Operating System Structures

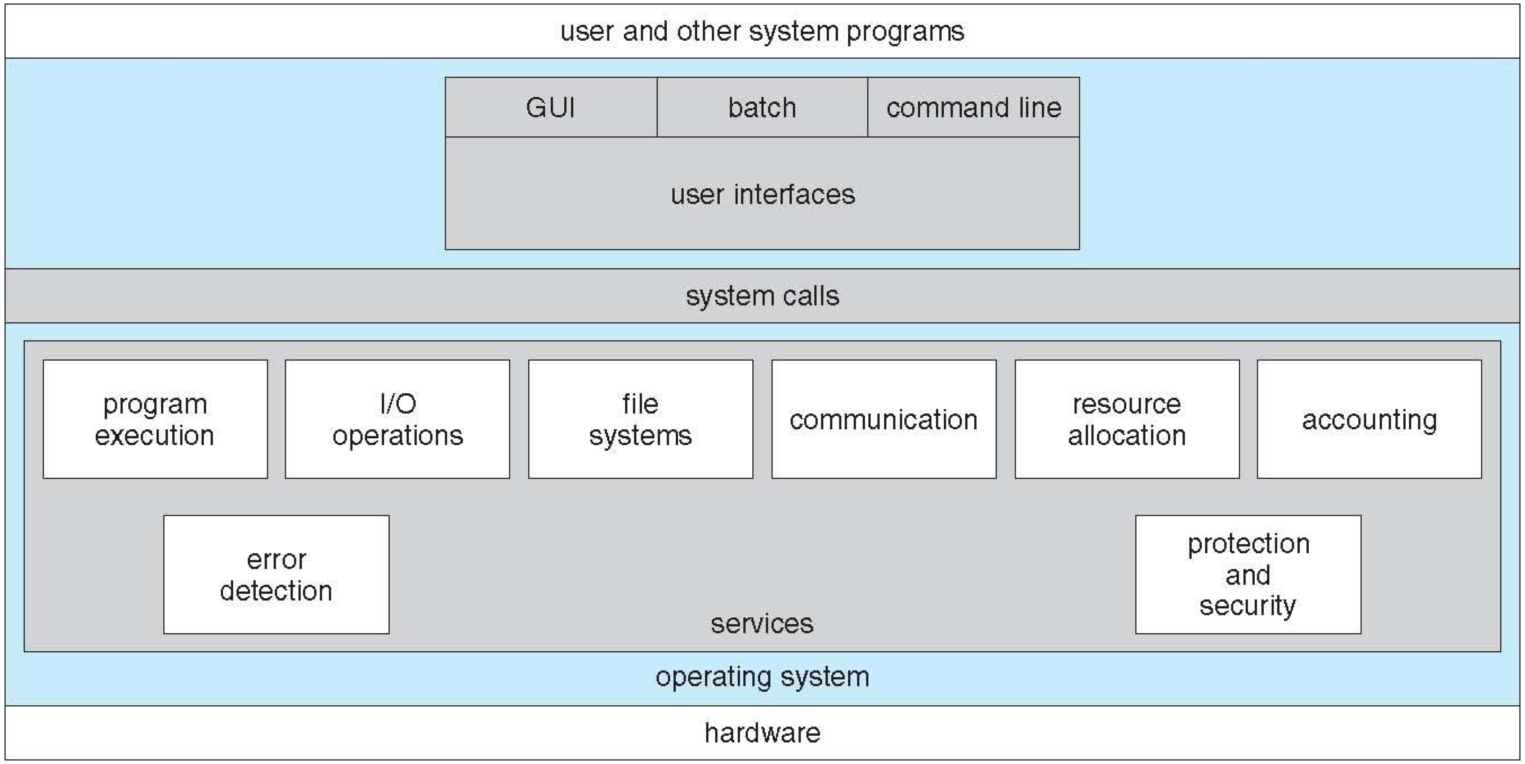
OS services. Access interface to OS services. Shell. System calls and OS API. System libraries and programs. OS design and implementation. Policy vs. mechanism. Monolithic, layered, microkernel design.

OS2: 30/1/2018
Textbook (SGG): Ch. 2.1-2.6,2.7.1-2.7.3,2.9.2

Operating System Services

- One set of OS services is to help the *user* use the computer:
  - User interface - Almost all operating systems have a user interface (UI)
    - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch**
  - Program execution – load a program into memory; run that program; end execution, either normally or abnormally (indicating error)
  - I/O operations - A running program may require I/O, which may involve a file or an I/O device
  - File-system manipulation - File system is of particular interest (usually stores almost all the data we care about). Programs need to read/write files and directories, create/delete them, search them, list file Information, manage permission.

A View of OS Services



OS Services (cont'd)

- Set of OS services that help the user (cont'd):
  - 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)
  - Error detection – OS needs to be constantly aware of possible errors
    - May occur in CPU, memory hardware, I/O devices, or user program
    - For each type of error, OS should take appropriate action to isolate or recover from it
    - Debugging facilities can greatly enhance users and programmer's abilities to use the system correctly

Operating System Services (cont'd)

- Another set of OS services is for efficient operation of the *system* via resource sharing
  - **Resource allocation –** When multiple users/jobs run concurrently, how to allocate resources among them?
    - Many types of resources; general goals: efficiency, fairness
  - **Accounting –** Keep track of who use how much and what kinds of resources (even if you don't charge money for the usage)
  - **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 (logically) with each other
    - **Protection** ensures accesses to resources or information are legal and valid
    - **Security** protects against active attackers (usually from outside the system, but not always) who deliberately plan and do harm

