Operating Systems: Design and Implementation

Second Edition

作業系統設計與實現

第二版

安德魯.坦尼鮑姆（Andrew S. Tanenbaum）

阿爾伯特.伍德豪爾（Albert S. Woodhull）

# 作者簡介

安德魯.坦尼鮑姆分別在麻省理工學院和加州大學伯克利分校獲得學士和博士學位。他現任位於荷蘭阿姆斯特丹市的Vrije大學電腦科學教授並領導著一個電腦系統研究小組。同時他還任一個研究並行、分佈及圖像系統的校際研究生院 － 電腦與圖像高級學院的院長。

坦尼鮑姆先前的研究領域包括編譯器、作業系統、網路和局域分散式系統，他現在的研究主要集中在可擴展到數百萬使用者的廣域分散式系統。對這些課題的研究使他在學報和會議上發表了70餘篇論文，並出版了五部專著。

坦尼鮑姆教授同時還主持開發了大量的軟體。他是Amsterdam編譯工具箱的總設計師，該工具箱被廣泛地用來開發可移植的編譯器，同時還用於MINIX的開發。他和他的博士研究生及程式師們一起設計了一個基於微內核的高性能分散式作業系統 － Amoeba。現在，以教學和研究為目的的使用者可以從Internet上免費獲得MINIX 和Amoeba軟體。

坦尼鮑姆的許多博士研究生在獲得學位後都取得了非常豐碩的成果，這令坦尼鮑姆非常自豪，因為這是他誨人不倦的結果。

坦尼鮑姆教授同時還是ACM的會士、IEEE高級會員、荷蘭皇家藝術和科學院院士，他曾獲得1994年ACM Karl V. Karlstrom 傑出教育獎和1997年ACM/SIGCSE 電腦科學教育傑出貢獻獎。他被列入Internet上的 Who's Who in the World 名單，他在WWW上的主頁地址為：http://www.cs.vu.nl/~ast/。

阿爾伯特.伍德豪爾分別在麻省理工學院和華盛頓大學獲得學士和博士學位。他進麻省理工學院本來是想成為一名電氣工程師，可是後來卻成了生物學家。從1973年起他開始在位於麻省Amherst的Hampshire自然科學學院工作。當微型電腦慢慢多起來的時候，作為使用電子檢測儀器的生物學家，他開始使用微型電腦。他給學生開設的檢測儀器方面的課程逐漸演變為電腦介面和即時程式設計。

伍德豪爾博士對教學和科學技術的發展有濃厚的興趣，在進入研究生院之前他曾在奈及利亞教過兩年中學，近年來他曾幾次利用自己的假期在尼加拉瓜教授電腦科學。

他對電腦作為電子系統，以及電腦與其他電子系統的相互配合很感興趣。他最喜歡講授的課程有電腦體系結構、組合語言程式設計、作業系統和電腦通信。他還為開發電子器件及相關軟體擔當顧問。

在學術之外，伍德豪爾有不少興趣，包括各種戶外運動，業餘無線電製作和讀書。他還喜歡旅遊和學習別國語言。他的WWW主頁就存在一台運行MINIX的機器上，位址是：<http://minix1.hampshire.edu/asw/>。

# 前 言

多數作業系統教材都重理論而輕實踐，本書希望在這二者之間求取較好的平衡。本書詳細論述了作業系統的所有基本概念，包括進程、進程間通信、信號量、管程、消息傳遞、調度演算法、輸入/輸出、鎖死、設備驅動程式、記憶體管理、頁面調度演算法、檔案系統設計、安全與保護機制等。同時，本書也詳細討論了MINIX － 一個與UNIX相容的作業系統，並提供了完整的原始程式碼供學習之用。這樣的安排使讀者不僅學習到理論，而且能夠理解它們如何應用在一個實際的作業系統之中。

本書第一版在1987年出版時，曾引發了作業系統課程教學的一場小小的變革。在此之前多數課程都只講理論。隨著MINIX的出現，許多學校開始增加實驗環節以使學生瞭解實際的作業系統是如何運作的。我們認為這種趨勢是可取的，並希望通過本書第二版能進一步加強這種趨勢。

MINIX在其出現以來的十年間發生了許多變化，最初的代碼是為基於8088晶片、256K記憶體和兩個軟盤機的IBM PC機型編寫的，它基於UNIX 版本7。隨著時間的推移，MINIX在許多方面有所發展，比如當前版本可運行在眾多機型上，從16位真實模式的PC機到配有大容量硬碟的奔騰機（32位元保護模式），而且它不再基於UNIX版本7，而是基於國際上的POSIX標準（POSIX 1003.1和ISO9945－1）。與此同時，有許多新特徵被添加到MINIX中，在我們看來，所增加的特徵可能已經太多了，但有些人則認為還不夠，這最終導致了LINUX的誕生。MINIX還被移植到許多其他平臺上，包括Macintosh、Amiga、Atari和SPARC。本書只涉及MINIX2.0，到目前為止，該版本只能運行於基於80X86的機器，或者可模擬此類CPU的機器，以及SPARC機器。

與第一版相比，第二版有許多變化，原理性部分基本都被修改過，同時增加了大量新內容。最主要的變化是新的基於POSIX的MINIX，以及對其原始程式碼的剖析。另外，每本書都附帶一張CD-ROM，它包含了全部MINIX原始程式碼，以及在PC上安裝MINIX的說明（見CD-ROM主目錄下的README.TXT文件）。

在一台80X86 的PC機上安裝MINIX很方便。它需要一個至少30MB的硬碟分區，然後按照CD－ROM上README.TXT檔中的步驟進行即可。在列印README.TXT檔之前，先啟動MS-DOS（若運行WINDOWS，則按兩下MS-DOS圖示），然後鍵入

copy readme.txt prn

即可。該檔也可以用edit、wordpad、notepad等任何可以處理ASCII正文的編輯器進行流覽。

對於沒有PC機的學校和個人，有兩種解決辦法，即CD－ROM上提供的兩個類比程式。一個由Paul Ashton 為SPARC機器編寫，它作為使用者程式在Solaris上運行，此時MINIX被編譯成SPARC上的可執行檔。在這種模式下，MINIX不再是一個作業系統，而只是一個使用者程式，所以必須對其底層作一些修改。

另一個類比程式由Bochs軟體公司的Kevin.P.Lawton編寫，它解釋Intel 80386的指令集以及足以使MINIX運行所需的I/O指令。顯然在解譯器層次上運行使性能有所下降，但這使得學生更容易進行調試。該類比程式運行在所有支援M.I.T的X-Window的系統上，更詳細的資訊請參看CD－ROM上的有關文件。

MINIX仍在繼續發展，本書和CD－ROM中的內容僅僅反映了本書出版時的情況，有關MINIX的最新動態請訪問MINIX的主頁：http://www.cs.vu.nl/~ast/minix.html。MINIX也有USENET中的新聞群組：comp.os.minix，讀者可以訂閱該新聞群組。對於僅有Email的讀者可通過以下步驟來加入MINIX的郵件用戶通信組。給 listserv@listserv.nodak.edu 發一封信，其中只需一行字：“subscribe minix-1 <您的完整用戶名>”，此後你便會通過E-mail獲得很多的資訊。

講授本課程的教師可以從Prentice Hall公司獲得一份習題解答手冊。從WWW地址 http://www.cs.vu.nl/~ast/沿著“Software and supplementary material"連結可以獲得一些有用的PostScript檔，其中包含本書中所有的圖表，可供需要時使用。

在MINIX的開發專案中我們有幸得到了許多人的幫助。首先要感謝Kees Bot在MINIX標準化和軟體發佈中所作的大量工作，沒有他的幫助，我們不可能完成這件工作。他自己編寫了大量的代碼（如POSIX終端I/O）並修正了一些數年來一直存在的錯誤，他還整理了其他的代碼。

這些年來Bruce Evans、 Phlip Homburg、Will Rose和Michael Temari為MINIX的開發做了大量的工作。 有幾百人通過新聞群組對MINIX作出了貢獻，他們人數眾多，所作出的貢獻也各不相同，在此謹向他們一併表示感謝。

John Casey、 Dale Grit、 Frans Kashoek等人閱讀了本書的部分手稿並提出了寶貴建議，在此向他們表示謝意。

Vrije大學的許多學生測試了CD－ROM中MINIX的 β版本，他們是：Ahmed Batou, Goran Dokic, Peter Gijzel, Thomer Gil, Dennis Grimbergen, Roderick Groesbeek, Wouter haring, Guido Kollerie, Mark Lassche, Raymond Ris, Frans ter Borg, Alex van Ballegooy, Ries van der Velden, Alexander Wels以及Thomas Zeeman。我們對他們細緻的工作和詳盡的報告致以衷心的感謝。

阿爾伯特·S·伍德豪爾向他從前的幾位學生表示感謝，特別是Hampshire學院的Peter W. Young ，Nacional Autonoma de Nicaragua大學的Maria Isabel Sanchez 和William Puddy Vargas。

最後要向我們的家庭成員表示感謝。Suzanne 已是第十次在我埋頭寫作時給我支持，對Barbara是第九次，Marvin是第八次，甚至小Bram也是第四次了。他們的支持和愛心對我非常重要。（坦尼鮑姆）

至於阿爾伯特的Barbara，這倒是第一次，假如沒有她的支援，耐心和幽默，我們是不可能完成這一工作的，對我的兒子Gordon而言，由於在編寫本書時，他大部分時間都不在家中，而是在大學學習，因此是非常幸運的。但是他的理解和關心深深吸引著我從事本書的編寫工作，有這樣一個兒子是令人非常愉快的。（伍德豪爾）

安德魯.S.坦尼鮑姆

阿爾伯特.S.伍德豪爾

譯 序

坦尼鮑姆教授是國際知名的電腦科學家和教育家。他在作業系統、分散式系統以及電腦網路領域都有很深的造詣。自八十年代以來，他已先後出版了一系列面向大學生和研究生的教材性質的專著，並被世界各國的許多大學廣泛採用。這本書就是他的最新專著之一。

作業系統是電腦系統中最核心和最底層的軟體，對作業系統的深入學習關係到對整個系統運作機制的全面理解，因此一本好教材也顯得愈發重要。本書的英文版出版於1997年，其中涵蓋了作業系統課程的所有內容，即傳統上的進程管理、記憶體管理、檔管理和設備管理。同時其中又包含了許多新內容，如執行緒、基於消息傳遞的系統構造模型、日誌結構檔案系統、安全和保護機制、RAM盤及CD－ROM設備等，而用作例子的CPU則為Interl Pentium。這使得讀者一方面能夠學習作業系統的經典內容，另一方面又能夠瞭解和跟蹤當前的最新技術和研究成果。

本書的另一個特點是基本原理與具體實例，即MINIX緊密結合。第2到第5章的前半部分講述原理，後半部分則詳細地解釋這些原理在MINIX的設計和實現中的應用。通過閱讀這些部分能夠把握MINIX原始程式碼的組織方式，並理解那些很關鍵或者很難懂的代碼。這部分內容非常翔實，有時甚至逐行地解釋附錄中所列的來源程式。對作業系統課程多年的授課經驗以及相關的科研工作使我們認識到：詳細地剖析一個象MINIX這樣的作業系統對於掌握作業系統設計與實現的精髓是大有裨益的。

正因為上述原因，我們真切地感受到將這本書翻譯、介紹給國內讀者將是一件非常有意義的事，衷心希望我們付出的勞動能對國內的作業系統教學和實踐有所説明和促進。

本書的第一章，第二章，第三章由王鵬翻譯，劉福岩和陸寧也參加了部分工作；第四章由朱鵬翻譯；第五章由敖青雲翻譯。全書由尤晉元教授審校並統稿。

在整個翻譯過程中，上海交通大學電腦系系統軟體研究室的師生給予了許多幫助。並且在電腦系95級本科生的作業系統課程中進行了試用，許多學生提出了很好的建議，在此向他們表示衷心的感謝。

特別要感謝本書的責任編輯鄧又強先生，本書的順利出版與他的辛勤勞動和熱情支持是分不開的。

雖然在翻譯過程中我們盡力恪守“信，達，雅”的準則，但不當和疏漏之處在所難免，敬請讀者提出寶貴建議。

譯 者

1998年4月于上海交通大學

# 第一章 引 言

電腦如果離開了軟體將成為一堆廢銅爛鐵。有了軟體，電腦可以對資訊進行存儲、處理和檢索，可以顯示多媒體文檔、搜索Internet並完成其他工作。電腦軟體大致分為兩類：系統軟體和應用軟體。系統軟體管理電腦本身及應用程式；應用軟體執行使用者最終所需要的功能。最基本的系統軟體是作業系統（operating system），它控制電腦的所有資源並提供開發應用程式的基礎。

現代電腦系統包含一個或多個處理器、若干記憶體（常稱為RAM－隨機存取記憶體）、磁片、印表機、網路介面及其他輸入/輸出設備。編寫一個程式來管理所有這些器件以正確地使用它們，即使不考慮優化也是一件很困難的事情。如果每個程式師都必須處理磁片如何工作，再加上每讀一個磁片塊都有幾十種因素可能導致操作出錯，那麼很多程式簡直沒法寫。

許多年以前人們就認識到必須找到某種方法將硬體的複雜性同程式師分離開來。經過不斷探索和改進，目前採用的方法是在裸機上載入一層軟體來管理整個系統，同時給使用者提供一個更容易理解和進行程式設計的介面，這被稱為虛擬機器（virtual machine）。這樣一層軟體就是作業系統。

這種處理方式如圖1－1所示。底層是硬體，它本身可能包括兩層或多層。最低一層是物理器件，包括積體電路晶片、連線、電源、監視器等，它們的構造和工作方式屬於電氣工程師的範圍。

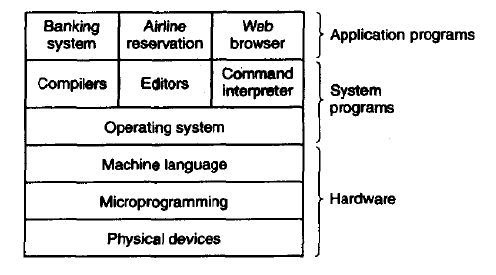


圖 1-1 電腦系統由硬體、系統程式和應用程式組成。

接著是微程式（microprogram），通常存放在唯讀記憶體中，它是一層很原始的軟體，用來控制設備並向上一層提供一個更清晰的介面。微程式實際上是一個解譯器，它先取得機器語言指令，如ADD， MOVE和JUMP等，然後通過一個動作序列來執行這些指令。例如為了執行一條ADD指令，微程式必須先確定運算資料的位置，然後取數，相加，最後存放得數。由微程式解釋執行的這一套指令稱為機器語言。機器語言並不是硬體的組成部分，但硬體製造商通常在手冊中給出機器語言的完整描述，所以許多人將它認作真正的“電腦”。

採用精簡指令集電腦（RISC）技術的電腦沒有微程式層，其機器指令通過硬體邏輯直接執行。例如Motorola 680X0有微程式，而IBM PowerPC 則沒有。

機器語言典型地有50到100條指令，大多數用來完成資料傳送、算數運算和數值比較等操作。在這個層次上，通過向特殊的設備寄存器寫特定的數值來控制輸入/輸出設備。例如將磁片位址、記憶體位址、讀位元組數和操作類型（讀/寫）等值寫入特定的寄存器便可完成硬碟讀操作。實際操作往往需要更多的參數，而操作完成後的返回狀態也非常複雜。進一步而言，對於許多I/O設備，時序在程式設計中的作用非常重要。

作業系統的主要功能之一就是將所有這些複雜性隱藏起來，同時為程式師提供一套更加方便的指令，比如，“從檔中讀一個資料塊”在概念上比低層的“移動磁頭臂，等待旋轉延遲”之類的細節來得簡單、方便。

在作業系統之上是其他系統軟體，包括命令直譯器（shell）、視窗系統、編譯器、編輯器及類似的獨立于應用的程式。要注意它們本身並不是作業系統的組成部分，儘管它們通常由電腦廠商提供。這一點很重要，作業系統專指在核心態（kernel mode），或稱管態（supervisor mode）下運行的軟體，它受硬體保護而免遭使用者的篡改。編譯器和編輯器運行在用戶態（user mode）。如果用戶不喜歡某一個編譯器，他可以自己重寫一個，但他卻不可以寫一個磁片中斷處理常式 － 因為這是作業系統的一部分，而且硬體阻止使用者對它進行修改。

系統軟體之上是應用軟體，這些軟體可以是購買的或者是用戶自行開發的，它們用來解決特定的問題，如文書處理、表格處理、工程計算或者電子遊戲等。

## 1.1 什麼是作業系統

多數電腦使用者都使用過作業系統，但要精確地給出作業系統的定義卻很困難，部分原因是作業系統完成兩項相對獨立的任務，下面我們逐項進行討論。

### 1.1.1 作業系統作為虛擬機器

對多數電腦而言，在機器語言一級的體系結構（指令集、存儲組織、I/O和匯流排結構）上程式設計是很困難的，尤其是輸入輸出操作。例如考慮使用多數PC機採用的NEC PD765控制器晶片（或功能等價的晶片）來進行軟碟I/O操作。PD765有16條命令，它通過向一個設備寄存器裝入特定的資料來執行這些命令，命令資料長度從1到9位元組不等，其中包括：讀寫資料、移動磁頭臂、格式化磁軌、初始化、檢測磁片狀態、重定、校準控制器及設備等。

最基本的命令是讀數據和寫資料。它們均需要13個參數，所有這13個參數被封裝在9個位元組中。這些參數指定的資訊有：欲讀取的磁片塊位址、每條磁軌的磁區數、物理介質的資料記錄格式、磁區間隙、以及對已刪除資料位址標識的處理方法等。當磁片操作結束時，控制器晶片返回23個狀態及出錯資訊，它們被封裝在7個位元組中。此外，程式師還要注意步進電機的開關狀態。如果電機關閉，則在讀寫資料前要先啟動它（有一段較長的加速時間）。還要注意電機不能長時間處於開啟狀態，否則將損壞軟碟，所以程式師必須在較長的啟動延遲和可能對軟碟造成損壞之間作出折衷。

顯然，程式師不想涉及硬體的這些具體細節（也包括硬碟，它與軟碟不同，但同樣很複雜）。他需要的是一種簡單的高度抽象的設備。一種典型的抽象是一張磁片包含了一組命名的檔，每個檔可以被打開，然後進行讀寫，最後被關閉。其中的一些細節如數據記錄格式、當前步進電機的開啟狀態等則對使用者隱藏。

這種將硬體細節與程式師隔離開來、同時提供一個簡潔的命名檔方式的程式，就是作業系統。與磁片抽象類別似，它還隱藏了其他許多低層硬體的特性，包括中斷、時鐘、記憶體等。總之，作業系統提供的每一種抽象都較低層硬體本身更簡單、更易用。

從這個角度看，作業系統的作用是為使用者提供一台等價的擴展電腦，或稱虛擬機器，它比低層硬體更容易程式設計。本書的內容正是詳細說明作業系統如何做到這一點。

### 1.1.2 作業系統作為資源管理器

上述虛擬機器模型是一種自頂向下的觀點。按照自底向上的觀點，作業系統則用來管理一個複雜系統的各個部分。現代電腦都包含處理器、記憶體、時鐘、磁片、滑鼠、網路介面、雷射印表機以及其他許多設備，從這個角度看，作業系統的任務是在相互競爭的程式間有序地控制這些設備的分配。

設想在一台電腦上運行的三個程式同時試圖在一台印表機上輸出計算結果，那麼可能頭幾行是程式1的輸出，接下來幾行是程式2的輸出，然後又是程式3的輸出等等，最終列印結果將是一團糟。作業系統採用將列印輸出緩衝到磁片上的方法可以避免這種混亂。當一個程式結束後，作業系統將暫存在磁片檔上的輸出結果送到印表機，同時其他程式可以繼續運行產生新的輸出結果，而這些程式並不知道這些輸出沒有立即送至印表機。

當一台電腦（或網路）有多個使用者時，因為用戶間可能相互影響，所以管理和保護記憶體、I/O設備以及其他設備的需求隨之增加。而且使用者往往不僅需要共用硬體，還要共用資訊（檔、資料庫等）。總之，此時作業系統的首要任務是跟蹤資源的使用狀況、滿足資源請求、提高資源利用率、以及協調各程式和使用者對資源的使用衝突。

## 1.2 作業系統發展歷史

作業系統經歷了一個漫長的發展過程，下面對此進行簡要的回顧。由於在歷史上作業系統與電腦體系結構存在非常密切的聯繫，我們將按照電腦的換代歷程講述作業系統的發展狀況。

第一台真正的數位電腦是英國數學家Charles Babbage（1792－1871）設計的。Babbage投入畢生精力去建造他的“分析機”，但卻沒能讓它成功地運行起來。因為它是純機械式的，而當時的技術不可能使分析機的零部件達到他所需要的精度。很顯然，分析機沒有作業系統。

有趣的是，Babbage認識到他的分析機需要軟體，於是他雇傭了一個年輕的女子，英國著名詩人拜倫的女兒Ada Lovelace為他工作。Ada由此成了世界上第一位程式師，Ada程式設計語言就是用她的名字命名。

### 1.2.1 第一代電腦（1945－1955）：真空管和插板

從Babbage之後一直到二戰，數位電腦幾乎沒有什麼進展。在40年代中期，哈佛大學的Howard Aiken、普林斯頓高等研究院的John.Von Neumann（馮.諾依曼）、賓夕法尼亞大學的J.Presper Eckert和William Mauchley、德國電話公司的Konrad Zuse、以及其他一些人都成功地使用真空管建造了電腦。這些機器非常龐大，往往使用數萬個真空管，佔據幾個房間，然而其運算速度卻不如現在最便宜的個人電腦。

在電腦出現的早期，每台機器都有一個小組專門來設計、製造、程式設計、操作和維護。程式設計全部採用機器語言，通過在一些插板上的硬連線來控制其基本功能，這時沒有程式設計語言（甚至沒有組合語言），作業系統更是聞所未聞。機器的使用方式是程式師提前在牆上的機時表上預約一段時間，然後到機房將他的插板插到電腦裡，在接下來的幾小時裡計算自己的題目。這時的電腦很不可靠，因為幾萬個真空管中任何一個發生故障，電腦就無法運行。這個階段基本上所有的題目都是數值計算問題，例如計算正弦和余弦函數表。

到50年代早期，出現了穿孔卡片，這時就可以不用插板，而是將程式寫在卡片上然後讀入電腦，但其他過程則依然如故。

### 1.2.2 第二代電腦（1955－1965）：電晶體和批次處理系統

50年代發明的電晶體極大地改變了電腦的狀況。這時的電腦已經很可靠，廠商可以成批地生產電腦並賣給客戶，客戶可以長時間地使用它來完成一些有用的工作。至此，第一次將設計人員、生產人員、操作員、程式師和維護人員分開。

這個時期電腦安裝在空調房間裡，有專人操作。由於其價格昂貴，僅有少數大公司、主要的政府部門和大學才買得起。運行一個作業（一個或一組程式）時，程式師首先將程式寫在紙上（用FORTRAN或組合語言），然後用穿孔機制成卡片，最後將這些卡片交給操作員。

電腦運行完當前任務後，其計算結果從印表機上輸出，操作員從印表機上取得運算結果並送到輸出室，程式師就可從該處取到運算結果，然後，操作員再從卡片上讀入另一個任務。如果需要FROTRAN編譯器，操作員還要從別處取來讀入電腦。當操作員在機房裡走來走去時，許多機時被浪費掉了。

由於當時電腦非常昂貴，很自然地人們開始想辦法減少機時的浪費，答案就是批次處理系統。其思想是：在作業輸入室收集到較多的作業，然後用一台相對廉價的電腦（如IBM 1401電腦，它適用於讀卡片、拷貝磁帶和列印輸出，但不適用於作數值運算。）將它們讀到磁帶上，另外用較昂貴的電腦如IBM 7094來完成真正的計算。該模型如圖1－2所示。

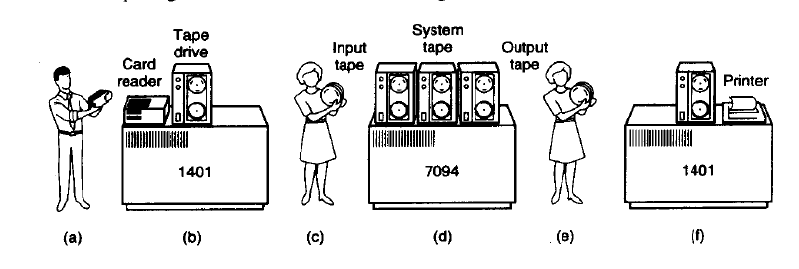


圖 1-2 一種早期的批次處理系統。 (a)程式師將卡片拿到1401機處 (b)1401將批次處理作業讀到磁帶上 (c)操作員將輸入帶送至7094處 (d )7094進行計算 (e)操作員將輸出帶送至1401處 (f)1401列印輸出

在收集到一批次工作之後，輸入磁帶被送到機房裡裝到磁帶機上。操作員隨後裝入一個特殊的程式（現代作業系統的前身），它從磁帶上將第一個作業讀入並運行，其輸出寫到第一盤輸出磁帶上，而不是列印出來。每個作業結束後，作業系統自動地讀入下一個作業運行。當這一批次工作完全結束後，操作員取下輸入和輸出磁帶，將輸入磁帶換成下一批次工作，然後把輸出磁帶拿到一台1401機器上進行離線列印。

一個典型的輸入作業結構如圖1－3所示。它由一張＄JOB卡片開始，該卡標識出所需的最大執行時間（分鐘）、計費標識、以及程式師的名字。隨後是一張＄FORTRAN卡片，它通知作業系統從系統磁帶上裝入FROTRAN語言編譯器。在此之後是待編譯的來源程式，然後是＄LOAD卡片，它通知作業系統裝入剛編譯好的目的程式（編譯好的目的程式通常寫到暫存磁帶上，需要顯式裝入）。接著是＄RUN卡片，它告訴作業系統運行該程式並使用其後的資料。最後，＄END卡片標識作業結束。這些原始的控制卡片是現代作業控制語言和命令直譯器的先驅。

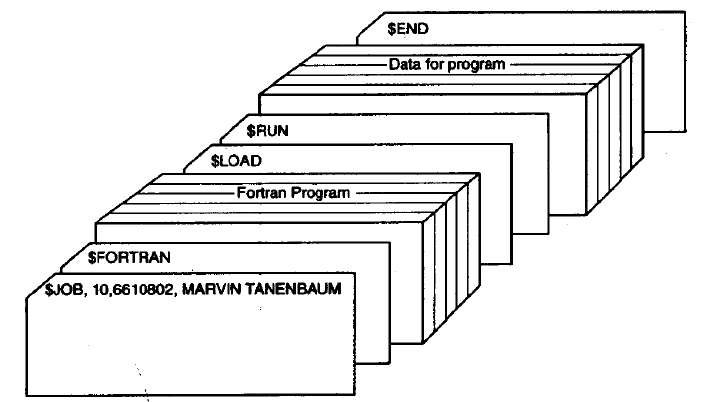


圖 1-3 一個典型的FMS作業的結構。

第二代電腦主要用於科學計算，例如解偏微分方程。這些題目大多用FORTRAN語言和組合語言程式設計。典型的作業系統是FMS（FORTRAN Monitor System－FORTRAN監控系統）和IBSYS（IBM為7094機配備的作業系統）。

### 1.2.3 第三代電腦（1965－1980）：積體電路晶片和多道程序

在60年代初期，多數電腦廠商都有兩條完全不同並且互不相容的生產線：一條是面向字的複雜科學計算和工程計算的電腦，如IBM 7094；另一條是面向字元的商用電腦，如IBM 1401，主要被銀行和保險公司用於磁帶歸檔和列印服務。

對廠商來說，開發和維護兩種完全不同的產品是很昂貴的。同時，許多新的電腦使用者開始時只需要一台小電腦，到後來則可能需要一台較大的電腦，而且要求能夠更快地執行原有的程式。

IBM公司試圖通過引入360系統來解決這兩個問題。IBM 360是一個軟體相容的電腦系列，在該系列中，低檔機與1401相當，高檔機則比7094功能強很多。這些電腦只在價格和性能（最大記憶體容量、處理器速度、允許的I/O設備數量等）上有差異。由於所有的電腦都有相同的體系結構和指令集，因此為一種型號機器編寫的程式可以在其他所有型號的機器上運行（起碼在理論上可行），而且360被設計成既可用於科學計算，又可用於商業計算，這樣一個系列的電腦便可以滿足所有使用者的要求。在隨後的幾年裡，IBM陸續推出了360的後續機型，如用戶熟知的370、4300、3080和3090系列。

360是第一種採用積體電路（小規模）晶片的主流機型。與採用分立電晶體製造的第二代電腦相比，其性能價格比有很大提高。360很快就獲得了成功，由此其他主要廠商也很快採納了系列兼容機的思想。這些電腦至今仍在各地的計算中心使用，但其應用正在急劇地萎縮。

“單一家族”思想的最大優點同時也是其最大的缺點。原因是所有軟體，包括作業系統，都要能夠在所有機器上運行 － 從小的代替1401的機器到用於科學計算的相當於7094的大型機；從只能帶很少外部設備的機器到能帶很多外設的機器；從商業領域到科學計算領域等，總之，要有效地適用於所有的用途。

IBM無法寫出同時滿足這些需求相互衝突的軟體，其結果是一個龐大的極其複雜的作業系統，它的規模比FMS高大約二到三個數量級。其中包含有數千名程式師寫的數百萬行組合語言代碼。同時，其中也有成千上萬處錯誤。這就導致IBM不斷地發行新版本來更正這些錯誤，而新版本在改正老錯誤的同時又引入新錯誤，所以錯誤數可能保持大致相同，而不是減少。

OS/360的設計者之一Fred Brooks後來寫過一本書（Brooks,1975）來描述他在開發OS/360過程中的經驗。這裡不可能複述該書的全部內容，不過其封面是一群史前動物陷入一個泥坑不能自拔，Silberschatz和Galvin的著作的封面也表達了同樣的觀點。

拋開OS/360的龐大和存在的問題，它和其他公司的類似的第三代作業系統的確很好地滿足了大多數用戶的要求。同時它們也使第二代作業系統缺乏的幾項關鍵技術得到廣泛應用。其中最重要的是多道程序（multiprogramming）。在7094機上，若當前作業因等待磁帶或其他I/O而暫停時，CPU就只能簡單地踏步直至該I/O結束。對於CPU操作密集的科學計算問題，I/O操作較少，因此浪費的時間很少；然而對於商業資料處理，I/O操作等待時間通常占到80－90％，這時必須採取某種措施減少CPU時間的浪費。

經過探索找到的解決辦法是將記憶體分為幾個部分，每一部分存放不同的作業，如圖1－4所示。當一個作業等待I/O操作完成時，另一個作業可以使用CPU。如果記憶體中可以存放足夠多的作業，則CPU利用率可以接近100％。在主存中同時駐留多個作業需要特殊的硬體來對其進行保護，以避免作業的資訊被竊取或受到攻擊，幸運的是360及其他第三代電腦都配有此類硬體。

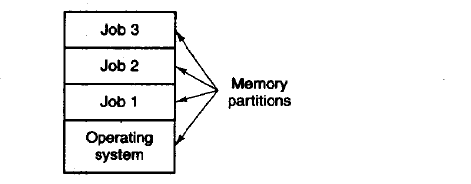


圖 1-4 記憶體中有三個作業的一個多道程序系統。

第三代電腦的另一個新特性是：卡片被拿到機房後能夠很快地將一個作業從卡片讀入磁片。於是無論任何時刻當一個作業運行結束，作業系統就能將一個新作業從磁片讀出，裝入空出來的記憶體區域運行，這種技術叫做spooling（Simultaneous Peripheral Operation On Line － 連線的即時外部設備操作），同時該技術也用於輸出。當採用了spooling技術後，就不再需要IBM 1401機，也不必再將磁帶搬來搬去。

儘管第三代作業系統很適於大型科學計算和繁忙的商務資料處理，但其實質上仍舊是批次處理系統。許多程式師很懷念第一代電腦的使用方式，那時他們可以獨佔一台機器幾個小時，可以即時地調試他們的程式。而對第二代電腦，一個作業從提交到取回運算結果往往長達數小時。更有甚者，一個誤用的逗號就會導致編譯失敗，從而可能浪費程式師半天時間。

程式師們希望很快得到回應，這種需求就導致了分時系統的出現。它實際上是多道程序的一個變種，不同之處只是每個使用者都有一個連線終端。在分時系統中，假設有20個用戶登錄，而其中17個在思考或喝咖啡，則CPU可輪流分配給那三個需要得到服務的作業。由於偵錯工具的用戶常常只發出簡短的命令（如編譯一個原始檔案），而很少執行費時的長命令（如將一個上百萬條記錄的檔排序），所以電腦能夠為一些使用者提供快速的互動式服務，同時在CPU空閒時還能運行後臺的大作業。第一個分時系統CTSS是由M.I.T在一台改裝過的7094機上開發成功的（Corbato等，1962），但直到第三代電腦廣泛採用了必需的保護硬體之後分時系統才逐漸流行開來。

在CTSS研製成功之後，M.I.T、貝爾實驗室和通用電氣公司（GE，當時一個主要的電腦製造廠商）決定開發一種“公用計算服務系統” － 一種能夠同時支援數百名分時用戶的機器。它借鑒於供電系統 － 當你需要電能時，只需將電氣設備接到牆上的插座即可。該系統稱作MULTICS（MULTiplexed Information and Computing Service），其設計者著眼于建造一台機器來滿足整個波士頓所有用戶的計算需求。在當時看來，僅30年之後只花幾千美元就能買一台計算能力遠遠超過他們的GE－645的個人電腦的想法完全是科學幻想。

MULTICS引入了電腦領域許多概念的雛形，但其研製難度卻超出了所有人的預料。在開發過程中貝爾實驗室退出了該專案，通用電氣公司也退出了電腦領域。最終MULTICS被成功地應用在M.I.T的實際生產環境以及其他幾十個系統中。但“公用計算服務系統”的概念卻隨著電腦價格的暴跌而被人們遺棄，不過MULTICS對隨後的系統卻有著巨大的影響，詳細請參閱（Corbato etc., 1972；Corbato and Vyssotsky, 1965；Daley and Dennis, 1968；Organick, 1972；Saltzer, 1974）。

第三代電腦的另一個主要進展是小型機的崛起，這以1961年DEC的PDP－1作為起點。PDP－1電腦只有4K個18 比特的記憶體，每台售價120,000美元（不到IBM 7094的5％），該機型非常熱銷，對於某些非數值的計算，它幾乎和7094一樣快。PDP－1開闢了一個全新的產業。很快PDP有了一系列機型（與IBM系列機不同，它們互不相容），其頂峰為PDP－11。

貝爾實驗室一位曾參加過MULTICS研製的電腦科學家Ken Thompson，在一台無人使用的PDP－7機器上開發了一個簡化的單用戶版MULTICS，他的工作導致了後來UNIX作業系統的誕生。UNIX在學術界變得很流行，同時也包括政府部門和許多公司。

有專門的著作講述UNIX的歷史（例如Salus, 1994）。簡單地說，由於UNIX的原始程式碼公開，所以許多組織都開發了他們各自的UNIX版本，這些版本互不相容，所以非常混亂。為了使同一個程式在所有不同的UNIX系統上都能運行，IEEE擬定了一個UNIX的標準，稱作POSIX，該標準現在被大多數UNIX支援。POSIX定義了相互相容的UNIX系統必須支援的一個最小的系統調用介面，實際上，一些其他作業系統現在也支援POSIX介面。

### 1.2.4 第四代電腦（1980－現在）：個人電腦

隨著大型積體電路的發展，晶片在每平方釐米的矽片上可以集成數千個電晶體，於是個人電腦時代到來了。從體系結構上看，個人電腦與PDP－11並無二致，但就價格而言卻相去甚遠。通常公司的一個部門或大學裡的一個院系配備一台小型機，而個人電腦卻使每個人都能擁有自己的電腦。在商業、大學或政府部門使用的功能最強的個人電腦通常稱為工作站，它實際上只是大一點的個人電腦，通常工作站之間通過網路互連起來。

隨著計算能力越來越容易獲得，尤其是具有高品質圖形功能的互動式計算的普及，為個人電腦編制軟體成為一項重要的產業。此類軟體多數對使用者很友好，也就是說用戶無需掌握太多的電腦知識，而且基本不用怎麼學習便能夠使用這些軟體，這與先前的OS/360 完全不同，OS/360的作業控制語言JCL非常複雜，為了介紹這種語言，已經專門編寫了幾本書。

在個人電腦和工作站領域有兩種主流作業系統：微軟的MS－DOS和UNIX。MS－DOS廣泛用於IBM PC及其他採用Intel 80X86晶片的電腦。儘管MS－DOS的最初版本相當簡陋，但其後續的版本包含了許多新特性，其中有許多源自於UNIX。MS－DOS的後續產品－Windows起初運行於MS－DOS之上（與其稱之為作業系統，不如說它更象一個shell），但從1995年開始發佈的Windows95是一個真正可引導的作業系統，因此它不再需要MS－DOS的支援。微軟的另一個作業系統是Windows NT，它在某些層次上與Windows95相容，但其核心則完全重寫。

另一種主要的作業系統是UNIX，它在工作站和高檔電腦領域（如網路服務器）佔據了統治地位，尤其對於採用高性能RISC晶片的電腦。儘管這些電腦通常供一個使用者專用，但它們往往具有小型機的計算能力，所以為它們配備最初為小型機設計的作業系統 － UNIX是很順理成章的。

從80年代中期開始出現一種有趣的發展趨勢，就是運行網路作業系統（network operating systems）和分散式作業系統（distributed operating systems）（Tenenbaum, 1995）的個人電腦網路的崛起。在網路作業系統中，使用者知道多台電腦的存在。他能夠登錄到一台遠地機器上並將檔從一台機器拷貝到另一台機器，每台電腦都運行自己本地的作業系統，有自己的本地用戶（或多個用戶）。

網路作業系統與單處理機的作業系統沒有本質區別。它們需要一個網路介面控制器以及一些低層軟體來驅動它 ，同時還需要一些程式來進行遠端登入和遠地檔訪問，但這些附加物並未改變作業系統的本質結構。

與之相反，一個分散式作業系統在使用者看來就象一個普通的單處理機系統。儘管它實際上由多個處理機組成，但用戶不會感知到他們的程式在哪個處理機上運行，或者他們的檔存放在哪裡，所有這些均由作業系統自行高效地完成。

真正的分散式作業系統不僅僅是在單機作業系統上增添一小段代碼，其原因是分散式系統與集中式系統有本質的區別。例如，分散式系統通常允許一個應用在多台處理器上同時運行，因此需要更複雜的處理器調度演算法來獲得最大的並行度。

網路中的通信延遲往往導致分散式演算法必須能適應資訊不完備、資訊過時甚至資訊不正確的環境。這與單機系統完全不同，對於後者，作業系統掌握整個系統的完備資訊。

### 1.2.5 MINIX的歷史

在UNIX的早期（版本6），原始程式碼可以免費獲得並被人們加以廣泛的研究。澳大利亞新南威爾士大學的John Lions甚至專門寫了一本小冊子逐行地解釋UNIX原始程式碼（Lions, 1976）。許多大學的作業系統課程就採用這本小冊子作為教材。

在AT&T發佈版本7時，它開始認識到UNIX的商業價值，於是發佈的版本7許可證禁止在課程中研究其原始程式碼以免其商業利益受到損害。許多學校為了遵守該規定，就在課程中略去UNIX的內容而只講作業系統理論。

不幸的是，只講理論使學生對實際的作業系統產生一種片面的認識。書本上作為重點講述的內容，如進程調度演算法，實際中並沒有那麼重要；而實際系統中很重要的內容，如I/O系統和檔案系統又因為缺乏理論性而被忽略。

為了扭轉這種局面，本書的作者之一坦尼鮑姆決定編寫一個在用戶看來與UNIX完全相容，然而內核全新的作業系統 － MINIX。MINIX沒有借用AT&T一行代碼，所以不受其許可證的限制，它可以被班級和個人用來學習。通過它讀者可以剖析一個作業系統，研究其內部如何運作。其名稱源於“小UNIX”，因為它非常簡潔，一般程度的讀者就能夠讀懂它。

除合法性以外，MINIX與UNIX相比還有另一個優點：它比UNIX晚出現十年，並且其代碼採用了一種更加模組化的組織方法。例如，MINIX的檔案系統不是作業系統的一部分而是作為一個使用者程式運行。另一個不同之處在於UNIX著重效率而MINIX著重可讀性（數百頁的程式通常認為是可讀的），其代碼中有數千行注釋。

MINIX最初設計成與UNIX版本7相容。UNIX版本7由於其簡潔和優美而被奉為典範，有人甚至認為它不僅超過了其前邊的所有版本，而且超過了其後邊的所有版本。隨著POSIX的出現，MINIX在保持既有的向後相容性的同時開始向POSIX靠攏。這種漸進的演變在電腦界相當普遍，沒有一家廠商希望在引進新系統的同時使客戶所有原先的程式作廢。本書中介紹的MINIX基於POSIX標準（本書前一版本的MINIX基於UNIX版本7）。

和UNIX一樣，MINIX用C語言編寫，從而很容易被移植到其他機器上。由於IBM PC機廣泛得到使用，所以其最初版本在IBM PC上實現，隨後被移植到了Atari、Amiga、 Macintosh和SPARC等平臺上。MINIX一直恪守“small is beautiful”的原則，因此最早的版本甚至不需要硬碟就可以運行，這使許多學生都能夠達到其硬體要求（這種情況在現在看起來很奇怪，可是在80年代中期MINIX剛出現的時候，硬碟非常昂貴）。隨著其功能和規模的增長，MINIX也發展到需要一個硬碟才能運行，但它只要求一個30M位元組的分區即可。與之相比，有些商用UNIX系統建議提供多達200M位元組的磁碟分割。

對於多數IBM PC用戶來說，MINIX運行起來很象UNIX。許多基本程式，如cat、 grep、 ls、make及shell程式都與UNIX中的對應程式有相同的功能。與作業系統本身一樣，這些公用程式均由作者和他的學生以及其他人員從頭重寫。

本書自始至終以MINIX作為例子，然而MINIX的大部分內容，除了代碼本身外，均適用於UNIX。其中的許多內容還適用於一些其他的作業系統，在閱讀本書時應記住這一點。

有些讀者可能對MINIX和LINUX之間的關係感興趣。在MINIX發佈後不久，便出現了一個面向它的USENET新聞群組，在數周之內便有多達40000個用戶訂閱該新聞群組。其中的大多數人都想向MINIX中加入一些新特性以使之更大、更有用。每天都有數百人提供自己的建議、思想甚至代碼。而MINIX的作者在幾年內一直堅持不採納這些建議，目的是使MINIX保持足夠的短小精悍，以便於學生理解。人們最終意識到不可能動搖作者的立場，於是一個芬蘭學生Linus Torvalds決定編寫一個類似MINIX的系統，但是它特徵繁多、面向實用而非教學，這就是LINUX。

## 1.3 作業系統基本概念

作業系統與使用者程式的介面由作業系統提供的“擴展指令”集定義。儘管有多種不同的實現方法，這些擴展指令傳統上稱作“系統調用”。為了真正地理解作業系統的運作機制，有必要仔細研究這個介面。各個作業系統提供的系統調用各不相同（儘管基本概念大致相同）。

本書在講法上有兩種選擇：一種是泛泛而籠統地介紹系統調用（如：作業系統有讀檔的系統調用）；另一種是選擇一個確定的系統，講述該系統的系統調用（如：MINIX有一條READ系統調用，它有三個參數：一個指定所操作的檔，一個指定將讀到的資料存放在何處，最後一個指定讀多少位元組）。

本書採用後者，這樣可以更細緻地觀察作業系統的內部操作。在1.4節中將詳細地介紹UNIX和MINIX都提供的系統調用。為簡潔起見，我們只講述MINIX，多數情況下UNIX系統中對應的系統調用基於POSIX標準。下面先簡要地描述一下MINIX以獲得一些感性認識，這些內容同樣適用於UNIX系統。

MINIX系統調用大致分為兩類：與進程有關的系統調用和與檔有關的系統調用。

### 1.3.1 進程

在MINIX及所有作業系統中，一個重要概念是進程（process）。一個進程本質上是一個程式的執行。每個進程有其自己的位址空間，從0到一個最大值，進程可以讀寫該空間中的內容。位址空間中包括可執行程式、程式的資料及堆疊。與每個進程相關的還包括一組寄存器、程式計數器、指標和其他硬體寄存器，以及所有其他運行該程式所需要的資訊。

進程的概念將在第二章詳細討論，此處希望讀者對進程建立一種直觀的感覺。為此我們考慮分時系統的工作：分時作業系統週期性地掛起一個進程然後啟動運行另一個進程，例如在過去的一秒鐘內，第一個進程已運行完分配給它的時間片，則要被暫停。

在上述進程被暫時掛起的情況下，之後的某個時刻它將被再次啟動，而且要求再次啟動時的狀態與先前暫停時完全相同，這就意味著在將其掛起時該進程的所有資訊都要被保存下來。例如進程以讀方式打開了若干檔，每一個檔都有一個指標來指示當前的讀寫位置。當該進程被掛起時，這些指標均要被記錄下來以保證後面的READ系統調用能夠正確地執行。在許多作業系統中，除進程位址空間以外的所有資訊均存放在作業系統管理的一張表中，稱為進程表。進程表是一個資料結構的陣列，當前系統中的每個進程都要佔用其中一項。

所以一個（掛起的）進程包括兩部分：進程的位址空間 － 稱作核心映射（core image），以及對應的進程表項（包含寄存器值及其他資訊）。

在與進程管理有關的系統調用中最關鍵的是完成進程創建和進程終止的系統調用。考慮一個典型的例子。一個命令直譯器（shell）進程從終端上讀命令，此時，用戶剛鍵入一條命令要求編譯一個程式，為此shell必須首先創建一個進程來執行編譯器。當執行編譯的進程結束時，它執行一條系統調用來終止自己。

若一個進程能夠創建一個或多個進程（稱為子進程），而且這些子進程又可以創建子進程，則很容易得到如圖1－5所示的進程樹。為某項任務而合作的這些相關進程需要通信以同步其操作，這種通信稱為進程間通信（interprocess communication），第二章將對此進行詳細討論。

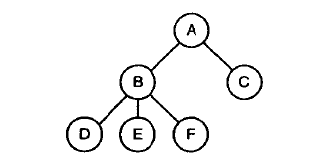


圖 1-5 一棵進程樹，進程A創建兩個子進程B和C，進程B創建子進程D、E和F。

其他系統調用的功能包括：申請更多的記憶體（或釋放不再需要的記憶體）、等待一個子進程結束、載入並執行另一個程式等。

在某些情況下，需要向一個運行進程傳送資訊，而該進程並不在資訊發送處附近等待接收資訊。例如分別在兩台機器上的兩個進程通過在網路上發送消息進行通信。為了保證一條消息或消息的應答不會丟失，發送者要求它所在的作業系統在指定的若干秒後給它一個通知，這樣如果它尚未收到確認消息就可以進行重發。在設定該時鐘後，程式可以繼續做其他工作。

經過了指定的秒數後，作業系統向這個進程發送一個信號。接收到信號將導致該進程暫時掛起，將寄存器值保存到堆疊中，並啟動一個特殊的信號處理過程，例如重傳一條丟失的消息。信號處理常式結束後，進程從收到信號的中斷點處繼續執行。信號是對硬體中斷的軟體類比，除了超時以外，還有很多其他原因可以導致產生信號。許多類型的陷入是由硬體檢測的，如執行非法指令或非法訪問位址，此類操作都將導致向違例進程發送相應的信號。

MINIX中的每一個合法使用者都有一個由系統管理員分配的使用者標識（uid），MINIX中的每一個進程都記錄有啟動它的使用者的uid。子進程的uid與其父親相同。系統中有一個特殊的用戶 － 超級用戶，他擁有特殊的權力，許多保護規則對超級使用者無效。在大型系統中，只有系統管理員知道超級使用者的口令。但很多普通用戶，尤其是學生，總是費盡心機地尋找系統的安全性破綻以使自己無需該口令便成為超級用戶。

### 1.3.2 文件

另一類系統調用面向檔案系統。正如前面提到的，作業系統的主要功能之一是遮罩硬體設備的特殊細節以便為程式師提供一個簡潔方便的與設備無關的檔模型。顯然，一個作業系統需要有相應的系統調用來創建檔、刪除檔、讀檔和寫檔。在讀一個檔之前首先要打開它，在讀完之後要關閉它，完成此類功能的系統調用都是存在的。

檔的存放通過目錄完成，一個目錄下可以存放一組檔。例如一個學生可能給他所選的各門課程創建一個目錄，另外設立一個目錄存放電子郵件，再一個目錄存放其WWW主頁。這樣就需要用到創建和刪除目錄的系統調用，以及將一個已經存在的檔放在一個目錄中，從一個目錄中刪除一個檔等。目錄項可以是檔或者其他目錄，該模型就產生了層次型檔案系統，如圖1－6所示。

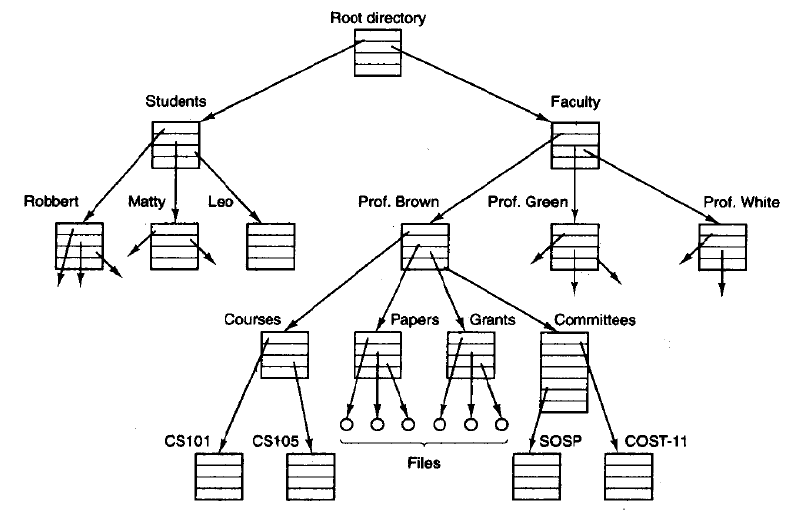


圖 1-6 大學裡一個系的檔案系統。

進程和檔都可以組織成樹狀結構，但有許多不同之處。進程樹的層次一般都不會很深（很少超過三層），而檔層次常多達四層、五層或更多。進程樹的層次結構是暫時性的，通常最多存在幾分鐘，而目錄層次則可能長達數年之久。進程和檔在屬主及保護方面也是有區別的。典型地，只有父進程可以控制和訪問子進程，而對於檔和目錄則通常存在一種機制使屬主以外的其他用戶也可以訪問該檔。

