Chapter 2

作業系統結構

章節目標

- 認識作業系統提供的服務
- 說明系統呼叫如何提供作業系統服務
- 說明啟動作業系統的過程

2.1 作業系統服務

- 作業系統提供執行程式的環境
- 作業系統為程式及程式的使用者提供一定的服務
- 一套作業系統服務提供對使用者有幫助的功能
  - 使用者介面：絕大多數作業系統擁有**使用者介面** (user interface, UI)
    - **圖型使用者介面** (graphical user interface, GUI)
    - **命令行介面** (command-line interface, CLI)
  - 程式執行：系統必須把程式載入記憶體並且執行
    - 該程式必須有能力去結束本身的執行工作，無論是以正常或不正常的方式 (指出錯誤)
  - I/O 作業：一個正在執行的程式可能需要輸入和輸出
    - I/O (輸入/輸出) 工作可能包含一個檔案或一項 I/O 裝置

2.1 作業系統服務

- 檔案系統的使用：檔案系統讓人特別有興趣
  - 當然，程式需要讀寫檔案和目錄，它們也會需要用名稱來建立或刪除一些檔案、搜尋檔案和列出檔案資料
  - 最後，有些程式包含許可管理，依檔案本身來允許或拒絕存取檔案
- 通信：在很多情況下，一個行程必須與另一個行程交換資訊
  - 這種通信是在同一部電腦上執行的行程之間的通信，或藉由電腦網路連接不同電腦內執行之行程的通信
  - 藉由**共用記憶體** (shared memory) 或**訊息傳遞** (message passing) 技術可以完成通信
  - 封包 (packet) 經由作業系統在行程間移動

2.1 作業系統服務

- 錯誤偵測：作業系統需要不斷地偵測和修正的錯誤。錯誤可能產生
  - 在 CPU 和記憶體硬體中 (譬如記憶體錯誤或電力中斷)
  - 在 I/O 裝置中 (例如磁碟的同位錯誤、網路連接失敗或印表機缺紙)
  - 或在使用者程式中對於任何一類的錯誤，作業系統都應該採取適當的行動，以確保正確且持續的運算
  - 有時候，除了停止系統別無選擇；其它時候，可能必須停止造成錯誤的行程，或是傳回錯誤碼給行程以便偵錯並更正錯誤

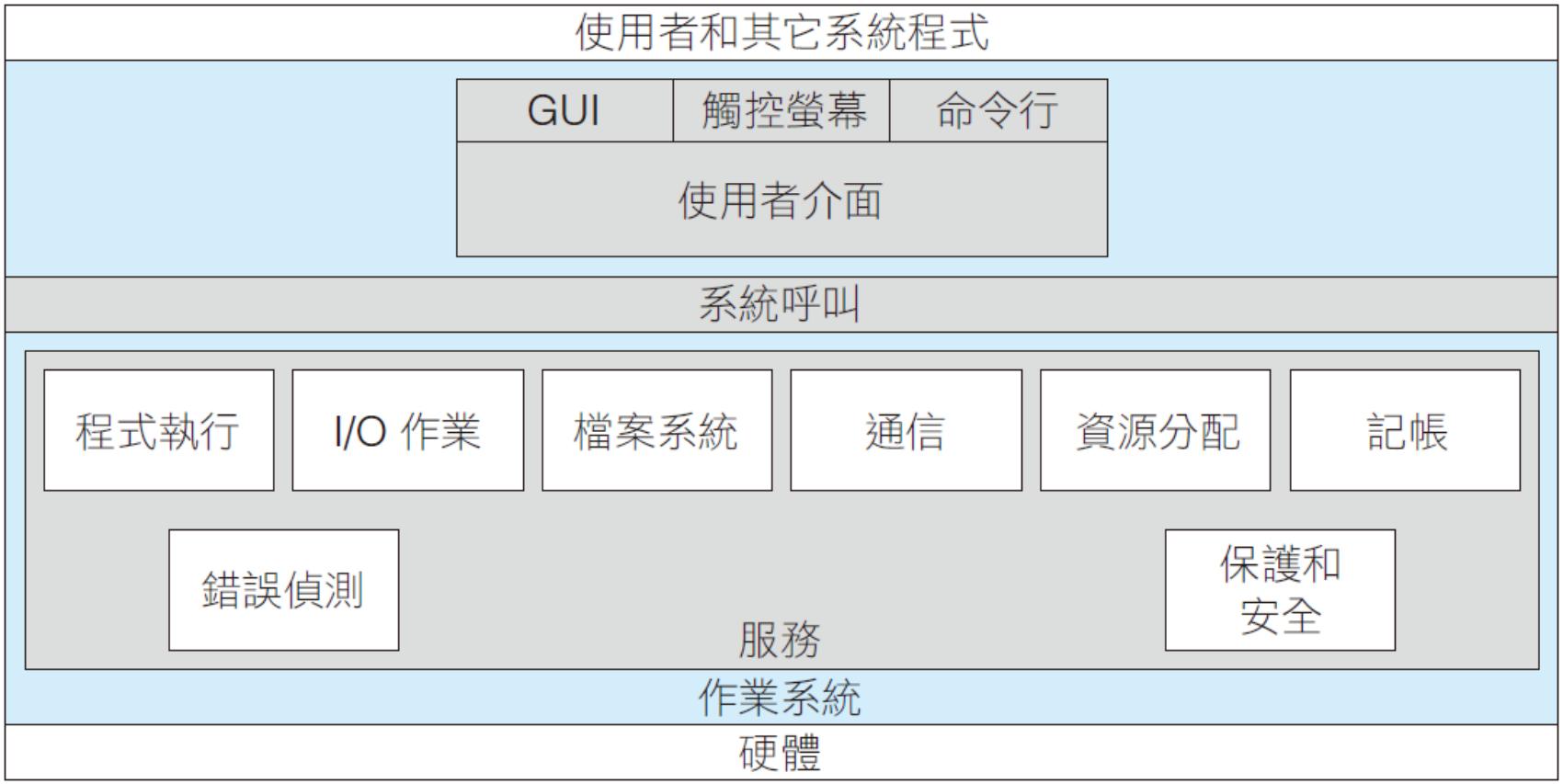
2.1 作業系統服務

- 作業系統的另一組功能並不是為了幫助使用者而存在，而是為了確保系統本身能有效運作。具多行程的系統可以藉由在不同行程之間共用電腦資源來提高效率
  - 資源分配：當有許多使用者或許多工作同時執行的時候，就必須將電腦資源分配給他們。作業系統管理許多不同類型的資源
    - 有些 (如 CPU 週期、主記憶體、檔案儲存等) 可能會有特別配置碼 (special allocation code)
    - 其它 (例如 I/O 裝置) 可能就有較普通的要求與釋放碼
  - 紀錄檔記錄：我們想追蹤哪些程式使用多少的資源，以及哪些種類的電腦資源

2.1 作業系統服務

- 保護和安全：在多使用者電腦系統或網路電腦系統中存放資料的人，可能希望控制該資料的使用
  - 當一些互無關聯的行程同時執行時候，任何一個行程都有可能干擾其它使用者或是作業系統本身
  - 保護的工作牽涉到確保所有對系統資源的存取都在控制之中
  - 系統不受外界干擾的安全性 (security) 也很重要
  - 此種安全性是從要求每一個使用者需要經過系統認證開始，通常是藉由密碼才能允許存取系統資源
  - 此種安全推廣到保護外界 I/O 裝置 (包括網路配接器) 不受不正常的存取，並且記錄下所有的這種連接以便做可疑之偵測