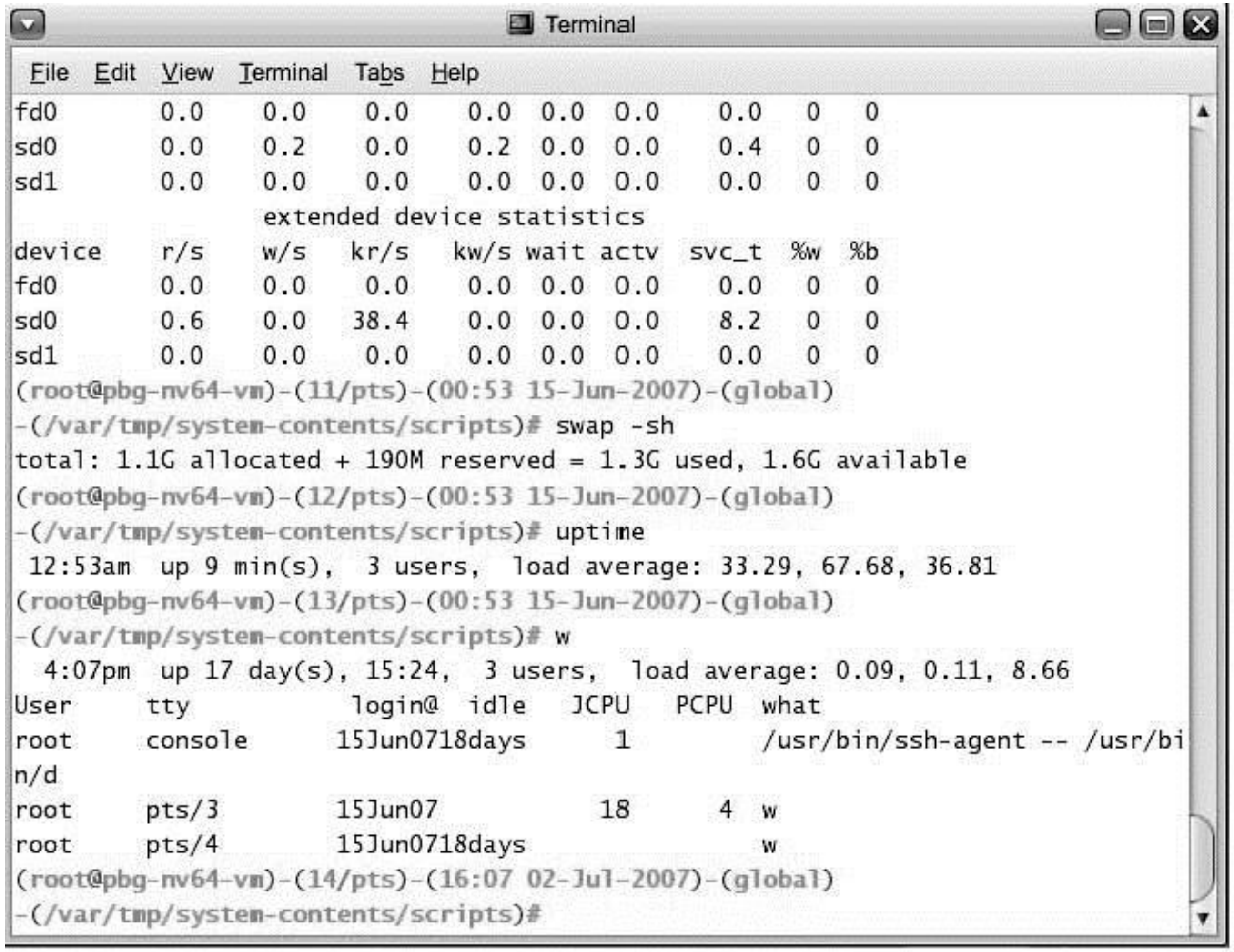
Access interface to OS services - CLI

- Command Line Interface (CLI) or **command interpreter** allows direct command entry.
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented – **shells**
    - You implemented a simple shell in Lab 1
  - Primarily fetches a command from user and executes it
    - Sometimes commands built-in, sometimes just names of programs
      - If the latter, adding new features doesn’t require shell modification
    - In Lab 1, you created a separate process for executing each command, e.g., system(2) (C) or ProcessBuilder (Java)

Access interface to OS services - GUI

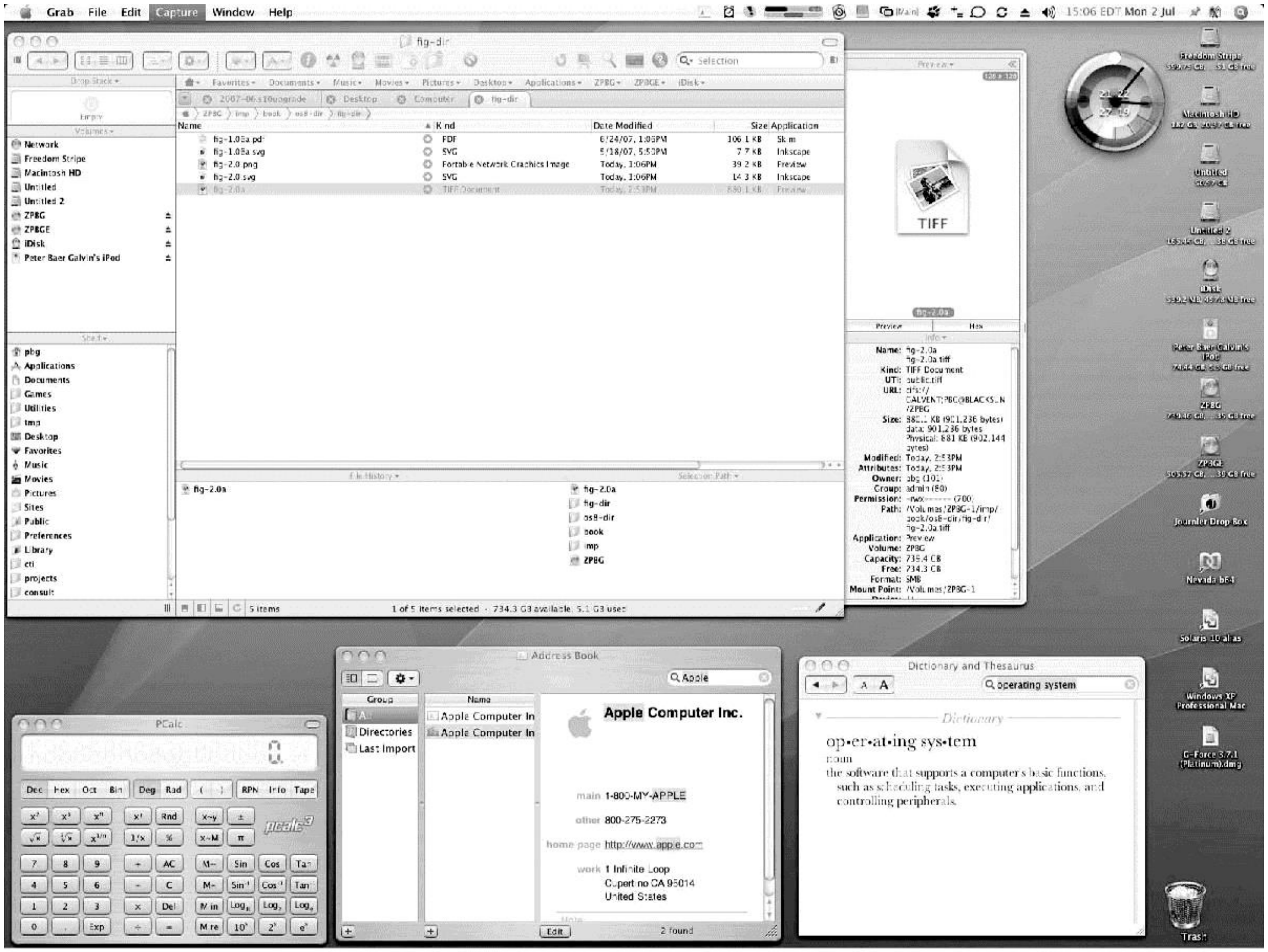
- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions - get information, set options, execute function, open directory (known as **folder**), open file, etc
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS/X has "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

Bourne Shell Command Interpreter



```
fd0        0.0    0.0    0.0    0.0  0.0  0.0    0.0   0    0
sd0        0.0    0.2    0.0    0.2  0.0  0.0    0.4   0    0
sd1        0.0    0.0    0.0    0.0  0.0  0.0    0.0   0    0
                  extended device statistics
device     r/s    w/s   kr/s   kw/s wait actv  svc_t  %w   %b
fd0        0.0    0.0    0.0    0.0  0.0  0.0    0.0   0    0
sd0        0.6    0.0   38.4    0.0  0.0  0.0    8.2   0    0
sd1        0.0    0.0    0.0    0.0  0.0  0.0    0.0   0    0
(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# uptime
 12:53am  up 9 min(s),  3 users,  load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# w
  4:07pm  up 17 day(s), 15:24,  3 users,  load average: 0.09, 0.11, 8.66
User     tty           login@  idle   JCPU   PCPU  what
root     console      15Jun0718days      1         /usr/bin/ssh-agent -- /usr/bi
n/d
root     pts/3        15Jun07           18      4  w
root     pts/4        15Jun0718days                w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
-(/var/tmp/system-contents/scripts)#
```

