Chapter 2: System Structures

Chien Chin Chen

Department of Information Management
National Taiwan University

Objectives

- To describe the **services** an operating system provides to users, processes of the system.
- To discuss the various ways of **structuring** an operating system.
- To explain how operating systems are installed and customized and how they boot.

Operating System Services (1/4)

- One set of operating-system services provides functions that are **helpful to the user** (or user processes):
  - **User interface** - almost all operating systems have a user interface (UI).
    - ***Command-line interface*** (CLI): require a program to allow entering and editing of text commends.
    - ***Graphics user interface*** (GUI): a window system with a pointing device and a keyboard to enter commends.
    - ***Batch***: commands and directives are entered into files to be executed.
  - **Program execution** - the system must be able to load a program into memory and to run that program, end execution, either normally or abnormally.

Operating System Services (2/4)

- **I/O operations** - a user program may require I/O.
  - For efficiency and protection, users cannot control I/O devices directly.
  - The operating system must provide a means to do I/O.
- **File-system manipulation** - user programs need to read/write/create/delete/search files and directories.
  - The operating system provides permission management to allow or deny access to files or directories.
- **Communications** – user processes may exchange information, on the same computer or between computers over a network.
  - Communications may be via *shared memory* or through *message passing*.
- **Error detection** – the operating system needs to be constantly aware of possible errors.
  - And fix errors generated from hardware (disk fail) or software (arithmetic error).

Operating System Services (3/4)

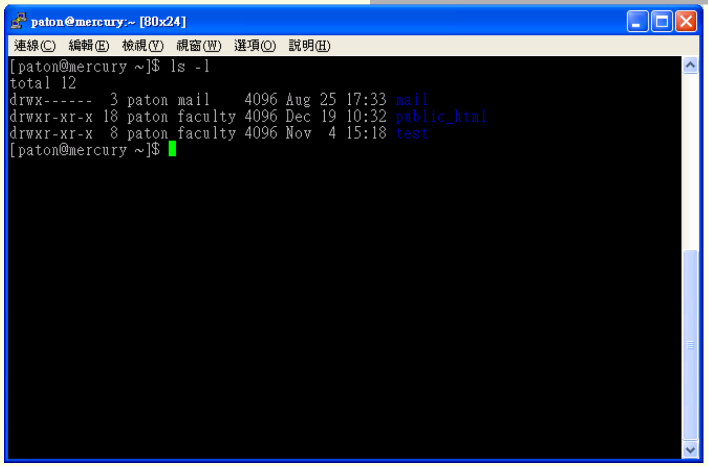
- For systems with **multiple users** (processes), another set of operating-system functions exists for ensuring **the efficient operation of the system itself**.
  - **Resource allocation** - when multiple users or multiple jobs running concurrently, resources must be allocated to each of them.
    - CPU, memory, file storage …
    - Operating systems have *CPU-scheduling routines* to determine the best way to use the CPU.
  - **Accounting** - To keep track of which users use how much and what kinds of computer resources.
    - Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.

Operating System Services (4/4)

- **Protection and security** - a multiuser or networked computer system may want to control use of user 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 (e.g., network adapters) from invalid access attempts.

User Operating-System Interface – CLI (1/4)

- ***Command-line interface*** (interpreter):
  - Primarily **get** and **execute** the next **user-specified command**.
  - Many of the commands are to manipulate files/directories.
    - Create/delete/list/print/copy/execute…



User Operating-System Interface – CLI (2/4)

- Two ways to implement commands and command interpreters:
  - The command interpreter contains the code to execute the command.
    - For example, a command to delete a file may cause the interpreter to jump to a section of its code that makes the appropriate system call.
    - The number of commands that can be given determines the size of the interpreter.
  - An alternative approach implements **most commands** through system programs.
    - If the interpreter does not understand the command ...
    - It **identify** the command file and **load** it into memory for **execution**. 在特定的目錄下去讀指令

User Operating-System Interface – CLI (3/4)

- (cont.)
  - An UNIX example: `rm file.txt` remove
    - The interpreter search for a file called `rm` (`/bin/rm`).
    - **Load** it into memory and **execute** it with the parameter `file.txt`.
    - The function `rm` is completely defined by the code in the file `/bin/rm`.
  - Programmers can add new commands to the system easily.
  - The interpreter program can be **small**, and does not have to be changed for new commands to be added.
  - Used mostly among operating system, e.g., UNIX.

User Operating-System Interface – CLI (4/4)

- CLI is sometimes implemented in kernel, sometimes by systems program.
- An operating system can have multiple獨立的program interpreters to choose from, known as ***shells***.
  - For example, on UNIX and Linux systems, there are *`Bourne/C/Korn`*…shell.
  - The name 'shell' originates from shells being an outer layer of interface between the user and the innards of the operating system (the kernel).
  - Most shells provide similar functionality with only minor differences; most users choose a shell based upon personal preference.
    - E.g., the syntax of *shell script*.

給shell跑的program

User Operating System Interface – GUI (1/2)

- A GUI provides a ***desktop*** metaphor interface.
  - ***Icons*** represent files, programs, actions, etc.
  - A ***mouse click*** can invoke a program, select a file …
- GUI Timeline:
  - Experimentally appeared in the early 1970s.
  - Became widespread by Apple Macintosh computer (Mac OS) in the 1980s.
  - Dominated by Microsoft Windows (3.1, NT, 95, 98, 2000, XP, Vista).
- UNIX systems have been dominated by command-line interface.
  - Although there are various GUI interface available.
    - X-Windows systems, K Desktop Environment (KDE) by GNU project (open source – source code is in the public domain).

User Operating System Interface – GUI (2/2)

- Preference:
  - Many UNIX users prefer a command-line interface.
  - Most Windows user are pleased to use the Windows GUI and never use the MS-DOS shell interface.
- Nevertheless, many systems now include both CLI and GUI interfaces.

