第9章　虛擬機器

對嵌入式系統的開發者而言，虛擬機器 (Virtual Machine) 是相當有用的工具。在本書中，我們用虛擬機一詞代表模擬指令集的軟體，而用模擬器一詞代表模擬電腦行為的軟體。

然而，有些軟體既會模擬指令集，又會模擬電腦的行為，像是 VMWare、Virtual PC、Virtual Box 這樣的軟體，也被我們視為是一種虛擬機器。在大部分的情況下，在我們不需要去區分虛擬機與模擬器的時候，我們會使用虛擬機器一詞統稱所有的模擬程式，而不去使用模擬器一詞。

在本章中，我們將介紹虛擬機的設計原理，並且說明如何撰寫一個 CPU0 的虛擬機器，以實際案例說明虛擬機器的運作原理。然後，在最後一節的實務案例中，我們將介紹 Java 的 JVM 虛擬機與 Virtual PC 虛擬機的使用方式，讓讀者實際操作這些虛擬機。

9.1 簡介

虛擬機器通常會在一台電腦上模擬出另一台電腦的指令與行為，例如我們可以在安裝了 ARMware 虛擬機之後，在處理器為 IA32 的 PC 上執行 ARM 處理器的執行檔。此時，ARMware 會模擬 ARM 的指令集，並表現出與 ARM 處理器相同的行為模式。

有了虛擬機，程式設計師可以不需要目標電腦硬體，改在虛擬機上測試程式，就好像在目標系統上測試一樣。藉由虛擬機，作業系統與嵌入式系統的開發者，可以輕易的測試程式，等到目標電腦的硬體被開發出來之後，才進行硬體的測試。如此，還可以讓軟體人員與硬體人員同時開發，以節省專案開發的時間。

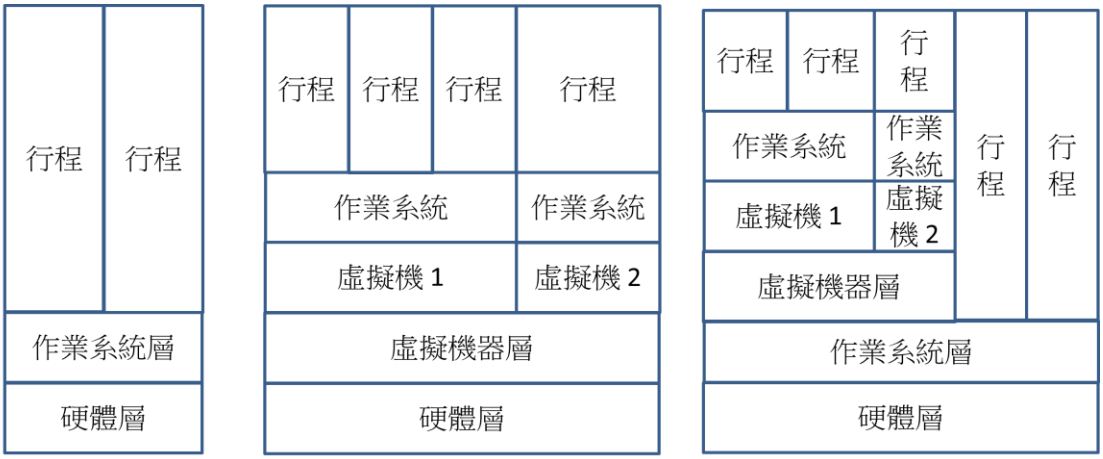
然而，虛擬機本身，就有很多不同的概念，很容易引起誤解。有些虛擬機模擬整個電腦硬體系統，因此被稱為『系統虛擬機』 (像是 Virtual PC)。有些則是單純的模擬指令集，但並不模擬真實的硬體系統環境，因此被稱為『程序虛擬機』 (像是昇陽 Java 的 JVM)。

系統虛擬機又有數種不同的結構，有些直接控制電腦硬體，稱為『原生式虛擬機』 (Native VM) 。有些則安裝在作業系統之上，透過作業系統才能接觸到硬體，稱

為『寄生式虛擬機』(Hosted VM)。圖 9.1 顯示了原生式虛擬機與寄生式虛擬機的架構差異，其中 (a) 是沒有虛擬機實的情況，而 (b) 則是原生式虛擬機的架構，(c) 則是寄生式虛擬機的架構。

目前我們經常使用的虛擬機，像是 VMWare、Virtual PC、Virtual Box、Wine、Bochs、Qemu 等，都屬於寄生式虛擬機。您可以在具有 Windows 作業系統的電腦上安裝 Virtual PC，然後再安裝一個 Linux 作業系統於 Virtual PC 的虛擬環境中，這讓 Linux 作業系統可以在 Windows 上執行，而不需要使用兩台電腦。

由於個人電腦盛行的關係，我們很少看到原生式虛擬機，在電腦發展的歷史上，IBM System/370 是一個很成功的原生式虛擬機架構，作業系統被安裝在虛擬機之上，因此可以在一台電腦硬體上安裝許多個作業系統，彼此之間完全獨立，即使其中一個作業系統垮了也不會影響到另外一個。



(a) 無虛擬機時　　(b) 原生式虛擬機　　(c) 寄生式虛擬機

圖 9.1 原生與寄生式的虛擬機

單純解譯指令集的虛擬機，通常被稱為程序虛擬機 (Process Virtual Machine) 也稱為應用層虛擬機 (Application Virtual Machine)，此種虛擬機可視為一種虛擬碼的解譯器，像是 Java 中的 Java Virtual Machine (JVM) 與微軟 .NET 所使用的 CLR 虛擬機器，其功能通常是為了讓程式碼跨越平台執行而設計的，並沒有對映到真實存在的電腦硬體或 CPU，因為對作業系統而言，這種虛擬機只是一個應用程式而已。

虛擬機器的概念並非近年來才出現的，早在 1967 年，IBM 就在 CP-40 這台機器上設計了 CP/CMS 這個虛擬機，並且發展出後續的 VM 家族機器，成功的在商業上運用了虛擬機器的概念，這種機器就屬於上述的 Native VM，是一種原生

式的虛擬機。

同樣的，程序虛擬機的概念也並非近年來才有的，早在 1975 年 Pascal 的發明人 Nichlaus Wirth 就設計出了 P-code 虛擬機，用以執行 Pascal 編譯出來的虛擬碼。1995 年 Java 在網路風潮時利用類似 P-code 虛擬機的概念，達成跨平台程式運行的目標，重新炒熱了此一主題。隨後，微軟也於 .NET 平台當中再度實作此一概念，稱為 Common Language Runtime (CLR)。

9.2 中間碼

編譯器所產生的中間碼，通常我們稱為 p-code 或 Pseudo Code，此種 p-code 其實就是一種具有跨平台能力的虛擬機器碼，相當類似於 JAVA 的 JVM 當中所使用的 bytecode，這是本節所要討論的主題。

在前一章的編譯器主題當中，我們曾經看過一種 p-code 中間碼，假如我們撰寫一個程式以解譯這些中間碼，並根據中間碼進行對應的模擬動作，那麼這樣的一個程式就被稱為虛擬機，所以，虛擬機其實就是一種中間碼的解譯器。

在任意的電腦硬體平台上，我們只要具有執行這些中間碼的虛擬機，就可以在不同架構的電腦上執行相同的中間碼。於是，這些中間碼就成了一種可以跨越平台執行的程式碼，這就達成了 Java 當初所提出來的理想 Write Once – Run Anywhere，跨平台執行的目的。