The Mac OS X GUI



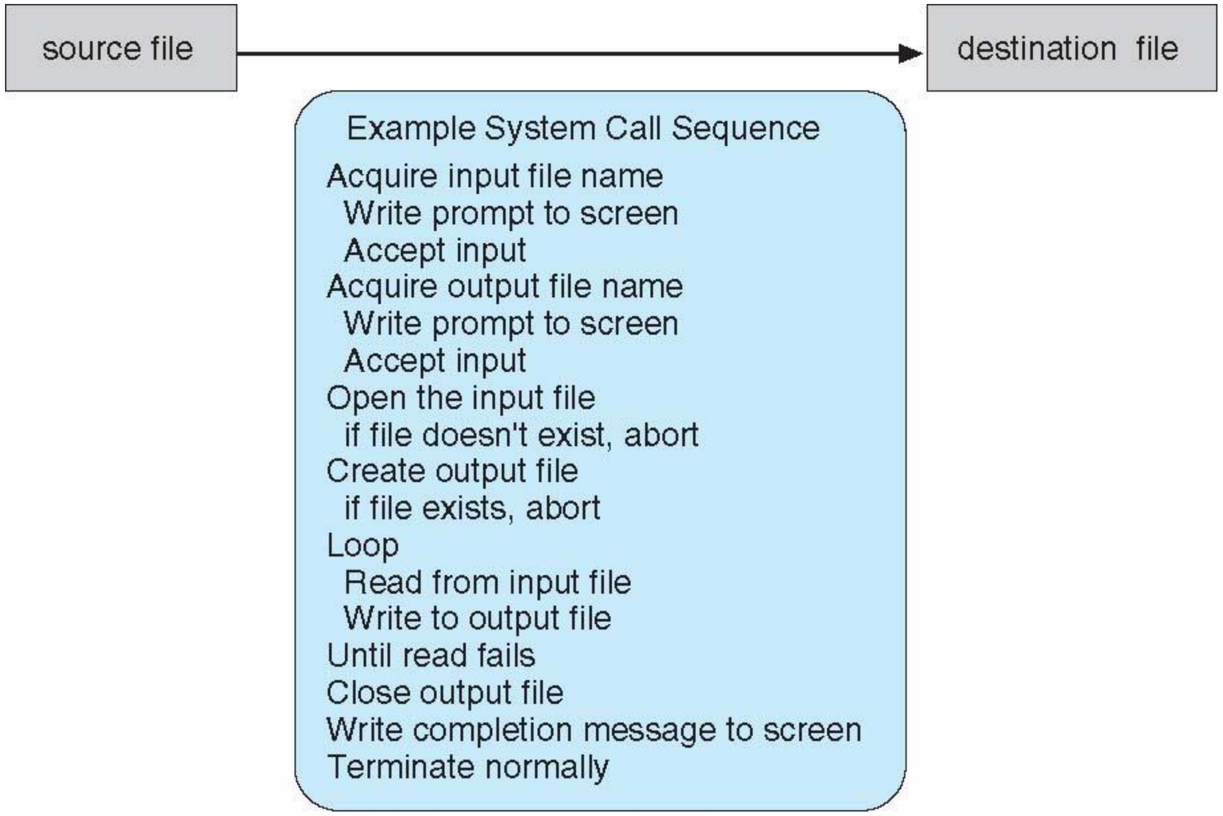
System Calls

- Programming interface to the services provided by the OS kernel
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **application program interface (API)**, but sometimes system calls are made directly
- 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 the Java virtual machine (JVM)
  - E.g., if you use C, you will be using POSIX pthread library in Lab 2; Java classes may also encapsulate system calls within (e.g., ProcessBuilder)
- Why use APIs rather than system calls?

  (Note: unless otherwise stated, names of system calls used throughout this text are generic)

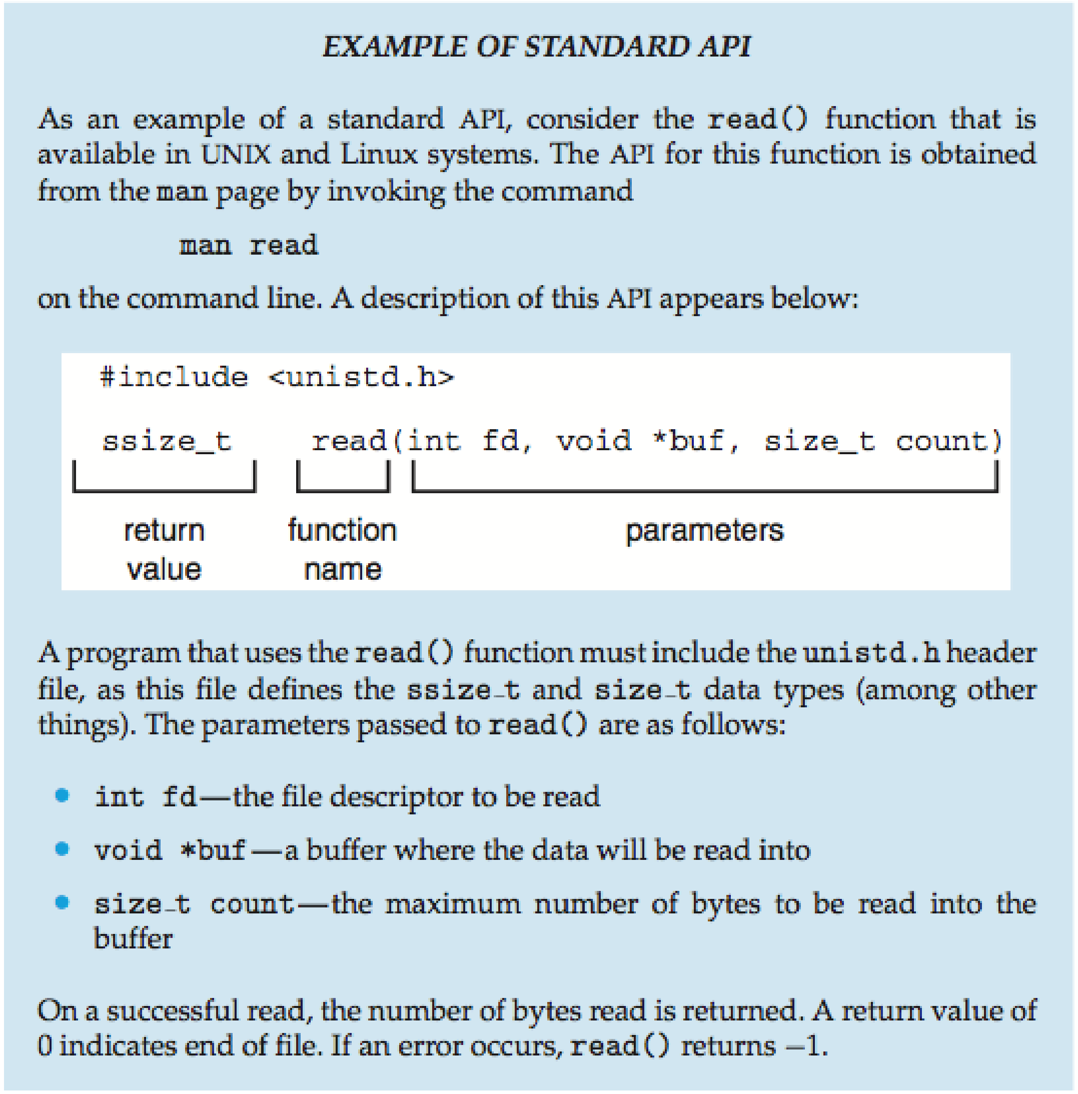
Example of System Calls

- System call sequence to copy the contents of one file to another 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 system call: read() in C