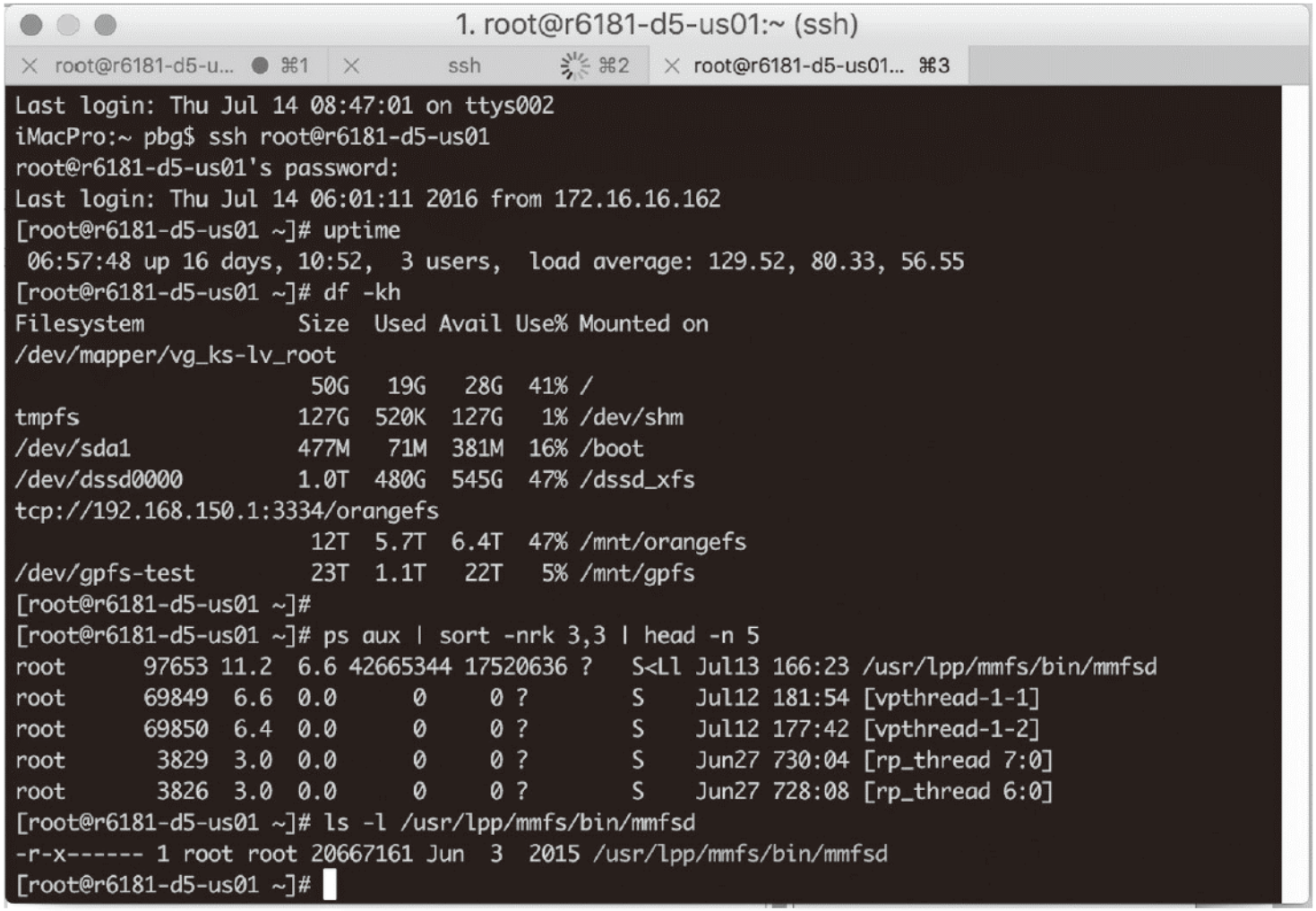
圖 2.1　作業系統服務的概觀



2.2 使用者與作業系統介面

- 一個方法是提供命令行列介面或**命令解譯器** (command interpreter)，它允許使用者直接地鍵入命令
- 在有多個命令解譯器可選擇的系統上，這些命令解譯器稱為**外殼** (shells)

圖 2.2　在 macOS 中的 bash shell 命令解譯器



2.2.2 圖型使用者介面

- 與作業系統介面的第二個策略是經由使用者友善的圖型使用者介面或 GUI
  - 它不像命令列介面是直接輸入命令，而是讓使用者利用具有滑鼠為基礎的視窗和具有具**桌面** (desktop) 特徵的表單系統
  - 使用者移動滑鼠，以便將滑鼠指標定位在螢幕 (桌面) 的影像或**圖像** (icon)，這些圖像代表程式、檔案、目錄和系統功能
  - 根據滑鼠指標的位置，按下滑鼠的按鍵，可以呼叫程式、選擇檔案或目錄——即稱 **folder** ——或拉下一個含有命令的表單

2.2.2 圖型使用者介面

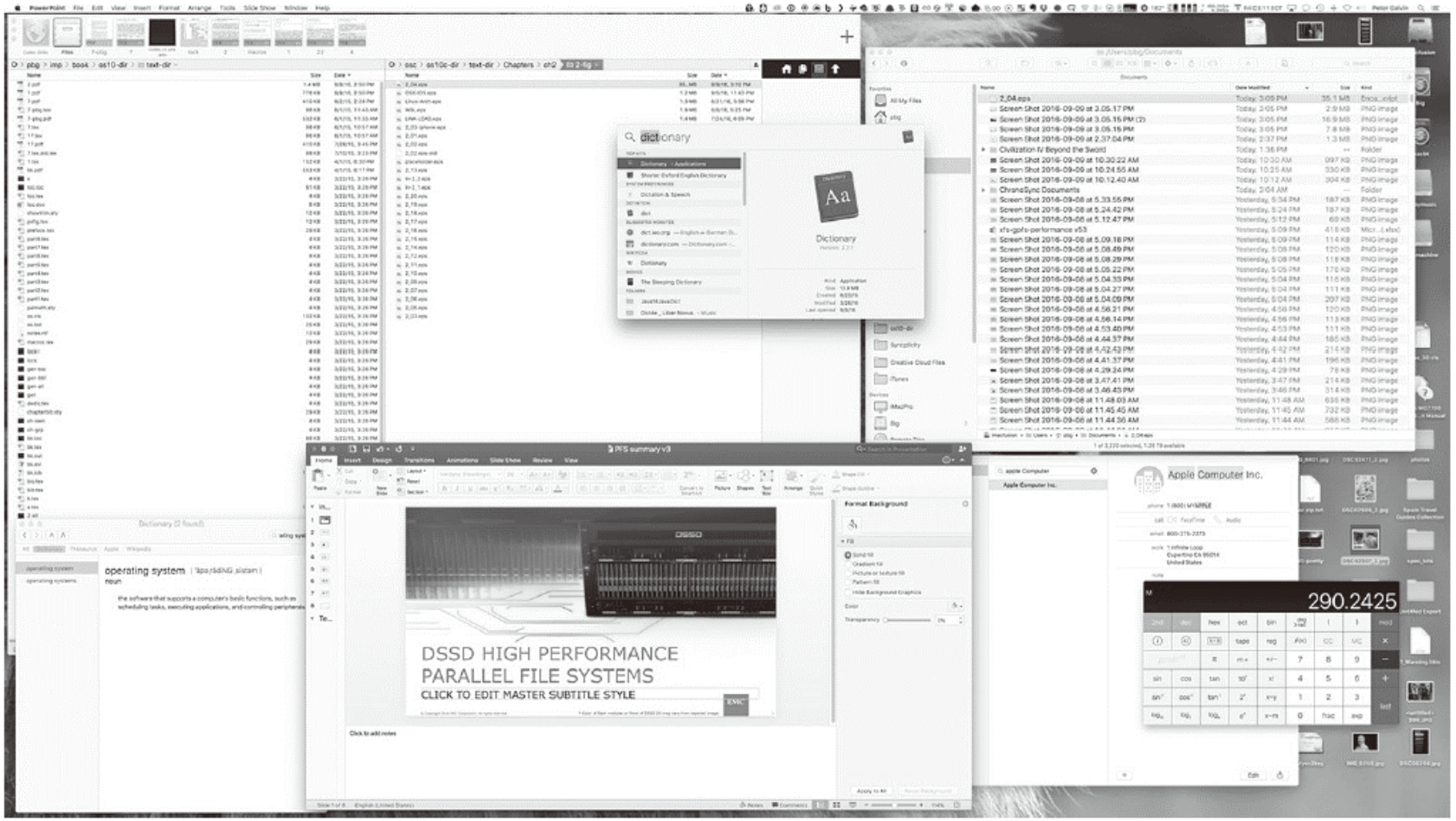
- 圖型使用者介面最初產生的部份原因，是由於 Xerox PARC 研究機構在 1970 年代初期的研究
- 麥金塔作業系統的使用者介面隨著年代進行不同的改變
  - 最明顯的是 macOS 出現採取 Aqua 介面
- 傳統上，UNIX 系統是由命令行介面所支配
  - UNIX 有不同 GUI 介面可用
  - 另外，有來自各種不同的開放原始碼計畫在圖型使用者介面已經有重要發展
    - 諸如 ***K Desktop Environment*** (或 ***KDE***) 和 GNU 計畫的 *GNOME* 桌面