目錄層次結構中的每一個檔都可以用一個從根目錄開始的路徑名來確定，這種絕對路徑名中包含了從根目錄到該檔的所有中間目錄，相互之間由正斜杠隔開。在圖1－6中，文件CS101的路徑名是/Facaulty/Prof.Brown/Courses/CS101。起始的正斜杠表示這是一個從根目錄起始的絕對路徑。

進程在任一時刻都有一個當前工作目錄，非正斜杠起始的路徑名均在此目錄開始搜索。例如在圖1－6中，若當前工作目錄為/Facaulty/Prof.Brown，則使用路徑名Courses/CS101與/Facaulty/Prof.Brown/Courses/CS101等效。進程可以通過系統調用改變當前工作目錄。

MINIX中的檔和目錄通過一個9比特的保護碼來進行保護。保護碼分成三個3比特的域，分別對應著檔主、同組用戶和其他用戶。每個域有一位元標識讀許可權，一位元標識寫許可權，一位元標識執行許可權。如保護碼rwxr-x--x表示：檔主可以讀、寫、執行；同組用戶可以讀和執行，不能寫；其他用戶只能執行，不能讀寫。對目錄來說，x表示搜索許可權，短橫表示不具備相應許可權。

在檔讀寫之前，首先要將其打開，執行打開操作時將檢查其存取權限，若存取權限許可，系統將返回一個小的整數，稱作檔描述符，供後續操作使用；若存取權限不夠則返回一個錯誤碼。

MINIX中另一個重要概念是可安裝檔案系統。多數個人電腦都配有一個或多個軟碟機，為了對此類可移動介質（還包括CD－ROM）提供一種簡潔的訪問方式，MINIX允許將軟碟上的檔案系統連結到主文件樹上。在圖1－7（a）所示的情形中，在調用MOUNT系統調用之前，RAM盤（主存中的一個類比盤）包含最基本的根檔案系統，軟盤機0的軟碟中則包含了另一個檔案系統。

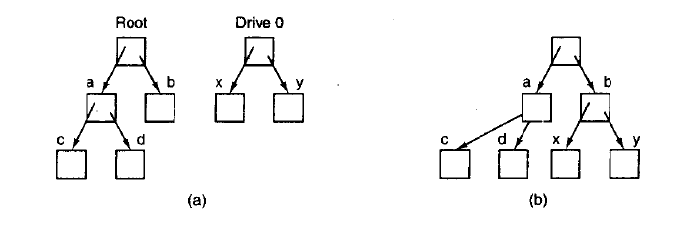


圖 1-7 (a)在安裝前，驅動器0中的檔不可訪問。 (b)安裝後，驅動器0中的檔成為檔樹的一部分。

然而，軟盤機0中的檔案系統無法訪問，因為無法確定其上所存檔的路徑名。MINIX不允許用驅動器名或數位作首碼表示的路徑名，這種表示方式是設備相關的，作業系統應儘量避免。在MINIX中，使用MOUNT系統調用便可以將軟盤機0中的檔案系統連結到根檔案系統的任一目錄。在圖1－7（b）中軟盤機上的檔案系統被安裝到目錄b下，這樣就可以訪問檔/b/x和/b/y。如果目錄b下原先存有檔，則在軟盤機0被安裝期間這些檔無法訪問，因為目錄“/b”現在指向軟盤機0的根目錄。（這種情況通常不會造成很多不便，其原因是檔案系統總是被安裝在空目錄下）

MINIX中另一個重要概念是設備檔（special file）。提供設備檔的目的是使I/O設備使用起來更類似於檔。在這種方式下，設備檔的讀寫可以使用與普通檔相同的系統調用。設備檔分為兩類：塊設備檔（block special files）和字元設備檔（character special files）。塊設備檔指那些由可以隨機存取的資料塊組成的設備，如磁片。打開一個塊設備檔，然後讀第四個塊，程式可以直接訪問設備上的第四塊而不管其檔案系統的格式。類似的，字元設備檔指那些以字元流方式進行操作的設備，如印表機，數據機等。

最後瞭解一下與進程和檔都相關的管道（pipe）。管道是一種用來連接兩個進程的虛擬檔，如圖1－8所示。當進程A欲向進程B發送資料時，它把管道檔視作輸出檔，向其中寫資料，進程B則可將管道檔視作輸入檔，從中讀數據。於是，MINIX中的進程間通信很象普通檔的讀寫。一個進程判斷其輸出是普通檔還是管道的唯一方法是調用一條特殊的系統調用。

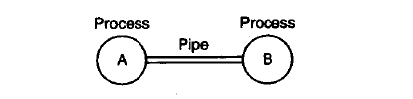


圖 1-8 一個管道連接兩個進程。

### 1.3.3 外殼（shell）

MINIX作業系統是指完成系統調用的代碼，編輯器，編譯器，組合語言程式，連結程式以及命令直譯器等均不是作業系統的組成部分。下面大概介紹一下MINIX的命令直譯器：shell。儘管它本身不是作業系統的一部分，但它體現了作業系統的許多特性並很好地說明了系統調用的具體用法，同時它也是終端使用者與作業系統之間的介面。

當使用者登錄進入系統時，同時將啟動一個shell，它以終端作為標準輸入和標準輸出，它首先顯示系統提示符，這通常是一個美元符號，它提示使用者shell正在等待接收命令。若用戶鍵入

date

則shell創建一個子進程並使其運行date程式。在該子進程運行期間，shell等待它結束。當子進程結束後，shell再次顯示系統提示符並等待下一個輸入。

用戶可以將標準輸出重定向到一個檔，如：

date > file

同樣地，也可以將標準輸入重定向，如：

sort < file1 > file2

該命令將調用sort程式，從file1中取輸入，並將輸出送到file2。

通過管道可以將一個程式的輸出作為另一個程式的輸入，因此

cat file1 file2 file3 | sort > /dev/lp

將調用cat程式將這三個檔合併，結果送到sort程式按字典序排序，sort的輸出又重定向到檔/dev/lp，這正是印表機的設備檔案名（一般將所有設備檔都放在/dev/目錄下）。

如果在一條命令後加上一個“&”符號，則shell將不等待其結束而直接顯示出系統提示符。所以

cat file1 file2 file3 | sort > /dev/lp&

將啟動sort程式作為幕後工作執行，這樣就可以在本條命令尚未結束時允許用戶繼續下面的工作。shell還有許多其他有用的特性，由於篇幅所限，不能一一討論，具體請參閱有關shell的參考書。

## 1.4 系統調用

現在開始討論作業系統與應用程式之間的介面－系統調用。儘管此處的討論基於POSIX標準（國際標準9945－1）和MINIX，但當前多數其他作業系統都有完成相同功能的系統調用，至多在細節上不盡相同。由於系統調用的實現往往與機器有關，而且總是用組合語言表述，所以為使C程式能夠使用系統調用，必須額外提供一個常式庫。

為了清楚地瞭解系統調用的內在機制，我們以READ系統調用為例子。它有三個參數：第一個指定所操作的檔，第二個指定使用的緩衝區，第三個指定要讀的位元組數。在C程式中調用該系統調用的方法如下：

count = read(file, buffer, nbytes);

本系統調用將真正讀到的位元組數返回給count變數，正常情況下這個值與nbytes相等，但當讀至檔結尾符時則可能比nbytes小。

若由於參數非法或磁片操作錯導致該系統調用無法執行，則count被置為-1，同時錯誤碼被放在一個全域變數errno中。程式應該經常檢查系統調用的返回值以確定其是否正確地執行。

MINIX的全部53條系統調用列於圖1－9中，這些系統調用分為六大類。下面將逐個解釋每條系統調用的功能。由於個人電腦的資源管理功能非常有限（起碼和有許多用戶的大型機相比是如此），所以這些系統調用在很大程度上定義了作業系統應提供的主要功能。POSIX標準指定了許多過程，但並未明確規定它們以什麼形式出現，是系統調用、庫函數、或其他別的形式。在有些情況下，POSIX的過程在MINIX中以庫常式形式出現，在其他情況下，鑒於幾個常式僅有細微的差別，則將它們都包括在一條系統調用中。

進程管理

pid = fork()

pid = waitpid(pid, &statloc, opts)

s = wait(&status)

s = execve(name, argv, envp)

exit(status)

size = brk(addr)

pid = getpid()

pid = getpgrp()

pid = setsid()

I = ptrace(req, pid, addr, data)

創建一個與父進程相似的進程

等待一個子進程結束

waitpid的老版本

替換一個進程的核心映射

終止進程的執行並返回狀態

設置資料段大小

返回調用進程的標識號

返回調用進程的組號

創建一個新的會話並返回其組標識

用於調試

信號

s = sigaction(sig, &act, &oldact)

s = sigreturn(&context)

s = sigprocmask(how, &set, &old)

s = sigpending(set)

s = kill(pid, sig)

residual = alarm(setconds)

s = pause()

定義對信號的處理操作

從信號返回

檢查或修改信號遮罩碼

獲得阻塞信號集合

替換信號遮罩碼並使進程掛起

設置時間鬧鐘

將調用進程掛起直到下一個信號

文件管理

fd = creat(name, mode)

fd = mknod(name, mode, addr)

fd = open(file, how, ...)

s = close(fd)

n = read(fd, buffer, nbytes)

n = write(fd, buffer, nbytes)

pos = lseek(fd, offset, whence)

s = stat(name, &buf)

s = fstat(fd, &buf)

fd = dup(fd)

s = pipe(&fd[0])

s = ioctl(fd, request, argp)

s = access(name, amode)

s = rename(old, new)

s = fcntl(fd, cmd, ...)

創建一個檔

創建普通、設備檔或目錄i-節點

打開一個檔進行讀、寫或讀寫

關閉一個打開檔

從一個檔讀數據到一個緩衝區

從緩衝區將資料寫入檔

移動文件指標

獲取一個檔的狀態資訊

獲取一個檔的狀態資訊

為打開檔分配一個新檔描述符

創建一個管道檔

對檔進行特殊操作

檢查檔是否可訪問

文件改名

文件加鎖及其他操作

目錄及檔案系統管理

s = mkdir(name, mode)

s = rmdir(name)

s = link(name1, name2)

s = unlink(special, name, flag)

s = mount(special, name, flag)

s = umount(special)

s = sync()

s = chdir(dirname)

s = chroot(dirname)

創建一個新目錄

刪除一個空目錄

創建一個新檔name2指向name1

刪除一個目錄項

安裝一個檔案系統

卸裝一個檔案系統

將緩衝的資料塊回寫到磁片

改變工作目錄

改變根目錄

保護

s = chmod(name, mode)

uid = getuid()

gid = getgid()

s = setuid(uid)

s = setgid(gid)

s = chown(name, owner, group)

oldmask = umask(complmode)

改變檔的保護位

獲取調用進程的uid

獲取調用進程的gid

設置調用進程的uid

設置調用進程的gid

改變檔的屬主和組

改變模式遮罩碼

時間管理

seconds = time(&seconds)

s = stime(tp)

s = utime(file, timep)

s = times(buffer)

獲取從1970年1月1號以來的時間

設置從1970年1月1號以來的時間

設置檔的最後存取時間

獲取到當前所用的使用者和系統時間

圖 1-9 MINIX的系統調用，返回值s在出錯時為 －1， fd為檔描述符，n為位元組計數。其他返回碼由其字面意思確定。

### 1.4.1 進程管理系統調用

第一類系統調用用於進程管理。先看fork系統調用，fork是創建進程的唯一途徑。它實際上是做一個調用它的進程的精確拷貝，包括檔描述符，寄存器值等所有內容。調用fork後，原進程和所得的拷貝各自執行，互不相關。在執行fork時，二者所有的對應變數都有相同的值，但由於子進程在創建過程中對父進程的資料作了一個拷貝，所以在此之後，其中任一進程中變數值的改變不會對另一個進程產生任何影響（正文段不可修改，它由父、子進程共用）。正確情況下fork的返回值對子進程為0，對父進程為一個正整數，即子進程的標識號pid。通過該返回值可以將父子進程區分開來。

多數情況下，執行完fork後，子進程需要執行與父進程不同的代碼，例如對於一個shell，它首先從終端讀取命令，然後創建一個子進程來執行該命令，等待子進程執行完畢，然後再讀取下一條命令。為了等待子進程結束，父進程執行一條waitpid系統調用。該系統調用使調用進程阻塞直到子進程中的任一個結束（若將其第一個參數置為－1），也可以等待一指定的子進程結束。waitpid結束時，子進程的終止狀態值（正常結束或異常結束，以及正常結束時的返回值）被放在第二個參數指向的位址單元。waitpid有若干選擇項可用。waitpid 取代了先前的wait系統調用，wait系統調用雖然已經過時，但目前仍舊提供，其目的是為了保持相容性。

現在來看shell如何使用fork。當鍵入一條命令時，shell首先創建一個新進程。該子進程需執行該使用者命令，這通過調用exec系統調用實現，exec用其第一個參數指定的可執行檔替換其核心映射，一個高度簡化的shell框架如圖1－10所示。

while (TRUE) { /\* 無限迴圈 \*/

read\_command(command, parameters); /\* 從終端讀取輸入 \*/

if ( fork() != 0 ) { /\* 創建子進程 \*/

/\* Parent code \*/

waitpid( -1, &status, 0); /\* 等待子進程退出 \*/

} else {

/\* Child code \*/

execve(command, parameters, 0); /\* 執行命令 \*/

}

}

圖 1-10 一個簡化的shell，在本書中，TRUE被定義為 1。

一般情況下，exec有三個參數：待執行的檔案名，指向參數陣列的指標和指向環境變數陣列的指標。系統提供了若干庫常式來簡化這些參數的使用，包括execl, execv, execle和execve。本書採用exec來泛指所有這些系統調用。

對於如下一條命令

cp file1 file2

其功能是為檔file1作一個拷貝file2，在shell創建一個子進程後，子進程執行程式cp，同時向該程式傳遞執行的參數：原始檔案名和目的檔案名。

cp程式的主函數格式如下：

main( argc, argv, envp)

這裡argc是命令列中包括程式名在內的參數個數。對於以上例子，argc為3。

第二個參數argv 是一個指向陣列的指標。該陣列中第i個因素就是命令列中第i個字串，此處argv[0]為：“cp”，argv[1]為“file1”，argv[2]為“file2”。

第三個參數envp是一個環境變數指標。環境是由一系列形如“name = value”的字串組成的，用來將環境資訊諸如終端類型，用戶主目錄等傳遞給程式。在圖1－10中，沒有向子進程傳遞任何環境變數，因此execve的第三個參數為空。

EXEC看起來很複雜，但是你不必為此對其他系統調用也感到擔憂，實際上EXEC是最複雜的系統調用，其他系統調用都比它簡單得多，例如EXIT系統調用的作用是結束一個進程，它只有一個參數指定其出口值（0~255）。這個值通過wait和waitpid系統調用中的變數status，返回給父進程。status的低位元組存放結束狀態，0為正常結束，其他值均表示出錯。status的高位元組包含子進程的出口值（0~255）。例如若父進程執行：

n = waitpid( -1, &status, options);

則父進程被阻塞直至有一個子進程結束。如果子進程結束時將4作為exit的參數值，則父進程喚醒時n被置為該結束子進程的pid,而status的值為0x0400。

MINIX中進程的存儲空間分為三部分：正文段（text segment，即程式碼），資料段（data segment，即變數）和堆疊段（stack segment）。資料段是向上增長而堆疊向下增長，如圖1－11所示。在兩者之間是空閒區。堆疊的增長隨程式的執行自動進行，而資料段的擴展則通過BRK系統調用顯式地完成，BRK有一個參數指定資料段的結束位址，它可以比當前值大（資料段擴展），或比當前值小（資料段縮小）。該參數必須小於堆疊指標，否則堆疊和資料段將重疊，這是不允許的。

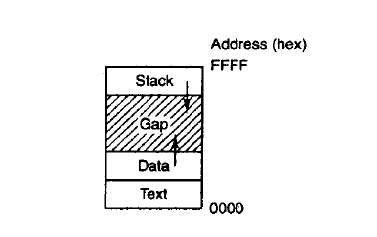


圖 1-11 進程有三個段：正文段、資料段和堆疊段。在本例中，所有三個段在同一個位址空間中，但是也支援分離的指令空間和資料空間。

出於為程式師的方便考慮，系統提供了一個庫函數sbrk來改變資料段大小，它的參數是資料段的變化量（負數表示資料段縮小）。其原理是跟蹤資料段的當前值，（該值由BRK返回）。計算其新的大小，然後申請所需的空間。BRK和sbrk均與實現密切相關，所以不是POSIX的組成部分。

另一個簡單的系統調用getpid返回調用進程的處理序識別碼，注意在調用fork時，只有父進程能夠獲得子進程的處理序識別碼。如果子進程要得到自己的處理序識別碼，它必須使用getpid。同樣地，getgrp返回進程的組標識號。SETSID系統調用啟動一個新的會話（session），並將進程組的pid設為調用者的pid。會話與POSIX的可選特性－作業控制有關，MINIX不支持作業控制，所以對SETSID不作進一步的討論。

最後一個系統調用是ptrace，它被調試器用來對被偵錯工具進行控制，通過ptrace，調試器可以讀寫被控進程的位址空間並實施其他控制。

### 1.4.2 信號管理系統調用

儘管大多數進程間通信是計畫好的，但同時還需要處理不可預知的通信問題，例如用戶使用文字編輯器要求列出一個大檔的全部內容，但隨即他認識到該操作並不需要，這時就需要一種方法來中止編輯器的工作。在MINIX中，用戶可以通過DEL鍵作到這一點，按DEL鍵實際上是向編輯器發送一個信號，編輯器收到此信號即停止列印檔案的內容。信號還可用來報告硬體捕獲到的特定的陷入，如非法指令或浮點運算溢出，超時也通過信號實現。

當一個信號被發送給一個事先並未聲明願意接收它的進程時，該進程只是簡單地被撤銷，即被殺死。進程可以用SIGACTION系統調用來聲明它準備接收某些類型的信號，並同時提供兩個位址：一個是信號處理過程的位址，另一個用於保存該信號的原先處理過程的位址。執行完SIGACTION系統調用後，此進程若接收到相關類型的信號，則先將該進程的當前狀態壓棧，然後調用指定的信號處理過程。信號處理過程可能很長，它本身也可以調用系統調用，但一般情況下，它往往很短。信號處理過程結束後，它調用SIGRETURN以繼續執行被該信號中斷的操作，類似於硬體中的中斷返回。SIGACTION替換了原先的SIGNAL系統調用，出於相容性的考慮，SIGNAL仍然提供。

在MINIX中信號可以被阻塞，被阻塞的信號一直被掛起，直到阻塞解除。這段時間內它不被傳遞，但也不會丟失。SIGPROCMASK系統調用允許一個進程定義其阻塞的信號集，其實現方法是向核心提交一張點陣圖。進程也可以查詢當前因阻塞而掛起的信號集，為此可使用系統調用SIGPENDING，它以點陣圖方式返回該資訊。最後，SIGSUSPEND系統調用使進程可以原子性地設定一張阻塞信號點陣圖並將其掛起。

除了提供一個函數以捕獲一個信號（也就是在接收到該信號時插入執行此函數）外，程式也可以使用常數SIG\_IGN來忽略指定類型的信號，或者使用SIG\_DFL來恢復缺省的信號處理過程。缺省的處理方式隨信號而異，可以是撤銷該進程，或者是忽略該信號。為了說明SIG\_IGN的用法，請看以下命令，其功能是讓shell創建一個後臺進程。

command &

這裡希望DEL信號不要對後臺進程產生影響，所以shell在執行fork之後，exec之前需要執行

sigaction (SIGINT, SIG\_IGN, NULL)；

及

sigaction (SIGQUIT, SIG\_IGN, NULL)；

來忽略DEL和quit信號。（quit信號由CTRL-\產生，它與DEL信號的作用基本相同，不同之處在於如果它未被捕獲或忽略，則它將產生被撤銷進程的核心映射檔）對於前臺進程（命令不帶&），這些信號不能被忽略。

按DEL鍵並不是發送信號的唯一途徑，使用KILL系統調用可以向另一個進程發送信號（前提條件是兩進程有相同的使用者標識號，也即無關的進程間不能發送信號）。再看上一個例子，假設一個後臺進程已被啟動，但隨後發現它應被終止，此時SIGINT和SIGQUIT都已被遮罩，所以只能使用KILL來向它發送一個信號。向後臺進程發送信號9（SIGKILL）將撤銷該進程，SIGKILL不能被捕獲或忽略。

對於許多即時應用，需要在一段指定時間後，中斷進程的原有操作，以進行某種其他處理，例如在不可靠的通信線路上重傳一個丟失的包。為了處理此類情況，系統提供了ALARM系統調用，ALARM的參數指定一個以秒為單位的時間間隔，一旦該時間段到點就向該進程發送一個SIGALRM信號。在任意時刻一個進程只能設定一個時間鬧鐘。如果進程先設定一個10秒的時間鬧鐘，在3秒後又設定一個20秒的時間鬧鐘，則只有其中一個有效，即在第二個ALARM調用之後20秒會發送一個SIGALRM信號。如果ALARM的參數為0，則所有掛起的SIGALRM信號都被取消。若不捕獲SIGALRM信號，則其缺省的處理方法是撤銷該進程。

某些情況下進程在信號到達之前不要做任何操作，例如一個測驗閱讀和理解速度的CAI系統，它先在螢幕上顯示一些課文，然後調用ALARM設定在30秒後向自己發送一個信號，以啟動程式進行一些處理。當學生閱讀課文時，程式無需執行任何操作。它可以採用的一種方法是執行空操作迴圈以等待時間到，但假如此時系統中還有其他進程運行，這將浪費CPU時間，好的方法是使用PAUSE系統調用，它將掛起調用進程直至信號到來，這段時間裡別的進程便可以使用CPU。

### 1.4.3 檔管理系統調用

許多系統調用與檔案系統有關，這裡僅討論對單個檔進行操作的系統調用，下一節將討論對目錄和檔案系統進行操作的系統調用。創建新檔要使用CREATE系統調用，其參數指定檔路徑名和保護模式，所以

fd = creat ("abc", 0751);

將創建一個名為abc的檔，其保護模式為0751（在C語言中，開頭的0表示一個常數為八進制），它的低9位指明檔主（7表示可讀、寫、執行），同組用戶（5表示可讀、執行）及其他用戶（1表示只可執行）的操作許可權。

CREAT在創建檔的同時還以寫方式將其打開，而不管檔模式是什麼。CREAT返回的檔描述符fd可用於對該檔執行寫操作。如果對一個已經存在的檔進行CREAT操作，在操作許可權許可的情況下該檔內容將被破壞，CREAT屬於已經過時的系統調用，因為OPEN系統調用也可以創建新檔，但系統仍提供CREAT以保持相容性。

創建設備檔使用MKNOD，而不是CREAT，典型的用法為：

fd = mknod("/dev/ttyc2", 020744, 0x0402);

這將創建一個名為“/dev/ttyc2”的檔（二號控制台通常用的檔案名），並將其模式代碼置為八進制的020744（意為該檔是字元設備檔，保護模式為rwxr--r--），第三個參數的高位元組指定其主設備號為4，低位元組指定其次設備號為2。主設備號可以取任何值，但名為/dev/ttyc2的檔次設備號應當為2。MKNOD只能被超級用戶使用。

讀寫一個檔之前首先必須用OPEN系統調用將其打開。OPEN的第一個參數指定文件路徑名，可使用絕對路徑名或相對於當前工作目錄的相對路徑名；第二個參數指定打開方式O\_RDONLY,O\_WRONLY或O\_RDWR（分別表示唯讀，只寫和可讀可寫），OPEN返回的文件描述符可用於文件讀寫。檔在操作完畢後要用CLOSE關閉，這樣該檔描述符可供其後的CREAT和OPEN系統調用再次使用。

最常用的系統調用當屬READ和WRITE，下面將對READ進行詳細討論，WRITE與它具有相同的參數。

多數程式對檔的讀寫操作都是順序進行的，但有些卻需要隨機地訪問檔的任意部分。每個檔都有一個指標指明其當前讀寫位置。在順序讀寫時，該指標通常指向下次要讀寫的位元組。使用LSEEK系統調用可以直接修改檔指標的值，這樣隨後的READ或WRITE就可在檔的任一位置進行操作，甚至可以超越檔案結尾。

LSEEK有三個參數：第一個指定檔描述符，第二個指定檔的位置，第三個指明檔位置是相對於檔開頭、當前位置、還是檔案結尾。LSEEK的返回值是檔指針被修改之後的絕對位置。

對每一個檔，MINIX記錄了如下資訊，包括檔案類型（普通檔、設備檔以及目錄等），檔大小，最後修改時間等等。程式可以通過STAT和FSTAT系統調用獲取這些資訊，其不同之處僅在於STAT通過檔案名來指定檔，而FSTAT則使用檔描述符，這樣FSTAT很適用於已打開的檔，尤其是標準輸入和標準輸出這類檔案名可能不可知的情況。STAT和FSTAT的第二個參數指定一個用來存放所獲取資訊的資料結構，如圖1－12所示。

struct stat {

short st\_dev /\* i-節點所駐留的設備 \*/

unsigned short st\_ino; /\* i-節點號 \*/

unsigned short st\_mode; /\* 模式 \*/

short st\_nlink; /\* 連結數 \*/

short st\_uid; /\* 使用者標識號 \*/

short st\_gid; /\* 組標識號 \*/

short st\_rdev; /\* 設備檔的主/次設備號 \*/

long st\_size; /\* 文件大小 \*/

long st\_atime; /\* 最後存取時間 \*/

long st\_mtime; /\* 最後修改時間 \*/

long st\_ctime; /\* 對i-節點最後修改時間 \*/

};

圖 1-12 STAT和FSTAT系統調用所用的存放返回資訊的資料結構。在實際代碼中，其中的某些資料類型使用符號名。

DUP系統調用常用於對檔描述符的操作，例如一個程式需要關閉標準輸出（檔描述符為1），代之以使一個普通檔成為標準輸出，隨後向標準輸出寫一些資訊，最後恢復原先的狀態。為實現這些功能可以先關閉檔描述符1，再打開另一個檔，這時該檔就成為標準輸出（設標準輸入正被使用），但這樣處理無法恢復原先的標準輸出。

解決辦法是先調用

fd = dup(1);

該操作將為標準輸出分配一個新的檔描述符fd，並使對fd的操作與直接對標準輸出的操作完全一樣。隨後將標準輸出關閉，再打開一個新檔，至此該檔將被作為標準輸出。當需要恢復原先的標準輸出時，先關閉檔描述符1，再執行

n = dup(fd);

將最小的檔描述符號，即1，定向到fd所指的檔，最後將fd關閉就恢復了最初狀態。

DUP系統調用有一個變種，它允許將任一未使用的檔描述符定向到一個指定的打開檔，其調用方法為：

dup2 (fd, fd2);

此處fd指向一打開的檔，fd2為一個未使用的檔描述符，執行完這條語句後fd2將指向fd所指向的文件。若fd指向標準輸入（檔描述符0），fd2為4，則在此調用後描述符0和4都指向標準輸入。

如前所述，MINIX中的進程間通信使用管道，若一用戶鍵入

cat file1 file2 | sort

shell將創建一個管道並將第一個進程的標準輸出資訊寫到管道中，於是第二個進程的標準輸入便可以從該管道中讀取。PIPE系統調用創建一個管道並返回兩個檔描述符，一個用於寫，另一個用於讀，PIPE的調用格式為：

pipe(&fd[0]);

這裡fd是由兩個整數組成的陣列，fd[0]存放供讀使用的檔描述符，fd[1]存放供寫使用的檔描述符。通常典型的用法是在本條語句之後調用一個fork創建一個子進程，然後父進程關掉用於讀的檔描述符，子進程關掉用於寫的檔描述符（或者相反），這樣便可以做到一個進程從管道讀數據，另一個向管道寫資料。

圖1－13提供了一個框架過程，其中創建了兩個進程，通過管道將進程1的輸出導向進程2（更實際的例子還要進行錯誤檢查，並作參數處理）。其處理過程如下：首先創建一個管道，隨後調用fork，將父進程作為管道中的進程1，子進程作為進程2，由於待運行的兩個檔process1和process2並不知道它們是管道的一部分，所以必須對檔描述符進行控制以使進程1的標準輸出和進程2的標準輸入都指向管道。父進程首先關掉從管道讀的檔描述符和標準輸出，隨後執行DUP，這樣就使檔描述符1可被用於向管道寫。注意DUP總是返回最小的可用檔描述符，在這裡即為1。然後程式關閉另一個管道檔描述符。

#define STD\_INPUT 0 /\* 標準輸入的檔描述符 \*/

#define STD\_OUTPUT 1 /\* 標準輸出的檔描述符 \*/

pipeline(process1, process2)

char \*process1, \*process2; /\* 指向程式名的指標 \*/

{

int fd[2];

pipe(&fd[0]); /\* 創建一個管道 \*/

if (fork() != 0) {

/\* 父進程執行如下語句 \*/

close(fd[0]); /\* 進程1不需要從管道讀 \*/

close(STD\_OUTPUT); /\* 準備新的標準輸出 \*/

dup(fd[1]); /\* 將標準輸出指向fd[1] \*/

close(fd[1]); /\* 此檔描述符不再需要 \*/

execl(process1, process1, 0);

} else {

/\* 子進程執行如下語句 \*/

close(fd[1]); /\* 進程2不需要向管道寫 \*/

close(STD\_INPUT); /\* 準備新的標準輸入 \*/

dup(fd[0]); /\* 將標準輸入指向fd[0] \*/

close(fd[0]); /\* 此檔描述符不再需要 \*/

execl(process2, process2, 0);

}

}

圖 1-13 建立一個兩進程的管道的程式框架。

在EXEC系統調用之後，父進程將保留檔描述符0和2，而檔描述符1則用於向管道中寫。子進程的代碼與父進程類似。execl的參數被重複，因為其第一個參數是待執行的檔案名，而第二個參數是執行檔的第一個參數，對多數程式來說，該參數都是檔案名。

下一個系統調用IOCTL適用於所有設備檔，例如它可用於設備驅動程式，SCSI設備驅動程式用它來控制磁帶機和CD－ROM。但其主要還是用於字元設備檔，尤其是終端。POSIX定義了許多函數，最終都轉化為IOCTL調用。庫函數tcgetattr和tcsetattr使用IOCTL來改變終端的模式和各種屬性。

終端最常用的模式是Cooked模式，在該模式下刪除鍵和終止鍵能正常地工作，CTRL\_S和 CTRL\_Q用來停止和恢復終端輸出，CTRL\_D為檔結束符，按DEL鍵產生一個中斷信號，而CTRL-\產生一個退出信號並強制進行核心映射轉儲。

在raw模式下，所有這些功能都被取消，每個字元都被不加處理地送給程式，而且不等一行結束就將從終端讀到每一個字元發送給程式。與此不同的是在Cooked模式下，終端輸入的資料等到一行結束才送給程式。

Cbreak模式介於上述兩者之間，用作編輯的刪除鍵和終止鍵，以及CTRL\_D被遮罩，但CTRL\_S,CTRL\_Q,DEL和CTRL\_\則仍然有效，與raw模式一樣，單個字元不等一行結束就送給程式（如果禁止行內編輯功能，則沒必要等待接收到完整的一行，因為使用者不可能象在cooked模式下那樣改變主意並刪除它）。

POSIX並不採用以上所列的術語Cooked, raw 和 Cbreak，POSIX中的正規模式對應於Cooked模式。在正規模式中定義了11個特殊字元，輸入也是以行為單位進行。在POSIX的非正規模式中，讀數據操作由一個最小接受字元數和一個以0.1秒為單位的時間段決定。POSIX標準有具有很大的靈活性，其中有許多標誌可以被設置，使得非正規模式使用起來很象Cooked模式和raw模式。原先的術語：Cooked模式、raw模式和Cbreak模式更具描述性，因此以後仍使用它們。

IOCTL有三個參數，例如調用tcsetattr函數設置終端參數最終將轉換成以下調用：

ioctl (fd, TCSETS, &termios);

第一個參數指定一個檔，第二個參數指定操作類型，第三個參數指定一個POSIX資料結構的位址，其中包含了各種標誌及控制字元的陣列。除了TCSETS以外，還有一些其他的操作碼，其功能包括：推遲對終端參數所作的修改直到全部輸出被送出，將未讀取的輸入資訊作廢，返回參數的當前值等。

ACCESS系統調用檢查對一個檔是否具有某種存取權限，因為有些程式可以用另一使用者的使用者標識號運行，所以ACCESS非常有用。SETUID機制將在稍後介紹。

RENAME系統調用將檔更名，其參數分別指定老檔案名和新檔案名。

FCNTL系統調用對檔進行控制，它有些類似IOCTL（這兩條系統調用一般被熟練程式師使用）。FCNTL有若干選項，最常用的是文件加鎖。使用FCNTL可以對一個檔的一部分加鎖和解鎖，也可以檢測一個檔的某個部分是否被上鎖。FCNTL本身不包含任何與鎖操作有關的語義，這要由程式師自行定義。

### 1.4.4 目錄管理系統調用

下面介紹對目錄和檔案系統進行操作的系統調用。首先是MKDIR和RMDIR，分別用來創建和刪除空目錄。LINK系統調用允許同一個檔按不同路徑名出現，一種典型的應用是允許開發小組的幾個成員共用一個檔，同時該檔出現在每個人自己的目錄下。共用一個檔不同於給每個人一個私有的拷貝，對前者，任何一個人所作的修改都對其他人可見－只存在一個檔；而對後者，所作的修改只有自己可見，而不會更新其他人的拷貝。

圖1－14的例子解釋了LINK的工作原理，兩個用戶ast和jim，在他們各自的目錄下都有一些檔，如果ast執行以下語句：

link ("/usr/jim/memo", "/usr/ast/note" );

則jim目錄下的檔memo將以檔案名note出現在ast的目錄下，此後/usr/jim/memo和/usr/ast/note指的是同一個文件。

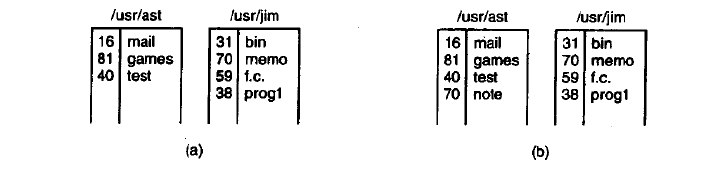


圖 1-14 (a)將/usr/jim/memo連結到ast的目錄之前的兩個目錄。 (b)同樣兩個目錄在連結後。

理解LINK的工作原理有助於掌握其功能，MINIX中的每個檔都有一個唯一的數字 － i-節點（i-node）號來標識它。i-節點號是i-節點表的索引值，每個檔有一個i-節點，裡面存放有檔主以及該檔所佔用的磁片塊等資訊。目錄實際上也是檔，只是其內容存放的是一些i-節點和檔案名的對應資訊。在圖1－14中，文件mail的i-節點號為16，其他文件都與此類似。LINK所做的只是創建一個新的目錄項，它有一個新檔案名，但i-節點號則是被連結的檔的i-節點號。圖1－14（b）中，兩個目錄項有相同的i-節點號70，所以它們指向同一個檔。如果兩個目錄項的任何一個以後用UNLINK系統調用刪除，另外一項還存在，則相關檔也繼續存在；如果兩項都被刪除，那麼MINIX檢測到沒有目錄項指向該檔（i-節點中包含一個欄位，它記錄指向該檔的目錄項數），則該檔被從磁片上刪除。

如前所述，MOUNT系統調用可將兩個檔案系統合併成一個。通常情況是先有一個存在於RAM盤上的根檔案系統，其中包含有常用命令的可執行檔及其他常用檔。然後用戶可以在驅動器0中插入一張存有使用者程式的軟碟。使用MOUNT系統調用就可以將軟碟上的檔案系統安裝到根檔案系統下，如圖1－15所示。執行安裝操作的典型C語句為：

mount ("/dev/fd0", "/mnt", 0)；

其中第一個參數是軟盤機的設備檔案名，第二個參數指定在檔樹中的安裝點。

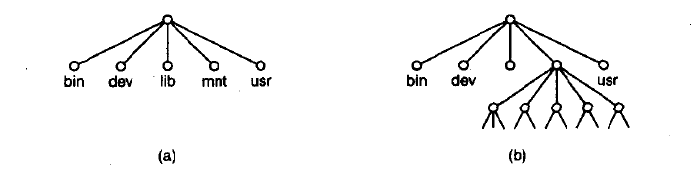


圖 1-15 (a)安裝前的檔案系統。 (b)安裝後的檔案系統。

執行完MOUNT後，軟盤機上的檔即可通過路徑名訪問而與具體的物理設備無關，就是說軟碟插入哪一個驅動器都是可以的。使用MOUNT命令，用戶可以將可移動介質集成到單一的集成的檔層次結構中而不必關心檔在哪台具體設備上。上例中只涉及到軟碟，實際上硬碟，CD－ROM和硬碟的分區（或稱次設備）都可以這樣安裝，當一個檔案系統不再需要時，可以用UMOUNT系統調用將其卸裝。

MINIX在記憶體中開闢了一個緩衝區以保存最近經常訪問的磁片資料，這樣可以避免再重複地從磁片上讀數據。如果緩衝區中的某資料塊被修改，而在其被寫回磁片之前系統發生崩潰，那麼檔案系統可能被損壞。為避免這種現象，必須週期性地將這些緩衝區中的資料寫回磁片。系統調用SYNC用來將被修改的緩衝區資料寫回磁片。MINIX啟動後，一個名為update的程式被啟動作為後臺進程運轉，它每隔30秒執行一次SYNC操作，將更新了的資料寫回磁片。

與目錄操作有關的另外兩個系統調用是CHDIR和CHROOT，CHDIR改變當前工作目錄，CHROOT改變根目錄。若先調用

chdir ("/usr/ast/test")；

然後打開文件xyz，則這將打開"/usr/ast/test/xyz"文件。CHROOT的功能與此類似，進程改變其根目錄後所有絕對路徑名都將從該新的根目錄開始。只有超級用戶可以執行CHROOT，而且即使超級用戶也很少使用CHROOT。

### 1.4.5 許可權管理系統調用

MINIX中每個檔都有一個包含11個比特的保護方式碼，其中的9比特標識檔主，同組用戶和其他用戶的操作許可權。CHMOD系統調用可以改變檔的保護方式，例如，要使一個檔對檔主之外的所有用戶唯讀，可以調用：

chmod("file", 0644);

保護方式碼的另外兩個比特，02000和04000分別是SETGID位和SETUID位。使用者在執行一個SETUID位元設置的程式時，在進程運行期間它的有效使用者標識被設為檔主的使用者標識號uid。該特性常被用來使一般用戶可以執行那些通常只有超級用戶才能執行的功能。例如只有超級使用者才能使用MKNOD來創建設備檔，將MKNOD程式的屬主置為超級用戶，同時將其保護碼設為04755就可以使一般用戶能夠執行MKNOD，而同時又對一般用戶施以嚴格的限制。

當進程執行一個SETUID或SETGID位設置的檔時，它便獲得了與其真實uid或gid不同的有效uid或gid。有時進程需要獲得其真實和有效的uid和gid， MINIX為此提供了GETUID和GETGID系統調用以獲取這些資訊。GETUID同時返回真實uid和有效uid，GETGID同時返回真實gid和有效gid。為此，系統提供了四個庫常式來獲取相應的資訊，分別是：getuid, getgid, geteuid, getegid。

一般用戶不能改變他們的uid，除非執行一個SETUID的程式。但超級用戶則擁有特別的權力：他可以使用SETUID系統調用設置自己的有效和真實uid，也可以使用SETGID設置有效和真實gid，他還可以使用CHOWN改變檔的屬主。總之超級使用者不受系統保護規則的約束。

在許可權管理方面，還有兩條系統調用可以被一般使用者進程執行。UMASK設置一個內部遮罩，它在創建檔時使用。例如執行了

umask(022)；

後，CREAT和MKNOD指定的文件許可權碼都要與022的反碼（755）相與，所得的結果才是最終的保護碼，所以如下調用：

creat ("file", 0777);

將把檔的保護碼設置為0755而非0777。由於該遮罩被子進程繼承，所以如果在登錄後shell執行了一次UMASK操作，則此次登錄期間所有的使用者進程都將受到該遮罩的約束，這樣就保證不會因為用戶的疏忽而導致檔被非法訪問。

對於一個檔主為ROOT的SETUID程式，因為其有效使用者標識為超級使用者，所以它可以訪問任何檔。在許多情況下，使程式瞭解調用此程式的使用者對某個檔是否擁有存取權限是很有意義的。

實際上有意義的是檢查真正uid是否對檔有存取權限，ACCESS系統調用提供了這樣的檢查方法，ACCESS的mode參數若為4則檢查讀許可權，為2則檢查寫許可權，為1檢查執行許可權，而且允許使用這幾者的組合。例如若mode為6，則只有當進程的真正uid使用者對檔可讀可寫時再返回成功（0），否則返回-1。當mode為0時僅檢查檔是否存在，同時檢查從根目錄直到該檔的所有目錄是否允許搜索。

1.4.6 時間管理系統調用

MINIX有4條用於時間管理的系統調用。TIME系統調用返回當前距1970年1月1日零時的時間，以秒為單位。STIME用來設置系統時間（僅由超級用戶執行）。UTIME允許檔主（或超級使用者）修改存儲在檔i-節點中的時間，例如touch命令就使用UTIME將檔時間設為當前時間。

最後是TIMES系統調用，它返回進程的計賬資訊，通過它可以知道進程已經佔用了多少CPU時間，以及作業系統本身的操作所用的時間（即執行系統調用），同時也返回其所有子進程所用的使用者時間與系統時間的總和。

**1.5 作業系統結構**

至此我們已經瞭解了作業系統的外部特性（即程式師介面），現在開始觀察其內部的組成結構。下面將討論四種我們都實際嘗試過的組織結構，即整體式系統，層次式系統，虛擬機器系統和客戶－伺服器系統。這四種結構並不包括一切，但通過對它們的研究我們將對實際作業系統的設計建立一些基本的概念。

**1.5.1 整體式系統**

整體式系統是最常用的組織方式，但常被人們形容為“一鍋粥”，其結構實際就是“無結構”，整個作業系統是一堆過程的集合，每個過程都可以調用任意其他過程。使用這種技術時，系統中的每一過程都有一個定義完好的介面，即它的人口參數和返回值，而且相互間的調用不受約束。

在整體式系統中，為了構造最終的目標作業系統程式，開發人員首先將一些獨立的過程進行編譯，然後用連結程式將其連結在一起成為一個單獨的目的程式。從資訊隱藏的觀點看，它沒有任何程度的隱藏－每個過程都對其他過程可見。（與此相對的是將系統分成若干個模組，資訊被隱藏在這些模組內部，在外部只允許從預先定好的調用點訪問這些模組）

但即使在整體式系統中，也存在一些程度很低的結構化。作業系統提供的服務（系統調用）的調用過程是這樣的：先將參數放入預先確定的寄存器或堆疊中，然後執行一條特殊的陷入指令，即訪管指令或核心調用（kernel call）指令。

這條指令將機器由使用者態切換到核心態，並將控制轉到作業系統。該過程如圖1－16所示。（多數CPU有兩種狀態：核心態：供作業系統使用，該狀態下可以執行機器的所有指令；用戶態：供使用者程式用，該狀態下I/O操作和某些其他操作不能執行。）

作業系統隨後檢查該調用的參數以確定應執行哪條系統調用，這如圖1－16（2）所示。然後，作業系統查一張系統調用表，其中記錄了每條系統調用的執行過程，這一步操作如圖1－16（3）所示，它確定了將調用的服務過程。當系統調用結束後，控制又返回給使用者程式（第4步），於是繼續執行系統調用後面的語句。

圖 1-16 系統調用的操作過程：(1)使用者程式陷入核心 (2)作業系統確定所請求的服務編號 (3)作業系統調用服務過程 (4)控制返回到使用者程式

這種組織方式提出了作業系統的一種基本結構：

1 一個用來調用被請求服務常式的主程序。

2 一套執行系統調用的服務常式。

3 一套支援服務常式的實用過程。

在這種模型中，每一條系統調用都由一個服務常式完成；一組實用過程用來完成若干服務常式都需要用到的功能，如從使用者程式獲取資料等，這種將各種過程分為三層的模型如圖1－17所示。

圖 1-17 整體式作業系統的簡單結構模型。

**1.5.2 層次式系統**

圖1－17所示的系統進一步通用化就成為層次式系統，即上層軟體基於下層軟體之上。按此模型構造的第一個作業系統是E.W.Dijkstra和他的學生在荷蘭的 Eindhoven 技術學院開發的THE系統（1968年）。THE系統是為荷蘭製造的Electrologica X8電腦（記憶體為32K個27比特的字）配備的一個簡單的批次處理系統。

該系統分為六層，如圖1－18所示。第零層進行處理器分配，當發生中斷或時鐘到達期限時由該層軟體進行進程切換。在第零層之上有若干個順序進程運行，編寫這些進程時就不用再考慮多個進程在單一處理器上運行的細節，總之，第零層提供了CPU基本的多道程序功能。

圖 1-18 THE作業系統的結構。

第一層進行記憶體管理，它為進程分配記憶體空間，當記憶體用完時則會在用作對換的512K字的磁鼓上分配空間。在第一層之上，進程不用再考慮它是在記憶體還是在磁鼓上，因為第一層軟體保證在需要訪問某一頁面時，它必定在記憶體中。

第二層軟體處理進程與操作員控制台之間的通信，在第二層之上則可認為每個進程都有它自己的操作員控制台。第三層軟體管理I/O設備和相關的資訊流緩衝。在第三層之上，每個進程都與適當抽象了的設備打交道而不必考慮物理設備的細節。第四層是使用者程式層，使用者程式在此不必考慮進程、記憶體、控制台和I/O設備等環節。系統操作員進程位於第五層。

MULTICS對層次化概念進行了更進一步的通用化，它不採用層而是由許多同心的環構成，內層的環比外層的環有更高的特權級。當外層環的程序呼叫內層環的過程時，它必須執行一條類似系統調用的TRAP指令，TRAP指令執行前要進行嚴格的參數合法性檢查。儘管在MULTICS中作業系統是各個使用者進程位址空間的一部分，硬體仍然能夠對單個進程（實際上是記憶體中的一個段）的讀，寫和執行許可權進行保護。

實際上THE分層方案只是在設計上提供了一些方便，因為系統的各個部分最終仍然被連結成一個完整的單個目的程式，而在MULTICS中，上述環形方案在運行中是實際存在的且由硬體實現。環形方案的一個優點是它很容易被擴展，以構造用戶子系統。例如在一個系統中，教授可以寫一個程式來檢查學生編寫的程式並打分，將教授的程式放在第n個環中運行，而將學生的程式放在第n+1個環中運行，則學生無法篡改教授給出的成績。

**1.5.3 虛擬機器系統**

OS/360的最早版本是純粹的批次處理系統，然而許多360的用戶希望使用分時系統，於是IBM公司和另外的一些研究小組決定開發一個分時系統。隨後IBM提供了一套分時系統TSS/360，但它非常龐大，運行緩慢，幾乎沒有什麼人用，該系統在花費了約五千萬美元的研製費用後最終被棄之不用（Graham, 1970）。但IBM設在麻省劍橋的一個研究中心開發了一個完全不同的系統，最終被IBM用作為產品，該系統目前仍然在IBM的大型主機上廣泛使用。

該系統最初命名為CP/CMS，後來改為VM/370（Seawright and Mackinnon,1979）。它基於如下的思想：一個分時系統應該提供以下特性：（1）多道程序， （2） 一個具有比裸機更方便、介面擴展的電腦。VM/370的主旨在於將此二者徹底地隔離開來。

該系統的核心稱作虛擬機器監控程序，它在裸機上運行並具備多道程序功能。它向上層提供了若干台虛擬機器，這如圖1－19所示。但與其他作業系統不同的是：這些虛擬機器不是那種具有檔等良好特徵的擴展電腦，而僅僅是裸機硬體的精確複製，它包含有：核心態/用戶態，I/O功能，中斷，以及真實硬體具有的全部內容。

圖 1-19 帶CMS的VM/370結構。

因為每台虛擬機器都與裸機完全一樣，所以每台虛擬機器可以運行裸機能夠運行的任何作業系統。不同的虛擬機器可以運行不同的作業系統而且往往如此。某些虛擬機器運行OS/360的後續版本作批次處理或交易處理，而同時另一些運行一個單使用者交互系統供分時使用者使用，該系統稱作CMS（會話監控系統）。

當CMS上的程式執行一條系統調用時，該系統調用陷入其自己的虛擬機器的作業系統，而不是VM/370，這就象在真正的電腦上一樣。CMS然後發出正常的硬體I/O指令來執行該系統調用。這些I/O指令被VM/370捕獲，隨後VM/370執行這些指令，作為對真實硬體類比的一部分。通過將多道程序功能和提供虛擬機器分開，它們各自都更簡單，更靈活和易於維護。

現在虛擬機器的思想被廣泛採用：例如在奔騰CPU（或其他Intel的32位CPU）上運行老的MS－DOS程式。在設計奔騰晶片的硬體和軟體時，Intel和Microsoft都意識到要使老軟體能夠在新硬體上運行，於是Intel在奔騰晶片上提供了一個虛擬8086模式，在此模式下，奔騰機就象一台8086電腦一樣，包括1M位元組內的16位元定址方式。

