本書的目的在於，讓讀者透過實際動手操作組成電腦系統之作業系統（以下簡稱「OS」）與硬體的方式，在確認其特性的同時一併學習。本書的說明對象 OS 是 Linux。

Linux 系統，分為 Kernel 這個屬於系統的核心部分的程式，以及這之外的部分。正確來說，Linux 一詞，僅單指核心，不過本書對運作於 Linux 核心上的具有類似 UNIX 界面的 OS，為求方便，也統稱其為 Linux。至於 Kernel 的部分，則會標示為「Linux 核心」或是「核心」。

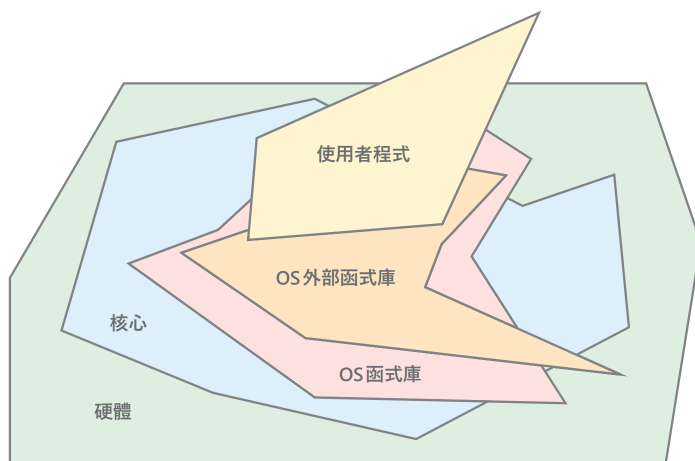
現代的電腦系統已被階層化、細分化，使用者也越來越不會意識到 OS 或硬體。Linux 也是一樣的情形。關於階層化，常常會用到如圖 00-01 這類「理想的結構」來描繪，並以「處理某一階層的人，只需要對下一個階層的部分有所了解即可」做說明。

圖 00-01　電腦系統的階層（理想的結構）

舉例來說，營運管理工程師只需要了解應用程式的外部規格即可，應用程式開發人員只需要了解函式庫即可等等。

但是實務上的系統，是像圖 00-02 所示，各個階層複雜地彼此連接在一起，有很多問題不是只知曉一部分就可以解決的。而且，像這類會大幅橫跨階層的知識，實際上大多都得透過長時間的實務經驗來自行學習。

圖 00-02 電腦系統的階層（現實）



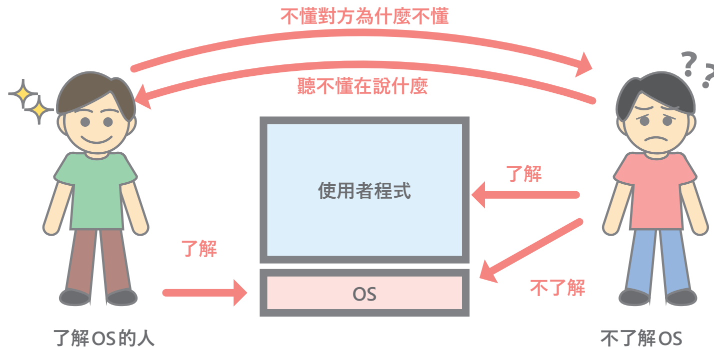
期望各位讀者可以透過本書，對於 Linux 與其中的核心、甚至硬體與其上層直接連接在一起的部分，能夠有充分的理解。如此一來，各位就會有能力處理如下所示的項目。

- 原因屬於核心或硬體等低階層的故障分析
- 考量到效能的編碼方式
- 對系統的各種統計數據／微調參數所代表意義有所理解

市面上有一些以 OS 的工作原理為主題的書籍／文章。明明知道這個事實，那麼為什麼我還是要撰寫新的書籍呢？這是因為現存的書籍／文章大多都不是針對特定某個 OS，而是只有針對其背後的理論進行解說或針對如 Linux 等特定 OS 的實作部份的原始碼進行解說這兩個類型。以這些書籍的編寫方式來說，會使得各位需要繞遠路才能達到上述的目標。如果讀者們在閱讀本書之前就已經對於 OS 有超出常人興趣的話影響不大，但是這對於並非如此的大多數的讀者來說，學習的門檻卻非常地高。因此，不論是對於新手還是老手來說，都很容易陷入「OS 是個充滿神祕與困難的東西」的困境。