2.2.2 圖型使用者介面

- 在 Linux 和各種不同的UNIX 系統上執行 KDE 和 GNOME 桌面
  - 而且兩者在開放原始碼執照下是可以使用的，也就是它們的原始碼在特定版權條款下可以很容易被閱讀和修改

圖 2.3 iPhone 的觸控螢幕

圖 2.4 macOS GUI



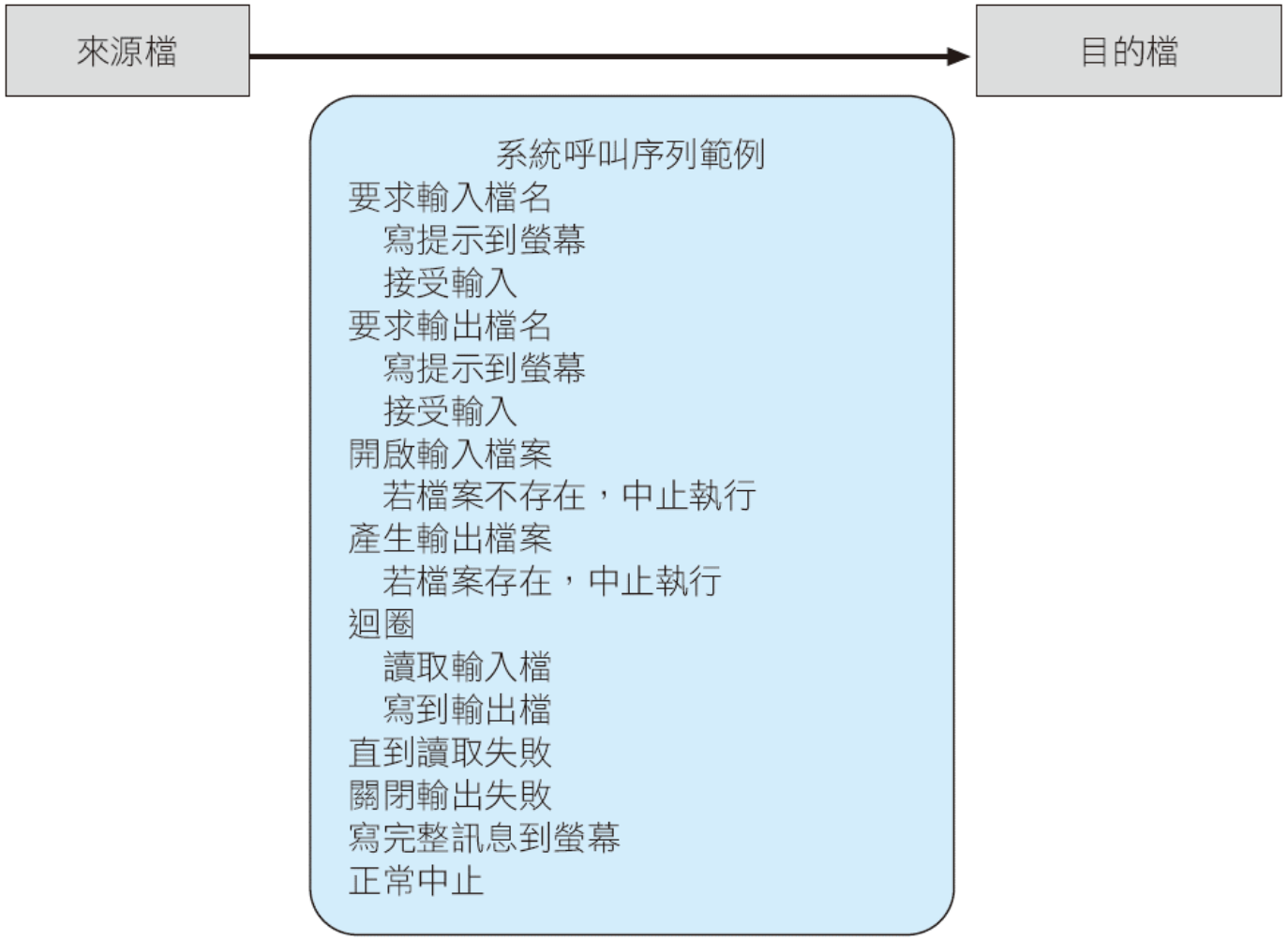
2.3 系統呼叫

- **系統呼叫** (system call) 提供一個由作業系統服務的介面
  - 這類呼叫一般以 C 或 C++ 寫成的函數，雖然低階工作 (例如，硬體必須直接存取的工作) 可能需要以組合語言指令來寫
- 你可以看到，即使簡單的程式也會大量地使用作業系統
  - 通常系統每秒執行成千的系統呼叫
  - 然而，大多數程式設計者從未瞭解這個層次細節
  - 基本上，應用程式開發人員依照應用程式介面 (application programming interface, API) 設計程式

2.3 系統呼叫

- API 指定一組可用的函數給應用程式設計者，包括
  - 傳給每個函數的參數和預期傳回值
- 對於應用程式設計者而言，三個最常用的 API 是
  - Windows 系統的 Windows API
  - 以 POSIX 系統為基礎的 POSIX API (事實上，包括所有 UNIX、Linux 和 macOS 的版本)
  - 設計在 Java 虛擬機上執行 Java 程式的 Java API
- 注意——除非特別說明——本書使用的系統呼叫名稱是通用的例子
  - 每一個作業系統對於每一個系統呼叫都有自己的名稱

圖 2.5　如何使用系統呼叫的範例



標準 API 的範例

- 考慮 UNIX 和 Linux 系統可以使用的函數 read() 作為標準 API 的範例。此函數的 API 可以藉由在命令列執行以下的指令後，由 man 的分頁取得