虛擬8086模式被MS-Windows, OS/2及其他作業系統用於運行MS－DOS程式。程式在虛擬8086模式下啟動，執行一般的指令時它們在裸機上運行，但是當一個程式試圖陷入系統來執行一條系統調用時，或者試圖執行受保護的I/O操作時，將發生一條虛擬機器監控程序的陷入。

此時有兩種設計方法：第一種，MS－DOS本身被裝入虛擬8086模式的位址空間，於是虛擬機器僅僅將該陷入傳回給MS－DOS，這種處理與在真正的8086上運行是一樣的，當MS－DOS後來試圖自行執行I/O操作時，該操作被捕獲而由虛擬機器監控程序完成。

另一種方法是虛擬機器監控程序僅僅捕獲第一條陷入並自己執行I/O操作，因為它知道所有的MS－DOS系統調用，並且由此知道每條陷入企圖執行什麼操作。這種方法不如第一種方法純，因為它僅僅正確地模擬了MS－DOS，而不包括其他作業系統，相比之下，第一種方法可以正確地類比其他作業系統。但另一方面，這種方法很快，因為它不再需要啟動MS－DOS來執行I/O操作。在虛擬8086模式中真正地運行MS－DOS的另一個缺點是MS－DOS頻繁地對中斷遮罩位進行操作，而模擬這些操作是很費時的。

需要注意的是上述方法都不同於VM/370，因為它們模擬的並不是完整的奔騰硬體而只是一個8086。在VM/370系統中，可以在虛擬機器上運行VM/370本身，而在奔騰系統中，不可能在虛擬8086上運行Windows，因為Windows不能在8086晶片上運行。其最低版本也需要在80286上運行，然而奔騰晶片不提供對80286的模擬。

在VM/370中，每個使用者進程獲得真實電腦的一個精確拷貝，在奔騰晶片的虛擬8086模式中，每個使用者進程獲得的是另一套硬體（8086）的精確拷貝。進一步而言，M.I.T的研究人員構造了一個系統，其中每個使用者都獲得真實電腦的一個拷貝，只是佔用的資源是全部資源的一部分（Engler et al.， 1995）。於是一台虛擬機器可能佔用磁片的第0塊到1023塊，另一台可能佔用1024到2047塊等等。

在核心態下運行的最底層軟體是一個稱作“外核”（exokernel）的程式，其任務是為虛擬機器分配資源並確保資源的使用不會發生衝突。每台用戶層的虛擬機器可以運行其自己的作業系統，就象VM/370和奔騰的虛擬8086一樣。不同在於他們各自只能使用分配給它的那部分資源。

“外核”方案的優點在於它省去了一個映射層。在別的設計方案中，每台虛擬機器認為它有自己獨立的磁片，編號從0到最大，於是虛擬機器監控程序必須維護一張表來完成磁片位址的映射（也包括其他資源）。有了“外核”之後，這種映射就不再需要了。外核只需記錄每台虛擬機器被分配的資源。這種方法的另一個好處是以較少的開銷將多道程序（在外核中）與使用者作業系統代碼（在使用者空間）分離開來，因為外核需做的工作是使各虛擬機器互不干擾。

**1.5.4 客戶/伺服器系統**

VM/370將傳統作業系統的大部分代碼（實現擴展的電腦）分離出來放在更高的層次上，即CMS，由此使系統得以簡化，但VM/370本身仍然非常複雜，因為類比許多虛擬的370硬體不是一件簡單的事情（尤其是還想作得高效時）。

現代作業系統的一個趨勢是將這種把代碼移到更高層次的思想進一步發展，從作業系統中去掉盡可能多的東西，而只留一個最小的核心。通常的方法是將大多數作業系統功能由使用者進程來實現。為了獲取某項服務，比如讀檔中的一塊，使用者進程（客戶進程client process）將此請求發送給一個伺服器進程（server process），伺服器進程隨後完成此操作並將回答資訊送回。

該模型示於圖1－20，核心的全部工作是處理客戶與伺服器間的通信，作業系統被分割成許多部分，每一部分隻處理一方面的功能，如檔服務、進程服務、終端服務或記憶體服務。這樣每一部分變得更小，更易於管理。而且由於所有伺服器以使用者進程的形式運行，而不是運行在核心態，所以它們不直接訪問硬體。這樣處理的結果是：假如在檔案伺服器中發生錯誤，檔案伺服器可能崩潰，但不會導致整個系統的崩潰。

圖 1-20 客戶/伺服器模型。

客戶－伺服器模型的另一個優點是它適用於分散式系統（參閱圖1－21）。如果一個客戶通過消息傳遞與伺服器通信，客戶無需知道這條消息是在本機就地處理還是通過網路送給遠地機器上的伺服器。在這兩種情況下，客戶機的處理都是一樣的：發送一個請求，收回一個應答。

圖 1-21 在一個分散式系統中的客戶/伺服器模型。

圖1－21中所描繪的核心只處理客戶與伺服器之間的消息傳遞，這與實際系統不完全符合。某些作業系統功能（如向物理I/O設備寄存器寫入命令字）靠使用者空間的程式是很難完成的，解決這個問題的方法有兩種：一種是設立一個運行於核心態的專用伺服器進程，它具有訪問硬體的絕對權力，但仍舊通過平常的消息機制與其他進程通信。

另一種方法是在核心中建立一套最少的機制（mechanism），而將策略（policy）留給使用者空間中的伺服器進程。例如核心可能將向某特定位址發送的一條消息理解為：取該消息的內容並將其裝入某台磁片的I/O設備寄存器來啟動讀盤操作。此例中核心甚至不對消息的內容進行合法性檢查，而只是將它們機械地拷貝進磁片設備寄存器（顯然這裡需要使用某種方案以限制此類消息只能發給授權的進程）。機制與策略分離是一個重要的概念，它在作業系統的許多方面都經常出現。

**1.6 各章節內容簡介**

典型的作業系統由四部分構成：進程管理，I/O設備管理，記憶體管理和文件管理。MINIX也分為這樣四部分，隨後的四章將分別討論這四個論題，第六章列出了閱讀材料和參考文獻。

第二章到第五章結構大致相同，首先闡述該論題的基本原理，然後介紹MINIX中的相應內容（同樣適用於UNIX）。最後詳細地討論MINIX中的實現。如果讀者僅對系統的基本原理感興趣，他可以略過實現部分的內容而不會引起任何的不連貫。但是如果希望瞭解一個真實的作業系統（MINIX）是如何工作的，那麼應該閱讀包括MINIX代碼在內的全部內容。

**1.7 小結**

對作業系統有兩種觀點：資源管理器觀點和擴展的電腦觀點。從資源管理器觀點看，作業系統的任務是高效地管理整個系統的各個部分；從擴展的電腦觀點看，其任務是為使用者提供一台比物理電腦更易於使用的虛擬電腦。

作業系統的歷史很長，從早期的代替操作員手工操作的系統，到現在的多道程序系統。

任何作業系統的核心都是一套系統調用，系統調用界定了作業系統能完成的功能。對於MINIX，其系統調用分為六大類，第一類與進程創建和終止有關。第二類處理信號。第三類針對文件讀寫。第四類進行目錄管理。第五類對資訊進行保護。第六類用於時間管理。

作業系統有若干種構造模型，常見的有整體式模型，層次式模型，虛擬機器模型和客戶－伺服器模型。

習 題

1. 作業系統的兩個主要功能是什麼？

2. 什麼是多道程序？

3. 什麼是spooling？你認為將來高檔個人電腦會將spooling作為標準特性嗎？

4. 在早期的電腦中，每一個位元組資料的讀寫都是CPU直接進行處理的（那時沒有DMA－直接記憶體訪問），這種組織結構對多道程序技術的出現有什麼影響？

5. 為什麼分時計算沒有被第二代電腦廣泛採用？

6. 下列哪種指令應該只在核心態下執行？

1 遮罩所有中斷

2 讀時鐘日期

3 設置時鐘日期

4 改變記憶體映射圖

7. 請指出個人電腦作業系統與大型主機作業系統的不同之處。

8. 一個MINIX檔的屬主uid為12，gid為1，該檔的許可權保護碼為rwxr-x---，另一用戶的uid為6，gid為1，他試圖執行該檔，結果如何？

9. 在實際應用中，系統中僅存在一個超級用戶將導致許多安全問題，為什麼？

10. 客戶－伺服器模型在分散式系統中很流行，它能夠用於單機系統嗎？

11. 在分時系統中為什麼需要進程表？在個人電腦系統中只存在一個進程，它完全控制整台電腦的使用，在這樣的系統中是否還需要進程表？

12. 塊設備檔和字元設備檔的本質區別是什麼？

13. 在MINIX系統中，使用者2對用戶1所屬的一個檔建立了一個連結，隨後使用者1刪除了此檔，此時用戶2訪問該檔，其結果如何？

14. 為何CHROOT系統調用僅限於超級使用者使用？（提示：資訊保護問題）

15. MINIX中為什麼設立一個update進程在系統中一直運行？

16. 在什麼情況下忽略SIGALRM信號是有意義的？

17. 寫一個（組）程式，嘗試使用MINIX的所有系統調用。對每一條系統調用，嘗試使用不同的參數，包括錯誤參數，驗證這些錯誤參數是否被檢測出來。

18. 寫一個類似與圖1－10的shell，要求其中包括足夠多的代碼以便能夠對其進行測試。可以加入一些特性如輸入，輸出的重定向，管道和後臺進程等。

**第二章 進 程**

我們現在開始深入地研究作業系統（特別是MINIX）是如何設計和構造的。 作業系統中最核心的概念是進程：一個對正在運行的程式的抽象。作業系統的其他所有內容都圍饒著進程，所以作業系統的設計者（及學生）盡可能早地理解進程是很重要的。

**2.1 進程介紹**

所有現代的電腦都能同時做幾件事情。當一個使用者程式正在運行時，電腦還能夠同時讀盤，並向螢幕或印表機輸出正文。 在一個多道程序系統中，CPU由這道程式向那道程式切換，使每道程式運行幾十或幾百毫秒。 然而嚴格地說，在一個瞬間，CPU只能運行一道程式。在1秒鐘期間，它可能運行多道程序，這樣就給用戶一種並行的錯覺。有時人們所說的偽並行就是指CPU在多道程序之間快速地切換，以此來區分它與多處理機（兩個或更多的CPU共用物理記憶體）系統真正的硬體並行。人們很難對多個並行的活動進行跟蹤。因此，經過多年的努力，作業系統的設計者發展了一種模型（順序進程），使得並行更容易處理。該模型及其使用正是本章的主題。

**2.1.1 進程模型**

在該模型中，電腦上所有可運行的軟體，通常包括作業系統，被組織成若干順序進程，簡稱進程（processes）。一個進程就是一個正在執行的程式，包括程式計數器、寄存器和變數的當前值。從概念上說，每個進程擁有它自己的虛擬CPU。 當然，實際上真正的CPU在各進程之間來回切換。但為了理解這種系統，考慮在（偽）並行情況下運行的進程集，要比我們試圖跟蹤CPU如何在程式間來回切換簡單得多。這種快速的切換稱作多道程序，正如在上一章所看到的。

在圖2-1（a）中，我們看到在一個多道程序電腦的記憶體中有四道程式。在圖2-1（b）中，我們看到四個進程各自擁有自己的控制流程（即自己的程式計數器），並且每個都獨立地運行。在圖2-1（C）中，我們看到在觀察一段足夠長的時間後，所有的進程都有所進展。但在一個給定的瞬間僅有一個進程真正在運行。

圖2-1 （a）多道程序中的四道程式。 （b）四個獨立、順序進程的概念模型。 （c）在任意時刻僅有一個程式活躍。

由於CPU在各進程之間來回切換，每個進程執行運算的速度是不確定的，而且當同一進程再次運行其運算速度通常也不可再現。所以，進程的程式設計絕不能對時序作任何固定的假設。例如考慮一個I/O進程用流式磁帶機恢復被備份的檔，它執行一個10000次的空迴圈以等待磁帶機達到正常速度，然後發出命令讀取第一個記錄。如果CPU決定在空迴圈期間將處理機調度給其他進程，則磁帶機進程可能在第一條記錄通過磁頭之後還未被再次調度。 當一個進程具有此類嚴格的即時要求時，也就是一些特定事件一定要在所指定的若干毫秒中發生，那麼必須採取特殊措施來保證它們一定在這段時間中發生。然而，通常大多數進程並不受CPU多道程序或其他進程相對速度的影響。

進程和程式之間的區別是很微妙的，但卻非常重要。一個類比可以使我們更容易理解這一點。想像一位元有一手好廚藝的電腦科學家正在為他的女兒烘制生日蛋糕。他有做生日蛋糕的食譜，廚房裡有所需的原料：麵粉、雞蛋、糖、香草汁等等。在這個比喻中，做蛋糕的食譜就是程式（即用適當形式描述的演算法），電腦科學家就是處理機（CPU），而做蛋糕的各種原料就是輸入資料。進程就是廚師閱讀食譜、取來各種原料、以及烘制蛋糕的一系列動作的總和。

現在假設電腦科學家的兒子哭著跑了進來，說他被一隻蜜蜂螫了。電腦科學家就記錄下他照著食譜做到哪兒了（保存進程的當前狀態），然後拿出一本急救手冊，按照其中的指示處理螫傷。這裡，我們看到處理機從一個進程（做蛋糕）切換到另一個高優先順序的進程（實施醫療救治），每個（進程）擁有各自的程式（食譜和急救書）。當蜜蜂螫傷處理完之後，電腦科學家又回來做蛋糕，從他離開時的那一步繼續做下去。

這裡的關鍵思想是：一個進程是某種類型的一個活動，它有程式、輸入、輸出、及狀態。單個處理機被若干進程共用，它使用某種調度演算法決定何時停止一個進程的工作，並轉而為另一個進程提供服務。

進程的層次結構

支援進程概念的作業系統必須提供某種途徑來創建所需要的進程。在一些非常簡單的系統，或那種設計為僅有一個應用運行的系統（例如，即時地控制一個設備）中，可能在系統啟動時，以後所需要的所有進程都已存在。然而在多數系統中，需要有某種方法以便按需創建或撤銷進程。在MINIX系統中，進程通過調用FORK系統調用來創建進程，它將創建一個與調用進程相同的進程。子進程同樣也能執行FORK，所以有可能形成一棵完整的進程樹。在其他作業系統中，也具有若干系統調用，用來創建一個進程、裝入它的記憶體、並使其開始運行。不管系統調用的具體形式如何，系統都需為進程提供一種方法，使其能夠創建其他進程。注意，每個進程只有一個父進程，但可以有O個、1個、2個或更多個子進程。

作為進程樹如何使用的一個簡單例子，讓我們來看MINIX在啟動時是怎樣對其自己進行初始化的。在MINIX的引導映射中有一個稱為init的特殊進程，當它開始運行時，它讀取一個記錄了存在多少個終端的檔，然後為每個終端創建一個新的進程，這些進程等待使用者進行登錄。如果登錄成功，登錄進程執行一個shell程式來接受命令。這些命令可能啟動更多的進程，依次類推。這樣，系統中的所有進程都屬於一棵進程樹，而init進程則是進程樹的根（init 和shell的代碼未列在本書中，它們可從其他地方取到）。

進程的狀態

儘管每個進程是一個獨立的實體，有它自己的程式計數器和內部狀態，但進程之間經常需要交互作用。一個進程的輸出結果可能作為另一個進程的輸入。在shell命令

cat chapter1 chapter2 chapter3 | grep tree

中，第一個進程運行cat，將三個檔連接並輸出。第二個進程運行grep，它從輸入中選擇所有包含單詞“tree”的那些行。根據這兩個進程的相對速度（這取決於這兩個程式的相對複雜度和各自所分配到的CPU時間），可能發生“grep”準備就緒可以運行，但輸入還沒有到這種情況。於是它就必須被阻塞直到輸入到來。

當一個進程在邏輯上不能繼續運行時，它就阻塞，典型的例子是它在等待可以使用的輸入。還可能有這樣的情況，一個概念上能夠運行的進程被迫停止，其原因是作業系統調度另一個進程佔用處理機。這兩種條件是完全不同的。第一種情況下，掛起是程式自身所固有的（在用戶命令列被鍵入之前，你無法執行它）；第二種情況則是由系統引起的（沒有足夠的CPU，所以不能使每個進程都有一台它私用的處理機）。在圖2-2中我們看到進程三種狀態的轉換圖。這三種狀態是：

1 運行態（在該時刻實際佔用處理機）

2 就緒態（可運行，因為其他進程正在運行而暫時地被掛起）

3 阻塞態（除非某種外部事件發生，否則不能運行）

圖2-2 一個進程可處於運行態、阻塞態、就緒態。圖中示出各狀態之間的轉換。

前兩種狀態在邏輯上很類似。這兩種狀態下的進程都希望運行，只是在後者中，暫時沒有CPU分配給它。第三種狀態與前兩種狀態不同，該狀態的進程不能運行，即使CPU空閒也不行。

這三種狀態之間有四種可能的轉換關係。轉換1在進程發現它不能繼續運行下去時發生。在某些系統中，進程需要執行一個系統調用－BLOCK，來進入阻塞狀態。在其他系統中，包括MINIX，當一個進程從管道或設備檔（例如終端）讀取資料時，如果沒有可以使用的輸入，則進程自動被阻塞。

轉換關係2和3是由進程調度程式引起的，它是作業系統的一部分，進程甚至感知不到調度程式的存在。在系統認為運行進程佔用處理機的時間已經過長，決定讓其他進程佔用處理機時，發生轉換2。在系統已讓其他進程享有了它們應有的CPU時間而重新輪到該進程來佔用處理機時，發生轉換3。調度程式的主要內容是決定哪個進程應當運行，及它應運行多長時間。這是很重要的一點，我們將在本節的後邊對其進行討論。已經提出許多演算法，它們力圖從系統作為一個整體的角度平衡需求和效率之間的競爭，並公平地對待各進程。

當一個進程等待的一個外部事件發生時（例如一些輸入到達），發生轉換4。如果此時沒有其他進程運行，則轉換3將立即被觸發，該進程便開始運行。否則它將處於就緒態等待CPU空閒。

使用進程模型，我們就易於想像系統內部的操作狀況。一些進程運行著一些程式，這些程式執行使用者鍵入的命令，另一些進程是系統的一部分，它們的任務是處理下列一些工作：諸如執行檔服務請求、管理運行磁碟機和磁帶機的細節等。當發生一個磁片中斷時，系統作出決定，停止運行當前進程，而轉向磁片進程，該進程在此之前因等待該中斷而處於阻塞態。這樣，我們可以不再考慮中斷，而是考慮使用者進程、磁片進程、終端進程等等。這些進程在等待時總是處於阻塞狀態。所等待的事件發生時，則它們被解除阻塞，並成為可被調度運行的進程。

這個觀點即引出了圖2-3的模型。這裡最低層是作業系統的調度程式，在它上面有許多進程。所有關於中斷處理、啟動和中止進程的具體細節被隱藏在調度程式中。實際上，它是一段非常短小的程式。作業系統的其他部分被簡潔地組織成進程形式。圖2-3的模型被MINIX使用，但是其中的調度程式應不僅僅被理解為對進程的調度安排，同時也包括中斷處理和所有的進程間通信。不過，作為近似描述，它示出了其基本結構。

圖2-3 按進程組織的作業系統中最低層處理中斷和進程調度，其上是一些順序進程。

2.1.2進程的實現

為了實現進程模型。 作業系統維持著一張表格（一個結構陣列）即進程表(process table)。 每個進程佔用一個進程表項。該表項包含了進程的狀態、它的程式計數器、棧指標、記憶體分配狀況、打開檔狀態、計費和調度資訊，以及其他在進程由運行態轉到就緒態時必須保存的資訊，只有這樣才能使進程隨後被再次啟動，就象從未被中斷過一樣。

在MINIX中，進程管理、記憶體管理和檔管理是由系統中的幾個獨立模組分別處理的，所以進程表被分為幾個部分，各模組維護它們各自所需要的那些域。圖2-4示出了一些重要的域。與本章有關的域位於第一列，給出其他兩列僅僅是為了建立一點概念，即系統的其他部分需要哪些資訊。

進程管理

記憶體管理

文件管理

寄存器

正文段指針

UMASK 遮罩

程式計數器

資料段指針

根目錄

程式狀態字

bss段指針

工作目錄

棧指針

退出狀態

檔描述符

進程狀態

信號狀態

有效uid

進程開始時間

處理序識別碼

有效gid

使用的CPU時間

父進程

系統調用參數

子進程的CPU時間

進程組

各種標誌位元

下次報警時間

真實uid

訊息佇列指針

有效uid

掛起的信號位元

真實gid

處理序識別碼

有效gid

各種標誌位元

信號點陣圖

各種標誌位元

圖2-4 MINIX進程表的某些域。

在瞭解進程表之後，就可以對一台有單個CPU、多個I/O設備的電腦如何維持多個順序進程的假像作更多的解釋。接下來的內容從技術上說是對圖2-3中MINIX調度程式如何工作的一個描述，但多數現代的作業系統從本質上來說都差不多。與每類I/O設備（例如軟碟、硬碟、計時器、終端）相關的都有一個靠近記憶體底部的位置，稱作中斷向量。它包含中斷服務程式的入口位址。假設當一個磁片中斷發生時，使用者進程3正在運行，則中斷硬體將程式計數器、程式狀態字、可能還有一個或多個寄存器壓入（當前）堆疊，電腦隨即跳轉到磁片中斷向量所指的地址處。這是硬體做的操作，從這裡開始，軟體就接管了一切。

中斷服務程式的工作從把當前進程全部寄存器值存入進程表項開始。當前進程號及一個指向其表項的指標被保存在全域變數中以便能夠快速地找到它們。隨後將中斷存入的那部分資訊從堆疊中刪除，並將棧指標指向一個被進程管理者所使用的臨時堆疊。一些動作，諸如保存寄存器值和設置棧指標等無法用C語言描述，所以由一個短小的組合語言常式來完成。當該常式結束後，它調用一個C過程來完成剩下的工作。

MINIX中的進程間通信通過消息完成，所以下一步是構造一條發給磁片進程的消息，這時磁片進程正在阻塞並等待該消息。這條消息通知說發生了一條中斷，以此將它和那些由使用者進程發送的消息加以區分。那些消息發出讀磁片塊之類的請求。現在磁片進程的狀態由阻塞轉換到就緒，然後，中斷服務程式調用調度程式。在MINIX中，不同的進程有不同的優先順序，以此向I/O設備服務常式提供比使用者進程更好的服務。如果當前磁片進程是優先順序最高的就緒進程，則它將被調度運行。如果被中斷進程具有與它相等或更高的優先順序， 則它將被再次調度運行，而磁片進程將只得等待一會兒。

不論哪種情況，被組合語言中斷代碼所調用的C過程現在返回，組合語言代碼為新的當前進程裝入寄存器值和記憶體映射並啟動它運行。圖2-5中總結了中斷處理和調度的過程。值得注意的是各系統在細節上略有不同。

圖2-5 當一個中斷發生後作為作業系統最低層的調度程式的工作步驟。

2.1.3執行緒

在我們剛才所討論的傳統的進程中，每個進程中只存在一條控制線索和一個程式計數器。但在有些現代作業系統中，提供了對單個進程中多條控制線索的支持。這些控制線索通常被稱為執行緒(threads)，有時也稱為輕量進程(lightweight processes)。

在圖2－6（a）中我們看到三個傳統的進程。每個進程有自己的位址空間和單一的控制線索。與此相反，在圖2－6（b）中我們看到一個進程有三條控制線索。儘管這兩種情況都有三個執行緒，但在圖2-6（a）中各執行緒在不同的位址空間中操作，而在圖2－6（b）中所有三個執行緒共用同一個位址空間。

圖2-6（a）三個進程各有一個執行緒。（b） 一個進程有三個執行緒。

作為使用多執行緒的一個例子，我們考慮一個檔案伺服器進程。它接收讀寫檔的請求並將所請求的資料送回或者接收更新了的資料。為了提高性能，伺服器在記憶體中維護一個快取記憶體，裡邊存放最近用過的檔，在需要時從該緩存中讀數據或向其中寫資料。

這種情況就非常適合圖2-6（b）中的模型。當一個請求到來時，將它遞交給一個執行緒。如果這個執行緒因等待磁片傳輸而中途阻塞，其他執行緒仍舊可以運行，這樣伺服器就可以在磁片I/O進行的同時繼續處理新的請求。圖2-6（a）的模型就不合適，因為在這裡所有的檔案伺服器執行緒共用同一塊快取記憶體非常關鍵，而圖2-6（a）中的三個執行緒並不共用相同的位址空間，所以無法訪問同一塊快取記憶體。

使用執行緒的另一個例子是WWW流覽器，例如Netscape和Mosaic。許多Web頁面都包含有多幅很小的圖像。對Web頁面上的每一幅小圖像，流覽器都必須與頁面的駐留網站建立一條單獨的連接以索取該圖像。這樣就有大量的時間浪費在了建立和釋放這些連接上。通過在流覽器內設立多個執行緒，便可以同時請求傳輸多幅圖像，在多數情況下這樣做顯著地提高了性能，因為對於小圖像，限制因素在於建立連接的時間，而不在於傳輸線的速率。

當同一位址空間中有多個執行緒時，圖2-4中的幾個域就不再針對進程，而是針對執行緒，於是就需要一張單獨的執行緒表，每個執行緒佔用一項。針對每個執行緒的資訊包括程式計數器、寄存器值及狀態。需要程式計數器是因為執行緒可以象進程一樣被掛起和恢復運行。需要寄存器值是因為當執行緒被掛起時，它的寄存器值必須被保存下來。最後，執行緒象進程一樣，可處於運行，就緒或阻塞態。

在有些系統中，作業系統感知不到執行緒的存在，換言之，執行緒完全在用戶空間進行管理。例如，當一個執行緒將被阻塞時，它在停止之前選擇並啟動它的後繼執行緒。目前有幾個用戶級的執行緒套裝軟體用得很普遍，包括POSIX的P－執行緒和MACH的C－執行緒套裝軟體。

在另外一些系統中，作業系統知道每個進程中存在多個執行緒，所以當一個執行緒阻塞時，作業系統會選擇下一個運行的執行緒，它可能來自同一個進程，或者其他進程。為了進行調度，核心必須有一張執行緒表，其中列出了系統中所有的執行緒，這與進程表很類似。

儘管這兩個種選擇看起來是等價的，但它們的性能相差甚遠。在使用者空間管理的執行緒其切換速度比需要核心調用的情況快得多。這一事實有力地支援將執行緒管理放在用戶空間。而另一方面，當執行緒完全在用戶空間管理時，若一個執行緒阻塞（例如，等待I/O或處理頁面故障），則核心將整個進程阻塞，因為它甚至不知道其他執行緒的存在。這一事實又有力地支持將執行緒放在核心進行管理。最後的結果是兩種系統都被使用，同時還提出了各種混合方案（Anderson et al., 1992）

不論執行緒是放在核心還是用戶空間，它們都帶來了一大堆必須解決的問題，而且這些問題相當程度上改變了程式設計模型。首先來考慮對FORK系統調用的影響。如果父進程有多個執行緒，那麼子進程是否也應該有這些執行緒？如果不是，那麼它可能無法正常工作，因為可能所有的執行緒都是必不可少的。

然而，如果子進程獲得了與父進程一樣多的執行緒，那麼當一個執行緒阻塞在一條READ調用時，比如鍵盤，這時是否兩個執行緒都阻塞在鍵盤上？當鍵入一行內容時，是否兩個執行緒都得到一份拷貝？還是只有父進程得到？還是只有子進程得到？對於打開網路連接也存在同樣的問題。

另一類問題與多執行緒共用許多資料結構有關。若一個執行緒關閉一個檔而另一個執行緒正在讀該檔，將有什麼後果？假設一個執行緒注意到記憶體不夠並開始申請更多的記憶體，但此時發生執行緒切換，新運行的執行緒也注意到這個問題並再次申請記憶體。那麼這裡是申請一次呢，還是兩次？在幾乎所有設計時未考慮執行緒的系統中，其庫常式（例如申請記憶體的常式）都不可重入，如果前一個調用尚在啟動時進行第二次調用，則必然會引起崩潰。

另一個問題就是錯誤報告。在UNIX中，一條系統調用執行完之後，其狀態將放在一個全域變數errno。如果一個執行緒執行系統調用，在它讀取errno之前，另一個執行緒也執行一條系統調用並清除了errno原先的值，則情況會怎樣？

下面考慮信號。有些信號在邏輯上是針對執行緒的，另一些則不是。例如，當一個執行緒調用ALARM，則將產生的信號傳遞給執行調用的執行緒是很合理的。當核心能夠感知到執行緒時，通常它可以保證由正確的執行緒獲得該信號。當核心感知不到執行緒時，執行緒套裝軟體必須以某種方式跟蹤鬧鐘信號。對於用戶級執行緒還有另一個麻煩：一個進程（例如在UNIX中）某一時刻只能定一個時間鬧鐘，而若干執行緒各自獨立地調用ALARM，則將發生混亂。

其他信號，例如鍵盤中斷，不是特定於執行緒的。那麼應該由誰來捕捉它們？是一個專門的執行緒？所有的執行緒？還是一個新創建的執行緒？這些方法都存在問題。進一步地，如果一個執行緒修改了信號處理常式而未通知其他執行緒，將發生什麼情況？

執行緒引起的最後一個問題是堆疊管理。在許多系統中，當堆疊發生溢出時，核心只是自動地提供更多的堆疊空間。當一個進程有多個執行緒時，它必須也有多個堆疊。如果核心感知不到所有這些堆疊，則發生堆疊故障時它不能自動地將其擴展，實際上，它可能甚至意識不到記憶體故障與堆疊增長有關。

這些問題當然並不是不法克服，但它們確實表明僅僅向一個現有系統中引入執行緒而不進行徹底的重新設計是根本不行的。起碼系統調用的語義要重新定義，庫常式也要重寫。而且所有這些改變必須保持向後相容，以保證現存的只有一個執行緒的進程的可用性。關於執行緒的其他資訊，請參閱（Hauser et al., 1993和Marsh et al., 1991）。

2.2進程間通信

進程經常需要與其他進程通信。例如，在一個shell管道中，第一個進程的輸出必須傳送到第二個進程，這樣沿著管道傳遞下去。因此在需要通信的進程之間，最好使用一種結構較好的方式，而不要使用中斷。在隨後幾節中，我們就來看進程間通信(IPC)問題。

簡單地說，這裡有三方面內容。第一方面已經在上面提到過：一個進程如何向另一個進程傳送資訊。第二方面必須要保證兩個或多個進程在涉及臨界活動時不會彼此影響（設想兩個進程都試圖攫取最後100K記憶體的情況）。協力廠商面涉及存在依賴關係時進行適當的定序：如果進程A產生資料，進程B列印資料。則B在開始列印之前必須等到A產生了一些資料為止。從下一節開始我們將討論這三個問題。

2.2.1競爭條件

在有些作業系統中，協作進程可能共用一些彼此都能夠讀寫的公用存儲區。它可能在記憶體中，也可能是一個共用檔；這裡共用記憶體的位置並不影響通信的本質及其帶來的問題。為了理解實際中進程間通信如何工作，我們考慮一個簡單但很普遍的例子，一個周邊同作列印程式。當一個進程需要列印檔案時，它將檔案名放在一個特殊的印表機周邊同作系統（Spooler）目錄下。另一個印表機精靈進程週期性地檢查否有檔需要列印，若有它就列印之並將該檔案名從Spooler目錄下刪掉。

設想我們的Spooler目錄有許多（潛在地有無限多個）槽，編號依次為0，1，2 .....，每個槽存放一個檔案名。同時設有兩個共用變數，out，指明下一個被列印的檔。in，指向目錄中下一個空閒的槽。這兩個變數可以被保存在一個所有進程都可訪問的兩個字的檔中。某一時刻，0到3號槽空閒（其中的檔已經被列印完畢），4到6號槽被佔用（其中存有待列印的檔案名）。幾乎在同一時刻，進程A和進程B都決定將一個檔排隊列印，這種情況示於圖2-7。

圖2-7 兩個進程試圖在同一時刻訪問共用記憶體。

在Murphy法則1生效時，可能發生以下的情況。進程A讀到in的值為7，將它存在一個區域變數next\_free\_slot中。此時發生一次時鐘中斷，CPU認為進程A已運行了足夠長的時間，決定切換到進程B。進程B也讀取in，同樣得到值為7，於是它將要列印的檔案名存入7號槽，並將in的值更新為8。然後它繼續執行其他操作。

最後進程A接著從上次中止的地方再次運行，它檢查變數next\_free\_slot，發現其值為7，於是將列印檔案名存入7號槽，這將把進程B存在那裡的檔案名刪掉。然後它將next\_free\_slot加1，得到值為8，就將8存到變數in中。此時Spooler目錄內部是一致的，所以印表機精靈進程將發現不了任何錯誤，但進程B卻永遠得不到任何列印輸出。類似這樣的情況，兩個或多個進程讀寫某些共用資料，而最後的結果取決於進程運行的精確時序，就稱為競爭條件(race conditions)。調試包含有競爭條件的程式是一件很頭痛的事。大多數的運行結果都很好，但在極少數情況下發生一些無法解釋的奇怪的事情。

2.2.2臨界區

如何避免競爭條件?實際上凡牽涉到共用記憶體、共用檔、以及共用任何資源的情況都會引發與前邊類似的錯誤。要避免這種錯誤，關鍵是要找到某種途徑來阻止多於一個的進程同時讀寫共用的資料。換言之，我們需要的是互斥(mutual exclusion) － 即某種手段以確保當一個進程在使用一個共用變數或檔時，其他進程不能做同樣的操作。前述問題的癥結就在於，在進程A對共用變數的使用未結束之前進程B就使用它。為實現互斥而選擇適當的原語是任何作業系統的主要設計內容之一。也是我們後續章節中要詳細加以討論的。

避免競爭條件的問題也可以用一種抽象的方式進行描述。一個進程的一部分時間做內部計算或其他一些不會引發競爭條件的操作。在某些時候進程可能會訪問共用記憶體或共用檔，或執行其他一些會導致競爭的操作。我們把對共用記憶體進行訪問的程式片段稱作臨界區(critical region)，或臨界段(critical section)。如果我們能夠適當地安排使得兩個進程不可能同時處於臨界區，則就能夠避免競爭條件。

儘管這樣的要求防止了競爭條件，但它還不能保證使用共用資料的併發進程能夠正確和高效地進行操作。對於一個好的解決方案，我們需要以下4個條件：

1 任何兩個進程不能同時處於臨界區。

2 不應對CPU的速度和數目作任何假設。

3 臨界區外的進程不得阻塞其他進程。

4 不得使進程在臨界區外無休止地等待。

2.2.3忙等待的互斥

本節將看幾種實現互斥的方案。在這些方案中，當一個進程在臨界區中更新共用記憶體時，其他進程將不會進入其臨界區，也不會帶來任何麻煩。

關中斷

這種最簡單的方法是使每個進程在進入臨界區後先關中斷，在離開之前再開中斷。中斷被關掉後，時鐘中斷也被遮罩。CPU只有在發生時鐘或其他中斷時才會進行進程切換，這樣關中斷之後CPU將不會被切換到其他進程。於是，一旦進程關中斷之後，它就可以檢查和修改共用記憶體，而不必擔心其他進程的介入。

這種方案不好，因為把關中斷的權力交給使用者進程是不明智的。設想一下若一個進程關中斷之後不再開中斷，其結果將會如何？系統可能會因此終止。而且，如系統有兩個或多個共用記憶體的處理器，則關中斷僅僅對執行本指令的CPU有效，其他CPU仍將繼續運行，並可以訪問共用記憶體。

另一方面，對核心來說，當它在更新變數或清單的幾條指令期間將中斷關掉是很方便的。因為當諸如就緒進程佇列之類的資料狀態不一致時發生中斷，則將導致競爭條件。所以得出結論：關中斷對於作業系統本身是一項很有用的技術，但對於使用者進程則不是一種合適的通用互斥機制。

鎖變數

作為第二種嘗試，我們來尋找一種軟體解決方案。設想有一個共用（鎖）變數，初值為0。當一個進程想進入其臨界區時，它首先測試這把鎖。如果鎖的值為0，則進程將其置為1並進入臨界區。若鎖已經為1，則進程將等待直到其變成0。於是，0就表示臨界區內沒有進程，1表示已經有某個進程進入了臨界區。

不幸的是，這種想法也含有與Spooler目錄一樣的紕漏。假設一個進程讀鎖變數的值並發現它為0，而恰好在它將其置為1之前，另一個進程被調度運行，將鎖變數置為1。當第一個進程再次能運行時，它同樣也將鎖置為1，則此時同時有兩個進程處於臨界區中。

可能你會想,先讀取鎖變數,緊挨著在改變其值之前再檢查一遍它的值，這樣便可以解決問題。但這實際上無濟於事。如果第二個進程恰好在第一個進程完成第二次檢查之後修改鎖變數，則同樣還會發生競爭條件。

嚴格地輪換法

互斥的第三種方法示於圖2-8。與本書中幾乎所有其他程式一樣，這裡的程式段用C編寫。之所以選擇C是由於實際的作業系統普遍用C編寫（偶而有用C++），而基本上不用象Modula2或Pascal那樣的語言。

while（TRUE）{ while（TRUE）{

while（turn!=0） /\*等待\*/; while（turn!=1） /\* 等待 \*/; critical\_region（）; critical\_region（）;

turn=1; turn=0;

noncritical\_region（）; noncritical\_region（）;

} }

（a） （b）

圖2-8 臨界區問題的一種解法。

在圖2-8中，整型變數turn初值為0，它用於跟蹤輪到哪個進程進入臨界區來檢查或更新共用記憶體。一開始進程0檢查turn，發現它是0，於是進入臨界區。 進程1同樣也發現它是0，於是執行一個等待迴圈不停地檢測它是否變成了1。 持續地檢測一個變數直到它具有某一特定值就稱作忙等待(busy waiting)。忙等待是應該避免的，因為它浪費CPU時間。只有在有理由預期等待時間很短時才使用忙等待。

當進程0離開臨界區時，它將turn置為1，以允許進程1進入其臨界區。假設進程1很快便離開了臨界區，則這時兩個進程都處於臨界區之外，turn的值被置為0。現在進程0很快便執行完了其整個迴圈，它再次執行到非臨界區的部分，並將turn置為1。此時，進程0結束了其非臨界區的操作並回到迴圈的開始，但很不幸，這時它不能進入臨界區，因為turn的值為1，而進程1還在忙於非臨界區的操作。這說明，輪流進入臨界區在一個進程比另一個慢很多的情況下並不是一個好辦法。

這種情況違反了以上條件3：進程0被一個臨界區之外的進程阻塞。再回到Spooler目錄的問題，如果我們現在將臨界區與讀寫Spooler目錄相聯繫，則進程0有可能因為進程1在做其他事情而被禁止列印另一個檔。

實際上，該方案要求兩個進程嚴格地輪流進入它們的臨界區。例如若用周邊同作方式列印檔案，那麼任何一個進程都不可能在一輪中列印兩個檔。儘管該演算法的確避免了所有的競爭，但由於它違反了條件3，所以不能作為一個很好的備選方案。

Peterson解決方案

一個荷蘭數學家T.Dekker將輪換法和鎖變數及警告變數的思想相結合，最早提出了一個不需要嚴格輪換的軟體互斥解法。關於Dekker的演算法，請參閱（Dijkstra，1965）。

在1981年，G.L.Peterson發現了一種簡單得多的互斥演算法。這使得Dekker的方法不再有任何新意。Peterson的演算法示於圖2-9。該演算法由兩個用ANSI C寫的過程組成。ANSI C要求為所定義和使用的所有函數提供函數原型，但為了節省篇幅，在本例和後邊的例子中我們將不給出函數原型。

#define FALSE 0

#define TRUE 1

#defiine N 2 /\*進程數\*/

int turn; /\*輪到誰了?\*/

int interested[N]; /\*所有值初始為0（FALSE）\*/

void enter\_region（int process） /\*進程號為0或1\*/

{

int other; /\*另一個進程的進程號\*/

other=1-process; /\*另一個進程\*/

interested[process]=TRUE; /\*標識出希望進入臨界區\*/

turn=process; /\*設置標誌位元\*/

while（turn==process&&interested[other]==TRUE）; /\*空語句\*/

}

void leave\_region（int process） /\*process：即將離開臨界區的進程\*/

{

interested[process]=FALSE; /\*標識將離開臨界區\*/

}

圖2-9 完成互斥的Peterson方案。

在使用共用變數（即進入其臨界區）之前，各進程使用其進程號0或1作為參數來調用enter\_region，該調用在需要時將使進程等待，直到能安全地進入。進程在完成對共用變數的操作之後，將調用leave\_region，表示操作已完成，若其他進程希望進入臨界區，則現在可以進入。

我們來看這個方案如何工作。起初沒有任何進程處於臨界區，現在進程0調用enter\_region，它通過將其陣列元素置位元和將turn置為0來標識它希望進入臨界區。由於進程1並不想進入臨界區，所以enter\_region很快便返回。如果進程1現在調用enter\_region，它將在此處掛起直到interested[0]變成FALSE，該事件只有當進程0調用leave\_region退出臨界區時才會發生。

現在考慮兩個進程幾乎同時調用enter\_region的情況。它們都將自己的進程號存入turn。但只有後一個被保存進去的進程號才有效，前一個是無效的。假設進程1後存，則turn為1。當兩個進程都運行到while語句時，進程0將迴圈0次並進入臨界區，而進程1將不停地迴圈，並不得進入臨界區。

TSL指令

現在來看一種需要硬體支援的方案。許多電腦，特別是那些為多處理機設計的電腦，都有一條指令叫做測試並上鎖（TSL）。其工作如下所述：它將一個記憶體字讀到一個寄存器中，然後在該記憶體位址上存一個非零值。讀數和寫數操作保證是不可分割的－即該指令結束之前其他處理機均不允許訪問該記憶體字。執行TSL指令的CPU將鎖住記憶體匯流排以禁止其他CPU在本指令結束之前訪問記憶體。

為了使用TSL指令，我們要使用一個共用變數lock來協調對共用記憶體的訪問。當lock為0時，任何進程都可以使用TSL指令將其置為1並讀寫共用記憶體。當操作結束時，進程用一條普通的MOVE指令將lock重新置為0。

這條指令如何被用來防止兩個進程同時進入臨界區呢？解決方案示於圖2-10中。其中示出了使用4條指令的組合語言常式。第一條指令將lock原來的值拷貝到寄存器中並將lock置為1，隨後這個原先的值與0相比較。如果它非零，則說明先前已被上鎖，則程式將回到開頭並再次測試。經過或長或短的一段時間後它將變成0（當前處於臨界區中的進程退出臨界區時），於是子常式返回，並上鎖。清除這個鎖很簡單，程式只需將0存入lock即可，不需要特殊的指令。

enter\_region：

tsl register，lock |複製lock到寄存器，並將lock置為1

cmp register，#0 | lock等於0嗎?

jne enter\_region |如果不等於0，已上鎖，再次迴圈

ret |返回調用程式，進入臨界區

leave\_region：

move lock ， #0 |置lock為0

ret |返回調用程式

圖2-10 用TSL指令上鎖和清除鎖。

現在就有一種很明確的解法了。進程在進入臨界區之前先調用enter\_region。這將導致忙等待，直到鎖空閒為止。隨後它獲得鎖變數並返回。在進程從臨界區返回時它調用leave\_region，這將把lock置為0。與臨界區問題的所有解法一樣，進程必須在正確的時間調用enter\_region和leave\_region，解法才能奏效。如果一個進程有欺詐行為，則互斥將會失敗。

2.2.4 睡眠和喚醒

Peterson解法和TSL解法都是正確的，但它們都有忙等待的缺點。這些解法在本質上是這樣的：當一個進程想進入臨界區時，先檢查是否允許進入，若不允許，則進程將踏步等待，直到許可為止。

這種方法不僅浪費CPU時間，還可能引起預料不到的結果。考慮一台電腦有兩個進程，H優先順序較高，L優先順序較低。調度規則規定只要H處於就緒態它就可以運行。在某一時刻，L處於臨界區中，此時H變到就緒態準備運行（例如，一條I/O操作結束）。現在H開始忙等待，但由於當H就緒時L不會被調度，也就無法離開臨界區，所以H將永遠忙等待下去。這種情況有時被稱作優先順序翻轉問題(priority inversion problem)。

現在讓我們來看幾條進程間通信原語，它們在無法進入臨界區時將阻塞，而不是忙等待。最簡單的是睡眠（SLEEP）和喚醒（WAKEUP）。SLEEP系統調用將引起調用進程阻塞，即被掛起，直到另一進程將其喚醒。WAKEUP調用有一個參數，即要被喚醒的進程。另一種方法是SLEEP與WAKEUP各有一個參數，即一個用於匹配SLEEP和WAKEUP的記憶體位址。

生產者 － 消費者問題

作為如何使用這些原語的一個例子，我們考慮生產者－消費者問題（也稱作有界緩衝區問題）。兩個進程共用一個公共的固定大小的緩衝區。其中的一個，生產者，將資訊放入緩衝區；另一個，消費者，從緩衝區中取出資訊（該問題也可被推廣到m個生產者，n個消費者的情況，但出於簡單起見，我們只考慮一個生產者，一個消費者的情況）。

麻煩之處在於當緩衝區已滿，而此時生產者還想向其中放入一個新的資料項目的情況。解決辦法是讓生產者睡眠，待消費者從緩衝區中取走一個或多個資料項目時再喚醒它。同樣地，當消費者試圖從緩衝區中取資料而發現緩衝區為空時，它就睡眠，直到生產者向其中放入一些資料時再將其喚醒。

這種方法聽起來很簡單，但它包含與前邊Spooler目錄問題一樣的競爭條件。為了跟蹤緩衝區中的資料項目數，我們需要一個變數count。如果緩衝區最多存放N個資料項目，則生產者代碼將首先檢查count是否達到N，若是，則生產者睡眠；否則生產者向緩衝區中放入一個資料項目並遞增count的值。

消費者的代碼與此類似：首先看count是否為0，若是則睡眠；否則從中取走一個資料項目並遞減count的值。每個進程同時也檢測另一個是否應睡眠，若不應睡眠則喚醒之。生產者和消費者的代碼示於圖2-11。

為了在C語言中表示諸如SLEEP和WAKEUP的系統調用，我們將用對庫函數的調用形式來表示。儘管它們不是標準C庫的一部分，但在實際上具有這些系統調用的所有系統中都應具有這兩種庫函數。圖中未示出的過程enter\_item和remove\_item用來記錄將資料項目放入緩衝區和從緩衝區中取出資料項目。

# define N 100 /\* 緩衝區內的槽數 \*/

int count =0; /\* 緩衝區內資料項目個數 \*/

void producer（void）

{

while（TRUE） { /\* 無限迴圈 \*/

producer\_item（）; /\* 產生下一個資料項目 \*/

if（count==N）sleep（）; /\* 緩衝區滿，進入睡眠 \*/

enter\_item（）; /\* 將一個資料項目放入緩衝區 \*/

count=count+1; /\* 增加緩衝區內資料項目個數 \*/

if（count==1）wakeup（consumer） /\* 緩衝區為空？\*/

}

}

void consumer（void）

{

while（TRUE）{ /\* 無限迴圈 \*/

if（count==0）sleep（）; /\* 緩衝區空，進入睡眠 \*/

remove\_item（）; /\* 從緩衝區中取走一個資料項目 \*/

count=count-1; /\* 遞減緩衝區中資料項目個數 \*/

if（count==N-1）wakeup（producer） /\* 緩衝區滿？\*/

consume\_item（）; /\* 列印資料項目 \*/

}

}

圖2-11 含有競爭條件的生產者－消費者問題。

現在回到競爭條件的問題。這裡有可能會出現競爭條件，其原因是對count的訪問未加限制。有可能出現以下情況：緩衝區為空，消費者剛剛讀取count的值發現它為0。此時調度程式決定暫停消費者並啟動運行生產者。生產者向緩衝區中加入一個資料項目，將count加1。現在count的值變成了1。它推斷認為由於count剛才為0，所以消費者此時很可能在睡眠，於是生產者調用wakeup來喚醒消費者。

不幸的是，消費者此時在邏輯上並未睡眠，所以喚醒信號丟失。當消費者下次運行時，它將測試先前讀到的count值，發現它為0，於是去睡眠。這樣生產者遲早會添滿整個緩衝區，然後睡眠。這樣一來兩個進程都將永遠睡眠下去。

這裡問題的實質在於發給一個（尚）未睡眠進程的喚醒信號丟失了。如果它沒有丟失，則一切都很正常。一種快速的彌補方法是修改規則，加上一個喚醒等待位(wakeup waiting bit)。當向一個清醒的進程發送一個喚醒信號時，將該位置位。隨後，當進程要睡眠時，如果喚醒等待位為1，則將該位元清除，而進程仍然保持清醒。

儘管在本例中喚醒等待位解決了問題，但很容易就可以構造出一些例子，其中有兩個或更多的進程，這時一個喚醒等待位就不敷使用。我們可以再打一個補丁，加入第二個喚醒等待位，或者甚至是8個、32個，但原則上講這並未解決問題。

2.2.5 信號量

信號量是E. W. Dijkstra在1965年提出的一種方法，它使用一個整型變數來累記喚醒次數，以供以後使用。在他的建議中引入一個新的變數類型，稱作信號量(semaphore)。一個信號量的值可以為0，表示沒有積累下來的喚醒操作；或者為正值，表示有一個或多個被積累下來的喚醒操作。

Dijkstra建議設兩種操作：DOWN和UP（分別為推廣後的SLEEP和WAKEUP）。對一信號量執行DOWN操作是檢查其值是否大於0。若是則將其值減1（即，用掉一個保存的喚醒信號）並繼續。若值為0，則進程將睡眠，而且此時DOWN操作並未結束。檢查數值、改變數值、以及可能發生的睡眠操作均作為一個單一的、不可分割的原子操作(atomic action)完成。即保證一旦一個信號量操作開始，則在操作完成或阻塞之前別的進程均不允許訪問該信號量。這種原子性對於解決同步問題和避免競爭條件是非常重要的。

