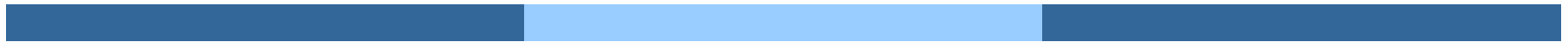


Chapter 2: 系統結構 (System Structures)



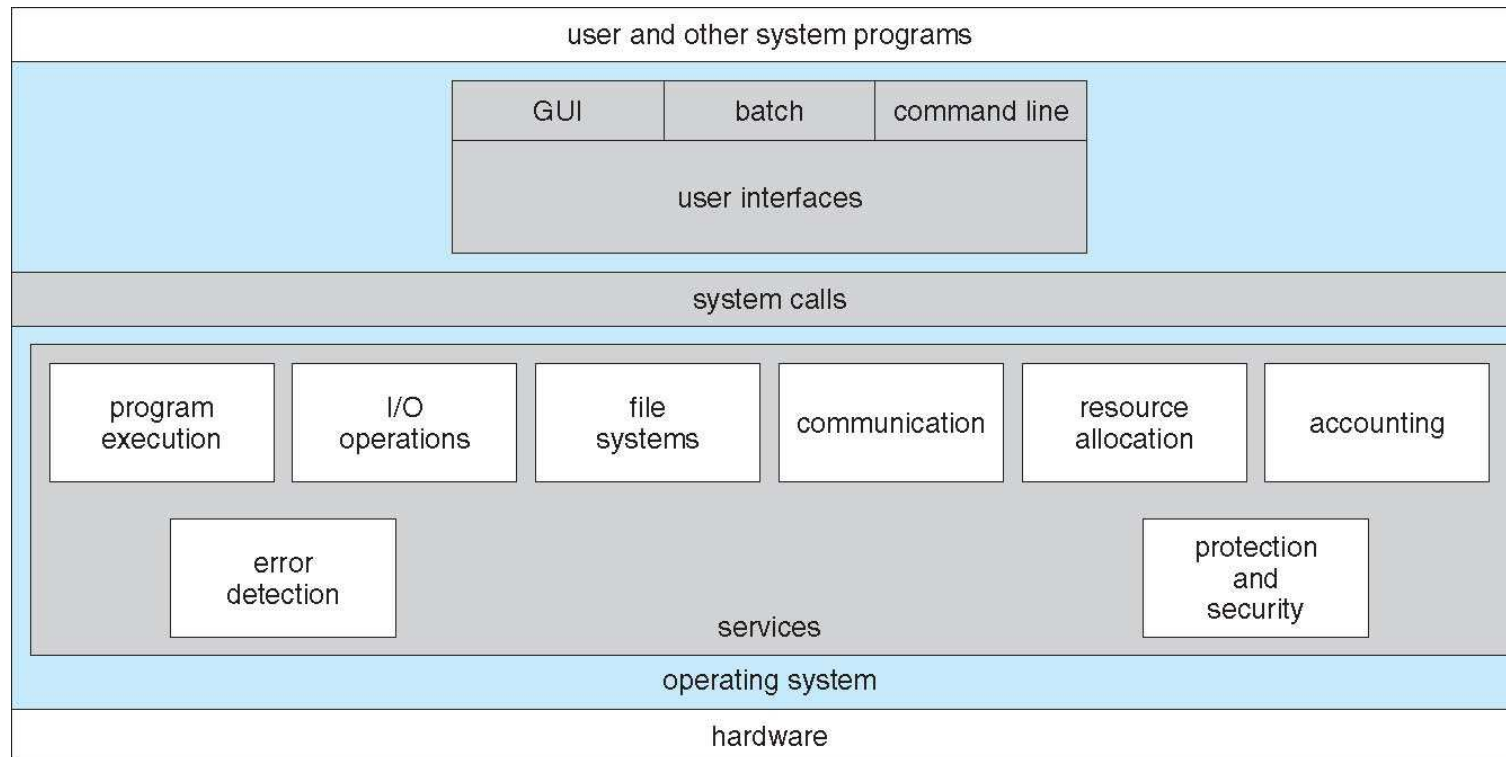
Chapter 2: System Structures

- 作業系統服務(Operating System Services)
- 使用者與作業系統介面(User and Operating System Interface)
- 系統呼叫(System Calls)
- 系統呼叫的類型(Types of System Calls)
- 系統程式(System Programs)
- 系統的設計和製作(Operating System Design and Implementation)
- 作業系統結構(Operating System Structure)
- 作業系統除錯(Operating System Debugging)
- 作業系統生成(Operating System Generation)
- 系統啟動(System Boot)

章節目標

- 描述作業系統提供那些服務給使用者、行程及其它系統
- 討論建構作業系統的不同方式
- 說明作業系統是如何安裝、客製化和如何啟動

作業系統提供的服務及關係



作業系統服務(Operating System Services)

- 作業系統提供執行程式的環境，並為程式與使用者提供服務
- 對使用者有幫助的服務有：
 - 使用者介面 (User Interface,UI)，種類有：
 - ▶ 命令行介面(Command-Line Interface，CLI)
 - ▶ 圖形使用者介面(Graphics User Interface，GUI)
 - ▶ 批次(Batch)
 - 程式執行 (Process Execution)
 - I/O處理(I/O operations)
 - 檔案系統操作 (File-system Manipulation)
 - 通信(Communications)：電腦內或電腦間不同行程的資訊交換
 - ▶ 可藉由共用記憶體(shared memory)或藉由訊息傳遞(message passing)技術達成
 - 錯誤偵測 (Error Detection)：需不斷地偵測可能的軟硬體錯誤，對於任何一類錯誤，都應該採取適當的行動以確保系統正常的運作。

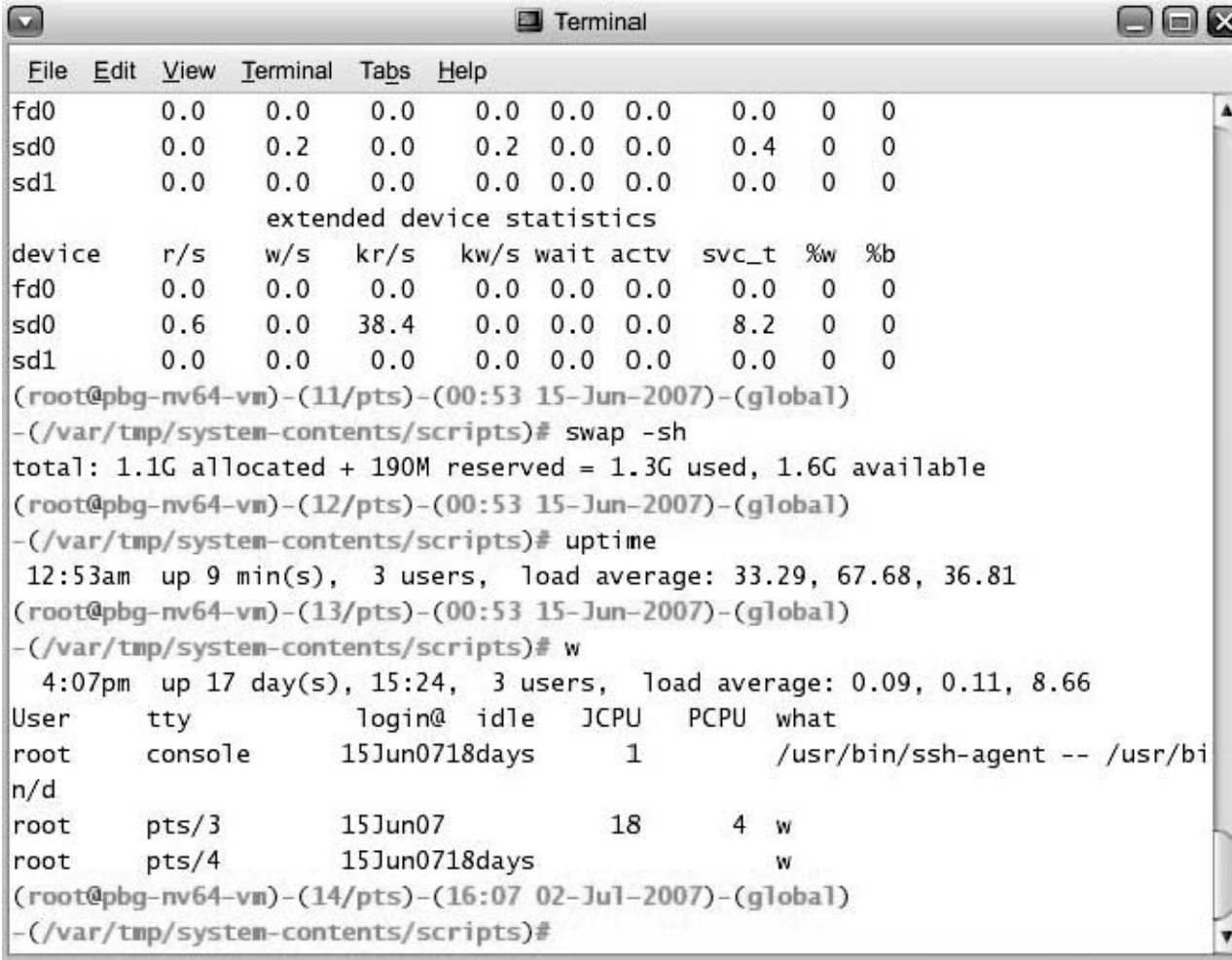
作業系統服務(Operating System Services)(續)