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>
```

$$\underbrace{\texttt{ssize\_t}}_{\text{return value}}\quad\underbrace{\texttt{read}}_{\text{function name}}\underbrace{\texttt{(int fd, void *buf, size\_t count)}}_{\text{parameters}}$$

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:

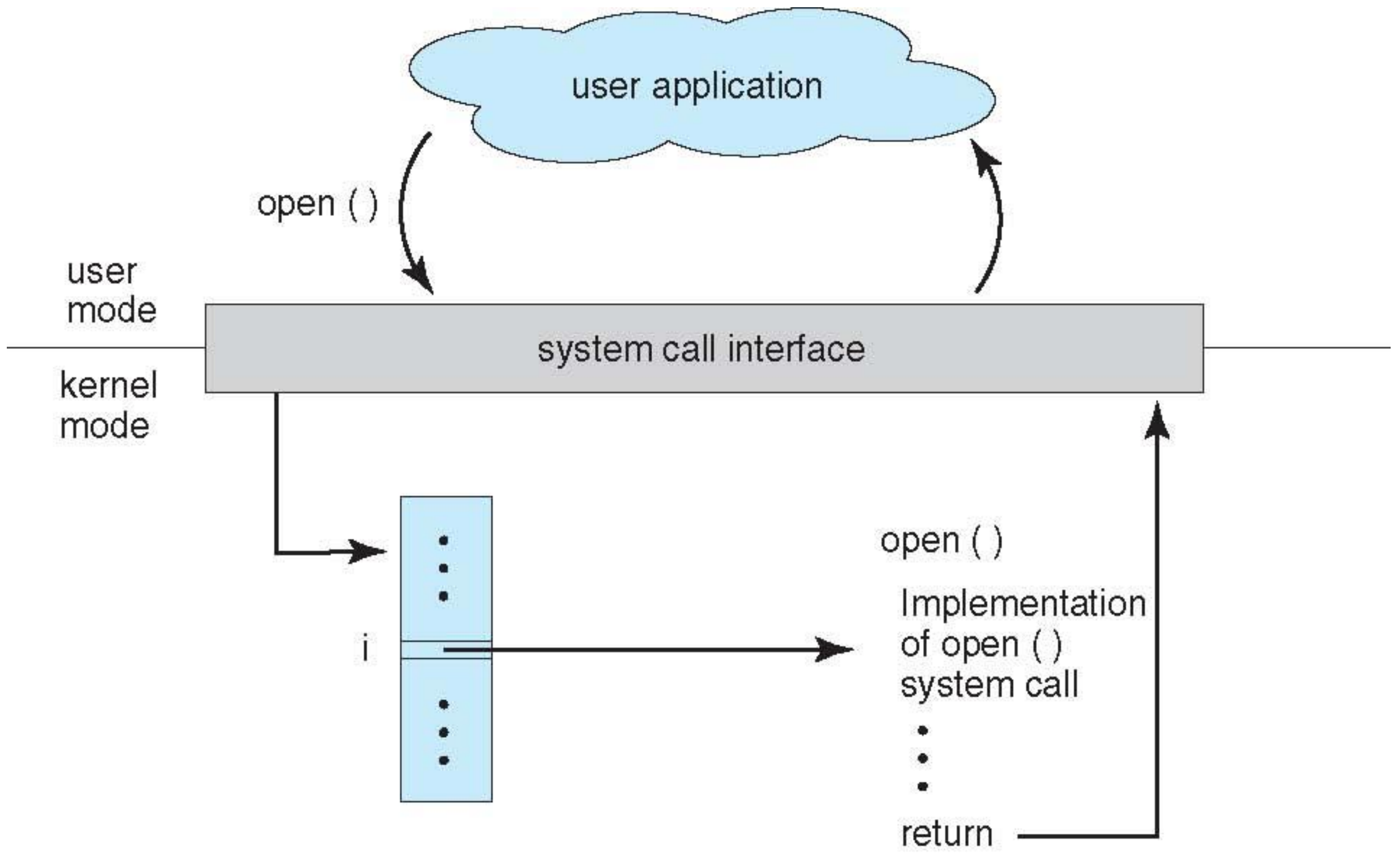
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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

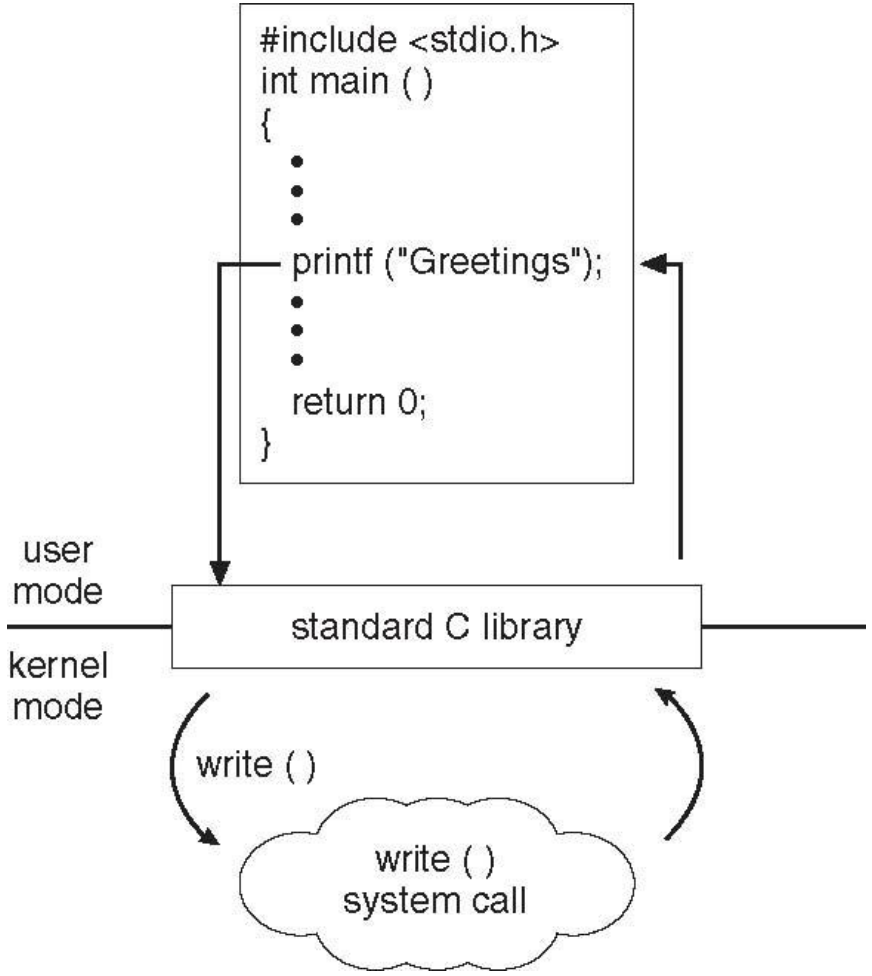
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
  - *Recall*: it's like interrupt, but it's a *software* interrupt (*trap* instruction)
- System call interface (e.g., documented as Unix/Linux manpages) invokes intended system call in OS kernel and returns status of the system call and any return values
- Caller needs know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call (usage is just like a function or library 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), e.g., libc (C) or JCL (Java)

API – System Call – OS Relationship



Standard C Library Example

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

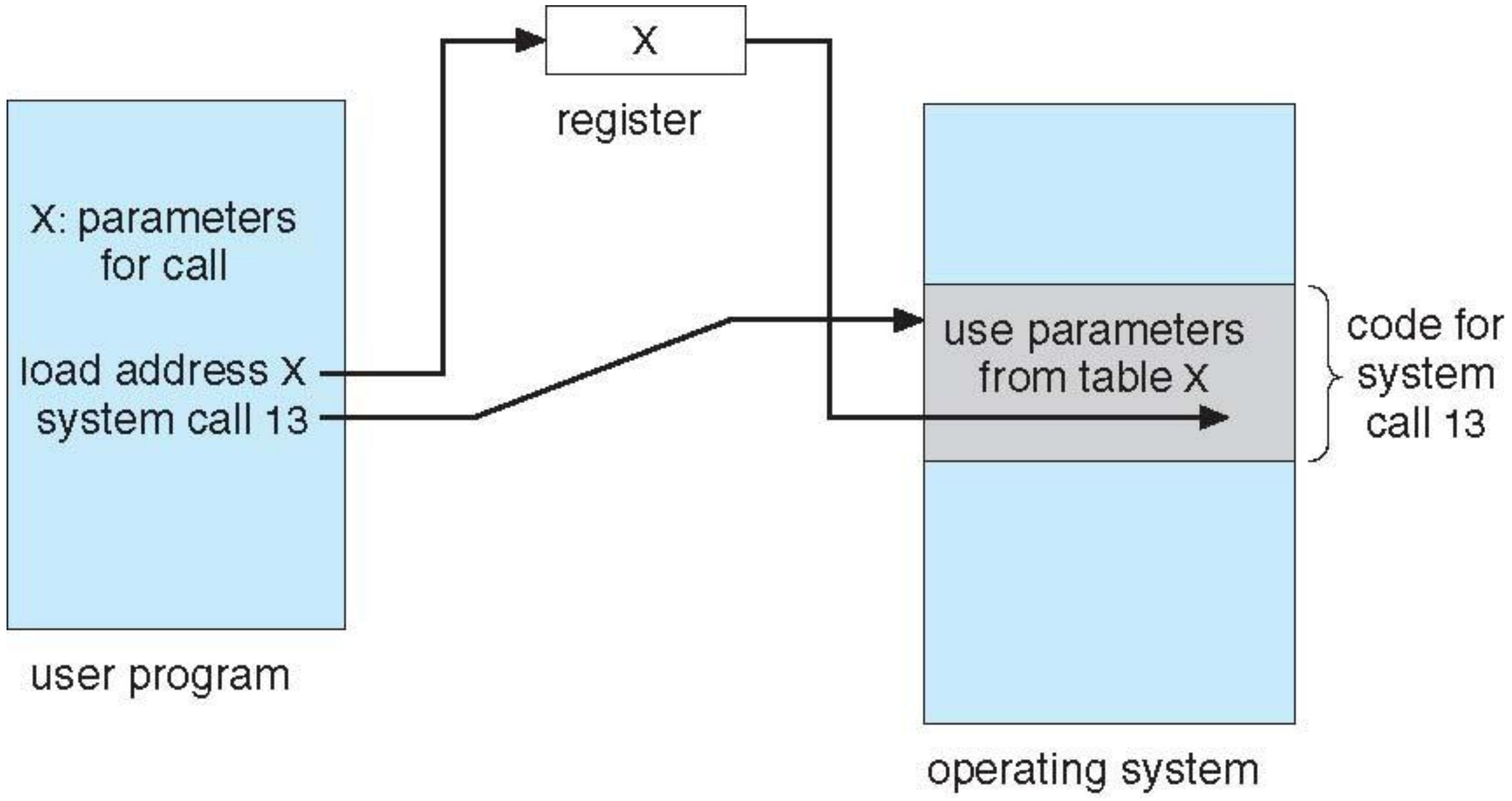