man read

- API 的描述如下：

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

傳回值　函數名稱　參數

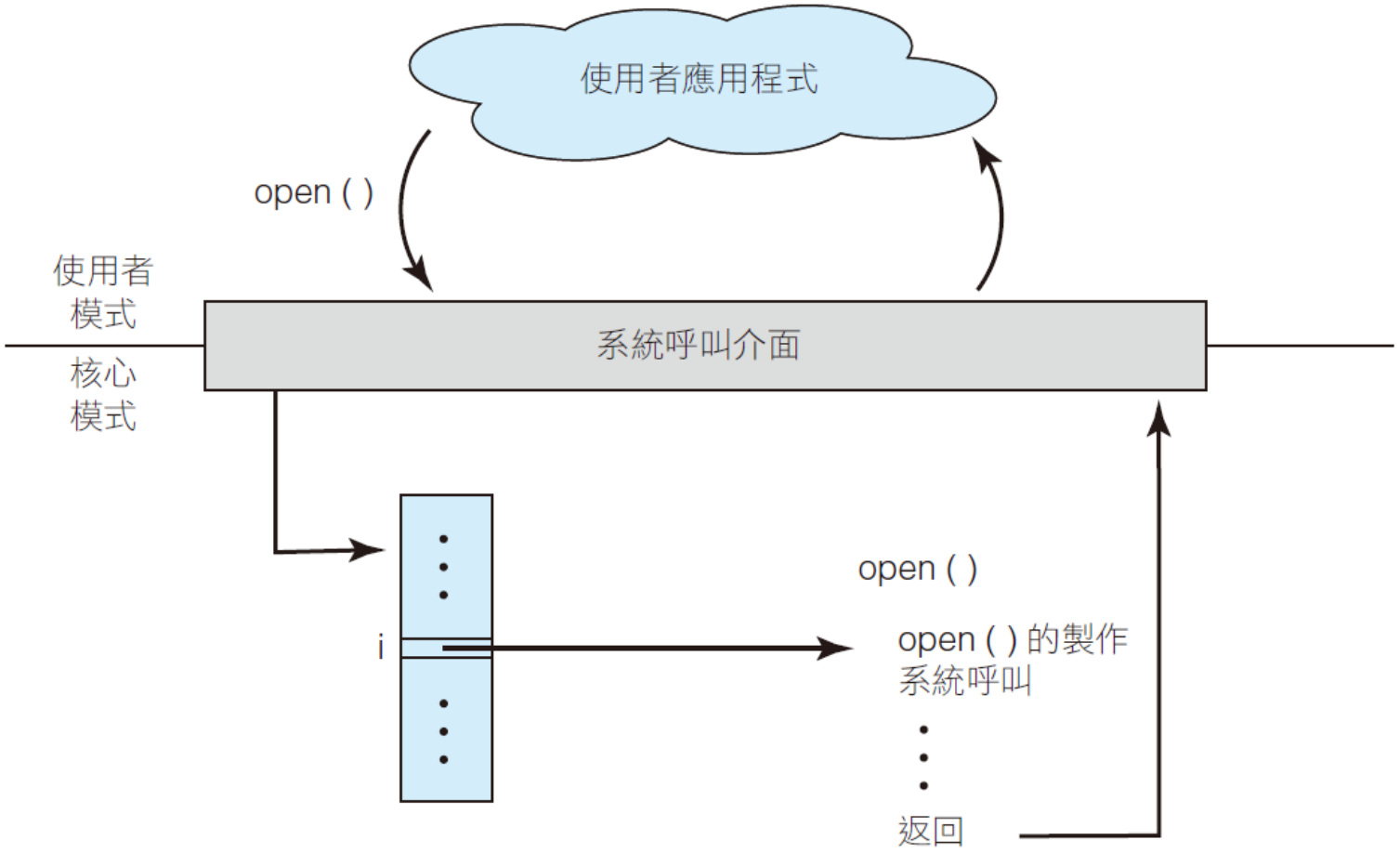
標準 API 的範例

- 使用函數 read() 的程式必須包含標頭檔 unistd.h，當這個檔案定義 ssize_t 和 size_t 等資料型態。參數以下面的格式傳遞給 read()：
  - int fd —— 被讀取檔案的檔案指標
  - void *buf —— 資料被讀入到此緩衝區
  - size_t count —— 被讀入到緩衝區的資料最大位元組數
- 成功讀取後，讀取到的位元組數目會被傳回。傳回值 0 表示檔案結束，如果有錯誤發生，read() 傳回 –1

2.3 系統呼叫

- RTE 對於大多數程式語言，執行階段的支援系統 (內含於編譯器的一組函數建成之程式庫) 提供一個 **系統呼叫介面** (system-call interface)，這個介面扮演著作業系統提供之系統呼叫的連結
  - 此系統呼叫介面攔截在 API 中的函數呼叫，然後啟動作業系統裡面必要的系統呼叫
  - 通常，每個系統呼叫都與一個數字相關聯，而系統呼叫介面則根據這些數字去維護一個索引表格
  - 然後，此系統呼叫介面啟動作業系統核心中指定的系統呼叫，並且傳回系統呼叫狀態和任何回傳數值

圖 2.6　使用者應用程式載入 open() 系統呼叫的處理



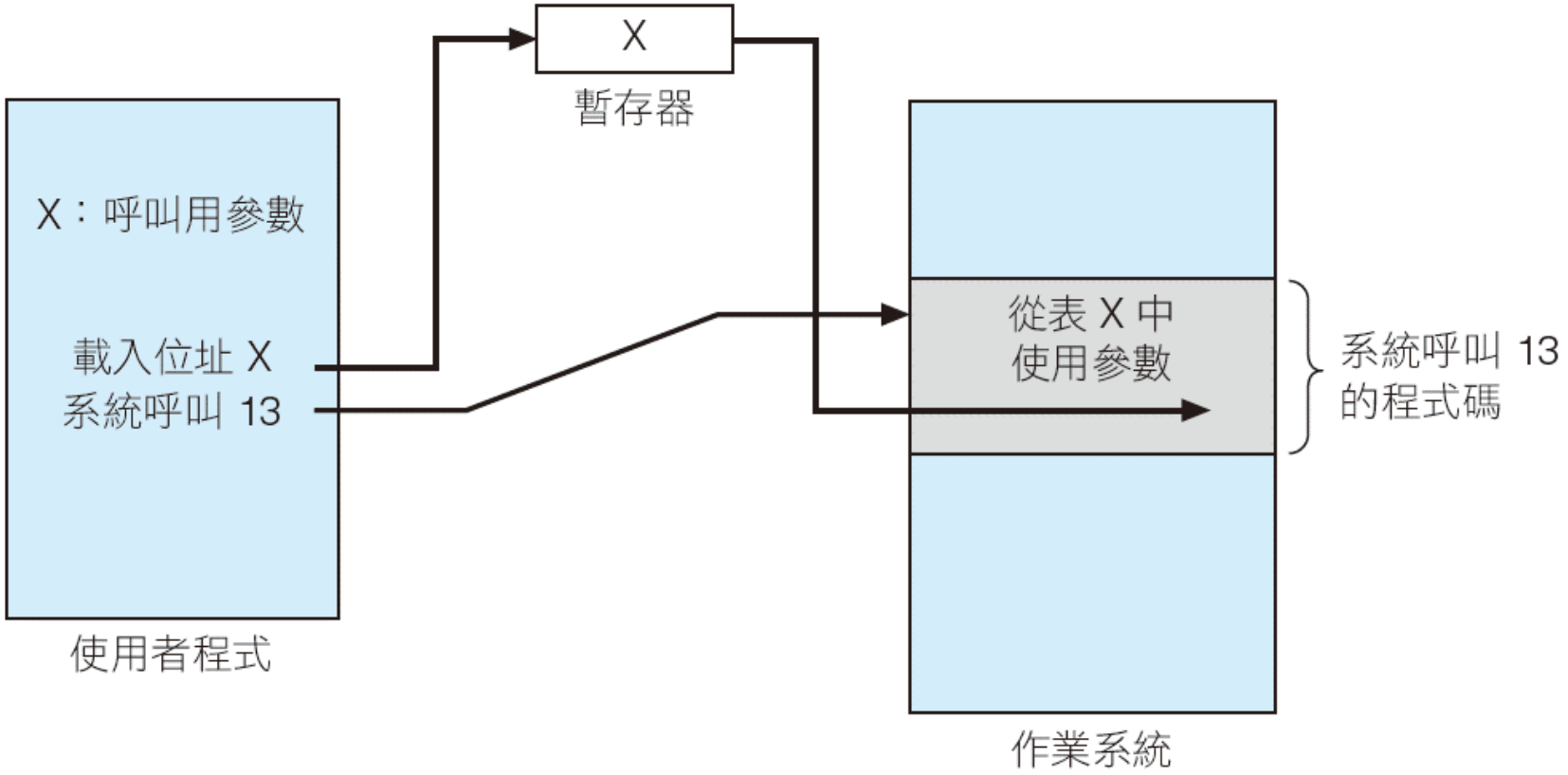
2.3 系統呼叫

- 根據所使用的電腦，系統呼叫會以不同的方式出現。實際上需要的資訊往往比我們在系統呼叫所指定的還要多
  - 資訊的類型與數量視其作業系統和呼叫的特性而定
- 通常有三種方法可用來傳遞參數至作業系統
  - 最簡單的一種是將參數傳遞於暫存器 (register) 中
    - 但有些情況，參數會比暫存器還多
    - 在這些情況下，參數通常以區塊 (block) 或表格 (table) 的方式儲存在記憶體中，而此區塊的位址則是以置於暫存器中的參數來傳遞 (圖 2.7)