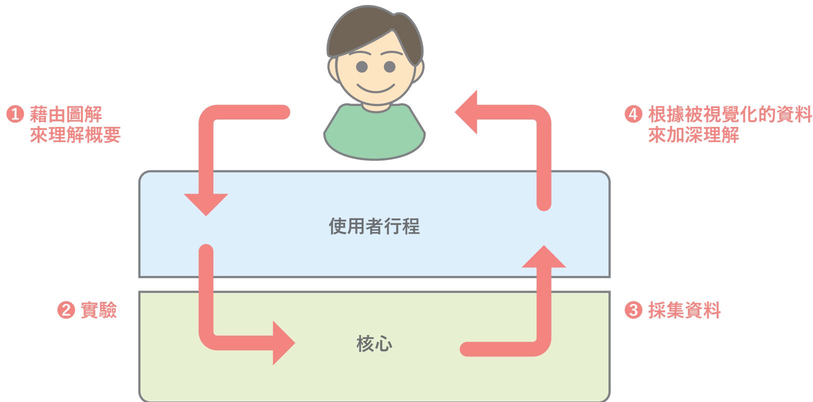
筆者實際上親眼目睹過很多次，對 OS 很了解的人跟不了解的人之間，發生了如圖 00-03 所呈現的溝通不良，筆者也曾經是當事人之一。說不定各位讀者也曾經有過類似經驗吧。

圖 00-03　OS 專家與外行人之間的溝通不良



為了改善這個狀況，本書將不會探討艱深的理論，而是以 Linux 為中心，在進到實作階段前就對 Linux 的工作原理進行解說。如本書的書名「圖解 LINUX 核心工作原理」所示，本書整體的結構是以圖 00-04 所示流程，針對 Linux 個別的功能以淺顯易懂的方式來編寫而成的。

圖 00-04　本書內容的學習、理解流程



本書所刊載的實驗，雖然它們是以不用親自嘗試也可以看得懂內容的方式撰寫，但還是強烈建議各位讀者在各自的環境上，試著實際操作並確認其結果。這是因為「只閱讀書」跟「閱讀後實際嘗試看看」的理解程度相較之下，後者的學習效果絕對遠高於前者。

本書將實驗程式所有的原始碼，依照使用的場合彙整於各個頁面上。

第 2 章

行程管理（基礎篇）

一個系統上大多存在複數個行程。我們只需要執行 `ps aux` 指令，就可將存在於系統中的全部行程給列舉出來。

```
$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND   ←❶
...
sat        19261  0.0  0.0  13840  5360 ?        S    18:24   0:00 sshd: sat@pts/0
sat        19262  0.0  0.0  12120  5232 pts/0    Ss   18:24   0:00 -bash
...
sat        19280  0.0  0.0  12752  3692 pts/0    R+   18:25   0:00 ps aux
$
```

❶這行是用來標示以下被輸出各行所代表意義的標頭行。這之後所列出的 1 行代表 1 個行程。上述之中 COMMAND 欄位代表指令名稱。在這邊我們不會做詳細的說明，不過 ssh 伺服器 sshd（PID=19261）在 bash（PID=19262）啟動之後，在這狀態下執行 `ps aux`。

ps 指令所輸出的標頭行，只需要使用 `--no-header` 選項就能夠刪除。接著讓我們來查看在筆者環境上行程的數量是多少吧。

```
$ ps aux --no-header | wc -l
216
$
```