System Calls (1/11)

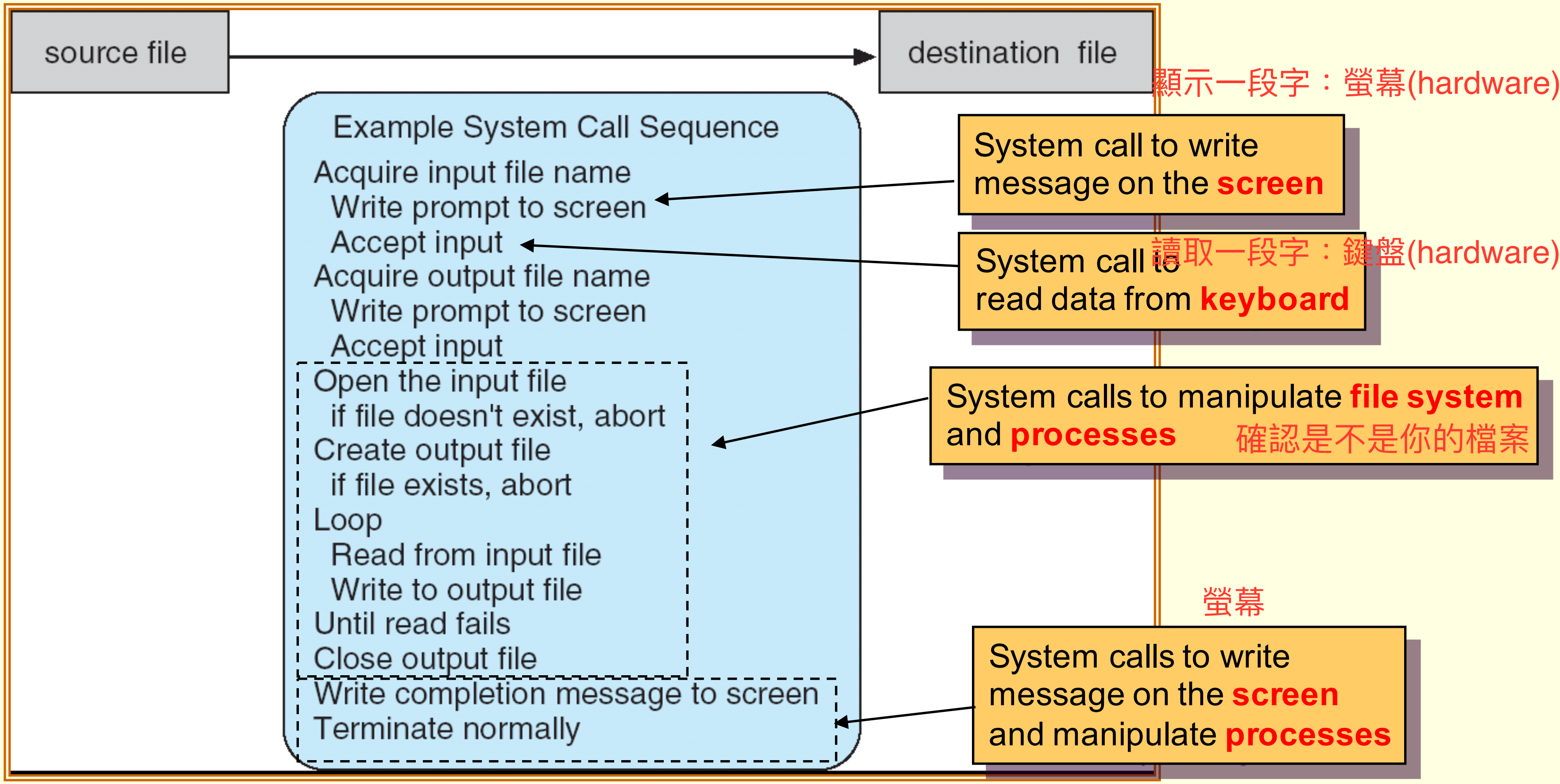
- Can be regarded as a ***programming interface*** ***to the services provided by the OS***.
  - Called by user applications.

service就是一個一個的routine,存在OS裡(kernel mode)
希望使用者寫程式叫他 (user mode)

- Are generally available as ***routines***, typically written in a high-level language (C or C++).
  - Also in low-level assembly language (for accessing hardware).

System Calls — An Example Program of File Copy (2/11)

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



Even a simple program makes heavy use of system calls!!

System Calls (3/11)

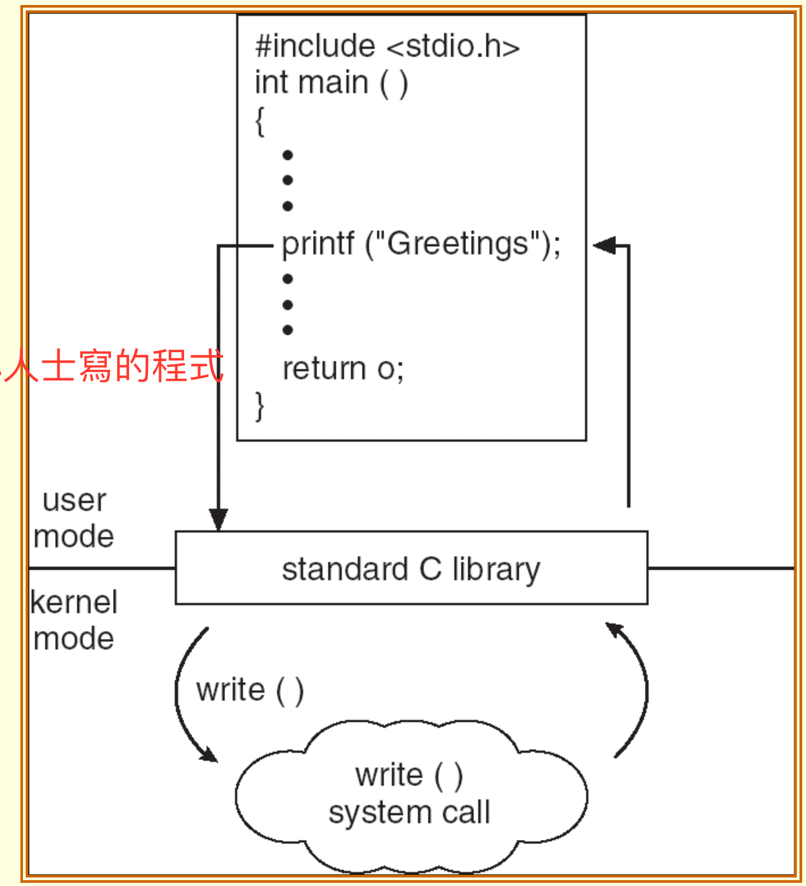
- Programs mostly access system services via a high-level ***application program interface*** (API) rather than using system call directly.
  - API specifies a set of functions (specifications) that are available to programmers.
  - **The functions of the API invoke the actual system calls on behalf of the programmer**.
- 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).