指令集的設計對 CPU 的架構有重大影響，同樣的，中間碼的設計也決定了虛擬機的架構。目前，在虛擬機的指令設計上，大致有三種策略，第一種是採用變數直接存取的方式，這種機器被我們稱之為記憶體機 (Memory Machine)，第二種是採用以暫存器為主的架構，這種虛擬機被我們稱為暫存器機 (Register Machine)，而第三種則是採用堆疊式的架構，我們稱之為堆疊機 (Stack Machine)。

讓我們看看這三種機器的中間碼會有何不同，範例 9.1 分別顯示了這三種中間碼，三個程式都是實作 sum=1+2+…+10 的虛擬機組合語言，但是由於架構的不同，也導致程式內容差異甚大。

在範例 9.1 當中，讀者可以看到 (a) 欄記憶體機的組合語言，這也就是我們在前一章中所看到的 p-code，由於 p-code 指令可以直接對記憶體變數進行運算，因此可以用 p-code 作為記憶體機的組合語言。

範例 9.1 三種虛擬機的組合語言

(a) 記憶體機

```
    =   0       sum
    =   0       i
FOR0:
    CMP     i   10
    J   >       _FOR0
    +   sum i   T0
    =   T0      sum
    +   i   1   i
    J           FOR0
_FOR0:
    RET         sum
```

(b) 暫存器機

```
    LDI   R1, 0
    ST    R1, sum
    LDI   R2, 0
    ST    R2, i
    LDI   R3, 1
    LDI   R4, 10
    CMP R2, R4
    JGT   _FOR0
    ADD R1, R1, R2
    ADD R2, R2, R3
    JMP _FOR0
    ST    R1, sum
    RET
i:  RESW    1
sum:RESW    1
```

(c) 堆疊機

```
    PUSH 0
    POP sum
    PUSH 0
    POP i
FOR0:
    PUSH i
    PUSH 10
    CMP
    PUSH _FOR0
    JGT
    PUSH sum
    PUSH i
    ADD
    POP sum
    PUSH i
    PUSH 1
    POP i
_FOR0:
    PUSH sum
    RET
i:  RESW    1
sum:RESW    1
```

在範例 9.1 的 (b) 欄當中，顯示了 CPU0 的組合語言，由於 CPU0 是一個以暫存器為主的處理器，因此只要寫一個程式以解譯 CPU0 的指令集，就相當於實作了一個暫存器機。

在範例 9.1 的 (c) 欄當中，所顯示的組合語言就是我們所未曾見過的了，這種組合語言相當特別，除了 PUSH 與 POP 之外，其他指令幾乎都沒有參數，而這也正是堆疊機的主要特性，將所有參數都存入堆疊當中，然後才進行運算。以下，我們將更詳細的說明堆疊機的架構與運作原理。

堆疊機

圖 9.2 顯示了堆疊機的架構圖，在堆疊機的 CPU 中，有一個理論上容量無限的

堆疊結構，可以讓指令先將參數推入後，再對堆疊最上層的元素進行運算。

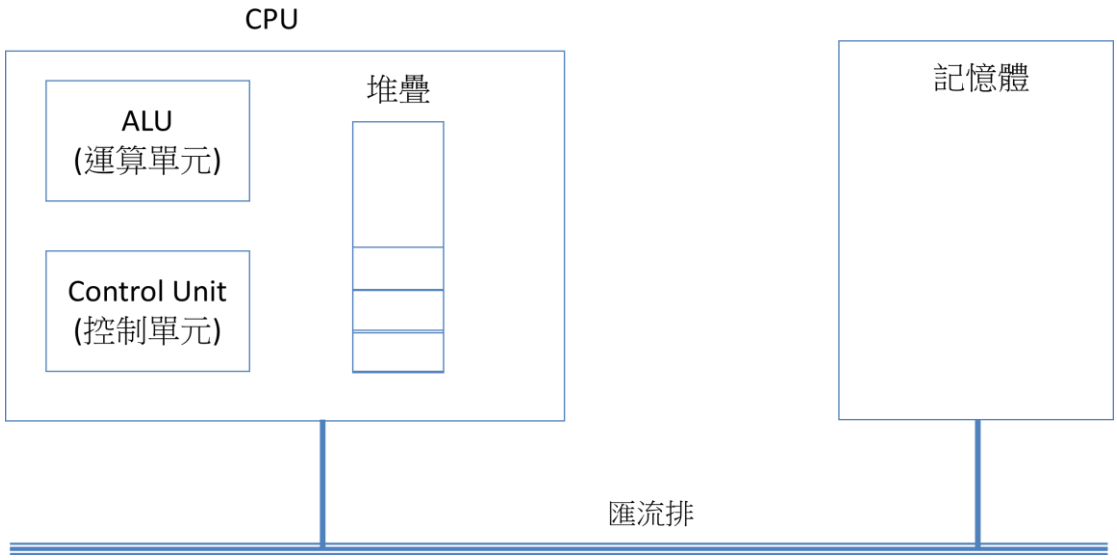


圖 9.2 堆疊機 (Stack Machine) 的架構圖

舉例而言，我們可以先利用 PUSH 6; PUSH 2; PUSH 3; 等指令，將參數推入堆疊後，然後再利用 ADD 與 SUB 指令，執行 6-(2+3) 的動作，其程式如範例 9.2 所示。

範例 9.2 堆疊機的組合語言

堆疊機組合語言	說明
PUSH 6	推入 6 到堆疊中
PUSH 2	推入 2 到堆疊中
PUSH 3	推入 3 到堆疊中
ADD	將 3 與 2 相加 = 5
SUB	將 6 與 5 相減 = 1

範例 9.2 的執行過程如圖 9.3 所示，前三個 PUSH 指令乃是將 6, 2, 3 推入堆疊當中，後面的 ADD 則進行 2+3=5 的動作，此時堆疊會剩下 6,5 兩個元素，接著在執行 SUB 時，就會導致堆疊中只剩下 6-5=1 的結果，於是完成了 6-(2+3) 的計算過程。

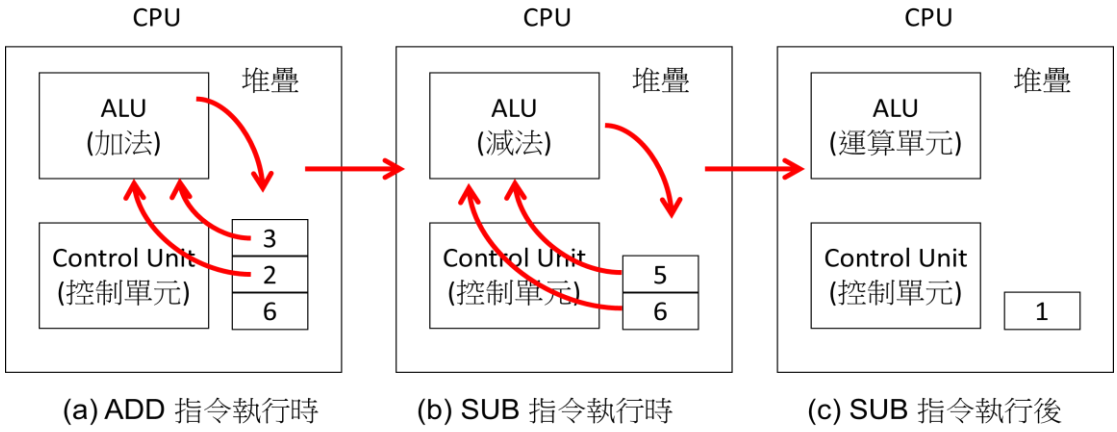


圖 9.3<範例 9.2> 的堆疊機執行過程

堆疊機的架構在今日越來越顯得重要，因為 Java 的 JVM 與微軟的 .NET 等重要平台全都採用了此一堆疊式的架構，但是堆疊機並不是最近才被發明出來的。

1978 年的 UCSD Pascal 的虛擬機 UCSD p-Machine 是最廣為人知的 pcode 虛擬機，該虛擬機就是一種採用堆疊架構的堆疊機。堆疊機的好處是很容易移植到不同的處理機上，因為在電腦硬體中，暫存器的數量有多有少，並沒有一個定值。堆疊機將暫存變數儲存在堆疊中，因此，不需要考慮有多少暫存器的問題。Nichlaus Wirth 利用堆疊機的設計方式，有效的縮短了指令的長度。