2.3 系統呼叫

- Linux 就是採用這種方法
  - 如果有 5 個或更少的參數，就用暫存器
  - 如果有 5 個以上的參數，就使用區段方法
- 參數也可以由程式放置於 [或者**推放** (push)] **堆疊** (stack) 上，再由作業系統從堆疊中**取回** (pop)
- 有些作業系統比較喜歡用區塊或堆疊方法，因為這種方式不限制傳遞參數的數量或長度

圖 2.7　以表格方式傳遞參數



2.3.3 系統呼叫的類型

- 系統呼叫也可概略地分成六大類：
  - 行程控制 (process control)
  - 檔案管理 (file manipulation)
  - 裝置管理 (device manipulation)
  - 資訊維護 (information maintenance)
  - 通信(communication)
  - 保護 (protection)

行程控制

- 執行中的程式可能會因為正常的結束 (end()) 或不正常的中止執行 (abort()) 而停止它的執行
  - 如果系統呼叫是以不正常的情形下中止執行中的程式
    - 或如果程式碰到一個問題而造成錯誤陷阱
  - 有時候需要將記憶體傾印出來和產生一些錯誤訊息
  - 傾印是寫到磁碟中，而且可以用**偵錯程式** (debugger)
    - 一個系統程式用來幫助程式設計人員發現並修正錯誤或**缺陷** (bug)
    - 找出問題的原因

檔案管理

- 建立檔案，刪除檔案 (create file, delete file)
- 開啟，關閉 (open, close)
- 讀出，寫入，重定位置(read, write, reposition)
- 獲取檔案屬性，設定檔案屬性 (get file attributes, set file attributes)

裝置管理

- 要求裝置，釋回裝置 (request device, release device)
- 讀出，寫入，重定位置 (read, write, reposition)
- 獲取裝置屬性，設定裝置屬性 (get device attributes, set device attributes)
- 邏輯上地加入或移除裝置 (logically attach or detach devices)

資訊維護

- 取得時間或日期，設定時間或日期 (get time or date, set time or date)
- 取得系統資料，設定系統資料 (get system data, set system data)
- 取得行程、檔案或裝置的屬性 (get process, file, or device attributes)
- 設定行程、檔案或裝置的屬性 (set process, file, or device attributes)

通信 & 保護

- 通信 (communications)
  - 建立、刪除通信連接 (create, delete communication connection)
  - 傳送、接收訊息 (send, receive messages)
  - 傳輸狀況訊息 (transfer status information)
  - 連接或移除遠程裝置 (attach or detach remote devices)
- 保護 (protection)
  - 獲取檔案權限 (get file permissions)
  - 設置檔案權限 (set file permissions)

Windows 和 UNIX 系統呼叫的範例

	Windows	UNIX
行程控制	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
檔案管理	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
裝置管理	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()

Windows 和 UNIX 系統呼叫的範例

	Windows	UNIX
資訊維護	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
通信	CreatePipe()	pipe()
	CreateFileMapping()	shm_open()
	MapViewOfFile()	mmap()
保護	SetFileSecurity()	chmod()
	InitializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

標準 C 程式庫的範例

- 標準 C 程式庫提供一部份系統呼叫介面給許多版本的 UNIX 和 Linux。讓我們以一個 C 程式呼喚 printf() 敘述作為範例。C 程式庫攔截此呼叫，並呼叫作業系統中需要的系統呼叫 (或一些呼叫)—在此範例中是 write() 系統呼叫。C 程式庫接收 write() 傳回的數值，並傳遞回使用者程式

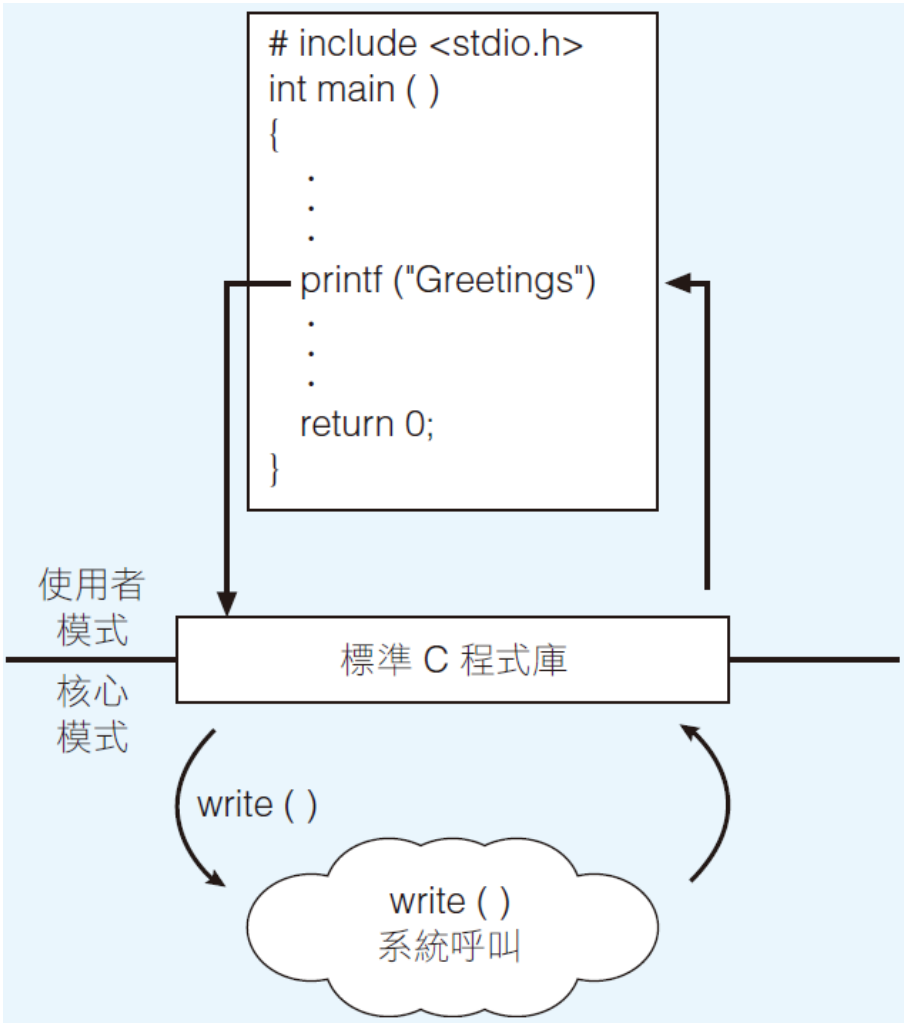
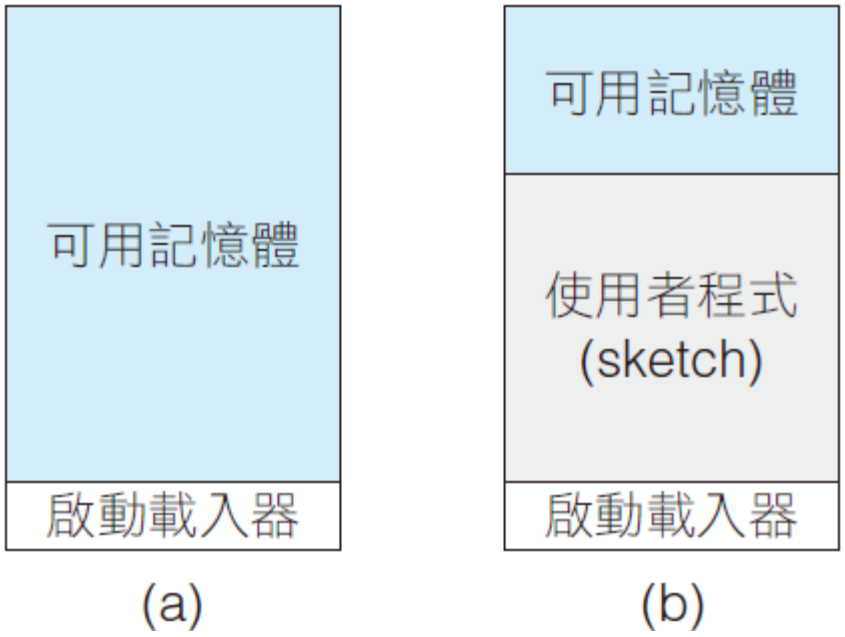


