ECE30021/ITP30002 Operating System

System Structure
(OSC:Ch.2)

This lecture note is taken from the instructor's resource of Operating System Concept, 9/e and then partly edited/revised by Shin Hong.

Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users

- An operating-system typically provides services of
  - **user interface**: e.g., Command-Line (CLI), Graphics User Interface (GUI), Batch
  - **program execution (process control)**: the system must be able to load a program into memory, to run the program, and to end an execution either normally or abnormally (indicating error)
  - **I/O operations**: communication via files or device drivers

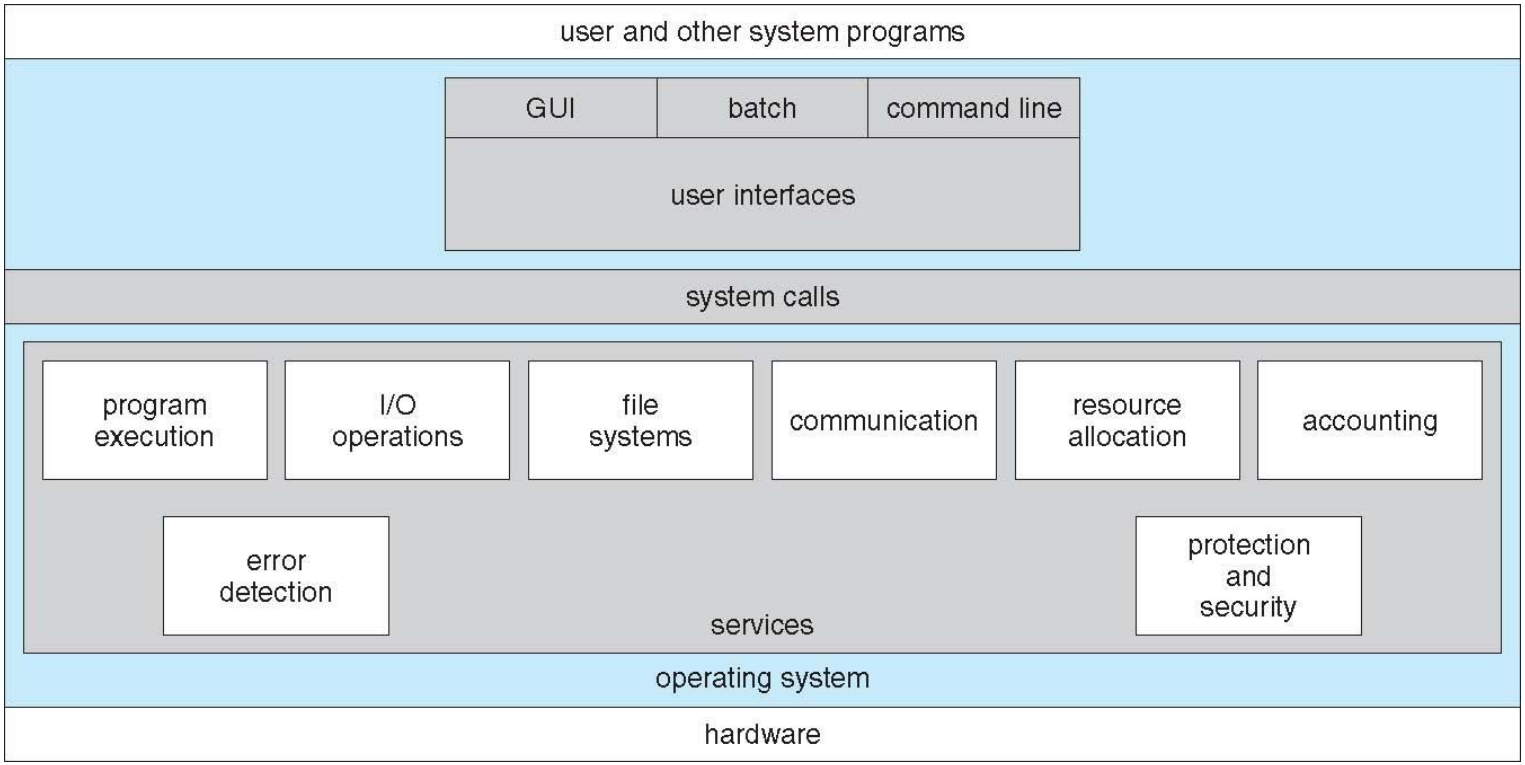
Operating System Services

- An operating-system typically provides services of (cont'd)
  - **File-system manipulation**: read and write files and directories, create and delete them, search them, list file info, manage permission
  - **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**: be constantly aware of possible errors
    - errors may occur in CPU, memory, 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.)