堆疊機的架構在今日越來越顯得重要，因為 Java 的 JVM 與微軟的 .NET 等重要平台全都採用了此一堆疊式的架構。甚至，有一種低階的程式語言稱為 Forth，其設計原理完全採用堆疊機的架構，Forth 語言常被用來撰寫硬體相關的低階程式。

在本節中，我們已經介紹完虛擬機的中間碼，以及三種虛擬機的指令設計方式，接著，讓我們說明如何才能用程式撰寫出一個真實的虛擬機，我們將在下一節當中說明虛擬機的演算法。

9.3 CPU0 的虛擬機

為了更進一步說明虛擬機的設計原理，我們將以 CPU0 為例，說明如何設計可執行 CPU0 機器碼的的虛擬機。透過這種方式，我們可以在任何一個實體機器上架構出虛擬的 CPU0 處理器，讓任何電腦都可以執行 CPU0 的機器碼。

說穿了，只要能解譯機器碼，模擬其指令執行方式，就能撰寫出虛擬機。當載入

器將目的碼載入到記憶體的某個型態為 byte 的陣列變數中後，就可以從起始指令開始，逐步解譯指令並模擬指令的執行方式，這樣就能將硬體化為軟體，讓虛擬機的行為與真實的處理器一致。

為了要模擬 CPU0 的指令，我們必須逐一解讀 CPU0 中的每一種指令，其動作幾乎就如附錄 A 中的 CPU0 指令表一樣，我們只要根據該指令表實作每一個指令的動作即可。

但是，指令表所給的是單一指令的特殊動作，在 CPU0 處理器中，每個指令都有一些共同的動作，像是指令提取時，必須將指令從記憶體載入到 IR 暫存器當中，而且在提取完成後，必須將程式計數器 PC 加上 4 ，這樣才能完成整個指令的模擬的動作。

在圖 9.4 中，我們寫出了 CPU0 虛擬機的演算法，若讀者想進一步瞭解 CPU0 虛擬機的實作方法，可以參考第 12 章中的 CPU0 虛擬機實作一節，以及本書所附光碟中的 ch12/CPU0.c 程式，以進一步理解詳細的實作原理。

CPU0 虛擬機器的演算法	說明
// Virtual Machine for CPU0	CPU0 的模擬器類別
// global variable	共用變數
byte m[]	m 為記憶體
int32 IR	IR 為指令暫存器
int32 R[16]	R[] 為一般暫存器
PC is R[15]	PC 為 R[15] 的別名
LR is R[14]	LR 為 R[14] 的別名
SP is R[13]	SP 為 R[13] 的別名
// function	函數區
Algorithm run(objFile)	模擬執行目的檔
startAddress = readObjFile(objFile, m)	讀取目的檔到記憶體中
PC = startAddress	設定程式計數器為起始位址
LR = -1	設定連結暫存器為-1，跳離用
stop = false	設定停止旗標為尚未停止
while not stop	進入迴圈，開始執行指令
R[0] = 0	R[0] 暫存器永遠為 0
load IR from m[PC..PC +3]	擷取指令到 IR 暫存器中
PC=PC+4	將 PC 加 4 (下一個指令)
op = bits(IR, 24..31)	取出指令碼 op
ra = bits(IR,20..23)	取出暫存器代號 ra

rb = bits(IR, 16..19)	取出暫存器代號 rb
rc = bits(IR, 11..15)	取出暫存器代號 rc
c5 = bits(IR, 0..4);	取出位元 0..5 放入 c5 中
c12 = bits(IR, 0, 11)	取出位元 0..11 放入 c12 中
c16 = bits(IR, 0, 15)	取出位元 0..15 放入 c16 中
c24 = bits(IR, 0, 23)	取出位元 0..23 放入 c24 中
if (bits(IR, 11, 11)!=0) c12 |= 0xFFFFF000;	處理 c12 可能為負數的情況
if (bits(IR, 15, 15)!=0) c16 |= 0xFFFF0000;	處理 c16 可能為負數的情況
if (bits(IR, 23, 23)!=0) c24 |= 0xFF000000;	處理 c24 可能為負數的情況
caddr = R[rb]+c16	計算 LD, ST,… 的位址欄
raddr = R[rb]+R[rc]	計算 LDR,STR, … 的位址欄
switch op	根據指令碼 op 跳到對應程式
LD: load R[ra] from m(caddr..caddr+3)	處理 LD 指令
ST: store R[ra] into m(caddr..caddr+3)	處理 ST 指令
LB : load R[ra] form m[caddr]	處理 LB 指令
SB : store R[ra] into m[caddr]	處理 SB 指令
LDR : load R[ra] from m[raddr..raddr+3]	處理 LDR 指令
STR : store R[ra] into m[raddr..raddr+3]	處理 STR 指令
LBR : load R[ra] from m[raddr]	處理 LBR 指令
SBR : store R[ra] into m[raddr]	處理 SBR 指令
LDI : R[ra] = cx	處理 LDI 指令
CMP : cc = compare(R[ra], R[rb]); set cc into bits(SW, 31..30)	處理 CMP 指令
MOV : R[ra] = R[rb]	處理 MOV 指令
ADD : R[ra] = R[rb]+R[rc]	處理 ADD 指令
SUB : R[ra] = R[rb]-R[rc]	處理 SUB 指令
MUL : R[ra] = R[rb]*R[rc]	處理 MUL 指令
DIV : R[ra] = R[rb] / R[rc]	處理 DIV 指令
AND : R[ra] = R[rb] and R[rc]	處理 AND 指令
OR : R[ra] = R[rb] or R[rc]	處理 OR 指令
XOR : R[ra] = R[rb] xor R[rc]	處理 XOR 指令
ROL : R[ra] = R[rb] rol c5	處理 ROL 指令
ROR : R[ra] = R[rb] ror c5	處理 ROR 指令
SHL : R[ra] = R[rb] shl c5	處理 SHL 指令
SHR : R[ra] = R[rb] shr c5	處理 SHR 指令
JEQ : if (cc is =) PC = PC +c24	處理 JEQ 指令
JNE : if (cc is not =) PC = PC+ c24	處理 JNE 指令
JLT : if (cc is <) PC = PC+ c24	處理 JLT 指令

JGT : if (cc is >) PC = PC+ c24	處理 JGT 指令
JLE : if (cc in {<, =}) PC = PC+c24	處理 JLE 指令
JGE : if (cc in {>,=}) PC = PC+ c24	處理 JGE 指令
JMP : PC = PC+c24	處理 JMP 指令
SWI : R[14] = PC; PC= c24	處理 SWI 指令
JSUB : R[14] = PC; PC=PC+ c24	處理 JSUB 指令
RET :	處理 RET 指令
if (R[14]<0)	如果連結暫存器<0
stop=true	則跳離
else	否則
PC=LR	返回上一層
PUSH : SP= SP-4;	處理 PUSH 指令
store R[ra] into m[SP..SP+3]	
POP : SP=SP+4;	處理 POP 指令
load R[ra] from m[SP..SP+3]	
PUSHB : SP=SP-1;	處理 PUSHB 指令
store R[ra] into m[SP]	
POPB : SP=SP+1;	處理 POPB 指令
load R[ra] from m[SP]	
end switch	
print PC, IR, R[ra]	印出相關暫存器以便觀察
end while	
dumpRegisters()	印出所有暫存器以便觀察
End Algorithm	

