作業系統：Overview

**目錄**

[作業系統：Overview 1](#_Toc204439022)

[1. 電腦系統組織 1](#_Toc204439023)

[2. 作業系統的角色 2](#_Toc204439024)

[3. 不同角色對OS的觀點 2](#_Toc204439025)

[4. 作業系統提供的服務(Operating-System Services) 3](#_Toc204439026)

[5. 核心(kernel) 3](#_Toc204439027)

[6. Polling（輪詢） 4](#_Toc204439028)

[7. 中斷(Interrupts) 4](#_Toc204439029)

[7.1 以I/O system來看Interrupts(Chapter 12) 4](#_Toc204439030)

[7.2 中斷向量表(Interrupt Vector Table) 5](#_Toc204439031)

[8. 系統呼叫（System Calls） 6](#_Toc204439032)

[9. Direct Memory Access(DMA) 7](#_Toc204439033)

[10. Multiprogramming & Multitasking 8](#_Toc204439034)

[11. 電腦系統架構(Computer-System Architecture) 8](#_Toc204439035)

[11.1 Symmetric Multiprocessing(SMP 對稱多處理) 9](#_Toc204439036)

[11.2 Multicore Systems(多核心系統) 9](#_Toc204439037)

[11.3 NUMA(非一致記憶體存取) 9](#_Toc204439038)

[11.4 叢集系統(Clustered Systems) 10](#_Toc204439039)

[12. 開機流程(Bootstrap Process) 11](#_Toc204439040)

[13. 雙模式(Dual-Mode) 11](#_Toc204439041)

[14. 計時器(Timer) 12](#_Toc204439042)

[15. Linker & Loader 13](#_Toc204439043)

[16. I/O Systems 13](#_Toc204439044)

[17. I/O Hardware 13](#_Toc204439045)

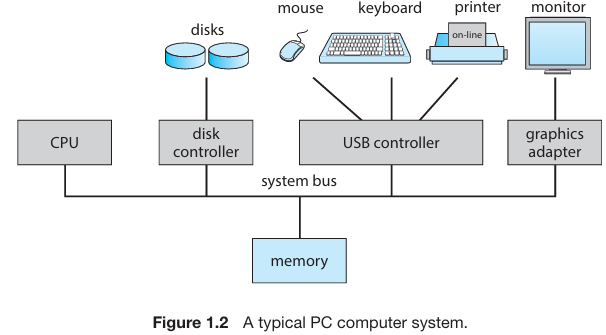
[17.1 記憶體對應 I/O(Memory-Mapped I/O) 14](#_Toc204439046)

# 電腦系統組織

現代電腦系統主要涵蓋一個或多個 CPU，多個裝置控制器(device controller)，並透過系統匯流排(bus)連接彼此與記憶體(memory)。每個controller責一類裝置(如：磁碟、音訊、圖形)，有些控制器可同時連接多個裝置(例如 USB hub)。

這些裝置控制器(device controller)內部會有緩衝區(local buffer)與特殊暫存器(registers)，負責與其裝置間的資料搬移。而作業系統為每個控制器提供驅動程式(device driver)，讓OS能以一致的方式管理不同裝置。

Figure 1：典型的PC架構

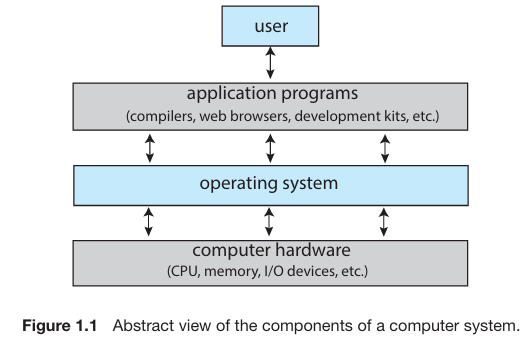


# 作業系統的角色

計算機系統可以粗略分為四個部分：

* 硬體(hardware)：CPU、記憶體與 I/O 裝置，提供資源。
* 作業系統(operating system)：協調硬體與應用程式的使用。
* 應用程式(application programs)：如文字處理器、編譯器、網頁瀏覽器等。
* 使用者(user)：最終使用者。

Figure 2：計算機系統的組成



# 不同角色對OS的觀點

對一般使用者(User View)而言，作業系統就是操作電腦的介面，例如透過螢幕、鍵盤與滑鼠來執行應用程式。大多數使用者**不會關心**系統資源如何被共享或分配。因此，作業系統的**設計重點**在於提升使用便利性(ease of use)，接著才考量效能(performance)與安全性(security)，

對於系統觀點(System View)，作業系統是**最貼近硬體的程式**管理並分配各種硬體資源，其角色可分為兩大部分：

1. 資源分配者(Resource Allocator)：OS管理並分配各種硬體資源，例如CPU 時間、記憶體、儲存空間、I/O 裝置，目標是追求資源使用的**效率(efficiency)與公平性(fairness)。**
2. 控制程式(Control Program)：OS控制使用者程式的執行，防止錯誤行為與不當資源使用。尤其是對 **I/O 裝置**的直接操作，需特別加以管理與保護。