- An operating-system typically provides services of (cont'd)
  - **Resource allocation**: When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - managing CPU cycles, main memory, file storage, I/O devices, etc.
  - **Accounting**: To keep track of which users use how much and what kinds of computer resources
  - **Protection and security**: The owners of information stored in a multiuser or networked computer system may want to control use of that information
    - **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



User Operating System Interface - CLI

CLI or command line 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
- Sometimes commands built-in, sometimes just names of programs
  - If the latter, adding new features doesn't require shell modification

User Operating System Interface - GUI

- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc.
  - Invented at Xerox PARC

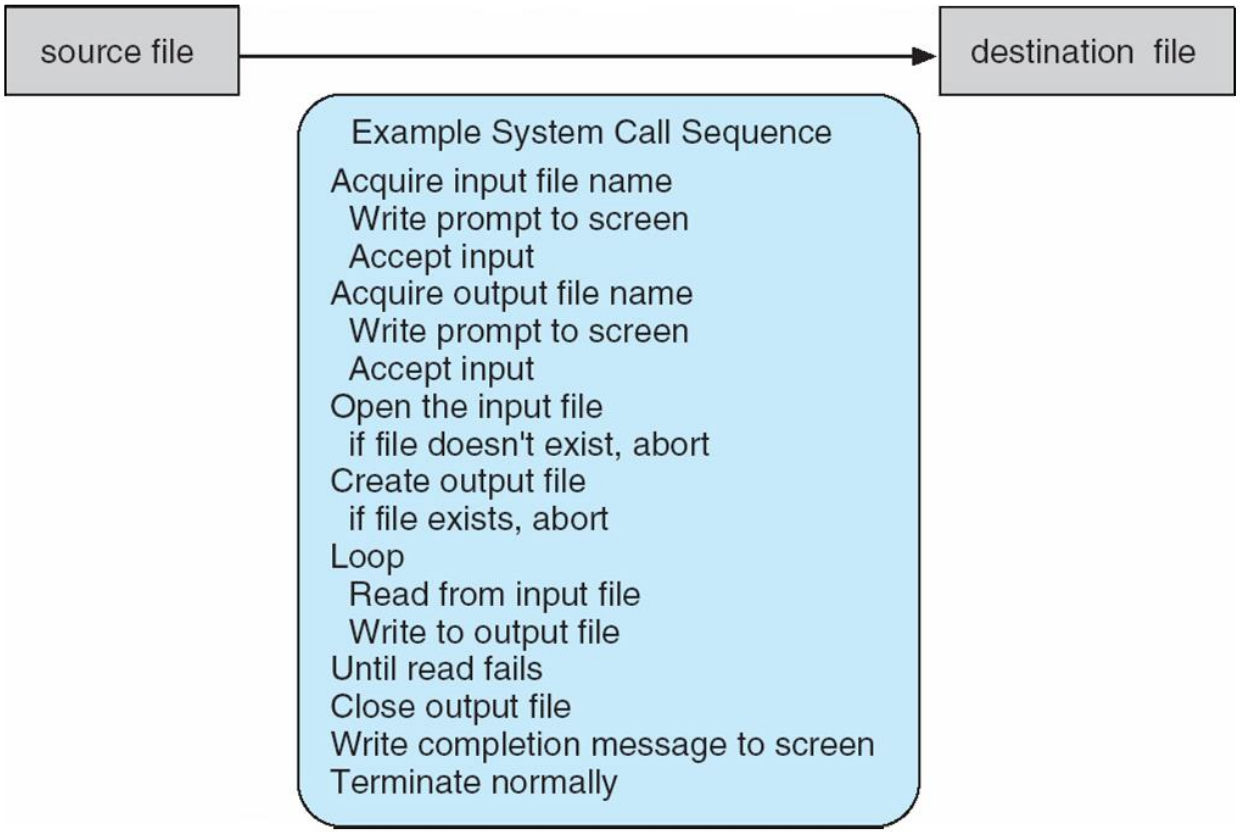
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use
  - Examples
    - WinAPI for Windows
    - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
    - Java API for the Java Platform

Example of System Calls

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



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:

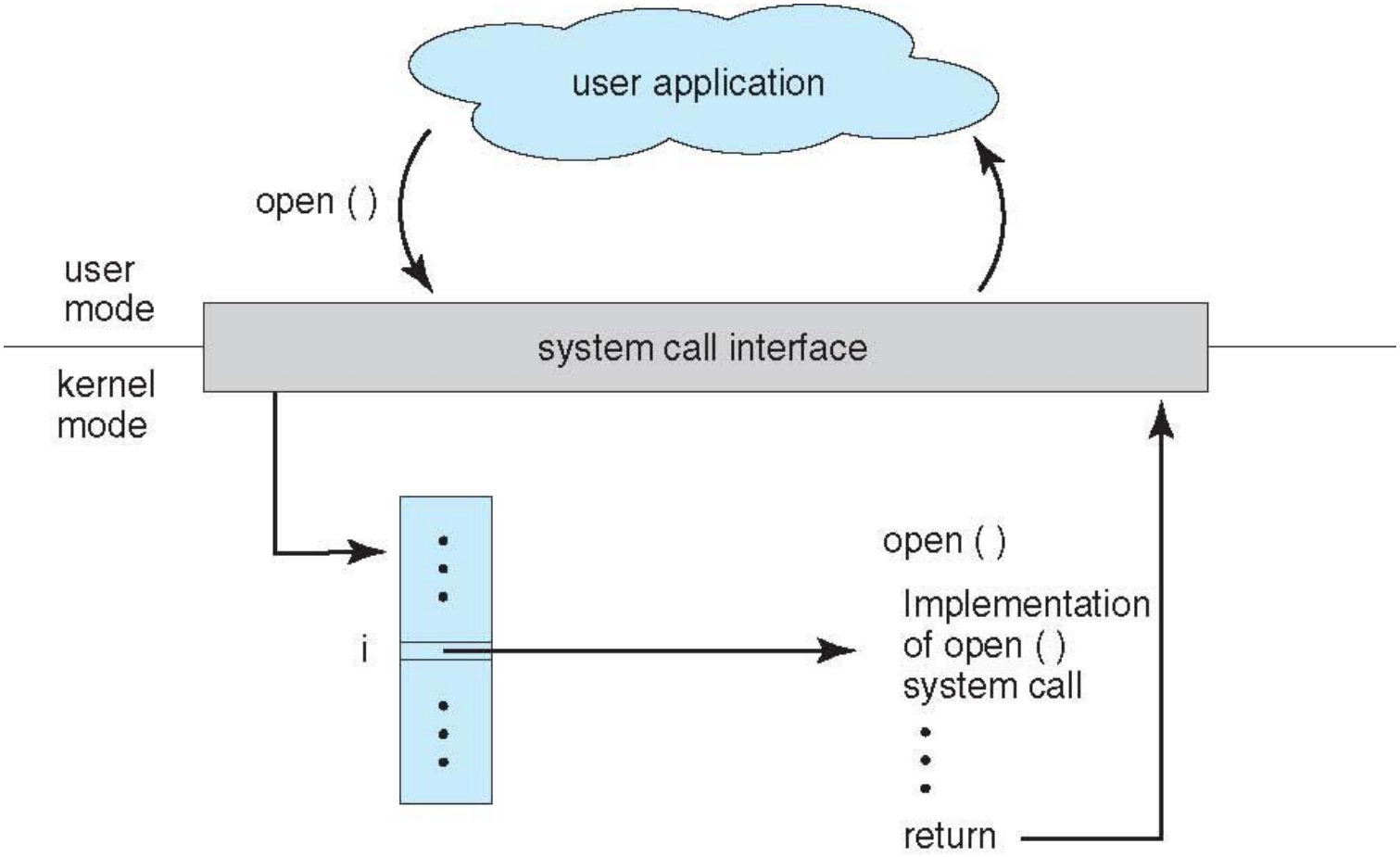
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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