共有 216 個行程存在。這 216 個行程各自在處理什麼呢？它們是如何受到管理的呢？本章將針對 Linux 用於管理這些行程的行程管理系統進行說明。

行程的建立

建立新行程的目的，可分為以下這兩種。

a. 將同一個程式的處理分成複數的行程來進行處理（例：網路伺服器當收到複數請求（request）時的受理）。

b. 建立其他的程式（例：從 bash 建立各種新程式）。

為了實現以上的內容，在 Linux 上會使用到 `fork()` 函數與 `execve()` 函數[*1]。

[*1] 我們只需要執行 `man 3 exec`，就可以查看到很多 execve() 函數的變種。

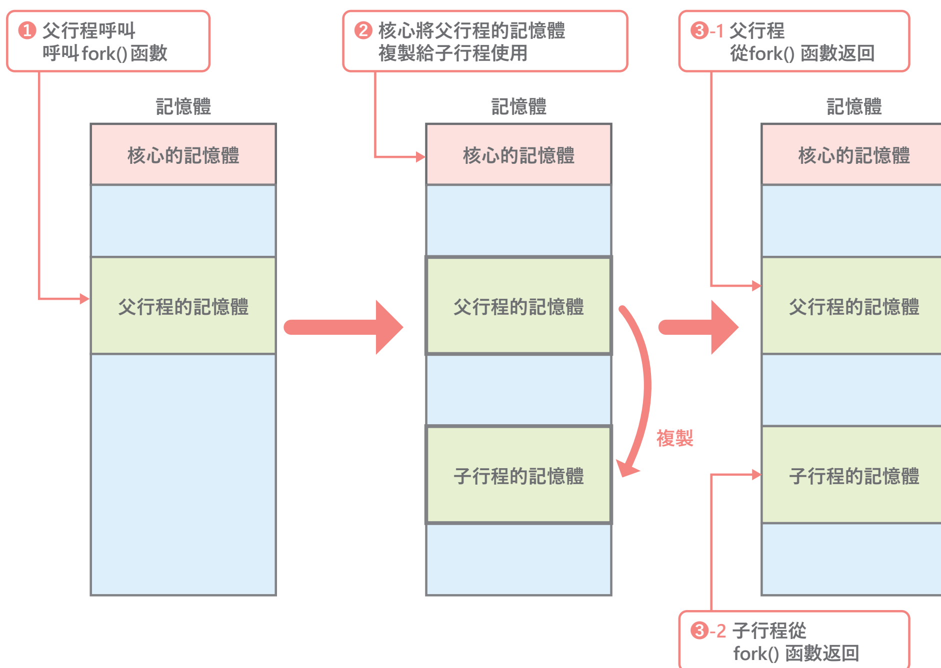
就內部來說，是各自對 clone()、execve() 這些系統呼叫進行呼叫。就前述 a. 來說，只使用到 fork() 函數；就 b. 來說，則是使用到 fork() 函數與 execve() 函數這兩種。

可將相同行程分裂成 2 個行程的 fork() 函數

當我們發出 fork() 函數後，會對已發出行程進行複製，雙方都可從 fork() 函數返回。被進行複製的原始行程則被稱為「父行程（parent process）」，因複製而新建立的行程便稱為「子行程（child process）」。這時候所經歷的流程如下（圖 02-01）。

1. 父行程呼叫 fork() 函數。
2. 確保子行程用記憶體區域，將父行程複製到該區域的記憶體中。
3. 父行程與子行程雙方都從 fork() 函數返回。如後續會說明到的，由於父行程與子行程的 fork() 函數的傳回值是不同的，所以可讓處理分支（後續說明）。

圖 02-01 以 fork() 函數建立行程



但是，實際上從父行程的記憶體複製到子行程的這個處理，會使用到於第 7 章介紹的寫入時複製（Copy-on-Write）這個功能，以非常小的負擔來完成處理。因此，在 Linux 上將相同程式的處理分成複數行程來處理時，多餘負載（overhead）是非常小的。

讓我們透過製作以下規格的 fork.py 程式（列表 02-01），來查看以 fork() 函數建立行程的狀況吧。

1. 呼叫 fork() 函數讓行程的流程分支。
2. 父行程在輸出自己的行程 ID 以及子行程的行程 ID 之後終止。子行程在輸出自己的行程 ID 之後終止。

列表 02-01 fork.py

```
#!/usr/bin/python3
import os, sys
ret = os.fork()
if ret == 0:
        print("子行程： pid={}, 父行程的 pid={}".format(os.getpid(), os.getppid()))
        exit()
elif ret > 0:
        print("父行程： pid={}, 子行程的 pid={}".format(os.getpid(), ret))
        exit()
sys.exit(1)
```

就 fork.py 程式來說，當返回 fork() 函數的時候，以父行程來說，子行程的行程 ID 如果是子行程的話就會返回 0。行程 ID 一定是 1 以上的數值，所以我們可以利用這點，讓父行程與子行程在呼叫 fork() 函數後處理分支。

那麼，讓我們來執行看看吧。

```
./fork.py
父行程： pid=132767, 子行程的 pid=132768
子行程： pid=132768, 父行程的 pid=132767
```

