序

主旨

Linux 是一個自由的 UNIX 作業系統實作，本書介紹 Linux 程式設計介面（Programming Interface），如系統呼叫（system call）、函式庫（library），以及其他 Linux 提供的底層介面。在 Linux 系統的每個程式都會直接或間接的利用這些介面完成工作，例如：檔案 I/O、用以建立或刪除檔案目錄、建立新的行程（process）、啟動程式、設定計時器、同一部電腦上的行程與執行緒（thread）之間的溝通，以及不同電腦上的行程如何透過網路溝通，這些底層介面即是所謂的系統程式設計介面（system programming interface）。

雖然筆者致力於研究 Linux，不過也有關注相關的規範標準與可攜性（portability）的議題，讓 Linux 特有的（Linux-specific）討論細節可以明確地跟多數 UNIX 系統功能、POSIX 與 Single UNIX Specification 規範的討論有所區隔。本書對於 UNIX / POSIX 規範的介面亦提供完整的介紹，協助程式設計師開發可通用於其他 UNIX 系統的應用程式，或是可攜的跨平台應用程式。

誰適合這本書

本書適合有下列目的之讀者：

- 程式設計師、軟體規劃師：開發 Linux、UNIX 系統或其他符合 POSIX 規範的系統之應用程式。
- 程式設計師：想要開發可以通用於其他的 UNIX 系統、作業系統與 Linux 之間的應用程式。
- 老師與學生：將開設或選修 Linux / UNIX 系統的程式設計課程。
- 系統管理員與「求知若渴的使用者」：想要深入了解 Linux / UNIX 程式設計介面，以及研究系統程式的實作細節。

筆者假設讀者曾經寫過程式，可以不需具備系統程式經驗，但需具備「the C programming language」此書的知識背景與操作 shell 及常見的 Linux ／ UNIX 指令的能力。如果讀者才剛開始接觸 Linux 或 UNIX 系統，可以從第 2 章開始閱讀，這個章節以程式設計的觀點來引導讀者了解 Linux ／ UNIX 系統的基本概念。

> （Kernighan & Ritchie，1988）是經典的 C 語言導讀本。（Harbison & Steele，2001）對 C 有深入的探討，並涵蓋 C99 標準的異動內容。（van der Linden，1994）是欣賞 C 語言面貌的另一個選擇，不僅相當有趣也富有教育性。（Peek 等人，2001）則簡潔有力的介紹如何使用 UNIX 系統。
>
> 在本書中，常常會看到像這樣的小字體段落，目的是用來輔助說明主文的內容，包含基本原理、實作細節、背景資訊、歷史典故，以及其他相關的主題等。

Linux 與 UNIX

因為其他 UNIX 系統的大多數 API 在 Linux 都已經有實作，所以本書也可以當作開發符合標準規範的 UNIX 系統程式（即為 POSIX）之 API 大全。我們不僅想開發可攜式應用程式，也要結合 Linux 的擴充功能（Extension）與 UNIX 的 API。原因是除了 Linux 很受歡迎之外，有些時候是考量效能因素，或因為 UNIX 標準 API 沒有提供的功能，因此所有的 UNIX 系統都會支援這些擴充功能。

筆者為了讓本書滿足各種 UNIX 實作系統的開發需求，已經完整地涵蓋了 Linux 特有的程式設計功能，這些功能分別是：

- *epoll*，取得檔案 I/O 事件通知的機制。
- *inotify*，監控檔案與目錄變更的機制。
- Capability，這個機制讓我們在一個行程中可以細分何時才需要使用超級使用者（superuser）的管理員特權執行，而非整個行程都具有特權使用者的權限擴充屬性（extened attribute）。
- i-node 旗標。
- *clone()* 系統呼叫。
- /proc 檔案系統。
- Linux 特有的實作細節：檔案 I/O、訊號（signal）、計時器（timer）、執行緒（thread）、共享函式庫（share library）、行程間通訊與通訊端（socket）。

本書的使用方式與架構

讀者至少可以用下列兩種方式來使用本書：

- 作為 Linux / UNIX 程式設計介面的導讀教材，讀者可以從第 1 章開始依序閱讀本書，為了縮減本書的篇幅，後續章節若提到前述章節的相關內容，則會盡量透過引用的方式。
- 可當 Linux / UNIX 程式設計介面的完整開發手冊，本書使用大量的索引與交互參照以便利讀者隨意翻閱有興趣的主題。