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

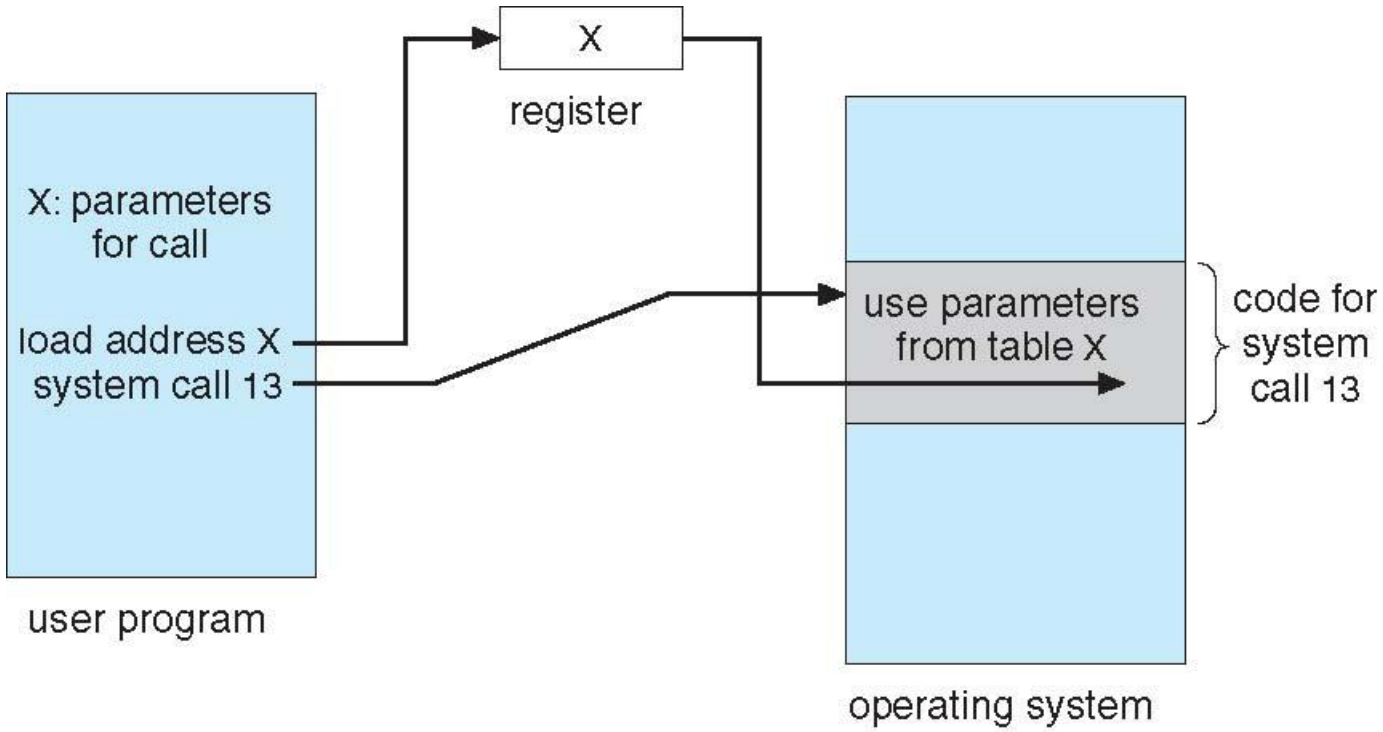
API – System Call – OS Relationship



System Call Parameter Passing

- Three general methods used to pass parameters to the OS
  - Store in registers
    - In some cases, may be more parameters than registers
  - Store in program stack
    - Just like function call
  - Store in a block (or table) in user memory space, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



x86 Linux System Call Convention

1. User-level applications use as integer registers for passing the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`. The kernel interface uses `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8` and `%r9`.
2. A system-call is done via the `syscall` instruction. The kernel destroys registers `%rcx` and `%r11`.
3. The number of the syscall has to be passed in register `%rax`.
4. System-calls are limited to six arguments, no argument is passed directly on the stack.
5. Returning from the `syscall`, register `%rax` contains the result of the system-call. A value in the range between -4095 and -1 indicates an error, it is `-errno`.
6. Only values of class INTEGER or class MEMORY are passed to the kernel.

* System V Application Binary Interface – AMD64 Architecture Processor Supplement, Nov 17, 2014

Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - 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

Types of System Calls (Cont'd)

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- 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'd)

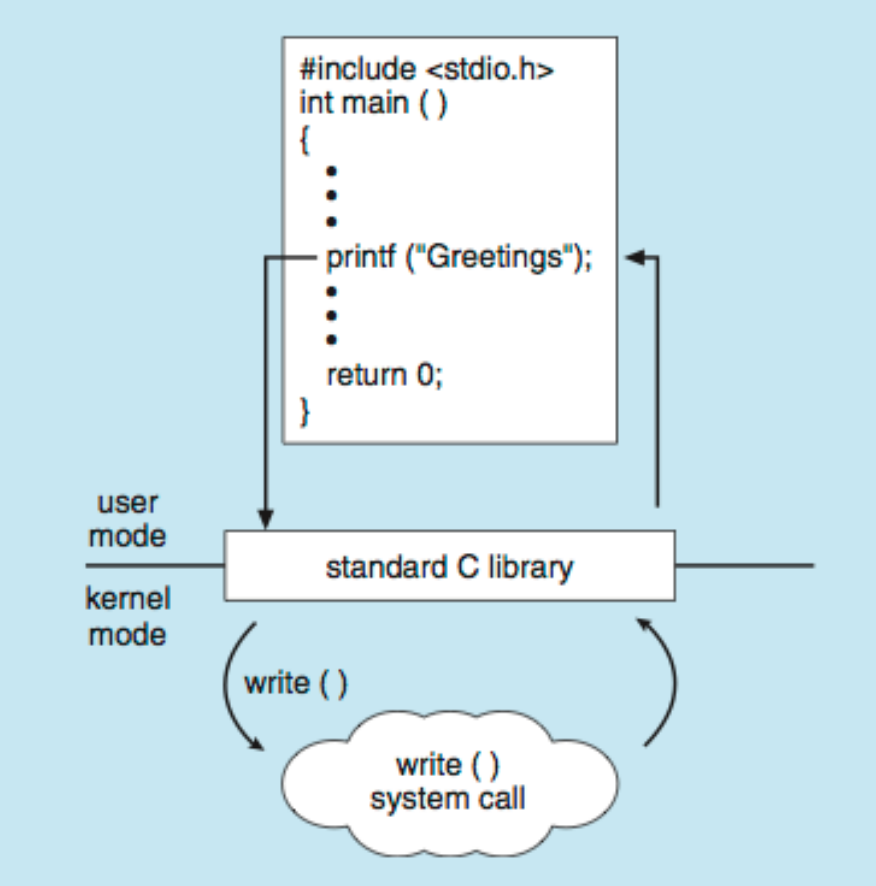
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if message passing model to host name or process name
    - From client to server
  - Shared-memory model create and gain access to memory regions
  - transfer status information
  - attach and detach remote 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() 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()

