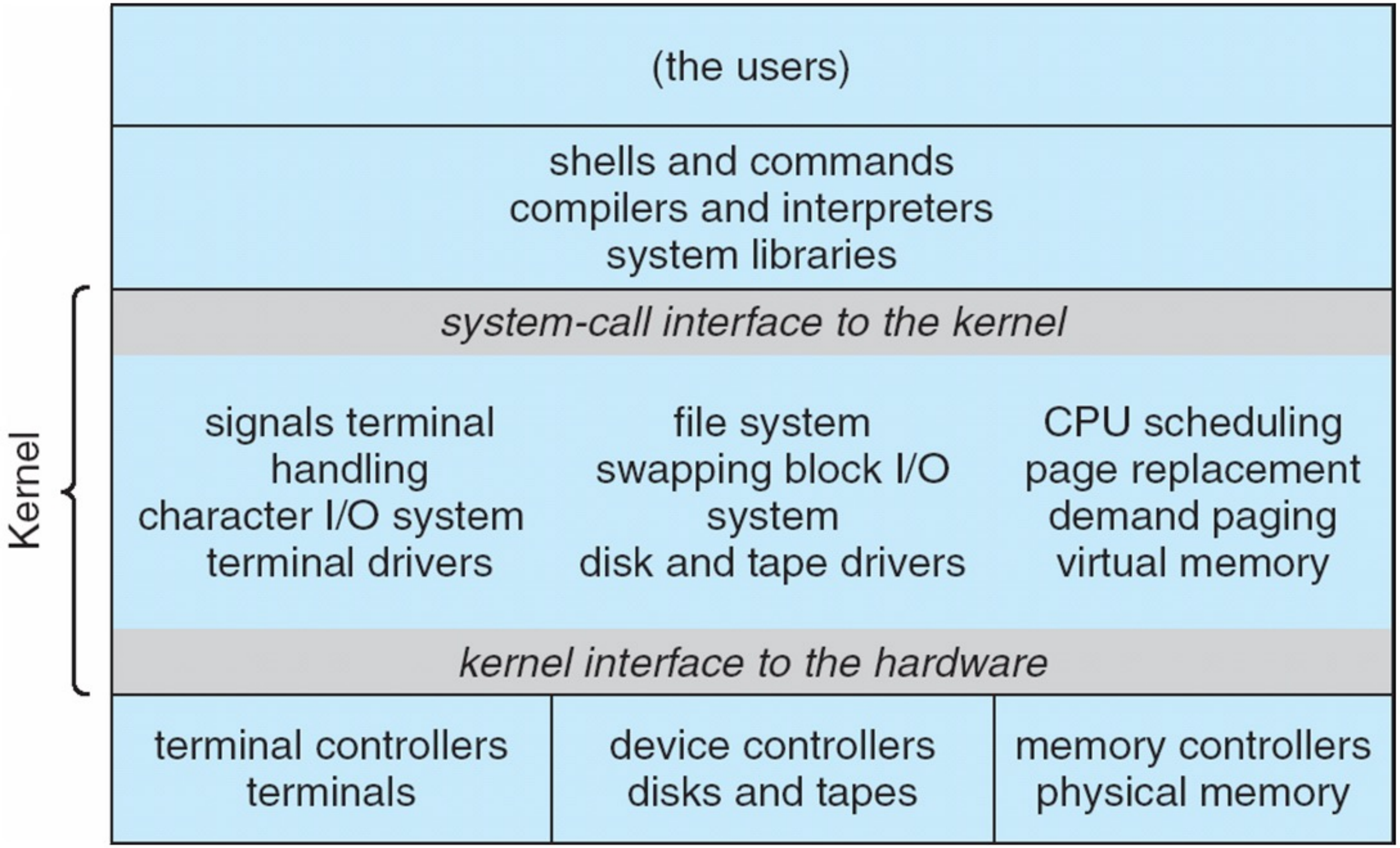
(a) At system startup (b) running a program

UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

Traditional UNIX System Structure

Beyond simple but not fully layered



(the users)

shells and commands
compilers and interpreters
system libraries

system-call interface to the kernel

signals terminal handling
character I/O system
terminal drivers

file system
swapping block I/O system
disk and tape drivers

CPU scheduling
page replacement
demand paging
virtual memory

kernel interface to the hardware

terminal controllers
terminals

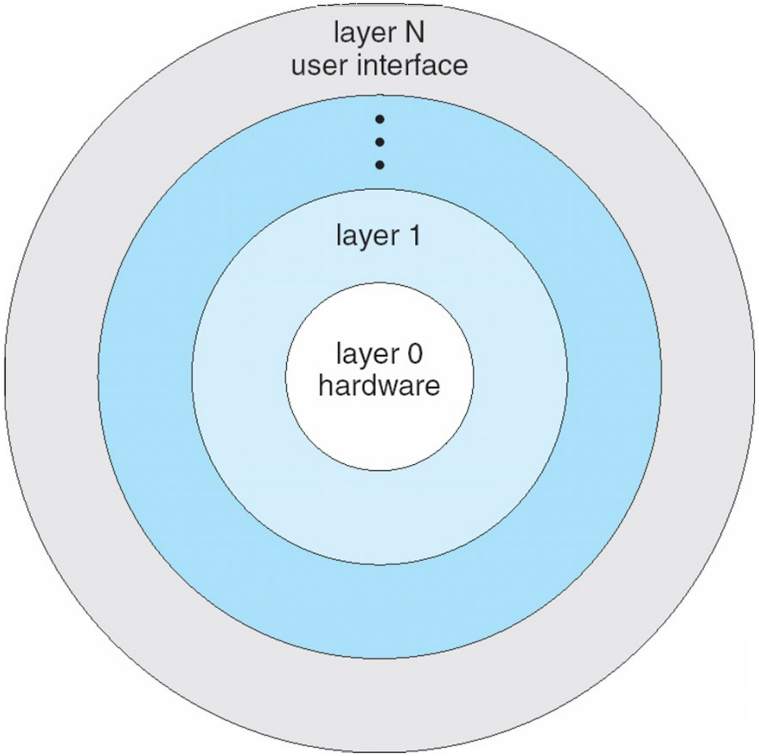
device controllers
disks and tapes

memory controllers
physical memory

Kernel

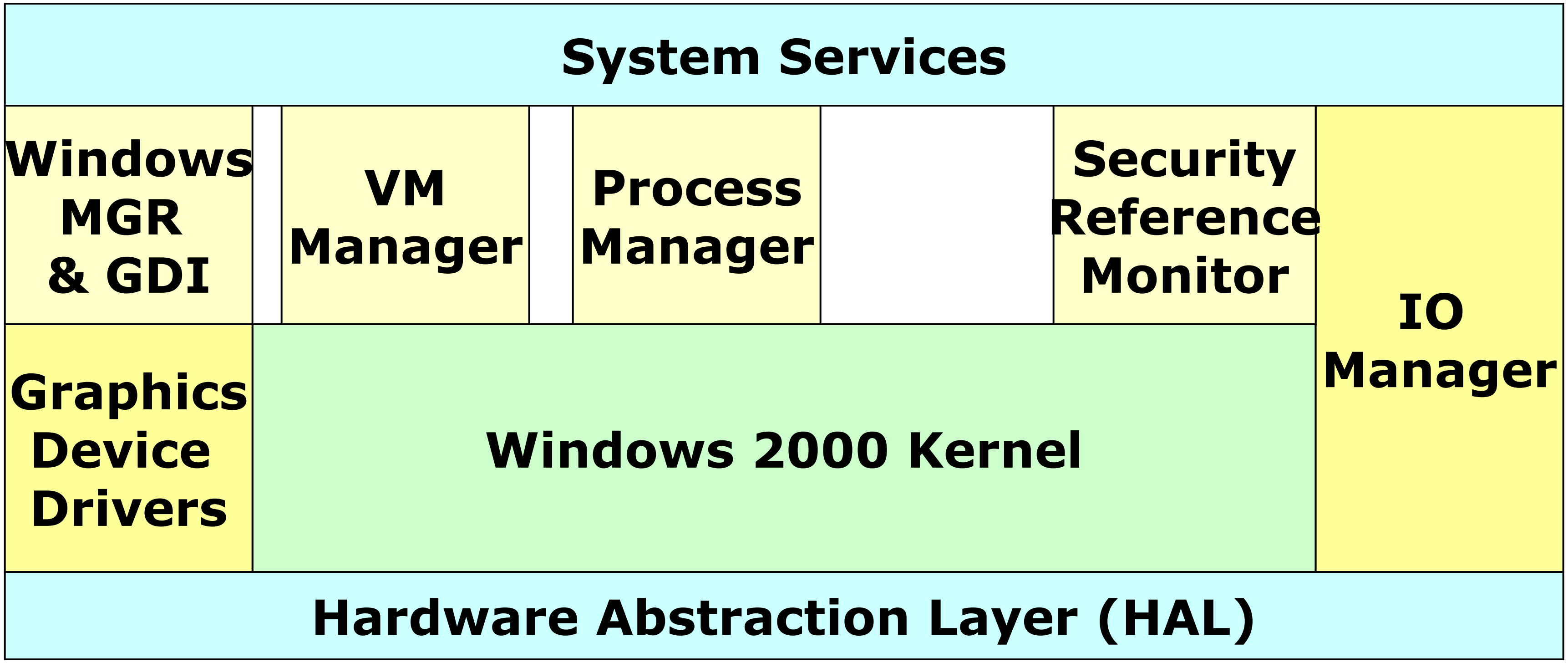
Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



Layered Approach (Cont’d)

- Example of Windows 2000.