```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

System Call Parameter Passing

- System call is like function calls, so we need to pass parameters to it
- Three general methods to pass parameters to OS
  - Simplest: pass the parameters in *registers*
    - Issue: there may be more parameters than registers
  - Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by OS
  - Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter (e.g., C pointer) in a register
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table

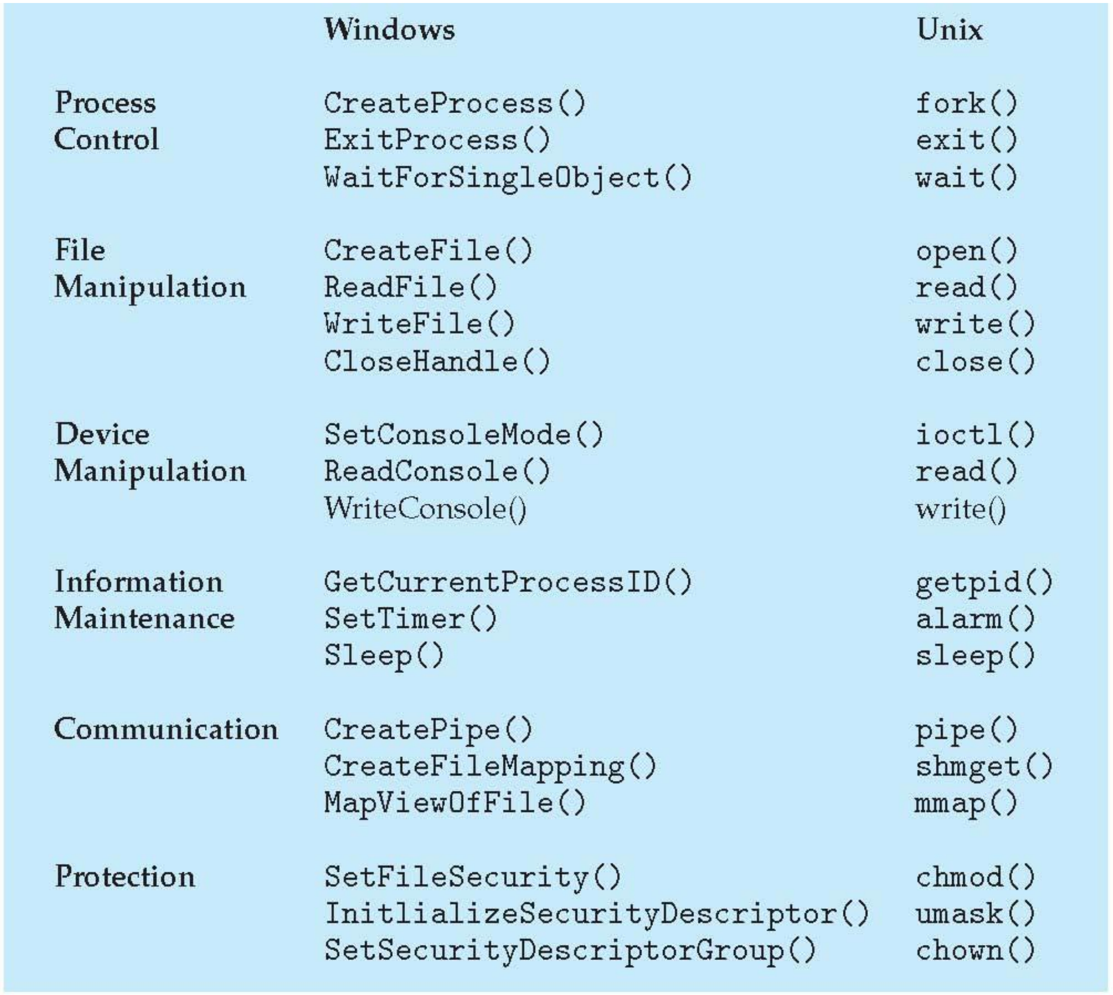


Types of System Calls

- Generally, one set corresponding to each OS subsystem
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communications
  - Protection

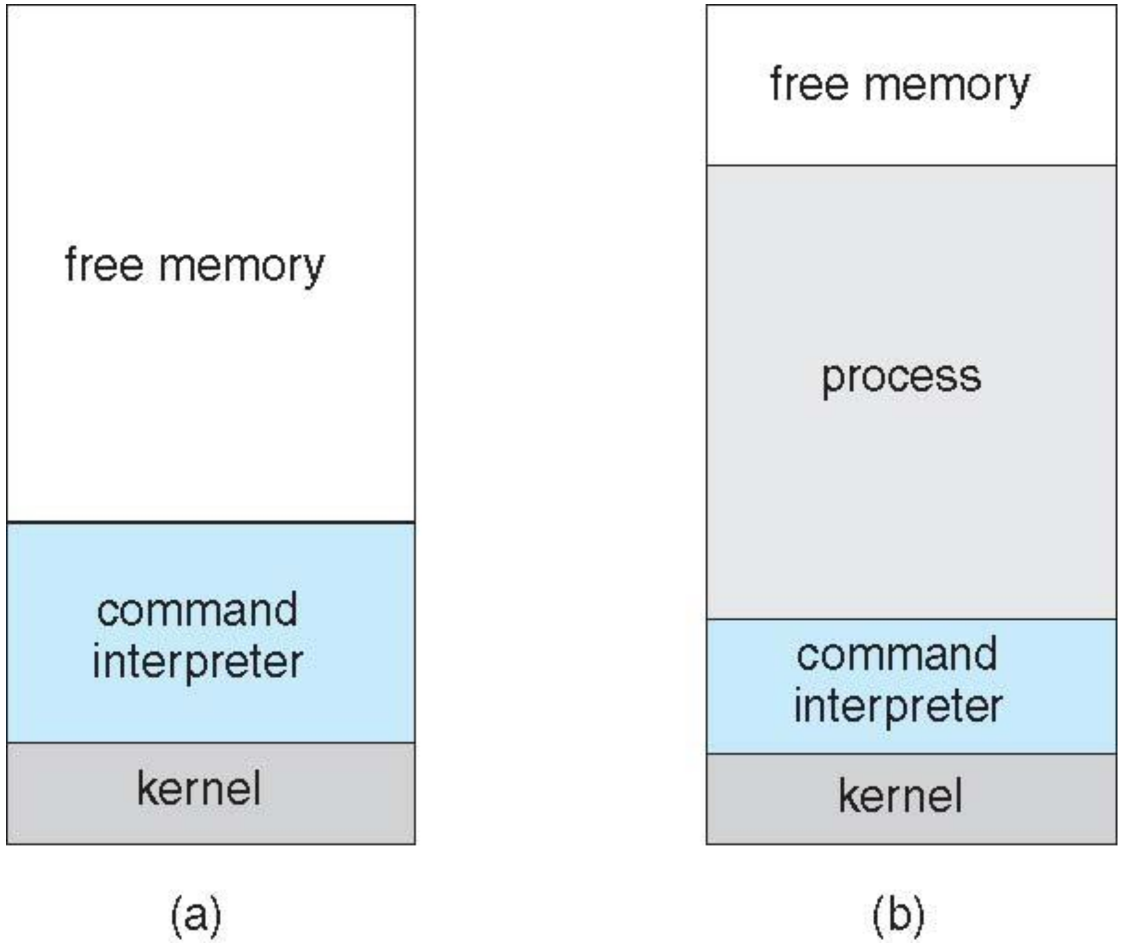
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN
Established in collaboration with MIT