- 為確保系統本身有效運作而提供的服務有：
 - 資源分配 (Resource Allocation)：當有許多使用者或許多工作同時執行的時候，必須將電腦資源分配給他們。
 - 會計核算 (Accounting)：追蹤使用者或程式使用了多少資源及哪些資源。這些資料提供會計核算或統計之用途。統計資料對改善計算服務是很有用的資訊。
 - 保護和安全(Protection and security)

使用者與作業系統介面-CLI Interfaces

- 命令直譯程式(command interpreter)允許直接輸入命令並執行它
 - 可將CLI包含在核心內，亦可將CLI製作成系統程式
 - 有許多不同風格的殼程式(shells)
 - ▶ 如：Bourne shell, C shell, Bourne-Again shell, Korn shell, etc.
 - 支援的命令可分成兩大類
 - ▶ 內建命令(Built-in commands)
 - 包含於殼程式內，加入新命令須修改shell
 - ▶ 外部命令(External commands)
 - 以個別執行檔的形式存在，加入新功能時不需要修改shell

Bourne Shell Command Interpreter



```
File Edit View Terminal Tabs Help
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-vn)-(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-vn)-(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-vn)-(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-vn)-(14/pts)-(16:07 02-Jul-2007)-(global)
-(/var/tmp/system-contents/scripts)#
```


使用者與作業系統介面- GUI Interfaces

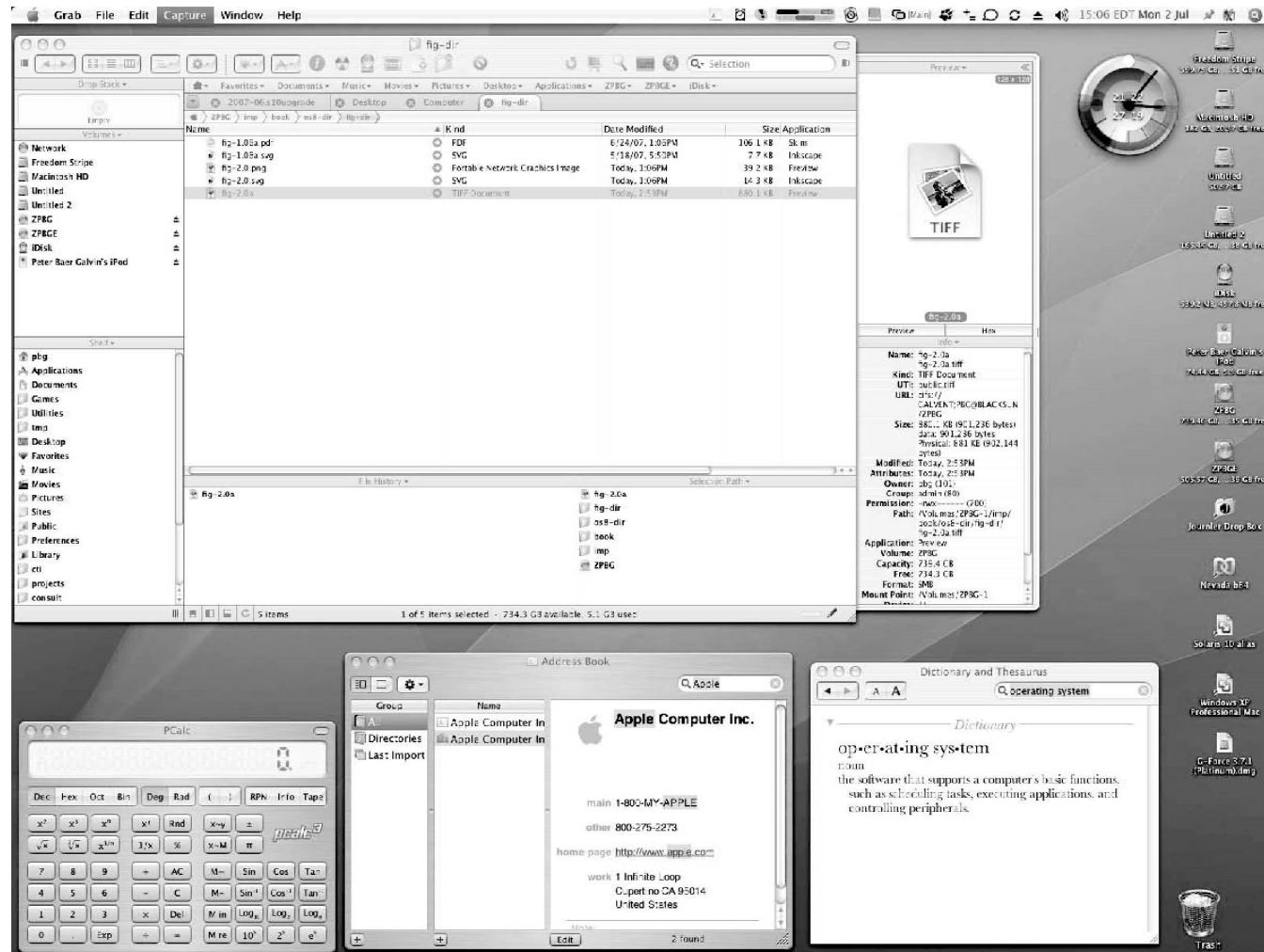
■ 使用者友善的桌面介面

- 通常是使用滑鼠、鍵盤和螢幕
- 使用 **圖像(Icons)**代表程式、檔案和動作
- 根據滑鼠指標的位置，按下滑鼠的不同按鍵，可以產生不同的動作，例如提供資訊、執行某些功能、或開啟目錄(又稱為 **folder**)
- 由Xerox PARC 研發出來

■ 目前許多系統同時提供CLI和GUI介面

- Microsoft Windows 是包含有CLI 的 GUI
- Apple Mac OS X 是使用“Aqua” GUI 介面，但在其底層則是UNIX 的核心和shell
- Unix 和 Linux是以CLI為主，但有不同的GUI介面可以選用 (CDE, KDE, GNOME)

The Mac OS X GUI



觸控介面(Touchscreen Interfaces)

- 可不用滑鼠
- 根據手勢來控制或做選擇
- 利用虛擬鍵盤來輸入文字



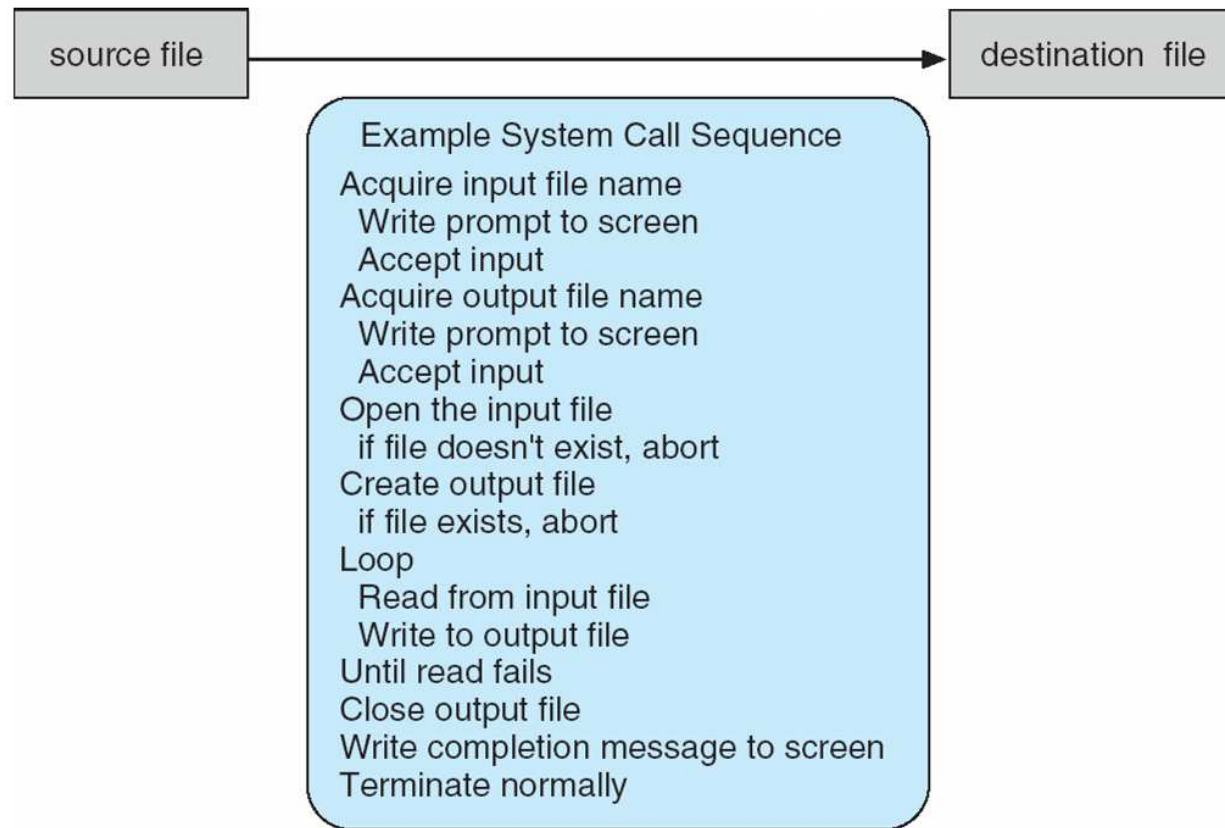
iPad觸控介面

系統呼叫(System Calls)