System Services

Windows MGR & GDI

VM Manager

Process Manager

Security Reference Monitor

IO Manager

Graphics Device Drivers

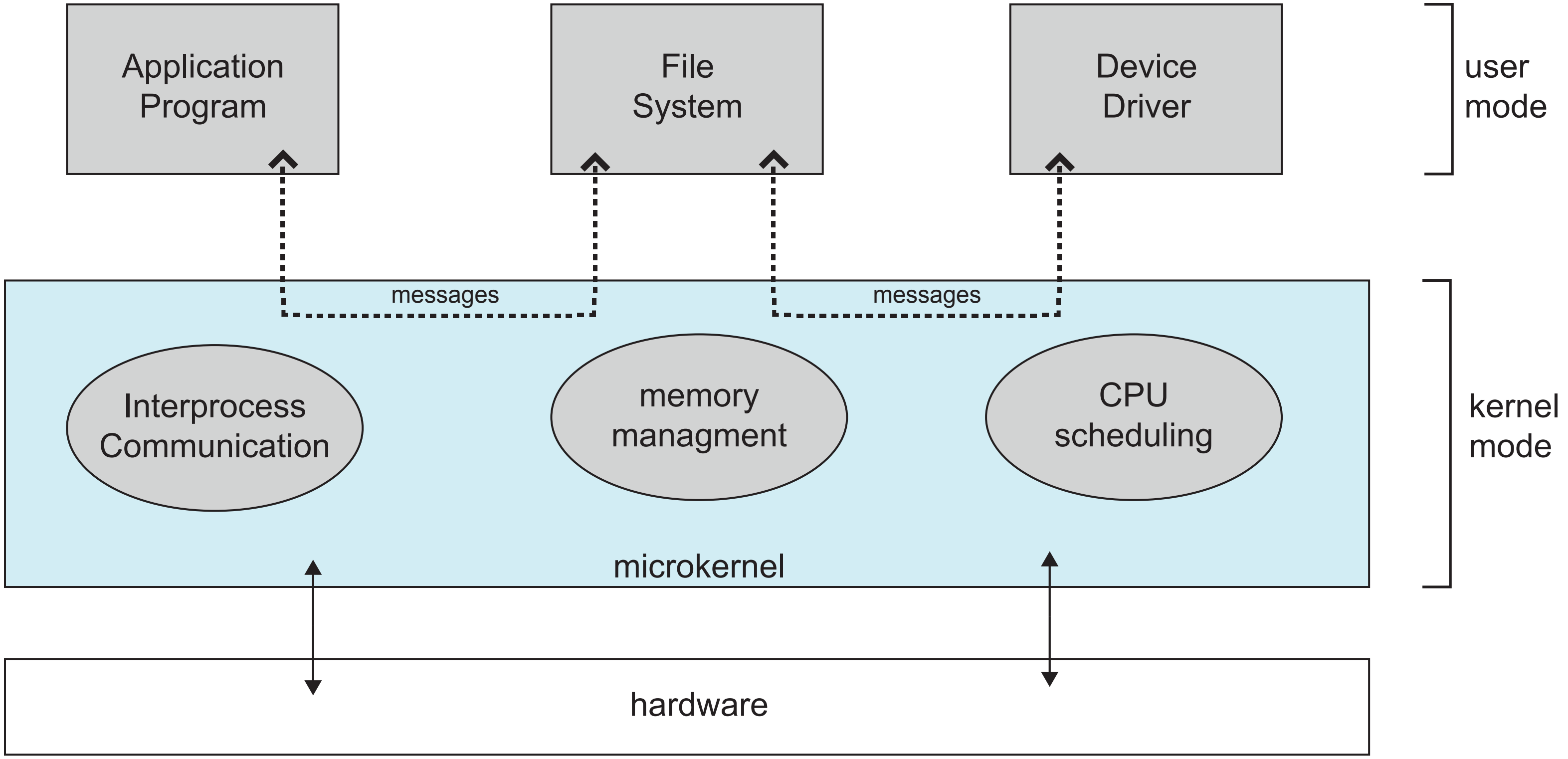
Windows 2000 Kernel

Hardware Abstraction Layer (HAL)