從以上內容我們可得知的是，行程 ID 為 132767 的行程分支了，而新行程 ID 為 132768 的行程被建立了，以及在發出 fork() 函數之後，可根據 fork() 函數的傳回值來判斷各自的處理是分支的。

關於 fork() 函數，剛接觸時實在很難馬上理解它到底是在做什麼的，希望各位可以透過重複閱讀本節內容及範例程式碼來融會貫通。

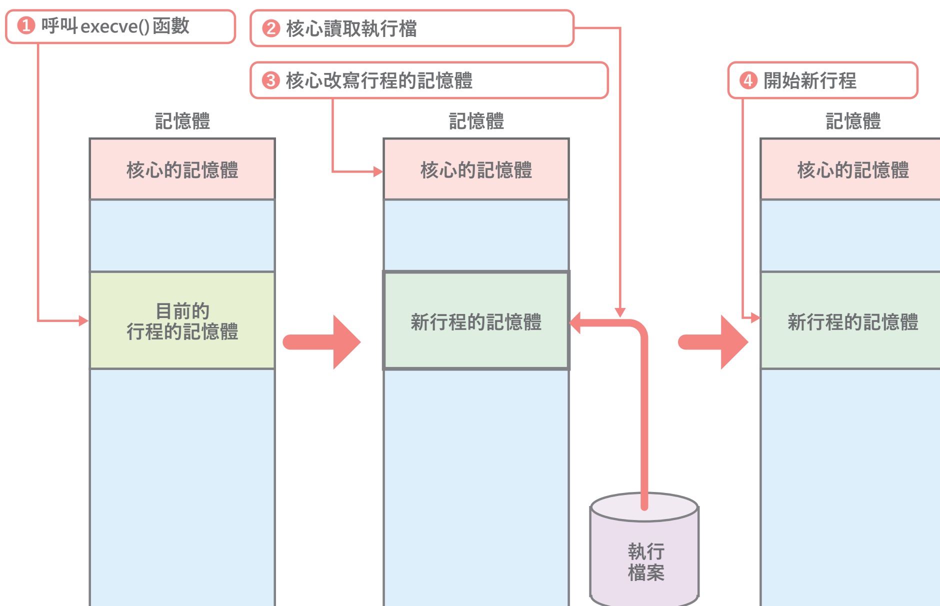
啟動其他程式的 execve() 函數

透過 fork() 函數建立行程的複製之後，會在子行程上發出 execve() 函數。藉著這個步驟，子行程會被置換成其他的程式。這一連串處理的流程如下。

1. 呼叫 execve() 函數
2. 從 execve() 函數的參數所指定的執行檔讀取程式，讀取用以配置到記憶體上（這稱為記憶體映射（memory map））所需要的資訊。
3. 將目前行程的記憶體以新行程的資料進行覆寫。
4. 將行程從新行程建立後首先必須執行的命令（入口點 (Entry Point)）來開始執行。

也就是說，對 fork() 函數而言，相較於行程數的增加，當要去建立其他的程式時，並不是採取增加行程數的方式，而是採用將某個行程以其他東西做置換的方式（圖 02-02）。

圖 02-02　透過 execve() 函數來置換成其他行程



將這個以程式形式呈現的就是 fork-and-exec.py 程式（列表 02-02）。在呼叫 fork() 函數後，子行程會透過 execve() 函數置換成 echo <pid> 大家好指令。

第 10 章

虛擬化功能

在實體機器上所安裝的 OS 上，再將其他 OS 安裝在虛擬機器上，像這種運用方式已經很普遍了。然而，筆者的認知是，理解這個實現方法的人似乎不多。

所以本章的目的，在於改善這種狀況，讓各位理解並接受「虛擬機器到底是什麼」。然而，虛擬化功能與第 4 章所說明過的虛擬記憶體是完全不同的。還真讓人混淆呢。

為了要能夠對虛擬化功能的機制有所理解，OS 及 OS 核心的知識是不可或缺的。

不過，各位讀者已經獲得本書中至前章為止的相關知識了。有鑑於此，相信各位讀者應該可以對本章的內容有近一步理解。如果遇到不清楚的地方，請適度地參考前章的內容。

什麼是虛擬化功能

虛擬化功能，是一個可在 PC 或伺服器等物理性機器上，運作虛擬機器所需的軟體功能，以及協助我們實現這點的硬體功能的組合。至於虛擬機器的用途，舉例來說，有下述這些項目。