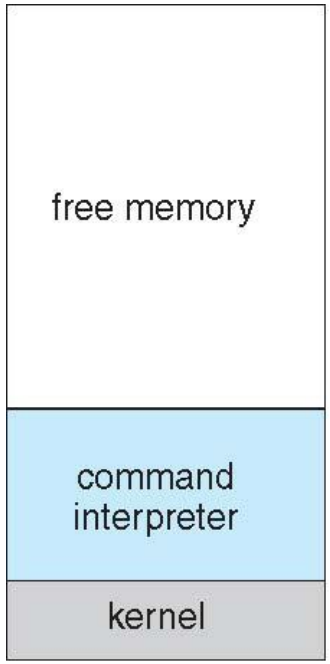
Standard C Library Example

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



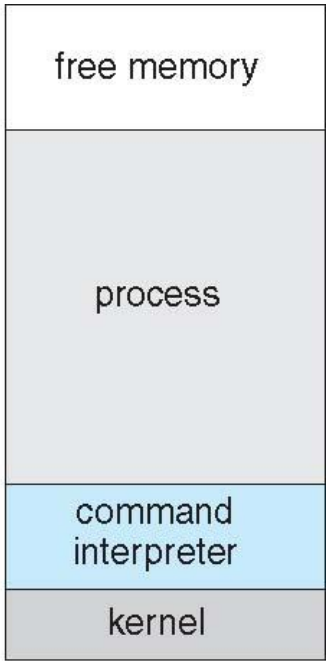
Example: MS-DOS

- 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 the kernel
- Program exit → shell reloaded



(a)

At system startup

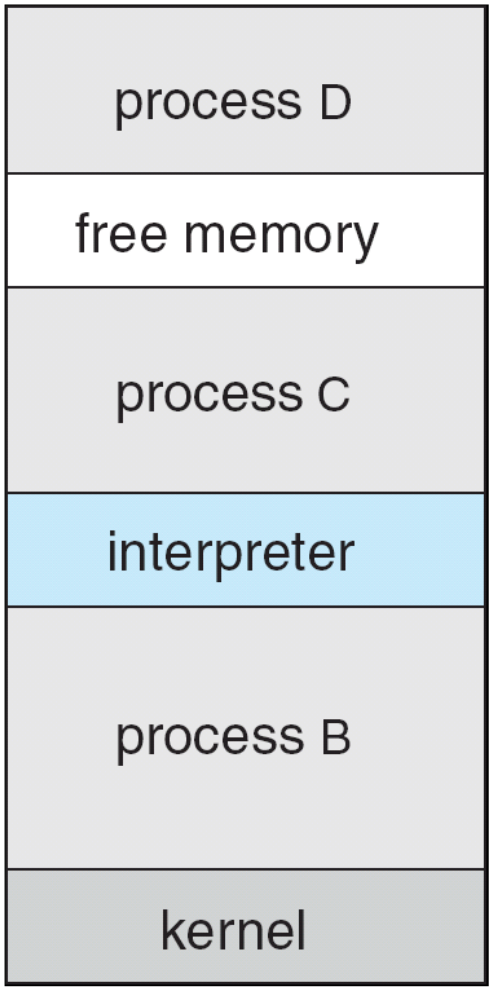


(b)

running a program

Example: FreeBSD

- Unix variant
- Multitasking
- User login →
  invoke user's choice of shell
- 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 when no error occurs
  - code > 0 when an error occurs



System Programs

- System programs provide a convenient environment for program development and execution.
  - File manipulation
  - Status information sometimes stored in a File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs
- Most users' view of the operation system is defined by syste m programs not the actual system calls
  - Many system programs are simply user interface wrapper of system calls; others are considerably more complex