圖 9.4 CPU0 的虛擬機之演算法

讀者從圖 9.4 的演算法當中應當可以看出，要撰寫一個虛擬機並不困難，只是有點煩瑣，需要針對每個指令進行模擬而已。

撰寫虛擬機並不困難，但是若要讓虛擬機的執行速度非常快，就必須花費一番功夫了。利用 C 語言撰寫虛擬機，然後利用最佳化機制，再加上內嵌組合語言，或者乾脆完全改用組合語言撰寫，甚至像 Java 的 JVM 虛擬機使用即時編譯 (Just in time compiler) 機制，在執行時期將中間碼轉換為機器碼後，直接以機器碼的方式執行等等，都是可以用來加速虛擬機速度的方法，這些議題本書就無法涵蓋了，有興趣的讀者請進一步參考相關文獻。

9.4 實務案例(一)：Java 的 JVM 虛擬機

Java 雖然是一個程式語言，但是由於設計訴求上的跨平台目標，使得 Java 特別適合用虛擬機來執行，由於 Java 已經是廣為使用的程式語言，因此，其虛擬機也就變得相當重要。目前最常見的 Java 虛擬機是昇陽的 JVM 虛擬機，但是在開放原始碼界還有 Kaffe、Jikes、Novell Mono, Apache Harmony, Google Dalvik, …等數十個虛擬機，JVM 並非唯一的 Java 虛擬機。

假如使用者撰寫了一個 Java 程式 (例如：HelloWorld.java)，此時，可以利用 Java 的編譯指令 (例如：javac HelloWorld.java)，將該程式編譯成 bytecode (例如：HelloWorld.class)。接著，再利用 Java 的載入指令 (例如：java HelloWorld)，呼叫 Java 的虛擬機器 JVM，將 bytecode 載入到記憶體當中解譯並執行。若您的電腦當中安裝了 Java 的 JDK 開發工具，您可以按照範例 9.3 的方式，將一個 Java 程式編譯成 bytecode，然後執行看看。

範例 9.3 Java 的程式與編譯執行過程

(a) Java 程式 HelloWorld.java	(b) 編譯與執行過程

class HelloWorld {
  public static void main (String[]) {
  println("Hello World!");
  }
}
	D:\ch10>javac HelloWorld.java
	
	D:\ch10>java HelloWorld
	Hello World!

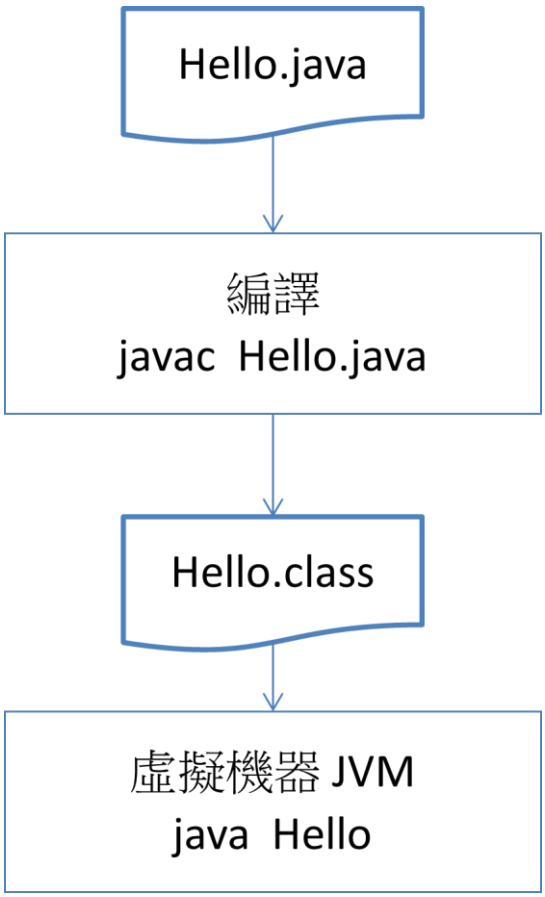


圖 9.5 Java 程式的編譯與執行方式

Java 的 Bytecode 可以直接在網路上流通，被執行前並不需要重新編譯，JVM 虛擬機會利用 Bytecode 解譯器執行 bytecode，或利用即時編譯器 (Just in Time Compiler : JIT) 將 Bytecode 進一步轉換成該電腦上的機器碼執行，圖 9.6 顯示了 Bytecode 的這種跨平台特性。

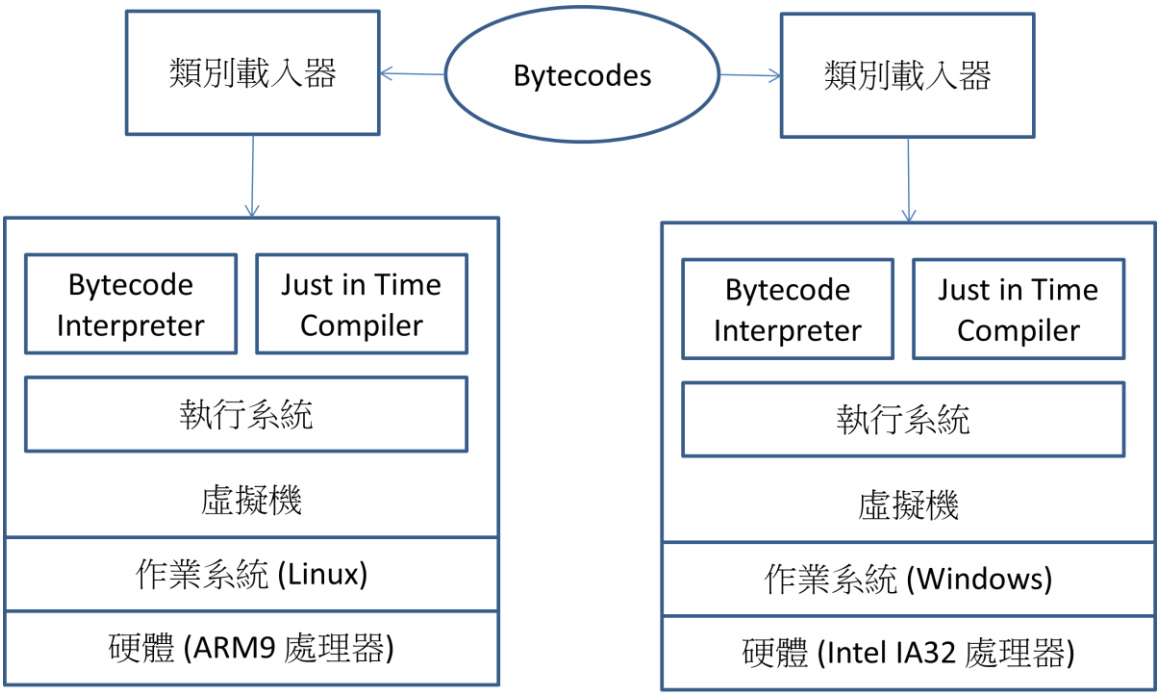


圖 9.6 利用虛擬機讓 Bytecode 跨平台執行

由於虛擬機本身就是一種軟體，所以在動態載入技術的實作特別簡單，因為載入器通常就屬於虛擬機的一部分，只要利用載入器載入目的檔後，再用虛擬機執行該目的碼即可。

舉例而言，範例 9.4 中的兩行 Java 程式會動態的載入一個由變數 name 所指定的 byte code 檔案，然後利用該檔案建立出對應的物件。這樣的技術讓 Java 程式可以在詢問使用者之後，再決定要載入哪一個類別，如此，就不需要在一開始時就載入所有的函式庫，達成動態連結與動態載入的效果。甚至可以在需要的時候，才透過網路從另一台電腦中下載目的碼並執行，這也是當初 Java Applet 的用意，但可惜的是在瀏覽器大戰中 JVM 的技術遭到敗北的命運。但是後來 Adobe Macromedia 公司的 Flash 技術，基本上也是一種瀏覽器中的虛擬機，而且成功的跨越瀏覽器執行，達成跨越網路執行的效果，Flash 可以視為 Java Applet 的復活版。