Portable Operating System Interface

System Calls (4/11)

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

善心人士寫的程式



mode的轉換

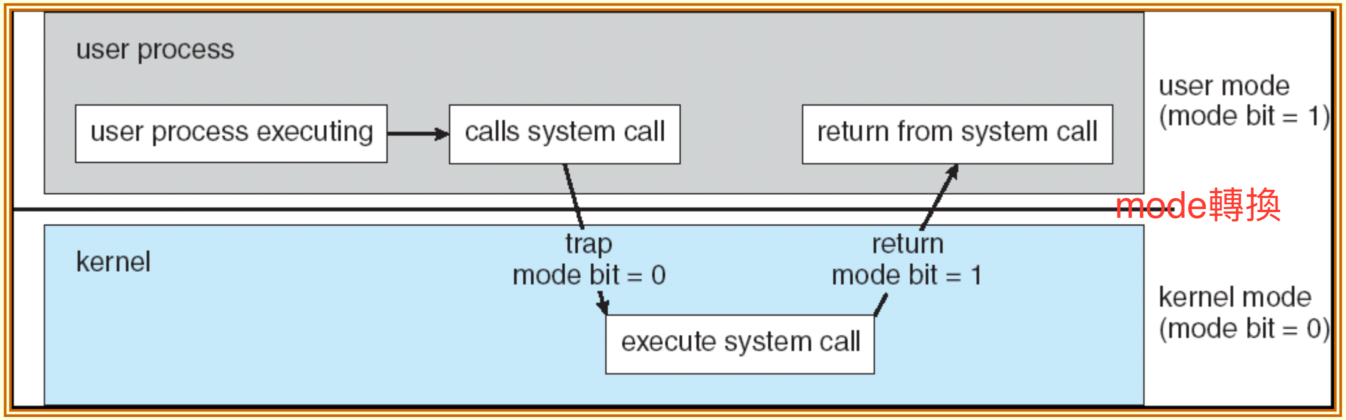
System Calls (5/11)

- Why use APIs rather than system calls?
  - **Program portability** — a program using an API can be expected to compile and run on any system that supports the same API.
    - E.g., your Windows programs on various versions of Windows.
  - Programming with API is more **simpler** than using system calls.

System Calls (6/11)

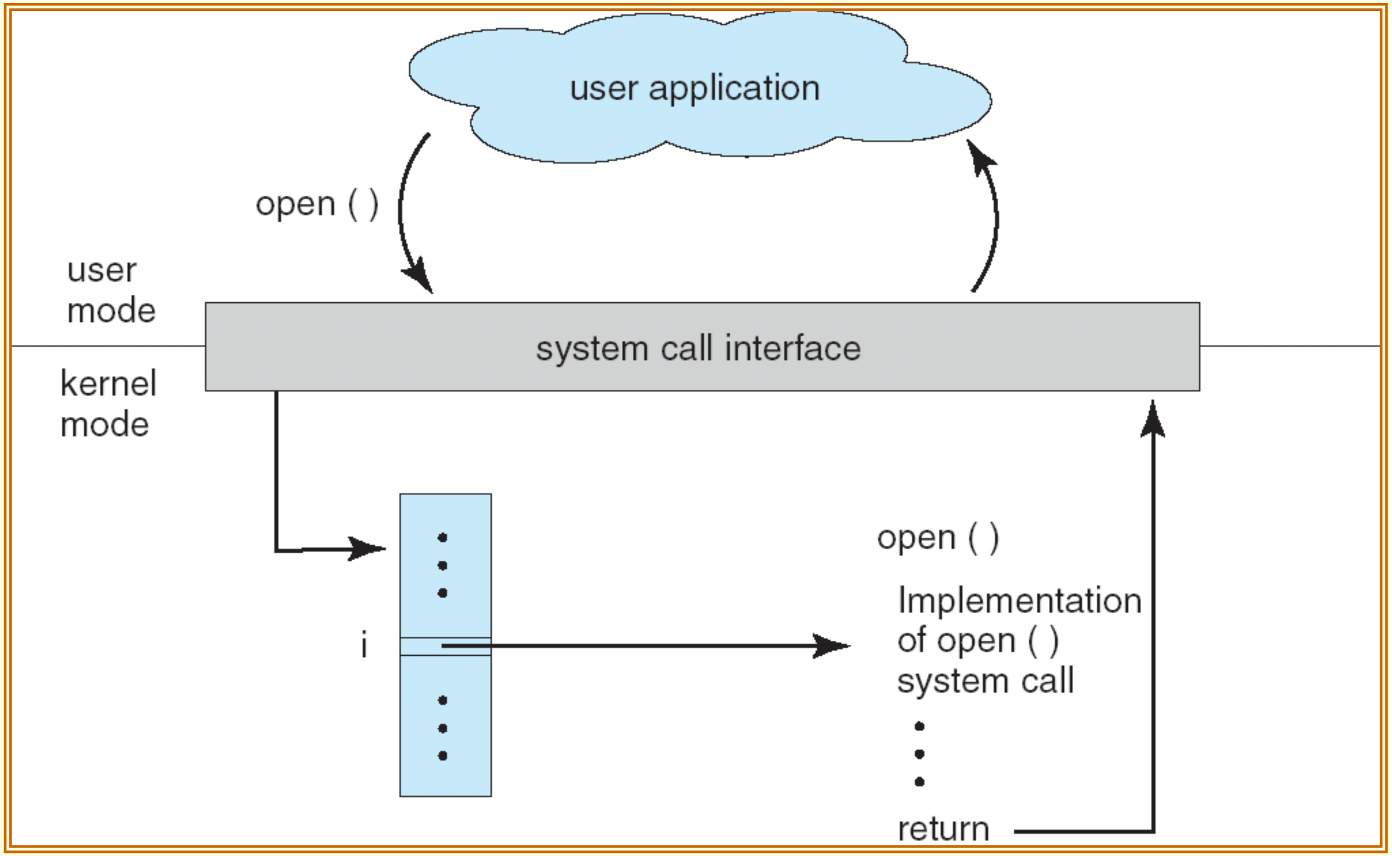
- As system calls are routines in kernel space, using it causes a **change in privileges**.
- How ?
  - Via software interrupt (e.g., `INT 0x80` assembly instruction on Intel 386 arch of Linux system).
- But before that …
  - Similar to hardware interrupt, we need a number (index) to indicate the required system call, which is store in the `EAX register`.
- System contains a table of code pointers.
  - Using the system call number, we jump to the address of the system call for execution.

API helps us wrap all the details by simply invoking a library function.



System Calls (7/11)

- How the operating system handles a user application invoking the `open()` system call through API.



System Calls (8/11)

- To **link** system call made available by the operating system:
  - The **run-time library** for most programming languages provides a **system-call interface**.
  - Typically, a **number** associated with each system call.
  - System-call interface (or kernel) maintains a **table** indexed according to these numbers.
  - The system call interface invokes intended system call in operating system kernel and returns status of the system call and any return values.

