系統呼叫

大綱

- 何謂系統呼叫
- 系統呼叫與程式間的關係
- Linux作業系統中的Syscalls
- 系統呼叫的運行方式
- 系統呼叫的參數驗證
- 本章重點回顧

何謂系統呼叫

- 系統呼叫提供行程一個界面，介於硬體與用戶層之間
- 目的：
  - 它為用戶層提供了一個漠然的硬體介面
  - 系統呼叫會確定系統資源存取的安全與穩定
  - 就是於用戶層與硬體間，產生一個虛擬的系統控制行程，存取資源與釋放掉佔有的資源。

何謂系統呼叫

- 它為用戶層提供了一個漠然的硬體介面
  - 使用者不需要瞭解實際的系統資源狀態
    - 例如：儲存介面、檔案系統、網路卡晶片等硬體資源

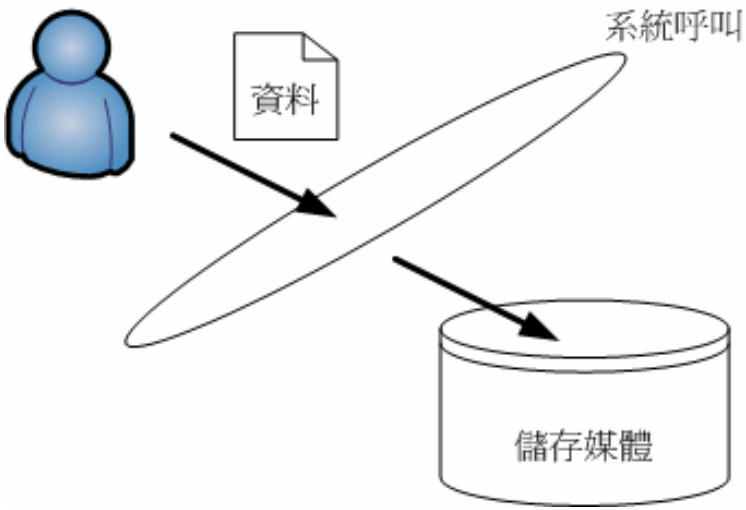
何謂系統呼叫

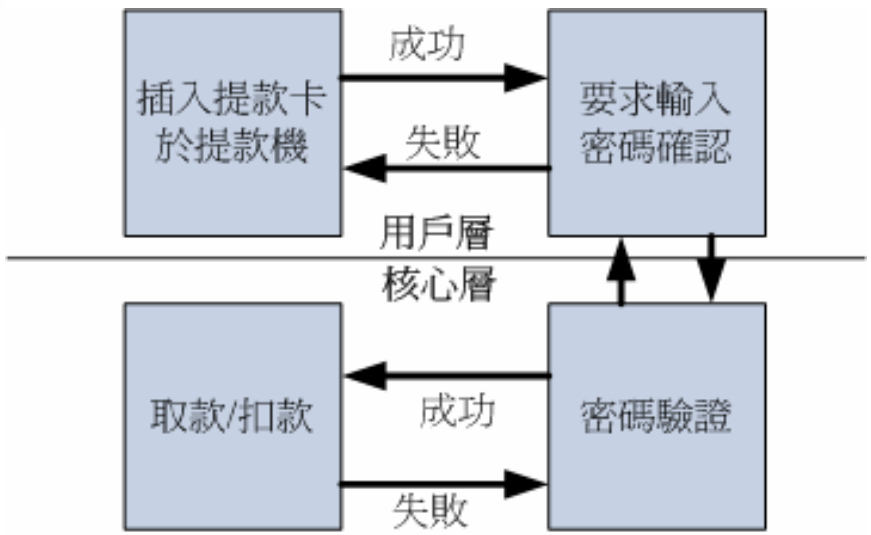
- 系統呼叫會確定系統資源存取的安全與穩定
  - 以核心作為一個硬體與程序間的仲裁角色
  - 可以整合系統的安全性
  - 減少由於程序設計錯誤導致的系統bug狀態。

何謂系統呼叫

- 於用戶層與硬體間，產生一個虛擬的系統控制行程，存取資源與釋放掉佔有的資源。
  - 由於透過核心的仲裁，可有效的管理資源
  - 達成系統的多工分配

- Linux作業系統中，行程要從用戶層進入到核心層唯一的方式，只能透過系統呼叫來達成





系統呼叫與程式間的關係

- 應用程式是透過應用程式設計介面（Application Programming Interface，API）規範所產生的，而不是指一個系統呼叫。
- 應用程式設計介面與系統呼叫間的關係標準 – POSIX
- Unix系統中有箴言「Provide mechanism, not policy」