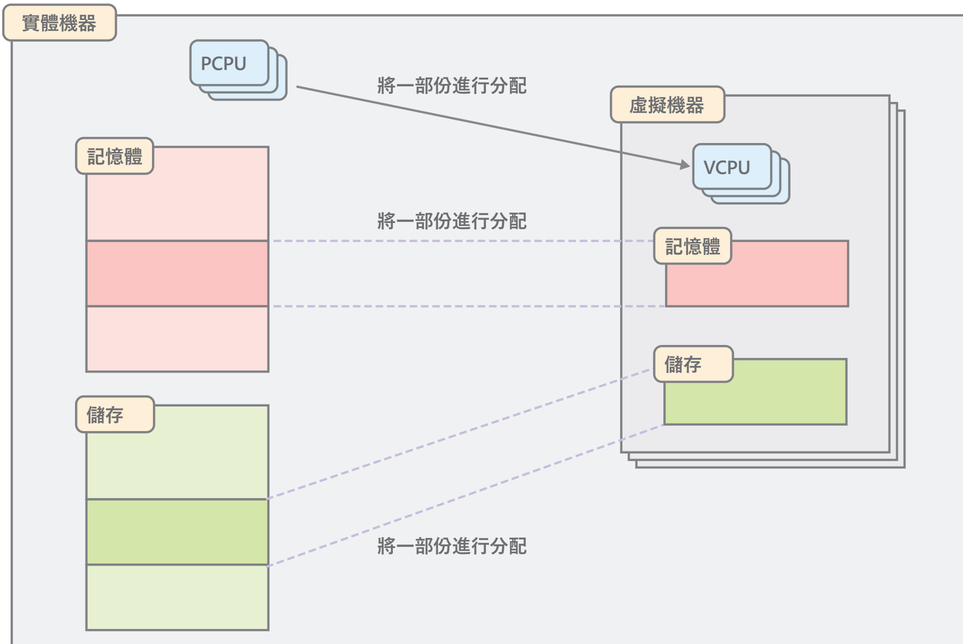
- 硬體的有效運用：在 1 台實體機器上運作複數個系統。在 1 台機器上建立有複數個虛擬機器的狀態下，向顧客提供租借服務的虛擬機器 IaaS (Infrastructure as a Service) 為其一例。
- 整合伺服器：將「由複數的實體機器所組成的系統」中的實體機器置換為虛擬機器，集中到更少的實體機器上。
- 延長傳統系統（Legacy System）的壽命：讓硬體支援已結束的舊系統在虛擬機器上運作[*1]。
- 在某個 OS 上運作別種 OS：在 Windows 上運作 Linux，反之亦然。
- 開發 / 測試環境：在沒有實體機器的情況下建構出與商務系統環境相同或是類似的環境。

舉例來說，筆者對於虛擬機器的使用方式如下所示。

- 在自家的 Windows 上運作 Linux。為了運作如遊戲或照片沖印等只對應到 Windows 的軟體，所以平常是使用 Windows，除此之外是使用 Linux。

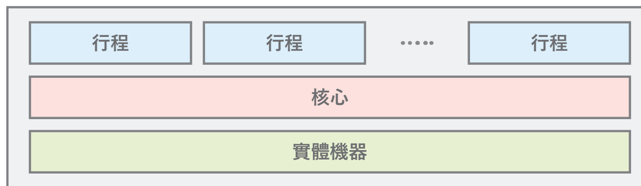
＊1　有時候虛擬機器也會將搭載於舊系統的軟體支援給關掉……。

圖 10-02 虛擬化軟體的構造



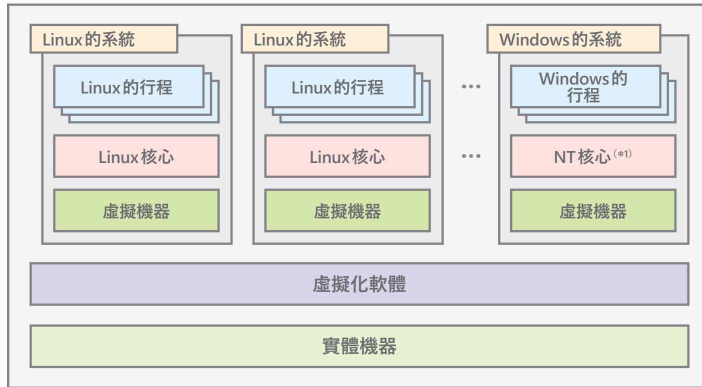
虛擬化軟體與虛擬機器之間的關係，跟核心的行程管理系統與行程之間的關係非常地相似。在實體機器上安裝 OS 時，系統的整體構造如圖 10-03 所示。