筆者將本書的章節分類為下列幾個部分：

1. **背景與概念**：UNIX、C 與 Linux 的歷史，UNIX 標準的概念（第 1 章）；以程式設計人員的觀點簡介 Linux 與 UNIX 的概念（第 2 章）；以及 Linux 與 UNIX 系統程式設計的基本概念（第 3 章）。
2. **系統程式設計介面的基本功能**：檔案 I/O（第 4 章與第 5 章）；行程（第 6 章）；記憶體配置（第 7 章）；使用者與群組（第 8 章）；行程識別（第 9 章）；時間（第 10 章）；系統限制與選項（第 11 章）；以及檢索系統與行程資訊（第 12 章）。
3. **系統程式設計介面的進階功能**：檔案 I/O 緩衝區（第 13 章）；檔案系統（第 14 章）；檔案屬性（第 15 章）；擴充屬性（第 16 章）；存取控制清單（第 17 章）；目錄與連結（第 18 章）；監控檔案事件（第 19 章）；訊號（第 20 章至第 22 章）；以及計時器（第 23 章）。
4. **行程、程式與執行緒**：建立行程、終止行程、監控子行程，與執行程式（第 24 章至第 28 章）；以及 POSIX 執行緒（第 29 章至第 33 章）。
5. **行程與程式的進階主題**：行程群組、作業階段與工作控制（第 34 章）；行程的優先權與排班（第 35 章）；行程資源（第 36 章）；Daemon（第 37 章）；撰寫安全的特權程式（第 38 章）；能力（capability）（第 39 章）；登入記帳（第 40 章）；以及共享函式庫（第 41 章與第 42 章）。
6. **行程間的通訊（*IPC*）**：IPC 概觀（第 43 章）；管線（pipe）與命名管線（FIFO）（第 44 章）；System V IPC 的訊息佇列（message queue）、號誌（semaphore）與共享記憶體（第 45 章至第 48 章）；記憶體映射（第 49 章）；虛擬記憶體操控（第 50 章）；POSIX IPC 的訊息佇列、號誌與共享記憶體（第 51 章至第 54 章）；以及檔案鎖（第 55 章）。

7. *Socket* 與網路程式設計：IPC 與 socket 網路程式設計（第 56 章到第 61 章）
8. 進階的 *I/O* 主題：終端機（第 62 章）；替代 I/O 模型（第 63 章）；以及虛擬終端機（第 64 章）。

範例程式

筆者設計了簡短並可完整執行的程式，用以示範書中所談到的程式設計介面，讀者可以直接在命令列的環境實驗這些程式，觀察每個系統呼叫與函式庫是如何運作的。因此，本書提供許多範例程式－大約一萬五千行的 C 語言原始碼（source code）與 shell 作業階段的紀錄。

雖然能夠閱讀與使用這些範例程式做實驗已經是好的開始，不過如果要紮實的學會本書所談的觀念，最有效的方法還是要自己動手寫程式，讀者可以先透過修改範例程式或重寫來驗證自己的想法。

書中所有的原始碼都可以從本書的網站下載取得，散佈版（Distribution version）的原始碼有很多是書裡沒有的，不過這些程式的註解都會詳細敘述功能與目的，有 *Makefile* 可以讓讀者便於編譯程式，還有說明檔（`README`）用於介紹每個程式的功能與用途。

本書的原始碼可在基於 GNU Affero General Public License（Affero GPL）version 3 的規範下自由地重新散佈（將完整的規範副本附於原始碼目錄中）。

習題

大部分章節後面都有附習題，有些習題的實驗可以用本書的範例程式進行、有些習題與該章節所談的觀念相關，還有一些習題為了讓讀者可以紮實的瞭解教材內容，會需要讀者撰寫程式。附錄 F 提供部分的習題解答。

標準與可攜性

本書特別關注可攜性的議題，讀者會經常看到書中引用相關的標準，尤其是綜合 POSIX.1-2001 與 Single UNIX Specification version 3（SUSv3）標準。你們也會注意到 POSIX.1-2008 與 SUSv4 標準在最近的修訂與改變。（由於 SUSv3 規範已有大幅的修訂，而且也是本書著作時最有影響力的 UNIX 標準，所以本書主要探討 SUSv3 標準，不過讀者可以將本書多數所談到的 SUSv3 規範內容視為與 SUSv4 規範相符，因為本書會將與 SUSv4 不同之處特別標註。）

對於非標準規範的 Linux 功能，筆者都會與其他的 UNIX 系統實作進行比較，不僅強調是 Linux 系統才有實作的功能，也會點出 Linux 在系統呼叫與函式庫的實作與其他 UNIX 系統實作有哪些的微幅差異。大多數的功能如果本書沒有特別提到是 Linux 才有的，讀者都可以將這些功能視為與 UNIX 系統實作的標準相同。

書中大部分的範例程式都在 Solaris、FreeBSD、Mac OS X、Tru64 UNIX 以及 HP-UX 系統經過測試（除了使用 Linux 特有功能所開發的程式例外）。為了改善範例程式的可攜性，在本書的網站有另外提供書中沒有的原始碼（extra code），可以作為某些範例程式的替代版本。

Linux 核心與 C 函式庫的版本

本書主要研究 Linux 2.6. *x* 版本的核心，也是本書在著作時最多人使用的版本，同時也涵蓋 Linux 2.4 版本的核心，並指出 Linux 2.4 與 2.6 之間的功能差異，如果是 Linux 2.6.*x* 系列核心的新功能，本書會精確的標註出核心版本（例如：2.6.34）。

關於 C 函式庫，本書主要著重於第 2 版的 GNU C 函式庫（*glibc*），並會特別標出與其他 *glibc* 2. *x* 版本的差異。

在本書即將出版之際，Linux 才剛發行 2.6.35 版核心，而 *glibc* 也在近期剛釋出了 2.12 版，本書目前主要與上述的軟體版本有關。當本書出版後，本書的網站上會提供關於 Linux 與 *glibc* 介面的更新資訊。