圖 2.9 Arduino 執行



(a) 系統起始；(b) 執行 sketch 程式

圖 2.10　FreeBSD 系統執行多程式

2.4 系統服務

- **系統服務** (system services) 又稱為**系統常式** (system utilities)
  - 它們提供一些程式開發與執行的便利環境，有些只是使用者和系統呼叫之間的介面
  - 其它則更加複雜，它們可分成這些類型：
    - 檔案管理：
      - 這些程式可以建立、刪除、複製、重新命名、列印、列出，以及一般地存取和操作檔案及目錄
    - 狀態資訊：
      - 有些程式會向作業系統詢問日期、時間、可用的記憶體或磁碟空間、使用者數量或其它類似的狀態資訊

2.4 系統服務

- 其它狀態資訊則較複雜，例如提供詳細的效能、登錄和除錯訊息
- 通常，這種程式格式化和列印輸出到終端機、其它輸出裝置、檔案，或者顯示在視窗的使用者介面
- 有些系統也支援登錄檔 (registry) 資訊，登錄檔資訊是用來儲存和取回組態的訊息

◆ 檔案修改：

- 有些文書編輯程式可用來產生和修改存放在磁碟或其它儲存體中的檔案內容
- 也可能有特殊指令來搜尋求檔案的內容或執行文書的轉換

2.4 系統服務

- 程式語言支援：
  - 編譯器、組譯器、除錯程式和普通程式語言的直譯器 (例如 C、C++、Java 和 Python) 都是由作業系統提供或是可以個別下載
- 程式的載入與執行：
  - 一旦某個程式組譯或編譯好之後，它必須載入記憶體中以便執行
  - 系統可以提供絕對載入程式、可重新定位載入程式、鏈結編輯程式和重疊載入程式來執行載入的工作
  - 對於高階語言及機器語言的除錯系統也是必要的

2.4 系統服務

- 通信：
  - 這些程式提供產生虛擬連接於行程、使用者及電腦系統之間的機制
  - 它們允許使用者傳送訊息到其它的螢幕上、瀏覽網頁、傳送電子郵件、遠端登錄或者從某一機器傳送檔案至另一機器
- 背景服務：
  - 所有一般用途系統有辦法在系統啟動時發出特定的系統程式行程
  - 有些行程在完成它們的工作後結束，而其它的繼續執行直到系統停止
  - 一直執行的系統程式行程被稱為服務 (services)、子系統 (subsystems) 或守護程序

2.4 系統服務

- 這些**應用程式** (application program) 包括
  - 網頁瀏覽器
  - 文書處理器
  - 文字格式化程式
  - 試算表
  - 資料庫系統
  - 編譯器
  - 繪圖
  - 統計分析套裝軟體
  - 遊戲程式

2.7 作業系統的設計和製作

- 在最高的設計層次之外，需求可能更難標明出來。需求可以分為兩個基本類型：
  - 使用者目的 (user goal)
  - 系統目的 (system goal)
- 使用者想要系統中具有某些明顯的性質
  - 系統必須便於使用、容易學習、容易使用、可靠、安全及快速
  - 當然，這些目的在系統的設計之中並不是非常有用，因為如何達成這些目標沒有一般性的定論

2.7 作業系統的設計和製作

- 對於那些必須設計、建立、維護，甚至操作作業系統的人，也有相似的需求集合必須加以定義：
  - 系統應該很容易設計、製作與維護
  - 它應該具有彈性、可靠的、沒有錯誤及有效率
- 再一次強調，這些需求很含糊而且可能用各式各樣的方法詮釋

2.7 作業系統的設計和製作

- 一個作業系統規格訂定與設計是一個高度創造性的工作
  - 雖然沒有教科書可以教你如何做，**軟體工程** (software engineering) 就是這些原則的理論基礎，現在我們回過頭來討論這些原則

2.7.2 機制與策略

- 一個重要的原則是將**機制** (mechanism) 從**策略** (policy) 中分離
  - 機制決定如何做某些工作，相對地，策略決定做什麼事
    - 例如，計時器
- 策略與機制的分開對提高作業系統的彈性非常重要
  - 策略可能因時或因地而改變
  - 在最差的情況，每一次策略的改變將牽連基礎機制的改變
  - 如果採用一個一般性的機制，則一個策略上的改變，只需重新定義某些系統的參數

2.7.3 製 作

- 早期的作業系統是由組合語言寫成的
  - 現在，大部份是用較高階的語言寫成，例如 C或 C++，而有少部份的系統是用組合語言撰寫
  - 實際上，經常有一種以上的高級語言被使用。可以使用組合語言與 C 語言來撰寫其最低層核心的程式別
    - 更高級別的常式可以使用 C 和 C++ 撰寫，而系統程式庫則可以用 C++ 甚至更高級別的語言編寫
  - Android 則提供了一個很好的例子：它的核心主要是用 C 語言和某種組合語言編寫的
  - 大多數 Android 系統程式庫都是用 C 或 C++ 編寫的，其應用程式的框架—系統提供開發人員介面—主要是使用 Java 撰寫的
  - 我們將在 2.8.5.2 節中詳細介紹 Android 的架構

2.7.3 製　作

- 使用高階語言或是系統製作語言來製作作業系統的優點，就和這些語言用在應用程式上相同：
  - 程式碼可以更快的寫好、更精簡、更容易瞭解和偵錯
  - 除此之外，改進編譯技術可以改善整個作業系統的程式
  - 最後，如果作業系統是用較高階語言撰寫，就很容易移植 (port) 到其它硬體上
  - 這對於能夠在多種不同硬體系統的作業系統執行而言相當重要，例如
    - 小型嵌入式設備
    - Intel x86 系統
    - 在手機和平板電腦上執行的 ARM 晶片