# 作業系統提供的服務(Operating-System Services)

OS的主要目的之一，是提供一個方便且高效的環境讓程式得以執行。OS對「使用者」與「程式」都提供一組共通的基本服務。這些服務可以分為「幫助使用者與應用程式的服務」與「幫助系統自身運作效率的服務」：

Table 1：幫助使用者與應用程式的服務

|  |  |
| --- | --- |
| 服務名稱 | **說明** |
| 使用者介面 (User Interface) | 提供 GUI(圖形介面)、CLI(命令列介面)、或觸控式介面，讓使用者與系統互動。 |
| 程式執行 (Program Execution) | 載入程式到記憶體並執行它。若程式執行完成或錯誤終止，OS要能處理結束程序。 |
| 輸入/輸出操作 (I/O Operations) | 管理所有 I/O 裝置。由於使用者無法直接操作硬體，因此OS必須提供中介方法來進行 I/O。 |
| 檔案系統操作 (File-System Manipulation) | 提供建立、讀寫、搜尋、刪除檔案與資料夾的功能。也包含權限控管(誰可以存取哪個檔案)。 |
| 程式間通訊 (Communications) | 同一台機器裡讓不同程序之間能夠溝通，可透過「共享記憶體」或「訊息傳遞(message passing)」來完成。 |
| 錯誤偵測 (Error Detection) | 作業系統要能持續監控錯誤，包括硬體錯誤(如記憶體或網路)、I/O 錯誤(如印表機沒紙)、或程式錯誤(如存取非法記憶體)。有時必須終止程式，有時可回傳錯誤代碼讓程式自行處理。 |

Table 2：幫助系統自身運作效率的服務

|  |  |
| --- | --- |
| 服務名稱 | 說明 |
| 資源分配 (Resource Allocation) | 當多個程序同時執行，系統要分配 CPU、記憶體、磁碟等資源。使用排程演算法(如 CPU scheduling)與資源管理方法。 |
| 使用記錄與帳務 (Accounting) | 系統紀錄每個程式使用了多少資源，例如 CPU 時間、記憶體、磁碟空間等，用來做帳務分析或統計。 |
| 保護與安全 (Protection and Security) | 確保不同程序間無法任意干擾彼此，也防止非法使用者入侵系統。包括帳號密碼登入、權限控制、防火牆等。 |

# 核心(kernel)

核心（Kernel）是作業系統中最重要、永遠在運作的核心程式，就像是作業系統的大腦，負責管理整台電腦的資源和行程。具體來說，Kernel要負責：

* 📦 管理記憶體（Memory）
* 🧮 分配 CPU 資源（CPU scheduling）
* 📁 控制檔案系統與資料存取
* 🛜 負責輸入/輸出裝置的協調（像鍵盤、滑鼠、印表機）
* 🧑‍💻 負責多個程式同時執行的管理（Multitasking）

Table 3：核心與其他程式的關係

|  |  |  |
| --- | --- | --- |
| 類型 | 說明 | 舉例 |
| 系統程式（System Programs） | 幫助使用者與作業系統互動，通常提供工具與介面，但不一定屬於核心的一部分。 | 指令列（Command Line）、設定工具、檔案總管 |
| 應用程式（Application Programs） | 使用者執行的程式，與作業系統無直接關係，但需要經過核心協助才能使用電腦資源。 | Word、Chrome、遊戲、音樂播放器 |

* 系統程式(system programs)：輔助作業系統，但不一定屬於核心。
* 應用程式(application programs)：用戶運行的程式，不參與作業系統運作。

# Polling（輪詢）

在電腦與外部裝置（像印表機、硬碟）進行資料交換時，Polling（輪詢）是一種最基本的方式。可以把它想像成：CPU 就像一位不斷敲門問「你好了沒？」的使用者。

Polling 是由 CPU 主動、不斷地詢問裝置是否準備好傳送或接收資料的一種通訊方法。雖然它簡單易實作，但因為要不斷「確認狀態」，會浪費大量 CPU 資源，導致其他工作延遲。大致步驟如下：

1. 主機輪詢 busy bit（狀態暫存器），直到裝置準備好
2. 寫入資料 到 Data-out
3. 設定 command-ready
4. 裝置控制器接收命令 → 執行 → 清除 busy

# 中斷(Interrupts)

了解完Polling，就可以來看看Interrupt，中斷(Interrupt)是CPU 與外部裝置之間通訊的重要機制，可讓系統即時回應硬體事件，而非不斷輪詢(polling)浪費效能。基本流程如下：