Examples of Windows and Unix System Calls



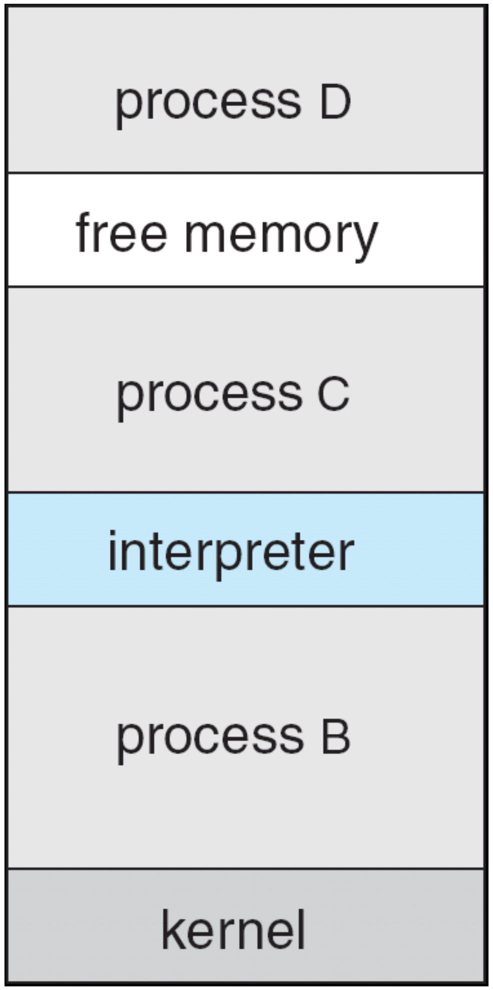
	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information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

MS-DOS execution



(a) At system startup; (b) running a program.
Note the reduced interpreter in (b).

FreeBSD Running Multiple Programs



System Programs

- System programs (e.g., shell commands like **ls**) provide a convenient environment (various tools) for program development and execution. They can be divided into:
  - File manipulation
  - Status information
  - File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Application programs
- Most users' view of OS is defined by the system programs, not the actual system calls
- Sometimes, a system call and a system program has the same name!
  - E.g., **write(1)** vs. **write(2)** – number is section of the API documentation: Sec. 1 is commands; Sec. 2 is system calls

System Programs

- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- File management - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- Status information
  - Some ask the 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 the terminal or other output devices
  - Some systems implement a registry used to store and retrieve configuration information

System Programs (Cont.)

- Editing files
  - Text editors to create and modify files (e.g., C programs)
  - Special commands to search contents of files or perform transformations of the text
- Programming language support – Compilers and assemblers
- Program loading and execution- Linkage editors, loaders, interpreters, debuggers for high-level and machine languages
- Communications – Provide mechanism for creating communication channels among processes, users, and computer systems
  - Allow users to send messages to one another’s screens, browse web pages, send emails, log in remotely (e.g., **ssh**), transfer files between machines (e.g., **sftp**)

OS 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
- Start by defining goals and specifications
- Affected by choice of hardware (e.g., desktops, laptops, smart phones, embedded systems) and purpose of system (e.g., scientific computing, gaming, multimedia, general purpose)
- *User* goals vs. *system* goals
  - User goals – OS should be convenient to use, easy to learn, reliable, safe, fast
  - System goals – OS should be easy to design, implement, and maintain, as well as (i) make users happy (reliable, fast, …) while balancing between needs of many users (flexible, fair, …)

OS Design and Implementation (Cont.)

- Important principle to separate

  Policy: *What* will be done?
  Mechanism: *How* to do it?

- The separation of policy from mechanism allows maximum *flexibility*, e.g., if policy decisions evolve due to internal or external conditions.

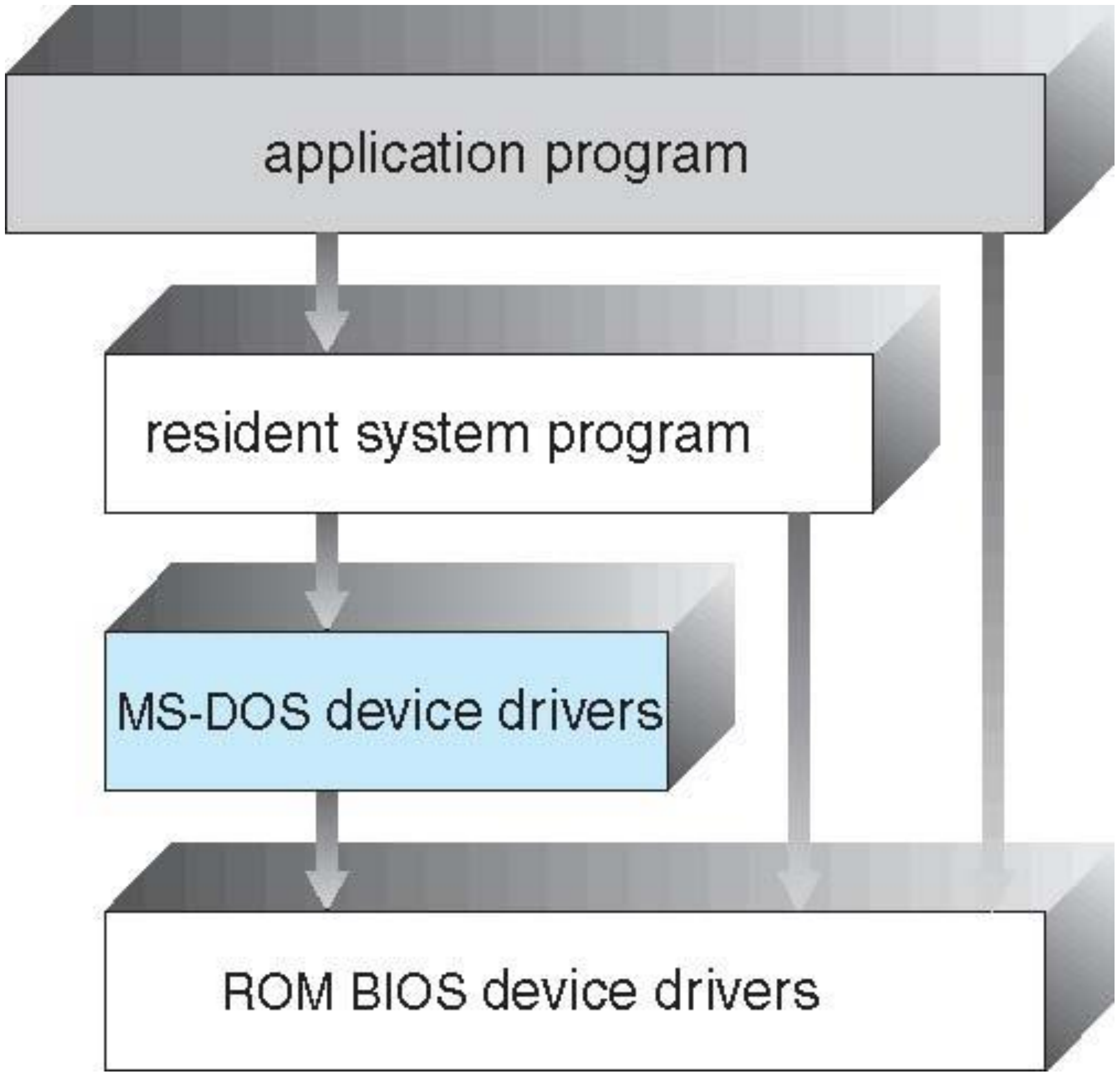
Example 1: Mechanism: r/w/x permission bits for a file to enforce read/write/execute access to it; Policy: what access rights to grant to what users?