- 系統呼叫是請求OS提供服務的程式介面(programming interface)
- 系統呼叫通常以高階語言撰寫 (C or C++)，僅某些低階工作以組合語言撰寫
- 大部分程式是使用高層次的應用程式介面(Application Program Interface, API) 來要求OS服務，而非直接使用系統呼叫
 - 三種最常使用的API
 - ▶ Windows的Win32 API
 - ▶ POSIX(Portable Operating System Interface) API(包括幾乎所有版本的 UNIX、Linux、和Mac OS X)
 - ▶ Java API
 - 程式開發使用APIs 而不是直接使用系統呼叫的優點有
 - ▶ 程式移植性(Portability)高
 - ▶ 較容易使用
 - 透過執行階段支援程式庫(run-time support library)所提供的系統呼叫介面(system-call interface)，將API內的函數呼叫連結到OS的系統呼叫。

系統呼叫範例

- 拷貝一個檔案的內容到另一檔案的程式其可能用到的系統呼叫



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)
```

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

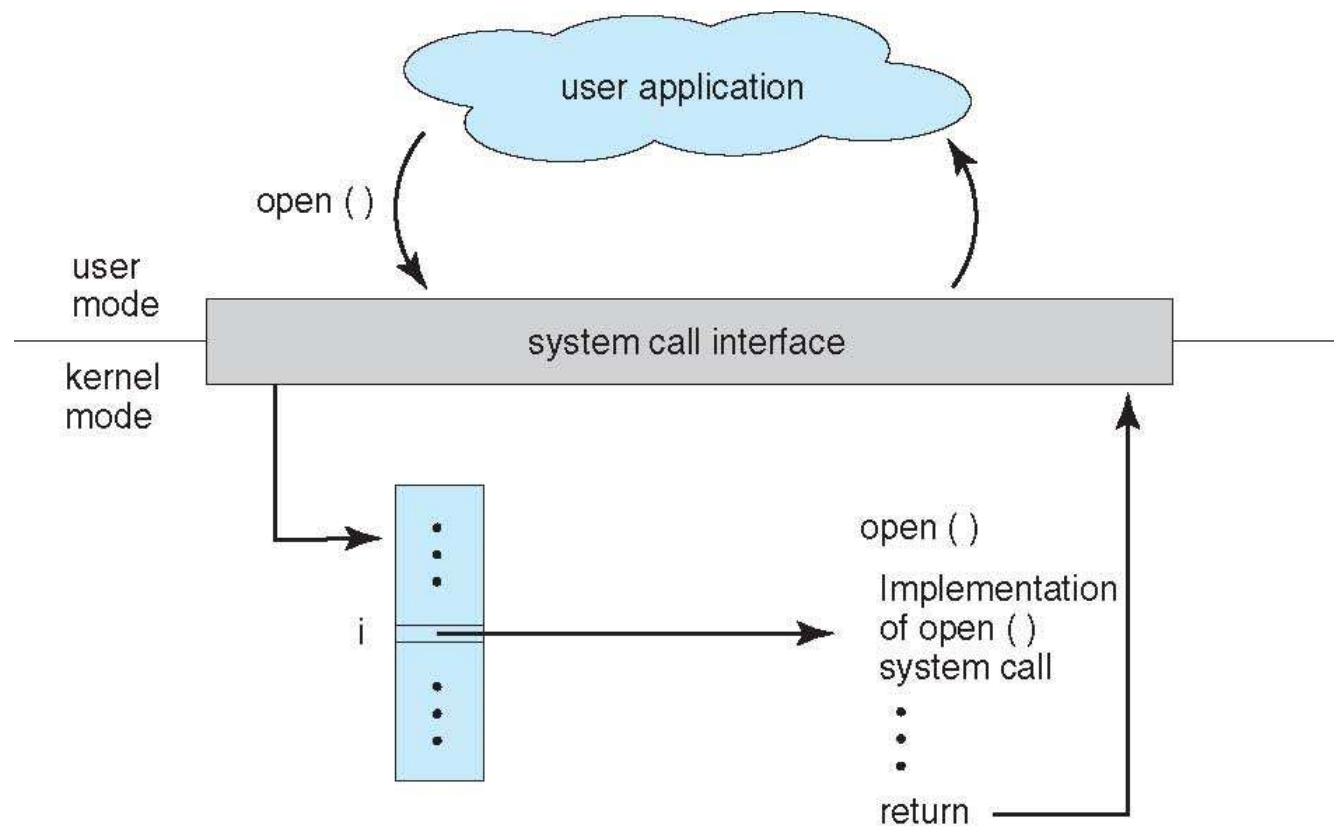
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

系統呼叫內部做法

- 每個系統呼叫都有一個唯一編號，系統呼叫介面使用一個表格來進行編號與實際核心內系統呼叫程式碼的連結。
- 系統呼叫介面呼喚OS核心內指定的系統呼叫，並傳回系統呼叫的狀態值，和任何傳回值。
- 透過API的方式，呼叫者不需要知道系統呼叫的實際做法
 - 呼叫者只要遵循API用法，並知道該呼叫OS處理的結果即可
 - 藉由執行階段支援程式庫(run-time support library)的處理，程式設計者可以完全不管系統呼叫介面的處理細節

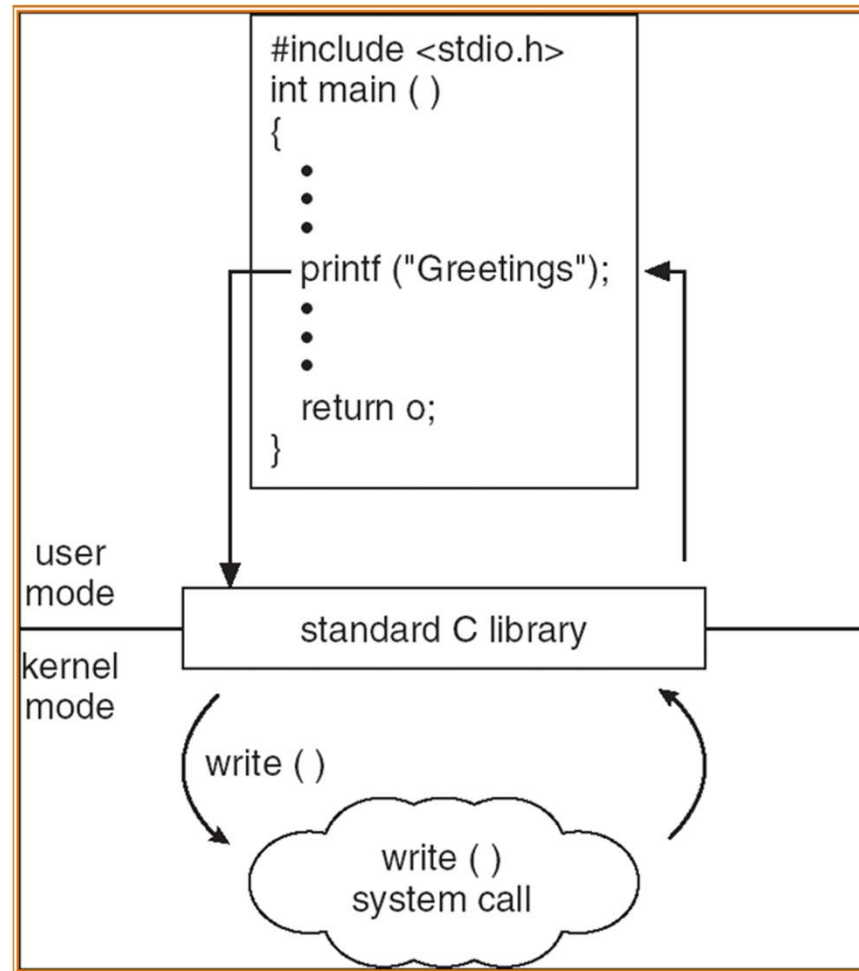
API – System Call – OS 間的關係

- 當程式呼叫了open()系統呼叫時，OS的處理情形。



標準C程式庫的範例

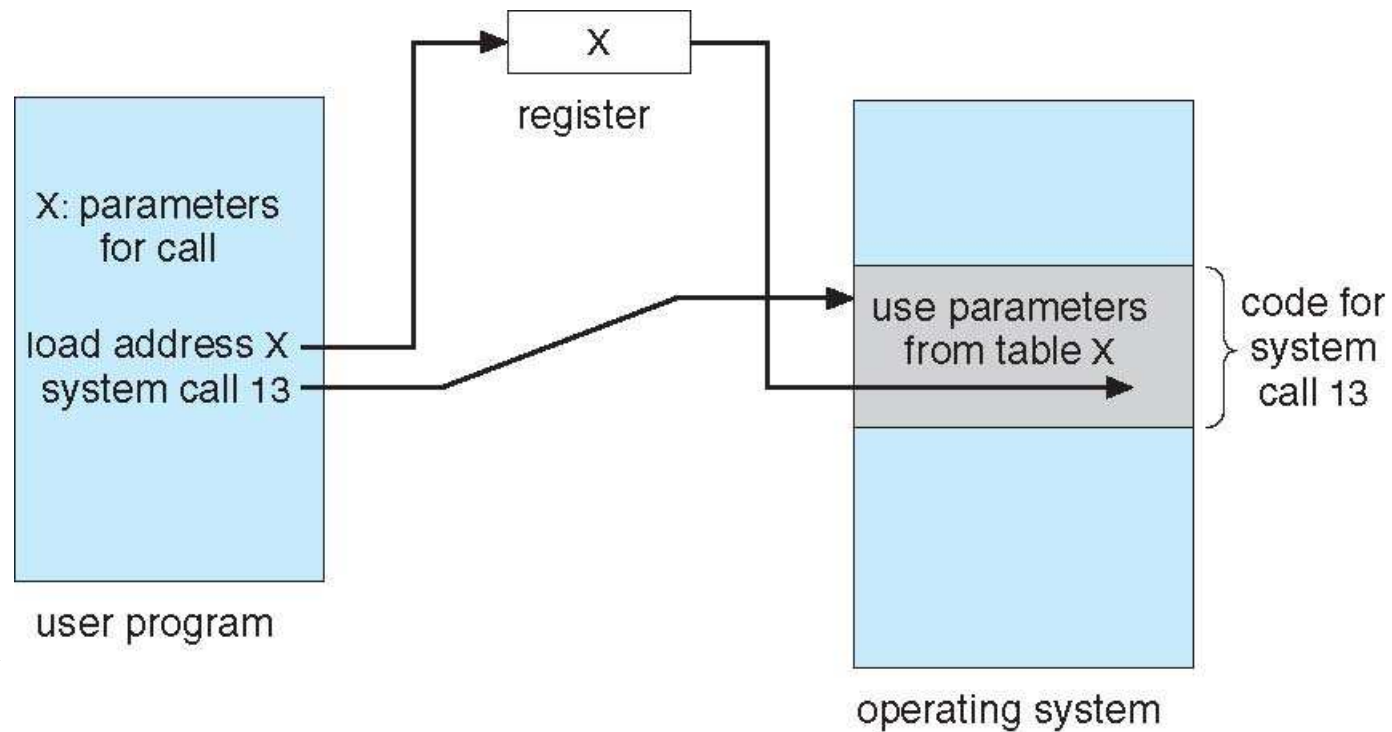
- C程式呼叫printf()，它再呼叫write()系統呼叫的處理情形。



系統呼叫參數的傳遞

- 系統呼叫時有時候需要更多的資訊，資訊的類型與數量則根據作業系統和呼叫的特性而定。這些需要的資訊，便須透過一些方法來傳給OS。
- 傳遞參數(parameters)給作業系統之方法有三種：
 - 透過暫存器 (Registers) 來傳遞
 - ▶ 簡單但會有數量的限制
 - 將參數放在表格 (Table)內，然後把表格位址透過一個暫存器來傳遞
 - ▶ 沒有數量限制
 - ▶ Linux和Solaris就是採用這種方法
 - 將參數存到堆疊 (Stack) 中，作業系統再從堆疊取出來
 - ▶ 沒有數量限制

經由表格傳遞參數



系統呼叫之種類

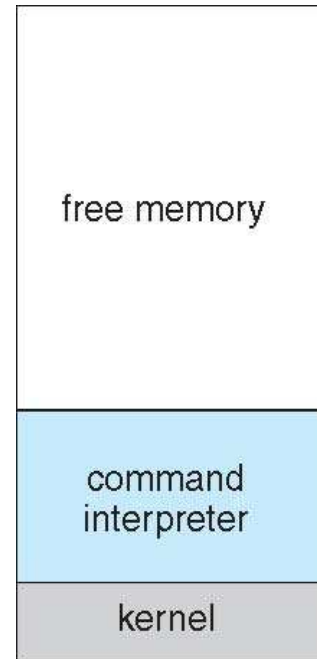
- 行程控制 (Process Control)
- 檔案管理 (File Management)
- 裝置管理 (Device Management)
- 資訊管理 (Information Maintenance)
- 通訊 (Communications)
- 保護 (Protection)

系統呼叫-行程控制