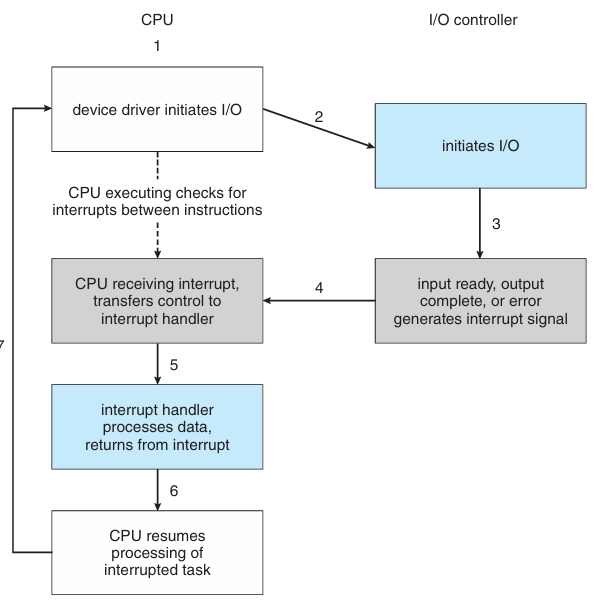
1. CPU 執行使用者程式
2. 裝置控制器完成任務後發出中斷訊號(interrupt signal)
3. CPU 接收到中斷後，暫停目前執行的程式，並根據中斷向量表(Interrupt Vector Table)跳到對應的中斷服務程式(ISR)
4. ISR(Interrupt Service Routine) 處理完事件後，還原先前 CPU 狀態，繼續執行原本的程式

## 以I/O system來看Interrupts(Chapter 12)

那如果以I/O system來看Interrupts(這邊是恐龍書Chapter 12)，就會是：

1. CPU 啟動 I/O
2. 裝置完成後產生 中斷信號（interrupt request）
3. CPU 儲存狀態 → 跳轉至中斷處理程序（ISR）
4. 處理完後返回原本任務

Figure 3：Interrupt-driven I/O cycle



## 中斷向量表(Interrupt Vector Table)

上述在提中斷(Interrupts)的過程中，當裝置發生中段信號，CPU便會儲存當前狀態，進而去處理中斷處理程序（ISR）。而存放中斷處理程序（ISR）得地方就是中斷向量表(Interrupt Vector Table)。

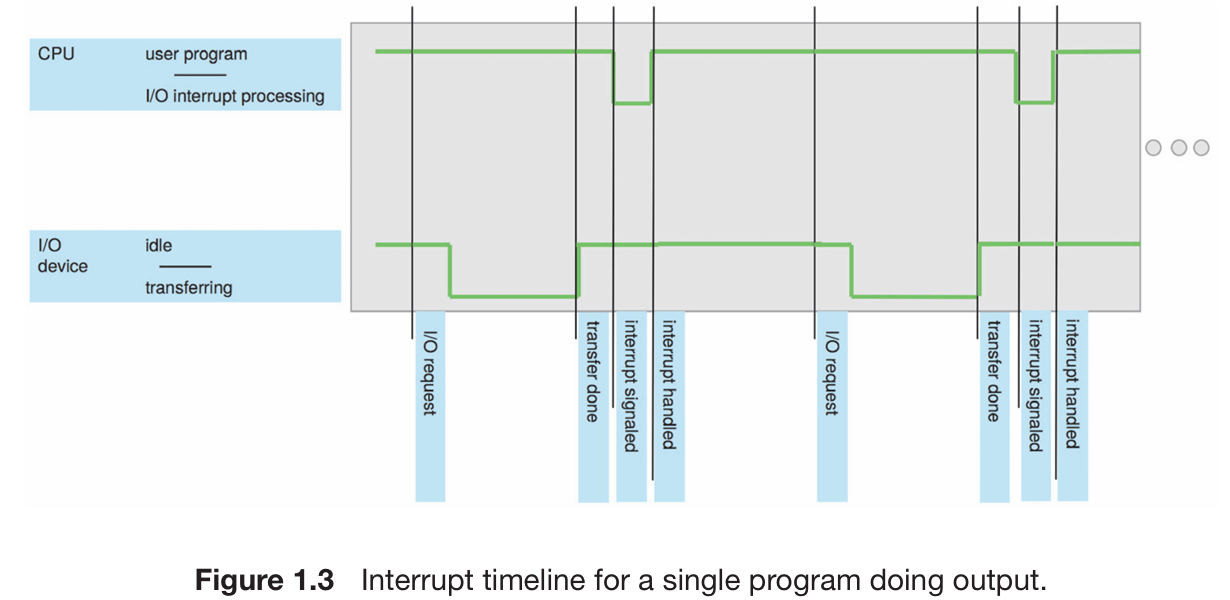
他是一個指標陣列，能讓CPU能根據中斷編號快速跳轉處理。中斷類型可以分為：

就是存放ISR的地方（是一個指標陣列），讓CPU能根據中斷編號快速跳轉處理。中斷類型可以分為：

* Non-maskable：不可屏蔽中斷，無法被忽略，例如硬體錯誤
* Maskable：可屏蔽中斷，可由軟體設定是否暫時忽略

而當多個裝置共用同一條中斷線時，便可以使用中斷連鎖機制(Interrupt Chaining)，即一個 ISR 執行後，主動查詢其他可能發出中斷的裝置，依序處理。

Figure 4：中斷處理的基本流程圖(Interrupt Handling Flow)