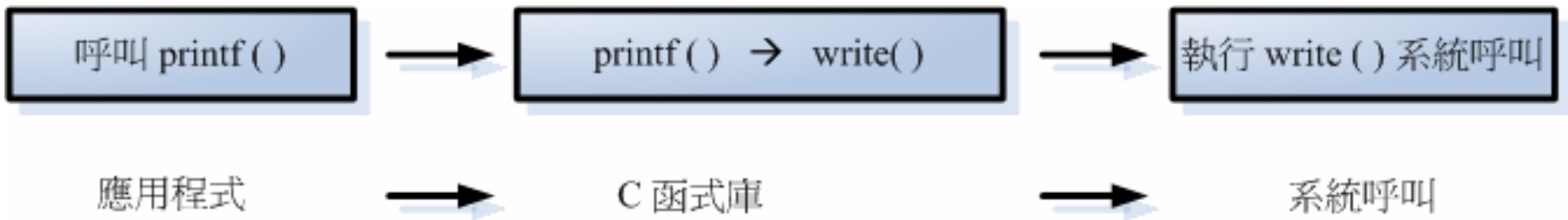
系統呼叫與程式間的關係

- Why POSIX
  - POSIX定義了核心與應用程式間的呼叫關係
  - 核心如果支援 POSIX
  - 應用程式如果支援 POSIX
  - 則應用程式理論上，可以在該核心上面運作
  - 軟體開發者不用重複撰寫多個應用程式！

系統呼叫與程式間的關係

- What is “Provide mechanism, not policy”
  - Linux 以 C 程式語言作為開發介面
  - C 程式語言提供了 C 函式庫供使用者直接引用
    - 『函式庫』就是將程式中可能常用的功能先寫成一段可呼叫的程式碼，如此未來的程式設計人員就不需要重複撰寫該段code
  - C函式庫又分為兩大類：
    - 標準 C 函式庫：常用於程式設計中
    - 核心系統呼叫介面：用於核心呼叫
  - 程式設計師只要搞動標準C函式庫即可，減輕開發陣痛

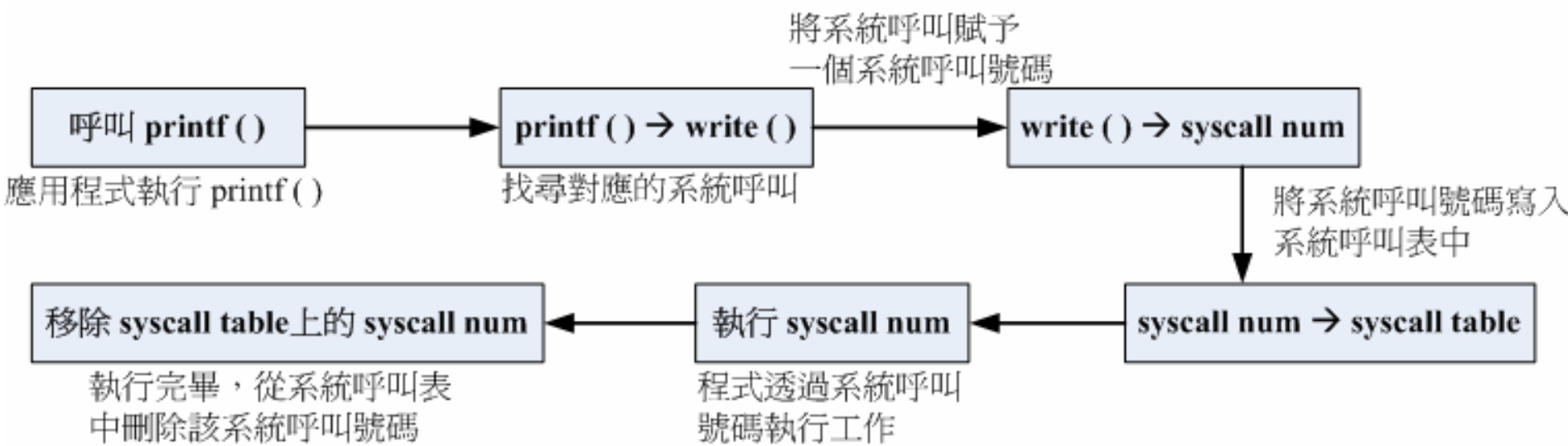
系統呼叫與程式間的關係範例



- C 函式庫提供 printf() 讓程式設計師撰寫程式/指令之用
- 實際在核心可以操作的指令為 write() 這個系統呼叫的函式庫
  - 作業系統會主動的設計 printf()與 write() 之間的對應；
  - 程式設計師只要知道 printf() 怎麼用就好了，不需理會 write() 函式庫。
- 從此，一支C程式語言可以跨平台運作了！只要該平台支援C的 printf()函式庫即可！

系統呼叫的運行方式

- System Call在Linux作業系統中又常被稱為Syscalls，主要會透過功能的需求而被呼叫來使用



系統呼叫於核心的運作方式

- 核心所在的記憶體空間是受保護的，一般程式無法直接存取該區塊
  - 若可直接存取，則作業系統將無安全性與穩定性！
- 一般使用者程式需要透過『軟體中斷』訊號告知核心層，此程式需要例外的需求處理
  - X86架構的軟體中斷：int 0x80 中斷
  - Torvalds的軟體中斷：sysenter/sysexit 指令集

系統呼叫的參數驗證

- 每個行程所攜帶的參數在進入核心層前都必須透過一些確認：
  - 當指標於用戶層指向一個記憶體區域時，行程不能進入核心層進行存取核心資源。(因非指向核心)
  - 行程指標指向行程記憶體空間中的一個區域時，行程不能夠進入核心中讀取其他人的資料。(因非指向核心)
  - 當記憶體被標註為可讀取或是可寫入時，行程才可以進行讀取，行程本身不能忽略記憶體存取的限制。

本章重點回顧

- 了解何謂系統呼叫。
- 了解系統呼叫與程式間的關係。
- 了解到Linux作業系統於核心中所提供的Syscalls的運作方式。