在其他程式語言使用程式設計介面

雖然範例程式是以 C 語言撰寫的，讀者也可以使用其他的程式語言來應用本書所介紹的介面。例如，C++、Pascal、Modula、Ada、Fortran、D 等編譯式程式語言，或 Perl、Python 與 Ruby 這類直譯式程式語言。（Java 使用的是與上述不同的方法，請參考［ Rochkind，2004 ］。）

為了使其他程式語言（除了 C++）也能有固定的定義與函式宣告，需要使用一些其他的技術；而要傳遞函式參數到連結的 C 程式，也要額外處理一些工作。雖然使用別的程式語言會有些地方不太一樣，不過基本觀念還是相通的，所以即使讀者以其他程式語言進行開發，也是能透過本書對應相關的 API 的資訊。

關於作者

筆者從 1987 年開始使用 UNIX 與 C，手邊拿著 Marc Rochkind 第一版的 *Advanced UNIX Programming* 和一份已經被翻了無數次的 C shell 使用手冊的影印本，在一台 HP Bobcat 工作站前坐了幾個星期的時間。筆者至今仍使用這樣的練功方式學習，對於任何想要學習新軟體技術的人，筆者的建議是：投入時間閱讀文件（如果有文件），並開始練習寫一些小型測試程式（並逐步遞增程式的規模），直到讀者自認為已經充分了解這個軟體。使用這樣的方法長期自我訓練可以快速的成長茁壯，這是筆者的親身體會，本書的範例程式可以引領讀者如何自我修練。

筆者主要的工作是軟體工程師與規劃師，同時也是一位熱血的老師，投入幾年的時間在學術界與產業界授課。筆者已經教授過許多為期單週的 UNIX 系統程式設計課程，而這些授課的經驗成就了這本書。

筆者接觸 Linux 的時間大約只有 UNIX 的一半，而從接觸到 Linux 開始，筆者越來越沉迷於在 Linux 核心與使用者空間（user space）之間的 Linux API。這讓筆者有機會參與許多相關的活動，並陸續地回報 POSIX / SUS 標準的輸入（input）與臭蟲（bug）。筆者測試與規劃如何修正 Linux 核心中新增的使用者空間 API（幫忙找出許多 API 程式碼及設計上的錯誤並除錯），筆者經常受邀擔任與 API 及其相關文件主題的研討會講者，也受邀參與許多 Linux 核心開發者年會的場合（the annual Linux Kernel Developers Summit）。受邀到這些活動主要是因為筆者致力於 Linux 的 *man-page* 專案。（*http://www.kernel.org/doc/man-pages/*）。

man-page 專案提供 Linux 手冊（manual）第 2、3、4、5 及第 7 節的內容，主要與本書探討的主題相同，用來敘述 Linux 核心與 GNU C 函式庫提供的 API。筆者參與 *man-page* 的維護已經超過了十年的時間，自從 2004 年開始，筆者即是該專案的維護者，負責的工作有撰寫文件、閱讀核心與函式庫的原始碼，以及撰寫程式來驗證文件的內容。（撰寫 API 文件是好的除錯方式。）在總共多達約 900 頁的 *man-page* 中，其中有 140 頁的第一作者都是筆者，而且是另外 125 頁的共同作者之一，可以算是對於 *man-page* 貢獻最多心力的人，所以讀者可能在閱讀這本書以前，就曾經讀過筆者所寫的 *man-page* 手冊，期望這些手冊能夠對大家有所助益，並讓本書帶給讀者更多的收穫。

誌謝

如果沒有許多人的幫忙，本書不會有現在豐富的內容，很高興筆者能夠在此表達對他們的感謝。

3

系統程式設計概念

本章涵蓋各式系統程式設計所預先需要的主題。我們先導讀系統呼叫及詳細敘述在執行之間所發生的步驟。我們接著探討函式庫的函式，以及它們與系統呼叫的差異，並搭配（GNU）C 函式庫的介紹。

為了判斷呼叫是否成功，無論我們在何時建立系統呼叫或是呼叫函式庫的函式，我們應該都要檢查呼叫的傳回值。我們會說明如何執行這類檢查，並介紹一些書中範例使用的大多數函式，以從系統呼叫及函式庫的函式中診斷錯誤。

最後我們透過探討各式與可攜（portable）程式設計的相關議題進行總結，特別是使用功能測試巨集（feature test macro），以及 SUSv3 所定義的標準系統資料型別。

3.1 系統呼叫（System Call）

系統呼叫（*system call*）是控制的核心進入點，讓行程（process）要求核心代為執行一些動作。核心透過系統呼叫的應用程式設計介面（API，application programming interface），建立允許程式存取服務的範圍。這些服務諸如：建立一個新的行程、執行 I/O，以及建立管線（pipe）做為行程間的通訊（inter process communication）。（*syscalls(2)* 使用手冊會列出 Linux 的系統呼叫）。