- 正常結束執行(end)，中止執行(abort)
- 載入(load)及執行(execute)其他程式
- 建立行程(create process)，終止行程(terminate process)
- 取得行程屬性(get attributes)，設定行程屬性(set attributes)
 - 如：優先權 (Priority)、執行時間
- 等待時間要求(wait time)
- 等待事件發生(wait event)，發送事件 (signal event)
- 配置及釋放記憶體空間

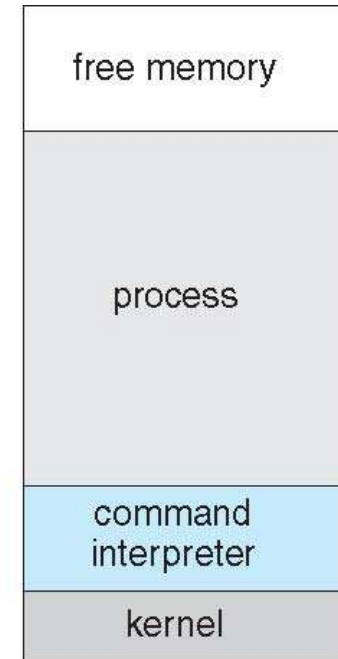
Example: MS-DOS

- 單工(single-tasking)
- 當系統啟動時
shell(command interpreter)被載入
- 不能在一個程式中建立另一個行程
- 記憶體空間僅給一個程式使用
- 載入程式到記憶體時，會覆蓋掉核心外的所有內容
- 程式結束時，shell 被重新載入



(a)

MSDOS系統啟動
後記憶體的使用
情形

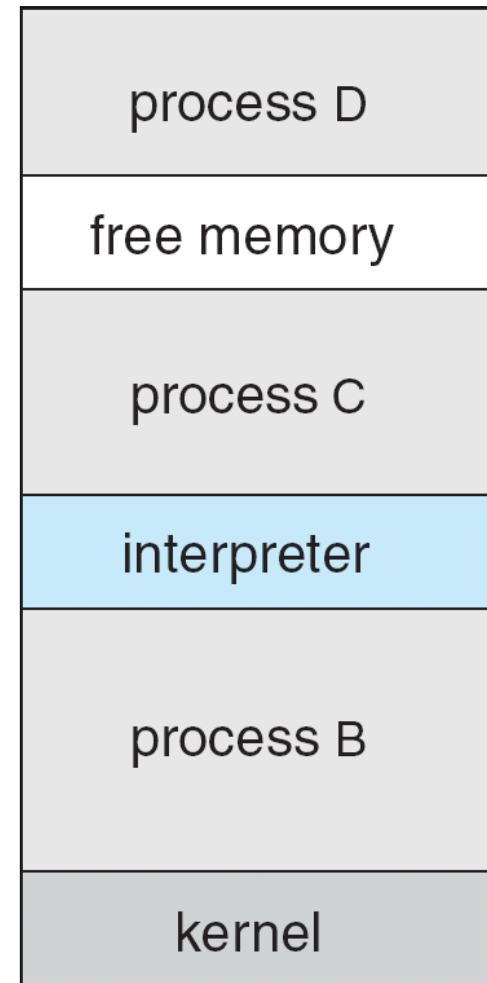


(b)

當正在執行程式
時記憶體的使用
情形

Example: FreeBSD

- 自 Unix 改版而來
- 是一個多工系統(multitasking system)
- 當使用者登入後，啟動使用者選擇的shell
- Shell透過fork()系統呼叫以產生新行程，並透過exec()系統呼叫來載入程式並開始執行。
- Shell 等待行程結束或將行程置於背景執行
- 若將行程置於背景執行，則shell可繼續接受使用者輸入的命令。
- 當行程結束時，返回碼若是 0，代表沒有錯誤。否則，代表錯誤碼。



FreeBSD同時執行多個程式時記憶體的使用情形

系統呼叫-檔案管理

- 建立(create)檔案、刪除(delete)檔案
- 開啟(open)、關閉檔案(close)
- 讀出(read)、寫入(write)、重定讀寫位置(reposition)
- 讀取檔案屬性，設定檔案屬性

系統呼叫-裝置管理

- 請求(request)裝置，釋回(release)裝置
- 讀出、寫入、重定讀寫位置
- 讀取裝置屬性、設定裝置屬性
- 邏輯上地連接(attach)或解除(detach)裝置

系統呼叫-資訊管理

- 取得時間或日期，設定時間或日期
- 取得系統資料，設定系統資料
- 取得行程、檔案或裝置的屬性
- 設定行程、檔案或裝置的屬性

系統呼叫-通信

- 行程間通訊(Interprocess Communication) 有兩種模式
 - 訊息傳遞模式(Message-passing Model)
 - ▶ 行程可透過直接或間接共用信箱(mailbox) 來進行資訊交換。
 - ▶ 適合較小量的資料交換。
 - ▶ 較容易實作
 - 分享記憶體模式(Shared-memory Model)
 - ▶ 一個行程是利用映射記憶體 (Map Memory) 的系統呼叫方式來跟另一個行程通訊。
 - ▶ 通訊的行程藉由讀或寫分享記憶體來進行資訊交換。
 - ▶ 傳輸速度快。
- 應支援的系統呼叫有：
 - 建立、刪除通信連接通道(connection)
 - 傳送、接收訊息
 - 傳輸狀態訊息
 - 連接(attach)或解除連接(detach)遠端裝置

系統呼叫-保護

- 資源存取的控制(Control access to resources)
- 獲得或設定允許權(get/set permissions)
- 允許和拒絕使用者的存取(allow/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() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System Call Table References