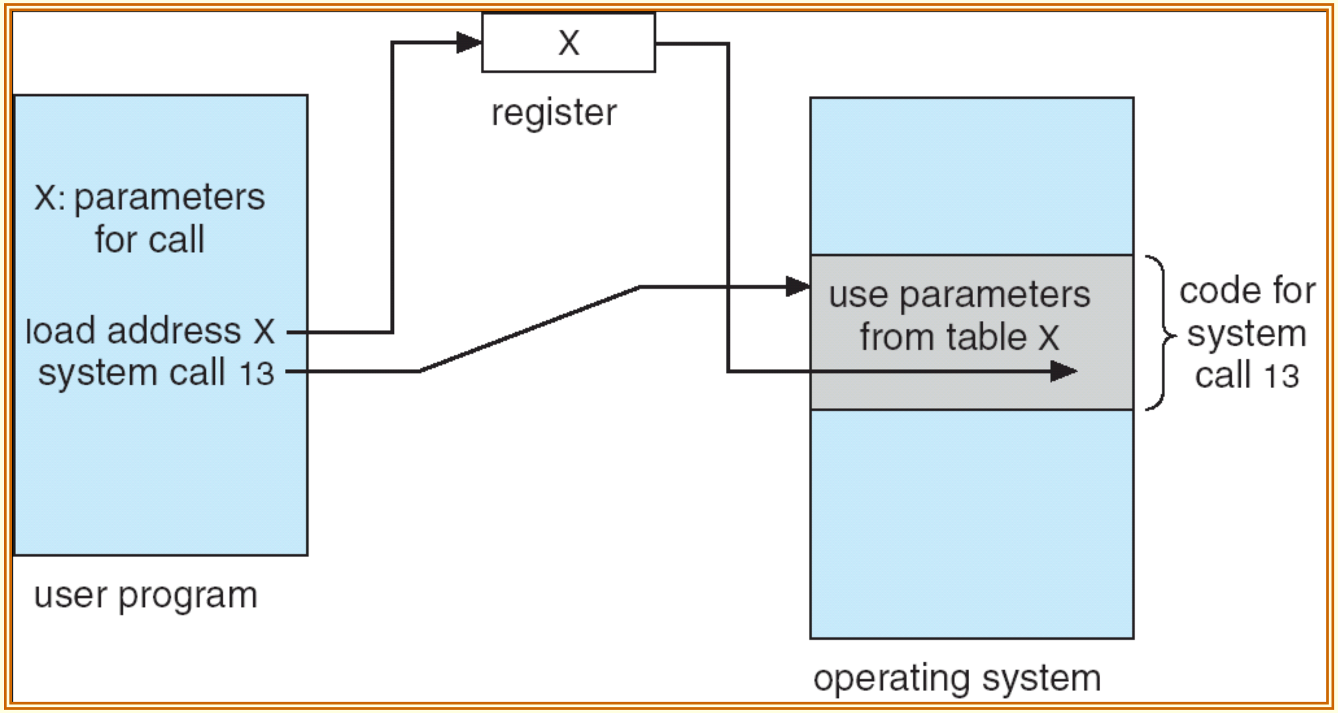
System Calls (9/11)

- With the help of API:
  - The caller need know nothing about how the system call is implemented.
  - Just needs to obey API and understand what the operating system will do as a result of that system call.
  - Details of the operating system are hidden from programmer.

System Calls (10/11)

- Often, more information is required than simply identity of desired system call
  - E.g., system call parameters.
- Three general methods used to pass parameters to the OS:
  - Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers.
  - Parameters are stored in a *block* in memory, and address of block passed as a parameter in a register.
    - This approach taken by Linux and Solaris.
  - Parameters are *pushed* onto a ***stack*** by the program and *popped* off the stack by the operating system.
  - Block and stack methods are popular because they do not limit the number or length of parameters being passed.

System Calls (11/11)



Types of System Calls (1/9)

- Many of today's operating system have hundreds of system calls.
  - Linux has 319 (or more) different system calls.

- Those system calls can be grouped roughly into five major categories:
  - Process control.
  - File management.
  - Device management.
  - Information maintenance.
  - Communications.

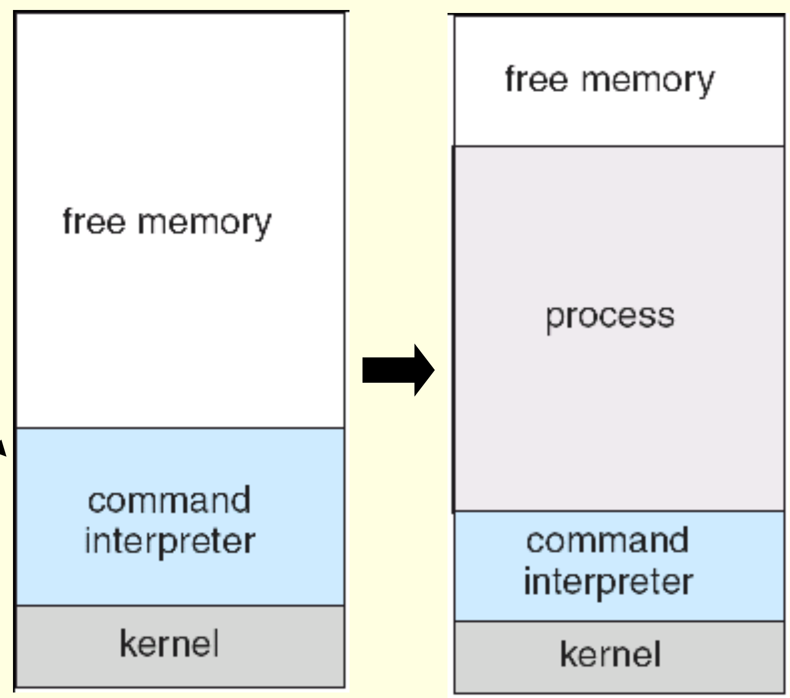
Types of System Calls — **Process Control** (2/9)

- `end` and `abort`:
  - A running program needs to be able to halt its execution either normally or abnormally.
  - The operating system then transfers control to the invoking command interpreter to read the next command.
- `load` and `execute`:
  - A process executing one program may want to load and execute another program.
  - The existing process can be lost, saved, or allowed to continue execution concurrently with the new process.
  - Chapter 6 (synchronization) discusses coordination of concurrent processes in great detail.
- `wait time/event`:
  - Having created new processes, we may need to wait for them to finish their execution.
  - We may want to wait for a certain amount of time to pass.
  - We may want to wait for a specific event to occur.

Types of System Calls — Process Control (3/9)

- Two popular variations in process control:
  - **Single-tasking system**: the MS-DOS operating system.

It has a command interpreter that is invoked when the computer started.



The command interpreter loads the program into memory and run the process.