在繼續介紹系統呼叫如何運作的細節以前，我們先提一些通用的觀念：

- 系統呼叫將處理器的狀態從使用者模式（user mode）切換為核心模式（kernel mode），讓 CPU 可以存取受保護的核心記憶體。
- 系統呼叫組合（system calls set）是固定的，每個系統呼叫由單個唯一的號碼識別。（一般程式不會知道這個編號機制，程式是透過名稱識別系統呼叫）。
- 每個系統呼叫可以有一個參數集，用來設定要從使用者空間（user space）（如：行程的虛擬記憶體空間）傳輸到核心空間（kernel space）的資訊，反之亦然。

從程式設計的觀點，呼叫一個系統呼叫看起來就像是呼叫一個 C 函式，然而，在場景背後，在執行一個系統呼叫期間有許多步驟在進行。我們以「x86-32」硬體平台為例，依系統呼叫在此平台發生的步驟順序進行探討，其步驟如下：

1. 應用程式可藉由呼叫 C 函式庫的包裝函式（wrapper function）以產生系統呼叫。

2. 包裝函式必須將全部的系統呼叫參數提供給陷阱處理常式（system call trap-handling routine）（這裡是簡述）。這些參數透過堆疊（stack）傳遞給包裝函式，但是核心希望它們可以好好待在特定的暫存器裡，所以包裝函式會將參數複製到暫存器中。

3. 因為全部的系統呼叫進入核心的方式都一樣，所以核心需要一些識別系統呼叫的方法。為此，包裝函式將系統呼叫的編號複製到特定的 CPU 暫存器（`%eax`）。

4. 包裝函式會執行一個 *trap* 機器指令（`int 0x80`），讓處理器從使用者模式切換到核心模式，並執行系統陷阱向量（trap vector）`0x80` 指向的程式碼（即十進位的 128）。

   > 大多數的近代 x86-32 架構都會實作 `sysenter` 指令，提供一個比傳統 `int 0x80` trap 指令更快速的方法來切換到核心模式。在核心 2.6 及 *glibc* 2.3.2 以後都有提供使用 `sysenter`。

5. 為了回應在位置 `0x80` 的 trap，核心會呼叫 *system_call()* 常式（位在組合語言檔 `arch/x86/kernel/entry.S`），以處理此 trap。這個處理常式（handler）進行的工作如下：

   a）將暫存器值儲存到核心堆疊（6.5 節）。

   b）檢查系統呼叫編號的有效性。

c）可以系統呼叫編號檢索一張系統呼叫服務常式表格（核心變數 *sys_call_table*），以找出合適的系統呼叫服務常式（system call service routine），並進行呼叫。若系統呼叫服務參數需要任何參數，則先檢查這些參數的可用性；例如，檢查指向使用者記憶體中有效位置的位址。接著，服務常式會執行要求的任務，可進行呼叫以修改參數所指定位址的值，以及在使用者記憶體與核心記憶體之間傳輸資料（如：在 I/O 操作裡）。最後，服務常式會將結果狀態（result status）傳回 *system_call()* 常式。

d）從核心堆疊回存暫存器值，並將系統呼叫的傳回值存在堆疊。

e）回到包裝函式，同時將處理器切回使用者模式。

6. 若系統呼叫服務常式的傳回值指出有錯誤發生，則包裝函式以此值設定 *errno* 全域變數（參考 3.4 節）。接著從包裝函式返回到呼叫者時，會提供一個整數的傳回值，用以表示系統呼叫的成功與否。

> 在 Linux 上，系統呼叫服務常式遵循著一個慣例，以非負數的傳回值代表成功。在一個發生錯誤的情況，常式會傳回負數，是某個負值的 *errno* 常數。當傳回負值時，C 函式庫的包裝函式會將負數改成正數，並將結果複製到 *errno*，再傳回 -1 做為包裝函式的結果，用以通知呼叫的程式有錯誤發生。
>
> 此慣例的成立是基於，假設系統呼叫服務常式不會在執行成功時傳回負值。然而，在一些常式上，這樣的假設不會成立。通常這不是問題，因為 *errno* 的負數值範圍不會與傳回的負數值重疊。然而，此慣例在一種情況下會有問題：在 *fcntl()* 系統呼叫的 F_GETOWN 操作，我們會在 63.3 節中說明。

圖 3-1 呈現的是前述使用 *execve()* 系統呼叫的順序。在 Linux/x86-32 系統，*execve()* 的系統呼叫編號是 11（__NR_execve）。因此，在 *sys_call_table* 向量中，entry 11 包含 *sys_execve()* 的位址，為此系統呼叫的服務常式。（在 Linux 系統，通常系統呼叫服務常式的命名格式是 *sys_xyz()*，這裡的 *xyz()* 是問題中的系統呼叫）。

上一段所提供的資訊已經足以應付本書所需的知識，然而，重點在於，即使是個單純的系統呼叫，很少工作要處理，但系統呼叫依然存在著小但仍有影響的負載。

> 舉個進行系統呼叫產生的負載範例，以 *getppid()* 系統呼叫為例，它單純傳回呼叫者行程的父行程 ID。筆者的某台 x86-32 系統使用 Linux 2.6.25，完成一千萬次的 *getppid()* 呼叫大約需要 2.2 秒。平均每個呼叫大約花費 0.3 微秒（microsecond）。提供另一個對照組，我們在相同的系統上，將單純傳回一個整數的 C 函式執行一千萬次僅需 0.11 秒，大約是呼叫 *getppid()* 所需時間的 1/20。當然，多數系統呼叫的負載都比 *getppid()* 更大。