圖 10-03 在實體機器上安裝 OS 的情形



相較於此，在虛擬機器上安裝 OS 的情形，如圖 10-04 所示。

圖 10-04 在虛擬機器上安裝 OS 的情形



*1) Windows的核心部分的名稱

如各位所見，這是一個在虛擬化軟體的上方還有如圖 10-03 所示系統存在的構造。我們可得知除了在圖 10-04 上以「……」來表示省略的部分之外，還有 2 個 Linux 與 1 個 Windows 系統存在。虛擬機器與實體機器，除了裝置的組成之外並沒有什麼不同，只要是支援虛擬機器所提供的各種硬體的 OS 的話，都可以安裝。

實現虛擬化軟體的方法有很多種。舉例來說，有在物理硬體上直接安裝被稱為虛擬機管理程式（Hypervisor）的虛擬化軟體的方式，以及在既有 OS 上作為應用程式來運作的方式等存在。關於具體的軟體名稱，有名的如下所示。

- VMware 公司的各種產品
- Oracle 公司的 VirtualBox
- Microsoft 公司的 Hyper-V
- Citrix Systems 公司的 Xen

本章所使用的虛擬化軟體

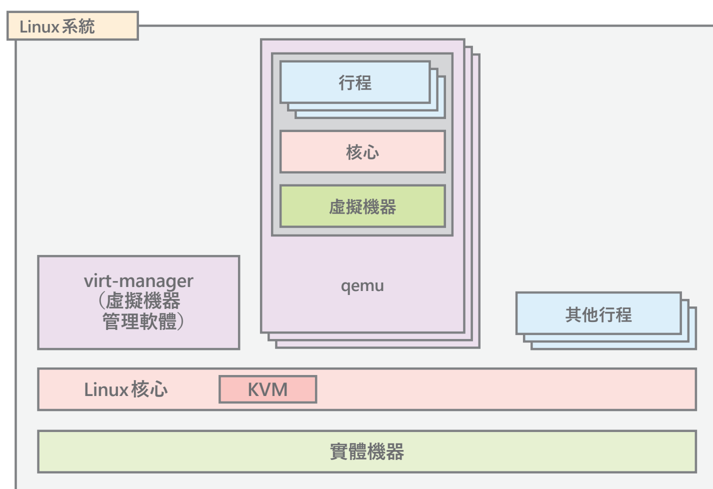
本章將透過下述 3 個軟體的組合，來建立、管理虛擬機器。

- KVM（Kernel-based Virtual Machine）：由 Linux 核心所提供的虛擬化功能。

- QEMU：CPU、硬體的模擬器。與 KVM 搭配使用的時候，不會使用 CPU 的模擬部分。
- virt-manager：建立、管理、移除虛擬機器。建立之後加以執行的部分就是 QEMU 的工作。

至於為什麼會選擇這個組合，是因為這些都屬於開放原始碼軟體（OSS），而且在大多的 Linux 發行版上的使用方式都很簡單。搭配上述軟體來做使用時，系統的結構如圖 10-05 所示。

圖 10-05 虛擬化系統結構範例



在這邊，通常會將在實體機器上運作的 OS 稱為「主機作業系統（host OS）」，而在虛擬機器上運作的 OS 稱為「客戶機作業系統（guest OS）」。

virt-manager 與 QEMU，對 Linux 核心來說只是普通的行程罷了。在虛擬化軟體的旁邊，一般的行程都可以運作。虛擬機器從建立到移除為止的流程如下所示。