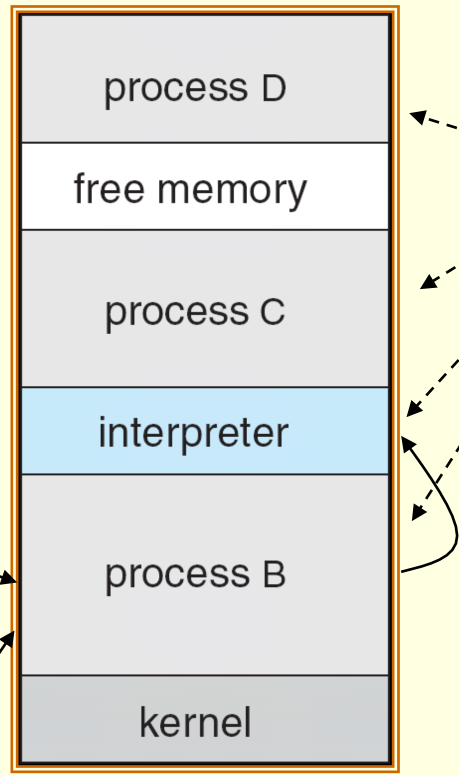
May writing over most of itself to give the process as much memory as possible.

When the process terminates, the interpreter reloads itself from disk and wait for the next user commands.

Only one process and can not create a new process

Types of System Calls — Process Control (4/9)

- **A multitasking system**: the FreeBSD (derived from Berkeley UNIX).



Multiple processes are running concurrently.

When a user logs on to the system, the **shell** is running.

To start a new process, the shell execute a `fork()` system call.

Then, the selected program is loaded into memory via `exec()` system call.

The shell can run the process in the **background** and immediately requests another command.

Chapter 3 discusses the `fork()` and `exec()` system calls

Types of System Calls — **File Management** (5/9)

- `create` and `delete`:
  - Able to create and delete files/directories.
- `open` and `close`:
  - Able to open and close a file.
- `read`, `write`, and `reposition`:
  - Able to read, write, or skipping to the end/head of the file.
- Other system calls for obtaining/setting file/directory attributes.

Types of System Calls — **Device Management** (6/9)

- The various **resources** (memory, disks, file, …) controlled by the operating system can be thought of as **devices**.

- To access a resource, a process has to:
  - First `request` the device, to ensure **exclusive** use of it.
  - Then we can `read`, `write`, and `reposition` the device.
  - After we are finished with the device, we `release` it.

  - The similarity between I/O devices and files is so great that many operating systems (UNIX) merge the two into a combined file-device structure.
    - A set of system calls is used on files and devices.
    - Sometimes, I/O devices are identified by special file names.

ex.雷射印表機的file

有些system call可以用在一個file or 一個device上

Types of System Calls — **Information Maintenance** (7/9)

- Many system calls exist simply for the purpose of transferring information between the user program and the operating system.
  - `time` and `date` return the current time and date of the system.
  - Other system calls can return the number of current users, the amount of free memory or disk space, …
  - Get and set processes attributes.

Types of System Calls — **Communication** (8/9)

- There are two common models of interprocess communication:
  - **Message-passing model**:
    - The communicating processes (may be on different computers) exchange messages with one another to transfer information.
    - Client and server (daemon) architecture.
      - Client: ask for connecting communication.
      - Server: wait for connection.
    - Require system calls to `build up/terminate` connection, `read`, and `write` messages.
  - **Shared-memory model**:
    - Processes use `shared memory create/attach` system calls to create and gain access to regions of memory owned by other processes.
    - They can then exchange information by reading and writing data in the shared areas.

Types of System Calls — Communication (9/9)

- Message passing:
  - Is useful for exchanging smaller amounts of data, because no conflicts need be avoided.
  - Is easy to implement.

- Shared memory:
  - Allows maximum speed of communication, since it can be done at memory speeds when it takes place within a computer.
  - However, problems exist in the areas of protection and synchronization between the processes sharing memory.

System Programs (1/3)

- A perspective of operating systems is a collection of **system programs**.
  - System programs provide a convenient environment for program execution and development.
  - Most users' view of the operation system is defined by system programs, not the actual system calls.
  - **Some of them are just user interfaces to system calls!!**
- Categories of system programs:
  - **File manipulation**:
    - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

System Programs (2/3)

- **File modification**:
  - Text editors to create and modify files.
- **Status information**:
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Typically, these programs format and print the output to the terminal or other output devices
- **Programming-language support**:
  - Compilers, assemblers, debuggers and interpreters sometimes provided.
- **Program loading and execution:**
  - Loaders to load assembled or compiled programs into memory for execution.

System Programs (3/3)

- **Communications:**
  - Provide the mechanism for creating virtual connections among processes, users, and computer systems.
- In addition to system programs, application programs are supplied to solve common problems or perform common operations.
  - Web browsers, word processors, database systems, games …
- The view of the operating system seen by most users is defined by the application and system programs, rather than the actual system calls.

Operating System Design (1/2)

- The first is to define **requirements** and **specifications**.
- However, there is no unique solution to the problem of defining the requirements for an operating system.
  - Requirements can be affected by choice of hardware, type of system.
    - Handheld devices vs. PCs.
    - Single process vs. multitasking.
- The requirement can be divided into *user goals* and *system 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.

Operating System Design (2/2)

- One important principle of system design is **the separation of *policy* from *mechanism***.
  - **Policy: What** will be done?
  - **Mechanism: How** to do it?
  - Example: *timer* is a **mechanism** for ensuring CPU protection, but deciding how long the timer is to be se is a a **policy** decision.
- Flexibility of the separation:
  - Policies are likely to change across places or over time.
  - The separation enables a change in policy to redefine certain policy parameters rather than changing the underlying mechanism.
- Most of Windows services mix mechanisms with policies to enforce a global look and feel.

Operating System Implementation (1/2)

- Traditionally, operating systems have been written in assembly language.
- Now, they are most written in higher-level languages such as C or C++.
  - The code can be **written faster** and is **easier to understand** and **debug**.
  - System is **easier to port** (to move to some other hardware).
    - The Linux operating system is written mostly in C and is available on a number of different CPUs, including Intel 80x86, Motorola 680X0, …
- Previous comments on higher-level languages: reduced speed and increased storage requirements.
  - Modern compiler techniques can perform complex analysis and optimizations that produce excellent code.