UP操作遞增信號量的值。如果一個或多個進程在該信號量上睡眠，無法完成一個先前的DOWN操作，則由系統選擇其中的一個（例如，隨機挑選）並允許其完成它的DOWN操作。於是，對一個有進程在其上睡眠的信號量執行一次UP操作之後，該信號量的值仍舊是0，但在其上睡眠的進程卻少了一個。遞增信號量的值和喚醒一個進程同樣也是不可分割的。不會有進程因執行UP而阻塞，正如在前面的模型中不會有進程因執行wakeup而阻塞一樣

在Dijkstra最早的論文中。他使用P和V而不是DOWN和UP，但因為對於不講荷蘭語的讀者來說採用什麼記號並無大的干係，所以我們將使用DOWN和UP。它們在Algol 68中首次被引入。

用信號量解決生產者 － 消費者問題

圖2－12中示出了用信號量解決丟失的喚醒問題。最重要的是它採用一種不可分割的方式來實現。通常是將UP和DOWN作為系統調用實現，而且作業系統只需在執行以下操作時短暫地關掉中斷，這些操作包括：檢測信號量、修改信號量、以及在需要時使進程睡眠。由於這些動作只需要幾條指令，所以關中斷不會帶來什麼副作用。如果使用多個CPU，則每個信號量應由一個鎖變數進行保護。通過TSL指令來確保同一時刻只有一個CPU在對信號量進行操作。讀者必須搞清楚使用TSL來防止幾個CPU同時訪問信號量，與生產者或消費者使用忙等待來等待對方騰出或填充緩衝區是完全不同的。信號量操作僅需幾個微秒，而生產者或消費者則可能需要任意長的時間。

#define N 100 /\* 緩衝區內槽數 \*/

typedef int semaphone; /\* 信號量是一種特殊的整形變數 \*/

semaphone mutex =1; /\* 控制對臨界區的訪問 \*/

semaphone empty=N; /\* 記錄緩衝區內空的槽數 \*/

semaphone full=0; /\* 記錄緩衝區內滿的槽數 \*/

void producer（void）

{

int item;

while（TRUE）{ /\* TRUE為常數1 \*/

produce\_item（&item）; /\* 產生一個需放入緩衝區的資料項目 \*/

down（&empty）; /\* 遞減空槽數 \*/

down（&mutex）; /\* 進入臨界區 \*/

enter\_item（item）; /\* 將一個新資料項目放入緩衝區 \*/

up（&mutex）; /\* 離開臨界區 \*/

up（&full）; /\* 遞增滿槽數 \*/

}

}

void consumer（void）

{

int item;

while（TRUE）{ /\* 無限迴圈 \*/

down（&full）; /\* 遞減滿槽數 \*/

down（&mutex）; /\* 進入臨界區 \*/

remove\_item（&item）; /\* 從緩衝區中取走一個資料項目 \*/

up（&mutex）; /\* 離開臨界區 \*/

up（&empty）; /\* 遞增空槽數 \*/

consume\_item（item）; /\* 對資料項目進行操作 \*/

}

}

圖2-12使用信號量的生產者--消費者問題。

該解決方案使用了三個信號量：full用來記錄滿的緩衝槽數目，empty記錄空的緩衝槽總數，mutex用來確保生產者和消費者不會同時訪問緩衝區。full的初值為0，empty的初值為緩衝區內槽的數目，mutex初值為1。兩個或多個進程使用的初值為1的信號量保證同時只有一個進程可以進入臨界區，它被稱作二進位信號量(binary semaphore)。如果每個進程在進入臨界區前都執行一個DOWN操作，並在退出時執行一個UP操作，則能夠實現互斥。

在有了一些進程間通信原語之後，我們回頭再觀察一下圖2－5中的中斷順序。在使用信號量的系統中，隱藏中斷機構的最自然的方法是為每一個I/O設備設置一個信號量，初值設為0。在啟動一個I/O設備之後，其管理進程對相關聯的信號量執行一個DOWN操作，於是立即被阻塞。當中斷到來時，中斷處理常式隨後對相關聯的信號量執行一個UP操作，這將使相關的進程再次成為就緒。該模型中，圖2－5中的第6步具體化為在設備的信號量上執行UP操作，所以在第7步中，調度程式將能夠運行設備管理程式。當然，如果這時有幾個進程就緒，則調度程式下次可以選擇一個最為重要的進程來運行。在本章後邊我們將看到如何進行調度。

在圖2－12的例子中，我們實際上通過兩種不同的方式來使用信號量。其間的區別是很重要的。信號量mutex用於互斥。它用於保證任一時刻只有一個進程讀寫緩衝區和相關的變數。互斥是避免混亂所必需的。

信號量的另一種用途是同步(synchronization)。信號量full和empty用來保證一定的事件順序發生或不發生。在本例中，它們保證當緩衝區滿的時候生產者停止運行，以及當緩衝區空的時候消費者停止運行。這種用法與互斥是不同的。

儘管信號量已經出現了三十多年，人們仍在研究它的應用。作為一個例子，請參閱（Tai 和 Carver，1996）。

2.2.6管程

有了信號量之後，進程間通信看來很容易了，對嗎？答案是否定的。仔細看圖2－12中向緩衝區放入資料項目，以及從中刪除資料項目之前的DOWN操作。假設將生產者代碼中的兩個DOWN操作交換一下次序，將使得mutex的值在empty之前被減1，而不是在其之後。如果緩衝區完全是滿的，生產者將阻塞，mutex值為0。這樣一來，當消費者下次試圖訪問緩衝區時，它將對mutex的執行一個DOWN操作，由於mutex值為0，則它也將阻塞。兩個進程都將永遠地阻塞下去，無法做如何有效的工作，這種不幸的狀況稱作鎖死(deadlock)。我們將在第三章中詳細地研究鎖死。

指出這個問題是為了說明使用信號量時要何等的小心！一處很小的錯誤將導致很大的麻煩。這就象用組合語言程式設計一樣，甚至更糟，因為這裡出現的錯誤都是競爭條件、鎖死、以及其他不可預測和不可重現的行為。

為了更易於編寫正確的程式，Hoare（1974）和Brinch Hansen（1975）提出了一種高級的同步原語，稱為管程(monitor)。他們兩人提出的方案略有不同，如下面將要描述的。一個管程是一個由過程、變數及資料結構等組成的一個集合，它們組成一個特殊的模組或套裝軟體。進程可在任何需要的時候調用管程中的過程，但它們不能在管程外的過程中直接訪問管程內的資料結構。圖2-13展示了用一種想像的類Pascal語言描述的管程。

monitor example

integer i;

condition c;

procedure producer（x）;

。

。

。

end;

prodecure consumer（x）;

。

。

。

end;

end monitor;

圖2-13 一個管程。

管程有一個很重要的特性，使它們能有效地完成互斥：任一時刻管程中只能有一個活躍進程。管程是一種程式設計語言的構件，所以編譯器知道它們很特殊，並可以採用與其他程序呼叫不同的方法來處理它們。典型的，當一個進程調用管程中的過程時，前幾條指令將檢查在管程中是否有其他的活躍進程。如果有，調用進程將掛起，直到另一個進程離開管程。如果沒有，則調用進程便進入管程。

對進入管程實現互斥由編譯器負責，但通常的做法是用一個二進位信號量。因為是由編譯器而非程式師來安排互斥，所以出錯的可能性要小得多。在任一時刻，寫管程的人無需關心編譯器是如何實現互斥的。他只需知道將所有的臨界區轉換成管程中的過程即可，而絕不會有兩個進程同時執行臨界區中的代碼。

儘管如我們上邊所看到的，管程提供了一種實現互斥的簡便途徑，但這還不夠。我們還需要一種辦法以使得進程在無法繼續運行時被阻塞。在生產者－消費者問題中，很容易將針對緩衝區滿和緩衝區空的測試放到管程的過程中，但是生產者在發現緩衝區滿的時候如何阻塞?

解決方法在於引入條件變數(condition variables)，及相關的兩個操作：WAIT和SIGNAL。當一個管程過程發現它無法繼續時（例如，生產者發現緩衝區滿），它在某些條件變數上執行WAIT，如full。這個動作引起調用進程阻塞。它也允許另一個先前被擋在管程之外的進程現在進入管程。

另一個進程，如消費者，可以通過對其夥伴正在其上等待的一個條件變數執行SIGNAL來喚醒正在睡眠的夥伴進程。為避免管程中同時有兩個活躍進程，我們需要一條規則來通知在SIGNAL之後該怎麼辦。Hoare建議讓新喚醒的進程運行，而掛起另一個進程。Brinch Hansen則建議要求執行SIGNAL的進程必須立即退出管程。換言之，SIGNAL語句只可能作為一個管程過程的最後一條語句。我們將採納Brinch Hansen的建議，因為它在概念上更簡單，並且更容易實現。如果在一個條件變數上正有若干進程等待，則對該條件變數執行SIGNAL操作，調度程式將在其中選擇一個使其恢復運行。

條件變數不是計數器，它們並不像信號量那樣積累信號供以後使用，所以如果向一個其上沒有等待進程的條件變數發送信號，則該信號將丟失。WAIT操作必須在SIGNAL之前。這條規則使得實現簡單了許多。實際上這不是一個問題，因為用變數很容易跟蹤每個進程的狀態。一個原本要執行SIGNAL的進程通過檢查這些變數便可以知道該操作是不需要的。

圖2-14中用類Pascal語言給出了使用管程解決生產者－消費者問題的解法。

你可能會覺得WAIT和SIGNAL操作看起來很象前面提到的SLEEP和WAKEUP。它們確實很象，但存在一點很關鍵的區別：SLEEP和WAKEUP之所以失敗是因為當一個進程想睡眠時另一個進程試圖去喚醒它。使用管程將不會發生這種情況。對管程過程的互斥保證了這樣一點：如果管程中的過程發現緩衝區滿，它將能夠完成WAIT操作而不用擔心調度程式可能會在WAIT完成之前切換到消費者進程。消費者進程甚至在WAIT完成且生產者被標誌為不可運行之前根本不允許進入管程。

monitor ProducerConsumer

condition full，empty;

integer count;

procedure enter;

begin

if count=N then wait（full）;

enter\_item;

count：=count+1;

if count=1 then signal（empty）

end;

procedure remove;

begin

if count=0 then wait（empty）;

remove\_item;

count：=count-1;

if count=N-1 then signal（full）;

end;

count：=0;

end monitor;

procedure producer;

begin

while true do

begin

produce\_item;

ProducerConsumer。enter

end

end;

procedure cosumer;

begin

while true do

begin

ProducerConsumer。remove;

consume\_item

end

end;

圖2-14 一個使用管程的生產者－消費者問題解法概要。 在一個時刻僅有一個管程過程活躍，

緩衝區有N個槽。

通過臨界區互斥的自動化，管程使並行程式設計比用信號量更容易保證正確性。但它也有缺點。圖2-14中之所以使用類Pascal，而不象其他例子那樣使用C語言並不是沒有原因的。正如我們早先說過的，管程是一個程式設計語言概念。編譯器必須要識別出管程並用某種方式對互斥作出安排。C、Pascal、及多數其他語言都沒有管程，所以指望這些編譯器來實現互斥規則是不可靠的。實際上，編譯器如何知道哪些過程屬於管程內部，哪些不屬於管程也是一個問題。

上述語言同樣也沒有信號量，但增加信號量是很容易的：你需要做的就是向庫裡加入兩段短小的組合語言程式代碼，以執行UP和DOWN系統調用。編譯器甚至用不著知道它們的存在。當然，作業系統必須知道信號量，但至少如果你有一個基於信號量的作業系統，你仍舊可以使用C或C++（或者甚至是BASIC，如果你喜歡的話）來寫使用者程式。如果使用管程，你就需要一種帶有管程的語言。有幾種語言帶有管程，如併發Euclid（Holt，1983），但非常少見。

與管程和信號量有關的另一個問題，是它們都被設計用來解決訪問一塊公共記憶體的一個或多個CPU上的互斥問題。通過將信號量放在共用記憶體中並用TSL指令來保護它們，我們可以避免竟爭。對於一個具有多個CPU、各CPU擁有自己的私有記憶體、由一個局域網相連的分散式系統，這些原語將失效。得出的結論是：信號量太低級，而管程在少數幾種程式設計語言以外無法使用。同時，這些原語均未提供機器間的資訊交換方法，所以我們還需要其他的東西。

2.2.7消息傳遞

前邊提及的其他的東西就是消息傳遞(message passing)。這種進程間通信方法使用兩條原語SEND和RECEIVE。它們象信號量一樣，是系統調用；而不象管程那樣是語言構件。因此，它們可以很容易地被加入庫常式。例如

send（destination，&message）;

和

receive（source，&message）；

前一個調用向一個給定的目標發送一條消息，後一個調用從一個給定的源（或者是任意源，如果接收者不介意的話）接收一條消息。如果沒有消息可用，則接收者可能阻塞直到一條消息到達。或者它也可以立即返回，並帶回一個錯誤碼。

消息傳遞系統的設計要點

消息傳遞系統面臨許多信號量和管程所未涉及的問題和設計難點。特別是對於通信進程位於網路中不同機器的情況。例如，消息有可能被網路丟失。為了防止消息丟失，發送方和接收方可以達成如下一致：一旦資訊被接收到，接收方馬上回送一條特殊的應答(acknowledgement)消息。如果發送方在一段時間間隔內未收到應答，則進行重發。

現在考慮消息本身被正確地接收，而應答資訊丟失的情況。發送者將重發資訊， 這樣接收者將接收到兩次。對於接收者來說，區分新消息和一條重發的老消息是非常重 要的。通常採用在每條原始消息中嵌入一個連續的序號來解決該問題。如果接收者收到一條消息，它具有與前一條消息一樣的序號，則它就知道這條消息是重複的，可以忽略。

消息系統還需要解決進程命名的問題，這樣才能明確在SEND和RECEIVE調用中所指定的進程。身份認證也是一個問題：客戶如何知道它是在與一個真正的檔案伺服器通信，而不是一個冒充者？

另一方面，對發送者和接收者在同一台機器上的情況，也存在若干設計問題。其中一點是性能。將消息從一個進程拷貝到另一個進段通常比信號量操作和進入一個管程要慢。為了使消息傳遞變得高效，已經做了許多工作。例如，Cheriton（1986）建議限制資訊的大小，使其能裝入機器的寄存器中，然後便可以使用寄存器進行消息傳遞。

用消息傳遞解決生產者－消費者問題

現在我們來看如何用消息傳遞而不是共用記憶體來解決生產者－消費者問題。圖2－15中給出了一種解法。我們假設所有的消息都有同樣的大小，並且尚未接收到的消息由作業系統自動進行緩衝。在該圖中，共使用N 條資訊，這就類似於一塊共用記憶體緩衝區中的N個槽。消費者首先將N條空消息發送給生產者。當生產者向消費者傳遞一個資料項目時，它取走一條空消息並送回一條填充了內容的消息。通過這種方式，系統中總的消息數保持不變，所以消息可以存放在預知數量的記憶體中。

如果生產者的速度比消費者快，則所有的消息最終都將被填滿，於是生產者將阻塞以等待消費者取用後返回一條空消息。如果消費者速度快，則正好相反：所有的消息均為空，等待生產者來填充它們，消費者阻塞以等待一條填充過的消息。

# deifne N 100 /\* 緩衝區中的槽數 \*/

void producer（void）

{

int item;

message m; /\* 訊息緩衝區 \*/

while（TRUE）{

produce\_item（&item）; /\* 產生一些資料放入緩衝區 \*/

receive（consumer，&m）; /\* 等待一條空消息到達 \*/

build\_message（&m，item）; /\* 構造一條消息供發送 \*/

send（consumer，&m）; /\* 向消費者發送一資料項目 \*/

}

}

void consumer（void）

{

int item，i;

message m;

for（i=0;i<N;i++）send（producer，&m）; /\* 發送N條空消息 \*/

while（TRUE）{

receive（producer，&m）; /\* 收到一條包含資料的消息 \*/

extract\_item（&m，&item）; /\* 從消息中析取資料 \*/

send（producer，&m）; /\* 回送空消息作為應答 \*/

consume\_item（item）; /\* 使用資料項目進行操作 \*/

}

}

圖2-15 用N條消息的生產者消費者進程。

消息傳遞可以有許多變體。對於初學者，我們來看如何對消息編址。一種方法是為每個進程分配一個唯一的位址，按進程為消息指定位址。另一種方法是引入一種新的資料結構，稱作信箱(mailbox)。一個信箱就是一個用來對一定數量的消息進行緩衝的地方，典型的情況是消息的數量在信箱創建時確定。當使用信箱時，SEND和RECEIVE調用中的位址參數使用信箱，而不是進程。當一個進程試圖向一個滿的信箱發消息時，它將被掛起，直至信箱內有消息被取走而為新消息騰出空間。

對於生產者－消費者問題，生產者和消費者均應創建足夠容納N條消息的信箱。生產者向消費者信箱發送包含資料的消息，消費者則向生產者信箱發送空消息。當使用信箱時，緩衝機制是很清楚的：目標信箱容納那些被發送但尚未被目標進程接收的消息。

使用信箱的另一種極端情況是徹底去掉緩衝。採用這種方法時，如果SEND在RECEIVE之前執行，則發送進程被阻塞，直到RECEIVE發生。執行RECEIVE時消息可以直接從發送者拷貝到接收者，不用任何中間緩衝。類似地，如果RECEIVE先被執行，則接收者阻塞直到SEND發生。這種策略常被稱為會合(rendezvous)原則。與帶有緩衝的消息方案相比，這種方案實現起來更容易一些，但卻降低了靈活性，因為發送者和接收者一定要以步步緊接的方式運行。

在MINIX（及UNIX）中使用者進程間的通信採用管道，它與信箱在效果上等價。採用信箱的消息系統和管道機制之間的區別實際在於管道沒有預先設定消息的邊界。換言之，如果一個進程向管道寫入10條100位元組的消息，而另一個進程從管道中讀取1000個位元組，則讀進程將一次性地獲得這所有10條消息。而在一個真正的消息系統中，每個READ操作將只返回一條消息。當然，如果進程能夠達成一致：總是從管道中讀寫固定大小的消息，或者每條消息都以一個特殊字元（如分行符號）結束，則不會有任何問題。構成MINIX作業系統的進程之間使用消息大小固定的真正的消息機制進行通信。

2.3 經典IPC問題

作業系統文化中有許多被廣為討論和分析的有趣的問題。以下幾節我們將討論三個較為著名的問題。

2.3.1 哲學家進餐問題

在1965年，Dijkstra提出並解決了一個他稱之為哲學家進餐的同步問題。從那時起，每個發明新的同步原語的人都希望通過解決哲學家就餐問題來展示其同步原語的精妙之處。這個問題可以簡單地描述如下：五個哲學家圍坐在一張圓桌周圍，每個哲學家面前都有一碟通心面，由於麵條很滑，所以要兩把叉子才能夾住。相鄰兩個碟子之間有一把叉子，餐桌如圖2-16所示。

圖2-16 哲學家就餐圖。

哲學家的生活包括兩種活動：即吃飯和思考（這只是一種抽象，即對本問題而言其他活動都無關緊要）。當一個哲學家覺得餓時，他就試圖分兩次去取他左邊和右邊的叉子，每次拿一把，但不分次序。如果成功地獲得了兩把叉子，他就開始吃飯，吃完以後放下叉子繼續思考。這裡的問題就是：為每一個哲學家寫一段程式來描述其行為，要求不能鎖死。（要求拿兩把叉子是人為規定的，我們也可以將義大利麵條換成中國菜，用米飯代替通心面，用筷子代替叉子。）

圖2-17給出了最淺顯的解法。過程take\_fork將一直等到所指定的叉子可用，然後將其取用。不幸的是，這種解法是錯誤的。設想所有五位哲學家都同時拿起左面的叉子，則他們都拿不到右面的叉子，於是發生鎖死。

# define N 5 /\* 哲學家數目 \*/

void philosopher（int i） /\* i：哲學家號從0到4號 \*/

{

while（TRUE）{

think（）; /\* 哲學家正在思考 \*/

take\_fork（i）; /\* 取左面叉子 \*/

take\_fork（（i+1）%N）; /\* 取右面叉子；%為取餘 \*/

eat（）; /\* 吃面 \*/

put\_fork（i）; /\* 放回左面叉子 \*/

put\_fork（（i+1）%N）; /\* 放回右面叉子 \*/

}

}

圖2-17 哲學家進餐問題的一種不正確解法。

我們可以將程式修改一下，規定在拿到左叉後，先檢查右面的叉子是否可用。如果不可用，則先放下左叉，等一段時間再重複整個過程。但這種解法也是錯誤的，儘管與前一種的原因不同。可能在某一個瞬間，所有的哲學家都同時啟動這個演算法，拿起左叉，看到右叉不可用，又都放下左叉，等一會兒，又同時拿起左叉。。。如此這樣永遠重複下去。對於這種情況，即所有的程式都在運行，但卻無法取得進展，就稱為饑餓(starvation)。（即使問題不發生在義大利或中國餐館也被稱為饑餓）

現在你可能會想：如果哲學家在拿不到右叉時等待一段隨機的時間，而不是等待相同的時間，則發生上述鎖步的機會就很小了。這種想法是對的，但在一些應用中人們希望一種完全正確的方案，它不能因為一串靠不住的亂數字而失效（想想核電站中的安全控制系統）。

對圖2－17中的演算法可進行下列改進，它既不會發生鎖死又不會發生饑餓：使用一個二進位信號量對五個think函數之後的語句進行保護。在哲學家開始拿叉子之前，先對信號量mutex執行DOWN。在放回叉子後，再對mutex執行UP。從理論上講，這種解法是可行的。但從實際角度來看，這裡有性能上的局限：同一時刻只能有一位哲學家進餐。而五把叉子實際上允許兩位哲學家同時進餐。

圖2－18中的解法不僅正確，而且對於任意位元哲學家的情況都能獲得最大的並行度。其中使用一個陣列state來跟蹤一個哲學家是在吃飯、思考還是正在試圖拿叉子。一個哲學家只有在兩個鄰居都不在進餐時才允許轉移到進餐狀態。第i位元哲學家的鄰居由宏LEFT和RIGHT定義，換言之，若i為2，則LEFT為1，RIGHT為3。

# define N 5 /\* 哲學家數目 \*/

# define LEFT （i-1+N）%N /\* i的左鄰號碼 \*/

# define RIGHT （i+1）%N /\* i的右鄰號碼 \*/

# define THINKING 0 /\* 哲學家正在思考 \*/

# define HUNGRY 1 /\* 哲學家想取得叉子 \*/

# define EATING 2 /\* 哲學家正在吃面 \*/

typedef int semaphore; /\* 信號量是一個特殊的整型變數 \*/

int state[N]; /\* 記錄每個人狀態的陣列 \*/

semaphore mutex=1; /\* 臨界區互斥 \*/

semaphore s[N]; /\* 每個哲學家一個信號量 \*/

void philosopher（int i） /\* i：哲學家號碼，從0到N-1 \*/

{

while（TRUE）{ /\* 無限迴圈 \*/

think（）; /\* 哲學家正在思考 \*/

take\_forks（i）; /\* 需要兩隻叉子，或者阻塞 \*/

eat（）; /\* 進餐 \*/

put\_forks（i）; /\* 把兩把叉子同時放回桌子 \*/

}

}

void take\_forks（int i） /\* i：哲學家號碼從0到N-1 \*/

{

down（&mutex）; /\* 進入臨界區 \*/

state[i]=HUNGRY; /\* 記錄下哲學家i饑餓的事實 \*/

test（i）; /\* 試圖得到兩隻叉子 \*/

up（&mutex）; /\* 離開臨界區 \*/

down（&s[i]）; /\* 如果得不到叉子就阻塞 \*/

}

void put\_forks（int I） /\* i：哲學家號碼從0到N-1 \*/

{

down（&mutex）; /\* 進入臨界區 \*/

state[i]=THINKING; /\* 哲學家進餐結束 \*/

test（LEFT）; /\* 看一下左鄰居現在是否能進餐 \*/

test（RIGHT）; /\* 看一下右鄰居現在是否能進餐 \*/

up（&mutex）; /\* 離開臨界區 \*/

}

void test（i） /\* i：哲學家號碼從0到N-1 \*/

{

if（state[i]==HUNGRY&&state[LEFT]!=EATING

&&state[RIGHT]!=EATING）{

state[i]=EATING;

up（&s[i]）;

}

}

圖2-18 一個哲學家就餐問題的解決方案。

該程式使用了一個信號量陣列，每個信號量對應一位元哲學家，這樣在所需的叉子被佔用時，想進餐的哲學家可以阻塞。注意每個進程將過程philosopher作為主代碼運行，而其他過程take\_forks、put\_forks和test只是普通的過程，而非單獨的進程。

2.3.2 讀者－寫者問題

哲學家問題對於多個競爭進程互斥地訪問有限資源（如I/O設備）這一類問題的建模十分有用。另一個著名的問題是讀者－寫者問題（Courtois et al., 1971），它為資料庫訪問建立了一個模型。例如，設想一個飛機定票系統，其中有許多競爭的進程試圖讀寫其中的資料。多個進程同時讀是可以接受的，但如果一個進程正在更新資料庫，則所有其他進程都不能訪問資料庫，即使讀操作也不行。這裡的問題是如何對讀者和寫者進行程式設計？圖2-19給出了一種解法。

typedef int semaphore;

semaphore mutex=1; /\* 控制對RC的訪問 \*/

semaphore db=1; /\* 控制對資料庫的訪問 \*/

int rc=0; /\* 正在讀或想要讀的進程數 \*/

void reader（void）

{

while（TRUE）{ /\* 無限迴圈 \*/

down（&mutex）; /\* 排斥對RC的訪問\*/

rc = rc+1; /\*又多了一個讀者\*/

if（rc == 1）down（&db）; /\*如果這是第一個讀者，那麼......\*/

up（&mutex）; /\*恢復對RC的訪問\*/

read\_data\_base（）; /\*訪問資料\*/

down（&mutex）; /\*排斥對RC的訪問\*/

rc = rc -1; /\*讀者又少了一個\*/

if（rc==0）up（&db）; /\*如果這是最後一個讀者，那麼......\*/

use\_data\_read（）; /\*非臨界區操作\*/

}

}

void writer（void）

{

while（TRUE）{

think\_up\_data（）; /\*非臨界區操作\*/

down（&db）; /\*排斥訪問\*/

write\_data\_base（）; /\*修改資料\*/

up（&db）; /\*恢復訪問\*/

}

}

圖2-19 一個讀者與作者問題的解決方案。

該解法中，第一個讀者對信號量db執行DOWN。隨後的讀者只是遞增一個計數器rc。當讀者離開時，它們遞減這個計數器，而最後一個讀者則對db執行UP，這樣就允許一個阻塞的寫者（如果存在的話）可以訪問資料庫。

這個解法在這裡隱含了一條很微妙的規則值得討論。設想當一個讀者在使用資料庫時，另一個讀者也來訪問資料庫，由於同時允許多個讀者同時進行讀操作，所以第二個讀者也被允許進入，同理第三個及隨後更多的讀者都被允許進入。

現在假設一個寫者到來，由於寫操作是排他的，所以它不能訪問資料庫，而是被掛起。隨後其他的讀者到來，這樣只要有一個讀者活躍，隨後而來的讀者都被允許訪問資料庫。這樣的結果是只要有讀者陸續到來，它們一來就被允許進入，而寫者將一直被掛起直到沒有一個讀者為止。假如每2秒鐘來一個讀者，而其操作時間為5秒鐘，則寫者將永遠不能訪問資料庫。

為了防止這種情況，程式可以略作如下改動：當一個讀者到來而正有一個寫者在等待，則讀者被掛在寫者後邊，而不是立即進入。這樣，寫者只需等待它到來時就處於活躍狀態的讀者結束，而不用等那些在它後邊到來的讀者。這種解法的缺點是併發性較低，從而性能較差。 Courtois等人給出了一個寫者優先的解法。詳細請參閱他的論文。

2.3.3 理髮師睡覺問題

另一個經典的IPC問題發生在理髮店裡。理髮店裡有一位理髮師、一把理髮椅和n把供等候理髮的顧客坐的椅子。如果沒有顧客，則理髮師便在理髮椅上睡覺，如圖2-20所示。當一個顧客到來時，他必須先叫醒理髮師，如果理髮師正在理髮時又有顧客來到，則如果有空椅子可坐，他們就坐下來等。如果沒有空椅子，他就離開。這裡的問題是為理髮師和顧客各編寫一段程式來描述他們的行為，要求不能帶有競爭條件。

圖2-20 睡覺的理髮師。

我們的解法使用三個信號量：customers，用來記錄等候理髮的顧客數（不包括正在理髮的顧客）；barbers，記錄正在等候顧客的理髮師數，為0或1；mutex，用於互斥。我們也需要一個變數waiting，它也用於記錄等候的顧客數，它實際上是customers的一份拷貝。之所以使用waiting是因為無法讀取信號量的當前值。在該解法中，進入理髮店的顧客必須先看等候的顧客數，如果少於椅子數，他留下來等，否則他就離開。

我們的解法示於圖2-21。當理髮師早晨開始工作時，他執行過程barber，這將導致他阻塞在信號量customers上，直到有人到來。這時他將去睡覺，如圖2-20所示。

當一個顧客到來時，他執行過程customer，首先獲取信號量mutex以進入臨界區。如果不久另一位顧客到來，則他只能等到第一位釋放了mutex後才能做別的事。顧客隨後檢查是否等候的顧客少於椅子數，如果不是，他就釋放mutex並離開。

如果有一把椅子可坐，則他遞增整型變數waiting，隨後對信號量customers執行UP，然後喚醒理髮師。此時顧客和理髮師都處於清醒狀態。當顧客釋放mutex時，理髮師獲得mutex，他進行一些準備後開始理髮。

當理完發後，顧客退出該過程，並離開理髮店。與前邊的例子不同，這裡顧客不執行迴圈，因為每個顧客只需理一次發。但理髮師必須執行迴圈以服務下一位元顧客。如果有顧客，則為顧客理髮，否則就去睡覺。

這裡需要指出的是儘管讀者－寫者問題和理髮師問題都不涉及資料傳遞，但它們仍屬於IPC範圍，因為它們涉及多進程間的同步。

# define CHAIRS 5 /\*為等待的顧客準備的椅子數\*/

typedef int semaphone ; /\*運用你的想像力\*/

semaphone customers=0; /\*等待服務的顧客數\*/

semaphone barbers=0; /\*等待顧客的理髮師數\*/

semaphone mutex=1; /\*用於互斥\*/

int waiting=0; /\*等待的顧客（還沒理髮的）\*/

void barber（void）

{

while（TRUE）{

down（customers）; /\*如果顧客數是0，則睡眠\*/

down（mutex）; /\*要求進程等待\*/

waiting=waiting-1; /\*等待顧客數減1\*/

up（barbers）; /\*一個理髮師現在開始理髮了\*/

up（mutex）; /\*釋放等待\*/

cut\_hair（）; /\*理髮（非臨界區操作）\*/

}

}

void customers（void）

{

down（mutex）; /\*進入臨界區\*/

if（waiting<CHAIRS）{ /\*如果沒有空的椅子，就離開\*/

waiting=waiting+1; /\*等待顧客數加1\*/

up（customers）; /\*如果必要的話，喚醒理髮師\*/

up（mutex）; /\*釋放訪問等待\*/

down（barbers）; /\*如果barbers為0，就入睡\*/

get\_haircut（）; /\*坐下等待服務\*/

} else {

up（mutex）; /\*店裡人滿了，走吧\*/

}

}

圖2-21 理髮師問題的一種解法.

2.4進程調度

在前幾節的例子中，我們經常遇到兩個或多個進程（例如，生產者和消費者）在邏輯上均可以運行的情況。當有多個進程就緒時，作業系統必須決定先運行哪一個。作業系統中作出這種決定的部分稱作調度程式(scheduler)，它使用的演算法稱作調度演算法(scheduling algorithm)。

再回到早期以磁帶上的卡片映射作為輸入的批次處理系統時代，那時的調度演算法很簡單：依次運行磁帶上的下一個作業。對於分時系統，則調度演算法要複雜一些，因為經常有多個使用者等待服務，而且同時可能存在多個批次處理流（例如，保險公司理賠）。即使在個人電腦上，也可能有若干使用者啟動的進程競爭CPU，更不要說還有後臺作業，例如網路或收發電子郵件的精靈進程。

在討論具體的調度演算法之前，我們應該考慮一下調度程式要達到的目標。畢竟調度程式關注的是確定策略，而並不提供機制。一個好的調度演算法應當考慮很多方面，其中可能有：

1 公平 - 確保每個進程獲得合理的CPU份額。

2 有效 - 使CPU百分之百地忙碌。

3 回應時間 - 使互動使用者的回應時間盡可能短。

4 周轉時間 - 使批次處理用戶等待輸出的時間盡可能短。

5 輸送量 - 使每小時處理的作業數最多。

對這些目標稍加思考便會發現其中有矛盾之處。為了使互動使用者的回應時間最短，應該對批次處理作業不予調度（除了淩晨3點到6點，當用戶休息時）。然而批次處理使用者可能不歡迎這種演算法，它違背了條件4。可以證明（Kleinrock，1975）任何一個偏向某些類型作業的調度演算法必將損害另一些作業。畢竟可利用的CPU時間是有限的。多給一個用戶就要少給另一個用戶，生活就是這樣。

調度程式必須面對的另一個麻煩是每個進程都不一樣，而且不可預測。有些進程花費很多時間等待I/O，而另一些進程在允許的條件下將連續使用CPU達幾個小時。 當調度程式啟動運行某些進程時，它根本不知道進程在阻塞前要運行多久（阻塞可能是因為I/O、信號量、或者其他原因）。為了保證不讓進程運行得太久，幾乎所有的電腦都內置一個電子計時器或時鐘，它將定期地發出中斷，通常每秒鐘50或60次（亦稱作50或60赫茲，簡寫為Hz）。但在許多電腦上，作業系統能夠根據需要將時鐘頻率設置成任意值。每發生一次時鐘中斷，作業系統都將運行，並決定當前進程是否應繼續運行，還是它已經佔用了足夠長的CPU時間，應該暫停讓其他進程運行。

允許將邏輯上可運行的進程暫時掛起的策略稱作可剝奪調度(preemptive scheduling)，這與早期批次處理系統使進程運行直到結束的方法正好相反。運行直到結束的調度方式稱作非剝奪調度(nonpreemptive scheduling)。正如我們在本章中看到的，進程可在任意時刻被不加警告地掛起，以便讓另一個進程運行。這導致了競爭條件以及防止競爭條件的信號量、管程、消息或其他複雜的方法。另一方面，允許一個進程運行它所希望的時間意味著一個計算圓周率小數點後邊十億位元的進程將使其他進程永遠得不到服務。

所以儘管非剝奪調度演算法簡單且易於實現，但它通常不適於具有多個競爭使用者的通用系統。另一方面，對於專用系統，如一個資料庫伺服器，主進程在收到請求時啟動一個子進程並讓其運行直到結束或阻塞則是很合理的。這種系統與通用系統的區別在於資料庫系統中的所有進程都處於單個主進程的控制之下，該主進程知道各子進程將做什麼，以及將花費的時間。

2.4.1 時間片輪轉調度

現在讓我們來看具體的調度演算法。一種最古老，最簡單，最公平且使用最廣的演算法是時間片調度。每個進程被分配一個時間段，稱作它的時間片，即該進程允許運行的時間。如果在時間片結束時進程還在運行，則CPU將被剝奪並分配給另一個進程。如果進程在時間片結束前阻塞或結束，則CPU當即進行切換。時間片輪轉調度很容易實現。調度程式所要做的就是維護一張就緒進程清單，如圖2-22（a）所示，當進程用完它的時間片後，它被移到佇列的末尾，如圖2-22（b）所示。

圖2-22 時間片輪轉調度。 a）就緒進程清單。 （b）進程B用完它的時間片後的就緒進程清單。

　 時間片輪轉調度中唯一有趣的一點是時間片的長度。從一個進程切換到另一個進程是需要一定時間的 - 保存和裝入寄存器值及記憶體映射，更新各種表格和佇列等。假如進程切換(process switch) - 有時稱為上下文切換(context switch)，需要5毫秒。再假設時間片設為20毫秒。則在做完20毫秒有用的工作之後，CPU將花費5毫秒來進行進程切換。CPU時間的20%被浪費在了管理開銷上。

為了提高CPU效率，我們可以將時間片設為500毫秒。這時浪費的時間只有1%。但考慮在一個分時系統中，如果有十個互動使用者幾乎同時按下回車鍵，將發生什麼情況？這時十個進程被掛在就緒佇列中，如果CPU空閒，則立即啟動第一個進程，第二個進程在大約1/2秒之後啟動，依次類推。假設所有其他進程都用足它們的時間片的話，最後一個不幸的進程不得不等待5秒鐘才獲得運行機會。多數用戶無法忍受一條簡短命令要5秒鐘才能做出回應。同樣的問題在一台支援多道程序的個人電腦上也會發生。

結論可以歸結如下：時間片設得太短會導致過多的進程切換，降低了CPU效率；而設得太長又可能引起對短的交互請求的回應變差。將時間片設為100毫秒通常是一個比較合理的折衷。

2.4.2 優先順序調度

　　　時間片調度做了一個內在的假設，即所有的進程同等重要．而擁有和操作多使用者電腦系統的人對此常有不同的看法．在一所大學裡，等級順序可能是教務長、教授、秘書、勤務人員、最後是學生．這種將外部因素考慮在內的需要就導致了優先順序調度．其基本思想很清楚：每個進程被賦予一個優先順序，優先順序最高的就緒進程被率先運行．

　　　即使在只有一個用戶的PC機上，也可能有多個重要程度不同的進程。例如，一個在後臺收發電子郵件的精靈進程應被賦予一個較低的優先順序，而在螢幕上即時地播放電影的進程則應賦予較高的優先順序。

　　　為了防止高優先順序進程無休止地運行下去，調度程式可能在每個時鐘滴答（即每一個時鐘中斷）降低當前進程的優先順序。如果這個動作導致其優先順序低於次高優先順序，則將進行進程切換。或者給每個進程設定一段它能夠連續使用CPU的時間片，一旦這段時間用完，則運行次高優先順序的進程。

　　　優先順序可以為靜態或動態。在一台軍用電腦上，將軍啟動的進程優先順序為100，上校為90，少校為80，上尉為70，中尉為60，依次類推。或者在一個商業計算中心，高優先順序作業每小時費用為100美元，中等優先順序每小時75美元，低優先順序每小時50美元。UNIX系統中有一條命令nice，它允許用戶為了照顧別人而自願降低其進程的優先順序，但從未有人用它。

優先順序也可以被系統動態地確定以達到某種目的。例如，有些進程為I/O密集型，其多數時間用來等待I/O結束。當這樣的進程需要CPU時，它應被立即分配CPU，以便啟動下一個I/O請求，這樣就可以在另一個進程計算的同時執行I/O操作。使這類進程長時間等待CPU只會造成它無謂地長時間佔用記憶體。使I/O密集進程獲得較好服務的一種簡單演算法是將其優先順序設為1/f，f為該進程上一時間片中計算時間所占的比重。一個在100毫秒中計算只佔用2毫秒的進程優先順序為50，而計算佔用50毫秒的進程優先順序則為2。全部時間都在計算的進程則為1。

很方便就可以將一組進程按優先順序分成若干類。在各類之間採用優先順序調度，而各類進程內部採用時間片輪轉調度。圖2-23顯示了一個有4類優先順序的系統，調度演算法如下：只要存在優先順序為第4類的就緒進程，就按照時間片輪轉法使其運行一個時間片，此時不理會較低優先順序的進程。若第4類進程為空，則運行第3類進程。若第4、第3類均為空，則按時間片法運行第2類進程。如果對優先順序不經常進行調整，則低優先順序進程很可能會發生饑餓。

圖2-23 一個有四類優先順序的調度演算法。

2.4.3 多重佇列

CTSS（Corbato et al., 1962）是最早使用優先順序調度的系統之一。但是CTSS存在進程切換速度太慢的問題，其原因是IBM 7094記憶體中只放得下一個進程，每次切換都需要將當前進程換出到磁片，並從磁片上讀入一個新進程。CTSS的設計者很快便認識到為CPU密集的進程設置較長的時間片，比頻繁地分給它們很短的時間片要高效（減少交換次數）。另一方面，如前所述，給進程長時間片又會影響回應時間。他們的解決辦法是設立優先順序類。屬於最高級類的進程運行一個時間片，屬於次高優先順序類的進程運行2個時間片，再次一級運行4個時間片，依次類推。當一個進程用完分配的時間片後，它被移到下一類。

我們看一個例子，有一個進程需要連續計算100個時間片。它最初被分配1個時間片，然後被換出。下次它將獲得2個時間片，接下來分別是4、8、16、32和64。當然最後一次它只使用64個時間片中的37個便可以結束工作。該過程需要7次交換（包括最初的裝入），而如果採用純粹的時間片輪轉則需要100次交換。而且，隨著進程優先順序的不斷降低，它的運行頻度逐漸放慢，從而為短的交互進程讓出CPU。

對於那些剛開始運行一段長時間，而後來又需要交互的進程，為了防止其優先順序降低過快可以採取這樣的策略：只要終端上有回車鍵按下，則屬於該終端的所有進程都被移到最高優先順序，這樣做的原因是認為此時進程即將需要交互。但可能有一天，一台重載的機器上有幾個用戶偶然發現，只需坐在那裡每過隨機的幾秒鐘敲一下回車鍵就可以大大地提高回應速度，於是他又告訴所有的朋友。這件事的結論是：實踐中行得通比理論上可行要困難得多。

已經有許多其他演算法可用來將進程劃分為優先順序類。例如，在伯克利製造的著名的XDS 940系統中（Lampson，1968），它有4個優先順序類，分別是終端、I/O、短時間片和長時間片。當一個等待終端輸入的進程最終被喚醒時，它轉到最高優先順序類（終端）。當一個等待一塊磁片資料的進程就緒時，它將轉到第二類。當進程在時間片用完時仍為就緒時，它被放入第三類。但如果一個進程已經多次用完時間片而從未因終端或其他I/O阻塞，它將被轉入最低優先順序類。許多其他系統也使用類似的演算法，以側重互動使用者和交互進程，而犧牲後臺進程。

2.4.4 最短作業優先

上述多數演算法都是為交互系統設計的。現在來看一種適用於執行時間可以預知的批次工作的調度演算法。例如一家保險公司，因為每天都做類似的工作，所以人們可以相當精確地預測一個處理1000起索賠的作業需要多長時間。當輸入佇列中有若干個同等重要的作業將被啟動時，調度程式應使用最短作業優先演算法。請看圖2-24，這裡有4個作業A、B、C、D，執行時間分別為8、4、4、4分鐘。若按圖中的次序運行，則A的周轉時間為8分鐘，B為12分鐘，C為16分鐘，D為20分鐘，平均為14分鐘。

圖2-24 一個最短作業優先調度的例子。

現在考慮使用最短作業優先演算法。如圖2-24（b）所示。現在周轉時間分別為4、8、12和20分鐘，平均為11分鐘。可以證明最短作業優先是最優的。考慮有4個作業的情況，其執行時間分別為a，b，c，d。 第一個作業在時間a結束，第二個在時間a+b結束，依次類推。平均周轉時間為（4a+3b+2c+d）/4，顯然a對平均值影響最大，所以它應是最短作業，其次是b，再次是c，最後的d只影響它自己的周轉時間。對任意數目作業的情況道理完全一樣。

由於最短作業優先常常伴隨著最短響應時間，所以如果它同時能夠被用於交互進程，那將是非常好的。在某種程度上，它的確可以做到這一點。交互進程通常遵循下列模式：等待命令，然後執行命令。如此不斷反復。如果我們將每一條命令的執行看作一個獨立的作業，則我們可以通過首先運行最短的作業來使回應時間最短。唯一的問題是如何從當前就緒進程中找出最短的那個。

一種辦法是根據進程過去的行為進行推測，並執行估計執行時間最短的那個。假設某終端上每條命令的估計執行時間為T0，現在假設測量到其下一次執行時間為T1，我們可以將這兩個值的加權和來改進我們的估計時間，即aT0+（1-a）T1。通過選擇a的值，我們可以決定是儘快忘掉老的執行時間，還是在一段長時間內還記住它們。當a=1/2時，我們可以得到如下序列：

T0，T0/2+T1/2，T0/4+T1/4+T2/2，T0/8+T1/8+T2/4+T3/2

我們看到，三輪過後，T0在新的估計值中的占的比重下降到1/8。

這種通過將當前測量值和先前估計值進行加權平均而得到下一個估計值的技術有時稱作老化。它適用於許多預測值必須基於先前值的情況。老化演算法在a=1/2時特別容易實現，只需將新值加到當前估計值上然後除以2（即右移一位）。

需要指出的是，最短作業優先演算法只在所有作業同時可用的情況下才是最優的。作為一個反例，考慮5個作業，A到E，執行時間分別為2，4，1，1，1；到達時間分別為0，0，3，3，3。

最初只有A和B可被選擇，因為另外三個作業尚未到達。使用最短作業優先將按照A，B，C，D，E的順序運行，其平均等待時間為4.6。而按照B，C，D，E，A的順序則平均等待時間為4.4。

2.4.5 保證調度演算法

一種完全不同的調度演算法是向使用者作出明確的性能保證，然後去實現它。一種很實際並很容易實現的保證是：若你工作時有n個用戶登錄，則你將獲得CPU處理能力的1/n。類似的，在一個有n個進程運行的單使用者系統中，若所有的進程都平等，則每個進程將獲得1/n的CPU時間。

為了實現其所作的保證，系統必須跟蹤各進程自創建以來已使用了多少CPU時間。然後它計算各進程應獲得的CPU時間，即自從創建以來的時間除以n。由於各進程實際獲得的CPU時間已知，所以很容易計算出真正獲得的CPU時間和應獲得的CPU時間之比。比率為0.5說明一個進程只獲得了應得時間的一半，而比率為2.0則說明它獲得了應得時間的2倍。於是該演算法隨後轉向比率最低的進程，直到該進程的比率超過次最低進程為止。

2.4.6 彩票調度演算法

儘管向用戶作出承諾並履行它是一個好主意，但實現卻很困難。不過有另一種演算法可以給出類似的可預見結果，而且實現起來簡單許多，這種演算法稱為彩票調度法（Waldspurger和Weihl 1994）。

其基本思想是為進程發放針對系統各種資源（如CPU時間）的彩票。當調度程式需作出決策時，隨機選擇一張彩票，持有該彩票的進程將獲得系統資源。對於CPU調度，系統可能每秒鐘抽50次彩票，每次的中獎者獲得20毫秒的執行時間。

George Orwell對此解釋為：“所有進程都是平等的，而某些進程則需要更多的機會。”更重要的進程被給予更多的額外彩票，以增加其中獎機會。如果共發出100張彩票，而一個進程持有20張，它就有20%的中獎概率，對於長時間運行，它將獲得大約20%的CPU時間。彩票演算法與優先順序調度完全不同，在後者中很難說清楚優先順序為40說明了什麼，而在前者則很清楚：進程擁有多少彩票份額，它就將獲得多少資源。

彩票調度法有幾點有趣的特性。例如，如果一個新進程創建並得到一些彩票，則在下次抽獎時，它中獎的機會與其持有的彩票數成正比。換言之，彩票調度的反應非常迅速。

合作進程如果願意的話可以交換彩票。例如，一個客戶進程向伺服器進程發送一條消息並阻塞，它可以把所有的彩票都交給伺服器進程，以增加其下一次被運行的機會。當伺服器進程結束後，它又將彩票交還給客戶進程以使其能夠再次運行。實際上，在沒有客戶時，伺服器根本不需要彩票。

彩票調度能用來解決其他演算法難以解決的問題。例如，一個視訊伺服器，其中有若干個進程將視頻資訊傳送給各自的客戶，但幀頻不同。假設其分別需要為10、20和25幀/秒的速度，則通過為這些進程分別分配10、20和25張彩票，它們將自動地按照正確的比例分配CPU資源。

2.4.7 即時調度

即時系統是那些時間因素非常關鍵的系統。例如，電腦的一個或多個外設發出信號，電腦必須在一段固定時間內作出適當的反應。一個實例是，電腦用CD-ROM放音樂時從驅動器獲得二進位資料，並且必須在很短的時間內將其轉換成音樂。如果這中間計算花的時間太長，音樂聽起來就會失真。其他即時系統還包括：醫院裡特護病房的監控系統、飛行器中的自動駕駛儀、以及核反應爐中的安全控制系統等。在這些系統中，遲到的回應即使正確，也和沒有回應一樣糟糕。

即時系統通常分為硬實時(hard real time)系統和軟即時(soft real time)系統。前者意味著存在必須滿足的時間限制；後者意味著偶爾超過時間限制是可以容忍的。這兩種系統中，即時性的獲得是通過將程式分成許多進程，而每個進程的行為都預先可知。這些進程通常生存週期都很短，往往在一秒內便運行結束。當檢測到一個外部事件時，調度程式按滿足它們最後期限的方式調度這些進程。

即時系統要回應的事件進一步分為週期性（每隔一段固定的時間發生）和非週期性（在不可預測的時間發生）。一個系統可能必須響應多個週期的事件流。根據每個事件需要多長的處理時間，系統可能根本來不及處理所有事件。例如，有m個週期性事件，事件I的週期為Pi，其中每個事件需要Ci秒的CPU時間來處理。則只有滿足以下條件

m

∑ Ci < 1

i=1 Pi

時，才可能處理所有的負載。滿足該條件的即時系統稱作是可調度的(schedulable)。

舉例來說，一個軟即時系統處理三個事件流，其週期分別為100，200和500毫秒。如果事件處理時間分別為50，30和100毫秒，則這個系統是可調度的，因為0.5+0.15+0.2 < 1。如果加入週期為1秒的第4個事件，則只要其處理時間不超過150毫秒，該系統仍將是可調度的。這個運算的隱含條件是上下文切換的開銷很小，可以忽略。