■ Linux System Call Table

- 64-bit: <http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64>
- 32-bit: <http://syscalls.kernelgrok.com/>
- Example: hello.s

■ Windows System Call Table

- <http://j00ru.vexillium.org/ntapi/>

系統程式(System Programs)分類

- 系統程式亦稱系統工具(system utilities)提供一個便利程式開發與執行的環境，可分成：
 - 檔案管理(File manipulation)
 - 狀態資訊(Status information)
 - 檔案的修改(File modification)
 - 程式語言支援(Programming-language Support)：如compiler、assembler、debugger、interpreter等。
 - 程式的載入與執行：如Loader、linkage editor、debugging system等
 - 通信：提供行程、使用者及電腦間建立連繫的機制
 - 背景服務或後台服務(Background services)
 - ▶ 一種在不需用戶干預的情況下執行於作業系統後台的行程稱為背景服務或子系統(subsystem)或daemon
 - ▶ 有些是為了系統的啟動，而後就結束了
 - ▶ 有些從系統啟動後一直執行，直到關機
 - ▶ 通常用於執行如提供了系統監測、行程排班，錯誤登入，和列印服務等任務。

系統程式(System Programs)分類(續)

- 應用程式(Application Programs)
 - 針對使用者的某種特殊應用目的所撰寫的軟體，例如，文字處理器，表格，會計應用，瀏覽器，媒體播放器，航空飛行模擬器，命令列遊戲，圖像編輯器等。
 - 應用軟體可能與電腦及其系統軟體相捆綁，也可以被分開發布，並且可能以私有、開源或通用專案的形式編寫。
- 大部分使用者對於作業系統的了解是藉由系統程式與應用程式所提供的功能而得，並非實際的系統呼叫。

作業系統的設計和實現 (Operating System Design and Implementation)

- 作業系統的設計與實現沒有一個統一完備的做法，但有些方法已經證明是成功的
- 由定義目標和規格開始
 - 會受到硬體和系統型態所影響
 - ▶ Batch, time sharing, single user, multiuser, distributed, real time,...
 - 確認使用者目標和系統目標
 - ▶ 使用者目標—作業系統應該使用方便、容易學習、可靠、和快速
 - ▶ 系統目標—作業系統應該容易設計、製作、維護、有彈性、可靠、沒有錯誤、有效率

作業系統的設計和實現(續)

- 製作上一個可採用的重要原則是將策略(Policy)及方法(Mechanism)分開考量
 - 策略(Policy)：決定要做什麼事？(What will be done?)
 - 方法(Mechanism)：決定如何做？(How to do it?)
 - 基於策略和方法分開的原則，設計出來的系統會比較有彈性(flexibility)
 - ▶ 通常策略會因時因地而有所改變，若採用的方法是適用範圍較廣且較不受策略影響的方法，則策略發生變動時，方法可能僅需修改部分參數的設定即可。