Microkernel System Structure

- Moves as much from the kernel into user space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

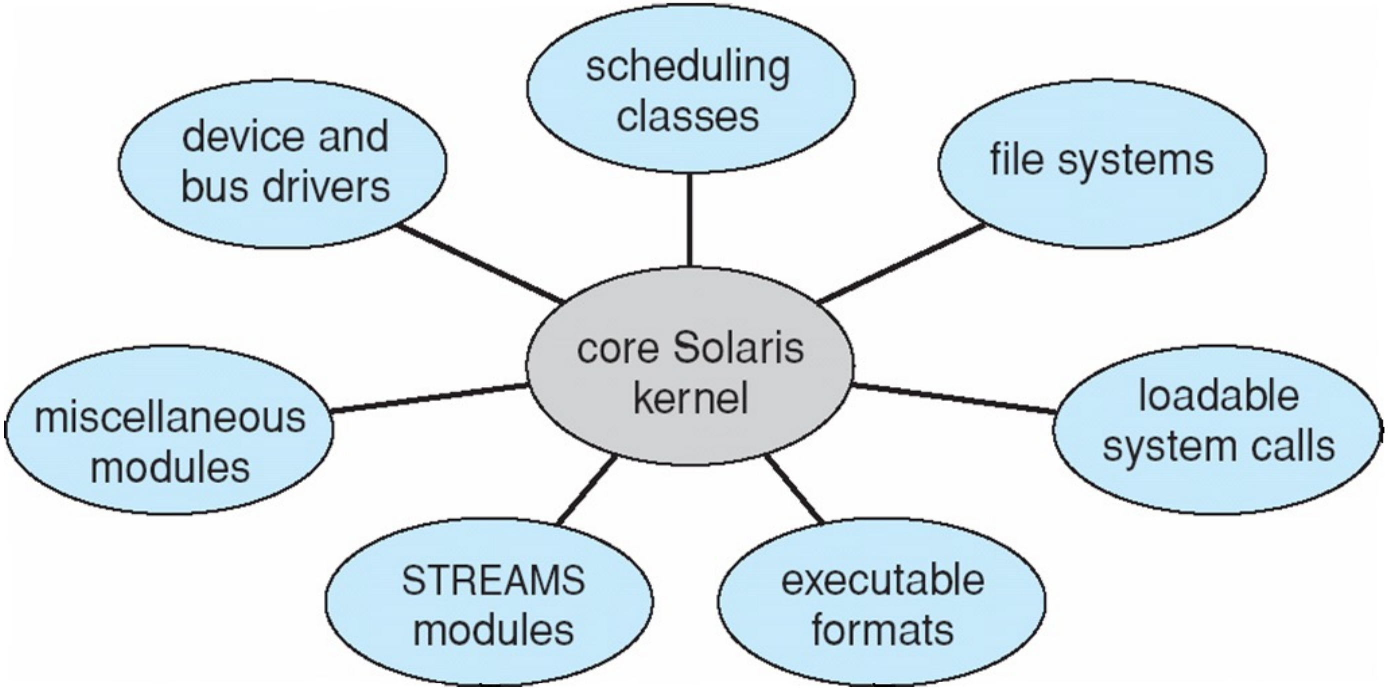
Microkernel System Structure (Cont'd)