即時調度演算法可以是動態或靜態。前者在運行時作出調度決定，後者在系統啟動之前完成所有的調度決策。我們簡要地考慮幾種即時調度演算法。經典的演算法是發生率單調演算法（Liu和Layland，1973）。該演算法事先為每個進程分配一個與事件發生頻率成正比的優先順序。例如，週期為20毫秒的進程優先順序為50，而週期為100毫秒的進程為10。運行時，調度程式總是調度優先順序最高的就緒進程，必要時將剝奪當前進程。Liu和Layland證明了該演算法是最優的。

另一種流行的即時調度演算法是最早截止優先演算法。當一個事件發生時，對應的進程被加到就緒佇列中。該佇列按照截止期限排序，對於一個週期性事件，其截止期限即為事件下次發生的時間。該演算法首先運行隊首進程，即截止時間最近的那個。

第三種演算法首先計算各進程的富餘時間，稱做作裕度(laxity)。如果一個進程需要運行200毫秒，而它必須在250毫秒內完成，則其裕度為50 毫秒，該演算法稱作最少裕度法，即選擇裕度最少的進程。

儘管在理論上通過使用這三種調度演算法中的一種可以將一個通用作業系統轉變為一個即時系統，但實際上，通用作業系統的上下文切換開銷太大，以至於只對那些時間限制較松的應用才能達到其即時性能要求。這就導致多數即時系統使用專用的即時操作系統。這些系統具有一些很重要的特徵，典型的包括：規模小、中斷時間很短、進程切換很快、中斷被遮罩的時間很短，以及能夠管理毫秒或微秒級的多個計時器等。

2.4.8 兩級調度法

到目前為止，我們或多或少總是假設就緒進程都在記憶體中。如果沒有足夠的記憶體，則某些就緒進程將全部或部分地被放在磁片上，這種情況對調度有很大影響，因為從盤上讀入一個進程運行比單純在記憶體中進行進程切換要慢幾個數量級。

處理被對換到外存的進程可以使用一種更實際的辦法，即使用兩級調度。就緒進程的一個子集首先被裝入記憶體，如圖2-25（a）所示。調度程式在隨後的一段時間裡只在這個子集中進行調度。一個高級調度程式週期性地將那些在記憶體中駐留時間足夠長的進程換出，而將那些在磁片上等候時間過長的進程換入。當這個操作完成後，如圖2-25（b）所示，低級調度程式再次在那些駐留在記憶體中的進程間進行調度。這樣，低級調度程式只關心當時在記憶體中的就緒進程，而高級調度程式則關心將進程在記憶體和磁片間來回交換。

圖2-25 兩級調度程式必須將進程在磁片和記憶體間來回移動，並從駐留在記憶體的進程中進行選擇。 三個不同時刻的情況示於（a）、（b）、（c）。

高級調度程式用於決策的標準包括：

1 進程自上次被換入或換出以來的時間

2 進程最近使用的CPU時間

3 進程的大小（小進程不參與高級調度）

4 進程的優先順序

這裡我們可以再次用到時間片輪轉、優先順序調度，或其他各種演算法。高級和低級調度程式可以使用相同或不同的演算法。

2.4.9 策略與機制

截至目前，我們隱含地假設系統中所有進程分屬不同的使用者，並且相互間競爭CPU。儘管通常是這樣，但有時會有這樣的情況：一個進程有許多子進程在其控制之下運行。例如，一個資料庫管理系統可能有許多子進程，每個子進程可能處理不同的請求，或者每個子進程實現不同的功能（請求的語法分析，訪問磁片等）。主進程完全可能掌握哪個子進程最重要（或最緊迫），哪個最不重要。但不幸的是，以上討論的調度演算法中沒有一個從使用者進程接受有關的調度決策資訊。這就導致調度程式很少能夠作出最優的選擇。

解決該問題的方法是將調度機制(scheduling mechanism)和調度策略(scheduling policy)分開。亦即將調度演算法以某種形式參數化，而參數可以由使用者進程來填寫。我們再來看資料庫的例子。假設核心使用優先順序調度演算法，但提供一條系統調用，一個進程可以使用它來設置或改變其子進程的優先順序。這樣儘管父進程本身並不進行調度，但它可以控制子進程如何被調度的細節。在這裡，機制位於核心，而策略則由使用者進程設定。

2.5 MINIX進程概述

在研究了進程管理、進程間通信以及進程調度的原理之後，我們現在來看這些原理在MINIX中的應用。MINIX與UNIX不同，UNIX的核心是一個不分模組的單塊程式，而MINIX本身就是一組進程的集合，它們相互之間、以及與使用者進程之間使用進程間通信機制 - 消息傳遞來進行通信。這種設計使得MINIX的結構更加模組化和靈活。例如，這使得很容易就可以將整個檔案系統替換成另一個完全不同的檔案系統，而無需重新編譯核心。

2.5.1 MINIX的內部結構

開始研究MINIX之前，我們首先大概流覽一下整個系統。MINIX被組織成4層，每一層執行一套定義得很完好的功能。這四層示於圖2-26。

圖2-26 MINIX的四層結構。

最底層捕獲所有的中斷和陷入，完成進程調度，並向高層提供一個採用消息進行通信的獨立順序進程模型。該層的代碼有兩大主要功能。第一是捕獲陷入和中斷、保存和恢復寄存器、調度以及向高層提供一個獨立順序進程模型。第二是處理消息機制：檢查目標進程的合法性、定位實體記憶體中的發送和接收緩衝區、以及從發送方向接收方拷貝資料。其中中斷處理的最底層部分用組合語言編寫，其餘部分和其他層次用C語言編寫。

第二層包括I/O進程，每類設備都有一個I/O進程。為了將其與普通使用者進程相區別，我們稱之為任務(tasks)。但任務與進程間的差別微乎其微。在許多系統中I/O任務被稱作設備驅動程式(device driver)。這裡“任務”和“設備驅動程式”可以換用。每一類設備都需要一個任務，包括磁片、印表機、終端、網路介面以及時鐘。如果有其他I/O設備，則它們也需要相應的任務。有一個任務 - 系統任務有些與眾不同，它不對應於任何I/O設備，我們將在下一章對這些任務進行討論。

第二層的所有任務和第一層的代碼連結成一個單一的二進位程式，稱作核心(kernel)。某些任務共用公共的子常式，但它們相互之間完全獨立，分別進行調度，並採用消息進行通信。從286開始的Intel處理器為每個進程賦予四種特權級中的一種。儘管任務與核心被編譯在一起，但當核心和中斷處理常式被執行時，它們被賦予較任務更高的特權級。所以真正的核心代碼可以訪問任一部分記憶體，以及任一處理器寄存器 - 實質上，核心可以使用系統中任何地方的資料執行任何指令。任務不能執行全部的機器指令，也不能訪問所有CPU寄存器或所有的記憶體。但在為較低特權級的進程執行I/O時，任務可以訪問屬於這些進程的記憶體區域。有一個任務 - 系統任務，它並不執行一般意義的I/O，其作用提供某些特定服務，例如當進程本身不允許在不同的記憶體區域間進行拷貝時，由系統任務執行此操作。當然在不提供多特權級的機器上，例如老式的Intel處理器，無法強制執行這些限制。

第三層包含向使用者進程提供有用服務的進程。這些伺服器進程在低於核心和任務的特權級上運行，不能直接訪問I/O埠。它們也不能訪問屬於自己的段以外的記憶體。記憶體管理器（Memory Manager - MM）負責執行所有牽涉到記憶體管理的系統調用，如FORK、EXEC和BRK。檔案系統（File System - FS）負責執行檔案系統的調用，READ、MOUNT和CHDIR。

正如我們在第一章開頭所指出的，作業系統做兩件事情：管理資源和通過系統調用方式提供擴展的電腦。在MINIX中，資源管理主要在核心（第1、2層）中，系統調用的解釋在第三層。檔案系統被設計成一個“伺服器”，並幾乎可以不加修改地移到一台遠端機器上。這也適用於記憶體管理器，儘管遠端記憶體管理器不如遠端檔案系統那樣有用。

第三層也可能存在其他的伺服器。圖2 - 26顯示出那裡有一個網路服務器。儘管本書中描述的MINIX不包括網路，但網路的源碼卻是標準MINIX發佈軟體的一部分。 系統很容易被重新編譯以包括網路。

此處正適於指出儘管伺服器是獨立的進程，但它們和使用者進程有一點不同，即它們在系統啟動的同時被啟動，而且在系統活躍期間不會終止。此外，儘管從它們禁用的機器指令來看，它們與使用者進程運行在相同的特權級上，但它們的執行優先順序比使用者進程高。為了加入一個新的伺服器，核心必須重新編譯。核心的啟動代碼在使用者進程開始運行之前將伺服器進程安裝在進程表的特權表項中。

最後，第四層包含所有的使用者進程 - shell、編譯器、編輯器以及用戶的a.out程式。一個運行系統通常有一些進程在系統引導時啟動，並一直運行，例如，一個精靈程式就是一個週期性運行或總是等待某個事件（例如網路上一個包的到達）的後臺進程。從某種意義上說，精靈進程是一個單獨啟動而作為一個使用者進程運行的伺服器。但與裝入特權進程表項的真正的伺服器不同，這些程式無法象記憶體和檔案伺服器那樣受到核心的特殊對待。

2.5.2 MINIX中的進程管理

MINIX中的進程遵從本章前邊所描述的通用進程模型。進程可以創建子進程，子進程又可以創建更多的子進程，這樣便構造出一棵進程樹。實際上，整個系統中所有的使用者進程都屬於以init（見圖2 - 26）為根節點的一棵進程樹。

這種情況是怎樣產生的呢? 當電腦開機時，硬體從引導盤上將第一道第一磁區讀入記憶體並從那裡開始執行。具體細節根據引導盤是軟碟還是硬碟而不同。在軟碟上，第一磁區包含了引導程式(bootstrap)。引導程式很小，因為它必須能容納在一個磁區裡。MINIX引導程式裝入一個更大的程式boot，由boot裝入作業系統。

相比之下，硬碟需要一個中間步驟。硬碟被分成若干分區(partition)，整個硬碟的第一個磁區包括一段小程式和磁碟分割表，通常稱為主引導記錄。程式部分被執行以讀入分區表並選擇活躍分區。活躍分區的第一個磁區有一個引導程式，它隨後被裝入並執行以查找並啟動程式boot，這與從軟碟引導完全相同。

不管哪種情況，boot將在軟碟或硬碟分區上找一個包含多個部分的檔，並將各部分裝到記憶體的適當位置。這些部分包括核心、記憶體管理器、檔案系統以及init - 第一個使用者進程。這個啟動過程並不簡單。所有那些屬於磁片任務和檔案系統範圍的操作在這兩部分活躍之前都要由boot完成。在後邊我們將回到MINIX如何啟動這一個主題。現在只需知道一旦裝入操作完成，核心便開始運行。

在其初始化階段，核心先啟動各任務，然後是記憶體管理器、檔案系統及所有在第三層運行的伺服器。當所有這些都開始運行並完成初始化之後，它們將阻塞，等待執行某種操作。當所有的任務和伺服器被阻塞之後，將執行第一個使用者進程 - init。init已經位於記憶體，不過在它啟動時其他所有部分均已就位，所以它可以作為一個獨立的程式從磁片上裝入。但是，由於init只啟動一次，而且不會再從盤上裝入，所以最簡便的方法是把它與核心、任務和伺服器一起包括在系統的映射檔中。

init啟動後先讀/etc/ttytab檔，該檔列出了所有可能的終端設備。那些可以作為登錄終端（在標準MINIX中，只包括控制台）的設備都在/etc/ttytab的getty域有一項。而且init為每個這樣的終端創建一個子進程。通常每個子進程都執行檔/usr/bin/getty，列印出一條資訊，然後等待輸入一個用戶名。隨後將該用戶名作為參數來調用 /usr/bin/login。如果某個終端需要特殊的處理（例如，一條撥號線），則/etc/ttytab可以指定一條命令（如/usr/bin/stty），在執行getty之前執行該命令並對線路進行初始化。

在成功地登錄之後，/bin/login執行用戶的shell（在/etc/passwd檔中指定，通常是/bin/sh或/usr/bin/ash）。shell等待用戶鍵入命令，並為每條命令創建一個新的進程。採用這種方式時，各個shell都是init的子進程，而使用者進程則是init的孫子進程，並且所有的使用者進程都是一棵進程樹的組成部分。

MINIX用於進程管理的兩條最重要的系統調用是FORK和EXEC。FORK是創建一個新進程的唯一途徑。EXEC允許一個進程執行一個指定的程式，當一個程式被執行時，將按照檔頭中指定的大小為其分配一部分記憶體。儘管在進程運行期間，資料段、棧段和空閒未使用部分的大小可以不時地改變，但進程分配到的記憶體總量將保持不變。

一個進程的所有資訊被保存在進程表中，進程表劃分成核心、記憶體管理器和檔案系統三部分，分別擁有它們各自所需要的那些域。當出現一個新進程（通過FORK），或者一個老進程結束（通過EXIT或信號）時，記憶體管理器首先更新它那部分進程表，然後向檔案系統和核心發送消息，以通知它們進行相應的操作。

2.5.3 MINIX中的進程間通信

MINIX提供了三條原語來發送和接收消息，它們均通過C庫常式調用。其中

send（dest，&message）

用來向進程dest發送一條消息，

receive（source，&message）

用來從進程source（或任何地方）接收一條消息，

send\_rec（src\_dst，&message）

用來發送一條消息，並等待同一個進程的應答。以上調用中第二個參數是消息資料的本地位址。核心中的消息傳遞機制將消息從發送者拷貝到接收者。應答消息（對於send\_rec）將覆蓋原先的消息。原則上該核心機制可以替換為另一套機制以實現分散式系統，即在網路上將消息從一台機器拷貝到另一台機器上。但在實踐中這很複雜，因為有時消息的內容可能是一個指向大型資料結構的指標，於是分散式系統也必須提供網路上資料本身的拷貝功能。

每個進程或任務都可以從/向同層和下一層中的進程或任務發送和接收消息，使用者進程不能直接與I/O任務通信，系統強制地執行這一限制。

當一個進程（作為特例，這裡也包括任務）向一個當前未在等待消息的進程發送一條消息時，發送者將阻塞，直到目標進程執行receive。換言之，MINIX使用會合的方法來避免對已發送而未接收到的消息進行緩衝的問題。儘管這沒有帶緩衝的方案靈活，但事實證明對MINIX來說它已經足夠了，而且由於不需要緩衝管理，所以簡單許多。

2.5.4 MINIX中的進程調度

中斷系統使多道程序作業系統持續不斷地工作。當進程請求輸入時，它們將阻塞以允許其他進程執行。當輸入可用時，當前運行進程被磁片、鍵盤或其他硬體中斷。時鐘也產生中斷，這種中斷使正在運行的未請求輸入的使用者進程最終放棄CPU，以使其他進程獲得運行的機會。MINIX最底層軟體的任務就是通過將中斷轉換成消息來對其加以隱藏。就進程（以及任務）而言，當一個I/O設備完成一個操作時，它向某些進程發送一條消息，將其喚醒並使之成為就緒。

每當一個進程被中斷時，不管中斷源是常規的I/O設備還是時鐘，都有機會重新確定哪個進程最需要運行機會。當然，在一個進程終止時也要執行該操作，但在類似MINIX這樣的系統中，由I/O操作和時鐘引起的中斷遠遠多於進程終止的情況。MINIX調度程式使用一個三級排隊系統，分別對應於圖2-26中的第2、3、4層。任務和伺服器一級的進程一直運行直到阻塞，而使用者進程則採用時間片輪轉調度。任務具有最高優先順序，記憶體管理器和檔案管理員次之，使用者進程最低。

當調度程式選擇一個進程來運行時，它首先檢查是否有就緒的任務，如果有一個或多個，則隊首的那個將運行。如果沒有任務就緒，則檢查並運行伺服器進程（MM或FS）。若沒有合適的伺服器進程，則運行一個使用者進程。如果沒有進程就緒，則選擇IDLE進程。這個迴圈一直執行到下一個中斷到來。

在每一個時鐘滴答，都將檢查當前進程是否是一個運行超過100毫秒的使用者進程。如果是，則調用調度程式來查看是否有另一個使用者進程在等待CPU，如果發現一個這樣的進程，則當前進程被移到佇列的末尾，而運行當前的隊首進程。任務、記憶體管理器和檔案系統不會被時鐘剝奪，不論它們已運行了多久。

2.6 MINIX中進程的實現

現在我們來說明實際代碼，先介紹一下將要用到的幾個術語。“過程”、“函數”以及“常式”這幾個詞可以混用。變數名、過程名以及檔案名將用斜體，如rw\_flag。當一條語句用變數名、過程名或檔案名開頭時，它將使用大寫，但真正的名字都使用小寫字母。系統調用使用小一點的大寫字母，如READ。

由於這本書和軟體是一直在不斷演變的，而不是在同一天交付出版，所以在代碼引用、列印出的代碼清單以及CD－ROM上的軟體之間可能會有小的出入。但這種差異一般只影響一、兩行代碼。本書中印刷出來的原始程式碼已經被簡化，從中刪去了那些與本書未討論的選擇項相關聯的部分。

2.6.1 MINIX原始程式碼的組織

原始程式碼從邏輯上分成兩個目錄。在一個標準的MINIX系統中，其完整的路徑分別為/usr/include/和/usr/src/ （一個綴後的“/”表示這是一個目錄）。目錄和真實路徑可能因系統而異，但一般來說最高層以下的目錄結構在所有系統中都是一樣的。本書中我們記作 include/ 和 src/。

include/ 目錄包含了許多符合POSIX標準的標頭檔，它又包含三個子目錄：

1 sys/ － 包含POSIX標頭檔

2 minix/ － 包含作業系統使用的標頭檔

3 ibm/ － 包含IBM PC特有定義的標頭檔

為了支援對MINIX和在MINIX環境下運行程式的擴展，在CD－ROM上同時提供了放在include/目錄下的其他檔和子目錄，這些也可在Internet上取到。例如， include/net目錄及其子目錄 include/net/gen 支援網路擴展。但本書中僅印出了編譯MINIX基本系統所需的檔，並且只討論這些內容。

src/ 目錄包含三個重要的子目錄，其中包括了作業系統的原始程式碼：

1 kernel/ － 第1層和第2層（進程、消息和驅動程式）

2 mm/ － 記憶體管理器代碼

3 fs/ － 檔案系統代碼

還有另外三個原始程式碼目錄在本書中未列出，也不加討論。但對於構造一個能夠工作的系統，它們是很重要的。

1 src/lib/ － 庫常式原始程式碼（如open，read）

2 src/tools/ － init程式原始程式碼，用於啟動MINIX

3 src/boot/ － 引導和安裝MINIX的代碼

MINIX的標準發佈軟體還包括另外幾個原始程式碼目錄。作業系統當然要支援在其上運行的命令（程式）。所以有一個很大的命令目錄src/commands/，其中包含公用程式（如cat，cp，date，ls，pwd）的原始程式碼。由於MINIX是一個用於教學的作業系統，這意味著對它常常要作修改，所以有一個 src/test/目錄包含有一些被設計用來對新編譯好的MINIX系統進行完整測試的工具。最後 /src/inet/目錄包含了重新編譯MINIX以使之支援網路的原始程式碼。

為了方便起見，在通過上下文能夠確定完整路徑的情況下，將只使用簡單的檔案名。但必須指出有效檔案名可能出現在不只一個目錄下。例如存在若干名為const.h的檔，其中定義了系統組成部分所用到的常數。一個目錄下的檔將放在一起討論，這樣就不會發生混淆。這些檔在附錄A中按照其在教材中討論的順序羅列，這樣更易於掌握。在這裡若使用一些書簽將很有説明。

同樣應該指出的是，附錄B包含附錄A中描述的所有檔的字母序列表。附錄C則包含一個索引，從中可查出MINIX所使用的巨集定義、全域變數以及過程的出處。

第1、2層的代碼位於 src/kernel/下。本章研究在該目錄下的一些檔，它們支援進程管理，這是圖2-26中所示的MINIX結構的最低層。該層包括完成以下功能的函數：系統初始化、中斷、消息傳遞以及進程調度。在第3章我們將研究該目錄下的其他檔，它們支持圖2－26中第2層的不同任務。在第4章我們將研究 src/mm/下的記憶體管理器。第5章將研究 src/fs/下的檔案系統。

編譯MINIX時，位於 src/kernel/、src/mm/和src/fs/下的所有原始檔案都被編譯成目的檔案。 src/kernel/下的所有目的檔案被連接成一個單一的可執行程式 kernel。在src/mm/中的目的檔案也被連結成一個可執行程式 mm，fs的情況也一樣。通過增加另外的伺服器進程可以達到擴展的目的。例如通過修改檔 include/minix/config.h便可以將 src/inet/下的檔編譯成 inet，從而增加網路支援。另一個可執行程式 init在 src/tools/ 目錄下生成。程式 installboot （其源碼在 src/boot/ 下）將名字加入上述程式中，通過填充以使其長度變成磁片磁區大小的整數倍（這樣可以在裝入這些不同部分時相互獨立），並且將它們拼接為一個單獨的檔。新檔就是作業系統的二進位映射，並且可被拷貝到根目錄、或一張軟碟的 /minix/ 目錄、或一個硬碟分區上。圖2－27示出了這些拼接的檔被分開和裝入後的記憶體佈局。當然，具體細節依賴於系統的配置。圖中的例子是一個為具有幾百萬位元組記憶體的機器配置的MINIX系統。這種配置可以分配大量的檔案系統緩衝區，但所得的大檔案系統無法被放在記憶體的低端部分，即640K以下的記憶體。如果大量地縮減緩衝區數目，則有可能使整個系統佔用不到640K的記憶體，還可以為幾個使用者進程留出空間。

圖 2-27 MINIX從磁片裝入記憶體之後的記憶體佈局。四個單獨編譯並被連結的部分

（若加上網路則為五部分）劃分得很清楚。圖中各部分的大小只是大概情況，

具體則依賴依賴於系統組態狀況。

很重要的一點是必須認識到MINIX包含三個或更多的獨立程式，其間通過消息傳遞進行通信。 src/fs/ 下的一個名為 panic的過程不會與 src/mm/ 下的 panic 過程衝突，因為它們最終被鏈入不同的可執行檔。作業系統的三個部分之間僅有的共同的常式是 lib/ 下的若干庫常式。這種模組化結構使得很容易做到對檔案系統的修改不會影響記憶體管理器。同時也可以做到從系統中去掉檔案系統，並將其放到另一台作為檔案伺服器的機器上，該檔案伺服器通過在網路上發送消息來與使用者機器通信。

作為MINIX模組性的另一個例子，編譯系統時是否帶網路支援不會對記憶體管理器有任何影響，而只會影響核心。因為乙太網系統任務及對其他I/O設備的支援，都在核心中編譯。當被啟動時，網路服務器被集成到MINIX系統成為一個伺服器進程。它與記憶體管理器和檔案伺服器具有相同的優先順序。網路服務器操作牽涉到大量資料的高速傳送，這需要比使用者進程更高的優先順序。不過除了乙太網任務外，網路功能也可以被使用者級進程執行。網路功能不是傳統的作業系統功能，對它的詳細討論超出了本書的範圍。後續章節的討論將基於一個不帶網路支援的MINIX系統。

2.6.2 公共標頭檔

include/ 及其子目錄包含了許多檔，其中定義了常量、宏、以及類型。POSIX標準需要其中的許多定義，並且指定了在 include/ 及其子目錄 include/sys 下其所需的每一個定義應放在哪一個檔中。這些目錄下的檔是標頭檔（header），或稱作包含檔，用尾碼 .h 標識。這些檔通過C來源程式中的 ＃include 語句引用，該語句是C語言的特性之一。包含檔使得更容易維護一個大系統。

編譯使用者程式時可能用到的標頭檔放在 include/ 目錄下；而傳統上 include/sys/ 目錄放那些編譯系統程式和公用程式所用的標頭檔。這種規定並不是絕對的，而且一個典型的編譯過程 － 無論是編譯使用者程式還是編譯作業系統的一部分，將同時包含這兩者下邊的檔。這裡我們要討論編譯標準的MINIX系統所用到的檔，先看 include/ 目錄，再看 include/sys/ 目錄。下一節將討論 include/minix/ 和 include/ibm/ 目錄下的所有檔。顧名思義，include/ibm/用於IBM類型的機器上的MINIX實現。

首先考慮的是真正通用的標頭檔，它們不直接被任何C原始檔案引用，而是被別的標頭檔包含。主控標頭檔 src/kernel/kernel.h，src/mm/mm.h 及 src/fs/fs.h 分別對應MINIX的三個主要部分，而這三部分必定包含在MINIX的每一個編譯版本中。每個頭檔都根據MINIX系統相應部分的需要進行剪裁，但每個檔都含有一個如圖2－28所示的起始部分。在本書的其他部分還將討論這些主控標頭檔。這裡只是強調各不同目錄下的標頭檔是一起使用的。在本節和下一節中我們將提到圖2－28中引用的每一個頭文件。

# include <minix/config.h> /\* 必須是第一個標頭檔 \*/

# include <ansi.h> /\* 必須是第二個標頭檔 \*/

# include <sys/types.h>

# include <minix/const.h>

# include <minix/type.h>

# include <limits.h>

# include <errno.h>

# include <minix/syslib.h>

圖 2－28 一個主控標頭檔的一部分，它保證了將那些所有C原始檔案都需要

的標頭檔包括在內。

討論從 include/ 目錄下的第一個標頭檔ansi.h 開始（0000行）。這是編譯MINIX系統任何一部分都要處理的第二個檔，第一個檔是 include/minix/config.h。ansi.h 的作用是測試編譯器是否符合ISO規定的標準C的要求。標準C也稱作ANSI C，因為在獲得國際承認之前它最早是由美國國家標準局開發的。一個標準C編譯器定義了若干宏，它們可在被編譯的程式中測試。\_\_STDC\_\_就是一個這樣的宏，而且它被一個標準編譯器定義為1，就像是C語言前置處理器讀入了如下的一行：

＃define \_\_STDC\_\_ 1

隨當前的MINIX版本發行的編譯器遵從標準C，但老版本的MINIX是在該標準被採納之前開發的。不過仍然可以用一個傳統的C編譯器（Kernighan & Ritchie）來編譯MINIX。MINIX希望能夠很容易地被移植到新的平臺上，允許使用老的編譯器正是該目標的一個部分。如果使用標準C編譯器，則將處理在0023到0025行之間的語句

# define \_ANSI

。Ansi.h 使用不同的方式定義了若干巨集，這具體依賴於宏 \_ANSI 是否定義。

Ansi.h 中最重要的宏是 \_PROTOTYPE。它允許用如下形式寫一個函數原型：

\_PROTOTYPE (return\_type function\_name, ( argument\_type argument, . . .) )

如果使用一個ANSI標準C編譯器，則將由C前置處理器將其轉換成：

return\_type function\_name( argument\_type, argument, . . . )

如果使用一個老式（即 Kernighan & Ritchie）編譯器，則將其轉換成：

return\_type function\_name( )

在結束 ansi.h 的討論前還要提及另一個特性。整個 ansi.h 檔被包括在兩行語句

＃ifndef \_\_ANSI\_H

和

# endif

之間。在 #ifndef \_\_ANSI\_H 的下面一行，\_\_ANSI\_H 緊接著被定義。在一次編譯中一個頭檔應該僅被包含一次。這種寫法保證了一個檔被包含多次時內容的正確性。我們將看到在 include/ 目錄下的所用標頭檔都使用這種技術。

include/ 目錄下第二個間接包含在每一個MINIX原始檔案中的標頭檔是 limits.h（0100行）。其中定義了許多基本的大小值，既有語言中的資料類型，如整數所占的位元數，也有作業系統的限制，如檔案名長度。errno.h （0200行）也被所有的主控標頭檔包含，它包含了系統調用失敗時從全域變數errno返回給使用者程式的錯誤碼。 errno也用來標識一些內部錯誤，如試圖向一個不存在的任務發送消息。在MINIX中，若返回值為負表示它是一個錯誤碼，但在返回使用者程式之前必須使它們為正值，這裡的技巧在於每個錯誤碼都在類似下面一行語句中定義：

# define EPERM (-SIGN 1)

（0236行）。作業系統各部分的主控標頭檔定義宏 \_SYSTEM，但在一個使用者程式被編譯後 \_SYSTEM 並未定義。如果 \_SYSTEM 被定義，則\_SIGN 定義為“-”，否則 \_SIGN無定義。

下面一組檔沒有被包含在所有的主控標頭檔中，但被MINIX所有三個部分的許多原始檔案使用。最重要的是 unistd.h （0400行）。該標頭檔定義了許多常量，其中多數是POSIX需要的。它也包含了許多C函數的原型，其中包括所有用於進行系統調用的C函數原型。 signal.h （0700行）定義了標準信號名，同時包含一些與信號相關的函數原型。隨後我們將看到，信號處理牽涉到MINIX的所有部分。

fcntl.h（0900行）採用符號方式定義了檔控制操作使用的許多參數。例如，它允許在 open 調用中使用宏 O\_RDONLY 來代替數值0作為參數。儘管該檔主要由檔案系統引用，但它的定義在核心和記憶體管理器中也多次用到。

include/ 中的其他檔不如上述檔用得多。stdlib.h （1000行）定義了大多數C程式，除最簡單的以外，編譯時要用到的類型、宏和函數原型。儘管在MINIX源碼中 stdlib.h 只被核心中少數幾個檔引用，它卻是編譯使用者程式時使用最頻繁的標頭檔之一。

正如在第3章的系統任務層將看到的那樣，作業系統的控制台和終端介面是很複雜的，因為各種不同類型的硬體必須通過一種標準的方式與作業系統和使用者程式交互。標頭檔 termios.h （1100行）定義了控制終端類型的I/O所用到的常量、宏和函數原型。最重要的資料結構是 termios 結構，它包含的內容有：標識操作模式的標誌位元、設置輸入輸出速率的變數、以及放置特殊字元的陣列（比如 INTR 和KILL）。termios是POSIX所需要的，同樣地，該檔中定義的許多宏和函數原型也是POSIX需要的。

然而，正如POSIX標準始終遵循的那樣，它並未提供可能用到的全部內容。在檔的後半部分，第1241行以後，提供了POSIX並不包含的擴展部分。其中有些擴展的意義是很明顯的。比如定義57,600或以上的串列傳輸速率，以及對終端顯示視窗的支援。正如任何合理的標準都是開放的那樣，POSIX標準並不排斥擴展。但在MINIX環境下想寫一個可移植到其他環境的程式時，就必須注意避免使用那些局限於MINIX的特徵。這一點很容易作到。在該檔和其他定義面向MINIX的擴展的檔中，這些擴展是通過以下語句控制的。

# ifdef \_MINIX

如果 \_MINIX未被定義，編譯器將根本不會感知到MINIX擴展。

include/ 下我們要討論的最後一個檔是 a.out.h （1400行）。它定義了可執行檔在磁片上的存儲格式，包括開機檔案執行的檔頭結構和編譯器產生的符號表。它只由檔案系統引用。

現在來看 include/sys/目錄。如圖2－28所示，MINIX系統各主要部分的主控標頭檔在包含 ansi.h 後隨即都包含 sys/types.h 文件（1600行）。types.h 中定義了許多MINIX使用的資料類型。使用這裡提供的定義可以避免對特定情況下所使用的基本資料結構的理解錯誤而導致的故障。圖2－29示出了其中定義的幾種資料類型，以及它們分別在16位和32位處理器上編譯時所占位數的差異。注意所有的類型名都以“\_t”結尾，這不僅僅是一種習慣，而是POSIX標準的規定。這是“保留尾碼”的一個例子，而且“\_t”不用於非資料類型的其他任何符號名。

圖 2－29 16位元和32位元系統上一些類型的大小，單位為比特。

儘管並不常用sys/ioctl.h （1800行），它也沒有被主控標頭檔包含，但其中還是定義了許多用於設備控制的巨集。它也包含了 IOCTL 系統調用的原型。在 include/termios.h 中提供了函數原型的許多POSIX函數已經取代了老的 ioctl 庫函數的許多功能，包括對終端、控制台及類似設備的處理，所以IOCTL系統調用在許多情況下並不直接被程式師調用，但該系統調用仍然是需要的。實際上，控制終端的POSIX函數被庫轉換成了IOCTL系統調用。而且，當前電腦設備的類型是不斷增多的，它們各自需要某種方式的控制。例如在該檔的結尾處有若干以DSPIO開頭的操作碼，它們就用來控制一個數位訊號處理器。實際上，本書中講述的MINIX和其他版本的MINIX之間的主要差別在於：為了突出本書的意圖，我們描述了一個隻帶有少量I/O設備的MINIX。其餘許多設備可被加進來，如網路介面、CD－ROM驅動器，音效卡等等。在本檔中它們的控制碼被定義為宏。

include/sys 下的若干其他檔在MINIX中得到廣泛使用。 sys/sigcontext.h （2000行）定義了一些結構，這些結構用來保存和恢復執行信號處理常式前後正常的系統操作。該檔用於核心和記憶體管理器。MINIX中提供了對跟蹤可執行檔和用調試器分析核心轉儲（core dump）的支持。 sys/ptrace.h （2200行）定義了使用 PTRACE 系統調用的各種可能操作。 sys/stat.h 檔（2300行）定義了我們在圖1－12中所看到的結構，它由 STAT 和FSTAT 系統調用返回。同時其中定義了 stat、fstat 以及其他一些用來對檔進行操作的函數的原型。在檔案系統和記憶體管理器中有幾處將引用該文件。

最後兩個檔沒有上述檔用得多。 sys/dir.h （2400行）定義了MINIX的目錄項結構。 dir.h 只被直接引用了一次，但包含它的檔在檔案系統中被廣為使用。它之所以重要是因為它定義了一個檔案名最長有多少個字元。最後 sys/wait.h （2500行）定義了WAIT和WAITPID系統調用所使用的巨集，它們本身在記憶體管理器中實現。

2.6.3 MINIX標頭檔

include/minix/ 和 include/ibm/ 目錄包含了MINIX特有的標頭檔。 include/minix/ 中的檔對任何平臺上的MINIX實現都是需要的（儘管其中有少數定義存在與平臺相關的替代物）。include/ibm/ 中定義了IBM 類型機器上的MINIX實現所特有的資料結構和巨集。

先從 minix/ 目錄開始。在前一節中我們注意到 config.h （2600行）被MINIX所有部分的主控標頭檔所包含，並且它是編譯器實際上處理的第一個檔。在許多情況下，當因硬體差異或對作業系統的使用方式不同而需要對MINIX的配置作修改時，只需要編輯該檔並對系統重新編譯即可。用戶可設置參數均放在該檔的第一部分。其中第一個是參數MACHINE，根據要在哪個平臺上編譯MINIX，它的值可以是IBM－PC、SUN－4、MACINTOSH或其他值。MINIX的大部分代碼是獨立於機器類型的，但一個作業系統總有一部分代碼與硬體相關。在書中有少數幾處所討論的代碼是對硬體而變的，我們使用的例子是IBM PC 兼容機，它使用32位元的高檔處理器晶片（80386，80486，奔騰或高能奔騰）。我們將其都看作32位元Intel晶片。MINIX也可以在老式的16為字長的IBM PC 上編譯，當然相應的機器相關部分要重寫。在一台PC上，編譯器自行決定MINIX應針對哪種機器進行編譯。PC上標準的MINIX編譯器是Amsterdam Comiler Kit （ACK）編譯器，它通過在 \_\_STDC\_\_之外再定義一個巨集 \_\_ACK\_\_來標識自己。同時它還定義一個宏 \_EM\_WSIZE，以指定目的機的字長（以位元組為單位）。在2626到2628行之間，宏\_WORD\_SIZE 被賦值為 \_EM\_WSIZE。在該檔的後邊以及其他MINIX原始檔案中有幾處都用到這幾個定義，例如第2647到2650行以如下的測試語句開始：

# if (MACHINE == IBM\_PC && \_WORD\_SIZE == 4 )

然後定義了一個32位元系統上檔案系統高速緩衝區的大小。

Config.h 檔中的其他定義允許在特殊安裝中進行其他需求的定制，例如，其中有一段允許編譯MINIX核心時裝入各種不同的設備驅動程式。這可能是MINIX原始程式碼中被改動得最多的部分。這一段的開頭是：

# define ENABLE\_NETWORKING 0

# define ENABLE\_AT\_WINI 1

# define ENABLE\_BIOS\_WINI 0

通過將第1行中的0改成1，可以將MINIX編譯成帶有網路支援。將ENABLE\_AT\_WINI定義成0和將ENABLE\_BIOS\_WINI定義成1，可以刪去AT類型（即IDE）的硬碟驅動程式代碼而使用PC BIOS來作為硬碟支援。

下一個檔是 const.h （2900行）。通過它可以理解標頭檔的另一種普遍用法。在這裡我們發現一些常量定義，在編譯一個新核心時它們不大會改變，但卻在許多地方用到。通過這些定義我們可以防止發生變數在多處定義不一致的錯誤，而且這種錯誤很難發現。在MINIX原始程式碼目錄樹中還有另外幾個檔也命名為 const.h，但其使用很有限。只在核心中使用的定義都包含在 src/kernel/const.h中。記憶體管理器將 src/mm/const.h 用於其局部定義。只有那些在MINIX系統一個以上的部分中用到的定義放在 include/minix/const.h 中。

const.h 中有幾處定義值得注意。EXTERN被定義為 extern （2906行）。在標頭檔中定義並被兩個以上的檔包含的全域變數被聲明為EXTERN類型，形式如下：

EXTERN int who;

如果變數採用以下方式聲明，

int who;

並被包含於兩個以上的檔中，則有些連結程式將會把它認作變數多重定義錯誤。而且，C語言參考手冊（Kernighan & Ritchie, 1980）顯式地禁止這種語法形式。

為了避免這類問題，必須在除一處之外的所有其他地方都寫成

extern int who;

在包含了 const.h之處EXTERN將被代換成 extern，而在定義該變數之前顯式地將EXTERN重新定義為空串，那麼這一問題便解決了。實現方法是：在MINIX的每一部分中，將全域定義放在一個指定檔－glo.h 中。例如 src/kernel/glo.h，該檔被間接地包含在每一個編譯版本中。在每個 glo.h 中有如下語句：

# define \_TABLE

# undef EXTERN

# define EXTERN

# endif

並且在MINIX每一部分的 table.c 檔中，在＃include 之前都有一行

# define \_TABLE

這樣，當標頭檔被包含進table.c 並展開成為其中的一部分時， extern 絕對不會出現在該檔中的任何地方（因為在 table.c 中EXTERN被定義為空串），並且全域變數的空間只被預留在一個地方，即 table.o 中。

如果你是C程式設計的新手，對上述內容不太理解，那麼其中的細節也並不特別重要。對有些連結程式而言，標頭檔的多次包含可能會導致一些問題，因為它可能引起被包含變數的多重聲明。上述EXTERN用法只是增強MINIX移植性的一種途徑，因為這樣一來，即使在連結程式不支援變數多重定義的機器上，MINIX也能夠被連結成功。

PRIVATE被定義為 static 的同義詞。對於那些不會在定義檔以外被引用的過程和資料，它們往往被聲明為PRIVATE以使其名字在文件之外不可見。作為一條通用的規則，所有的變數和過程都應被聲明成在盡可能局部的範圍有效。PUBLIC 被定義為空串，於是如下的聲明

PUBLIC void free\_zone ( Dev\_t dev, zone\_t numb )

經C前置處理器處理後成為

void free\_zone ( Dev\_t dev, zone\_t numb )

按照C語言的作用域規則，該語句意味著符號 free\_zone 從該檔輸出，並能被其他檔使用。PRIVATE和PUBLIC並不是必須的，但使用它們是為了儘量消除C語言作用域規則的副作用（C語言中一個符號名缺省是輸出到檔之外，但實際上應該正好相反）。

const.h 的其餘部分定義了在整個系統中使用的數值常量。其中有一段專門定義與機器或配置相關的內容。例如在全部的源碼中，記憶體大小的基本單位是click（塊）。塊的具體大小取決於處理器結構。對於Intel相容晶片、Motorola 68000、Sun SPARC等體系結構，click定義在2957到2965行。該檔中還包括了宏MAX和 MIN，因此 z=MAX（x, y）的功能是將 x和y中的較大值賦給z。

type.h （3100行）是另一個通過主控標頭檔被包含在每一個編譯版本中的檔。它包括了許多重要的類型定義，以及相關的數量詞。該文件中最重要的定義是3135到3146行的message。我們當然可以將message定義為一個包含若干位元組的陣列，但在程式設計實踐中最好將其處理為一個結構，其中包含有若干不同的可能的消息類型的聯合。消息定義了6種格式：mess\_1到mess\_6。一條消息包括如下的幾個部分：m\_source域，標識誰發送了該消息；m\_type域，標識消息類型（例如，發給時鐘任務的GET\_TIME）;以及資料欄。六種消息類型示於圖2－30。該圖中第1、第2種看起來很相似，第5、第6種也很相似。這種情況對於在32位元的Intel晶片上實現的MINIX成立，但對於在一台int、long和指針類型佔有不同空間的機器上則未必。定義6種消息類型將使得更易於在不同的體系結構上重新編譯。

圖 2-30 MINIX中使用的六種消息類型。消息元素的長度將隨機器結構而變化；圖中所示的是在32位指針機器，如Pentium (Pro)上的長度。

當需要發送一條消息，而其中包含3個整數和3個指標（或3個整數和2個指標）時，應使用圖2－30中的第1種格式。對其他格式依此類推。如何給第1種格式中的第一個整數賦值呢？假設這裡的消息為x，則x.m\_u指向該消息結構的聯合部分。我們使用x.m\_u.m\_m1來指向6種格式中的第1種。最後，我們用x.m\_u.m\_m1.m1i1來指向該結構中的第1個整數，這種使用方法相當麻煩，所以在定義了消息之後，一般都定義一個短一點的名字作為其相應的宏。於是用x.m1\_i1來代替x.m\_u.m\_m1.m1i1。這種短名字的格式都由以下幾部分組成：字母m，格式編號，一個底線，一個或兩個類型字母，它用來指明該域是一個整數、指標、長整數、字元、字串或是函數，以及一個序號，在一條消息中包含多個相同類型的域時用該序號對它們加以區分。

在討論消息格式時，順便指出作業系統及其編譯器通常對某些事物有共同的“理解”，比如一個結構的佈局等。而且這種共識簡化了系統的構造。在MINIX的消息中，int域有時用來存放 unsigned資料類型。在某些情況下這會導致溢出，但編碼時我們知道，MINIX編譯器只機械地將 unsigned 類型和 int 類型之間來回拷貝，而不進行溢出檢查。一種更規範的寫法是將每一個 int 域寫成 int 和 unsigned 的聯合。在消息中對long 也同樣處理，有些 long 類型的域被用來傳送 unsigned long 型資料。這樣做是不是有點騙人的感覺？有人也許這麼認為，但如果你要將MINIX移植到一個新平臺上，很顯然必須對消息的確切格式給予足夠的注意，而且在這裡你已經收到警告，編譯器的特性是另一個必須注意的因素。

include/minix 下還有另一個通過主控標頭檔被廣泛使用的檔，即 syslib.h （3000行）。它包含了在作業系統內部調用以訪問作業系統其他服務的C庫函數原型。本書不詳細討論C庫，但其中有許多是標準的，且對任何C編譯器都可用。然而，syslib.h 引用的C函數顯然是特定於MINIX的；並且，將MINIX移植到一個帶有不同編譯器的新系統將需要移植這些庫函數。幸運的是這並不難，因為這些函數只是抽取出函式呼叫的參數，將其插入一條消息結構中，然後發送該消息，接著從應答消息中將結果抽取出來。這中間許多庫函數的定義僅有十幾行甚至更少的C代碼。

當一個進程需要執行一條MINIX系統調用時，它向記憶體管理器（MM）或檔案系統（FS）發送一條消息。每條消息中含有所要求的系統調用序號。這些序號在下一個檔 callnr.h （3400行）中定義。

文件 com.h （3500行）主要包含從MM和FS發送到I/O任務的消息中所使用的公共定義。其中也定義了任務序號。為了將其與進程號分開，任務號是負數。該檔還定義了可被發送到每個系統任務的消息類型（函數代碼），例如，時鐘任務接收代碼SET\_ALARM（用於設置計時器）、CLOCK\_TICK（當時鐘中斷發生時）、 GET\_TIME（取真實時間）、 SET\_TIME（設置一天中的當前時間）、 REAL\_TIME 的值就是對GET\_TIME請求所返回的消息類型。

最後，include/minix/還包含了幾個更特殊的標頭檔。其中有boot.h （3700行），它被核心和檔案系統用來定義設備、以及訪問從 boot 程式傳入系統的參數。另一個是 keymap.h （3800行），它定義了若干結構，這些結構用來實現不同語言所需的字元集對應的特殊鍵盤配置。它也被那些生成和載入這些表格的程式使用。這裡的某些檔，如 partition.h （4000行）只被核心使用，而不被檔案系統或記憶體管理器使用。對於一個支援額外的I/O設備的實現，這裡有另外一些類似的檔，它們分別支援其他的設備。這些設備在該目錄下的位置需要解釋一下。理想情況下，所有的使用者程式應該僅僅通過作業系統來訪問設備，這些支援外部設備的檔應當放在 src/kernel/ 下。但是系統管理的實際情況要求，某些使用者命令能訪問系統級的結構，例如對硬碟進行分區的命令。正是為了支援此類公用程式，我們將這些特殊標頭檔放在 include/ 目錄樹下。

include/ibm 是這裡討論的最後一個特殊標頭檔目錄，它下面有兩個檔包含與 IBM PC系列機相關的資訊。一個是 diskparam.h ，它被軟碟系統任務使用。儘管該任務屬於標準的MINIX版本，但本書不詳細討論其原始程式碼，因為它和硬碟任務很類似。另一個檔是 partition.h （4100行），它定義了IBM 兼容機上使用的硬碟分區表和相關的常量。該檔有助於將MINIX移植到其他硬體平臺上。對於不同硬體， include/ibm/partition.h 必須可被替換為通常位於另一個相應目錄下的 partition.h 文件。但在 include/minix/partition.h 中定義的結構是MINIX內部使用的，它應對各種硬體都保持不變。

2.6.4 進程資料結構和標頭檔

現在來看 src/kernel/ 下的代碼。前兩節中的討論是圍繞一個典型的主控標頭檔進行的，這裡我們先來觀察核心的主控標頭檔 kernel.h （4200行）。它一開始先定義了3個宏。第1個 \_POSIX\_SOURCE是POSIX標準自行定義的一個特徵檢測巨集。所有這類宏都必須以一個底線“\_”開頭。定義 \_POSIX\_SOURCE 的作用是保證所有POSIX要求的符號和那些顯式地允許但並不要求的符號將是可見的，同時又隱藏掉任何屬於POSIX非官方擴展的附加符號。我們已經提到過接下來的兩個定義： 宏\_MINIX將為MINIX所定義的擴展而重載\_POSIX\_SOURCE的作用；而在編譯系統代碼時，如果要作與用戶代碼不同的事情，比如改變錯誤碼的符號，則可以對 \_SYSTEM宏進行測試。kernel.h隨後包含一些 include/ 及其子目錄 include/sys/ 和 include/minix/下的其他標頭檔，其中包括圖2－28中的所有檔，這些檔在前兩節中已經作過討論。最後將包含本地目錄src/kernel/ 下的另外四個標頭檔。

這裡可以向初學C語言的讀者說明 # include 語句如何引用檔案名。每個C編譯器都有一個查找包含檔的缺省目錄。在標準的MINIX中這通常是 /usr/include。當檔案名置於一對小於號和大於號（“<...>”）之間時，編譯器將到缺省包含目錄或其下的一個指定的子目錄去找包含檔。當檔案名置於雙引號（“...”）之間時，首先在目前的目錄（或一個指定目錄）下查找，如果未找到，則再到缺省目錄下找。

通過在所有其他核心原始檔案中寫這樣一行簡單的語句

# include "kernel.h"

便可以使所有原始檔案共用大量重要的定義。由於標頭檔的包含次序有時非常重要，kernel.h 同時還一次性地保證了這種次序的正確性。這等於將標頭檔“作對一次，然後便可忽略具體細節”的思想帶到了一個新高度。對檔案系統和記憶體管理器也存在類似的主控標頭檔。

下麵繼續看 kernel.h 中的四個本地頭文件。正如在公共標頭檔目錄 include/minix/ 下有 const.h 和 type.h 一樣，在 src/kernel/ 目錄下也有 const.h 文件和 type.h 文件。 include/minix/ 下的檔之所以被放在那裡是因為系統的許多部分，包括在系統控制下運行的程式都需要它們。 src/kernel/ 下的檔提供了僅為編譯核心所需的定義。FS和MM原始檔案目錄也包含有 const.h 和 type.h 檔，同樣這些檔定義了僅為系統相應部分使用的常量和類型。在主控標頭檔中包含的另外兩個檔 proto.h 和 glo.h 在主 include/ 目錄下沒有對應的檔。但我們將發現它有編譯FS 和 MM 時使用的對應檔。

const.h （4300行）包含有許多機器相關的值，這些值用於Intel的CPU晶片，但在別的硬體上編譯時則可能不同。這些值定義在第4302到4396行之間的語句中，這些語句被包括在以下兩條語句之中：

# if ( CHIP == INTEL )

和

# endif

當在一種 Intel 晶片上編譯MINIX時，宏CHIP 和INTEL 將被定義，並在include/minix/config.h （2763行）中設置成相應的值，於是與機器相關的代碼將被編譯。當MINIX被移植到基於Motorola 68000的系統時，可以通過加入以下代碼