範例 9.4 Java 當中動態載入技術的範例

```
…
Class type=ClassLoader.getSystemClassLoader().loadClass(name);
Object obj = type.newInstance();
…
```

如果將虛擬機技術與本書 1-8 章中的系統程式主題進行對照，那會形成一個很

好的對照，Java 的 javac 編譯器可以對照到 C 語言的 gcc 編譯器，bytecode 相當於虛擬機器上的目的檔，可對照到 ELF 格式的目的檔，虛擬機器 JVM 則可對照到真實的 CPU。

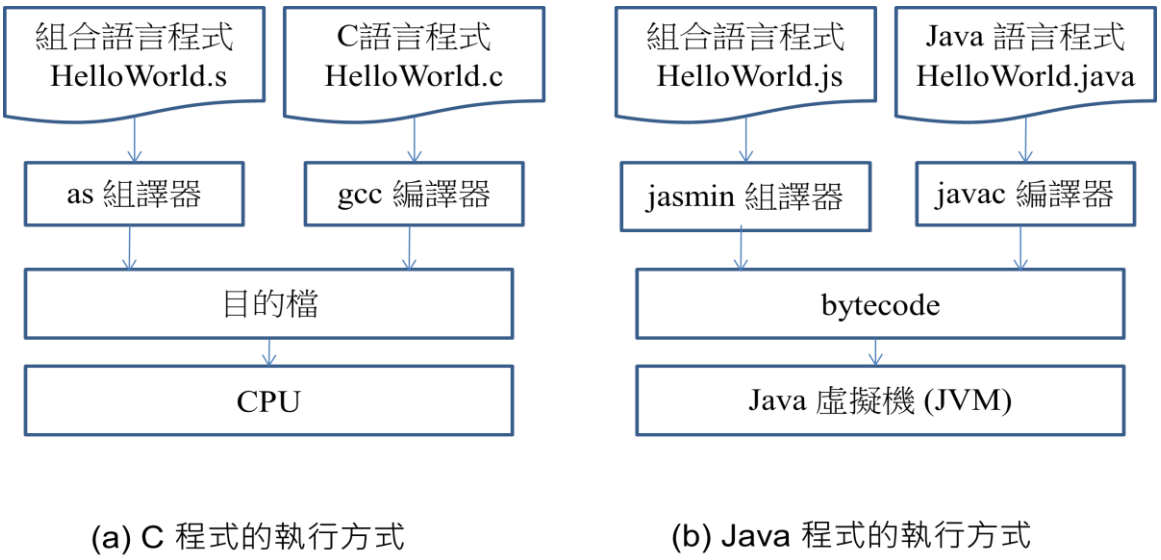


圖 9.7 Java 與 C 程式的執行模式對照

雖然昇陽沒有提供 Java 組合語言的組譯器，但是您可以在網路上找到一些，例如像是 Jon Meyer's 撰寫的 Jasmin 就是一個 java 的組譯器，這種組譯器提供了 bytecode 的組譯功能。這些工具可以幫助有興趣的讀者進一步的理解 Java 的組合語言。

在昇陽公司所提供的 Java 開發工具 JDK 中，雖然不包含 Java 的組譯器，但是包含了反組譯器 javap，範例 9.5 顯示了 javap 反組譯器的用法，讀者可以用 javap 觀察 java byte code 所對應的組合語言，以學習 Java 的組合語言寫法。

範例 9.5 利用 javap 指令將 bytecode 反組譯成組合語言

指令 javap -c HelloWorld 的輸出結果

```
Compiled from "HelloWorld.java"
class HelloWorld extends java.lang.Object{
HelloWorld();
  Code:
    0:    aload_0
    1:    invokespecial    #1; //Method java/lang/Object."<init>":()V
    4:    return
```

```
public static void main(java.lang.String[]);
  Code:
  0:    getstatic    #2; //Field java/lang/System.out:Ljava/io/PrintStream;
  3:    ldc          #3; //String Hello World!
  5:    invokevirtual #4;//Method java/io/PrintStream.println:(Ljava/lang/String;)V
  8:    return
}
```

9.5 實務案例(二)：微軟的 Virtual PC 虛擬機

在 Intel x86 電腦上常用的硬體虛擬機有 VMWare、Virtual PC 與 Virtual Box 等，其中，VMWare 功能較強大，但是需要付費購買，而 Virtual PC 與 Virtual Box 則完全免費，在本節當中，我們將使用 Virtual PC 作為範例，說明系統虛擬機的用法。

圖 9.8 顯示了利用 Virtual PC 虛擬機在 Windows XP 作業系統當中執行 Red Hat Linux 9.0 的情況。此時，Red Hat Linux 9 在 Windows XP 系統當中看起來像是一個應用程式，但實際上卻是一個完整的作業系統，只是其畫面並沒有占據整個螢幕而已，利用此種方式，我們可以在『宿主作業系統』的環境下『安裝』另一個『寄生作業系統』，而此『寄生作業系統』看起來就好像只是一個應用程式一般，但其運作行為與一個單獨的作業系統幾乎一模一樣。