Operating System Implementation (2/2)

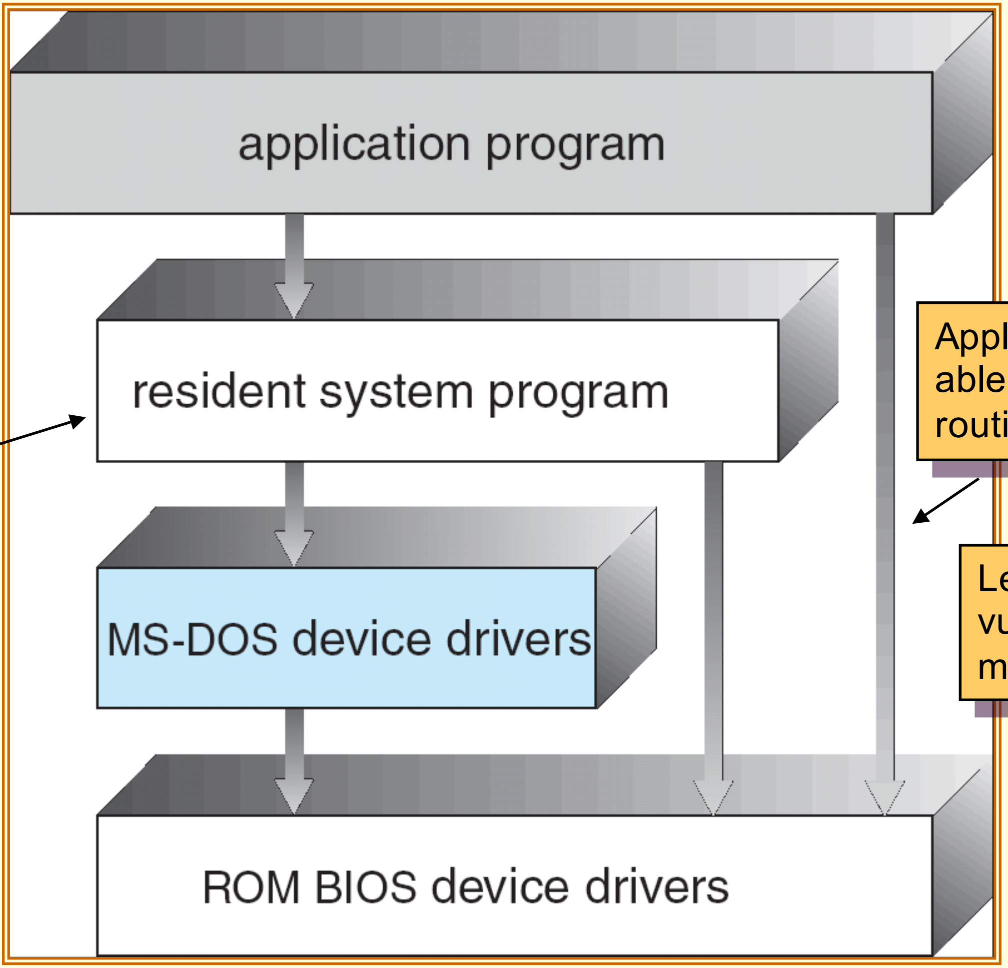
- Moreover, major performance improvements in operating systems (and other systems) are more likely to be the result of better **data structures** and **algorithms** than of excellent assembly-language code.

- Should pay more attentions on the **memory manager** and the **CPU scheduler**.
  - They are probably the most critical routines.

Operating-System Structure
Simple Structure (1/11)

- A common approach to implement an operating system is to partition the task into small components.
  - **Rather than have one monolithic system!!**
  - These components are interconnected and meld into a kernel.
  - But…
- **Simple structure**:
  - Many commercial system do not have well-defined structures initially.
    - Started as small, simple, and limited systems and then grew beyond their original scope.
    - For example, MS-DOS.

Operating-System Structure Simple Structure (2/11)



The interfaces and levels of functionality are not well separated.

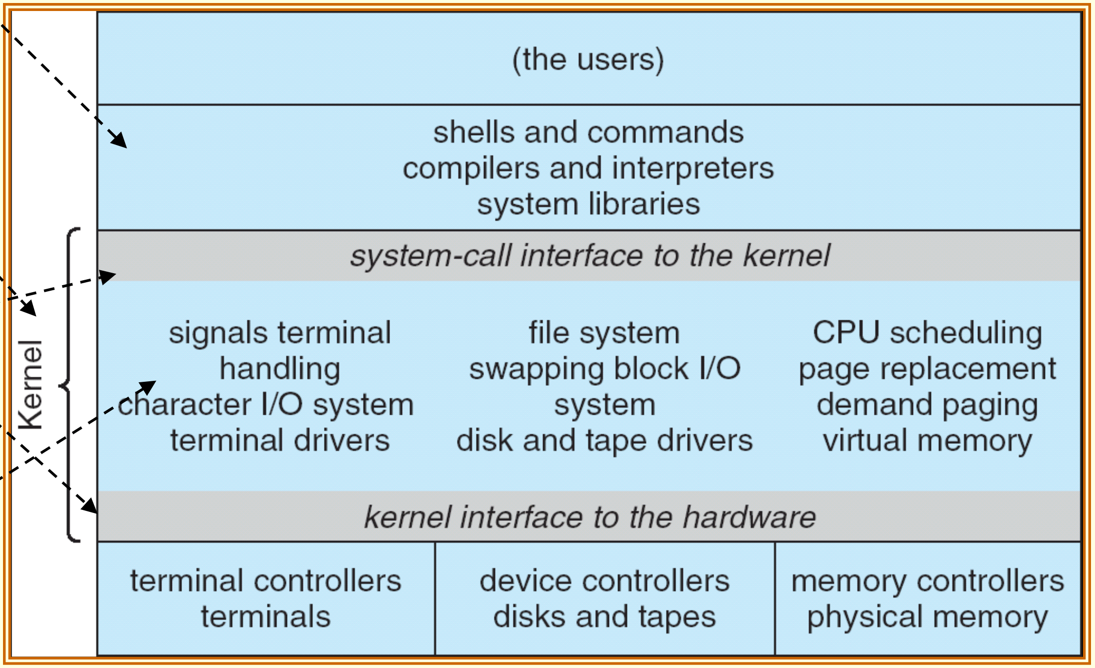
Application programs are able to access the basic I/O routines to access devices

Leave MS-DOS vulnerable to malicious programs

MS-DOS layer structure.

Operating-System Structure Simple Structure (3/11)

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.
- The UNIX OS consists of two separable parts:
  - **System programs**.
  - **The kernel**.



Interfaces to the bare hardware and system (application) programs

The kernel provides a lot of services, combined into **one monolithic level**.

Very difficult to implement and maintain.