# 系統呼叫（System Calls）

在 OS 中，幾乎所有重要事件都是透過「中斷（Interrupt）」來通知作業系統的。中斷有三種常見來源：

* 硬體中斷 (Hardware Interrupt)：裝置主動通知 CPU。例如滑鼠移動、磁碟完成寫入。
* 陷阱/例外 (Trap / Exception)：程式發生錯誤，系統介入。例如除以零、非法存取記憶體。
* 軟體中斷 (Software Interrupt)(也就是System Call（系統呼叫）)：程式主動請求系統幫忙做特權操作

當我們在程式裡執行像是 read()、write()、open() 這些指令時，其實並不是直接操作硬體，而是透過一種特別的方式「請求作業系統幫忙」，這個動作就叫做系統呼叫（System Call）。開發者寫程式通常使用的是 API（Application Programming Interface），例如read()。而這其實是「包裝好」的 system call，由作業系統提供的函式庫（如 Linux 的 libc）實作。優點是跨平台、簡單好寫，不用煩惱細節。

Figure 5：呼叫read時的System Calls邏輯

一張含有 文字, 螢幕擷取畫面, 圖表, 繪圖 的圖片

自動產生的描述

Table 4： System Calls 種類

|  |  |  |
| --- | --- | --- |
| **分類** | **常見功能** | **範例 system call** |
| Process Control | 建立、終止程序、等待、記憶體分配等 | fork(), exec(), exit() |
| File Management | 建立、開啟、讀寫、關閉檔案 | open(), read(), write() |
| Device Management | 請求/釋放裝置、讀寫、移動資料 | ioctl(), read(), write() |
| Information | 時間、日期、系統資訊、屬性查詢與設定 | gettimeofday(), getpid() |
| Communication | 進程之間的訊息交換（Message/Shared memory） | pipe(), shm\_open() |
| Protection | 設定/查詢權限、限制資源使用 | chmod(), umask() |

對於System Call的參數傳遞形式，如果是少量參數就直接用佔存器；如果參數較多，就使用stack(推入資料)或記憶體區塊(把所有參數放進記憶體，然後只傳這個位置)。

**備住**：Linux 結合以上兩種：少的用暫存器，多的就用 block。

Table 5：系統呼叫(System Call) vs. 中斷(Interrupt)

|  |  |  |
| --- | --- | --- |
| 特徵 | 系統呼叫(System Call) | 中斷(Interrupt) |
| 誰觸發 | 程式主動請求 | 硬體或裝置主動發出 |
| 時機 | 程式需要作業系統幫忙時 | 例如鍵盤輸入、網路封包到達時 |
| 控制權轉移 | 使用者程式請求進入核心 | 當前執行單元被「打斷」進入核心 |
| 是否可預期 | 可預期 | 多半不可預期 |
| 範例 | read(), write(), open() | 鍵盤輸入中斷、計時器中斷 |

# Direct Memory Access(DMA)

在電腦系統中，資料從裝置傳到記憶體（或反之）是很常見的動作，例如從硬碟載入影片、從 USB 拷貝照片。但這些「搬資料的工作」如果全都要經過 CPU，會發生什麼事呢？

在傳統的 I/O 模型中，資料傳輸是由 CPU 控制的：

1. CPU 從裝置讀取一筆資料
2. 再將資料寫入主記憶體
3. 每筆資料都要經過 CPU 操作（造成 overhead）(每一筆資料都要經過 CPU 手動「搬來搬去」)

這樣這種方式在處理大量資料（如硬碟檔案、音訊串流）時效率低落。CPU 資源被浪費在搬運資料上，而無法專注處理其他運算任務。

而Direct Memory Access（DMA） 是一種由硬體控制器（稱為 DMA 控制器）所主導的資料傳輸機制。它允許 I/O 裝置（如硬碟、音效卡、網卡）直接與主記憶體（RAM）進行資料傳輸，跳過中央處理器（CPU） 的介入，以節省 CPU 資源並提高效能。主機只需要事先提供這三個資訊：

* 來源/目的的記憶體位址
* 資料長度（要傳多少）
* 指令區塊（Command Block）：用來描述任務的控制結構

DMA步驟大致如下：

1. CPU 下命令：告訴「裝置驅動程式」（device driver），想把資料搬到記憶體某個位置

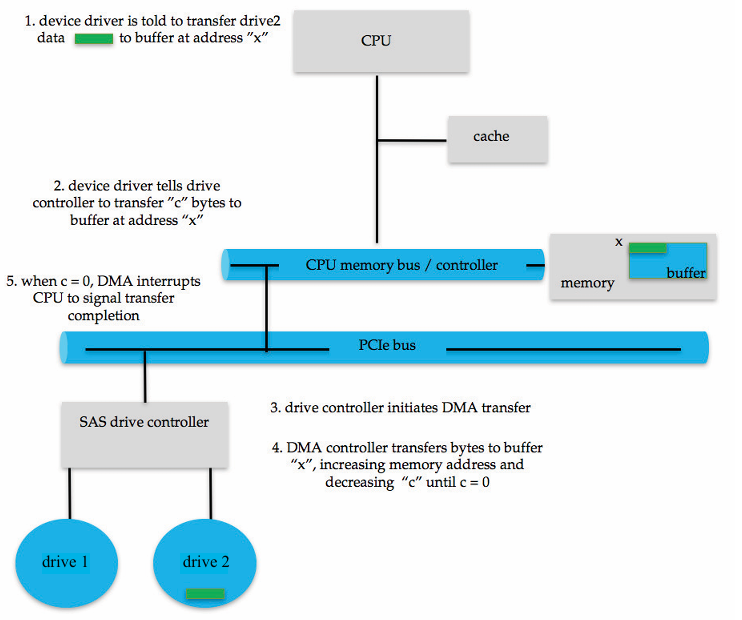
2. 驅動程式轉達詳細任務：轉給控制器（例如磁碟控制器），說：「請把 c bytes 的資料搬到記憶體 x」