System Programs (Cont'd)

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

System Programs (Cont'd)

- **Programming-language support**
  - Compilers, assemblers, debuggers and interpreters
- **Program loading and execution**
  - Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications**
  - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to others' screens, send email, log in remotely, transfer files from one machine to another

System Programs (Cont'd)

- **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
  - Run in user context not kernel context
  - Known as services, subsystems, daemons

Operating System Design and Implementation

- No absolute solution, yet some approaches have proven successful
- Goals: 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
- Separation of policy and mechanism
  - Policy: *What* will be done?
  - Mechanism: *How* to do it?
  - The separation of policy from mechanism is to allow maximum flexibility if policy decisions are to be changed later

Implementation

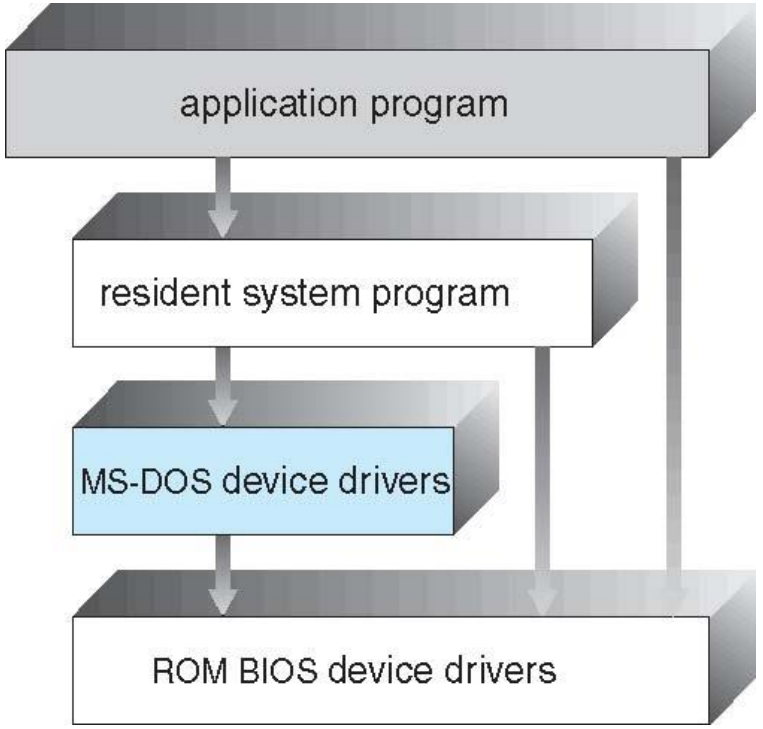
- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/I
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to port to other hardware
  - Less chance of optimization if compiler support is weak
- Emulation can allow an OS to run on non-native hardware

Operating System Structure

- General-purpose OS is a very large program
- Various ways to structure ones
  - Naive structure, e.g., MS-DOS
  - More complex, e.g., UNIX (layered)
  - Microkernel, e.g., Mach