 - ▶ 例如：Solaris所採取的排程方法是由一個可載入表格(loadable table)來決定，可因策略的改變載入不同的表，使其成為time shared, batch processing, real time, 或其他類型的運作模式。

作業系統的實現(Implementation)

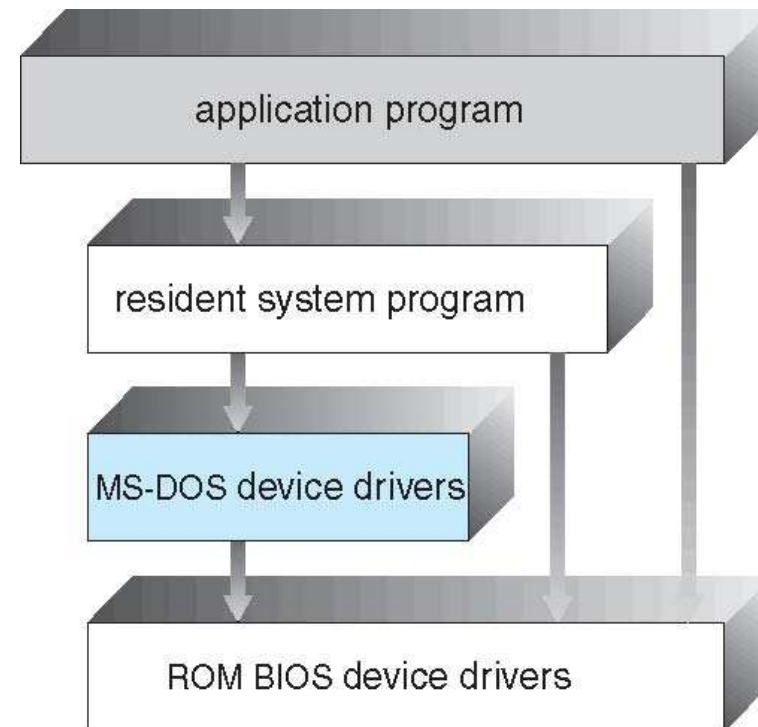
- 使用的電腦語言變化很大
 - 早期的作業系統是以組合語言
 - 然後是以系統程式語言，例如 Algol, PL/1
 - 現在 C, C++
- 實際上，是不同的語言混合使用
 - 最低層次的以組合語言
 - 主體是C
 - 系統程式是以C、C++和腳本語言(scripting languages)如 PERL, Python, shell scripts等
- 使用高階語言比較容易移植(port)到其它硬體

作業系統結構(Operating-System Structure)

- 作業系統結構討論 Operating-system structure discuss how the common components are interconnected and melded into a kernel
- 一般用途的作業系統是非常大的程式，由許多不同單元組成。
- 幾種不同的作業系統結構介紹
 - Simple Structure(簡易結構)
 - Layered Approach(分層結構)
 - Microkernel(微核心)
 - Modules(模組結構)

簡易結構(Simple Structure)

- 許多OS未有良好結構
- 例如 MS-DOS - 以最小的空間提供最多的功能
 - 不夠模組化
 - 有某些結構存在，但它的介面和功能層次未做好區隔

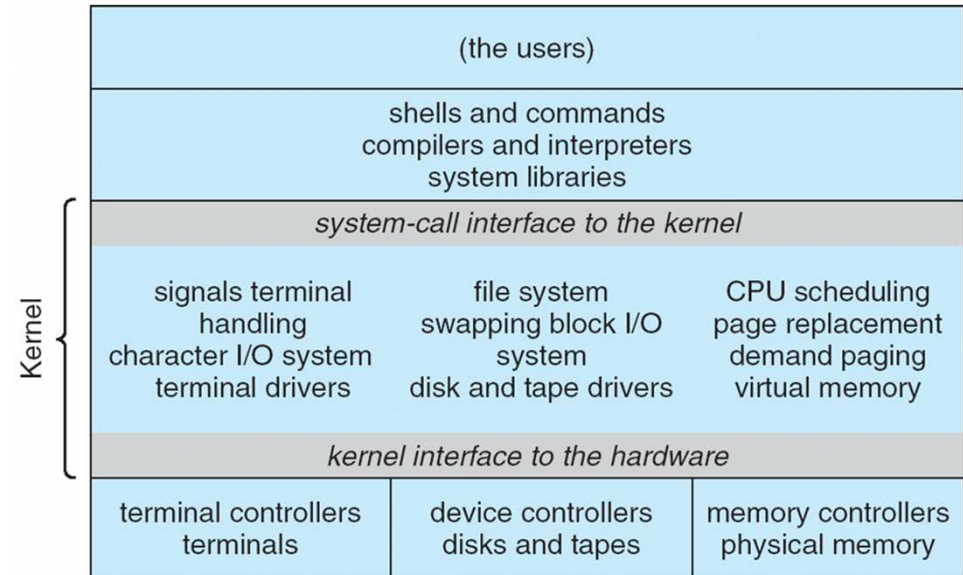


簡易結構(Simple Structure)(續)

■ 例如初始的UNIX

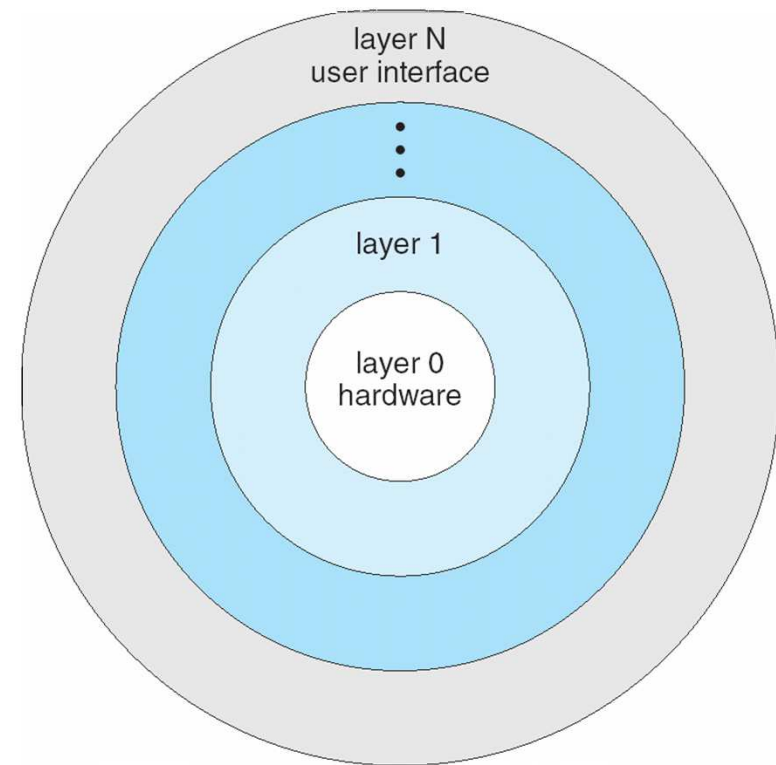
- 受限於硬體功能，只有有限的結構
- UNIX作業系統由系統程式及核心兩個部份構成
- 核心：
 - ▶ 在系統呼叫介面以下與實際硬體以上的部份
 - ▶ 提供檔案系統、CPU排班、記憶體管理、以及其它作業系統功能
 - ▶ 將許多功能放在同一層

■ 這種簡易未完全分層的結構使得系統難以維護及實現



階層式架構(Layered Approach)

- 作業系統的一個階層其實是一個抽象物件 (Abstract Object)。
- 作業系統被分為數層，每一層建立在底層上。底部層(第0層)是硬體，最高層(第N層)是使用者介面。
- 模組化, 就是每一層只能使用比它低一層的功能與服務。
- 階層式架構方法的主要優點是模組化 (Modularity)。就是每一層只能使用比它低一層的功能與服務。

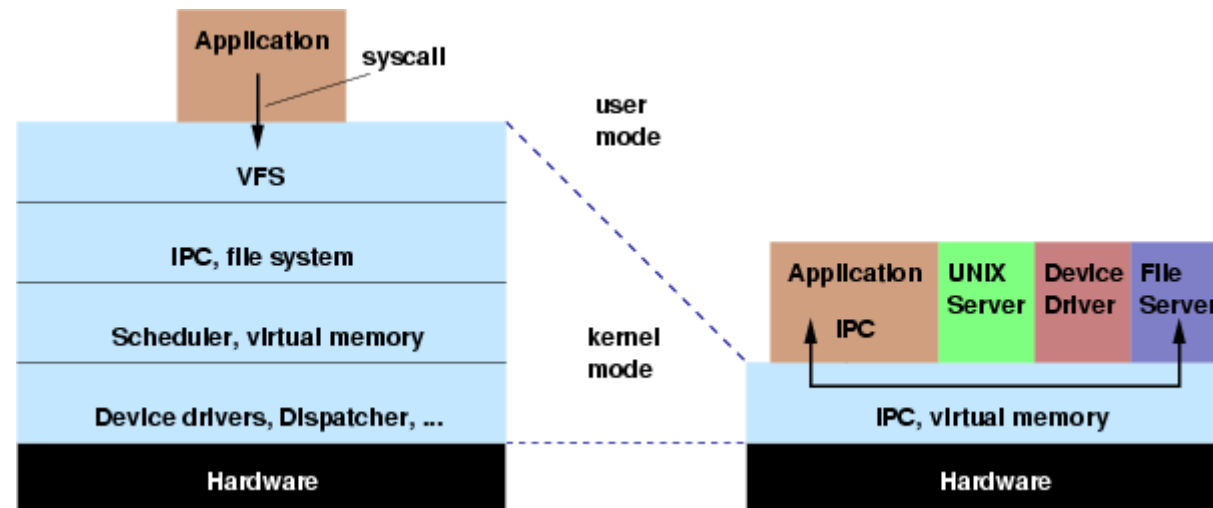


階層式結構(Layered Approach)(續)

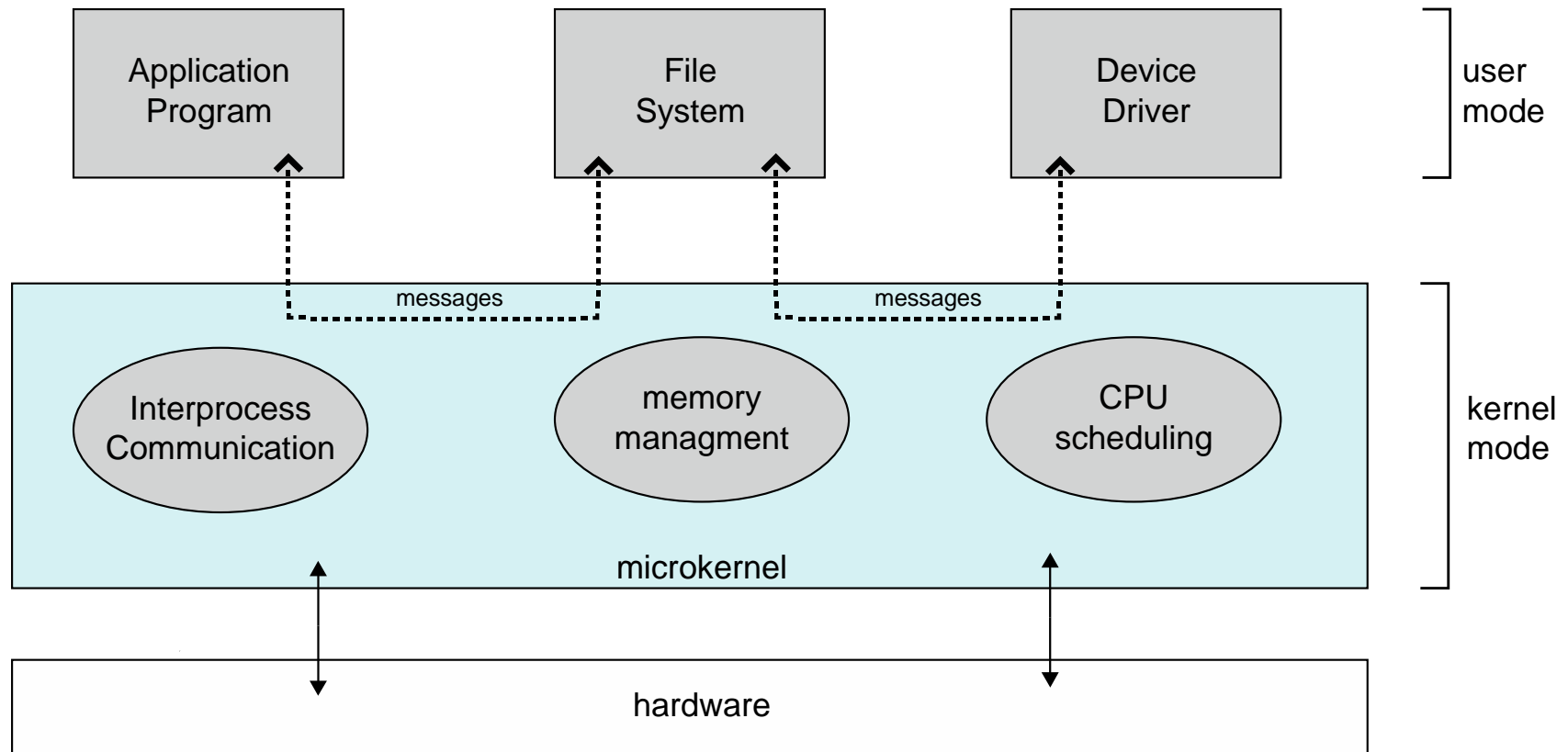
- 階層式架構讓除錯 (Debugging) 和系統驗證 (System Verification) 更簡單。
- 階層式架構方法主要的困難是層級如何界定。
- 階層式架構相對上比較沒有執行效率，因為，它的執行及資料必須一層一層地轉換如此會增加一些額外的負擔 (Overhead)。

微核心結構(Microkernel Structure)

- 將核心模組化 (Modularization)，將許多原放在核心非必要的基本元件移到使用者空間，使得核心僅由一群盡可能數量最小化的軟體程式組成。
- 雖然對於須保留在核心的元件尚無一致的共識，但應包含最低限度的行程管理，記憶體管理和行程間通訊等。