Operating-System Structure
Layered Approach (4/11)

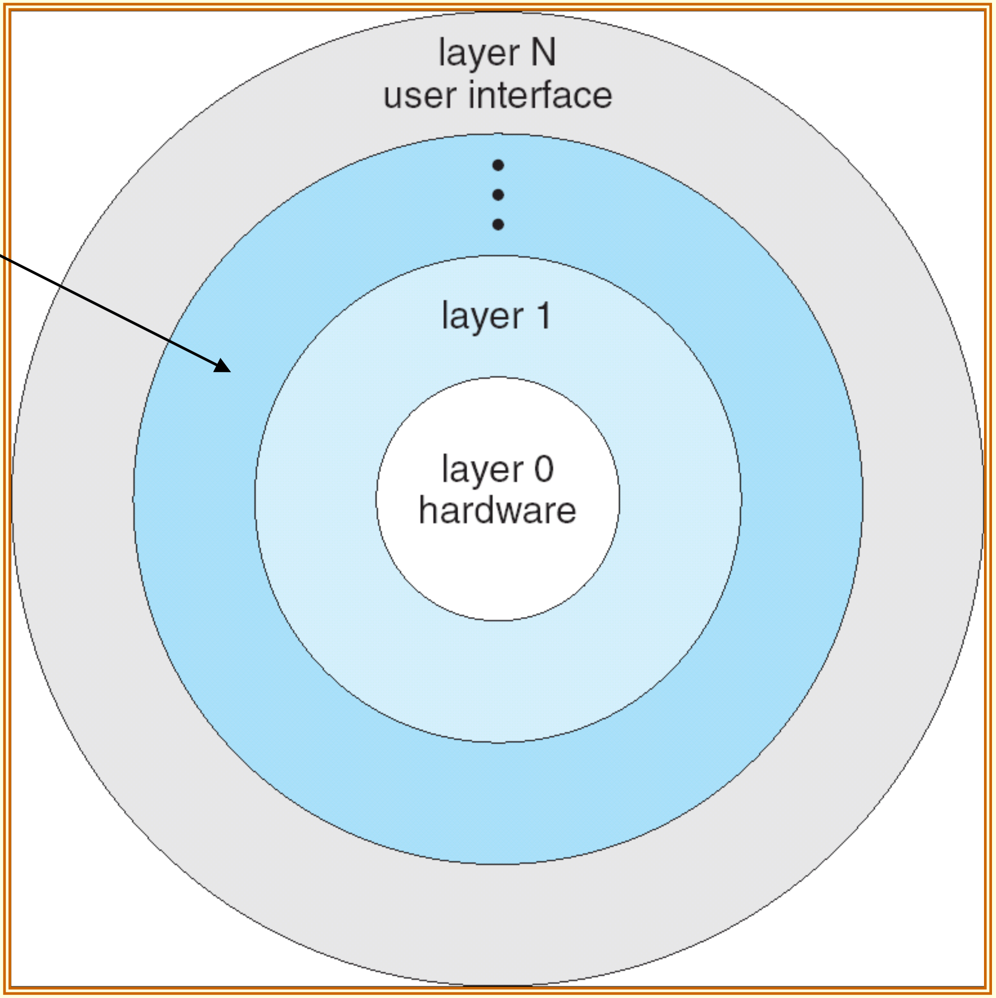
- With the improvements of hardware and programming techniques, operating systems can be broken into pieces of components.
  - That is **modular** operating systems.
    - Information hiding: hide the internal implementation detail of modules and provide external access **interfaces**.

- One way of modular system: **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.

Operating-System Structure Layered Approach (5/11)

Layer *M* consists of data structures and a set of routines that can be invoked by higher-level layer**s**.

Layer *M*, in turn, can invoke operations on lower-level layer**s**.



Operating-System Structure Layered Approach (6/11)

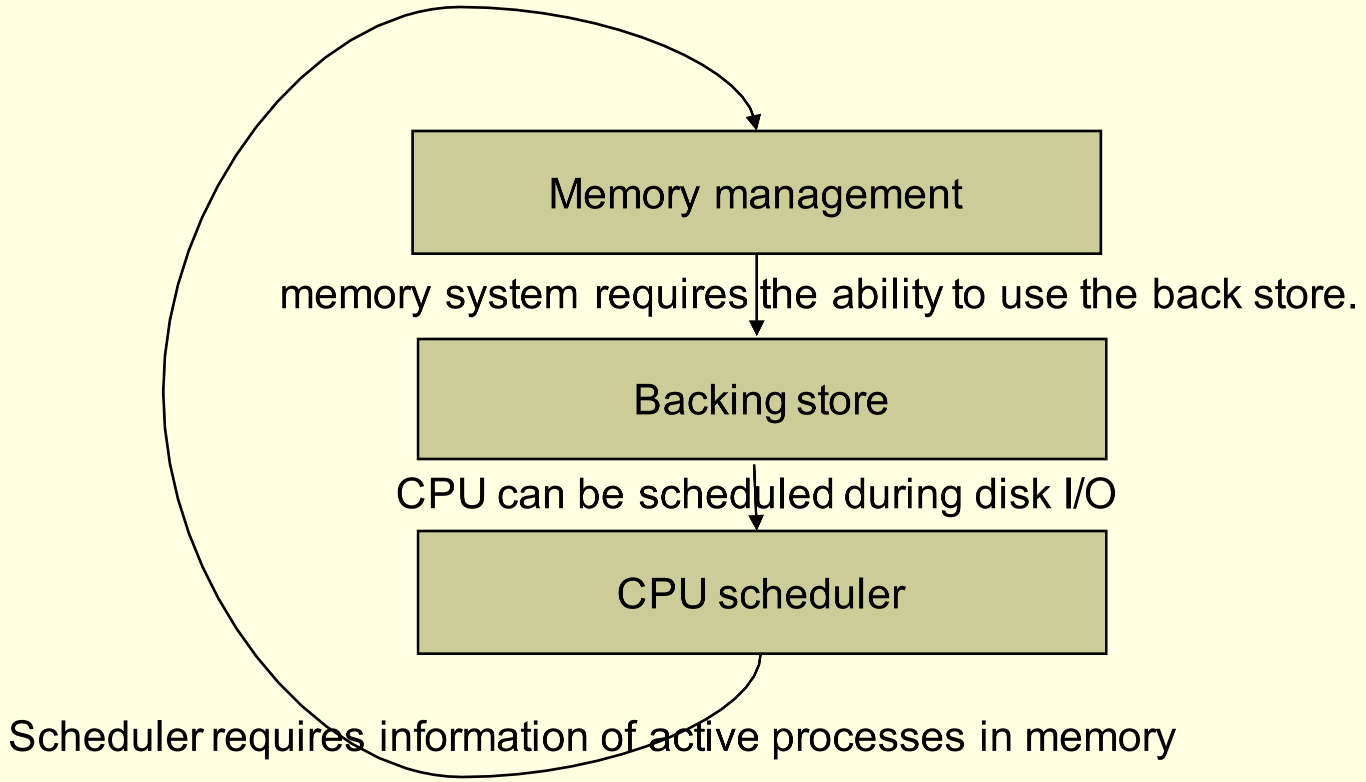
- The main **advantage** of the layered approach:
  - Simplicity of construction and **debugging**.
  - Layer-by-layer debugging, starting from layer 0.
  - If an error is found during the debugging of a particular layer, the error must be on that layer.

- The major **difficulty** of the layered approach:
  - Because only lower layers operations can be invoked, appropriately defining the various layers is difficult.
    - **System services usually tangle together.**
  - Layered implementation tend to be less efficient.
    - A function call on the top layer can lead to many lower-layer calls.
    - Function calls need to pass (redundant) parameters.