3. DMA 控制器接手

* 一筆一筆將資料搬到記憶體 x
* 每搬完一筆就更新位置（x++）
* 同時遞減剩下的資料量（c--）

4. 搬完後，通知 CPU（透過中斷）：「老闆，我搬好了，來驗收吧！」

Figure 6：Steps in a DMA transfer.



# Multiprogramming & Multitasking

* Multiprogramming：多個程式在記憶體，CPU 一次執行一個，等待時切換
* Multitasking：Multiprogramming + 快速切換

# 電腦系統架構(Computer-System Architecture)

現代電腦系統可能只用一個處理器，也可能有上百個處理器，設計和操作系統的支援方式會有所不同。早期電腦大多屬於**單處理器系統(Single-Processor Systems)**，一個 CPU 核心(core) 執行所有指令與處理所有任務。而現代，**多處理器系統(Multiprocessor Systems)**則成為現代標準配置。**多處理器系統(Multiprocessor Systems)**可以進一步細分：

## Symmetric Multiprocessing(SMP 對稱多處理)

所有 CPU 共用主記憶體與系統匯流排。每個 CPU 都是平等的，可以處理作業系統和應用程式。優點是 運行效率高，N 顆核心最多可同時跑 N 個程序。而缺點是當CPU太多時，彼此會爭搶匯流排，導致效能下降。

## Multicore Systems(多核心系統)

一顆實體晶片上面有多個核心(cores)，每個核心有自己的暫存器與 L1 cache，並共享 L2 cache。核心之間溝通快、耗能低，是現今最常見架構(如手機、筆電)。

## NUMA(非一致記憶體存取)

每個 CPU有自己的本地記憶體，彼此透過系統互連線互通資料，共享記憶體空間。適合大量 CPU 的擴充(scalability)（多用於伺服器）。若 CPU0 存取 CPU3 的記憶體，會比較慢(有延遲 latency)→ 需做「CPU 排程」與「記憶體分配」優化來避免延遲

Figure 7：對稱式多處理架構(Symmetric Multiprocessing, SMP)

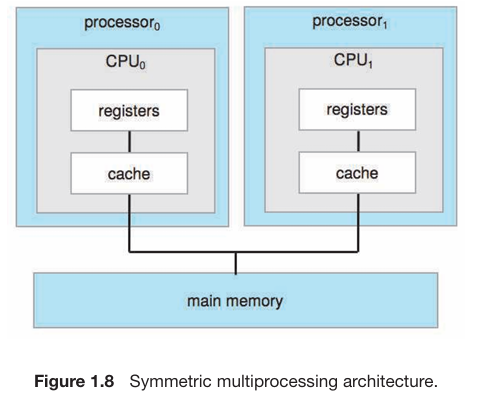


Figure 8：雙核心設計：兩個核心位於同一顆晶片上

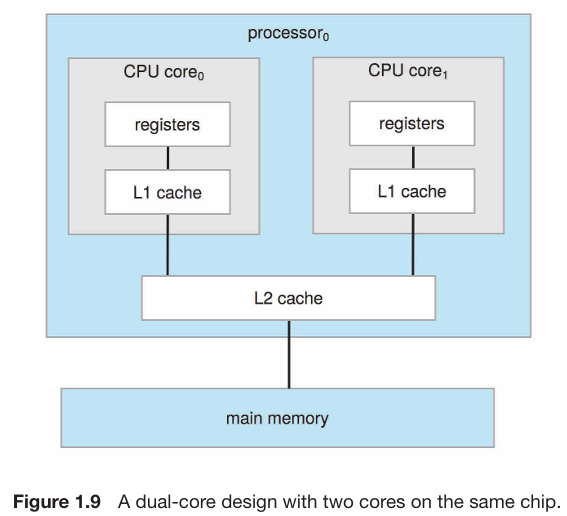
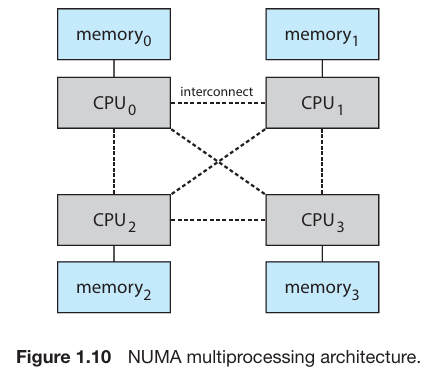