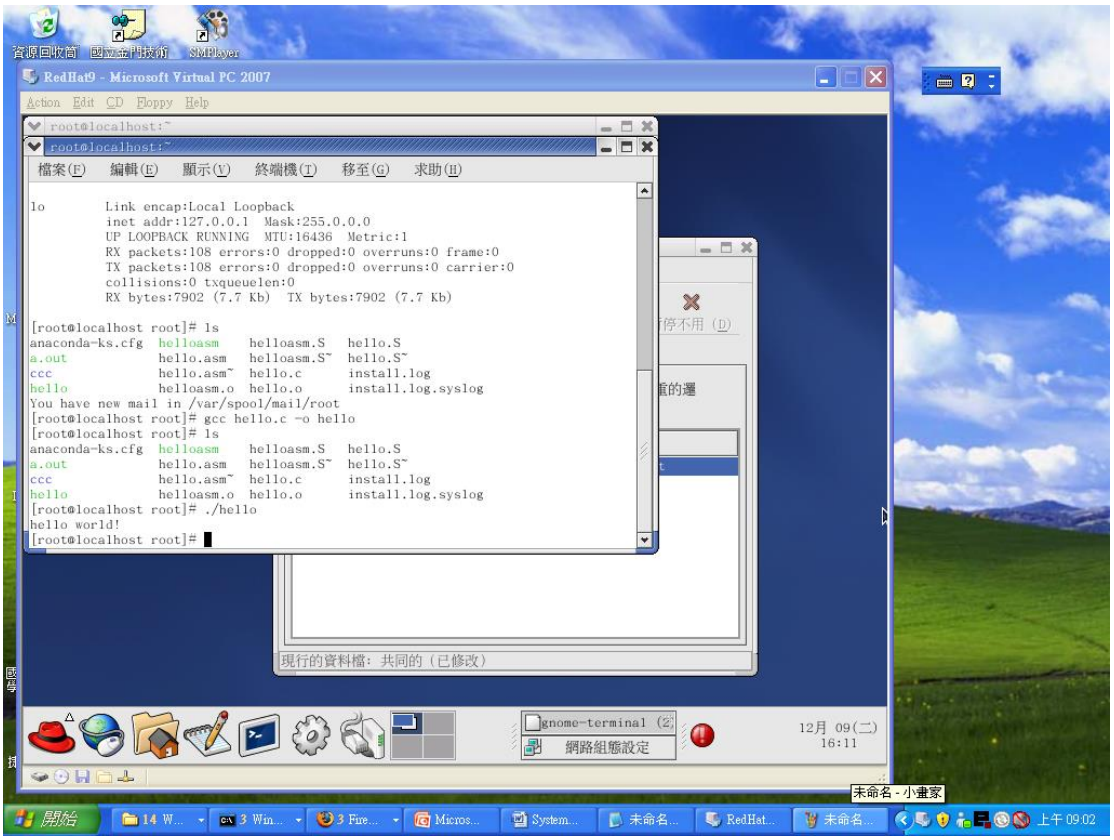


圖 9.8 在 Microsoft Windows 當中以 Virtual PC 軟體執行 Red Hat 9.0 的情況。

為了展示虛擬機的使用過程，我們將在 Windows XP 當中，以 Virtual PC 安裝一個寄生的 『DOS 作業系統』，以便說明虛擬機的使用方式。

選擇以『DOS 作業系統』的原因是，DOS 系統相當的小，因此安裝過程快速，如此，可以方便使用者練習 Virtual PC 的使用方式。而且，由於 DOS 作業系統乃是一個簡易的作業系統，沒有安全保護機制，因此，對於開發新作業系統的人員而言，是一個很好的練習平台，因此，我們將以 DOS 系統作為練習對象。

首先，請讀者從網路上下載 Virtual PC 2007 的版本[1]，安裝後請啟動之，讀者會看到如圖 9.9 的 Virtual PC Console 啟動視窗。

[1] 您可以從網址 http://www.microsoft.com/windows/downloads/virtualpc/default.mspx 當中下載 Virtual PC 2007, 筆者下載日期為 2009-03-17。

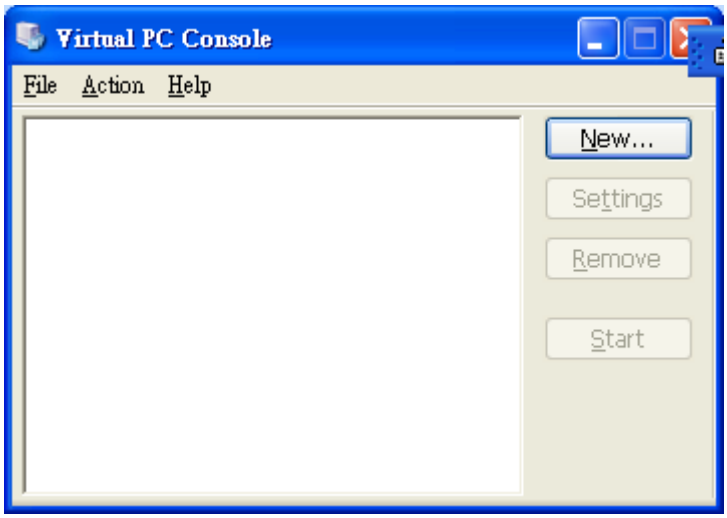


圖 9.9 Virtual PC 的啟動視窗

接著，請按下 New 按鈕，此後，會有一連串的選擇畫面，所有選項都請以預設值設定即可，但是在虛擬機儲存檔中您可以將檔案命名為 DOS 6.22.vmc，如圖 9.10 所示，以方便辨識。

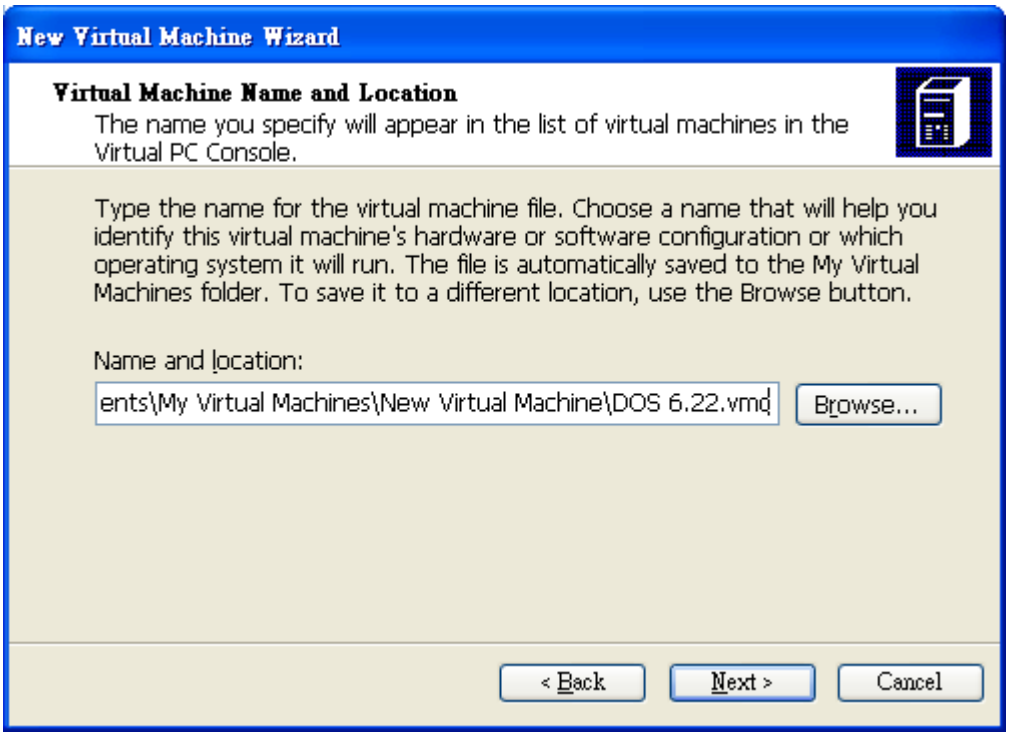


圖 9.10 Virtual PC 虛擬機的存檔畫面

接著，在 Virtual Hard Disk Options 畫面時，請選取 A new virtual hard disk，如此，才會建立一個新的虛擬硬碟，請讀者參考圖 9.11 的設定。最後，按下 Finish 完成後，即建立的一台新的虛擬機器。