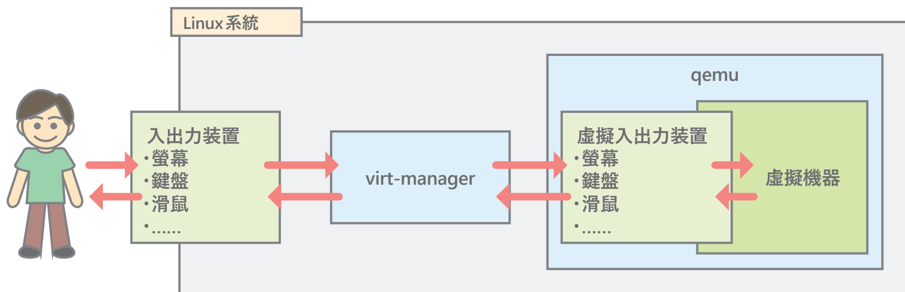
1. 由 virt-manager 建立新虛擬機器的雛型（CPU 的數量、記憶體容量、其他搭載硬體等）。
2. 由 virt-manager 根據上述雛型建立虛擬機器，並啟動 QEMU。
3. QEMU 與 KVM 協作，視需求運作虛擬機器（中間會遇到電源開啟／關閉或重新啟動）。
4. 由 virt-manager 將不需要的虛擬機器給移除。

virt-manager 可以對虛擬機器進行下述的操作。

- 從各虛擬機器所提供的視窗，顯示虛擬機器的螢幕輸出。
- 在上述視窗上，藉由鍵盤、滑鼠進行輸入時，可操作虛擬機器的鍵盤、滑鼠。
- 開啟／關閉虛擬機器的電源，進行重新啟動。
- 新增／刪除虛擬機器的裝置，將 iso 檔案掛載／卸載至虛擬 DVD 光碟機。

我們只需要想像成，各位讀者對於實體機器所做的事情，會由 virt-manager 代為對虛擬機器進行即可（圖 10-06）。

圖 10-06 virt-manager 的構造



Nested Virtualization

Column

到目前為止，我們有用到「在實體機器上運作虛擬機器」這樣的描述方式，不過實際在虛擬機器上，還有一個可以運作虛擬機器，名為「Nested Virtualization（巢狀虛擬化）」的功能存在。這個功能，是當我們想進行開發或測試時，可以在透過 IaaS 所借出的虛擬機器上，再建立一個虛擬機器的便利功能。

筆者所屬的 Cybozu, Inc. 公司的工作內容中，就有在 Google Compute Engine（Google 計算引擎）的虛擬機器上建構「由複數的虛擬機器所組成的虛擬資料中心[*a]」，以供 Continuous Integration（持續整合）等用途使用。

在使用到 Nested Virtualization 的情況下使用「實體機器」這個用語雖然是不太恰當的，不過，為了避免麻煩，本書會使用這個用語。是否可以使用 Nested Virtualization，會依 IaaS 或虛擬化軟體而異，至於各位讀者是否可在自己的環境上使用，需要自行查閱所使用的虛擬化軟體的使用手冊。

＊a　https://blog.cybozu.io/entry/2019/07/10/100000

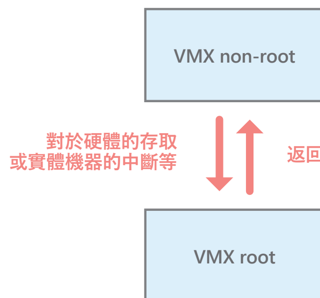
支援虛擬化的CPU功能

各位還記得在第 1 章所敘述到的，被分為使用者模式與核心模式的 CPU 功能嗎？如圖 10-07 所示，CPU 在運作行程的時候，是透過使用者模式來運作的。相較於此，以系統呼叫或中斷的發生為契機，運作核心的時候，是透過核心模式來運作的。在使用者模式下，從裝置的行程直接進行參照的話，不能被存取的資源會被設下存取限制，而另一方面，在核心模式下是什麼都辦得到。