Figure 9：NUMA(非一致性記憶體存取)多處理架構



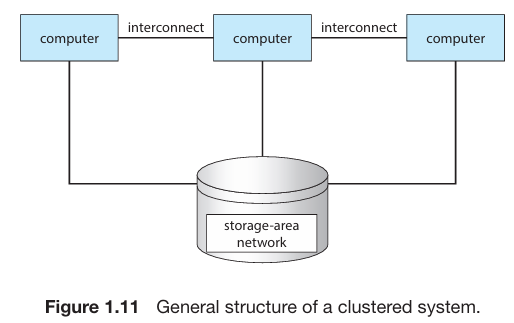
## 叢集系統(Clustered Systems)

多台電腦(節點)組成一個群組，透過網路互連(如 LAN 或 InfiniBand)。每台電腦是完整的系統，可有多核心。

Table 6：Clustered Systems的功能

|  |  |
| --- | --- |
| 高可用性 (High Availability) | 若有一台電腦掛掉，可以由另一台接管。又稱熱備援(hot standby)/ 對等備援(symmetric)。 |
| 高效能 (High Performance Computing) | 每個核心有自己的暫存器與 L1 cache，並共享 L2 cache。核心之間溝通快、耗能低， |
| 共享儲存 (Shared Storage) | 彼此之間採用SAN(Storage Area Network)共用資料。系統則需使用「分散式鎖定管理(Distributed Lock Manager, DLM)」來避免資料衝突。 |

Figure 10：叢集系統的一般架構



# 開機流程(Bootstrap Process)

電腦開機時，會執行bootstrap program(啟動程式)，bootstrap program會儲存在 firmware(如 ROM)，負責初始化 CPU、記憶體與裝置控制器，然後載入作業系統核心(kernel)。Kernel 開始執行後，會載入系統程式(稱為 daemon，常駐程式)，例如Linux 中第一個系統程式是 systemd。系統完成開機後，就等待「事件」(例如：鍵盤輸入、網路請求)。

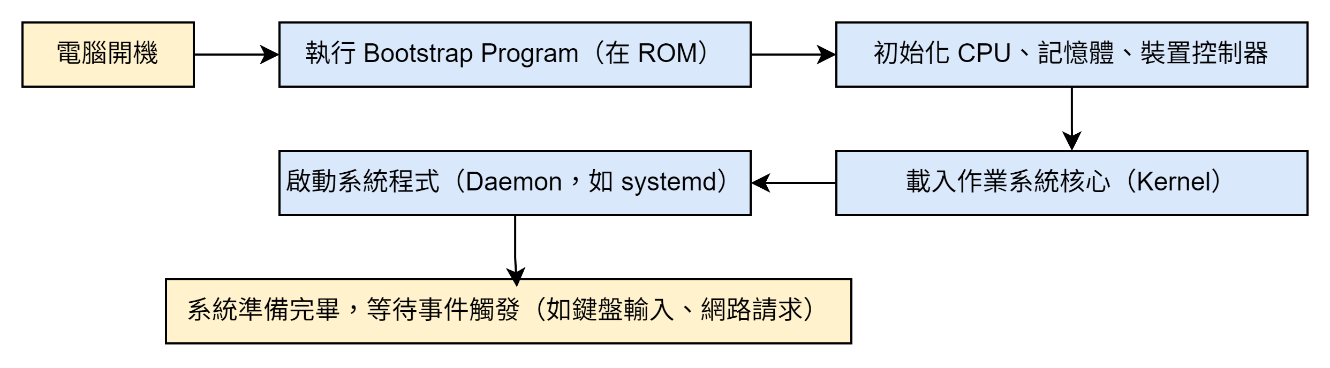
當我們對電腦開機後，硬體會執行bootstrap program(啟動程式)，bootstrap program(啟動程式)包含：

* 初始化 CPU（讓它有邏輯能思考）
* 設定記憶體
* 啟動裝置控制器（像硬碟、滑鼠、螢幕）
* (最重要)把作業系統（Kernel）從硬碟載入記憶體中

然後由 Kernel 正式接手，開始管理整台系統。而當 Kernel 成功載入後，它會開始做：

* 啟動系統背景程式（稱為 daemon，意思是常駐服務）(例如 Linux 系統中最早啟動的是 systemd)
* 設定好網路、檔案系統、使用者管理等
* 載入登入介面、桌面環境
* 最後，系統進入等待事件（event-driven）模式(等待user開始下指令)

Figure 11：開機流程



# 雙模式(Dual-Mode)

為了防止「惡意或寫壞的程式破壞系統」，作業系統設計了「雙模式（Dual-Mode）」安全機制，確保危險操作只能由 OS 核心來做。Dual-Mode可以分為：

* User mode(使用者模式)：執行使用者程式，受限、無法執行危險指令
* Kernel mode(核心模式)：執行作業系統程式，有最高權限，可直接操控硬體