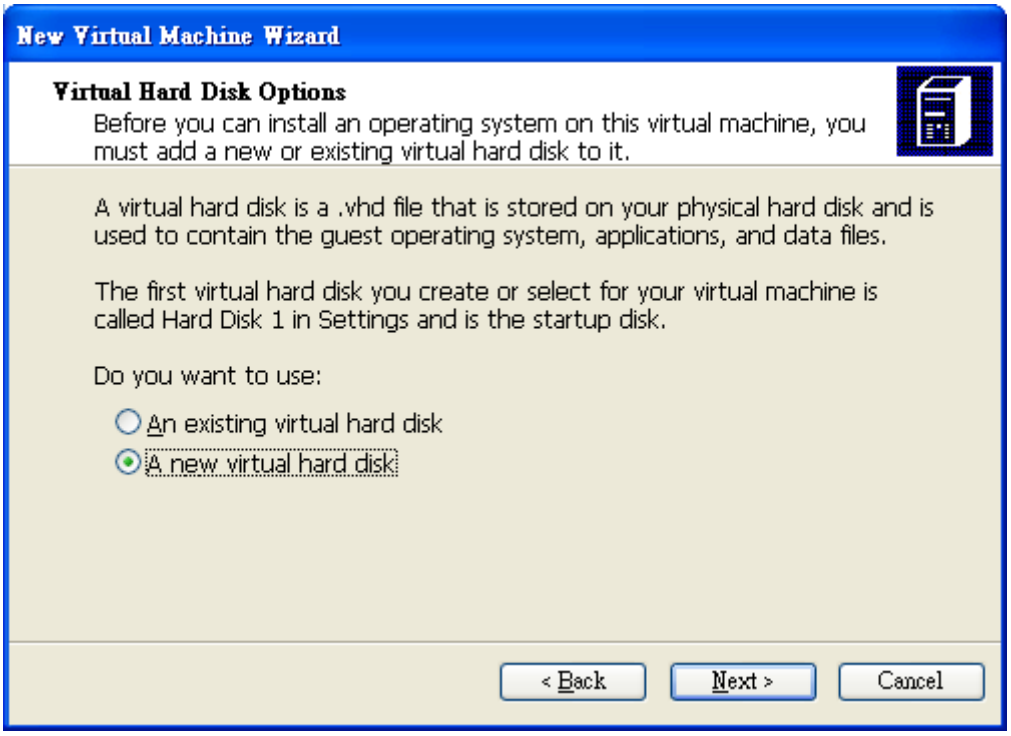


圖 9.11 選擇 A new virtual hard disk 以建立新的虛擬硬碟。

然而，到目前為止，我們所建立的虛擬機器是不具有作業系統的『硬體』模擬機，要在 Virtual PC 當中安裝 DOS 作業系統，首先我們必須取得 DOS 的開機磁片（軟碟片），但是，這對許多今日的電腦使用者而言，將會是一大困擾。因為，現在許多電腦都已不再具有軟碟機了。還好，在 Virtual PC 當中，不需要軟碟機也可以安裝 DOS 系統，只要先取得 DOS 開機磁片的映像檔即可，此硬像檔可以在網路上輕易的下載取得[2]。

當您取得 DOS 開機映像檔 (例如： 622C.IMG) 之後，請先於 Virtual PC Console 中，按下 Start 按鈕，啟動方才所建立的『硬體』虛擬機，如圖 9.12 所示。

[2]在 bootdisk 這個網站的網頁 http://www.bootdisk.com/bootdisk.htm 當中，可以取得下列壓縮檔 http://s93616405.onlinehome.us/bootdisk/622c.zip，解壓縮後即有 622C.IMG 此一軟碟映像檔，筆者下載日期為 2009-03-17。

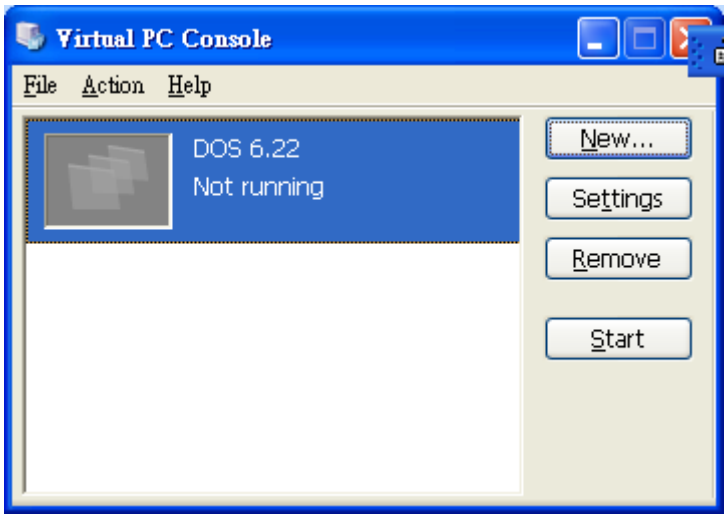


圖 9.12 請按下 Start 鍵啟動 DOS 虛擬機

此時的狀況，就好像您將一台未安裝作業系統的電腦開機一樣，其畫面如圖 9.13 所示。

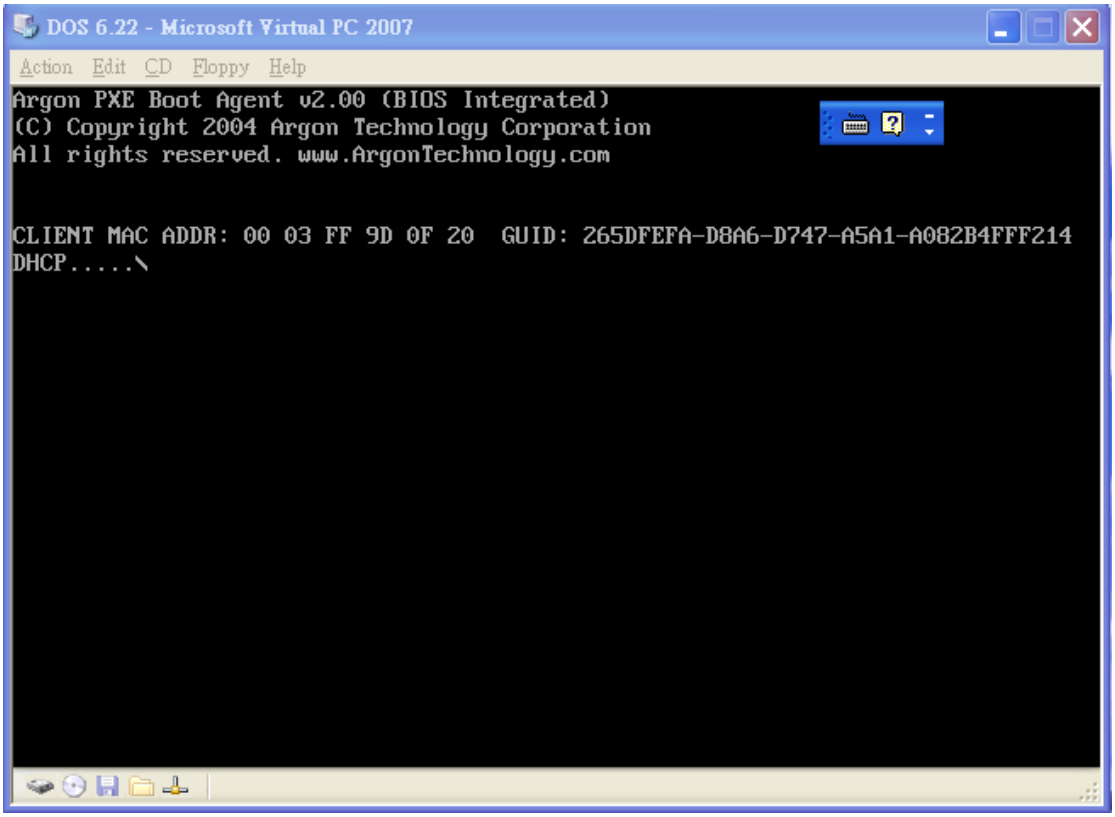


圖 9.13 Virtual PC 的虛擬機之啟動畫面

此時，您必須先插入安裝磁片或光碟，才能將作業系統安裝上去。但是，由於我們想要利用軟碟映像檔安裝 DOS，而不是從真正的軟碟機安裝，因此，我們可以選擇 Floppy/Capture Floppy Disk Image，並選取 DOS 的開機映像檔，以便指