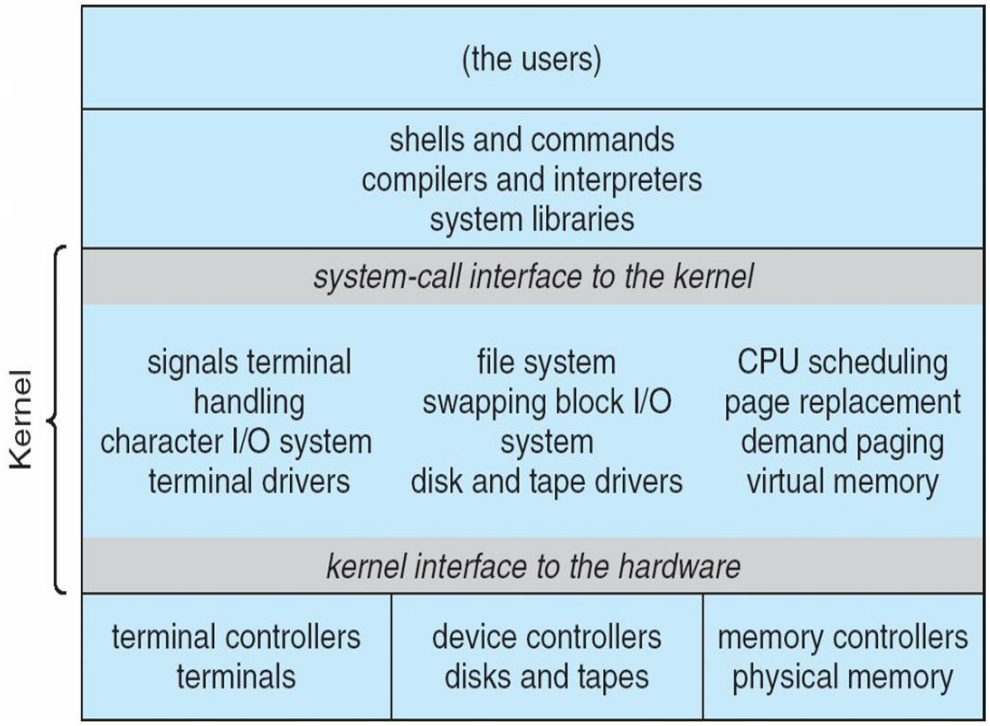
Simple Structure -- MS-DOS

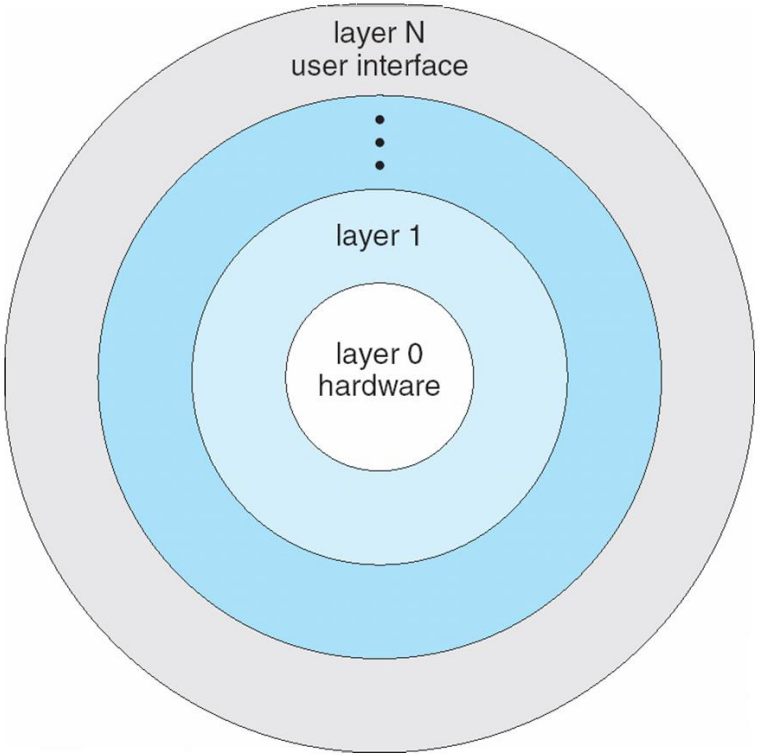
- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



Non Simple Structure -- UNIX

- UNIX consists of two separable parts
  - Systems programs
  - Kernel
- Traditional vs. modern (layered)