Figure 12：user mode與Kernel mode的交互形式

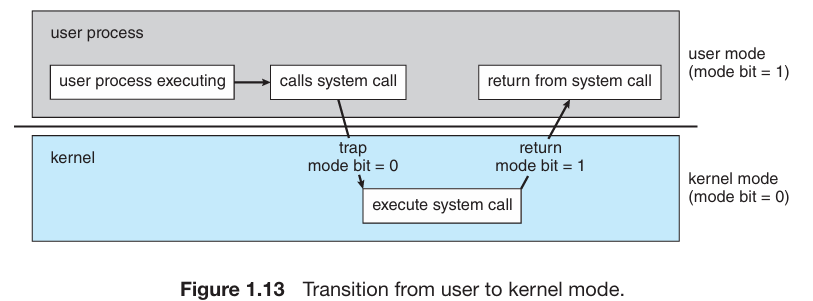
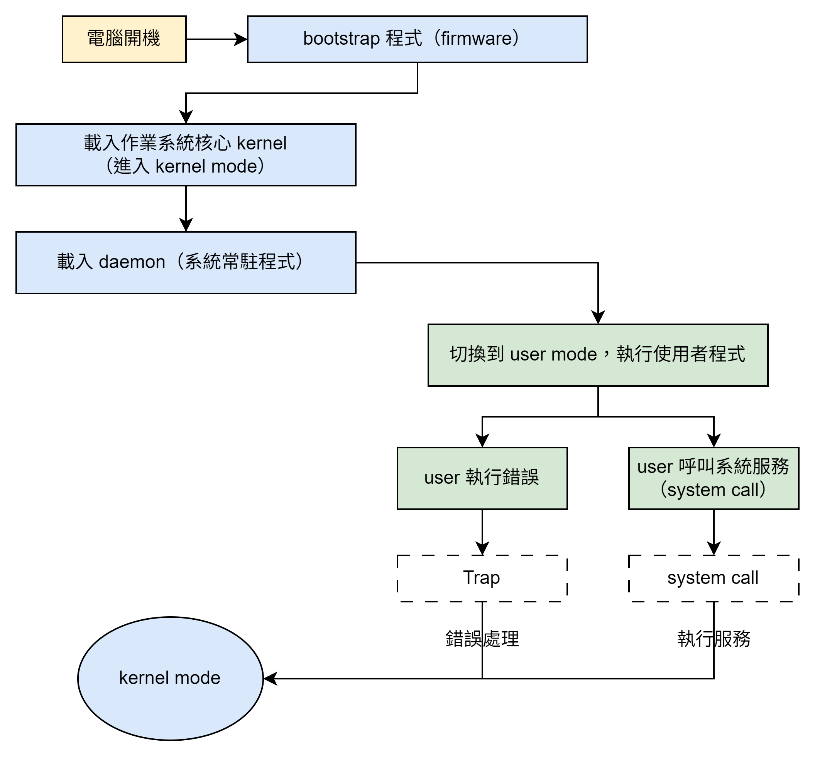


Figure 13：電腦開機到進入使用者模式



至於切換模式的方式，大概有這幾種：

* 使用者執行自己的應用程式（User Mode）
* 當需要系統幫忙（例如 read()），就會透過 System Call 做「軟體中斷」
  + CPU 進入 Kernel Mode，由 OS 幫你做這些「高危操作」
  + 執行完後，切回 User Mode 繼續

雙模式(Dual-Mode)最主要的核心是「只有作業系統可以從 User Mode 進入 Kernel Mode，且必須透過中斷機制或系統呼叫」

# 計時器(Timer)

在OS內的每樣任務，並不是都可以如期完成得。可能會有：

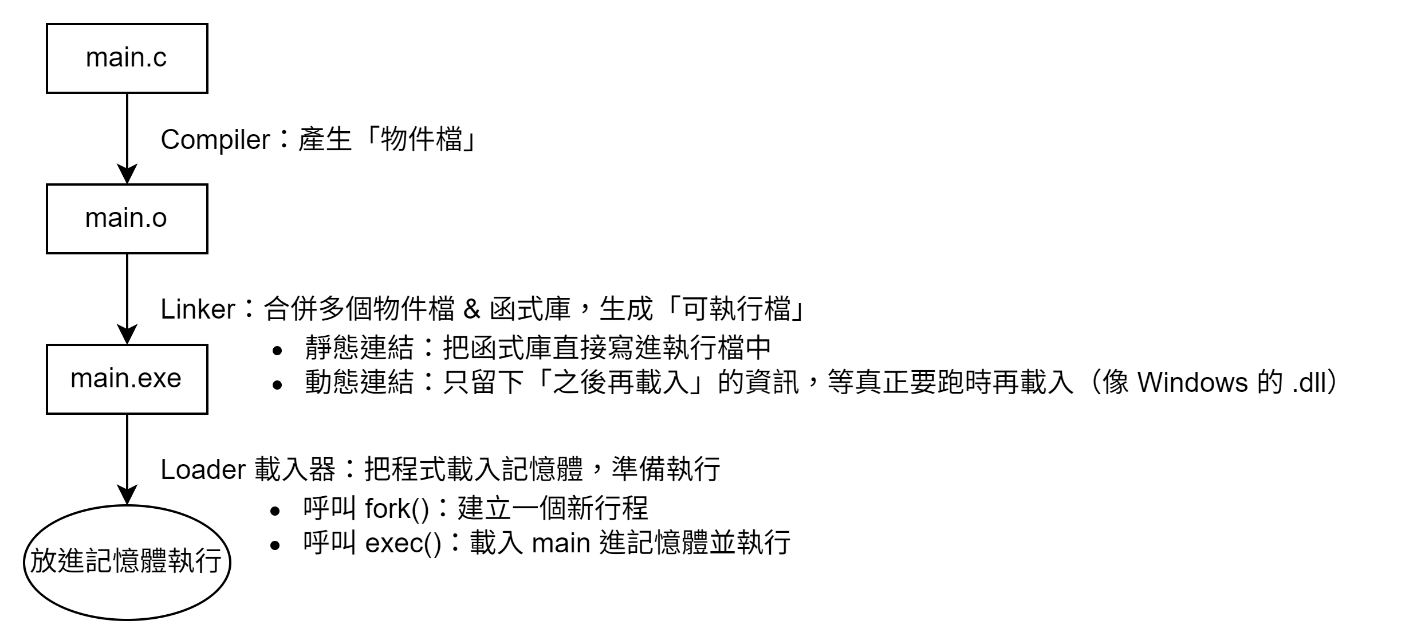
* 一直跑無限迴圈
* 佔用 CPU 太久，不讓其他程式上場
* 搶著吃資源，別人都不能用