因為，從 C 程式的觀點，呼叫 C 函式庫的包裝函式與呼叫相對應的系統呼叫服務常式差不多，本書後續會使用如「執行 *xyz()* 系統呼叫」這類字眼，用以表示「呼叫會執行 *xyz()* 系統呼叫的包裝函式」。

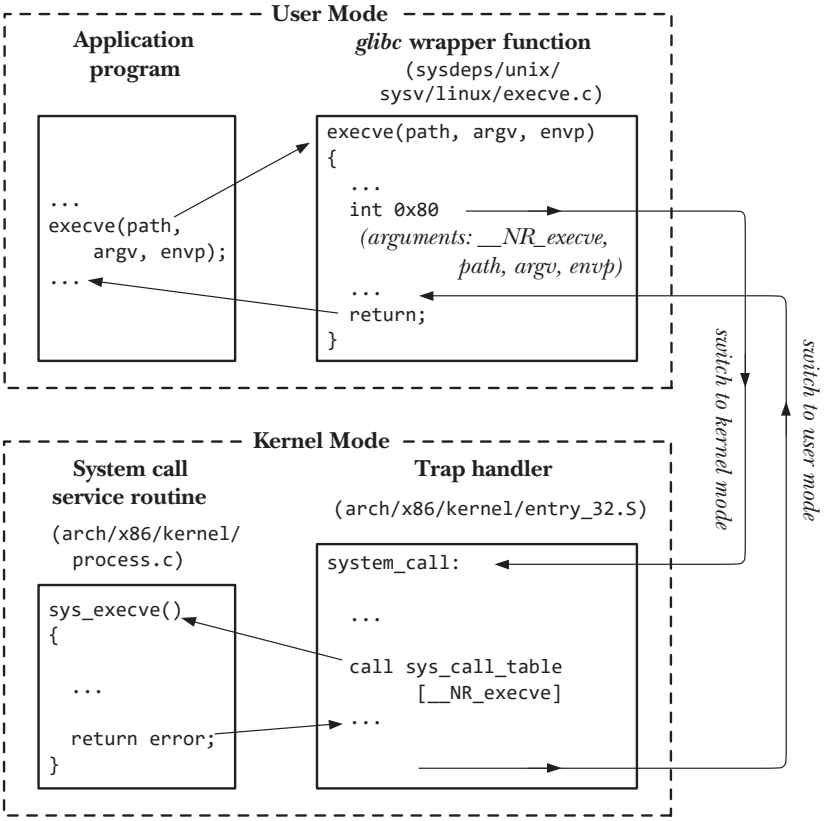

圖 3-1：執行系統呼叫所進行的步驟

附錄 A 所述的 *strace* 指令可追蹤程式使用的系統呼叫，若非用在除錯，則是單純找出程式正在做什麼事。

更多與 Linux 系統呼叫機制的相關資訊可以參考（Love，2010）、（Bovet & Cesati，2005）以及（Maxwell，1999）。

3.2 函式庫函式（library function）

一個函式庫函式（library function）就只是構成標準 C 函式庫的多個函式之一。（簡而言之，本書後續提及特定函式時，通常只會稱為**函式**，而不是**函式庫函式**）。這些函式的目的彼此大不同，比如有開啟檔案、將時間轉換為人們可讀的格式，以及比較兩個字元字串等任務。

許多函式庫函式不會使用任何的系統呼叫（如：字串處理函式）。另一方面，有些函式庫函式會架構於系統呼叫之上。例如：fopen 函式庫函式實際上是使用 *open()* 系統呼叫開啟檔案。通常設計函式庫函式是為了提供比底層系統呼叫更為友善的呼叫介面。例如：*printf()* 函式提供輸出格式及資料緩衝，但是 *write()* 系統呼叫則只能輸出一個區塊的位元組資料。同樣地，*malloc()* 及 *free()* 函式執行各種簿記（bookkeeping）任務，因此會比底層的 *brk()* 系統呼叫更易於配置及釋放記憶體。

3.3 標準 C 函式庫；GNU C 函式庫（*glibc*）

在各個 UNIX 平台上的標準 C 函式庫實作都有所差異，在 Linux 上最常用的實作是 GNU C 函式庫（*glibc*，*http://www.gnu.org/software/libc/*）。

> GNU C 函式庫的主要開發者與維護者原本是 Roland McGrath。之後，這項任務由 Ulrich Drepper 負責。
>
> Linux 也有各種 C 函式庫可用，包含記憶體需求較少的函式庫，適用於開發嵌入式裝置的應用程式。這類函式庫有 *uClibc*（*http://www.uclibc.org/*）及 *diet libc*（*http://www.fefe.de/dietlibc/*）。我們在本書僅探討 glibc，因為多數在 Linux 上開發的應用程式都是使用這個 C 函式庫。