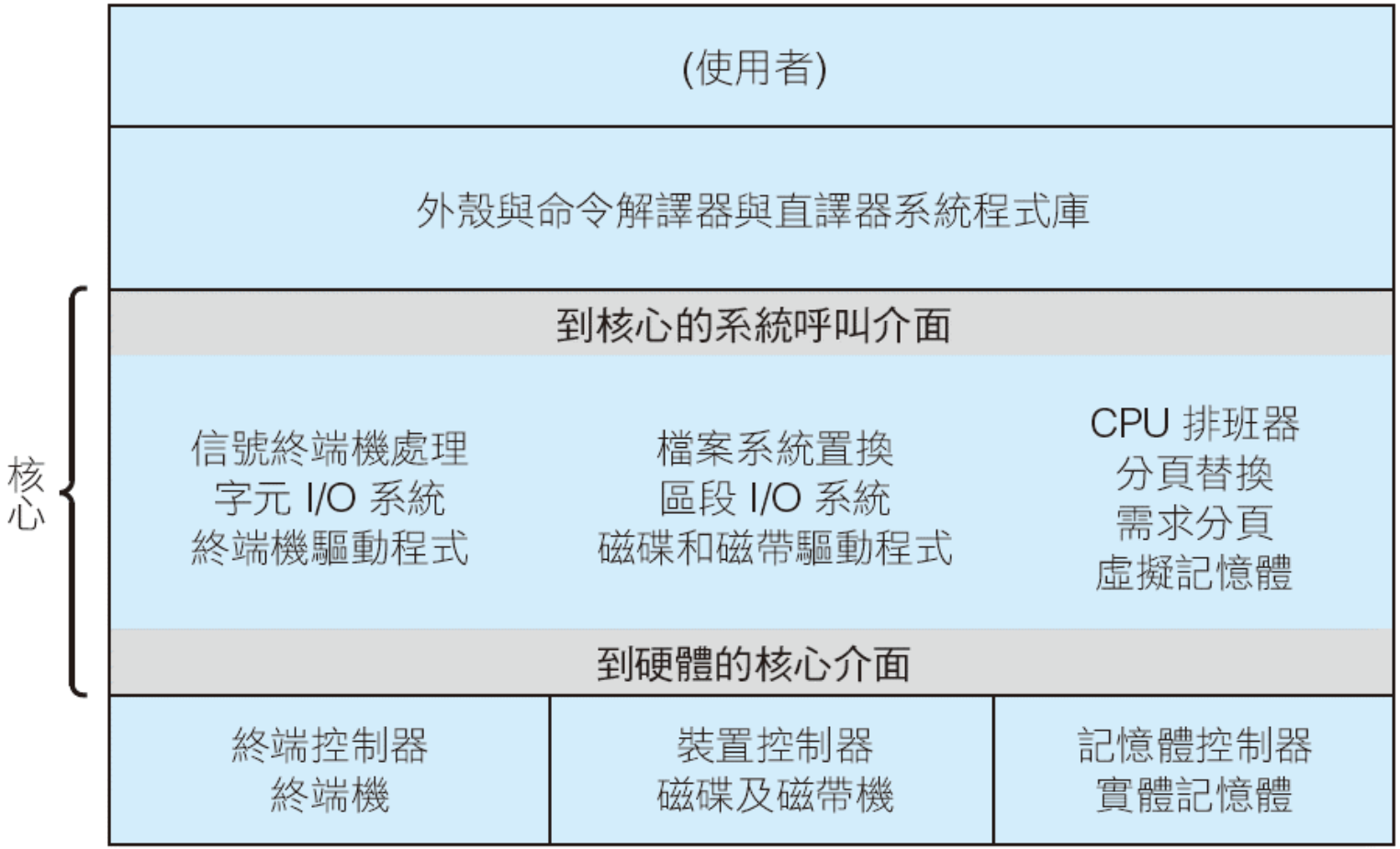
2.8 作業系統結構

- 一個像現代作業系統一樣大而複雜的系統，假如要有合適的功能和容易地修改，必須小心地設計
  - 2.8.1 單一結構
  - 2.8.2 分層方法
  - 2.8.3 微核心

2.8.1 單一結構

- 原始 UNIX 作業系統是這種有限架構的一個範例，它由兩個可分離的部分組成：
  - 核心
  - 系統程式
- 核心可以區分為一系列介面和裝置驅動程式，隨著 UNIX 的發展，這些驅動程式已經被擴展。如圖 2.12 所示
  - 我們可以將傳統的 UNIX 作業系統視為某種程度的分層
  - 一切都在系統呼叫介面之下，而實體硬體之上層是核心
  - 核心提供檔案系統、CPU 排程、記憶體管理以及通過系統呼叫確定的其它作業系統功能
  - 綜上所述，這是將大量功能組合到一個位址空間中的功能

圖 2.12　傳統的 UNIX 系統結構



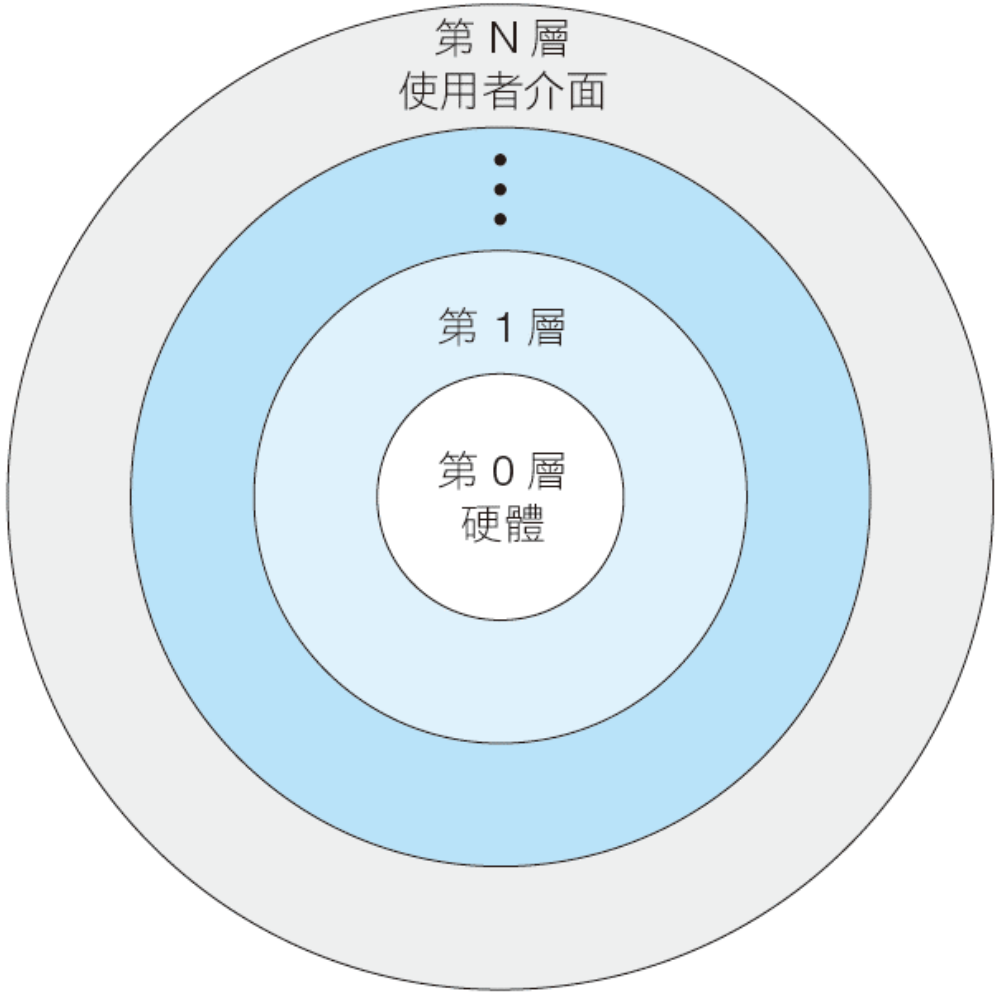
2.8.2 分層方法

- 作業系統層是一種抽象物件的實現，它負責資料和處理資料的操作。典型作業系統層
  - 例如，第 M 層
    - 它包括一些資料結構和一組可讓較高層次呼叫的一組函數
    - 相同地，第 M 層也可以呼叫較低層次的操作
- 分層方式最主要的好處是結構簡單和除錯
  - 層次選定之後，每一個層次只能使用較低層的功能與服務
  - 這種方式更容易做系統的除錯與驗證

2.8.2 分層方法

- 第一層改正時，不必考慮系統的其餘部份，因為根據定義，它只用了基礎的硬體 (假設它是正確的) 去完成它的功能
- 一旦第一層改正後，在第二層的工作可以假設它的功能正確
- 在特定層次除錯時，如果發現錯誤，我們可以知道錯誤必定在那一層，因為在它底下的層都已改正
- 如此一來，當系統分層時，系統的設計與製作都可以簡化