在這樣的前提下，OS就有計時器（Timer）的機制。負責定時中斷，強迫程式交出 CPU 控制權，目的是避免程式無限迴圈或長時間佔用 CPU。

# Linker & Loader

當你寫好一支程式，例如 main.c，你不能直接執行它。它必須經過以下流程：

Figure 14：從原始碼到可執行檔的載入與執行流程



# I/O Systems

其實鍵盤，滑鼠，喇叭這類電腦周邊裝智是超級難搞的。因為每個周邊裝置都有自己對應的操作功能跟需求。但總不能叫每個開發者都懂每個裝置的通訊協定吧。因此這邊提出的解法是，OS扮演「中介」的角色。統一管理五花八門的裝置，讓開發者可以統一用read()、write() 這種簡單方法就搞定。

當前在I/O技術的兩大趨勢(彼此衝突，但也並存)：

* 標準化（Standardization）：裝置介面越來越統一，讓驅動程式能通用。
* 多樣化（Diversification）：裝置越來越奇怪，每個都想有自己的通訊方式。

作業系統的挑戰就在於：在統一的管理架構中處理日益複雜的多樣裝置。

# I/O Hardware

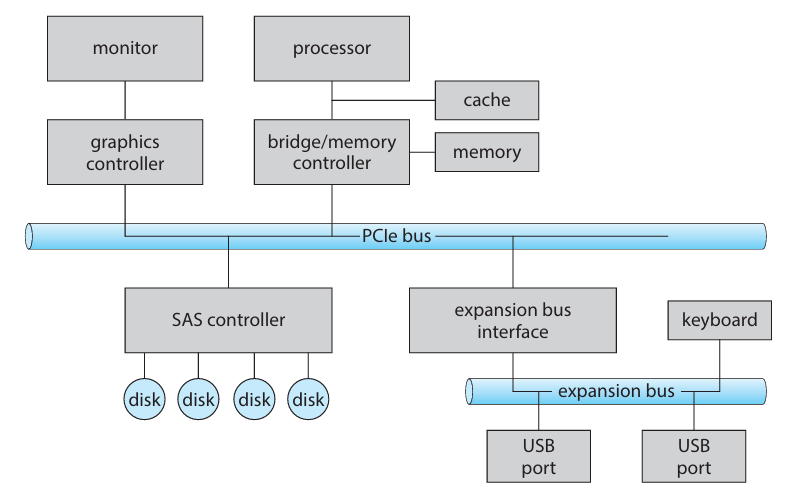
裝置透過 Port/BUS 傳送「電壓訊號」與系統互動，定義嚴格協定與傳輸時序。

* Port：早期的 I/O 通訊線（例如印表機的 LPT1）
* Bus（匯流排）：像資料的高速公路，常見如 PCIe、SATA、USB

Table 7：硬體介面與連線方式

|  |  |
| --- | --- |
| 名稱 | 說明 |
| Port（埠口） | 裝置連接點，如 USB 埠、Serial Port（串列埠） |
| Bus（匯流排） | 多裝置共用的傳輸線（如 PCIe、USB、SATA） |
| Daisy Chain | 鍊狀連接裝置（常見於 SCSI） |
| PHY (Physical Layer) | OSI 七層中的實體層，常用於資料中心術語 |

Figure 15：A typical PC bus structure



## 記憶體對應 I/O(Memory-Mapped I/O)

為了讓 CPU 跟裝置講話更方便，有一種常見設計叫做Memory-Mapped I/O：把裝置「偽裝成」記憶體的一部分。舉例來說：

* CPU 想要「檢查硬碟狀態」，就去讀某個記憶體位置（其實是對映到裝置的 status 暫存器）
* 要寫入資料到音效卡：就對某個記憶體地址「寫入資料」，裝置就會讀到。

透過這樣的方式，CPU 就可以用一般 mov 或 ld/st 指令來控制裝置，不需要額外的 I/O 指令集。

Figure 16：Device I/O port locations on PCs (partial)