Example 2: Mechanism: Periodic timer interrupt (clock *ticks*) to allow rescheduling of CPU to different ready processes; Policy: allow a process to run for how many ticks before we switch it out and switch in another process

Simple Structure

- MS-DOS – written to provide the most functionality in the least space (it's 1980s!)
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated (*monolithic* structure)
- MS-DOS doesn't support dual-mode operation
  - Implications?
  - Maybe okay since MS-DOS isn't meant for multi-user computing?

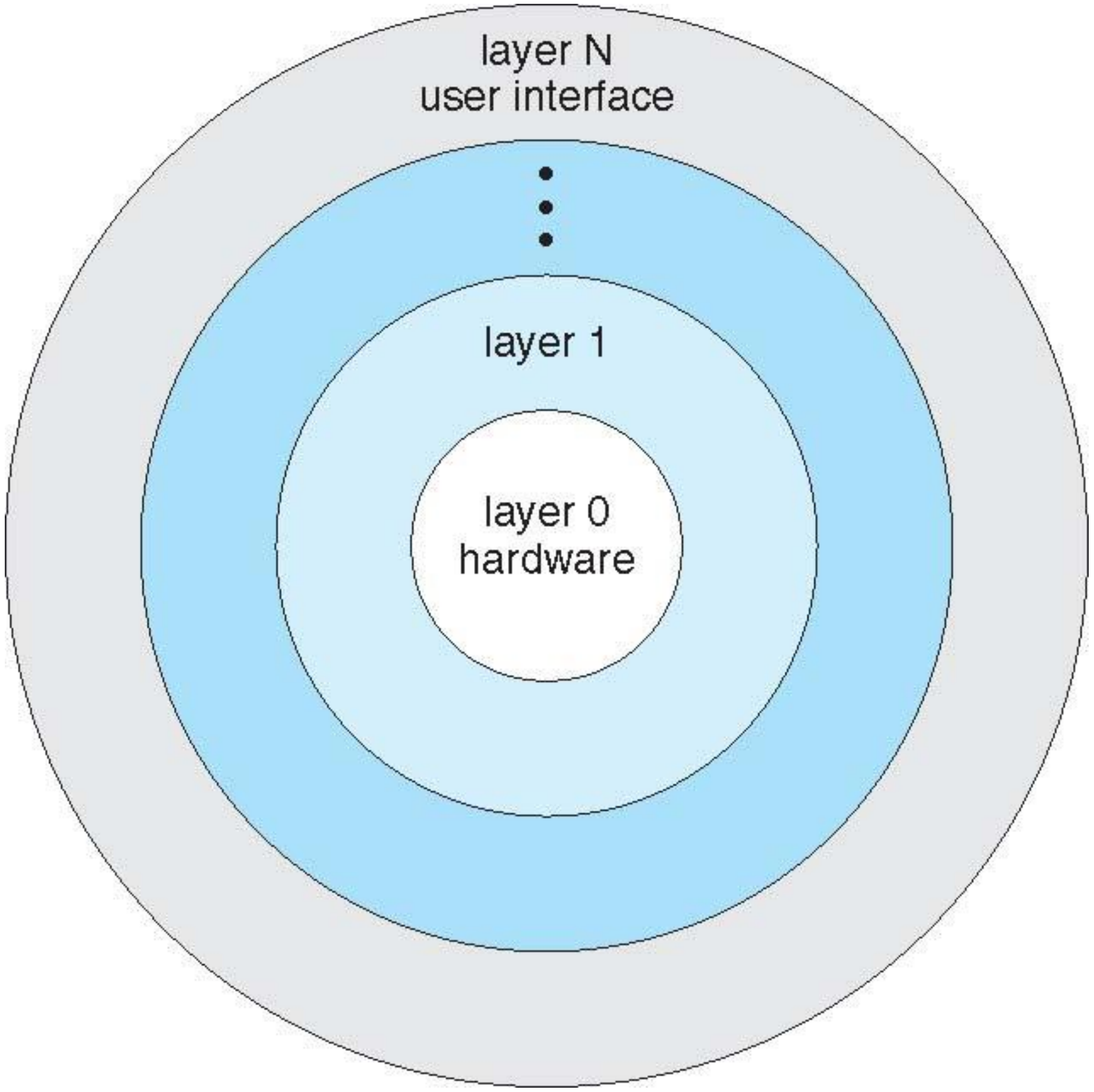
MS-DOS Layer Structure



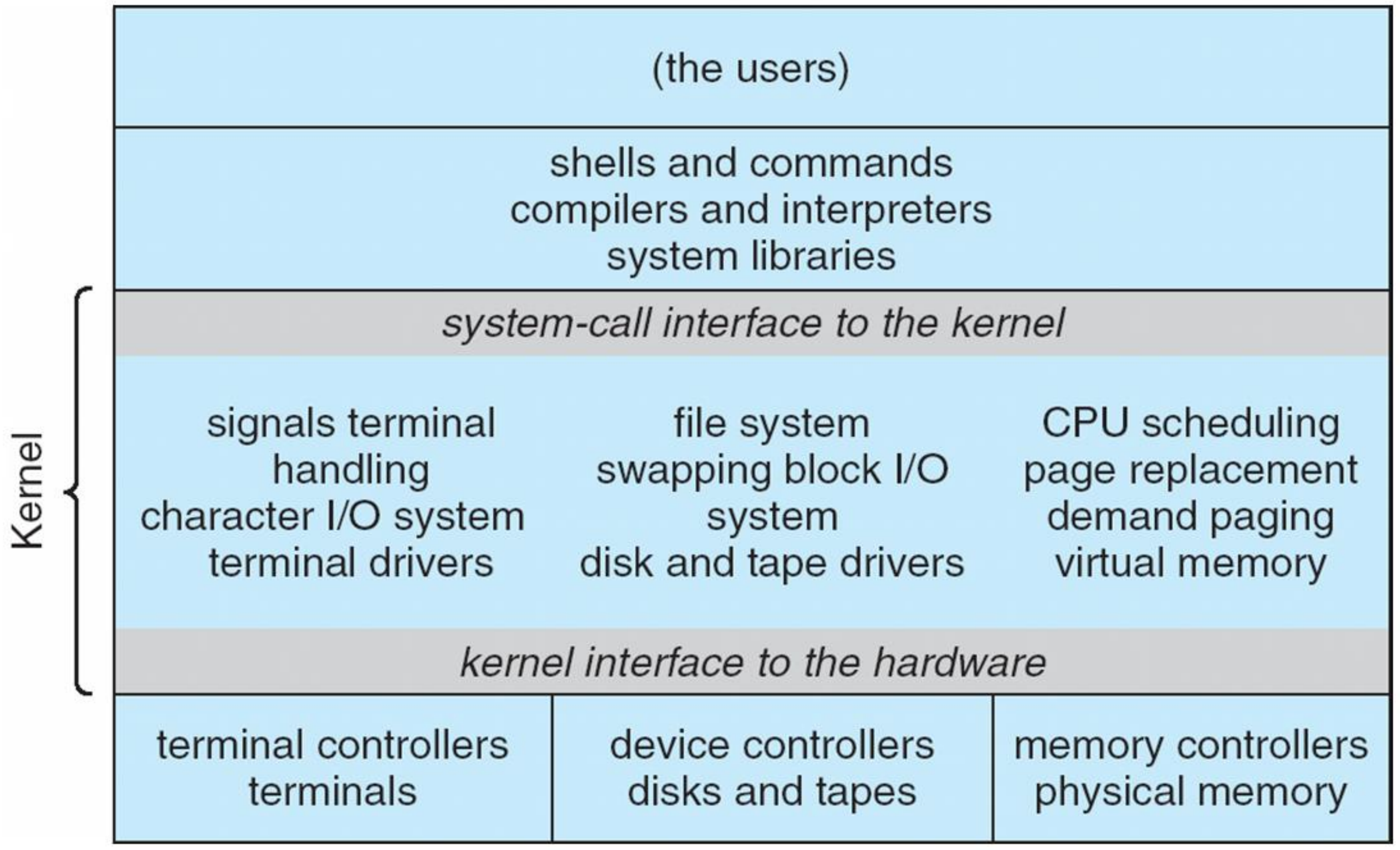
Layered Approach

- OS is divided into a number of layers (levels), each built on top of lower layers. Bottom layer (layer 0) is the hardware; highest layer (layer N) is the user programs.
- Most useful with dual-mode CPU support: OS layer in kernel mode, upper-layer applications in user mode.
- With modularity, layers are selected such that each uses functions and services provided by the layer immediately beneath (or sometimes generalized to only lower layers)
- Allows to focus our attention (one layer at a time)
  - If your program is buggy, what do you debug?
- Turns out to be generally useful across application domains
  - E.g., we'll see it again, in more details, when we study networking systems in this course