- 微核心的範例：Mach
 - Mac OS X核心(Darwin) 部分是建立在Mach上
- 使用者模組間使用訊息傳遞(message passing)作通信



微核心結構(Microkernel Structure)(續)



微核心結構(Microkernel Structure)(續)

■ 微核心結構的優點：

- 擴充性較高，新增的服務直接加到使用者空間即可，核心不需修改。
- 移植性較高
- 更安全性
- 更高的可靠度

■ 微核心結構缺點：

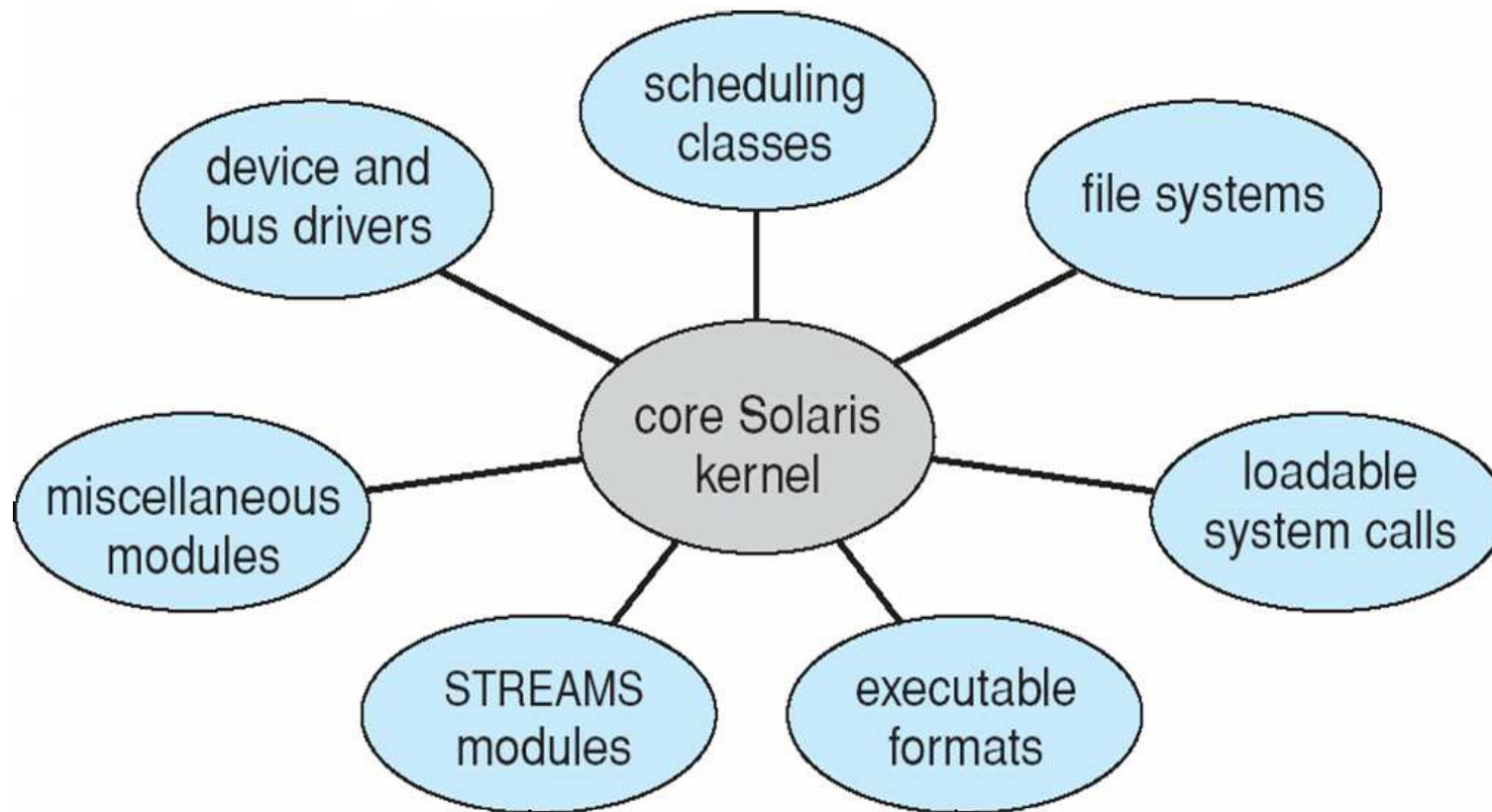
- 使用者空間到核心空間的通信會造成性能的額外負擔

模組(Modules)結構

- 採用可載入的核心模組(loadable kernel modules)可能是目前最好的OS設計做法
 - 核心可在開機時(boot time)或執行時(run time)動態連結外部的模組來提供額外服務
 - 每一個模組經由已知的介面和其它模組交談
 - 模組提供的服務在需要或開機時載入到核心
 - 目前多數的OS都有用到此方法，如:Linux, Mac OS, Windows.
- 與階層式架構比較
 - 相似的地方:每個核心模組單元均有一個已知的介面
 - 好處:模組結構更具彈性，因每一模組均可和其它模組交談。
- 與微核心結構比較
 - 相似的地方:主要核心模組僅需具備最基本功能及知道如何載入與如何與其他模組的通信的功能即可，所以主要核心模組會很小
 - 好處:模組結構會較有效率，因模組間不須透過訊息傳遞方式來通信，速度較快。

Solaris 的可載入模組做法

- Solaris 有一 core Solaris 模組及 7 種可載入的核心模組



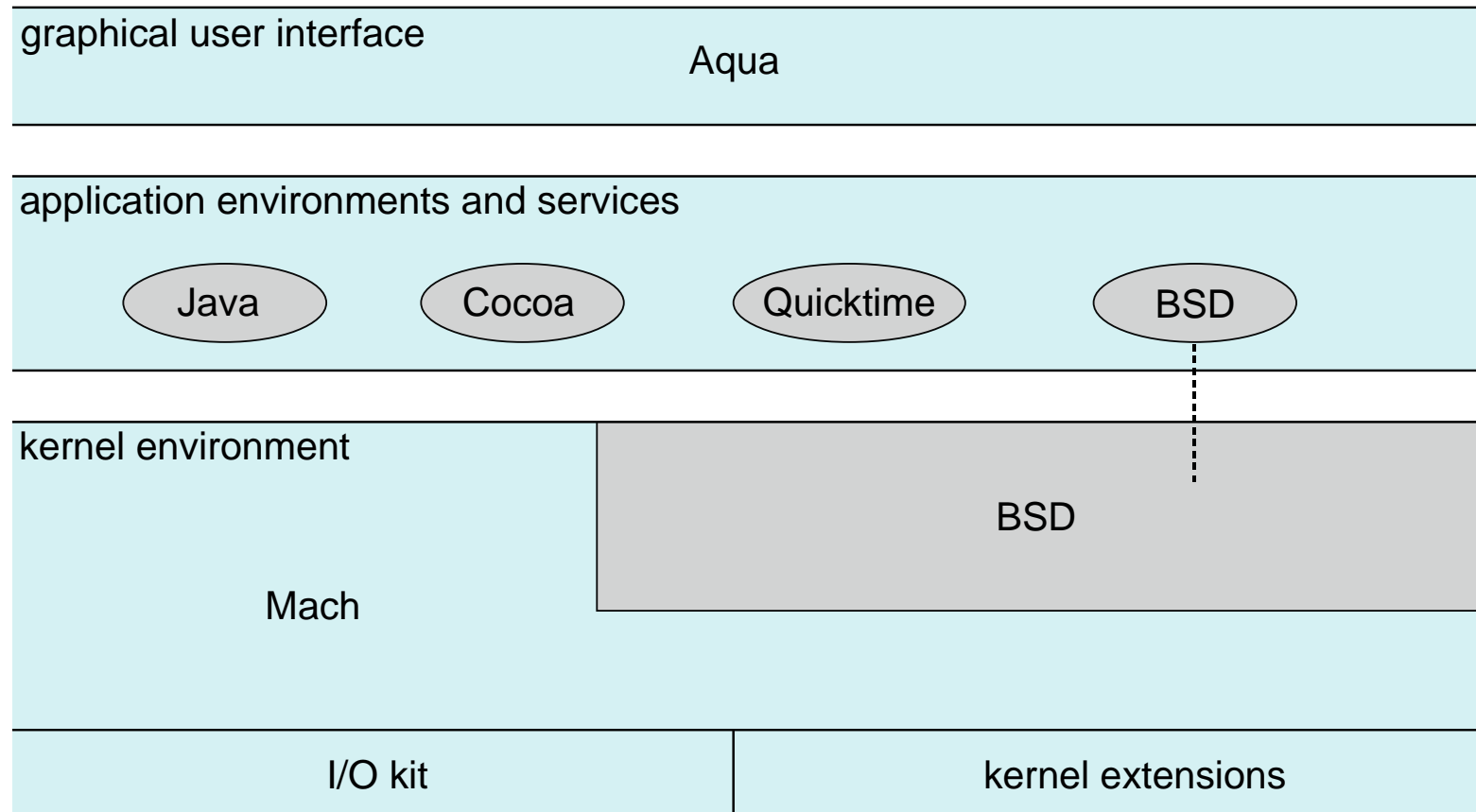
混和式系統結構(Hybrid Systems)

- 大部分現代的作業系統都不是只採單一結構
 - 組合不同的方法，以滿足性能、安全性和可用性等需求
- Unix和Solaris為 monolithic結構+loadable modules
- Windows為monolithic結構+子系統的微核心化做法+loadable modules
 - Apple Mac OS X 是混合、層狀的 Aqua UI，外加 Cocoa 程式環境
 - ▶ 由Mach微核心和BSD UNIX核心所組成的核心，外加I/O套件和動態可載入模組(稱為核心延伸)

混和式系統結構(Hybrid Systems)

- 大部分現代的作業系統都不是只採單一結構
 - 組合不同的方法，以滿足性能、安全和可用性等需求
- Unix和Solaris為 monolithic結構+loadable modules
- Windows為monolithic結構+子系統的微核心化做法+loadable modules
- Mac OS X 是混合階層結構、微核心及模組結構
 - 核心環境由Mach微核心、BSD UNIX核心、I/O kit及可載入的核心延伸模組所組成
 - Mach微核心包含記憶體管理、IPC(interprocess communication)及執行緒排程(thread scheduling)
 - BSD UNIX核心包含CLI介面、網路功能、檔案系統，及POSIX APIs

Mac OS X Structure



混和式系統結構(Hybrid Systems)(續)

■ iOS以Mac OS X為架構並加入行動裝置所需功能

- Apple給iPhone, iPad使用的行動作業系統
- 不能直接執行Mac OS X的應用程式
- 可以在不同的CPU架構(ARM 和Intel)執行
- 提供Cocoa Touch Objective-C API