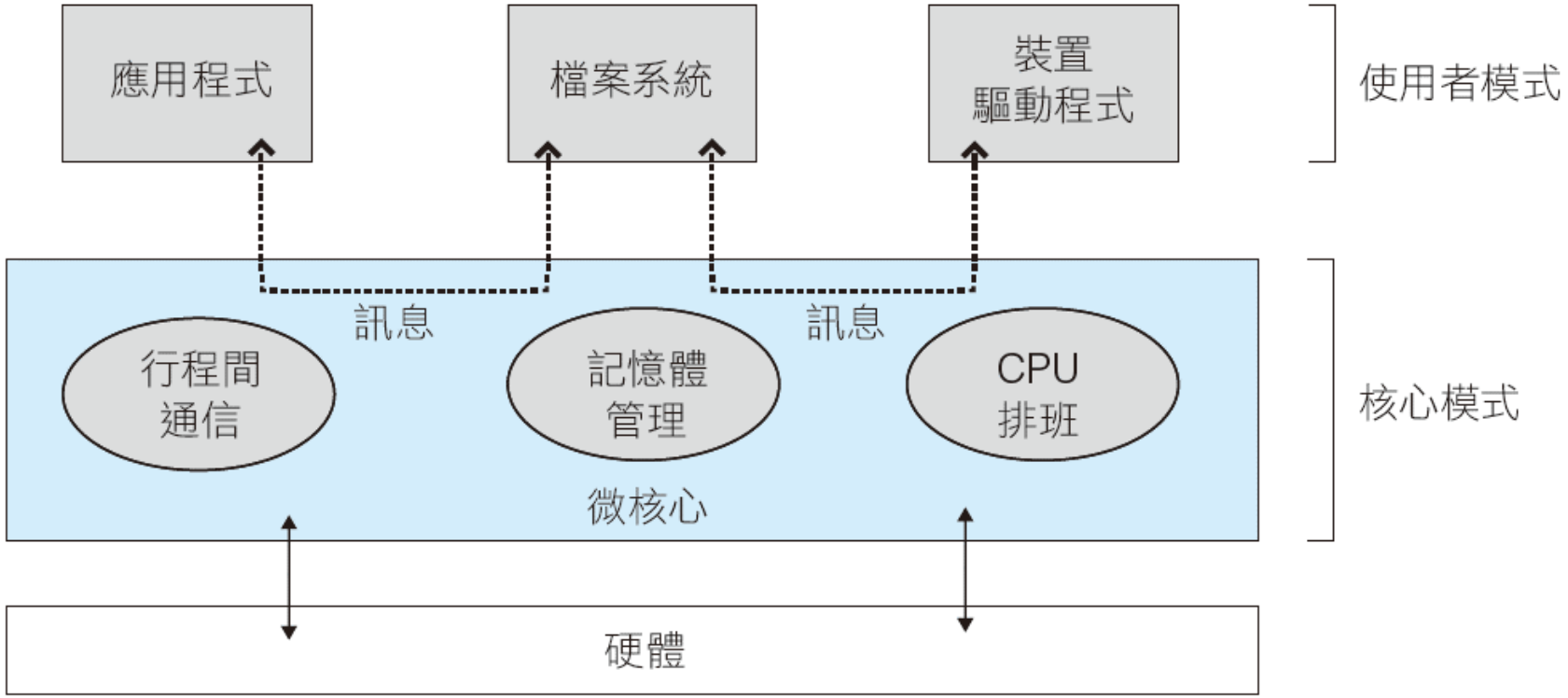
圖 2.14　分層作業系統



2.8.3 微核心

- 至於哪一部份應該留在核心，哪些應該在使用者空間製作，則沒有一致的意見
- 經由**訊息傳遞** (message passing) 提供通信
- 最著名的微核心作業系統可能是達爾文 (Darwin)
  - 它是 macOS 和 iOS 作業系統的核心元件

圖 2.15 典型微核心的架構



2.8.4 模 組

- 也許目前作業系統最好的設計方法是使用**可載入的核心模組** (loadable kernel module, LKM)
  - 在這裡，核心有一組主要的元件，而且在啟動時間或執行時間動態地連接額外服務
  - 這種使用動態可載入模組的策略，在現行 UNIX 製作中是相當常見的，如
    - Linux
    - macOS
    - Solaris
    - Windows

2.9 構建和啟動作業系統

- 為一種特定的機器配置專門設計、程式碼和執行一個作業系統是有可能的
  - 但更常見的是，作業系統被設計為在具有各種外圍配置的任何一類機器上運行
    - 2.9.1 作業系統產生
    - 2.9.2 系統啟動

2.9.1 作業系統產生

- 如果要從頭開始建構作業系統，則必須依循以下步驟：
  1. 撰寫作業系統原始碼 (或獲取先前編寫的原始碼)
  2. 為將在其上執行的系統配置作業系統
  3. 編譯作業系統
  4. 安裝作業系統
  5. 初始化電腦及其新的作業系統

2.9.1 作業系統產生

- 在一種極端情況下，系統管理員可以使用它來修改作業系統原始碼的副本
  - 然後將作業系統完全編譯
    - 稱為**系統構建** (system build)
- 另一個極端的作法，可以完全用模組來構成一個系統
  - 此情況下，在執行的時候才決定部份程式的取捨，而不是在編譯或鏈結時
  - 系統生成時所涉及的只是設置各參與描述系統配置

2.9.1 作業系統產生

- 這些方法之中最主要的差別是產生系統的大小與一般性，以及在硬體架構改變時修改系統的難易程度
  - 對於嵌入式系統，採用第一種作法並為特定的靜態硬體配置生成作業系統並不罕見
  - 但是，大多數支持桌上型電腦和筆記型電腦以及移動裝置的現代作業系統都採用了第二種作法

2.9.2 系統啟動

- 通過載入核心啟動電腦的過程稱為啟動系統
- 在大多數系統中啟動過程如下：
  1. 一小段稱為**啟動程式** (bootstrap program) 或**啟動載入器** (boot loader) 的核心定位程式碼
  2. 將核心載入到記憶體中並啟動
  3. 核心初始化硬體
  4. 掛載了根檔案系統

2.9.2 系統啟動

- 某些電腦系統使用多階段啟動過程：
  - 首次打開電腦電源時，將運行位於稱為 **BIOS** 的非揮發性記憶體中的小型啟動載入程式
  - 通常此初始啟動載入程式只執行其它操作，而僅載入第二個啟動載入程式
  - 該第二個啟動載入程式位於稱為**啟動區塊** (boot block) 的固定磁碟位置

2.9.2 系統啟動

- 最近的許多電腦系統已用 **UEFI** (統一可擴展硬體介面) 代替了基於 BIOS 的啟動行程
- 它遲早會啟動作業系統並掛載根檔案系統
  - 只有在此動作後，系統才真正**執行** (running)
- **GRUB** 是一個用於 Linux 和 UNIX 系統的開源啟動程式
  - 系統的啟動參數在 GRUB 檔案配置中設置，該檔案在啟動時載入

2.9.2 系統啟動

- 適用於大多數作業系統的啟動載入程式可啟動**恢復模式** (recovery mode) 或**單使用者模式** (single-user mode)
- 以診斷硬體問題、修復損壞的檔案系統，甚至重新安裝作業系統

2.10 作業系統除錯

- **除錯** (debugging) 是發現和修正系統中硬體與軟體錯誤的活動
  - 也包括**性能調整** (performance tuning)
  - 藉由移除發生在系統中處理的**瓶頸** (bottleneck) 來尋求性能改善

2.10.1 失敗分析

- 如果一個行程失敗了，大部份的作業系統會編寫錯誤資訊到一個**記錄檔案** (log file) 來警示系統操作者或使用者問題已經發生
- 作業系統也可以採取**核心傾印** (core dump)
  - 捕捉行程的記憶體
  - 以及儲存在檔案做後續的分析 (記憶體在早期的運算被稱為“核心”)

2.10.1 失敗分析

- 核心錯誤被稱做**當機** (crash)
  - 當一個行程發生錯誤，錯誤資訊會被儲存到一個記錄檔案
  - 同時記憶體狀態被儲存到**當機轉儲** (crash dump)