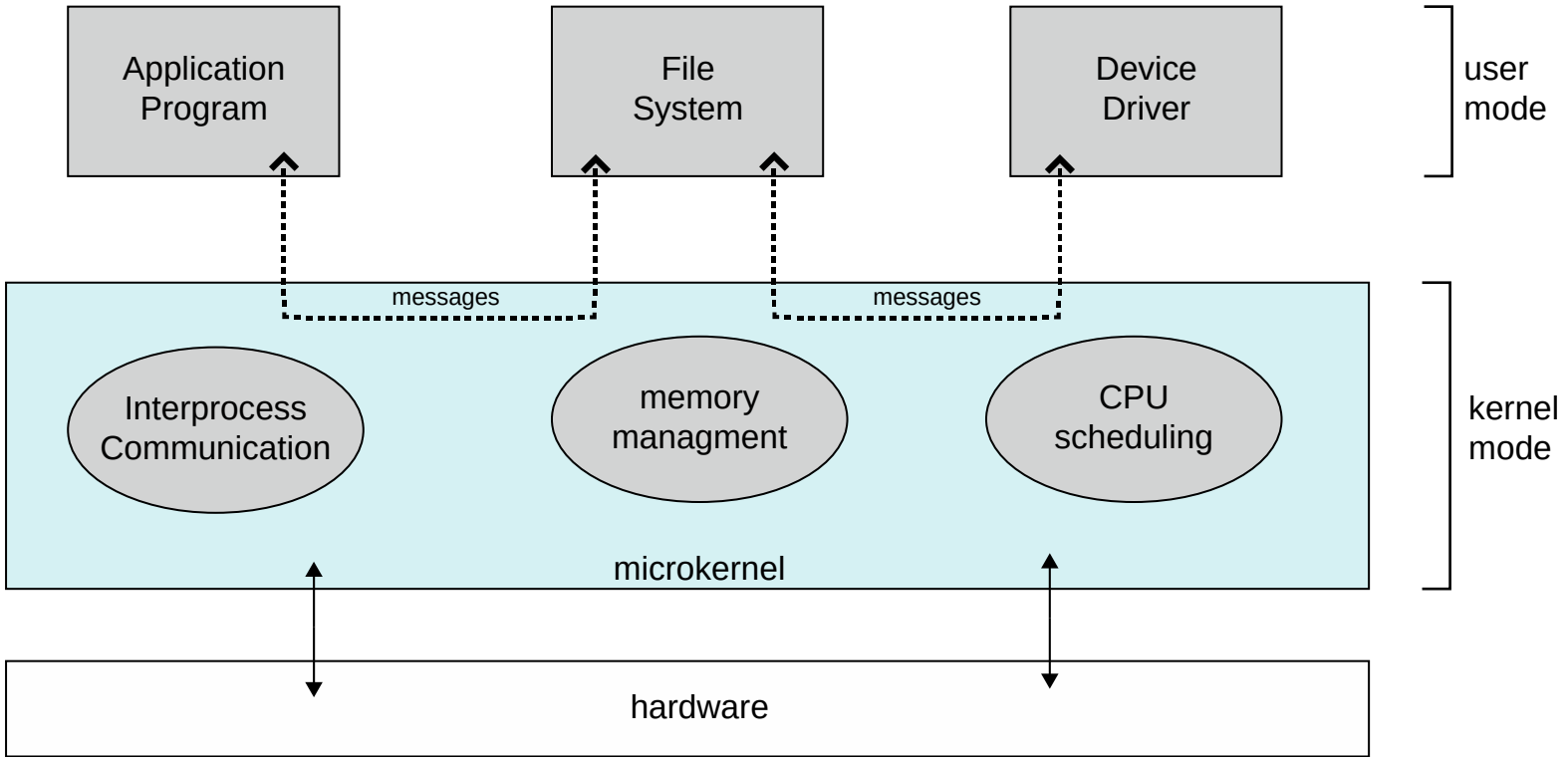




Microkernel System Structure

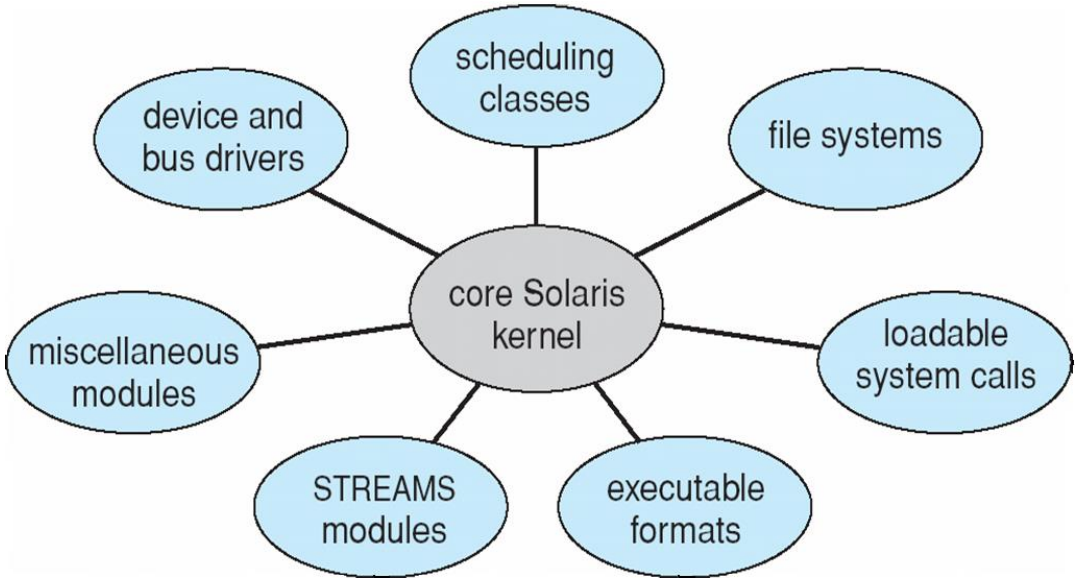
- Moves as much from the kernel into user space
- Mach example of microkernel
  - Mac OS X kernel (Darwin) partly based on Mach
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures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

Microkernel System Structure



Modules

- Many modern OSes implement loadable kernel modules
  - Uses object-oriented approach
  - Each core component is separated
  - Each module talks to the others over known interfaces
  - Each module is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc.



Hybrid Systems

- Most modern OSes are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security and 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, Aqua UI plus Cocoa programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called kernel extensions)