# if ( CHIP == M68000 )

和

# endif

同時在 include/minix/config.h 中將定義晶片類型的語句改為

# define CHIP M68000

而實現系統的移植。通過這種方式，MINIX可以處理那些與平臺相關的常量和代碼。這種語法結構對代碼的可讀性沒有什麼好處，因此應當儘量少用。實際上，為了保證可讀性，本書中印出的代碼已經刪節了許多針對68000和其他處理器的代碼。但CD－ROM和Internet 上的代碼則予以保留。

const.h中有一些定義值得特別的注意。其中有一些是機器相關的，例如重要的中斷向量和用在每次中斷後重定中斷控制器晶片的一些域的值。核心中的每個系統任務都有自己的堆疊，但在處理中斷時，將使用一個大小為 K\_STACK\_BYTES （定義於第4304行）的特殊的堆疊。這也定義在機器相關部分，因為不同的體系結構可能需要不同的堆疊空間。

其他定義是機器無關的，但它們被核心代碼的許多部分用到。例如，MINIX進程調度程式有 NQ（3）個優先順序佇列，分別命名為 TASK\_Q（最高優先順序）、SERVER\_Q（中等優先順序）和USER\_Q（最低優先順序）。使用這三個符號名是為了使原始程式碼更容易理解，但這三個宏所定義的數值則被編譯到執行程式中。最後，const.h 的最後一行將 printf 定義為一個宏，其真實值為 printk。這樣核心就可以使用核心內部定義的過程將出錯資訊之類的消息列印在控制台上。這個過程旁路了通常的機制。通常的機制需要從核心傳送消息到檔案系統，隨後又從檔案系統到印表機任務。當系統發生故障時，這套機制可能會失效。在這種情況下，我們將看到在一個核心過程panic中調用printf，也即 printk。正如你猜測的那樣，panic 檢測到嚴重錯誤時被調用。

文件 type.h （4500行）定義了幾個所有MINIX實現都要用到的原型和結構。tasktab結構定義了tasktab陣列中每一項的結構，memory結構（第4513到4516行）定義了唯一確定一片記憶體區域的兩個數值。這裡正好可以提到記憶體管理的幾個概念。塊是記憶體管理的基本單位，針對Intel晶片的MINIX中，一個塊是256個位元組。記憶體按phys\_clicks或vir\_clicks進行計量，phys\_clicks被核心用來存取系統中任何地方的存儲單元；vir\_clicks則被核心之外的進程使用。一個vir\_clicks存儲引用總是與分配給一個特定進程的記憶體段的基底位址相關，而且核心經常需要在這兩者之間來回轉換。由於一個進程可以使用vir\_clicks來引用其全部記憶體，這使得這種不便被部分地抵消。有人可能會設想使用同一種單位來指定這兩種類型記憶體的大小，但使用vir\_clicks更好一點，其原因是使用vir\_clicks時將進行檢查，以保證不會訪問超出當前進程之外的記憶體位址。這是現代Intel處理器（如奔騰和高能奔騰）保護模式(protected mode)的主要特徵之一。由於早期的8086和8088處理器不具備這種特徵，造成MINIX早期版本的設計者十分頭痛。

type.h還包含幾個機器相關的類型定義，如用於Intel處理器上的port\_t、segm\_t和reg\_t類型（第4525到4527行），它們分別用來訪問I/O埠、記憶體段和CPU寄存器。

資料結構也可能與機器相關。第4537到4558行為Intel處理器定義了stackframe\_s結構，它定義了如何將寄存器值保存到堆疊上。這個結構非常重要－在進程被投入運行狀態或被脫開運行狀態（如圖2－2所示）時，它被用來保存和恢復CPU的內部狀態。將其定義成可以用組合語言高速讀寫的格式，這將減少進程上下文切換的時間。segdesc\_s是與Intel處理器結構相關的另一個結構，它是保證進程不會發生記憶體訪問越限機制的一部分。

為了解釋不同平臺之間的差異，在該檔中保留了幾個針對Motorola 68000系列處理器的定義。Intel處理器族的晶片中有些使用16位元寄存器，有些使用32位寄存器，所以對於Intel體系結構，基本的reg\_t類型是無符號型。對於Motorola處理器reg\_t定義為u32\_t類型。這些處理器也需要一個stackframe\_s結構（第4583到4603行），但為了使相應的彙編代碼操作速度儘量地快，其佈局是不一樣的。Motorola體系結構根本不需要port\_t、segm\_t類型，也不需要segdesc\_s結構。同時也存在幾個專為Motorola體系結構定義的結構，它們沒有Intel晶片的對應物。

下一個檔proto.h（第4700行）是我們看到的最長的標頭檔。所有那些必須在其定義所在檔外被感知的函數的原型都放在proto.h中。它們都使用了前一節中提到的 \_PROTOTYPE技術，這樣，MINIX核心便既可以使用傳統的C編譯器（由Kernighan和Richie定義），例如MINIX初始提供的編譯器；又可以使用一個現代的ANSI標準C編譯器，例如MINIX版本2中所帶的編譯器。這其中的許多函數原型是與系統相關的，包括中斷和異常處理常式以及用組合語言寫的一些函數。本書中未討論的那些設備驅動程式所使用的函數的原型未被列出。針對Motorola處理器的條件編譯代碼也已從本文件及後面討論的文件中刪去。

包含在主控標頭檔中的最後一個核心標頭檔是glo.h（第5000行）。其中包含了核心的全域變數。宏EXTERN的作用已在檔include/minix/const.h的討論中說明。它通常被展開成extern。注意glo.h中的許多定義都以該宏開頭。當該檔被包含在定義了宏\_TABLE的 table.c中時，宏EXTERN的定義被強行取消。將glo.h包含在其他C原始檔案中將使table.c中的變數定義為核心中的其他模組所感知。held\_head 和held\_tail（第5013和5014行）是指向一個掛起的中斷佇列的指標。proc\_ptr（第5018行）指向當前進程的進程表項。當發生一條系統調用或中斷時，它指明應將寄存器值和處理器狀態保存在何處。sig\_procs（第5021行）記錄那些有信號等待發送給記憶體管理器進行處理的進程的數目。glo.h中有幾項用extern定義，而不是用EXTERN。其中包括sizes － 由引導監控程序填入的一個陣列、任務表tasktab、以及任務棧t\_stack。最後兩個是經初始化過的變數，這是C語言的特徵之一。宏EXTERN的使用與C風格的初始化並不相容，因為一個變數只可被初始化一次。

每個任務在t\_stack中有其自己的堆疊。在中斷處理期間，核心使用另外一個獨立的堆疊，因為只有執行中斷處理的組合語言級的常式訪問這一堆疊，因而不需要被全域範圍所感知，所以不在這裡對它作聲明，

還有另外兩個核心標頭檔，雖然不如包含在kernel.h中的那些用得多，但還是用得很廣泛的。第一個是proc.h（第5100行）。它將一個進程表項定義為一個結構proc（第5110到5148行）。在proc.h的後邊，它將進程表本身定義為proc結構的陣列 proc[NR\_TASKS + NR\_PROCS]（第5186行）。在C語言中，識別字的這種重用是允許的。宏NR\_TASKS定義在include/minix/const.h中（第2953行），而NR\_PROCS則定義在include/minix/config.h中。（第2639行）。它們共同決定進程表的大小。NR\_PROCS可以被改變以創建一個可以處理大量使用者的系統。因為進程表的訪問非常頻繁，並且計算陣列中的一個位址需要用到很慢的乘法操作，所以使用一個指向進程表項的指標陣列pproc\_addr（第5187行）來加快操作速度。

每個表項都包含有足夠的存儲空間來保存進程的寄存器值、堆疊指標、狀態、記憶體映射、堆疊長度、進程標識數、計費資訊、時間鬧鐘以及消息等資訊。每個進程表項的第一部分是一個stackframe\_s結構。進程被投入運行是通過將其進程表項的位址裝入其堆疊指標，然後從該結構中彈出全部的CPU寄存器值而實現的。當發送消息進程由於目標進程未處在等候狀態而無法完成一個SEND操作時，它被送到一個佇列中，該佇列由目標進程的一個p\_callerq域（第5137行）指向。於是，當目標進程最終執行RECEIVE操作時，它很容易找到等待向其發送消息的所有進程。p\_sendlink域（第5138行）用於將佇列中的成員連結起來。

當一個進程執行RECEIVE操作，但是沒有任何消息在等待它接收時，該進程將阻塞，同時將它期望接收消息的源進程序號保存在p\_getfrom中。訊息緩衝區的位址保存在p\_messbuf中。每個進程表項的最後三個域是p\_nextready、p\_pending和p\_pendcount（第5143到5145行）。p\_nextready用於將進程連結在調度程式佇列中，p\_pending是一個點陣圖，用於記錄那些尚未被傳送到記憶體管理器的信號（因為記憶體管理器沒有在等待一條消息）。p\_pendcount域是這些信號的計數值。

p\_flags域中的標誌位元定義了每個表項的狀態。如果其中任一位元被置位元，則進程將無法運行。各種標誌被定義和描述在第5154到5160行。如果該表項未被使用，則P\_SLOT\_FREE被置位。在執行一個FORK操作後，如果子進程的記憶體映射尚未建立起來，那麼NO\_MAP將被置位元以阻止子進程運行。SENDING和RECEIVING表示該進程被阻塞，其原因是它正在試圖發送或接收一條消息。PENDING和SIG\_PENDING表示已接收到信號。P\_STOP則在調試期間為跟蹤提供支援。

提供宏proc\_addr（第5179行）是因為在C語言中下標不能為負數。在邏輯上，陣列proc應從 －NR\_TASKS到＋NR\_PROCS。但在C語言中下標必須從0開始，所以proc[0]指向進程表項下標最小的任務，其他也依次類推。為了更便於記錄進程表項與進程之間的對應關係，我們可以使用

rp = proc\_addr(n);

將進程n的進程表項位址賦給rp，無論它是正還是負。

bill\_ptr（第5191行）指向正在對其CPU使用計費的進程。當一個使用者進程調用檔案系統，而檔案系統正在運行時，proc\_ptr（在glo.h中）指向檔案系統進程，但是bill\_ptr將指向發出該調用的使用者進程。因為檔案系統使用的CPU時間被作為調用者的系統時間來計費。

兩個陣列rdy\_head和rdy\_tail用來維護調度佇列。例如，rdy\_head[TASK\_Q]指向任務佇列中的第一個進程。

另一個被許多原始檔案包含的標頭檔是protect.h（第5200行）。該檔中幾乎所有的內容都與支援保護模式的Intel處理器（80286，80386、80486、奔騰、高能奔騰）體系結構的細節有關。對這些晶片的詳細描述不屬於本書的範圍。簡單地說，這些晶片都包含一些指向記憶體中描述符表(descriptor tables)的內部寄存器。描述符表定義了系統資源是如何使用的，並防止進程訪問屬於其他進程的記憶體區域。此外處理器提供了四種特權級(privilege levels)，MINIX使用其中的三種，它們的符號定義位於第5243到5245行。核心的最中心部分，即運行於中斷處理期間的部分和切換進程的部分運行在INTR\_PRIVILEGE特權級。在該特權級上，進程可以訪問全部的記憶體空間和CPU的全部寄存器。系統任務運行在TASK\_PRIVILEGE特權級。該特權級允許它們訪問I/O，但不能使用那些修改特殊寄存器（如指向描述符表的寄存器）值的指令。伺服器進程和使用者進程運行在USER\_PRIVILEGE特權級。運行在該特權級的進程不能執行某些指令，如訪問I/O埠、改變記憶體分配狀況，或改變處理器運行級別等等。特權級的概念對於熟悉現代CPU體系結構的讀者很容易理解，但對於那些通過組合語言來學習體系結構，或是只學習過低檔微處理器的讀者，他們可能從未遇到過此類運行級別的限制。

在核心目錄下還有幾個其他的標頭檔，但這裡只提及其中兩個。第一個是sconst.h（第5400行）。它包含彙編代碼所使用的常量。這些常量都是相對進程表項中stackfram\_s部分的偏移，其表示方式為彙編器可使用的一種格式。因為彙編代碼不由C編譯器處理，所以將其放在單獨的檔中將更簡單。同樣，由於這些定義均與機器相關，將其隔離出去將簡化MINIX向另一處理器的移植。該處理器將需要另一個版本的sconst.h。注意許多偏移被表示為前一值加上字母W，這裡W的值在第5401行被設為字長。這樣處理允許同一個檔被編譯成16位或32位版本的MINIX。

這裡有一個潛在的問題。標頭檔被假設為允許一個人提供一套正確的定義，隨後可以在許多地方使用而不必過多地注意細節。顯然，類似sconst.h中的重複定義違反了該原則。當然sconst.h只是一個特例，但在這種情況下，若修改sconst.h 或proc.h，必須保證其間的一致性。

最後一個檔是assert.h（第5500行）。POSIX標準要求必須有assert函數，它可以用來進行運行時測試、終止一個程式同時列印出一條消息。事實上，POSIX要求include/ 目錄下必須有一個assert.h 文件。那麼為什麼此處有另一個版本呢？答案是當使用者進程出錯時，可以依靠作業系統提供某些服務，比如在控制台上列印一條消息。但如果核心本身出錯，則正常的系統資源就未必靠得住。於是核心提供其自己的常式來處理 assert並列印消息，它獨立於通常的系統庫中的版本。

在kernel/ 目錄下還有幾個頭檔我們沒有討論過。它們支援I/O系統任務，我們將在下一章中合適的地方描述。但在進入可執行代碼之前，我們先看看table.c（第5600行）。其編譯生成的目的檔案將包含所有的核心資料結構，我們已經看到許多這類資料結構定義在glo.h 和proc.h中。在第5625行定義了宏\_TABLE，正好位於 #include語句之前。正如前面所解釋的，該定義導致EXTERN被定義為空串，於是為EXTERN之後的所有資料聲明分配存儲空間。除了 glo.h 和 proc.h中的結構以外，tty.h中定義的由終端任務使用的幾個全域變數也都在這裡分配存儲空間。

除了在標頭檔中聲明的變數之外，還有另外兩個地方可以為全域資料分配存儲空間。某些定義在table.c中直接進行定義。第5639到5674行為每個任務分配堆疊空間。對每個可選的任務，對應的宏ENABLE\_XXX（定義在檔 include/minix/config.h中）用來計算堆疊大小。於是若一個任務未被啟動則將不為它分配空間。按照這個規則，各種巨集ENABLE\_XXX用來確定各個可選的任務是否被表示在tasktab陣列中，如前邊src/kernel/type.h（第5699到5731行）中所聲明的，該陣列是由tasktab結構組成的。不論它是系統任務、伺服器進程還是使用者進程，例如init，對於每一個系統初始化期間啟動的進程都對應有一項。陣列的下標顯式地將任務號映射到相應的啟動過程。tasktab確定每一個進程所需的堆疊空間並為每個進程提供一個標識串。將tasktab放在這裡而不放在一個頭檔中，其原因是前述防止多重聲明的EXTERN技術對初始化的變數不起作用，也即任何時候都不能採用如下的寫法

extern int x=3;

使用前邊的堆疊大小定義可為所有系統任務分配堆疊空間（第5734行）。

雖然已經儘量將所有使用者可設置的配置消息單獨放在include/minix/config.h中，但是在將tasktab的大小與NR\_TASKS相匹配時仍可能會導致錯誤。在table.c的結尾處使用一個小技巧對這個錯誤進行檢測。方法是在這裡聲明一個dummy\_tasktab，聲明的方式是假如發生了前述的錯誤，則dummy\_tasktab的大小是非法的，從而導致編譯錯誤。由於啞陣列聲明為extern，此處並不為它分配空間（其他地方也不為其分配空間）。因為在代碼中任何地方都不會引用到它，所以編譯器不會受任何影響。

另一個分配全域空間的地方是在彙編代碼檔mpx386.s（第6483行）的末尾。在標號\_sizes處的空間分配將在核心資料段的最前邊放置一個魔數（籍此來標識一個合法的MINIX核心）。此處通過 .space虛擬指令來分配其他的空間。組合語言程式通過這種方式來預留空間使得可以強制地將 \_sizes陣列在物理上位於核心資料段的開始，這樣，boot程式就可以很容易地將資料放在正確的地方。引導監控程序讀取這個魔數，如果魔數正確，它將覆蓋之並用MINIX系統各部分的空間大小來初始化 \_sizes陣列。在初始化期間核心將使用這些資料。在啟動時，從核心看來，這是一個初始化了的資料區域。但是，核心最終在那裡找到的資料在編譯時還無法獲得。這些資料是在核心被啟動之前由引導監控程序補上的。這個過程是不大常見的，通常情況下，不會要求一個程式知道另一個程式的內部結構。從上電開始到作業系統正常運行這段時間是相當不平常的，要求使用不常用的技術。

2.6.5 引導MINIX

現在差不多可以來看執行代碼了。但在此之前先來搞清楚MINIX是怎麼被裝入記憶體的。當然MINIX是從硬碟上裝入的，圖 2－31示出了軟碟和分過區的硬碟的佈局。

圖 2－31 引導使用的磁片結構。 （a）未分區的磁片，第一磁區就是啟動區。

（b）分過區的磁片，第一個磁區是主引導記錄。

當系統啟動時，硬體（實際是ROM中的一個程式）讀取引導磁片的第一個磁區，並執行從那裡得到的代碼。在一個未分區的MINIX軟碟上，第一個磁區是一個啟動區，由它裝入引導程式，如圖2－31（a）所示。硬碟是分過區的，第一個磁區上的程式取同樣位於第一個磁區中的分區表，並裝入和執行活躍分區的第一個磁區，如圖2－31（b）所示（通常有且只有一個分區被標識為活躍）。一個MINIX分區與一個未分區的MINIX軟碟結構相同，其中有一個裝入引導程式的啟動區。

真實情況可能較圖示的略複雜一些，因為一個分區可能包含子分區。這種情況下分區的第一個磁區就是另外一個包含子分區表的主引導記錄。但最終控制權將傳到一個引導磁區，即一個不再被進一步細分的設備的第一個磁區。對軟碟，其第一個磁區總是一個引導磁區。MINIX確實允許將一張軟碟分區，但只有第一個分區可以引導。這張軟碟上沒有單獨的主引導記錄，而且也不允許有子分區。這樣便可以使用完全相同的方法來安裝被分區的和未分區的軟碟。軟碟分區的主要用途在於它可以方便地將一張安裝盤分成一個將拷貝到RAM盤的根映射，及一個在不需要時可被卸下來的可安裝部分。這樣便可以空出軟碟機以繼續安裝過程。

將MINIX引導磁區寫入硬碟時，要對它進行修改，修改的內容是將添加一個磁區號以便在其分區或子分區中找到boot程式。這個添加操作是非常必要的，因為在裝入作業系統之前無法通過目錄和檔案名來定位一個檔。該添加操作以及向硬碟寫引導磁區的操作由一個特殊的程式 installboot 完成。boot是MINIX的次級裝入程式，它不僅可以裝入作業系統，而且作為一個監控程序，它允許用戶改變、設置和保存不同的參數。boot從它所在分區的第二個磁區中尋找一套可使用的參數。象標準的UNIX一樣，MINIX保留每個硬碟設備的前1K位元組作為一個啟動區。但其中只有一個512位元組的磁區被ROM引導程式或主引導磁區裝入，這樣另外512位元組可以用來保存設置資訊。這些資訊控制引導操作，並且被傳到作業系統本身。缺省的設置顯示一個只有一個選擇項的功能表，即引導MINIX。但該設置資訊可以被改變，以顯示一個更複雜的功能表。這樣便允許引導其他作業系統（通過裝入並執行其他分區的引導磁區）或使用不同的選擇項來引導MINIX。缺省設置也可以被修改，以旁路掉該功能表而直接引導MINIX。

boot並不是作業系統的一部分，但它很精巧，可以使用檔案系統的資料結構來找到作業系統映射。缺省情況下，boot尋找檔 /minix，若存在 /minix/目錄，則查找其下的最新文件。但也可以修改引導參數以查找其他任何名字的檔。這種靈活性是不一般的，而且多數作業系統固定系統映射的檔案名。但MINIX是一個與眾不同的系統，它鼓勵使用者對其進行改動並生成新的實驗版本。這些都要求進行實驗的用戶應該有某種方法對多個版本進行選擇，因為只有這樣才能在一次實驗失敗後退回到上一個正確的版本。

被boot裝入的MINIX映射是這樣得到的：先對核心、記憶體管理器、檔案系統和init程式分別進行編譯，然後將所產生的各個檔連結而成。其中的每一個獨立檔都包含一個 include/a.out.h中定義的很短的頭。從這個頭所包含的資訊中，boot可以確定在可執行代碼裝入後需要為未初始化資料及各部分初始化資料預留多少空間，據此便可以將下一部分裝在合適的位址。前節中提到的\_sizes陣列也接收一份同樣的資訊，以便核心可以訪問由boot裝入的各模組的位置和大小資訊。可用於裝入引導磁區、boot程式、以及MINIX的記憶體區域取決於硬體。同樣，某些機器體系結構可能需要對可執行代碼內部的位址作一些調整以將其校準到程式裝入的真正位址。Intel處理器的分段體系結構無需進行該操作。由於裝入過程的細節各機器都不相同，而且boot本身不屬於作業系統，所以不再對其進行討論。重要的是作業系統以某種方式被裝入記憶體，一旦裝入過程結束，控制便被傳遞給核心的執行代碼。

此外，需要指出作業系統並不僅僅可以從本地硬碟裝入。無盤工作站可以通過網路從遠地硬碟裝入作業系統，當然這要求ROM中具有網路軟體。儘管具體細節各不相同，但大體上很類似。ROM代碼必須能夠從網路上獲得一個包含完整的作業系統的檔。如果MINIX通過這種方式裝入，則作業系統被裝入記憶體過程中的初始化過程幾乎無需修改。當然這需要一個網路服務器以及一個經過修改的，能夠從網路訪問檔的檔案系統。

2.6.6 系統初始化

基於IBM PC類型機器的MINIX若需要與舊的處理器相容則可被編譯成16位元模式，若在80386以上的晶片上為了獲得更高的性能則可以編譯成32位元模式。這兩種模式使用相同的C原始程式碼，根據編譯器本身是16位元或32位而產生相應的輸出。由編譯器本身定義的一個宏確定檔 include/minix/config.h 中宏 \_WORD\_SIZE 的值。MINIX執行代碼的第一部分是用組合語言寫的，並且針對16位或32位編譯器必須使用不同的原始檔案。初始化代碼的32位版本位於檔 mpx386.s中。對應的16位元系統則位於mpx88.s。這兩者同時也都包含對其他低層核心操作的組合語言支持。16位和32位的選擇在檔mpx.s中自動完成。mpx.s很短，其內容示於圖2－32。

# include <minix/config.h>

# if \_WORD\_SIZE == 2

# include "mpx88.s"

# else

# include "mpx386.s"

# endif

圖 2－32 如何選擇不同的組合語言原始檔案。

Mpx.s示出了C語言前置處理器中＃include 語句的一種非常規用法。通常 # include用來包含標頭檔，但也可以用來選擇原始程式碼的適當部分。使用 # if 語句來作到這一點將需要把兩個很大的檔mpx88.s 和 mpx386.s放在一個單獨的檔中。這樣作不僅不實用，而且浪費磁碟空間。因為在一個特定的安裝中可能這兩個檔中的一個根本就不被用到，而且可以被歸檔或刪除。在隨後的討論中我們將以32位的mpx386.s為例。

由於這是首次接觸可執行代碼，我們先講一下在本書中我們是如何對可執行代碼進行討論的。同時理解編譯一個大的C程式時使用的多個原始檔案是很困難的。通常我們一次只講述一個檔，而且按照這些檔的次序進行。我們從MINIX系統各部分的入口點開始並跟隨執行主線前進。當遇到調用一個支撐函數時，將首先簡略講一下調用的意圖，但並不對其進行詳細的內部描述，而是一直到被調用函數的定義處才詳細地對其進行討論。重要的從屬函式定義通常與其調用處放在同一個檔中，其前邊是發出調用的高層函數。但小的或通用的函數有時被集中放在單獨的檔中。同時，出於移植性的考慮，與機器相關和無關的代碼儘量分開放在獨立的文件中。組織代碼費了很多功夫，實際上在本書的寫作過程中為了方便讀者重寫了許多檔。然而大程式有許多分支，有時理解一個主函數需要讀懂它所調用的函數，所以備一些書簽並脫開書中的次序，以另一種順序觀察問題有時可能會有幫助。

在講完代碼組織方式之後，現在言歸正傳。MINIX的啟動牽涉到幾次控制權轉移，它們發生在mpx386.s中的組合語言常式和start.c及main.c中的C語言常式之間。我們將按執行次序進行討論，儘管這需要從一個檔跳到另一個檔。

一旦引導進程將作業系統裝入記憶體，控制權便轉到標號MINIX（在mpx386.s中，第6051行）。第一條指令是一條躍過幾個位元組資料的跳轉指令。這幾個位元組資料包括引導監控程序標誌（第6054行），該標誌由引導監控程序用來標識核心的不同特徵，其中最重要的是16位元/32位元系統標誌。引導監控程序總是從16位元模式開始，但在需要時將CPU切換到32位元模式。這個切換發生在控制權傳給MINIX之前。監控程序還建立一個堆疊。彙編代碼需要作許多工作，包括：建立一個堆疊框架以為C編譯器編譯的代碼提供適當的環境；拷貝處理器所使用的表格來定義記憶體段；以及建立各種處理器寄存器等。待這些工作結束後，初始化過程通過調用（第6109行）C函數cstart繼續進行。注意在組合語言代碼中用\_cstart引用該函數。這是因為C編譯器編譯的所有函數在符號表中其名字前都有一個底線，而且當編譯好的分立模組被連結時，連結程式查找這樣的名字。由於彙編器並不自動地添加這個底線，所以組合語言程式師必須顯式地加上這個底線以保證在目的檔案中能夠找到相應的符號名。cstart調用另一個常式來初始化通用描述元表(Global Descriptor Table) － 這是Intel 32位元處理器實現保護模式的核心資料結構，以及中斷描述符表(interrupt Descriptor Table) － 它用來為每種可能的中斷類型選擇執行代碼。在從cstart返回之後，lgdt和lidt指令（第6115和6116行）通過向其對應的定址寄存器裝入相應的值來將這些表格啟動。如下指令

jmpf CS\_SELECTOR: csinit

初看起來像是空操作，因為好象在其所處的地方有一個nop 指令序列一樣。它把控制轉到當時控制所在的位置，但這是初始化過程的一個重要部分。這個跳轉指令強制使用剛剛被初始化的結構。在對處理器寄存器作進一步的操作之後，MINIX以第6131行的一個跳轉（不是調用）操作結束，這樣便跳轉到了核心的main入口點（在main.c中）。在該處mpx386.s中的初始化代碼結束。該檔剩餘部分的代碼用來啟動或重啟動一個系統任務、進程、中斷處理常式或其他的支撐常式。出於效率的原因它們均用組合語言編寫，下一節還將對它們進行討論。

現在來看高層的C初始化函數。這裡最基本的策略是用高層的C代碼作盡可能多的操作。正如我們所看到的，這裡已經有了兩種版本的mpx代碼，如果將C代碼可能執行的操作都由C代碼承擔，則將省去兩大段彙編代碼。cstart做的第一件事（在cstart.c中，第6524行）是調用prot\_init來建立CPU的保護機制和中斷表。然後將引導參數拷貝到記憶體的核心部分以及將其轉換成數值。它還確定顯示器的型號、記憶體大小、機器類型、處理器操作模式（真實模式還是保護模式），以及是否可能返回到引導監控程序等。所有的資訊都保存在適當的全域變數中，這是為了使核心代碼的所有部分在需要時都能夠訪問到它們。

Main函數（在main.c中，第6721行）完成初始化，然後開始系統的正常運行。它調用intr\_init來配置中斷控制硬體。該操作之所以放在這裡是因為此前必須知道機器類型，因為完全依賴於硬體，所以該過程放在一個獨立檔中。該調用中的參數（1）指示intr\_init這是在為MINIX執行初始化，若使用參數（0）則再次初始化硬體，使其回到原始狀態。intr\_init通過兩個步驟來保證在初始化完成之前的任何中斷都不會生效。第一步向每個中斷控制器晶片中寫入一個位元組以使其無法響應外部中斷。隨後，用來訪問設備相關的中斷處理常式的表格中所有表項都填入一個常式的位址，該常式在收到一個偽中斷時將列印出一條資訊，它絕對沒有任何副作用。其後，在每個I/O任務運行其自己的初始化代碼時，這些表項逐個地被重填，以指向相應的中斷處理常式。最後，每個系統任務重定中斷控制器晶片中的一個二進位位元以啟動其自己的中斷信號輸入。

接下來調用mem\_init。它初始化一個陣列，該陣列定義系統中每個可用區塊的位址和大小。和中斷硬體的初始化一樣，記憶體初始化的細節是與硬體相關的。而且這裡將mem\_init分離到一個獨立的檔中以保持main函數的平臺無關性。

Main函數的主要部分用來建立進程表，這樣當調度到第一批任務和進程時，它們的記憶體映射和寄存器將能夠被正確地設置。進程表的所有表項都被標誌為空閒；用於加快進程表訪問的pproc\_addr陣列被第6745到6749行的迴圈進行初始化。第6748行的代碼

(pproc\_addr + NR\_TASKS) [t] = rp;

可被定義為：

pproc\_addr [ t + NR\_TASKS] = rp;

因為在C語言中a[i]只是 \*(a+i)的另一種寫法。所以如果你對a或i加一個常量將會是同樣的效果。對某些C編譯器而言，向陣列加一個常數將比向下標加一個常數產生稍微快一些的代碼。

main函數的大部分，即從第6762到6815行的迴圈對進程表進行初始化，使其擁有足夠的資訊來運行系統任務、伺服器進程和init。這些進程在啟動時都必須存在，而且在整個正常操作過程中任何一個都不會結束。在該迴圈的開始，rp被設為一個進程表項的位址（第6763行）。因為rp是指向一個結構的指標，所以結構的成員可以通過 rp->p\_name來訪問，如第6765行所示。在MINIX原始程式碼中廣泛使用這種記法。

系統任務當然被編譯進與核心相同的檔，關於其堆疊的需求資訊放在table.c定義的陣列tasktab中。由於任務被編譯進核心並且可以調用核心中的任意代碼、並可以訪問核心中任意處的資料，所以一個獨立任務的大小是沒有什麼意義的，於是每個任務的空間大小域都填入核心本身的長度。sizes陣列包含了核心、記憶體管理器、檔案系統和init以塊（click）為單位的正文段和資料段大小資訊。這些資訊在核心開始執行之前由boot添加到核心的資料區，並且在核心看來好象是編譯器提供了這些資訊。sizes中的前兩個資料是核心的正文段大小和資料段大小，接下來是記憶體管理器的正文段大小和資料段大小，依此類推。如果這四個程式均不使用分開的指令和資料空間，則正文段大小為0，將正文和資料混合在一起作為資料段。對於每個任務都將 sizeindex賦值為0（第6775行），這保證了對於所有任務都取用sizes陣列第0個元素（第6783和6784行）的值。第6778行中對sizeindex賦值將給每個伺服器進程和init提供它們各自針對sizes陣列的下標。

最早的IBM PC機將ROM放在可用記憶體的高端，在8088 CPU中記憶體大小的限制是1M位元組。現代的PC及其兼容機的記憶體通常比早期的PC大，但是出於相容性的考慮，它們仍將ROM放在同樣的地址。這樣RAM便不再連續，640KB和1MB之間有一塊區域是ROM。只要可能，引導監控程序就儘量將伺服器進程和init程式裝在ROM之上的區域。這主要是為檔案系統考慮，因為這樣一來就可以將一大塊記憶體作為快取記憶體而不會與ROM發生衝突。第6804到6810行之間的條件碼保證了對高端記憶體的使用被記錄在進程表中。

進程表中有兩項對應於兩個不按通常方式調度的進程，它們分別是IDLE和HARDWARE。IDLE是一個空迴圈，在系統中無其他進程就緒時就運行它。HARDWARE進程用於計費－它記錄中斷服務所用的時間。所有其他進程由第6811行的代碼放置在適當的佇列中。第6811行調用的函數是lock\_ready，在對佇列進行操作之前它先設置一個鎖變數switching，在佇列被修改之後則刪除該鎖。在這一點並不需要上鎖和解鎖，但這是標準方法，而且沒必要為一種只出現一次的情況額外編寫代碼。

對進程表中的每個表項進行初始化的最後一步是調用alloc\_segments。這個過程是系統任務的一部分，但當然此時還沒有任務運行，所以在第6814行它是作為一個普通程序呼叫的。這是一個與機器相關的過程，它將各進程使用的記憶體段的位置、大小及運行特權級設置到適當的蜮中。對於舊式的不支援保護模式的Intel處理器，它只定義段位址。對一個記憶體分配方法相異的處理器類型，alloc\_segments必須被重寫。

一旦對所有任務、伺服器和init初始化了進程表，系統基本上就可以運行了。變數bill\_ptr標明對哪個進程進行CPU使用的計費，它需要一個初值。該初值在第6818行賦值，此時IDLE是一個合適的選擇。隨後在調用下一個函數lock\_pick\_proc時可能會選擇其他進程。此時所有的系統任務均已就緒，並且當一個使用者進程運行時bill\_ptr將被改變。lock\_pick\_proc的另一項工作是將變數proc\_ptr指向下一個運行進程的進程表項。選擇下一個運行進程的方法是依次檢查系統任務、伺服器進程和使用者進程佇列。這裡檢查的結果是將proc\_ptr指向控制台任務的進程表項，它總是第一個被啟動的任務。

最後，main的工作至此結束。在許多C程式中main是一個迴圈，但在MINIX核心中，它的工作到初始化結束為止。第6822行中對restart的調用將啟動第一個任務，控制權從此不再返回到main。

\_restart是mpx386.s中的一個組合語言常式。實際上\_restart不是一個完整的函數，它是一個更大的過程的中間入口。我們在下一節將詳細討論該過程。這裡僅指出\_restart引發一個上下文切換，這樣proc\_ptr所指向的進程將運行。當 \_restart執行了第一次時，我們可以說MINIX正在運行－它在執行一個進程。\_Restart被反復地執行，每當系統任務、伺服器進程或使用者進程放棄運行機會掛起時都要執行\_restart，無論掛起原因是等待輸入還是在輪到其他進程運行時將控制器轉交給它們。

第一個排隊的任務（即進程表中佔用第0號表項的進程，也即佔用最小號表項的進程）總是控制台任務。這樣其他任務在啟動時便可以用它來報告進度或發生的問題。控制台任務一直運行到因試圖接收一條消息而阻塞。然後運行下一個系統任務，一直到它因同樣的原因阻塞。最後，所有的任務都將阻塞，這樣記憶體管理器和檔案系統便可以運行。這兩個進程在首次運行時都進行一些初始化操作，但它們同樣也將阻塞。最後init將為每個終端創建一個getty進程，這些getty進程將一直阻塞，直到終端上有鍵入的輸入，在這一點上，第一個用戶進行登錄。

至此我們已經從三個檔跟蹤了MINIX的啟動過程，兩個使用C語言，一個使用組合語言。mpx386.s檔中包含了附加的用於中斷處理的代碼，這些代碼將在下一節中說明。但在繼續之前先簡短地瞭解一下這兩個C檔中的其餘常式。在start.c中剩下的函數有兩個：k\_atoi（第6594行）用於將字串轉換成一個整數，k\_getenv（第6600行）用於在核心的環境中查找資料項目，該環境是引導參數的拷貝。這兩個函數都是標準函式程式庫常式的簡化版，之所以重寫它們是為了保持核心的簡潔。main.c中只剩下一個過程panic（第6829行），當系統發現無法繼續運行下去的故障時將調用它。典型的如無法讀取一個很關鍵的資料塊、檢測到內部狀態不一致、或系統的一部分使用非法參數調用系統的另一部分等。這裡對printf的調用實際上是調用printk，這樣當正常的進程間通信無法使用時核心仍能夠在控制台上輸出資訊。

2.6.7 MINIX的中斷處理

中斷硬體的細節與系統相關，但任何系統都必須具有與Intel - 32位CPU功能等價的部件。由硬體設備產生的中斷是一些電信號，它們首先由中斷控制器進行處理。中斷控制器是一片積體電路，它能夠檢測到許多這類電信號並在處理器的資料匯流排上為其生成唯一的資料格式。這些資料格式之所以必須唯一是因為：對所有這些設備，處理器本身只能有一個輸入，所以它無法辨認需要服務的具體設備。使用32位處理器的PC機通常有兩片中斷控制器晶片，其中每一個可以處理8個輸入，但其中有一片為從片，它的輸出線連到主片的一條輸入線，這樣一共可以掛接15個不同的外部設備，這如圖2－33中所示。

圖2－33 一台32位Intel PC上的中斷處理硬體。

該圖中，中斷信號出現在右側的IRQn信號線上。連到CPU INT管腳的連接線通知CPU發生了中斷。從CPU發出的INTA（中斷應答）信號使負責中斷的控制器晶片將資料放在系統資料匯流排上並通知處理器應執行哪個服務常式。在系統初始化期間，當main調用intr\_init時，對中斷控制器進行程式設計。這種程式設計內容決定了對應於各條輸入線的信號將向CPU送出什麼樣的資料，同時也決定了中斷控制器操作所用的其他參數。放在匯流排上的資料是一個8位元（二進位）的數值，它用作對一個表格的索引，該表格最多可包含256項。MINIX的表格中含有56個表項，其中實際用到35項，其餘21項保留供MINIX將來擴展使用。在32位的Intel處理器上，這張表中包含中斷門描述符，每個中斷門描述符是一個含有若干域的8位元組結構。

對中斷有幾種可能的回應方式，在MINIX使用的一種模式中，中斷門描述符中最重要的一個域指向服務常式可執行程式碼片段和其中的起始位址。CPU執行被選中的描述符所指向的代碼。其結果與執行如下的組合語言指令完全相同：

int <nnn>

唯一的差別在於，對於硬體中斷，<nnn>來自中斷控制器晶片的一個寄存器，而不是程式記憶體中的一條指令。

32位Intel處理器回應中斷時的任務切換機制很複雜，改變程式計數器以執行另一個函數只是其中很小的一部分。當CPU在一個進程運行期間接收到一個中斷時，它將建立一個新堆疊供中斷服務程式使用。該堆疊的位置由任務狀態段（Task State Segment - TSS）中的一項決定。整個系統中有這樣一個結構，它在cstart調用prot\_init時進行初始化，並且在每個進程啟動時被修改。其結果是每個中斷創建的新堆疊總是從被中斷進程的進程表項中stackframe\_s結構的結尾處開始。CPU自動地將幾個關鍵寄存器值壓入新堆疊，包括用來恢復被中斷進程本身堆疊及其程式計數器的那些寄存器。當中斷處理常式代碼開始運行時，它使用進程表中的這個區域作為其自己的堆疊，同時返回被中斷進程所需的大部分資訊也已被保存。中斷處理常式將其餘寄存器的內容壓棧，填充stackframe結構，然後切換到一個由核心提供的堆疊以在進行中斷服務過程中使用。

插斷服務常式的結束操作如下：從核心棧切換回進程表項中的stackfram\_s結構（它不必是最後一次中斷所創建的那個），顯式地彈出非硬體壓棧的寄存器值，並執行一條iretd（從中斷返回）指令。iretd恢復中斷前的狀態，恢復被硬體壓棧的寄存器，並切換回中斷前使用的堆疊。由此可知，中斷停止一個進程，而中斷服務結束則重啟動一個進程。被啟動的進程可能不是最近被停止的那個進程。與一般的組合語言教材中所講述的較簡單的中斷機制不同，在中斷期間被中斷進程的工作堆疊上並不保存任何內容。進一步而言，由於在中斷之後堆疊被重新創建於一個已知的位置（由TSS決定），所以多進程的控制得以簡化。啟動另一個進程所需的是：將堆疊指標指向該進程的stackframe結構，彈出先前顯式壓棧的寄存器值，然後執行一條iretd指令。

當接收到一個中斷時，CPU關掉所有的中斷，這保證了進程表項中的stackframe不會溢出。這個過程是自動進行的，但也存在可以關中斷和開中斷的彙編指令。中斷處理常式在切換到位於進程表之外的核心棧之後重新開中斷，當然在它切換回進程表中的堆疊之後必須再次關中斷。但當它在處理中斷時，其他中斷可以發生並被處理。CPU跟蹤下這些嵌套的中斷，並在中斷嵌套時使用一種更簡單的方法來完成插斷服務常式與被中斷處理常式之間的來回切換。當正在執行一個中斷處理常式（或其他核心代碼）時，若接收到新的中斷，則並不建立一個新堆疊，CPU把恢復被中斷代碼所需要的主要寄存器值壓入已存在的堆疊。當執行核心代碼時若遇到iretd指令，則同樣也使用簡化的返回機制。作為iretd動作的一部分，處理器通過檢查從堆疊彈出的程式碼片段選擇符來決定如何處理iretd指令。

前邊提到的特權級控制在運行進程和執行核心代碼（包括插斷服務常式）時對接收到中斷作出不同回應。當被中斷代碼的特權級和中斷後即將執行代碼的特權級相同時，使用較簡單的機制。只有當被中斷代碼的特權級低於中斷服務代碼時，才使用較複雜的機制。該機制用到TSS和一個新堆疊。一個程式碼片段的特權級記錄在程式碼片段選擇符中，並且因為這是中斷過程中壓棧的內容之一，所以從中斷返回時可以檢查其內容以決定iretd指令將執行什麼操作。當中斷服務期間創建一個新堆疊供使用時，硬體檢查並保證這個新堆疊起碼能夠裝得下需放在其中的最小的資訊集合。這樣就保護了特權級更高的核心代碼不會因為調用系統調用的使用者進程的堆疊空間不夠而崩潰。專門運行多進程作業系統的處理器刻意將這種機制固化在處理器中。

如果你對32位Intel CPU內部工作原理不很清楚，那麼這種操作過程可能會使你感到困惑。一般情況下我們儘量避免描述這類細節，但如能理解發生中斷和執行iretd指令時所發生的處理細節，則對理解核心如何控制進入和離開圖2－2中的“運行”態是非常有幫助的。硬體完成其中許多工作，這極大地簡化了程式師的工作，並使得系統更加高效，但卻使得通過讀軟體代碼來理解其內部操作變得更加困難。

MINIX核心中只有一小部分感知到硬體中斷。這部分代碼在mpx386.s中。其中每個中斷都有一個入口。從 \_hwint00到 \_hwint07（第6164到6193行）的所有入口點的原始程式碼看起來像是調用hwint\_master（第6143行），從 \_hwint08到 \_hwint15（第6222到6251行）的入口點像是調用hwint\_slave（第6199行）。每個入口點好象在該調用中傳遞一個參數，指明需要服務的設備。實際上這些不是調用，而是宏。由巨集hwint\_master定義的代碼的8份拷貝很類似，只有參數irq不同。同樣，hwint\_slave的8份拷貝也很類似。這似乎有些浪費，但這些類似的代碼非常緊湊。每個巨集展開後均不到40位元組。在中斷服務時速度是最重要的，而且這樣作省掉了裝入參數、調用子常式和檢查參數的開銷。

下面將繼續把hwint\_master作為函數而不是巨集來討論。回憶一下在hwint\_master開始執行之前，CPU已經在被中斷進程的進程表項的stackframe\_s結構中建立了一個新堆疊，而且若干關鍵寄存器值已經被保存在那裡。hwint\_master的第一個動作是調用save（第6144行），該子常式將以後重啟動被中斷進程所需的所有其他寄存器值壓棧。save其實可以作為線上代碼寫成巨集的一部分以加快速度，但這樣將使宏的大小增加一倍以上，而且save也需要被其他函數使用。我們將要看到save主要圍繞堆疊操作。當返回hwint\_master時，將使用核心棧而不是進程表項中的stackframe結構。下一步是操作中斷控制器以防止從產生當前中斷的中斷源接收到另一個中斷（第6145到6147行）。該操作遮罩中斷控制器對某一特定輸入的回應能力；CPU對全部中斷的回應能力是在第一次接收到中斷信號時繼承下來的，此時CPU尚未恢復這種特性。

第6148到6150行的代碼將中斷控制器復位，然後使CPU再次能夠從其他中斷源接收中斷。然後，第6152行的間接call指令使用當前正被服務的中斷編號來對一張與設備相關的低層常式位址表進行索引。它們雖然稱作低層常式，但卻用C編寫。它們執行的典型操作有：為輸入裝置進行服務，以及將數據傳到一個相應任務下次運行時可訪問的緩衝區。在從設備相關的低層常式返回之前可能已進行過許多處理操作。

在下一章將看到低層驅動程式代碼的示例。但為了理解hwint\_master中進行了哪些操作，我們現在指出低層代碼可能調用interrupt（在proc.c中，將在下節討論），並且interrupt將中斷轉換成一條消息，然後將該消息發向一個系統任務，而該系統任務對激發該中斷的設備進行服務。再進一步，對interrupt的調用將調用調度程式，並可能選擇該任務作為下一個運行物件。從設備相關代碼返回時，處理器回應全部中斷的能力再次被第6154行的cli指令遮罩，並且中斷控制器準備在所有中斷被再次打開時對激發當前中斷的設備作出回應。然後hwint\_master以一條ret指令（第6160行）結束。這裡看不出使用了什麼技巧。如果一個進程被中斷，在這裡使用的堆疊則是核心棧，而不是啟動hwint\_master之前由硬體建立的進程表中的堆疊。在本例中，save對堆疊的操作將把 \_restart的位址保留在核心棧上。這樣的結果是使一個系統任務、或者伺服器進程、或者使用者進程再次運行。該進程不大會是，實際上基本不會是原先運行的進程。這依賴於特定設備的插斷服務常式所產生消息的處理是否改變了進程調度佇列。這實際上就是多進程併發執行機制的中心內容。

在結束之前，再講一下當核心代碼正在執行時發生中斷的情況，此時正在使用的是核心棧，save將restart1的位址保存在核心棧中。這種情況下，核心先前正在執行的操作將在hwint\_master末尾的ret指令之後繼續下去。這樣中斷便可以嵌套，但當全部低層服務常式結束後， \_restart將最終執行。而且被中斷進程之外的一個進程可能被投入運行。

除了必須將主和從中斷控制器全部使能外，hwint\_slave（第6199行）很象hwint\_master。因為從中斷控制器接收到中斷將使這兩者都被遮罩。這裡有幾個細微的組合語言概念需要說明，首先在第6206行有一句

jmp . + 2

該語句指定跳轉到緊挨本指令之後的位址。該指令在這裡僅僅是為了增加一個小的延遲。最初的IBM PC BIOS設計者認為在連續的I/O指令之間有必要加上延時。儘管並非當前所有的IBM PC兼容機都需要這樣處理，我們還是遵循他們的例子。這種細緻的技術處理正是為什麼有人把硬體設備的程式設計當作深奧的技巧的原因之一。在6214行有一條條件轉移指令，其轉移的目標位址為6218行的一條帶有數位標號的指令：

0: ret

注意該條件轉移指令

jz 0f

並不是象前例那樣指定要跳過幾個位元組。這裡的0f並不是一個16進制數，這是MINIX編譯器所使用的組合語言程式指定一個本地標號的方法。0f表示向前跳轉到下一個數字標號0。普通的標號名不允許以數位開頭。另外一點有趣但容易混淆之處是同一個標號可以在同一檔的其他地方出現，如在hwint\_master的第6160行也出現標號0：。這種情況可能會比表面看上去更複雜，因為這些標號出現在巨集中，並且這些巨集在彙編器看到該代碼之前就被展開。於是彙編器實際上看到的代碼中存在16處標號0：。在宏中間聲明的標號可能發生的這種增殖情況就是組合語言提供本地標號的原因。當解析一個本地標號時，彙編器使用指定方向上相匹配的第一個標號，而忽略其在別處的出現。

現在來看save（第6261行），對它我們已經提到過幾次。它的名字指出了它的若干功能之一，即將一個中斷進程的上下文保存在CPU提供的堆疊上，該堆疊是進程表中的一個stackframe結構。save用變數\_k\_reenter來計算和確定中斷的嵌套級數。如果當前中斷發生時一個進程正在運行，則第6274行的指令

mov esp, k\_stktop

切換到核心棧，並且隨後的指令將 \_restart的位址壓棧（第6275行）。否則的話，核心棧已在使用中，則將restart1的位址壓棧（第6281行）。不論哪種情況，所使用的堆疊與入口時起作用的堆疊可能不同，而且調用它的常式中的返回位址被壓在已壓棧寄存器值的下邊。這些都造成一條普通的return指令不足以返回到調用者。結束save操作的兩個出口點指令

jmp RETADR\_P\_STACKBASE (eax)

（位於第6277和6282行）使用調用save時壓進棧的位址。

mpx386.s中的下一個過程是\_s\_call，它從第6288行開始。在講述其內部細節之前，先看看它是如何結束的。其結束處沒有ret 或jmp指令。在第6315行用cli指令關中斷之後，其流程轉到\_restart繼續執行。\_s\_call是中斷處理機制在系統調用中的對應物。在一軟體插斷，即執行一條int nnn指令後，控制權轉到\_s\_call。對軟體插斷的處理類似硬體中斷，不同之處在於用 int nnn指令中的nnn 作為對中斷描述符表的索引，而不是中斷控制器晶片提供的一個數位。這樣當進入\_s\_call時，CPU已經切換到了進程表（由TSS提供）中的棧，並且幾個寄存器值已經被壓入該棧。截止到調用\_restart時，對\_s\_call的調用在經過\_restart後最終以一條iretd指令結束，而且正如硬體中斷一樣，該指令將啟動proc\_ptr此時指向的進程。圖2-34對硬體中斷和使用軟體插斷機制的系統調用作了比較。

圖 2-34 (a)硬體中斷的處理過程。 (b)系統調用處理過程。

現在來看\_s\_call的細節。它還有另一個名字\_p\_s\_call，這是16位MINIX版本的殘跡，16位版本的MINIX分別具有保護模式和真實模式操作常式。在32位版本中，對這兩個標號的調用都到達這裡。使用系統調用的程式師用C寫的函式呼叫看起來與其他函式呼叫一樣，不論調用的是一個本地定義的函數還是一個C庫中的常式。支援一條系統調用的庫函數代碼構造一條消息，將消息的位址和目標處理序識別碼裝入CPU的寄存器，然後調用一條 int SYS386\_VECTOR指令。如上所述，其結果是將控制轉到 \_s\_call的起始處，同時幾個寄存器值已被壓入進程表中的一個堆疊上。