取得系統的 *glibc* 版本

我們有時會需要得知系統的 *glibc* 版本。我們可以將 *glibc* 共享函式庫視為執行檔，並直接在 shell 中執行以取得版本編號。當我們直接將函式庫當作執行檔執行時，它會顯示許多文字，包含版本編號。

```
$ /lib/libc.so.6
GNU C Library stable release version 2.10.1, by Roland McGrath et al.
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 4.4.0 20090506 (Red Hat 4.4.0-4).
Compiled on a Linux >>2.6.18-128.4.1.el5<< system on 2009-08-19.
Available extensions:
```

```
        The C stubs add-on version 2.1.2.
        crypt add-on version 2.1 by Michael Glad and others
        GNU Libidn by Simon Josefsson
        Native POSIX Threads Library by Ulrich Drepper et al
        BIND-8.2.3-T5B
        RT using linux kernel aio
For bug reporting instructions, please see:
<http://www.gnu.org/software/libc/bugs.html>.
```

在一些 Linux 平台上，GNU C 函式庫所在的路徑名稱並不是 `/lib/libc.so.6`。得知函式庫位置的一種方式是執行 *ldd*（列出動態相依性）程式，可查詢執行檔使用的動態連結函式庫（多數的執行檔以動態方式連結）。我們接著就能檢視函式庫相依性清單的結果，以找出 *glibc* 共享函式庫的位置：

```
$ ldd myprog | grep libc
        libc.so.6 => /lib/tls/libc.so.6 (0x4004b000)
```

應用程式可以取得系統目前 GNU C 函式庫版本的另一個方式：透過測試常數（testing constant）或呼叫一個函式庫函式。*Glibc* 從 2.0 版本起定義兩個常數：`__GLIBC__` 以及 `__GILBC_MINOR__`，這些常數可以在編譯期測試（在 `#if` 陳述句）。在一個裝有 *glibc* 2.12 的系統上，這些常數的值分別是 2 與 12。然而，這些常數只能用在同一個系統編譯的程式，不能用在其他有不同 *glibc* 版本的系統上。為了處理這個問題，程式可以在執行期呼叫 *gnu_get_libc_version()* 函式，已取得可用的 *glibc* 版本。

```
#include <gnu/libc-version.h>

const char *gnu_get_libc_version(void);
                    Returns pointer to null-terminated, statically allocated string
                                        containing GNU C library version number
```

函式 *gnu_get_libc_version()* 傳回指向字串的指標，比如是 2.12。

我們可透過使用 *confstr()* 函式解析（*glibc* 特有的）`_CS_GNU_LIBC_VERSION` 組態變數值，以取得版本資訊。這個呼叫會傳回如 *glibc 2.12* 的字串。

3.4 處理系統呼叫及函式庫函式的錯誤

幾乎每個系統呼叫與函式庫函式都會傳回幾種狀態值，用以表示呼叫成功或失敗。無論呼叫成功與否，我們都應該不斷檢查此狀態值。若呼叫失敗，則應該採取適當的處理，至少應該讓程式顯示錯誤訊息，警告有預期以外的事情發生。

雖然可透過排除這些檢查來節省輸入的時間（尤其是看過沒有檢查狀態值的 UNIX 與 Linux 程式範例之後），但是這個地方不該節省。因為若不檢查這些「不太可能會失敗」的系統呼叫或函式庫函式的傳回狀態，可能反而要浪費好幾個小時的時間除錯。

> 有些系統呼叫絕對不會失敗。例如：*getpid()* 只會成功並傳回行程 ID，而 *_exit()* 只會將行程結束。這類系統呼叫不需檢查其傳回值。

處理系統呼叫錯誤

每個系統呼叫的使用手冊都有說明可能的傳回值，介紹哪個值代表著錯誤。錯誤時通常是透過傳回 -1 來表示。因而，系統呼叫可用下列的程式碼檢查：

```
fd = open(pathname, flags, mode);       /* system call to open a file */
if (fd == -1) {
    /* Code to handle the error */
}
...
if (close(fd) == -1) {
    /* Code to handle the error */
}
```

當系統呼叫失敗時，會將全域整數變數 *errno* 設定為正值，這表示特定的錯誤。引用 <errno.h> 標頭檔可取得 *errno* 的宣告，以及各種錯誤編號的常數組。這些符號名稱全部都以 E 開頭。在每個使用手冊的 ERRORS 章節會提供每個系統呼叫可能傳回的 *errno* 值清單。這裡是個使用 *errno* 診斷系統呼叫錯誤的簡單範例：

```
cnt = read(fd, buf, numbytes);
if (cnt == -1) {
    if (errno == EINTR)
        fprintf(stderr, "read was interrupted by a signal\n");
    else {
        /* Some other error occurred */
    }
}
```

成功的系統呼叫與函式庫函式絕不會將 *errno* 重設為 0，所以此變數的值可能不是零，而是之前呼叫產生的錯誤。此外，SUSv3 允許執行成功的函式呼叫將 *errno* 設定為非零值（雖然只有少數函式這麼做）。因此，在檢查錯誤時，我們應該每次都要先檢查函式的傳回值是否指出錯誤，並接著檢測 *errno*，以得知發生錯誤的原因。