圖 10-07　CPU 的模式轉換：核心模式與使用者模式

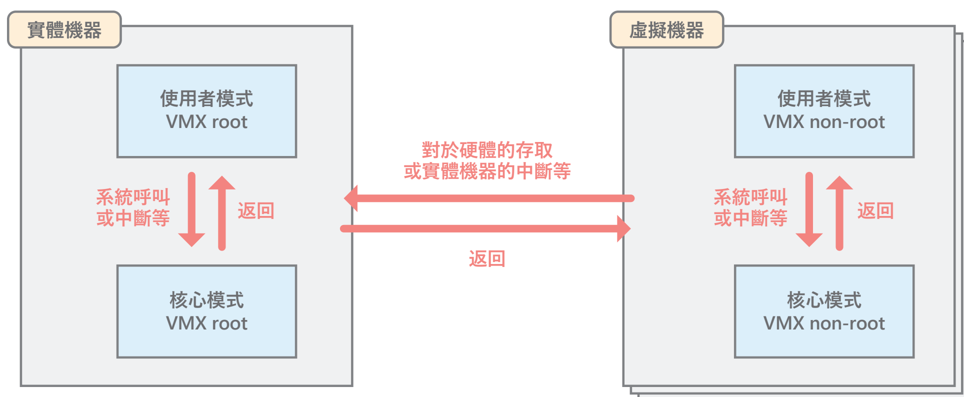
支援虛擬化功能的 CPU 則擴展了這個想法。具體來說，有一個在進行實體機器的處理時會用到的「VMX-root」模式，以及一個進行虛擬機器的處理時會用到的「VMX-nonroot」模式。在進行虛擬機器的處理時，對於硬體的存取或實體機器的中斷發生時，CPU 會回到 vmx-root 模式，自動地將控制移至實體機器上（圖 10-08）。

圖 10-08　CPU 的模式轉換：VMX-root 模式與 VMX-nonroot 模式



核心模式／使用者模式與 VMX-root 模式／VMX-nonroot 模式之間的關聯性如圖 10-09 所示。

圖 10-09 2 個種類的 CPU 模式



在 x86_64 架構的 CPU 上，如圖 10-09 所示 CPU 的虛擬化支援功能，於 Intel 公司的 CPU 上被稱為「VT-x」，於 AMD 的處理器上被稱為「SVM」。它們在功能上大同小異，不同之處在於用來實現功能的 CPU 層級的指令集上。不過，KVM 將這個差異給吸收了。這也是一個由核心所帶來的硬體功能的抽象化之一。

在各位讀者的環境上，VT-x 或 SVM 是否處於開啟狀態，可透過下述指令的發出來做確認。

```
$ egrep -c '^flags.*(vmx|svm)' /proc/cpuinfo
```

指令的輸出如果是 1 以上的話就代表開啟狀態，0 則代表關閉狀態。在這邊關於功能是否存在，不是以「有／沒有」來呈現，而是以「開啟／關閉」來呈現是有原因的。至於原因，CPU 本身雖然具有虛擬化功能，不過有時會透過 BIOS 被關閉功能，在這情況下，上述指令會將 0 返回。因此，輸出如果是 0 的話，為求謹慎還請各位確認一下 BIOS 的設定[*2]。

從下個章節開始，解說的部分原則上都會以虛擬化功能為開啟狀態的 CPU 為前提來進行說明。

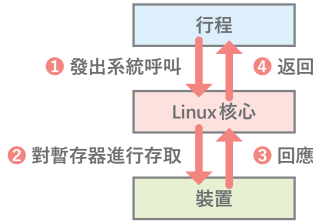
QEMU + KVM 的情形

在這章節中，我們將在虛擬機器上安裝有 Linux 的狀態下，來查看由 QEMU + KVM 所帶來的虛擬化的運作。

＊2 筆者曾在當 x86_64 CPU 開始搭載虛擬化功能時，購買了一台 CPU 搭載有該功能的 PC，不過在 BIOS 上是處於關閉狀態的，尚且，因為相關設定項目並不存在而使得筆者有過欲哭無淚的慘痛經驗。

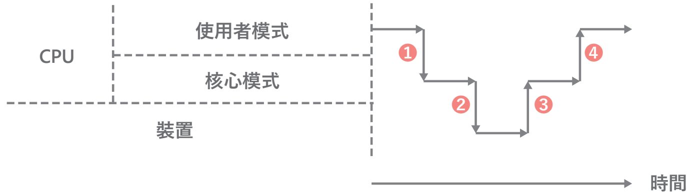
在這邊，讓我們透過由行程發出的系統呼叫，對某個裝置的暫存器進行存取的情形為例來做說明。實體機器上會如圖 10-10 所示。

圖 10-10 實體機器上的裝置存取



此時的 CPU 與裝置的處理流程如圖 10-11 所示。

圖 10-11 圖 10-10 中 CPU 與裝置的處理流程



這如果是虛擬化環境的話，便會如圖 10-12 與圖 10-13 所示。

圖 10-12 虛擬機器上的裝置存取