\_S\_call代碼的第一部分看起來象save函數的線上展開，並且將其餘必須保存的寄存器值保存下來。正如在save中一樣，執行一條

mov esp, k\_stktop

指令，這樣就切換到了核心棧，同時重新開中斷（軟體插斷與硬體中斷的相似之處還包括它們都關中斷）。其後將調用 \_sys\_call，下節中將對其進行討論。現在只需知道它將構造一條消息，然後發送該消息，這進一步將引發調度程式運行。於是，當\_sys\_call返回時，proc\_ptr有可能指向引發本系統調用的進程之外的另一個進程。在執行到達restart之前，一條cli指令將關中斷以保護即將被再次啟動進程的stackframe結構。

我們已經看到有幾種方法使執行線路到達\_restart函數（第6322行）：

1 在系統啟動時從main調用它。

2 在硬體中斷發生後從hwint\_master 或 hwint\_slave跳轉到它。

3 在系統調用發生後從 \_s\_call調用它。

在以上所有情況中調用\_restart時都要關中斷。如果\_restart檢測到存在未掛起的被服務的中斷，這些中斷是在處理其他中斷期間到達的，則它將調用unhold，這樣就允許在任何進程被重新開機之前將其他中斷轉換為消息。這將暫時地重新開中斷，但在unhold返回之前將再次關中斷。截至第6333行，下一個將運行的進程已經被選中。這時關中斷將保證該進程不會被改變。進程表的結構是精心設計的，它以一個stackframe結構開始，而且該指令

mov esp, (\_proc\_ptr)

使CPU的棧指標寄存器指向stackframe，指令

lldt P\_LDT\_SEL (esp)

隨後從stackframe中裝入處理器的本地描述符表寄存器。這使得處理器為使用將運行進程的記憶體段作好準備。接下來的指令將把該位址保存在將運行進程的進程表項中，下一次中斷的堆疊將建立在該表項中。然後，隨後的一條指令把這個位址保存在TSS中。當執行核心代碼（包括中斷服務代碼）時發生中斷，因為將使用核心棧，所以\_restart的第一部分便不再需要。而中斷服務結束應允許核心代碼繼續執行。標號restart1（第6337行）標示出在這種情況下應從何處繼續執行。在這一點上，k\_reenter被減1，以記錄一層可能的嵌套中斷已被處理，然後其餘指令將處理器恢復到下一進程上一次執行所處的狀態。倒數第二條指令修改棧指標，這樣使調用save時壓棧的地址被忽略。如果最後一重中斷是在進程執行時發生的，那麼最後一條指令iretd將結束這一系列操作返回到進程接下來應該執行的地方，恢復其剩餘的寄存器，包括堆疊段寄存器和棧指標。但是如果遇到經由了restart1的iretd，則使用的核心棧就不是一個stackframe，而是核心棧。而且這種情況也就不再是向一個被中斷進程的返回，而是在核心代碼執行期間所發生中斷的完成。在iretd執行期間，當程式碼片段描述符從堆疊彈出時CPU將檢測到這一點，而且在這種情況下，iretd的全部動作是使核心棧繼續保持使用。

關於mpx386.s還有一些內容。除了硬體和軟體插斷之外，CPU內部的各種錯誤條件可能激發異常(exception)。異常並不總是壞事，它們可以支援作業系統提供一種服務，例如為進程提供更多的記憶體，或者將當前換出的記憶體頁面換入等，儘管標準的MINIX並未實現這些服務。但異常發生時不應將其忽略。異常採用和中斷相同的處理機制，使用中斷描述符表中的描述符。該表中的表項指向16個異常處理入口。這些入口從\_divide\_error開始，以\_copr\_error結束，位於mpx386.s尾部的第6350到6412行。根據是否將一個錯誤碼壓棧，這些入口分別跳轉到exception（第6420行）或errexception（第6431行）。此處彙編代碼的處理與我們已經看到的很相似，寄存器被壓棧並且調用C常式 \_exception（注意其中的底線）來處理該事件。異常導致的結果各不相同，有的被忽略、有的導致系統崩潰、有的導致向進程發消息。在下一節中將討論\_exception。

還有另外一個入口點象中斷一樣處理，即 \_level0\_call（第6458行）。其功能在下一節討論，在那裡我們將討論它跳轉到的代碼 \_level0\_func。該入口點與中斷和異常的入口點同放在mpx386.s中是因為它也被int指令所調用。和異常處理常式一樣，它也調用save，因此跳轉到這裡的代碼最終也將以一條ret指令結束，而這條ret指令將發展到調用\_restart。mpx386.s中最後一個可執行函數是 \_idle\_task（第6465行），這是一個空迴圈，在系統中沒有就緒進程時則調用它。

最後，在該組合語言檔的末尾預留了一些資料存儲空間。這裡定義了兩個不同的資料段。在6478行聲明的資料段

.sect .rom

保證該存儲空間被分配在核心資料段的最開始處。編譯器在這裡放置一個魔數以便boot能夠驗證其裝入的檔是一個合法的核心映射。正如在討論核心資料結構時所講的，boot隨後將用 \_sizes陣列覆蓋這個魔數及其後的空間。這裡將為總共有16項的\_sizes陣列分配足夠的空間，以便向MINIX中增加其他的伺服器進程。在

.sect .bss

（第6483行）定義的其他資料存儲區，在核心通常未初始化變數區為核心棧和異常處理常式預留空間。伺服器進程和普通的使用者進程在可執行檔被連結時預留堆疊空間，並在執行時依靠核心來適當地設置堆疊段描述符和堆疊指標，核心也必須為它們自己進行這種處理。

2.6.8 MINIX的進程間通信

MINIX中的進程使用消息進行通信，這裡使用到進程會合的原理。當一個進程執行SEND時，核心的最底層檢查目標進程是否在等待從發送者（或任一發送者）發來的消息。如果是，則該消息從發送者的緩衝區拷貝到接收者的緩衝區，同時這兩個進程都被標記為就緒態。如果目標進程未在等待消息，則發送者被標記為阻塞，並被掛入一個等待將消息發送到接收進程的進程佇列中。

當一個進程執行RECEIVE時，核心檢查該佇列中是否存在向它發送消息的進程。若有，則消息從被阻塞的發送進程拷貝到接收進程，並將兩者均標記為就緒；若不存在這樣的進程，則接收進程被阻塞，直到一條消息到達。

進程間通信的高層代碼在proc.c中。核心的任務是將一個硬體中斷或軟體插斷轉換為一條消息，前者由硬體產生，後者則是請求系統服務（即系統調用）的途徑。這兩者很類似，以至可以用同一個函數處理，但將其分成兩個專門的函數會更高效。

首先看interrupt（第6938行）。在接收到一條硬體中斷後，相應設備的低層插斷服務常式調用該函數。interrupt的功能是將中斷轉換成向該設備所對應的系統任務發送一條消息，而且通常在調用interrupt之前幾乎不進行什麼操作。例如，針對硬碟驅動器的整個低層中斷處理常式只包含下面三行：

w\_status = in\_byte ( w\_wn->base + REG\_STATUS); /\* acknowledge interrupt \*/

interrupt (WINCHESTER);

return 1;

假如不需要讀取硬碟控制器上的一個I/O埠以獲得狀態資訊，則對interrupt的調用可以放在mpx386.s中，而不是象現在這樣放在at\_wini.c中。interrupt所做的第一件事是通過變數k\_reenter（第6962行）來檢查在接收當前中斷時是否已經有一個中斷正在被處理。若是，則將當前中斷排隊，同時interrupt返回。當前中斷將在以後調用unhold時處理。下一個動作是檢查該任務是否正在等待一個中斷（第6978到6981行）。如果任務未作好接收中斷的準備，則其p\_int\_blockd標誌被置位元－後邊我們將看到這將使得丟失的中斷可能被恢復，並且不發送消息。如果通過了這個測試則消息被發送。從HARDWARE向系統任務發送消息是很簡單的，因為任務和核心是編譯在同一個檔中的，因此可以訪問相同的資料區域。第6989到6992行的代碼完成消息的發送，其操作步驟是：在目標任務的訊息緩衝區的源和類型域中填入內容，將目標進程的RECEIVING標誌重定並將該任務解除阻塞。一旦該消息就緒則目標任務被調度運行。在下一節我們將詳細討論進程調度，在interrupt中第6997到7003行的代碼僅僅是一個預覽－這是被用來將一個進程排隊的過程ready的線上代換。這裡的代碼很簡單，因為從中斷產生的消息只發送到系統任務，這樣便無需確定要操作的進程佇列是三種佇列中的哪一個。

proc.c中下一個函數是sys\_call。它與interrupt類似：它將一個軟體插斷（即用來激發一條系統調用的int SYS386\_VECTOR指令）轉換為一條消息。但由於在這種情況下源和目的的可能範圍很寬，並且該調用可能需要發送、或接收、或既發送又接收一條消息，所以sys\_call要做的事情更多。與多數情況一樣，這意味著sys\_call的代碼比較簡短，因為它的工作通過調用其他過程來完成。首先調用isoksrc\_desk，這是proc.h（第5172行）定義的一個宏，它與proc.h（第5171行）中定義的另一個宏isokprocn協同工作。其功能是檢查並保證該消息指定的源進程和目標進程合法。在第7026行一個類似的測試，isuserp（也是proc.h中定義的一個宏）檢查該調用是否從一個使用者進程發出，若是，則它應該先申請發出一條消息，隨後接收一條應答，對使用者進程而言，這是唯一一種允許的調用方式。這裡所測試的錯誤是不大會發生的，但這樣的測試作起來也很容易，因為它們最終編譯成的代碼只是對幾個小整數進行一下比較。在這樣的層次上，作業系統對不大可能發生的錯誤進行測試是應該提倡的。這段代碼在系統活躍期間很可能每秒鐘要執行許多次。

最後，如果該調用需要發送一條消息，則調用mini\_send（第7031行），而如果需要接收一條消息，則調用mini\_rec（第7039行）。這兩個函數是MINIX中正常消息傳遞機制的核心，應加以仔細的研究。

Mini\_send（第7045行）有三個參數：調用進程、目標進程、及指向消息所在緩衝區的指標。它進行許多測試。首先它確認使用者進程只能向檔案系統或記憶體管理器發消息。在第7060行，用宏isuserp對參數caller\_ptr進行測試以確定調用進程是否使用者進程；用一個類似的函數issysentn對參數dest進行測試以確定它是檔案系統或記憶體管理器。如果這兩者的組合是不允許的，那麼mini\_send將以出錯結束。

下一個檢查內容是確認消息的目標進程是一個活躍進程，而不是進程表中的一個空表項（第7062行）。在第7068到7073行mini\_send檢查該消息是否完全處於使用者的資料段、程式碼片段或這兩者的間隙中。如不是則返回一個錯誤碼。

接下來對一個可能發生的鎖死進行測試。在第7079行上有一個測試，它確保該消息的目標進程沒有正在試圖向調用進程發送一條反向的消息。

mini\_send中最關鍵的測試是在7088到7090行。這裡將檢查目標進程是否正阻塞在RECEIVE上，這由進程表項中的p\_flags域中的RECEIVING位元標誌。如果它正在等待，則接著問：“它在等誰？”。如果它正在等待這一條消息的發送者或任一進程，則執行CopyMess來拷貝該消息，然後將接收者的RECEIVING位復位使其解除阻塞。copyMess在第6932行被定義為一個宏，它調用檔klib386.s中的組合語言常式。

另一方面，如果接收者沒有阻塞，或者被阻塞但在等待另一個進程的消息，則執行第7098到7111行的代碼以阻塞發送者並將其排隊。試圖向一個給定目標進程發送消息的所有進程被鏈在一個鏈表上，目標進程的p\_callerq域指向其隊首進程的進程表項。圖2－35（a）示出了當進程3無法向進程0發送消息的情況。如果隨後進程4也無法向進程0發送消息，則情況如圖2－35（b）所示。

圖 2－35 試圖向進程0發送消息的進程被排隊。

當sys\_call的參數function為RECEIVE或BOTH時它將調用mini\_rec（第6119行）。第7137到7151行的迴圈遍歷所有試圖向接收者發送消息的進程，以檢查其中是否有可接受的。若發現一個，則消息從發送者拷貝到接收者，隨後發送者被解除阻塞，狀態變為就緒，並從該佇列中刪除。

如果未發現合適的發送者，則檢查接收進程的p\_int\_blocked標誌是否指明對應該目標進程的一個中斷先前被阻塞了（第7154行）。如果是則構造一條消息－因為來自HARDWARE的消息無非就是其源進程域為HARDWARE，其類型域為HARD\_INT，這裡沒必要調用CopyMess。

如果未找到被阻塞的中斷，則所期望的訊息源和緩衝區位址被保存在其進程表項中，並通過將其RECEIVING位置位元來將其標誌為阻塞。第7165行對unready的調用將接收者從就緒進程佇列中刪除。如進程的p\_flags中有另外一位置位元，那麼一個信號可能正被掛起，於是該進程應很快有另一個運行機會以處理該信號，所以不調用unready，以避免阻塞該進程。

mini\_rec的倒數第二條語句（第7171和7172行）用來處理核心產生的SIGINT、SIGQUIT和SIGALRM信號。當其中一個發生時，如果記憶體管理器正在等待一條來自ANY的消息，則向其發送一條消息。否則，該信號被核心記錄下來直到記憶體管理器試圖從ANY接收消息為止。該條件的測試在這裡進行，並在需要的情況下調用inform來向其告知被掛起的信號。

2.6.9 MINIX的進程調度

MINIX使用與圖2－26所示結構密切相關的多級調度演算法。在該圖中我們看到I/O系統任務位於第2層，伺服器進程在第3層，使用者進程在第4層。如圖2－36所示，調度程式維護三個可運行進程佇列，每個佇列對應一層。陣列rdy\_head為每個佇列設一項，它指向隊首的進程。類似地，rdy\_tail陣列中的項指向隊尾的進程。這兩個陣列都在proc.h中被定義為EXTERN類型（第5192到5193行）。

圖 2－36 調度程式維護三個佇列，每個優先順序對應一個。

當一個阻塞進程喚醒時，它被追加到其佇列的尾部。由於存在rdy\_tail佇列，該追加操作速度很快。當一個運行中的進程阻塞、或一個就緒進程被一個信號撤銷時，該進程從調度程式的佇列中刪除。只有就緒進程被排隊。

有了上述佇列結構之後，調度演算法就很簡單了：找到優先順序最高的非空佇列，並選擇隊首進程即可。如果所有佇列均為空，則運行IDLE進程。在圖2－36中TASK\_Q具有最高優先順序。調度代碼在proc.c中，選擇最高優先順序佇列由pick\_proc（第7179）完成。該函數的主要工作是設置proc\_ptr。任何影響到選擇下一個運行進程的對這些佇列的改變都要再次調用pick\_proc。無論當前進程在什麼時候阻塞，都調用pick\_proc來重新調度CPU。

pick\_proc很簡單。對每個佇列都要進行檢測。首先是TASK\_Q，若其中有一個進程就緒，則pick\_proc將設置proc\_ptr並立即返回。接著檢測SERVER\_Q，同樣若有進程就緒，則設置proc\_ptr並返回。如果USER\_Q上有一個就緒進程，則bill\_ptr將指向該進程以對其即將使用的CPU時間進行計費（第7198行）。這保證了對於一個即將運行的使用者進程，將把系統為它所作的全部工作都計在它的帳上。如果沒有如何一個佇列具有就緒任務，則第7204行將把計費轉到IDLE進程並調度它運行。被選中執行的進程並不僅僅因為被選中而從其所在的佇列中刪除。

過程ready（第7210行）和unready（第7258行）分別用來將一個可運行進程掛入佇列和將一個不再就緒的進程從其佇列中刪除。如同我們已看到的mini\_send和mini\_rec均調用ready，它本來也可以被interrupt調用，但為了加快中斷處理的速度，在interrupt中將這部分代碼寫成了線上代碼。ready對三個進程佇列之一個進行操作，它直接將進程追加到佇列的尾部。

unready也對佇列進行操作，通常它是將佇列頭部的進程去掉，因為一個進程只有處於運行狀態才可被阻塞。這種情況下，如第7293行的例子所示，unready在返回之前要調用pick\_proc。一個未在運行的使用者進程若收到一個信號也可能進入非就緒狀態，若該進程沒在隊首，則將遍歷整個USER\_Q佇列來查找它，一旦找到則將其刪除。

儘管多數調度決策是在一個進程阻塞或解除阻塞時作出的，但調度仍要考慮到當前使用者進程時間片用完的情況。這種情況下，時鐘任務調用sched（第7311行）來將USER\_Q隊首的進程移到隊尾。該演算法的結果是將使用者進程按時間片輪轉方式運行。檔案系統、記憶體管理器和I/O任務絕不會被放在隊尾，因為它們肯定不會運行得太久。這些進程可以被認為是極其可靠的，而且在完成要做的工作後將阻塞。

在proc.c中還有另外幾個進程調度的支撐常式。其中五個：lock\_mini\_send、lock\_pick\_proc、lock\_ready、lock\_unready和lock\_sched在調用相關的函數之前用變數switching執行一個加鎖操作，在結束時再解鎖。proc.c的最後一個函數unhold在討論mpx386.s中的函數\_restart時就已經提到過。它遍歷被掛起的中斷佇列，對其中每個調用interrupt，其目的是在另一個進程被允許運行之前將每一條掛起的中斷轉換成一條消息。

概括起來，調度演算法維護三個優先順序佇列，分別對應I/O任務、伺服器進程和使用者進程。最高優先順序佇列的第一個進程總是被選中執行。系統任務和伺服器進程被允許執行到其阻塞為止，但時鐘任務監視使用者進程所使用的時間。若使用者進程的時間片用完，它被掛在其佇列的尾部，這樣便在相互競爭的使用者進程中達到了時間片輪轉的調度效果。

2.6.10 與硬體相關的核心支援

有幾個C函數與硬體關係極為密切。為了便於將MINIX移植到其他平臺，這些函數被隔離出來放在本節討論的exception.c、i8259.c和protect.c中，而不是和受它們支持的高層代碼放在相同的檔中。

exception.c包含例外處理常式exception（第7512行）。該常式被mpx386.s中的異常處理代碼的組合語言部分調用（作為\_exception調用）。由使用者進程引起的異常被轉換成信號。使用者自己編寫的程式是可能有錯的，但由作業系統本身引起的異常則表明發生了嚴重錯誤，並產生了不可恢復的故障。陣列ex\_data（第7522到7540行）確定發生這類錯誤時應列印的錯誤資訊，或是對每種異常應向使用者進程發送的信號。早期的Intel處理器並不能產生所有這些異常，該陣列每項的第三個域指出能產生各個異常的最低的處理器型號。這個陣列提供了已經實現了MINIX的Intel處理器家族進展的一個有趣的總結。如果嚴重錯誤是由所使用的處理器不支援某種中斷而引起，則7563行將打出一條建議更換CPU的資訊。

i8259.c中的三個函數用於系統初始化過程以對Intel 8259中斷控制器晶片進行初始化。intr\_init（第7621行）完成中斷控制器的初始化，它向控制器的幾個埠寫資料。有幾行代碼對由引導參數匯出的一個變數進行測試，例如在第7637行對第一個埠寫資料以適應不同型號的電腦。在第7638和7644行對參數mine進行測試，並將對MINIX和BIOS ROM均適合的一個數值寫入該埠。當推出MINIX時，可調用intr\_init來恢復BIOS向量，這樣便可以平滑地退回到引導監控程序。Mine選擇所使用的模式。要想理解這裡的完整細節需要仔細研究8259晶片，所以就不再詳細地講述。我們應指出第7642行對out\_byte的調用使主片只接受從片來的輸入，而對其他輸入均不予回應。同時第7648行的類似操作使從片對其輸入不予回應。並且該函數的最後一行預先將下一個函數spurious\_irq（第7657行）的位址裝入irq\_table的每一項中，這保證在真正的處理常式被安裝之前所產生的中斷將不會產生任何副作用。

i8259.c的最後一個函數是put\_irq\_handler（第7673行）。在初始化期間那些必須回應中斷的任務調用它以將其自己的處理常式位址裝入中斷表，覆蓋掉spurious\_irq的位址。

protect.c包含與Intel處理器保護模式相關的常式。通用描述元表（GDT）、本地描述符表（Local Descriptor Tables - LDTs）和中斷描述符表都位於記憶體中，它們提供對系統資源的受保護的訪問。GDT和LDT被CPU中特殊的寄存器所指向，GDT表項指向LDT。GDT對所有進程均可用並記錄了作業系統使用的記憶體區域的段描述符。通常對每個進程有一個LDT，它記錄了該進程所使用記憶體區域的段描述符。描述符是一個包含許多內容的8位元組結構，但段描述符中最重要的是描述一個記憶體區域的基址和限長域。IDT也由8位元組的描述符構成，其中最重要的部分是當中斷被啟動時所執行代碼的位址。

prot\_init（第7767行）由start.c調用以建立GDT（第7828到7845行）。IBM PC BIOS要求它按一種固定的方式排序，所有對它的索引都定義在protect.h中。每個進程LDT的空間都在進程表中分配。每個包含有兩個描述符，分別是程式碼片段和資料段描述符－記住這裡討論的段由硬體定義。這些段與作業系統中的段不同，作業系統中的段將硬體定義的資料段進一步分為資料段和堆疊段。從第7851到7857行，每個LDT的描述符放在GDT中。這些描述符實際上由函數init\_dataseg和init\_codeseg建立。在進程記憶體映射改變時LDT中的表項本身被初始化（即執行EXEC系統調用時）。

另一個需要進行初始化的處理器資料結構是任務狀態段（TSS），該資料結構在本檔的開頭定義（第7725到7753行），並為處理器寄存器和其他任務切換時應保存的資訊提供空間。MINIX只使用某些域的資訊，這些域定義了當發生中斷時在何處建立新堆疊。在第7867行對init\_dataseg的調用保證它可以用GDT進行定位。

為了理解在最底層上MINIX是如何工作的，可能最重要的是：理解異常、硬體中斷、或是int <nnn> 指令如何啟動那些事先寫好的服務代碼的執行。這通過中斷門描述符表完成，陣列gate\_table（第7786到7818行）由編譯器用異常和硬體中斷處理常式的位址來初始化，隨後它被第7873到7877行的迴圈用來對中斷門描述符的大部分進行初始化，具體通過調用int\_gate函數完成。餘下的向量SYS\_VECTOR、SYS386\_VECTOR和LEVEL0\_VECTOR需要不同的特權級，它們由該迴圈後邊的代碼進行初始化。

有充分的理由解釋為什麼按照描述符方式來組織資料，其原因源於硬體細節，以及維持先進的處理器與16位的286處理器之間的相容性這兩個方面。幸運的是我們通常可以將這些細節留給Intel處理器的設計者。對其中大部分而言，C語言允許我們回避具體的細節，但在實現一個真正的作業系統時不可避免地要在某些點上面對這些細節。圖2－37示出了一種段描述符的內部結構。注意在C程式中可被引用為簡單的32位不帶正負號的整數的基底位址被分成三部分，其中兩部分被許多1位元、2位元或4位元的二進位組合隔開。20位的限長被分成16位元和4位元兩部分。限長可被解釋為位元組數、或者是以4096位元組為一頁的頁面數，這具體依賴於G（細微性）位。其他描述符，例如那些用來指定中斷處理方式的描述符，具有不同的結構，但其結構也同樣地複雜。第4章將更加詳細地討論這些描述符。

圖 2－37 一個Intel處理器的段描述符格式。

protect.c中定義的大部分其他函數，用於將C程式中使用的變數和類似於圖2－37所示的供機器使用的格式複雜的描述符資料之間進行轉換。init\_codeseg（第7889行）和init\_dataseg（第7906行）在操作上很類似，都用於將傳給它們的參數轉換成段描述符。而它們則調用下一個函數sdesc（第7922行）來完成此項工作。sdesc函數的工作就是對圖2－37所示的繁瑣的結構進行具體處理。在系統初始化期間並不使用init\_codeseg和init\_dataseg。此外，當啟動新進程時，為了給進程分配適當的記憶體段，系統任務將調用它們。seg2phys（第7947行）只被start.c調用，它執行sdesc的逆操作，即從一個段描述符中析取出段基底位址。int\_gate（第7969行）在為中斷描述符表構造表項時執行一個類似於init\_codeseg和init\_dataseg的函數。

protect.c的最後一個函數enable\_iop（第7988行）使用了一種不大規範的技巧。我們已經在幾個地方指出作業系統的功能之一是對系統資源進行保護，而MINIX使用的方法之一是利用處理器特權級來禁止使用者程式執行某些指令。然而MINIX同時也希望能夠在小規模的系統上運行，這些系統可能只有一個使用者或少許幾個相互信任的用戶。在這樣的系統中，使用者寫的程式很可能會直接訪問I/O埠，例如用於科學實驗中的資料獲取。檔案系統中有一個小秘密－當檔 /dev/mem或 /dev/kmem被打開時，記憶體管理器將調用enable\_iop，它將為執行I/O操作而改變處理器特權級，以允許當前進程執行讀寫I/O埠的指令。要想描述清楚設置這些函數的意圖比描述它們更為複雜。該函數實際上只是將其調用進程stackframe項中一個字的兩個比特置位元，這個字在進程下次運行時將被裝入CPU狀態寄存器。因為它只對當前的調用進程有效，所以其他函數沒必要撤銷該操作。

2.6.11 公用程式和核心庫

最後，核心有一個用組合語言寫的支撐函式程式庫，及幾個位於檔misc.c中的C實用程式，這些組合語言函數在編譯klib.s時被包含進來。先看組合語言文件。klib.s（第8000行）是一個與mpx.s類似的短文件，mpx.s的功能是根據WORD\_SIZE的定義來選擇適當的機器版本。我們將討論的代碼在klib386.s（第8100行）中。其中有近24個實用常式，之所以用組合語言是出於效率的原因，或是因為它們根本就無法用C來編寫。

\_monitor（第8166行）使系統能夠返回到引導監控程序。在引導監控程序看來，整個MINIX是一個子常式，並且當MINIX啟動時，返回引導監控程序的位址被留在其堆疊中。\_monitor只需恢復各段選擇符和MINIX啟動時保存下來的棧指針，然後與從其他子常式同樣地返回即可。

下一個函數 \_check\_mem（第8198行）用於在系統啟動時確定記憶體大小。它對每16個位元組構成的單元進行一個很簡單的檢測，檢測有兩種模式：將每個比特置成“0”和“1”。

儘管可以使用 \_phys\_copy（如下所示）來進行消息的拷貝，然而有一個更快的專門過程 \_cp\_mess（第8243行）被用作此目的，它的調用方式為：

cp\_mess ( source, src\_clicks, src\_offset, dest\_clicks, dest\_offset );

這裡source 為發送者的進程號，它被拷貝到接收者緩衝區的m\_source域。源和目標位址以一個塊號來指定，典型地為包含該緩衝區的段基底位址，及一個相對於該塊的偏移。這種指定源和目標的方式比 \_phys\_copy所用的32位位址更高效。

定義 \_exit、\_\_exit和 \_\_\_exit（第8283到8285行）是因為編譯MINIX時可能用到的一些庫常式需調用標準C函數exit。從核心退出是一個沒有意義的概念，從核心退出後便無處可去。這裡所用的解決辦法是開中斷，然後進入一個無休止的迴圈。最終一個I/O操作或時鐘將發出一條中斷，由此恢復正常的系統操作。入口點 \_\_\_main（第8289行）是處理編譯操作的另一種方法。該方法在編譯一個使用者程式時很合理，但在核心中則沒什麼用處。它指向一條組合語言的ret（從子常式返回）指令。

\_in\_byte（第8300行）、\_in\_word（第8314行）、\_out\_byte（第8328行）和\_out\_word（第8342行）提供了對I/O埠的存取方法。在Intel硬體中I/O埠使用與記憶體分開的位址空間，並使用與記憶體訪問不同的指令。\_port\_read（第8359行）、\_port\_read\_byte（第8386行）、\_port\_write（第8412行）和\_port\_write\_byte（第8439行）完成資料塊在I/O埠之間的傳送。這種傳送主要用於與磁片相關的資料傳遞，這種操作必須比其他I/O調用的速度快。面向位元組的版本在操作中以8位而不是16位元來讀數據是為了與老的8位的外部設備相容。

有時一個系統任務需要臨時地關中斷，該操作是通過調用\_lock（第8462行）完成的。當中斷可被重新打開時，該任務調用 \_unlock（第8474行）來開中斷。所有這些操作是由一條機器指令完成的。與之相比， \_enable\_irq（第8488行）和 \_disable\_irq（第8521行）的代碼則更為複雜。它們工作在中斷控制器晶片的層次來使能和禁止單個的硬體中斷。

\_phys\_copy（第8564行）在C程式中使用如下語句進行調用：

phys\_copy (source\_address, destination\_address, bytes);

它將實體記憶體中任意處的一個資料塊拷貝到任意的另外一處。其中，兩個位址都是絕對位址，也就是位址0確實表示整個位址空間的第一個位元組，並且三個參數均為無符號長整數。

接下來的兩個小函數專用於Intel處理器。\_mem\_rdw （第8608行）返回記憶體中任意處的一個16位的字。其結果用0擴展後放在32位長的eax寄存器中。\_reset函數（第8623行）使處理器復位，這通過將一個空指標裝入處理器的中斷描述符表寄存器，然後執行一個軟體插斷來實現，它與硬體重定效果一樣。

下面兩個常式支援視頻顯示器，它們被控制台任務使用。 \_mem\_vid\_copy（第8643行）將一個字串從核心的記憶體區域拷貝到視頻顯示器的記憶體中，該字串中包含替換字元碼和若干屬性位元組。\_vid\_vid\_copy（第8696行）完成顯示器記憶體內部的資料塊拷貝。在某種程度上它更複雜一些，因為目的資料塊可能與來源資料塊之間有重疊，並且資料移動的方向非常關鍵。

檔中最後一個函數是 \_level0（第8773行），它允許在需要時任務可擁有最高特權級 － 0級。它用於這樣的操作，如復位CPU、或訪問PC的ROM BIOS常式。

misc.c中的C語言公用常式具有專門功能。mem\_init（第8820行）僅在MINIX首次啟動時被main調用。在一台IBM PC兼容機上可能存在兩個或三個不連續的記憶體區域。PC使用者稱為“常規”記憶體的最低端記憶體的大小，以及從PC ROM之上開始的記憶體區域（“擴展”記憶體）由BIOS告知引導監控程序，引導監控程序隨後又將其作為引導參數傳遞。該參數由cstart進行解釋並在引導時寫入low\_memsize和 ext\_memsize。第三個區域是“影子”記憶體，BIOS ROM可被拷貝到該區域以提高性能，因為ROM通常比RAM的存取速度慢。由於MINIX一般不使用BIOS，所以mem\_init力圖定位該片記憶體並將其加入它的可用記憶體區中，這是通過調用check\_mem來完成的。

下一個常式env\_parse（第8865行）也是在啟動時使用。引導監控程序可以在引導參數中將形如“DPETH0＝300：10”的任何字串傳給MINIX。env\_parse試圖找到一個第一個域與其第一個參數env匹配的串，然後析取出所需的域。代碼中的注釋解釋了該函數的用途，主要是為了説明使用者增加一些需要參數的新驅動程式。例中的“DPETH0”用於向支援網路的MINIX傳遞配置資訊。

本章討論的最後兩個例子是bad\_assertion（第8935行）和bad\_compare（第8947行）。只有在宏DEBUG被定義為TRUE時才對其進行編譯。它們支援assert.h中的宏。儘管本書中的代碼並不引用這兩個常式，但對於那些想構造自己修改了的MINIX的讀者，它們可以幫助進行調試。

2.7 小結

為了遮罩中斷的影響，作業系統提供了一個由併發運行的順序進程所構成的概念模型。進程之間可以使用進程間通信原語來相互通信，比如信號量、管程、消息等。使用這些原語是為了保證任一時刻不會同時有兩個進程進入臨界區。進程的狀態可以是運行、就緒、或阻塞，且當它或另一個進程執行一條進程間通信原語時可能引起狀態的改變。

進程間通信原語可用來解決諸如生產者－消費者、哲學家用餐、讀者－寫者以及理髮師睡覺等問題。但即使使用了這些原語，也必須注意避免錯誤和鎖死。目前已經有許多種調度演算法，包括時間片輪轉、優先順序調度、多級佇列以及策略驅動的調度等。

MINIX支援進程概念，並提供了用於進程間通信的消息。消息是不加以緩衝的，所以一條SEND操作只有在接收者正在等待它時才成功。同樣地，一條RECEIVE操作只有在消息已經可用時才成功。這兩種操作的調用進程在操作不成功時都將阻塞。

當中斷發生時，核心的最底層將構造一條消息，並將消息發送到與中斷設備相關的系統任務。例如，當磁片任務讀寫一塊資料時，它先向硬碟控制器發出一條命令，然後調用receive並阻塞。當資料準備好之後硬碟控制器將發出一條中斷。低層軟體隨後構造一條發給硬碟任務的消息並將其標誌為就緒。當硬碟任務被再次調度運行時，它將獲取該消息並進行處理。中斷處理常式當然也可以直接進行一些操作，例如時鐘中斷處理常式更新時間。

中斷可能引發任務切換。當一個進程被中斷時，該進程的進程表項中將建立一個堆疊，再次啟動該進程所需的全部資訊都放在這個新堆疊上。通過以下操作可以再次啟動任何一個進程：將棧指標指向其進程表項，然後執行一個指令序列來恢復CPU寄存器值，最後以一條iretd指令結束。調度程式決定堆疊指向哪個進程表項。

核心本身在運行時也可能發生中斷。CPU檢測到中斷後將使用核心棧，而不是進程表中的堆疊。這樣便允許中斷嵌套。當後來的插斷服務常式結束後，在它之前執行的插斷服務常式就可以運行直到結束。在所有中斷都被處理之後則可以重啟動一個進程。

MINIX調度演算法使用三個優先順序佇列。優先順序最高的是系統任務；下一個是檔案系統、記憶體管理器及其他伺服器進程；最低的是使用者進程。使用者進程每次運行一小段時間，這樣便達到時間片輪轉的調度效果。但其他進程則一直運行直到阻塞或被剝奪為止。

習 題

1 假設你正在設計一種先進的電腦體系結構，它使用硬體而不是中斷來完成進程切換，則CPU需要哪些資訊？請描述用硬體完成進程切換的工作過程

2 目前的電腦上，中斷處理常式至少有一小部分用組合語言編寫，為什麼？

3 書中認為圖2－6（a）的模型不適用於在記憶體進行快取記憶體的檔案伺服器，為什麼？可否使每個進程擁有自己的cache？

4 在使用執行緒的系統中，是每個執行緒有一個堆疊還是每個進程有一個堆疊，說明原因。

5 什麼是競爭條件？

6 寫一個shell程式，其功能是產生一個內容為一個整數序列的檔，要求它先讀取檔中的最後一個整數，將其加1，然後將這個新整數追加到檔的末尾。在系統前臺和後臺同時運行它並使用相同的檔案名。在因競爭而造成的故障發生之前它能運行多久？此處的臨界區是什麼？請對其進行修改以防止發生競爭。

（提示：使用

ln file file.lock

來將資料檔案加鎖）

7 對於前一問題，如下的語句

ln file file.lock

是一種很有效的加鎖機制嗎？說明原因。

8 兩個進程在一台共用記憶體的多處理機（即共用同一個記憶體）上運行時，圖2－8所示的採用變數turn的忙等待方案還奏效嗎？

9 現有一台電腦，不具備TEST AND SET LOCK指令，但有一條指令可以按原子操作方式將一個寄存器的值和一個記憶體字進行交換，能否利用該指令寫一個與圖2－10中enter\_region類似的常式？

10 給出一個框架，來描述一個可禁止中斷的作業系統如何實現信號量。

11 請說明如何只利用二進位信號量和普通的機器指令來實現計數信號量（即可以擁有任意大的數值的信號量）。

12 在 2.2.4節中描述了一個高優先順序進程H和低優先順序進程L的情況，它最終導致H陷入閉環，若採用時間片調度而不是優先順序調度，還能發生這種情況嗎？請進行討論。

13 管程內部的同步機制使用條件變數和兩個特殊操作WAIT和SIGNAL，一種更一般的同步方式只有一條原語WAITUNTIL，它以任意的布林運算式作為參數。例如

WAITUNTIL x<0 or y+z<n

這樣便不再需要SIGNAL原語。很顯然這個方案比 Hoare 和Brinch Hansen的方案更通用，但它從未被採用過，為什麼？（提示：從實現考慮）

14 一個快餐廳有四種職員：（1）領班，他們接收顧客點的功能表（2）廚師，準備飯菜（3）打包工，將飯菜裝在袋子裡（4）出納員，收錢並將食品袋交給顧客。每個職員可被看作一個進行通信的串列進程，它們採用的進程間通信方式是什麼？將該模型與MINIX中的進程加以比較。

15 假設有一消息傳遞系統使用信箱，一個進程向滿信箱發消息或從空信箱收消息都不會阻塞，進程對錯誤碼的處理方式為重試，直到成功為止，這種方案會帶來競爭嗎？

16 在圖2－20所示的哲學家用餐問題的解法中，為什麼過程take\_forks將狀態變數置為HUNGRY？

17 考慮圖2－20中的過程put\_forks，假設變數state[i]在對test的兩次調用之後被置為THINKING，而不是在調用之前。對於3位元哲學家的情況這個改動有什麼影響？對於100位元哲學家的情況呢？

18 從何種類型的進程可以在何時被啟動的角度來看，讀者－寫者問題可以通過幾種方式進行形式化。根據優先哪幾類進程的不同，請詳細地描述該問題的三種變體。對每種變體，請說明當一個讀者或寫者能夠訪問資料庫時情況將會怎樣，以及當一個進程對資料庫訪問結束後又將會怎樣。

19 CDC6600電腦使用一種有趣的稱作處理器共用的時間片調度演算法，它可以同時處理多達10個I/O進程，每條指令結束後都進行進程切換，這樣進程1執行指令1，進程2執行指令2，等等。進程切換由特殊的硬體完成，所以沒有開銷。如果在沒有競爭的條件下一個進程需要T秒鐘完成，那麼當有n個進程共用處理器時需要多長時間完成？

20 時間片調度程式通常維護一個有所有就緒進程組成的佇列，每個進程在佇列中出現一次。如果一個進程在佇列中出現兩次以上，情況將會怎樣？你能設想出這種情況出現的原因嗎？

21 對某系統進行監測後表明平均每個進程在I/O阻塞之前的執行時間為T。一次進程切換需要的時間為S，這裡S實際上即為開銷。對於採用時間片長度為Q的時間片調度法，對以下各種情況給出CPU利用率的計算公式。

(a) Q=∞

(b) Q>T

(c) S<Q<T

(d) Q=S

(e) Q 趨近於0

22 有5個待運行任務，各自預計執行時間分別是9, 6, 3, 5和X。採用哪種運行次序將使平均回應時間最短？ （答案依賴於X）

23 有5個批次處理任務A到E幾乎同時到達一計算中心。其預計執行時間分別為10， 6， 2， 4和8分鐘。其優先順序（由外部設定）分別為3，5，2，1和4，這裡5為最高優先順序。對於下列每種調度演算法，計算其平均進程周轉時間，進程切換開銷可忽略。

(a) 時間片輪轉

(b) 優先順序調度

(c) 先來先服務（按照次序 10， 6， 2， 4， 8）

(d) 最短作業優先

對（a），假設系統具有多道處理能力，每個作業均獲得公平的CPU份額，對（b）到（d）假設一時刻只有一個作業運行，直到結束。所有的作業都是完全的CPU密集型作業。

24 在CTSS上運行的一個進程需要30個時間片方能結束，則它需要被換入多少次？包括第1次，即開始運行之前。

25 使用一個參數 a=1/2的老化演算法來預測執行時間。從最早到最近的前4次執行時間分別為40，20，40和15毫秒，則下次執行時間預計為多長？

26 一個軟即時系統有4個週期性事件，其週期分別為50，100，300和250毫秒。假設其處理分別需要35，20，10和X毫秒，則該系統可調度所允許的X值最大是多少？

27 解釋兩級調度為什麼被廣泛採用？

28 MINIX在執行期間維護一個變數proc\_ptr，它指向當前進程的進程表項，為什麼？

29 MINIX對消息不加緩衝，請解釋這樣的設計會對時鐘和鍵盤中斷帶來什麼問題？

30 在MINIX中當一條消息被發送給一個睡眠進程時，將調用過程ready來將該進程掛入適當的調度佇列中，該過程首先要關中斷，為什麼？

31 MINIX中的mini\_rec過程包含一個迴圈，請解釋為什麼需要該迴圈。

32 MINIX使用圖2－23所示的調度方法，其中不同類型的進程有不同的優先順序。優先順序最低的進程（使用者進程）使用時間片調度法，而系統任務和伺服器進程則允許一直運行到阻塞。請問優先順序最低的進程是否會發生饑餓，為什麼？

33 MINIX適用于即時應用嗎？例如資料日誌記錄。若不適用，可對其作什麼改動？

34 設你有一個提供信號量的作業系統，請實現一條消息傳遞系統，寫出發送和接收消息的過程。

35 一個主修人類學、輔修電腦科學的學生參加了一個課題，調查非洲狒狒是否能被教會理解鎖死。他找到一處很深的峽谷，在上邊固定了一根橫跨峽谷的繩索，這樣狒狒就可以攀住繩索越過峽谷。同一時刻可以有幾隻狒狒通過，只要它們朝著相同的方向。但如果向東和向西的狒狒同時攀在繩索上則將發生鎖死（狒狒將被卡在中間），因為它們無法在吊在峽谷上時從另一隻的背上翻過去。如果一隻狒狒想越過峽谷，它必須看當前是否有別的狒狒正在逆向通過。使用信號量寫一個避免鎖死的程式來解決該問題。

36 繼續前一問題，但現在要避免饑餓。當一隻想向東去的狒狒到了繩索跟前，但發現有別的狒狒正在向西越過峽谷時，它將一直等到繩索可用為止。但在至少有一隻狒狒向東越過峽谷之前，不允許再有狒狒開始從東向西越過峽谷。

37 使用管程，而不是信號量來解決哲學家用餐問題。

38 在MINIX核心中增加一些代碼來跟蹤從進程（任務）i發送到進程（任務）j的消息個數，按下F4鍵時列印出該矩陣。

39 修改MINIX的調度程式以跟蹤每個使用者進程最近使用的CPU時間。當沒有任務或伺服器進程運行時，要求選擇使用CPU最少的使用者進程運行。

40 重新設計MINIX，使得每個進程的進程表中有一個優先順序域可用來對單個進程設置更高或更低的優先順序。

41 修改mpx386.s中的宏hwint\_master 和hwint\_slave，使得當前由save函數執行的操作改為由線上代碼執行。這樣作使代碼增大了多少？你能測出性能提高了多少嗎？

第三章 輸入/輸出系統

作業系統的主要功能之一是控制所有的輸入/輸出（Input/Output）設備，它必須向設備發佈命令，捕獲中斷並進行錯誤處理，它還要提供一個設備與系統其餘部分之間的簡單易用的介面。該介面應對所有設備盡可能地一致（設備無關性）。I/O部分的代碼佔據了整個作業系統的相當部分。本章研究作業系統如何管理輸入/輸出。

本章內容安排如下：首先介紹I/O硬體的基本原理，然後概述I/O軟體。I/O軟體可以分為若干層次，每一層都有一個定義良好的介面，我們將討論其中的每一層，以瞭解各層的功能以及它們如何結合在一起。

隨後討論鎖死。我們將給鎖死下一個準確的定義，講述鎖死產生的原因，提供對鎖死進行分析的兩種模型，並討論幾種鎖死預防的演算法。

接下來簡要地流覽一下MINIX中的I/O。在概述之後我們將詳細討論四種I/O設備 － RAM盤、硬碟、時鐘和終端。對每一種設備我們都將研究其硬體、軟體以及在MINIX中的實現。在本章的結尾，我們將簡短地討論一段MINIX代碼，它位於與I/O任務相同的層次但並不執行I/O任務，它為記憶體管理和檔案系統提供一些服務，例如從使用者進程獲取若干資料塊。

3.1 I/O硬體原理

不同的人對I/O硬體有不同的理解。在電氣工程師看來，I/O硬體就是一堆晶片、電線、電源、馬達和其他設備的集合體；而程式師則主要注意它為軟體提供的介面－硬體能夠接收的命令、它能夠完成的功能、以及能報告的各種錯誤等。本書主要立足於對I/O設備的程式設計，而並非對硬體的設計、製造和維護，因此我們的重點放在如何對硬體程式設計，而不是其內部的工作原理。然而，對許多I/O設備程式設計常常不可避免地牽涉到其內部操作。在以下三節中我們將介紹與I/O設備程式設計有關的硬體知識。

3.1.1 I/O設備

I/O設備可粗略地分為兩類：塊設備（block devices）和字元設備（character devices）。塊設備將資訊存儲在可定址的固定大小的資料塊中，通常資料塊大小的範圍從512位元組到32768位元組不等。塊設備的主要特徵是能夠獨立地讀寫單個的資料塊。磁片是最常見的塊設備。

從嚴格意義來說，按塊編址的設備和無法按塊編址的設備之間沒有明顯的界限。磁片是公認的按塊編址的設備，因為無論當前磁頭處於什麼位置，總可以定位到其他柱面並等待所需要的資料塊旋轉到磁頭下面。現在設想用一台8mm或DAT磁帶機作磁片備份。磁帶上也是固定大小的資料塊。如果命令磁帶機讀第n個資料塊，則通過先倒帶再前進總能定位到所需要的資料塊。該操作可比作磁片的定址，只不過更費時罷了。同樣，重寫磁帶中間的某個資料塊可能作得到，也可能作不到。即使將磁帶用作為隨機存取的塊設備是可能的，但是通常並不這樣使用它們。

另一類是字元設備。與塊結構相比，一個字元設備可以發送或接收一個字元流。字元設備無法編址，也不存在任何定址操作。印表機、網路介面、滑鼠（用作指點設備）、老鼠（用於心理學實驗）以及多數與磁片不同的設備均可被視作字元設備。

這種分類方法並不是最好的，有些設備就不適用於這種分類法。例如，時鐘是無法定址的，同時它也不產生或接收字元流。時鐘的全部功能就是按照預先定義的時間間隔發出中斷。存儲映射顯示器也不適用該分類模型。但總的來說，使用這種方法可以將控制不同I/O設備的作業系統軟體成分隔離開來。例如檔案系統僅僅控制抽象的塊設備，而將與設備有關的部分留給低層軟體，即設備驅動程式（device driver）去處理。

3.1.2 設備控制器

I/O設備通常包含一個機械部件和一個電子部件。為了達到設計的模組性和通用性，一般將其分開。電子部分稱為設備控制器或適配器。在個人電腦中，它常常是一塊可以插入主機板擴充槽的印刷電路板。機械部分則是設備本身。

控制器卡上一般都有一個接線器，可以將與設備相連的電纜線接進來。許多控制器可以控制2個、4個甚至8個相同設備。如果控制器和設備之間的介面採用標準介面，如ANSI、IEEE、ISO，或者事實上的標準，則多家廠商都可以製造與該介面匹配的控制器和設備。例如許多廠商生產IDE（集成設備電子器件）介面或SCSI（小型電腦系統介面）介面的硬碟驅動器。

之所以區分控制器和設備本身是因為因為作業系統大多與控制器打交道，而非設備本身。大多數小型電腦的CPU和控制器之間的通信採用圖3-1所示的單匯流排模型。大型主機則採用其他模型，常常是多條匯流排以及專門用於I/O的電腦，這種專用電腦被稱為I/O通道，它能夠減輕主CPU的部分工作負擔。

圖 3-1 一個連接CPU、記憶體、控制器和I/O設備的模型。

控制器與設備之間的介面通常是一種很低層次的介面。例如一個磁片，可能被格式化為每道16個512位元組的磁區。實際從驅動器讀出來的是一個位元流，以一個首碼開始，隨後是一個磁區的4096比特，最後是一個檢查和（亦稱作改錯碼ECC）。其中的首碼是磁片格式化時寫進磁片的，裡面包含有柱面數和磁區數、磁區大小之類的資料，以及同步資訊。

控制器的任務是將這個串列的位元流轉換成位元組塊並在需要時進行糾錯。通常該位元組塊是在控制器中的一個緩衝區中逐個比特彙集而成。在對檢查和進行校驗證實資料正確之後，該塊資料隨後被拷貝到主存中。

在同樣低的層次上，CRT終端控制器也是一個比特串列設備。它從記憶體中讀取欲顯示字元的位元組流，然後產生用來調製CRT射線的信號，最後將顯示結果打在螢幕上。控制器還產生當水準方向掃描結束後的折返信號以及當整個螢幕被掃描後的垂直方向折返信號。如果沒有CRT控制器，則作業系統程式師只能自己編寫程式來解決此問題。有了CRT控制器後，作業系統只需通過幾個參數對控制器進行初始化，輸入參數，諸如每行的字元數和每屏的行數，之後控制器將自行控制掃描波束。

每個控制器都有一些用來與CPU通信的寄存器。在某些電腦上，這些寄存器佔用記憶體位址空間的一部分，這種方案稱作記憶體映射I/O。680X0系列電腦就採用該方案。其他電腦則使用I/O專用的位址，每個控制器佔用其中的一部分。設備的I/O位址分配由控制器上的匯流排解碼邏輯完成。有些生產IBM PC兼容機的廠商採用與IBM不同的I/O編址方案。除I/O埠外，許多控制器還使用中斷來通知CPU它們已作好準備，使其寄存器可以讀寫。中斷首先是一個電信號事件。硬體的插斷要求信號線提供了中斷控制器晶片的物理輸入。這種輸入線的數量是有限的：奔騰系列的PC機向I/O設備提供15條可用中斷。有些控制器直接製作在主機板上，如IBM PC機的鍵盤控制器。對於那些單獨插在主機板上的控制器，有時它上面設有一些可用來設置IRQ號的開關或跳線器，以便避免IRQ衝突（對於具有隨插即用功能的板子，IRQ號可由軟體設定）。中斷控制器晶片將每個IRQ輸入映射到一個中斷向量，通過該中斷向量便可以找到相應的中斷服務程式。圖3-2給出了IBM PC機部分控制器的I/O位址、硬體中斷和中斷向量號。MINIX採用相同的硬體結構，但其中斷向量則與此處給出的MS-DOS的中斷向量不同。