有些系統呼叫（如：*getpriority()*）可以在執行成功時傳回 -1。為了判斷這類呼叫是否有錯誤發生，我們會在呼叫之前就將 *errno* 設定為 0，並在呼叫之後檢查。若呼叫傳回 -1 且 *errno* 不為零，則表示有錯誤發生。（這同樣也能套用在一些函式庫函式）。

在系統呼叫失敗之後，一般採取動作是依據 *errno* 的值印出錯誤訊息，這就是 *perror()* 及 *strerror()* 函式庫函式的功能。

函式 *perror()* 印出其 *msg* 參數指向的字串，並接著一個相對應於 *errno* 現值的訊息。

```
#include <stdio.h>

void perror(const char *msg);
```

處理系統呼叫錯誤的一個簡易方式如下：

```
fd = open(pathname, flags, mode);
if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

函式 *strerror()* 傳回其 *errnum* 參數中的錯誤編號所對應的錯誤字串。

```
#include <string.h>

char *strerror(int errnum);
                        Returns pointer to error string corresponding to errnum
```

由 *strerror()* 傳回的字串可以是靜態配置的，這表示下次的 *strerror()* 呼叫會覆蓋此字串。

若 errnum 是個無法確認的錯誤編號，則 *strerror()* 傳回一個 *Unknown error nnn* 格式的字串。在一些其他平台，*strerror()* 此時會傳回 NULL。

因為 *perror()* 與 *strerror()* 函式會跟系統的語系有關（10.4 節），所以會依據系統語系來顯示錯誤說明。

處理函式庫函式的錯誤

各種函式庫函式會傳回不同的資料型別，以及依錯誤傳回不同的值。（檢查每個函式的使用手冊）。依照我們的目的，可將函式庫函式分成下列幾類：

- 有些函式庫函式傳回錯誤資訊的方式與系統呼叫相同：傳回值為 -1 時，以 *errno* 指出錯誤。這類函式的例子是 *remove()*，可用來移除檔案（使用 *unlink()* 系統呼叫）或移除目錄（使用 *rmdir()* 系統呼叫）。診斷這些函式錯誤的方式與系統呼叫相同。
- 有些函式庫函式在執行失敗時並非傳回 -1，不過依然將 *errno* 設定為代表錯誤的數值。例如：*fopen()* 在發生錯誤時傳回一個 NULL 指標，並依據底層執行失敗的系統呼叫來設定 *errno*。函式 *perror()* 及 *strerror()* 可用於診斷這些錯誤。
- 其他函式庫函式完全不使用 *errno*。得知發生錯誤以及錯誤原因的方式須依據特定函式，這些都記載於函式的使用手冊。在這些函式使用 *errno*、*perror()* 或 *strerror()* 診斷錯誤是不對的。

3.5 本書的範例程式

本節會介紹本書範例程式採用的各種慣例與功能。

3.5.1 命令列選項與參數

書中許多範例程式的行為需要依據命令列選項及參數來決定。

傳統的 UNIX 命令列選項包含一開始的分隔號、一個識別選項的字母，以及一個選配參數。（GNU 工具集提供擴充的選項語法，包含兩個初始分隔號，接著一個識別選項的字串，及一個選配參數。）我們會使用標準的 *getopt()* 函式庫函式來解析這些選項（於附錄 B 介紹）。

我們在每個範例程式都會有個命令列語法，可為使用者提供簡易的協助：若使用 *-help* 選項，則程式會顯示命令列選項與參數語法的使用資訊。

3.5.2 常用的函式與標頭檔

多數的範例程式會引用一個標頭檔（header），內含常用的定義，並且也會使用一組常用的函式。我們在本節會探討標頭檔與函式。

常見的標頭檔

列表 3-1 列出本書每個程式最常用到的標頭檔。此標頭檔包含多數範例程式用到的各種其他標頭檔，並定義一個**布林**（*Boolean*）資料型別，以及定義用以計算兩個數值的最小值與最大值的巨集。使用此標頭檔可以讓我們的範例程式行數少一點。

列表 3-1：多數範例程式使用的標頭檔

lib/tlpi_hdr.h

```
#ifndef TLPI_HDR_H
#define TLPI_HDR_H      /* Prevent accidental double inclusion */

#include <sys/types.h>  /* Type definitions used by many programs */
#include <stdio.h>      /* Standard I/O functions */
#include <stdlib.h>     /* Prototypes of commonly used library functions,
                           plus EXIT_SUCCESS and EXIT_FAILURE constants */
#include <unistd.h>     /* Prototypes for many system calls */
#include <errno.h>      /* Declares errno and defines error constants */
#include <string.h>     /* Commonly used string-handling functions */

#include "get_num.h"    /* Declares our functions for handling numeric
                           arguments (getInt(), getLong()) */

#include "error_functions.h"  /* Declares our error-handling functions */

typedef enum { FALSE, TRUE } Boolean;

#define min(m,n) ((m) < (n) ? (m) : (n))
#define max(m,n) ((m) > (n) ? (m) : (n))