定由該映像檔開機，如圖 9.14 所示，然後按下 Action/Reset 以重新開機，如此，就完成了 DOS 系統的虛擬機設定動作了。

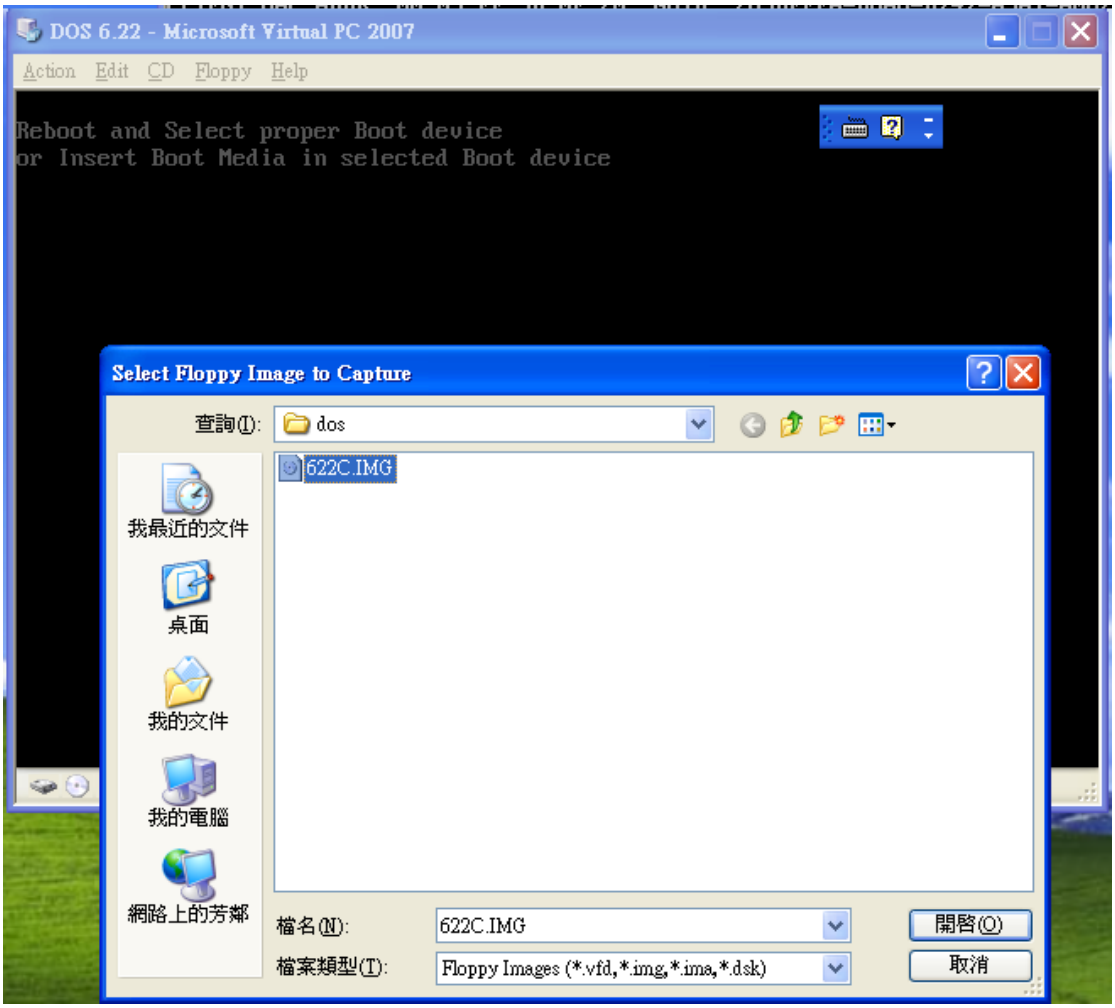


圖 9.14 在 Virtual PC 中指定 DOS 的軟碟開機映像檔

按下 Action/Reset 之後，您將會看到系統進入如圖 9.15 的 DOS 開機畫面，對於曾經使用過 DOS 的讀者而言，應該會覺得有似曾相識的感覺。沒錯，這就是那個 1980 年代的 DOS 作業系統了。

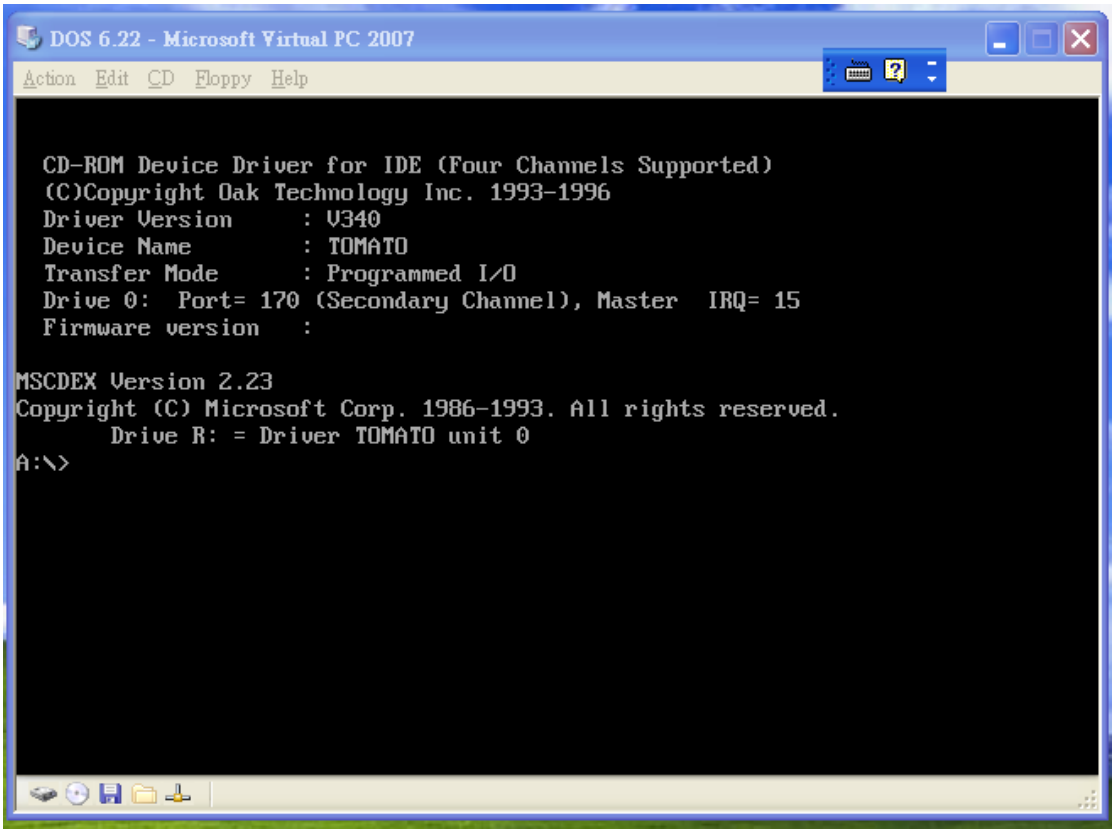


圖 9.15 在 Virtual PC 2007 中的 DOS 虛擬機開機畫面

當然， Virtual PC 虛擬機還有更多的功能，像是共用硬碟資料夾等，在此受限於本書篇幅，我們將不詳細介紹，有興趣的讀者請進一步查閱相關的書籍與文章。

習題

9.1 請說明何謂虛擬機器？

9.2 請說明何謂記憶體機？

9.3 請說明何謂堆疊機？

9.4 請說明何謂暫存器機？

9.5 請閱讀本書的第 12 章，並取得 ch12/CPU0.c 這個程式，看看這個虛擬機是如何設計的。

9.6 請寫出一個 Java 程式，並且使用 javac 編譯該程式，然後使用 java 指令執行該程式。

9.7 接續上一題，請使用 javap 將上一題產生的 bytecode 反組譯，並分析反組譯後的程式碼？

9.8 請安裝 Virtual PC，然後在其中安裝 DOS 作業系統。