- 媒體服務層提供繪圖、聲音和視訊服務
- 核心服務層提供雲端運算與資料庫處理
- 核心作業系統(Core OS)是建構在Mac OS X核心基礎上

Cocoa Touch

Media Services

Core Services

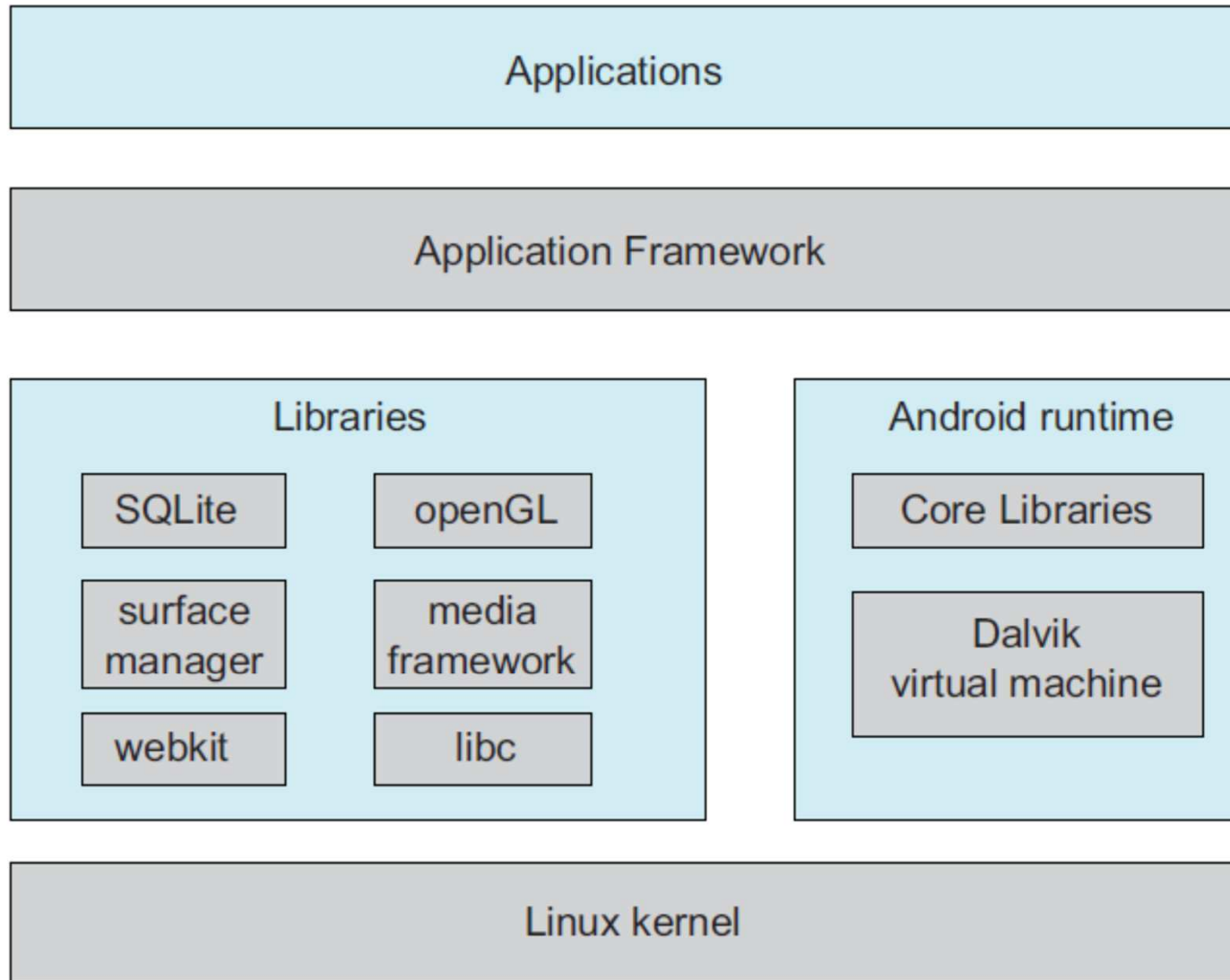
Core OS

混和式系統結構(Hybrid Systems)(續)

■ Android

- 由開放手機聯盟(Open Handset Alliance，主要是 Google)所發展
 - ▶ 開放原始碼
- 和iOS的架構相似
- Linux Kernel層
 - ▶ 提供行程、記憶體、裝置驅動程式管理，並加入電源管理
- 執行階段環境(Android runtime)包括了一組主要的函數庫和Dalvik虛擬機器
 - ▶ 以Java和Android API來發展Apps
 - ▶ Java類別的檔案被編譯成Java的位元組碼(bytecode)，然後轉換成在Dalvik VM上的可執行檔執行
- 程式庫(Libraries)包括了開發網頁瀏覽器(webkit)、資料庫 (SQLite)，多媒體、精簡後的libc等

Android Architecture

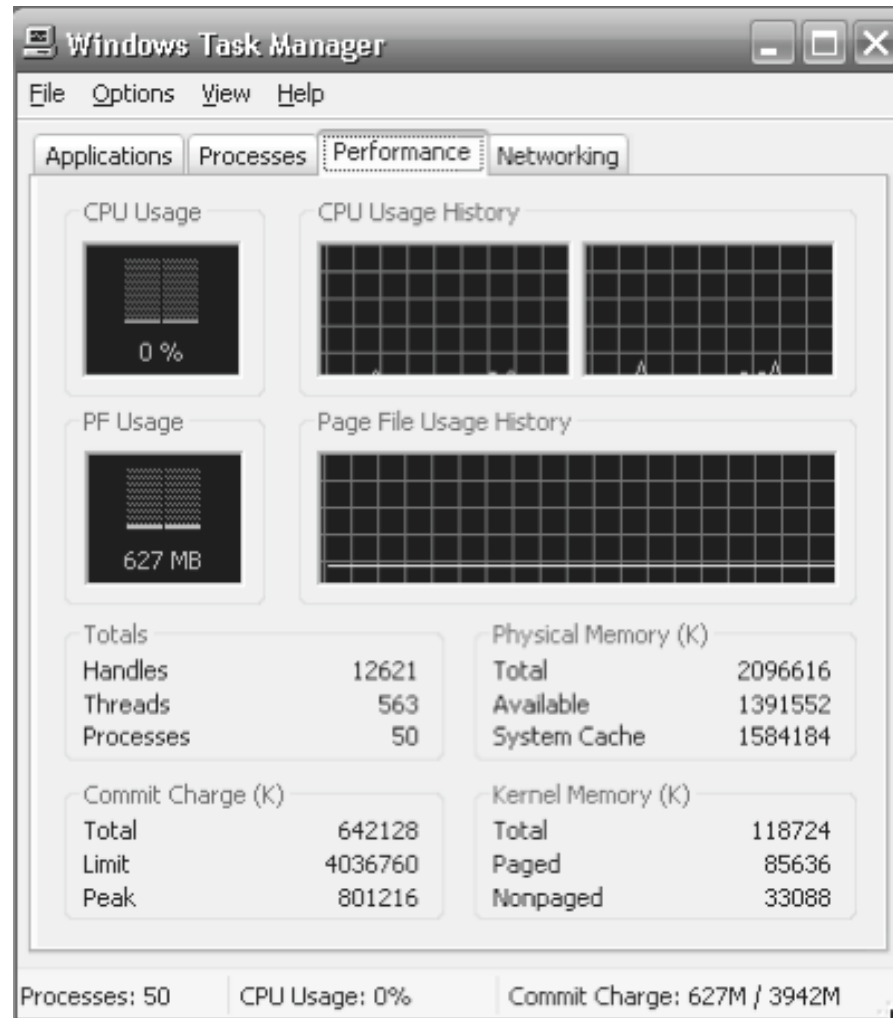


作業系統除錯(Operating-System Debugging)

- 除錯(Debugging)是發現和修正錯誤(bugs)，由於效能問題亦算一種錯誤，故除錯也包括效能調校(Performance Tuning)。
- 行程發生錯誤時，作業系統會將錯誤資訊保存於紀錄檔案(log files)
- 當行程發生嚴重錯誤而被強制結束時，作業系統會將行程當時的記憶體內容及狀態捕捉下來，保存在一個文件中，此行為稱為核心轉儲(core dump)。
- 作業系統失效會產生當機轉存(crash dump)檔，包含核心的記憶體內容。
- 性能調校可以使系統的性能最佳化
 - 可將活動的追蹤列表(trace listings)記錄下來作分析
 - 效能分析(Profiling)可得到特定指令的使用情形、函式呼叫的頻率及執行時間等。可用來決定程式的哪個部分應該被最佳化，從而提高程式的速度或者記憶體使用效率。
- Kernighan法則：「除錯的難度幾乎是寫該程式碼的兩倍。因此，如果你把程式寫的太精巧，那麼，你可能不夠聰明到可以解決你程式的錯誤。」

效能調校(Performance Tuning)

- 移除系統瓶頸 (bottlenecks) 來改善性能
- 作業系統必須有計算和顯示系統行為量測結果的工具。
- 例如, Linux的 top 程式, 或是Windows的工作管理員



作業系統生成(Operating-System Generation)

- 作業系統通常被設計成可給許多不同配備的電腦安裝及使用，因此OS必須為每個特定的電腦作組態及生成
- 可透過SYSGEN系統生成程式來獲得硬體系統的組態訊息
 - 用來建置可針對特定系統而編譯的核心
 - 生成更精簡與效率的核心

系統啟動(System Boot)

- 當系統電源啟動後，CPU內的指令暫存器(instruction register)會載入特定的記憶體位址，然後開始執行。
- 該位址為初始的bootstrap program儲存的地方
 - 儲存在ROM或EEPROM的一小段程式碼，透過它會找到核心、將它載入記憶體，然後開始執行。
- 有時候開機啟動會分成兩階段的步驟
 - ROM程式碼從磁碟的固定位置載入啟動區塊(boot block)，啟動區塊再從磁碟載入整個作業系統。
- Linux的GRUB允許從多個安裝的OS，選擇要啟動的OS

End of Chapter 2