#endif
```

lib/tlpi_hdr.h

錯誤診斷函式

為了簡化範例程式的錯誤處理，我們使用列表 3-2 宣告的錯誤診斷函式。

列表 3-2：常用的錯誤處理函式之宣告

lib/error_functions.h

```
#ifndef ERROR_FUNCTIONS_H
#define ERROR_FUNCTIONS_H

void errMsg(const char *format, ...);

#ifdef __GNUC__
```

```
        /* This macro stops 'gcc -Wall' complaining that "control reaches
           end of non-void function" if we use the following functions to
           terminate main() or some other non-void function. */

    #define NORETURN __attribute__ ((__noreturn__))
    #else
    #define NORETURN
    #endif

    void errExit(const char *format, ...) NORETURN ;

    void err_exit(const char *format, ...) NORETURN ;

    void errExitEN(int errnum, const char *format, ...) NORETURN ;

    void fatal(const char *format, ...) NORETURN ;

    void usageErr(const char *format, ...) NORETURN ;

    void cmdLineErr(const char *format, ...) NORETURN ;

    #endif
```

lib/error_functions.h

我們為了診斷系統呼叫與函式庫函式造成的錯誤，會使用 *errMsg()*、*errExit()*、*err_exit()* 與 *errExitEN()* 函式。

```
#include "tlpi_hdr.h"

void errMsg(const char *format, ...);
void errExit(const char *format, ...);
void err_exit(const char *format, ...);
void errExitEN(int errnum, const char *format, ...);
```

函式 *errMsg()* 可將訊息輸出至標準錯誤（standard error），其參數列與 *printf()* 相同，除了會自動將結尾的換行字元附加到輸出的字串。函式 *errMsg()* 會印出與 *errno* 現值對應的錯誤文字，包含錯誤名稱，比如 EPERM，加上 *strerror()* 傳回的錯誤描述，依據參數列指定的格式輸出。

函式 *errExit()* 的運作模式與 *errMsg()* 類似，可是還會終止程式，會呼叫 *exit()*，或是若有將 EF_DUMPCORE 變數定義為字串（非空字串），則透過呼叫 *abort()* 產生一個核心傾印檔（core dump file），以供除錯器使用（我們會在 22.1 節介紹核心傾印檔）。函式 *err_exit()* 與 *errExit()* 類似，但有兩個不同之處：

- 不會在印出錯誤訊息以前刷新（flush）標準輸出。
- 是透過呼叫 *_exit()* 終止行程，而非 *exit()*，這能讓行程終止而不用刷新 *stdio* 緩衝區，也不用呼叫 exit 處理常式。

關於在 *err_exit()* 操作的細部差異會在第 25 章清楚說明，到時會介紹 *_exit()* 與 *exit()* 之間的差異，並探討在以 *fork()* 建立的子行程中，如何面對 *stdio* 緩衝區及 exit 處理常式。至於現在，我們只需要知道，若我們寫一個函式庫函式，用來建立需要因為發生錯誤而終止的子行程時，*err_exit()* 是非常好用的。此類的終止不會刷新子行程的 *stdio* 緩衝區（父行程的複本），也不會呼叫父行程建立的 exit 處理常式。

函式 *errExitEN()* 與 *errExit()* 相同，但 *errExitEN()* 不會依據 *errno* 現值而印出相對應的錯誤訊息，而是依據 *errnum* 參數提供的錯誤編號（因而以 EN 為後綴字），印出相對應的錯誤訊息。

我們主要會在使用 POSIX 執行緒 API 的程式使用 *errExitEN()*。POSIX 執行緒函式藉由傳回錯誤編號（通常 *errno* 的數值是正數）來診斷錯誤（POSIX 執行緒執行成功時，會傳回 0），這點與傳統的 UNIX 系統呼叫在錯誤時傳回 -1 不同。

我們可使用下列程式碼診斷 POSIX 執行緒函式發生的錯誤：

```
errno = pthread_create(&thread, NULL, func, &arg);
if (errno != 0)
    errExit("pthread_create");
```

然而，這個方法的效率不彰，因為執行緒程式會將 *errno* 定義為巨集（擴展為一個函式呼叫，可傳回一個可修改的左值）。因而，每次使用 *errno* 都會導致發生一次函式呼叫。函式 *errExitEN()* 可以讓我們寫出與上列程式碼等價，但更有效率的程式：

```
int s;

s = pthread_create(&thread, NULL, func, &arg);
if (s != 0)
    errExitEN(s, "pthread_create");
```

> 在 C 的術語中，左值（lvalue）是個參照一塊儲存空間的表示式（expression）。最常見的左值範例是變數的識別器（identifier）。有些操作元（operator）也會產生左值，比如：若 *p* 是個指向儲存區域的指標，則 **p* 就是一個左值。在 POSIX 執行緒 API 底下，會將 *errno* 重新定義為一個函式，可傳回一個指向執行緒特有的儲存區域之指標（請見 31.3 節）。

為了診斷其他類型的錯誤，我們會使用 *fatal()*、*usageErr()* 及 *cmdLineErr()*。