- Recently, fewer layers with more functionality are being designed.
  - Providing the advantages of modularization. 而且慢
  - Avoiding the difficulties of layer definition and interaction.

Operating-System Structure Layered Approach (7/11)

- Example of tangled layers:



Operating-System Structure
Microkernels (8/11)

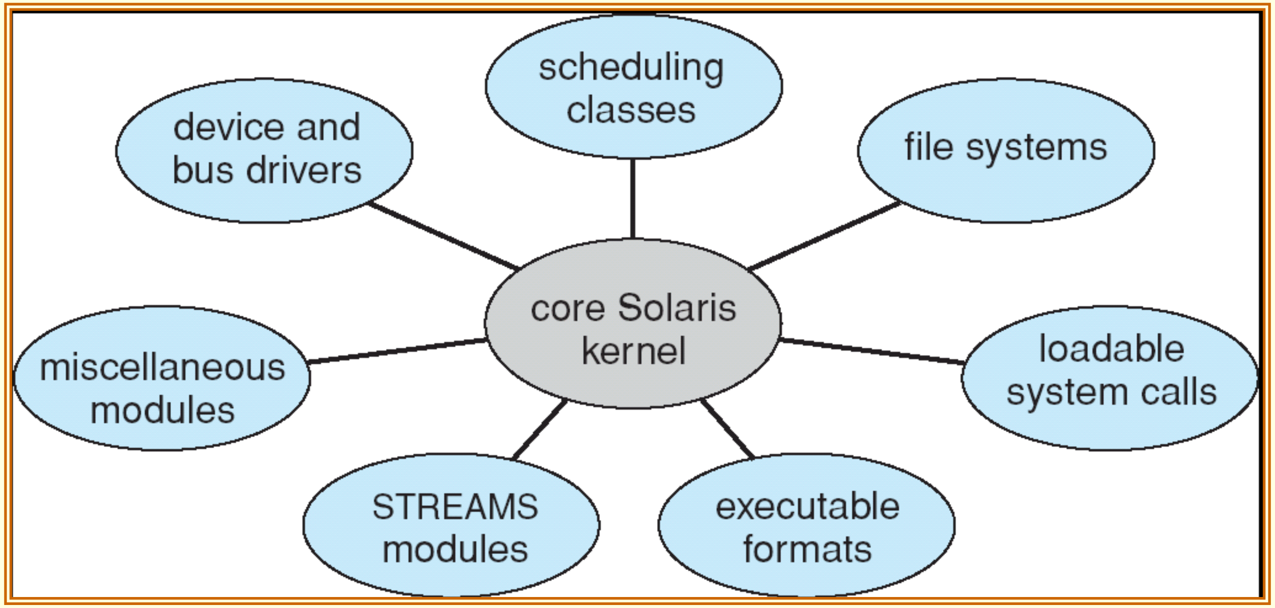
- As operating systems expanded, the kernel became large and difficult to manage.
- In the mid-1980s, CMU developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach.
  - Micro → removing all nonessential components from the kernel and implementing them as system and user-level programs (servers).
  - Typically, microkernels provide **process** and **memory** management, and a **communication** facility.
  - The client program and services communicate indirectly by exchanging message with the microkernel.

Operating-System Structure Microkernels (9/11)

- **Benefits**:
  - Easier to include new operating system services to a microkernel.
    - Do not require modification of the kernel.
  - The small kernel makes it easier to port to new hardware architectures.
  - More reliable and secure (less code is running in kernel mode).
- **Problems**: 慢到不行（kernel->變到很多地方）
  - Performance overhead of user space to kernel space communication.
  - Initial Windows NT (a micorkernel organization) → Windows NT 4.0 (moving layers from user space to kernel space).

Operating-System Structure
Modules (10/11)

- A better methodology for operating-system design involves using object-orient programming techniques to create a **modular kernel**.
  - Consists of a **core kernel**, and system service as **kernel modules**.
    - Each module talks to the others over known **interfaces**



Operating-System Structure Modules (11/11)

- Moreover, modules (system services) can be linked into the system either during boot time or **during run time** (that is, loaded as needed within the kernel).
  dynamic linking
  - Load different file system (ext2fs, FAT32 or NTFS) as needed, to save main memory.
- The module structure is similar to layered (communicate with interfaces) and microkernel approaches (a core), but with more flexible.
  - Any module can call any other module, but the layered approach can not.
  - Is efficient than microkernel approach because modules are in the kernel space and do not need to invoke message passing to communicate.
- The strategy of dynamically loadable modules is very popular in modern UNIX-based operating systems, such as Linux.

Operating System Generation (1/3)

- Operating systems are normally distributed on disk or CD-ROM.
  - They are designed to run on **any** class of machines **with different hardware configurations**.
- To generate a system for each specific computer site, a special program – **`SYSGEM`** – is needed.
  - It determines computer components by:
    - Reading a given file.
    - Asking the operator of the system for hardware information.
    - Probes the hardware directly.

Operating System Generation (2/3)

- The information must be determined:
  - **CPU**:
    - What CPU is to be used?
    - Number of CPUs.
    - Has extended instruction sets or floating point arithmetic.
  - **Memory**:
    - Size.
- **Devices**:
  - Type and model.
  - Interrupt number.
- **Operating-system options**:
  - Maximum number of processes to be supported.
  - CPU-scheduling algorithm.

Operating System Generation (3/3)

- Once the information is determined …
  - Source code of the operating system can be **modified** and completely **compiled** to produce a tailored operating system.
    - System generation is slower.
    - But more specific to the underlying hardware.
  - Or, the description can cause the selection of **modules** from a **precompiled library**, which are linked together to form the operating system.
    - Because the system is not recompiled, system generation is faster.
    - The resulting system may be general.
    - Easy to modify the generated system as the hardware configuration changes (such as, add a new hardware).

System Boot (1/2)

- The generated operating system must be made available by the hardware.
- How does the hardware know where the kernel is and how to load that kernel??
- ***Booting*** – the procedure of starting a computer by loading the kernel.
  - Power up or reset.
- Need a ***bootstrap program*** to*:*
  - Locate the kernel on the disk.
  - Load it into memory.
  - Start its execution.
  - A simple code stored in ROM or EPROM.
    - At a fixed location so that can be loaded and executed when computer is on.
  - But before that, it first initializes all aspects of the system:
    - CPU registers, device controller, the contents of main memory …

System Boot (2/2)

- Some computer systems (such as PCs) use a two-step booting process:
  - A simple bootstrap loader fetches a more complex boot program from disk.
  - Which in turn loads the kernel.
- The boot program stored in the ***boot block*** (a fixed location on disk) is usually sophisticated and modifiable and is to load an (or different) operating system into memory and begin its execution.
  - Then the operating system is said to be **running**.

End of Chapter 2