Modules

- Many modern operating systems implement **loadable kernel modules**
  - 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
- Overall, similar to layers but with more flexibility
  - Linux, Solaris, etc

Solaris Modular Approach



Hybrid Systems

- Most modern operating systems are 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***

Quiz

- In what ways is the modular approach similar to the layered approach? In what ways does it differ from the layered approach?

Quiz answer

- It is similar to layered system in that each kernel section has defined, protected interfaces, but it is more flexible than a layered system because any module can call any other module

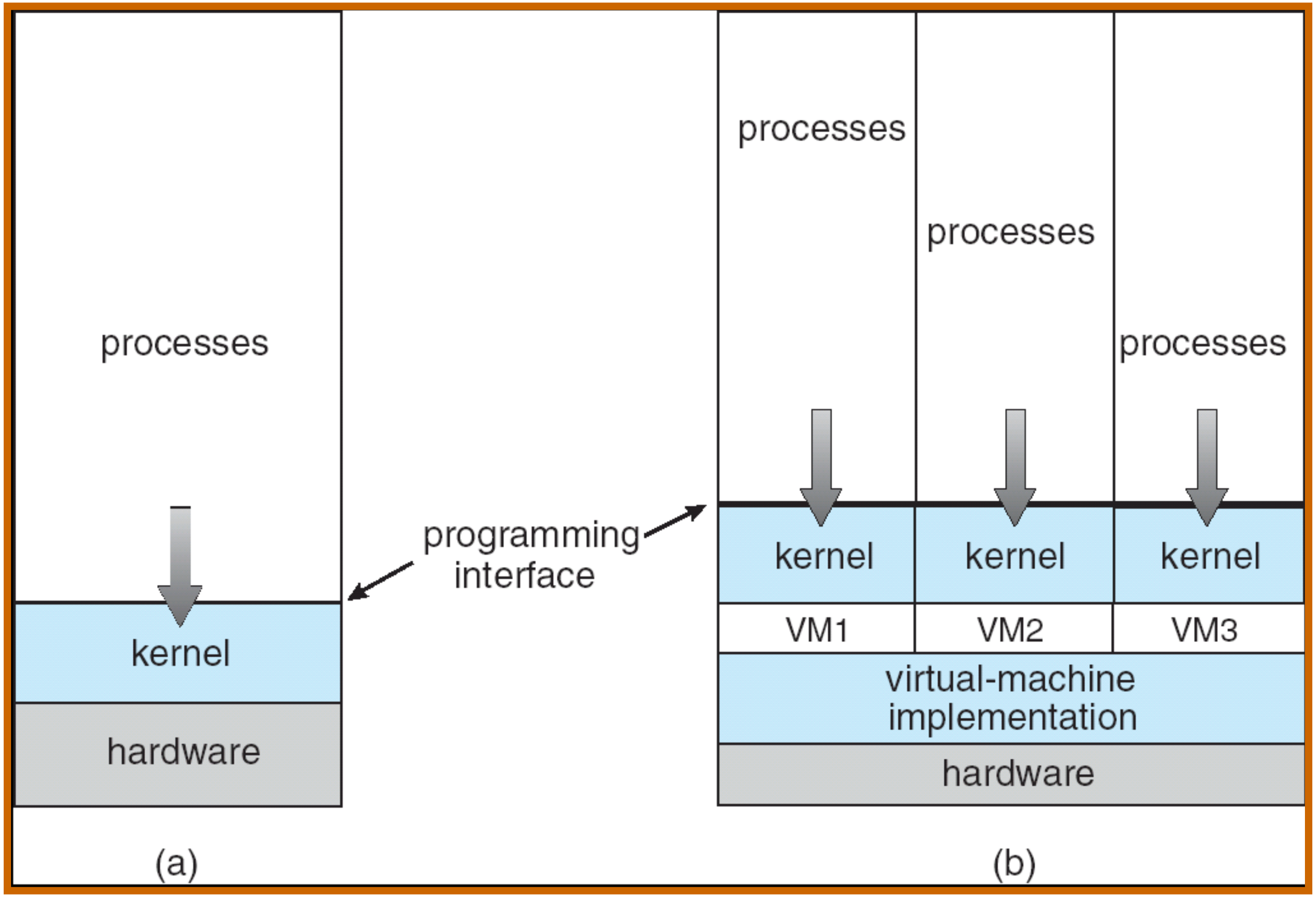
Virtual Machines

- A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
- A virtual machine provides an interface *identical* to the underlying bare hardware
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory
- Each **guest** provided with a (virtual) copy of underlying computer

Virtual Machines

- First appeared commercially in IBM mainframes in 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Protect from each other
- Some sharing of file can be permitted, controlled
- Commutate with each other, other physical systems via networking
- Useful for development, testing, research
- Is easy to debug, and security problems are easy to solve.
- **Consolidation** taking two or more separate systems and running them in virtual machines on one system

Virtual Machines



Virtual Machines

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine
- Hardware support needed
  - More support-> better virtualization

Quiz

- What is the main advantage for an operating-system designer of using a virtual-machine architecture? What is the main advantage for a user?

Quiz-Answer

- What is the main advantage for an operating-system designer of using a virtual-machine architecture? What is the main advantage for a user?

Answer: The system is easy to debug, and security problems are easy to solve. Virtual machines also provide a good platform for operating system research since many different operating systems may run on one physical system.

Outline

- 2.1 Operating System Services
- 2.2 User and Operating-System Interface
- 2.3 System Calls
- 2.4 System Services
- 2.7 Operating System Design and Implementation
- 2.8 Operating System Structure
- 2.10 Operating System Debugging

Operating-System 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 the process
- Operating system 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."

Questions

Q1. What are the user and system goals in operating system design?

Q2. What are two sets of the services and functions provided by an operating system? Briefly describe those two sets.

Q3. What are six major activities of an operating system with regard to process management?

Q4. What are six major categories of system calls?

Questions (Cont’d)

Q5. What are three different forms of a user interface that almost all an operating system have?

Q6. What is the fundamental idea behind a virtual machine?

Q7. What are the three major activities of an operating system in regard to memory management?

Q8. Why is the separation of mechanism and policy a desirable property?