Layered Operating System



Traditional UNIX System Structure



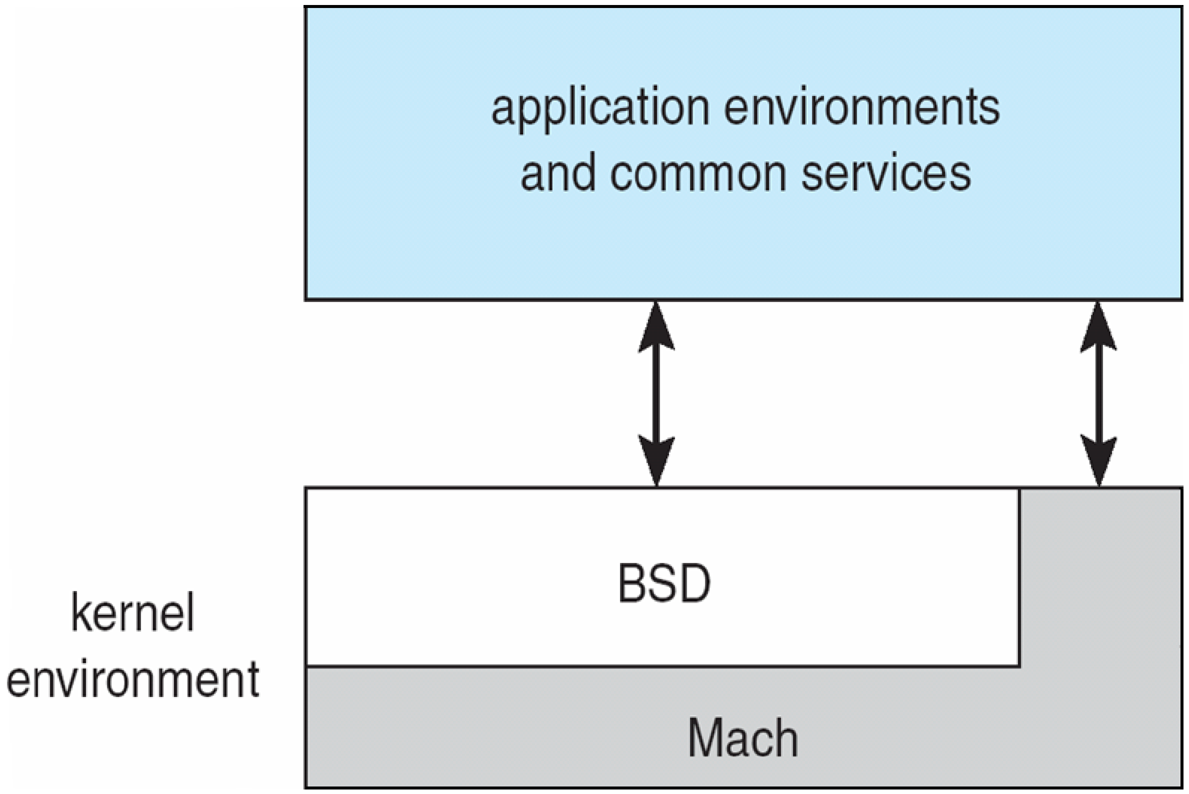
UNIX

- UNIX consists of two main parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides file system, CPU scheduling, memory management, and other OS functions; a large number of functions for one level

Microkernel System Structure

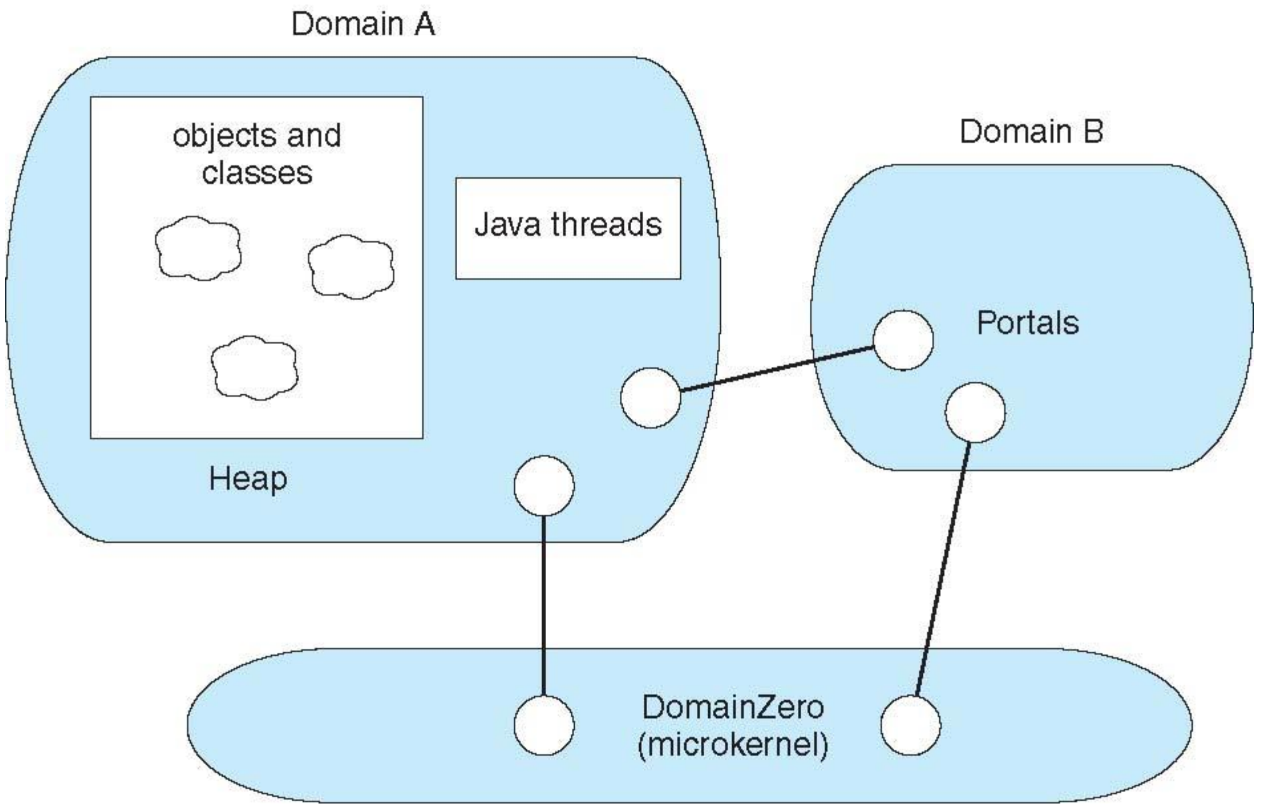
- Move as much services from the kernel into "*user*" space (e.g., Mach OS from CMU) – different OS subsystems run as different user-level processes (e.g., one process for memory management, another for CPU management, etc)
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port OS to new architectures
  - Better reliability and modularity: processes (i.e., OS subsystems) are protected from each other (cf. Unix OS subsystems all running in kernel mode)
  - User-space code easier to debug
- Disadvantages:
  - Performance overhead of communication between user space and kernel space

Mac OS X Structure



Java Operating Systems

The JX operating system


Domain A
objects and classes
Java threads
Heap
Domain B
Portals
DomainZero (microkernel)