圖 3-2 一台典型的運行MS-DOS的PC機中若干控制器、I/O位址、硬體中斷線和中斷向量示例

作業系統通過向控制器寄存器寫命令字來執行I/O功能。例如IBM PC的軟碟控制卡可以接收15條命令，包括讀、寫、格式化重新校準等。其中許多命令帶有參數，這些參數也要裝入控制器的寄存器中。某設備控制器接收到一條命令後，CPU可以轉向其他工作，而讓該設備控制器自行完成具體的I/O操作。當命令執行完畢後，控制器發出一個中斷信號，以便使作業系統重新獲得CPU的控制權並檢查執行結果。CPU仍舊是通過從控制器寄存器中讀取若干位元組資訊來獲得執行結果和設備的狀態資訊。

3.1.3 記憶體直接存取DMA

許多設備，尤其是塊設備都支援直接記憶體訪問，或稱DMA（direct memory access）。我們先看一下沒有使用DMA時磁片是怎麼讀數據的。首先控制器逐個比特地從設備完整地讀出一塊（一個或多個磁區）資料放入內部緩衝區中。然後計算該塊資料的檢查和以保證讀取正確。接著控制器發出中斷信號，作業系統開始逐個位元組（或字）地從控制器的設備寄存器中將資料讀入記憶體。由於一次只能讀一個位元組（或字），所以需要通過一個迴圈才能將整個資料塊讀完。

顯然，CPU迴圈地每次從控制器讀一個或幾個位元組是很費時的，為了解決這個問題，人們引入DMA將CPU從這項工作中解脫出來。使用DMA時，CPU除了告訴控制器資料塊的位址外，只需告訴控制器兩條資訊：資料塊將存放在記憶體的位址和要傳輸的位元組數，如圖3-3所示。

圖 3-3 一次DMA傳輸完全由控制器完成。

控制器首先從設備中將整個資料塊讀入內部緩衝區並進行校驗，接著它將第一個位元組（或字）拷貝至DMA記憶體位址處，隨後它對DMA位址和DMA計數分別增減剛剛傳送的位元組數。這個過程一直重複下去直到DMA計數變成0。此時設備控制器發出中斷信號，通知作業系統資料已讀取完畢，在這裡作業系統無需再將資料拷貝到主存中，因為資料已經在讀的過程中放在那裡了。

可能有人會有疑問：為什麼控制器從設備讀到資料後不立即將其送入記憶體，而是需要一個內部緩衝區呢？原因是一旦磁片啟動開始讀數據，從磁片讀出位元流的速率是恒定的，不論控制器是否作好接收這些比特的準備。若此時控制器要將資料直接拷貝到記憶體中，則它必須在每個字傳送完畢後獲得系統匯流排的控制權。如果由於其他設備的爭用而導致匯流排忙，則只能等待。如果在上一個字還未送入記憶體之前下一個字到達，控制器只得找另一個地方把它暫存起來。如果匯流排非常忙，則控制器可能需要大量的資訊暫存，而且還要作大量的管理工作。從另一方面來看，如果採用內部緩衝區，則在DMA操作啟動之前不需要使用匯流排，這樣控制器就能夠設計得很簡單，因為DMA到主存的傳輸對時間要求並不嚴格（有些舊式控制器的確直接訪問主存，而且內部緩衝區設計得很小，但當匯流排忙碌時，一次傳輸有可能被迫終止）。

上述兩個步驟的緩衝過程對I/O性能有重要影響。當資料被CPU或控制器從控制器傳送到記憶體時，下一個磁區正好通過磁頭下邊，同時磁區上存儲的資訊正在送入控制器。簡單的控制器不具有同時執行輸入和輸出的功能，因此當進行記憶體傳送時，當前通過磁頭下的那個磁區的資料將丟失掉。

這樣的結果是控制器只能隔一個塊讀取一個資料塊 ，為了讀完一條完整的磁軌，磁片需要旋轉兩周，一周讀偶數塊，一周讀奇數塊。如果資料通過匯流排從控制器傳到記憶體所用的時間比從磁片讀數據到控制器的時間長，則有可能讀一塊資料需跳過兩塊（或更多）。

有意地跳過一些塊以便為控制器騰出時間供其將資料傳送到記憶體的技術稱為交叉編址。磁片在格式化時，資料塊編號就考慮到了交叉係數。在圖3-4 (a)中，我們看到一個沒有交叉編址的8個塊的磁片編址方案。在圖3-4 (b)中，我們看到採用單交叉編址的方案。在圖3-4 (c)中則採用雙交叉編址方案。

圖 3-4 (a)不採用交叉編址 (b)單交叉編址 (c)雙交叉編址

這種磁片塊編址方案的意圖是：使作業系統能夠連貫地讀出資料塊，並同時達到硬體許可的最大速度。如果按圖3-4 (a)進行編址，而控制器只能作到隔一個塊讀取資料的速度，則作業系統為了保證將8個資料塊按照從0到7的順序讀出，磁片就需要旋轉8周（當然，若作業系統知道這個問題，則它可以用軟體方法來解決，但最好由控制器來解決交叉編址的問題）。

並非所有的電腦都使用DMA。反對意見認為主CPU比DMA控制器快得多，完全可以更快地完成這件工作（當I/O設備的速度不構成瓶頸時）。如果CPU無事可作，而又被迫等待慢速的DMA 控制器，這是完全沒有意義的。同時，省去DMA控制器，由CPU完成所有這些工作也能夠節省一些成本。

3.2 I/O軟體原理

現在來看I/O軟體。I/O軟體的總體目標很明確。其基本思想是，將軟體組織成一種層次結構，低層軟體用來遮罩硬體的具體細節，高層軟體則主要是為使用者提供一個簡潔、規範的介面。隨後幾節我們將瞭解這些目標是如何實現的。

3.2.1 I/O軟體的目標

I/O軟體設計的一個關鍵概念是沒備無關性。其含義就是使程式師寫出的軟體無需修改便能讀出軟碟、硬碟以及CD-ROM等不同設備上的檔。用戶可以簡單地輸入如下命令：

sort < input > output

就能夠從各種設備上獲得輸入，包括軟碟、硬碟、或者鍵盤，同時可以將輸出送到各種不同的設備上，如軟碟、硬碟、甚至是螢幕。不同設備之間的差異將由作業系統來處理，作業系統會調用不同的設備驅動程式來真正地將資料寫到輸出設備上。

與設備無關性緊密相關的是統一命名法。一個檔或設備名將簡單地只是一個字串或一個整數，而完全不依賴於設備。在UNIX中，所有的磁片可以以任何方式集成到檔案系統層次結構中去，使用者也不必知道哪個各字對應著哪個設備。例如，軟碟可以被安裝到目錄/usr/ast/backup下，這時拷貝一個檔至/usr/ast/backup/monday，將把檔拷到軟碟上。使用這種方式，所有的檔和設備的定位都統一通過路徑名來實現。

I/O軟體的另一個重要方面是錯誤處理。總的來說，錯誤應在盡可能接近硬體的地方處理。如果控制器發現一個讀錯誤，它應該儘量進行處理，如果控制器處理不了，則交給設備驅動程式，可能只需重讀一次就可以解決問題。許多錯誤是暫時性的，如磁頭被灰塵阻滯導致的讀錯誤，只需重複一次操作便可以消除。只有在低層軟體處理不了的情況下才通知高層軟體。在許多情況下，低層軟體可以自行處理錯誤而不被高層軟體感知。

同樣重要的另一個問題是同步（阻塞）－非同步（中斷驅動）傳輸。多數物理I/O是非同步傳輸－CPU在啟動傳輸操作後便轉向其他工作，直到中斷到達。如果I/O操作採用阻塞語義，那麼用戶程式設計將簡單許多－發出一條READ命令後，程式將自動被掛起，直到資料被讀到緩衝區中。使實際上是中斷驅動的操作對使用者程式具有阻塞語義則屬於作業系統的責任。

最後一個概念是專用設備和共用設備。某些設備可同時被多個使用者使用，如磁片，多個使用者同時打開多個檔是毫無問題的。另一些設備則在某一時刻只能供一個用戶專用，只有當前用戶使用完畢，其他用戶才能繼續使用，如磁帶機。兩個或多個用戶同時向一台磁帶機寫資料是肯定不行的。專用（非共用）設備帶來了許多問題。相應地，作業系統必須能夠同時處理這兩種設備。

通過將I/O軟體組織成以下四個層次，作業系統可以合理、高效地實現以上目標。

1中斷處理常式（底層）

2設備驅動程式

3與設備無關的作業系統軟體

4使用者層軟體（高層）

這四個層次與圖2-26中所示的四個層次完全對應（這並非偶然）。在隨後的章節中我們將自底向上依次進行討論。本章中我們的重點放在設備驅動程式（第2層），但同時也將涉及其他層次以瞭解各層之間如何協同工作。

3.2.2 中斷處理常式

中斷是應該儘量加以遮罩的一個概念，應該將其放在作業系統的底層進行處理，以便其餘部分盡可能少地與之發生聯繫。遮罩中斷的最好方法是將每一個進行I/O操 作的進程掛起，直至I/O操作結束並發生中斷。進程自己阻塞的方法有：執行信號量的DOWN操作、條件變數的WAIT操作；或接收一條消息等等。

當中斷發生時，中斷處理常式執行相應的操作，以解除相應進程的阻塞狀態。在一些系統中是執行信號量的UP操作。在管程中可能是對一個條件變數執行SIGNAL操作，另外一些系統則可能是向阻塞的進程發一條消息，總之其作用是將剛才被阻塞的進程恢復執行。

3.2.3 設備驅動程式

設備驅動程式中包括了所有與設備相關的代碼。每個設備驅動程式只處理一種設備，或者一類緊密相關的設備。例如，若系統所支援的不同品牌的所有終端只有很細微的差別，則較好的辦法是為所有這些終端提供一個終端驅動程式。另一方面，一個機械式的硬拷貝終端和一個帶滑鼠的智慧化圖形終端差別太大，於是只能使用不同的驅動程式。

在本章的前半部分我們瞭解了設備控制器的功能，知道每個控制器都有一個或多個寄存器來接收命令。設備驅動程式發出這些命令並對其進行檢查，因此作業系統中只有硬碟驅動程式才知道磁碟控制卡有多少個寄存器，以及它們的用途。驅動程式知道使磁片正確操作所需要的全部參數，包括磁區、磁軌、柱面、磁頭、磁頭臂的移動、交叉係數、步進電機、磁頭定位時間等等。

籠統地說，設備驅動程式的功能是從與設備無關的軟體中接收抽象的請求，並執行之。一條典型的請求是讀第n塊。如果請求到來時驅動程式空閒，則它立即執行該請求。但如果它正在處理另一條請求，則它將該請求掛在一個等待佇列中。

執行一條I/O請求的第一步，是將它轉換為更具體的形式。例如對磁片驅動程式，它包含：計算出所請求塊的物理位址、檢查驅動器電機是否在運轉、檢測磁頭臂是否定位在正確的柱面等等。簡言之，它必須確定需要哪些控制器命令以及命令的執行次序。

一旦決定應向控制器發送什麼命令，驅動程式將向控制器的設備寄存器中寫入這些命令。某些控制器一次只能處理一條命令，另一些則可以接收一串命令並自動進行處理。

這些控制命令發出後有兩種可能。在許多情況下，驅動程式需等待控制器完成一些操作，所以驅動程式阻塞，直到中斷信號到達才解除阻塞。另一種情況是操作沒有任何延遲，所以驅動程式無需阻塞。後一種情況的例子如：在有些終端上滾動螢幕只需往控制器寄存器中寫入幾個位元組，無需任何機械操作，所以整個操作可在幾微秒內完成。

對前一種情況，被阻塞的驅動程式須由中斷喚醒，而後一種情況下它根本無需睡眠。無論哪種情況，都要進行錯誤檢查。如果一切正常，則驅動程式將資料包傳送給上層的設備無關軟體。最後，它將向它的調用者返回一些關於錯誤報告的狀態資訊。如果請求佇列中有別的請求則它選中一個進行處理，若沒有則它阻塞，等待下一個請求。

3.2.4 與硬體無關的I/O軟體

儘管某些I/O軟體是設備相關的，但大部分獨立於設備。設備無關軟體和設備驅動程式之間的精確界限在各個系統都不盡相同。對於一些以設備無關方式完成的功能，在實際中由於考慮到執行效率等因素，也可以考慮由驅動程式完成。

圖3-5中羅列的功能一般都是由設備無關軟體完成的。在MINIX中，多數設備無關軟體屬於檔案系統，位於圖2-26的第三層。雖然檔案系統屬於第5章的內容，這裡我們仍需要簡單地介紹一下以瞭解I/O的一些特性。

圖 3-5 設備無關I/O軟體的功能。

設務無關軟體的基本功能是執行適用於所有設備的常用I/O功能，並向使用者層軟體提供一個一致的介面。

作業系統的一個主要論題是檔和I/O設備的命名方式。設備無關軟體負責將設備名映射到相應的驅動程式。在UNIX中，一個設備名，如/dec/tty00 唯一地確定了一個i-節點，其中包含了主設備號（major device number），通過主設備號就可以找到相應的設備驅動程式。i-節點也包含了次設備號（minor device number），它作為傳給驅動程式的參數指定具體的物理設備。

與命名相關的是保護。作業系統如何保護對設備的未授權訪問呢？多數個人電腦系統根就不提供任何保護，所有進程都可以為所欲為。在多數大型主機系統中，使用者進程絕對不允許訪問I/O設備。在UNIX中使用一種更為靈活的方法。對應於I/O設備的設備檔的保護採用通常的rwx許可權機制，所以系統管理員可以為每一台設備設置合理的存取權限。

不同磁片的磁區大小可能不同，設備無關軟體遮罩了這一事實並向高層軟體提供統一的資料塊大小，比如將若干磁區作為一個邏輯塊。這樣高層軟體就只和邏輯塊大小都相同的抽象設備交互，而不管物理磁區的大小。類似地，有些字元設備對位元組進行操作(Modem)，另一些則使用比位元組大一些的單元（網卡），這類差別也可以進行遮罩。

塊設備和字元設備都需要緩衝技術。對於塊設備，硬體每次讀寫均以塊為單元，而使用者程式則可以讀寫任意大小的單元。如果使用者進程寫半個塊，作業系統將在內部保留這些資料，直到其餘資料到齊後才一次性地將這些資料寫到盤上。對字元設備，使用者向系統寫資料的速度可能比向設備輸出的速度快，所以需要進行緩衝。超前的鍵盤輸入同樣也需要緩衝。

當創建了一個檔並向其輸入資料時，該檔必須被分配新的磁片塊。為了完成這種分配工作，作業系統需要為每個磁片都配置一張記錄空閒盤塊的表或點陣圖，但定位一個空閒塊的演算法是獨立於設備的，因此可以在高於驅動程式的層次處理。

一些設備，如CD-ROM記錄器，在同一時刻只能由一個進程使用。這要求作業系統檢查對該設備的使用請求，並根據設備的忙閑狀況來決定是接受或拒絕此請求。一種簡單的處理方法是通過直接用OPEN打開相應的設備檔來進行申請。若設備不可用，則OPEN失敗。關閉獨佔設備的同時將釋放該設備。

錯誤處理多數由驅動程式完成。多數錯誤是與設備緊密相關的，因此只有驅動程式知道應如何處理（如重試、忽略、嚴重錯誤）。一種典型錯誤是磁片塊受損導致不能讀寫。驅動程式在嘗試若干次讀操作不成功後將放棄，並向設備無關軟體報錯。從此處往後錯誤處理就與設備無關了。如果在讀一個用戶檔時出借，則向調用者報錯即可。但如果是在讀一些關鍵系統資料結構時出錯，比如磁片使用狀況點陣圖，則作業系統只能列印出錯資訊，並終止運行。

3.2.5 使用者空間的I/O軟體

儘管大部分I/O軟體屬於作業系統，但是有一小部分是與使用者程式連結在一起的庫常式，甚至是在核外運行的完整的程式。系統調用，包括I/O系統調用通常先是庫常式調用。如下C語句

count = write (fd, buffer, nbytes);

中，所調用的庫函數write將與程式連結在一起，並包含在運行時的二進位程式碼中。這一類庫常式顯然也是I/O系統的一部分。

此類庫常式的主要工作是提供參數給相應的系統調用並調用之。但也有一些庫例稱，它們確實做非常實際的工作，例如格式化輸入輸出就是用庫常式實現的。C語言中的一個例子是Printf函數，它的輸入為一個格式字串，其中可能帶有一些變數，它隨後調用write，輸出格式化後的一個ASCII碼串。與此類似的scanf，它採用與printf相同的語法規則來讀取輸入。標準I/O庫包含相當多的涉及I/O的庫常式，它們作為使用者程式的一部分運行。

並非所有的使用者層I/O軟體都由庫常式構成。另一個重要的類別的就是Spooling系統，Spooling是在多道程序系統中處理獨佔設備的一種方法。例如對於印表機，儘管可以採用打開其設備檔來進行申請，但假設一個進程打開它而長達幾個小時不用，則其他進程都無法列印。

避免這種情況的方法是創建一個特殊的精靈進程（守護進程daemon）以及一個特殊的目錄，稱為 Spooling目錄。列印一個檔之前，進程首先產生完整的待列印檔案並將其放在Spooling目錄下。而由該精靈進程進行列印，這裡只有該精靈進程能夠使用印表機設備檔。通過禁止使用者直接使用印表機設備檔便解決了上述印表機空占的問題。

Spooling還可用於印表機以外的其他情況。例如在網路上傳輸檔常使用網路精靈進程，發送檔前先將其放在一特定目錄下，而後由網路精靈進程將其取出發送。這種檔傳送方式的用途之一是Internet電子郵件系統。Internt通過許多網路將大量的電腦聯在一起。當向某人發送E-mail時，使用者使用某一個程式如send，該程式接收要發的信件並將其送入一個固定的Spooling目錄，待以後發送。整個E-mail系統在作業系統之外運行。

圖3-6總結了I/O系統，標示出了每一層軟體及其功能。從底層開始分別是硬體、中斷處理常式、設備驅動程式、設備無關軟體，最上面是使用者進程。

圖 3-6 I/O系統的層次結構及各層的主要功能。

該圖中的箭頭表示控制流。如當使用者程式試圖從檔中讀一資料塊時，需通過作業系統來執行此操作。設備無關軟體首先在資料塊緩衝區中查找此塊，若未找到，則它調用設備驅動程式向硬體發出相應的請求。使用者進程隨即阻塞直到資料塊被讀出。

當磁片操作結束時，硬體發出一個中斷，它將啟動中斷處理常式。中斷處理常式則從設備獲取返回狀態值並喚醒睡眠的進程來結束此次I/O請求，並使使用者進程繼續執行。

3.3 鎖死

電腦系統中有許多獨佔資源，它們在任一時刻都只能為一個進程使用，如平板式繪圖器、CD-ROM驅動器、CD-ROM記錄器、8mm DAT磁帶機、照相製版機、以及進程表項等。兩個進程同時向印表機輸出將導致一片混亂，兩個進程同時使用同一進程表項則可能導致系統崩潰。正因為如此，所有的作業系統都具有授權一個進程獨立地訪問某一資源的能力。

在許多應用中，一個進程需要獨佔地訪問不止一種資源，例如一家公司的業務是在1米寬的平板式繪圖器上繪製精確的美國地圖，地圖資訊從CD-ROM中讀出。假設進程A申請到了CD-ROM，稍後進程B申請到了繪圖器，現在進程A申請繪圖器導致阻塞等待，進程B申請CD－ROM也導致阻塞。至此兩個進程都被阻塞而且將永遠不能解除。這種狀態就是鎖死，鎖死應該盡力加以避免。

除了申請獨佔設備外，其他情況也可能導致鎖死。例如在一個資料庫系統中，為了避免競爭，可能將若干記錄加鎖。若進程A對記錄R1加鎖，進程B對記錄R2加鎖，隨後兩進程又各自試圖將對方的記錄加鎖，這時也將導致鎖死。因此，軟、硬體資源都有可能導致鎖死。

本節將研究鎖死如何產生，以及如何加以預防和避免。為方便起見，將以申請物理設備為例，例如，磁帶機、CD－ROM和繪圖器，但有關原理和演算法同樣適用于其餘類型的鎖死。

3.3.1 資源

進程對設備、檔等獲得獨佔性的訪問權時有可能會發生鎖死，為了盡可能地通用化，我們將這種需排它使用的物件稱為資源。資源可以是硬體設備（如磁帶機），或一組資訊（如資料庫中一個加鎖的記錄）。電腦中通常有多個資源。有些類型的資源有多個相同的實例，如三台磁帶機。當某一資源有若干拷貝時，其中任一個均可用來滿足對資源的請求。簡言之，資源是在任何時刻只能被單個進程使用的任何物件。

資源有兩類：可剝奪式和不可剝奪式。可剝奪式資源可從擁有它的進程處剝奪而沒有任何副作用，記憶體是一類可剝奪資源。例如，一個系統有512K使用者記憶體和一台印表機。若兩個512K記憶體的進程都要進行列印，進程A申請並得到了印表機，然後開始計算要列印的值。在它未完成計算任務之前，它的時間片用完並被換出。然後進程B開始運行並申請印表機，但未成功。此時有潛在的鎖死危險。因為進程A擁有印表機，而進程B佔有記憶體。然而幸運的是通過將進程B換出、進程A換人便可以實現剝奪進程B的記憶體。於是，進程A繼續運行並執行列印任務，然後釋放印表機。這個過程將不會發生鎖死。

相反地，不可剝奪資源無法在不導致相關計算失敗的情況下將其從屬主進程處剝奪。若一個進程已開始列印，那麼剝奪其佔用的印表機並分配給另一進程將使列印結果一片混亂。印表機是不可剝奪的。

總的來說，鎖死與不可剝奪資源有關，涉及可剝奪資源的潛在的鎖死通過在進程間重新分配資源通常能夠化解。所以，我們的重點放在不可剝奪資源。

使用一個資源的事件順序是：

1 申請資源

2 使用資源

3 釋放資源

若申請時資源不可用，則申請進程被迫等待。在一些作業系統中，申請失敗的進程自動被阻塞，在資源可用時再被喚醒。在其他系統中，資源申請失敗將返回一個錯誤碼，由申請進程等待一段時間後重試。

3.3.2 鎖死原理

鎖死的定義是：

若一個進程集合中的每一個進程都在等待只能由本集合中的另一個進程才能引發的事件，則這種情況被視為鎖死。

由於所有的進程都在等待，所以沒有一個進程能夠觸發那個（些）能夠喚醒本集合中另一個進程的事件，於是所有進程都將永遠地等待下去。

多數情況下，進程是在等待本集合中另一個進程釋放的資源。換言之，每個進程都在等待另一個進程所佔有的資源。但因為所有進程都無法運行，因而無法釋放資源，於是所有進程都不能被喚醒。至於進程數和佔用及申請的資源數並不重要。

鎖死的條件

Coffman 等人（1971）總結出了鎖死發生的四個條件：

1 互斥條件，每一資源或者被分配給一個進程，或者空閒。

2 保持和等待條件，已分配到了一些資源的進程可以申請新的資源。

3 非剝奪條件，已分配給一進程的資源不可被剝奪，只能被佔有它的進程顯式地釋放。

4 迴圈等待條件，系統必然有一條由兩個或兩個以上的進程組成的迴圈鏈，鏈中的每一個進程都在等待相鄰進程佔用的資源。

以上四個條件是鎖死發生的必要條件，只要一條或多條不成立，鎖死就不會發生。

鎖死模型

Holt（1972）提出了用有向圖可以建立以上四個條件的模型。圖中有兩類節點：圓形表示進程，方形表示資源。從資源節點到進程節點的弧表示該資源已被進程佔用。在圖 3－7(a)中，資源R當前被進程A佔用。

圖 3-7 資源配置圖。 (a)佔用一個資源 (b)請求一個資源 (c)鎖死

從進程到資源的弧表示進程當前正申請該資源並處於阻塞等候狀態。在圖3－7(b)中，進程B正在等待資源S。圖3－7(c)示出了鎖死狀態：進程C等待資源T，資源T被進程D佔用，進程D又在等待被進程C佔用的資源V。於是兩個進程都將等待下去。圖中的環表示涉及到這些進程和資源的鎖死。本例中的環為C－T－D－V－C。

再看資源圖的用法。假設有三個進程A，B，C及三個資源R，S，T。三個進程對資源的申請和釋放如圖3－8(a)－(c)所示。作業系統可以選擇任何一個非阻塞的進程來執行，因此它可以選擇A執行直至結束，然後運行B，最後運行C。

圖 3－8 一個鎖死如何發生和避免的例子。

這個順序不會導致鎖死（因為不存在資源的競爭）。當進程串列執行時，不可能在一個進程等待I/O時，另一個進程佔用CPU進行計算。所以嚴格的串列操作可能不是最優的。另一方面，如果進程完全不執行I/O操作，則最短作業優先調度將優於時間片調度，所以在某些情況下，串列執行可能是最優的。

考慮進程操作包含I/O和計算，則時間片法是一種合理的調度演算法。資源申請順序可能如圖3－6（d）所示。若按此循序執行，則相應的資源圖示於圖3－8（e）-（j）。在發生請求（4）之後，進程A阻塞等待S，如圖3－8（h）所示，隨後的兩步中B和C也將阻塞，最終導致產生迴圈和鎖死，圖3－8(j)中所示。

然而如前所述，作業系統不必局限於按某種特定次序來執行這些進程。對於一個可能導致鎖死的資源請求，作業系統可以簡單地不批准該請求，並由此將進程掛起（即不予調度）直到安全狀態。在圖3－8，設作業系統知道可能導致鎖死，它可以不把資源S分配給B，於是B被掛起。假設只運行過程A和C，則資源申請和釋放過程如圖3－8(k)，而非3-8(d)。該過程的資源圖示於圖 3－8(l)-(q)，其中無鎖死發生。

第(q)步執行完後，可將資源S分配給B，因為A已經結束，C已得到它所需要的所有資源，即使B最終因申請T而等待也不會發生鎖死，B只是需等C結束而已。

本章後邊我們將研究一個詳細的演算法，以達到對資源進行分配而不會鎖死。此處需要明確的是資源圖可用作分析一給定的申請/釋放序列是否將導致鎖死的一種工具。我們只需按申請和釋放的順序一步步地走下來，最後檢查其中是否包含有環路。如果有，則發生鎖死；反之則無。雖然我們的例子中只涉及一類資源僅包含一個實例，但以上原則完全可以推廣到同種資源含有多個實例的情況。

概括而言，處理鎖死有四種策略：

1忽略該問題。

2檢測鎖死並恢復

3謹慎地對資源進行動態分配，避免鎖死

4通過破壞上述四個必要條件來預防鎖死發生

隨後四節將依次討論這四種方法。

3.3.3 鴕鳥演算法

最簡單的方法是象鴕鳥一樣對鎖死視而不見。對該方法各人的看法不同。數學家認為不管花多大代價也要徹底防止鎖死的發生；工程師們則要瞭解鎖死發生的頻率、系統因其他原因崩潰的頻率、以及鎖死有多嚴重，如果鎖死平均每50年發生一次，而系統每個月會因硬體故障、編譯器錯誤或作業系統錯誤而崩潰一次，那麼大多數工程師不會不惜工本地去消除鎖死。

更具體地說，UNIX和MINIX潛在地受到某些鎖死的威助，不過這些鎖死從來沒有發生過，甚至從沒有被檢測到過。舉個例子，系統中進程的數目受進程表項多少的制約，進程表項是有限的資源，如果一個FORK調用由於進程表用完而失敗，那麼一種合理的辦法是等待一段隨機的時間後重試。

現假設一個UNIX系統的進程表有100項，有10個進程在執行，每一個都要創造12個子進程。在每個進程創建9個子進程後，進程表項被全部用完。則這10個進程將進入一個無休止的迴圈：執行FORK，失敗，等待一段時間後執行FORK，再失敗 ……這實際上是鎖死。發生這類事件的概率是很小的，但它的確存在！我們難道會為了消除這種狀況就放棄進程、FORK這些概念和方法嗎？

打開檔的最大數目受i-節點表大小的限制，所以當i-節點表滿時會發生類似的問題。磁片上的對換空間也是另一種有限的資源。實際上，幾乎作業系統中的每一種表格都代表了一種有限的資源。難道我們會因為可能出現類似的鎖死而拋掉這一切嗎？

UNIX處理這一問題的辦法是忽略它，因為大多數用戶寧可在極偶然的情況下發生鎖死，也不願限制每個使用者只能創建一個進程、只能打開一個檔等等。如果鎖死可以不花什麼代價就能夠解決，則什麼問題都沒有了。問題是，這種代價很大，而且常常給用戶帶來許多不便的限制。於是我們不得不在方便性和正確性之間作出折衷。

3.3.4 鎖死檢測和恢復

第二種技術是鎖死檢測和恢復，採用這種技術時，系統只需監視資源的申請和釋放。每次資源被申請或釋放時，資源圖隨之刷新，同時檢測是否存在環路。如果存在一個環，則環中的一個進程被撤銷，如果仍不能破除鎖死，則撤銷另一個進程。如此往復直至環路被破除。

一種更簡陋的方法甚至不維護資源圖，而是週期性地檢測進程是否連續阻塞超過一定時間，如一小時。一旦發現這樣的進程則將其撤銷。

這種策略常被用於大型主機，尤其是批次處理系統。因為在批次處理系統中，取消一個進程然後重新開機它通常是可以接受的。但必須注意要將該進程修改了的檔案修復到初始狀態，並消除掉它所導致的所有副作用。

3.3.5 鎖死預防

第三種策略是對進程施加適當的限制以從根本上消除鎖死。Coffman等人（1971）提出的四個條件為解決該問題提供了線索。如果能夠保證四個條件中至少有一個不成立，則鎖死將不會發生（Havender，1968）。

先考慮互斥使用條件。如果資源不被單一進程獨佔，則鎖死肯定不會發生。然而印表機這樣的設備不允許兩個進程同時使用，通過借助Spooling技術可以允許多個進程同時產生列印資料。該模型中，唯一真正申請物理印表機的進程是列印精靈進程，由於它決不會申請別的設備，所以不會因印表機而發生鎖死。

但並非Spooling技術適用於所有的設備（如進程表）。而且在Spooling時對磁碟空間本身的競爭可能導致鎖死。例如兩個進程分別佔用了可用的Spooling磁碟空間的一半而等待更多的空間，則必將導致鎖死。具體地說，如果列印精靈進程被設計為不等全部輸出放入Spooling目錄下就開始列印，則它可能在列印完第一批資料後空閒幾個小時等待剩餘資料就位。為了避免這種現象，它一般被設計成等完整的輸出檔就緒後才開始列印，這時便可能發生前述的情況，即系統將鎖死。

第二個條件似乎更容易解決。只要禁止已擁有資源的進程再申請其他資源便可以消除鎖死。一種實現方法是規定所有進程在開始執行前申請所需的全部資源。如果所需的資源全部可用則進行分配，於是它肯定能夠運行結束。如果有一個或多個資源正被使用，則不進行分配，進程阻塞。

一個直接的問題是許多進程直到運行時才知道它需要多少資源。另一個問題是這種方案的資源利用率無法優化。例如一個進程先從輸入磁帶上讀取資料，進行一小時的分析，最後將輸出寫到一條輸出磁帶上，同時將其在繪圖器上繪出。如果所有資源被提前申請，則有一小時時間輸出磁帶機和繪圖器無法使用。

另一種方案是當進程申請資源時先暫時釋放其當前佔用的所有資源，只有對該資源申請成功才收回其原先佔用的資源。

消除第三個條件（不可剝奪）比第二個困難。若一個進程已分配到一台印表機正在進行列印，如果因為它需要的繪圖器無法得到而強制性地將印表機剝奪掉，這將導致一片混亂。

消除最後一個條件－迴圈等待有幾種方法。一種是保證每一個進程在任何時刻只能佔用一個資源，如果要申請第二個，它必須先釋放第一個。若進程正在將一個大檔從磁帶機上讀出並送到印表機列印，則這種限制是不可接受的。

另一種方法是將所有資源賦予一個全域編號，如圖3－9(a)所示。進程申請資源必須按照編號的順序。進程可以先申請印表機，後申請磁帶機，但不可以先申請繪圖器，後申請印表機。

圖 3－9 （a）資源被賦予數字序號。（b）一個資源圖。

按此規則，資源配置圖中肯定不會出現環路。例如圖3－9(b)所示的兩個進程的例子，只有在A申請資源j，B申請資源i的情況下會發生鎖死。設i和j是不同資源，它們將具有不同的數值。若i>j，則A不允許申請j；若i<j，則B不允許申請i。不論哪種情況都不可能發生鎖死。

對於多進程的情況，以上邏輯依然成立。對於每一種情況，總有一個被分配的資源是編號最高的。佔用該資源的進程不可能申請其他已被佔用的各種資源，它或者將執行完畢，或者申請編號更高的資源，而編號更高的資源肯定是可獲得的。最終它將結束並釋放所有資源，這時另一個佔有最高編號資源的進程也可以執行完。依次類推，所有進程都可以執行完畢，因而沒有鎖死發生。

該演算法的一個變種是擯棄必須按昇冪申請資源的限制，而僅要求不允許進程申請編號比當前所佔有資源編號低的資源。若一個進程起初申請9號和10號資源，隨後將其釋放，它實際上相當於從頭開始，所以沒有必要阻止它現在申請1號資源。

儘管對資源編號的方法消除了鎖死的問題，但幾乎找不出一種使每個人都滿意的編號次序。當資源包括進程表項、Spooling磁碟空間、加鎖的資料庫記錄及其他抽象資源時，潛在的資源及各種不同用途的數目會變得如此之大，以至於使編號法根本無法使用。

鎖死預防的各種方法如圖3－10所示。

圖 3－10 鎖死預防途徑小結。

3.3.6 鎖死避免

從圖3-8中，我們看到鎖死避免不是通過對進程隨意強加一些規則，而是通過對每一次資源申請進行認真的分析來判斷它是否能安全地分配。問題是：是否存在一種演算法總能作出正確的選擇從而避免鎖死？答案是肯定的，但條件是必須事先獲得一些特定的資訊。本節我們將討論幾種鎖死避免的方法。

單種資源的銀行家演算法

Dijkstra (1965)提出了一種能夠避免鎖死的調度演算法，稱為銀行家演算法。它的模型基於一個小城鎮的銀行家，他向一群客戶分別承諾了一定的貸款額度。在圖3-11（a）中我們看到4個客戶，每個客戶都有一個貸款額度，銀行家知道不可能所有客戶同時都需要最大貸款額，所以他只保留10個單位的資金來為客戶服務，而不是22個單位。這裡將客戶比作進程，貸款比作設備，銀行家比作作業系統。

圖 3－11 三種資源配置狀態。（a）安全（b）安全（c）不安全

客戶們各自做自己的生意，在某些時刻需要貸款。在某一時刻，具體情況如圖3-11（b）所示。客戶已獲得的貸款（已分配的磁帶機）和可用的最大數額貸款稱為與資源配置相關的系統狀態。

一個狀態被稱為是安全的，其條件是存在一個狀態序列能夠使所有的客戶均得到其所有的貸款（即所有的進程得到所需的全部資源並終止）。圖3-11（b）所示的狀態是安全的，因為在有2單位資金可用的情況下，銀行家可以延遲除Marvin之外的所有請求，這樣便可以使Marvin運行結束，然後釋放所有的4個單位資金。如此這樣下去便可以滿足Snganne或者Barbarn的請求，等等。

考慮假如給Barbara另一個她申請的資源，如圖3-11（b），則我們得到如圖3-11（c）所示的狀態，該狀態是不安全的。如果忽然所有的客戶都申請，希望得到最大貸款額，而銀行家無法滿足其中任何一個的要求，則發生鎖死。不安全狀態並不一定導致鎖死，因為客戶未必需要其最大貸款額度，但銀行家不敢抱這種僥倖心理。

銀行家演算法就是對每一個請求進行檢查，檢查如果滿足它是否會導致不安全狀態。若是，則不滿足該請求；否則便滿足。檢查狀態是否安全的方法是看他是否有足夠的資源滿足一個距最大需求最近的客戶。如果可以，則這筆投資認為是能夠收回的，然後接著檢查下一個距最大需求最近的客戶，如此反復下去。如果所有投資最終都被收回，則該狀態是安全的，最初的請求可以批准。

資源軌跡圖

以上演算法描述了一種資源的情況（例如僅有磁帶機或僅有印表機，而不是多種資源）。在圖3-12中，我們看到一個處理兩個進程和兩種資源（印表機和繪圖器）的模型。橫軸表示進程A的指令執行過程，縱軸表示進程B的指令執行過程。進程A在I1處請求一台印表機，在I3處釋放，在I2處申請一台繪圖器，在I4處釋放。進程B在I5到I7之間需要繪圖器，在I6到I8之間需要印表機。

圖 3－12 兩個進程的資源軌跡圖。

圖中的每一點都示出了兩個進程的狀態。初始點為P，若A先運行，則在A執行一段指令後到達q，在q點若B開始運行，則軌跡向垂直方向移動。在單處理機情況下，所有路徑都只能是水準或垂直方向的。同時運動方向一定是向右或向上，而不會是向左或向下，因為進程的執行不可能後退。

當進程A由r向s移動，穿過I1線時，它請求印表機並獲得。當進程B到達t時，它申請繪圖器。

圖中的陰影部分很重要，打著從左下到右上斜線的部分表示在該區域中兩個進程都擁有印表機，而互斥使用的規則決定了不可能進入該區域。同樣的，另一種斜線的區域表示兩個進程都擁有繪圖器，且同樣不可進入。

如果系統一旦進入由I1 I2和I5 I6組成的矩形區域，那麼最後一定會到達I2和I6的交叉點，此時就發生鎖死。在該點處，A申請繪圖器，B申請印表機，而且這兩種資源均已被分配。這整個矩形區域都是不安全的，因此絕不能進入這個區域。在t處唯一的辦法是運行進程A直到I4，過了I4後則可以按任何路線前進，直到終點u。

多種資源的銀行家演算法

以上資源軌跡圖的方法很被難被擴充到系統中有任意數目的進程、任意種類的資源，並且每種資源有多個實例的情況。但銀行家演算法可以被推廣用來處理這個問題。圖3-13示出了其工作原理。

圖 3－13 多種資源的銀行家演算法。

在圖3-13中我們看到兩個矩陣。左邊的顯示出對5個進程分別已分配的各種資源數，右邊的則顯示了使各進程運行完所需的各種資源數。與單種資源的情況一樣，各進程在執行前給出其所需的全部資源量，所以系統的每一步都可以計算出右邊的矩陣。

最右邊的三個向量分別表示總的資源E、已分配資源P，和剩餘資源A。由E可知系統中共有6台磁帶機，3台繪圖器，4台印表機和2台CD-ROM。由P可知當前已分配了5台磁片機，3台繪圖器，2台印表機和2台CD-ROM。該向量可通過將左邊矩陣的各列相加得到，剩餘資源向量可通過從資源總數中減去已分配資源數得到。

檢查一個狀態是否安全的步驟如下：

1查找右邊矩陣中是否有一行，其未被滿足的設備數均小於或等於向量A。如果找不到，則系統將鎖死，因為任何進程都無法運行結束。

2若找到這樣一行，則可以假設它獲得所需的資源並運行結束，將該進程標記為結束，並將資源加到向量A上。

3重複以上兩步，直到所有的進程都標記為結束。若達到所有進程結束，則狀態是安全的，否則將發生鎖死。

如果在第1步中同時存在若干進程均符合條件，則不管挑選哪一個運行都沒有關係，因為可用資源或者將增多，或者在最壞情況下保持不變。

圖3－13中所示的狀態是安全的，因為進程B現在在申請一台印表機，可以滿足它的請求，而且保持系統狀態仍然是安全的（進程D可以結束，然後是A或E，剩下的進程最後結束）。

假設進程B獲得一台印表機後，E試圖獲得最後的一台印表機，若分配給E，可用資源向量將減到（1 0 0 0），這時將導致鎖死。則顯然E的請求不能立即滿足，必須延遲一段時間。

該演算法最早由Dijkstra於1965年發表。從那之後幾乎每本作業系統的專著都詳細地描述它，許多論文的內容也圍繞該演算法，但很少有作者指出該演算法缺乏實用價值。因為很少有進程能夠在運行前就知道其所需資源的最大值，而且進程數不是固定的，往往在不斷地變化，況且原本可用的資源也可能突然間變成不可用（如磁帶機可能會壞掉）。

總之，鎖死預防的方案過於嚴格，鎖死避免的演算法又需要無法得到的資訊。如果你能想到一種理論上和實際中都適用的通用解法，那麼就可以在電腦科學的雜誌上發表一篇論文。

對於特殊的應用有許多很好的演算法。例如在許多資料庫系統中，常常需要將若干記錄上鎖然後進行更新。當有多個進程同時運行時，有可能發生鎖死。

常用的一種解法是兩階段上鎖法。第一階段，進程試圖將其所需的全部記錄加鎖，一次鎖一個記錄。若成功，則資料進行更新並解鎖。若有些記錄已被上鎖，則它將已上鎖的記錄解鎖並重新開始執行，該解法有點類似提前申請全部資源的方法。

但這種方法不通用，在即時系統和程序控制系統中不能夠因為資源不可用而將進程中途終止並重新執行。同樣若一個進程已進行過網路消息的讀寫、更新檔、或其他不宜重複的操作，則將進程重新從頭執行是不可接受的。該演算法僅適用於那些在第一階段可以隨時停止並重新執行的程式，遺憾的是並非所有的應用都可以按這種方式組織。

3.4 MINIX I/O系統概述

MINIX的I/O結構如圖3－6所示。其中，上邊四層對應於圖2－26中所示的MINIX的四層結構。以下章節中我們將對每一層作大概的介紹重點放在驅動程式。中斷處理已在第二章作過介紹，設備無關的I/O將在第五章檔案系統中討論。

3.4.1 MINIX的中斷處理常式

許多設備驅動程式在啟動一些I/O設備後阻塞，等待某種消息到達。這種消息往往由該設備的中斷處理常式產生。另外一些設備驅動程式不啟動物理的I/O操作（如從RAM盤中讀數據，然後寫到一個記憶體映射的顯示器），也不使用中斷，所以也不等待從I/O設備發來的消息。前一章中已詳細討論了中斷產生消息以及造成任務切換的機制，我們在這裡不再贅述。中斷處理常式除了產生一條消息之外，還進行一些最底層的I/O處理工作。我們將在此概要地加以討論，細節將放在討論具體設備時敘述。

對於磁片設備，輸入輸出通常僅僅是命令設備執行某操作，然後等待該操作結束。大部分工作由磁碟控制卡完成，而中斷處理常式作的工作很少，硬碟的整個中斷處理常式只有3行代碼，所作的I/O操作僅僅是從控制器讀一個位元組從而確定控制器的工作狀態。如果所有的中斷處理都這麼簡單，那我們的工作就容易得多了。

然而有時低層中斷處理常式有更多的事情可做。消息傳遞的代價是較高的，所以對於中斷本身很頻繁、而每次中斷所處理的I/O操作量又很少的情況，更好的辦法是讓中斷處理常式作更多的工作，而將向對應的任務常式發送消息推遲到後面某一個中斷到來之時，即使相應驅動器任務會有較多工作可做。MINIX的時鐘就採用這種方法。並非每一個時鐘滴答都有許多事情要作，許多時候除了維護系統時間外幾乎無事可做。這種情況下，無需在每次時鐘中斷時都向時鐘任務（系統進程）發送消息。時鐘中斷處理常式每次遞增一個變數pending\_ticks，當前時間就是時鐘任務上次運行時記錄的值加上pending\_ticks所得的和。當時鐘任務接收到一條消息被喚醒時，它將pending\_ticks的值加到它所維護的時鐘變數上，並將pending\_ticks清零。時鐘中斷處理常式檢查幾個變數，當發現時鐘任務確實有工作可幹時，例如需要發送時鐘鬧鐘信號或進行進程調度，才向其發送消息，。它也有可能向終端任務發送消息。

終端任務採用另一種稍有不同的方案。它要處理幾種不同的硬體，包括鍵盤和RS－232串口線。這些設備每個都有其自己的中斷處理常式。鍵盤是典型的每次中斷I/O操作很少的設備。在PC機上當一個鍵被按下或釋放時發生一個中斷，若不考慮同時按下SHIFT和CTRL之類的物殊鍵，可以認為平均每次中斷接收到半個字元。因為終端任務無法對半個字元進行處理，所以必須等相應的資訊完全獲得之後才能向它發送一條消息。我們將在本章較後部分對此進行詳細說明，這裡我們只提及鍵盤中斷處理過程完成從鍵盤讀數據的操作並將可以忽略的事件濾掉，比如一個普通鍵的釋放（但諸如SHIFT這樣的特殊鍵是不能被忽略的）。所收到的鍵碼被放在一個佇列中供終端任務隨後進行處理。

鍵盤中斷處理常式與前述的中斷處理常式模型不完全一致，它並不向相關任務發送消息，而是將鍵碼放在一個佇列中，然後修改tty\_timeout變數，而時鐘中斷處理常式要讀這個變數。當中斷並不改變鍵碼佇列時，tty\_timeout也不變。在下一個時鐘滴答，若鍵碼佇列已發生變化，則時鐘中斷處理常式將向終端任務發一條消息。其他終端類型的中斷處理常式，例如RS－232串口線等，也採用相同的工作方式。在收到一個字元後終端任務很快就會收到一條消息，但當字元到達速率很快時沒有必要為每個字元產生一條消息，可以將若干字元累積起來放在一條消息中處理。而且當終端任務收到一條消息時，它將檢查所有的終端設備。

3.4.2 MINIX的設備驅動程式

MINIX中的每一類設備都有一個單獨的I/O處理任務（設備驅動程式）。這些驅動程式是完整的進程，每個都有其自己的狀態、寄存器、堆疊等。設備驅動程式相互之間進行通信，同時也與檔案系統進行通信。這些通信採用與MINIX進程間通信完全一樣的消息傳遞機制。每一個設備驅動程式放在一個單獨的原始檔案裡，如時鐘驅動程式放在clock.c中，其他如RAM盤、硬碟、軟碟等都各自有一個驅動程式原始檔案，其間的公用常式則放在driver.c 中。這在某種意義上講是將圖3-6中的設備驅動程式層又分為兩個子層。這種將與設備相關部分和與設備無關部分分開的方法能夠更靈活地處理不同類型的硬體設定。儘管不同類型的磁片驅動程式共用了一部分代碼，但它們各自作為獨立的進程運行。

終端驅動程式代碼的組織方法與此類似，與硬體無關的代碼放在tty.c中，而支援各種設備的代碼則分別放在各自獨佔的檔中，包括記憶體映射的控制台、鍵盤、串口線以及虛擬終端等，但對終端而言，是一個單獨的進程支援所有這些設備。

對於磁片和終端這樣的設備組，不僅有原始檔案，同時還有標頭檔，例如driver.h支援所有的塊設備驅動程式，tty.h則支持所有類型的終端設備。

設備驅動程式與其他進程的不同在於設備驅動程式全部被鏈入核心，所以它們都共用一個公用的位址空間。這樣一來，若幾個設備驅動程式共用一個過程，則在執行代碼中只存在該過程的一個拷貝。

這種設計方法高度地模組化，並保持了較高的效率，這也是與LINUX的幾處本質區別之一。MINIX中進程通過向檔案系統進程發送消息來讀一個檔，而檔案系統則向磁片驅動程式發送一條消息來請求所需的資料塊。這一順序過程示於圖3－14(a)。採用消息傳遞機制，我們可以使系統的各部分按一種標準的方法進行交互，同時將設備驅動程式放在核心空間將使得它們在需要時很容易就訪問到進程表以及其他關鍵資料結構。

圖3－14 使用者 － 系統通信的兩種組織方式。

在UNIX中，進程都有兩個部分：一個使用者空間部分和一個核心空間部分，如圖3－14(b)所示。當執行系統調用時，作業系統以一種特殊的方式從使用者空間部分切換到核心空間部分，這種結構是MULTICS的遺留物。在MULTICS中，這種切換是普通的程序呼叫，而不是像UNIX那樣陷入後將使用者部分狀態保存起來。

UNIX中的設備驅動程式只是能被進程的核心空間部分調用的核心過程。驅動程式需要等待一個中斷時，它調用一個核心過程從使自己睡眠，直到某一中斷處理常式將它喚醒，注意這裡睡眠的只是使用者進程自己，因為核心部分和使用者部分是一個進程的完全不同的兩個實體。

在作業系統的設計者中，關於整體式系統（如UNIX）和按進程構造的系統（如MINIX）的爭論永無休止。MINIX的結構更好（更模組化），各部分之間的介面更清晰，也更容易擴展到分散式系統。而UNIX則更高效，因為程序呼叫比消息傳遞要快。MINIX被分成許多進程，因為我們相信隨著電腦性能的提高，清晰的軟體結構更為重要。

本章中將討論RAM盤、硬碟、時鐘和終端的驅動程式。 MINIX實際上也包括軟碟驅動程式，但此處不予詳細討論。MINIX發佈的軟體中包含有其他一些設備的驅動程式，如RS－232串口線、SCS1介面、CD－ROM、乙太網卡、以及音效卡等等，將MINIX重新編譯即可將它們鏈入系統中。

這些任務與系統中其他部分的介面是一樣的：將請求的消息發給相關的任務，消息中的若干域用來存放操作碼（例如READ或WRITE）和相關的參數。任務則力圖執行收到的操作請求並返回一應答消息。

塊設備的請求和應答消息的域結構示於圖3－15。請求消息包括：發送或接收資料的緩衝區位址。應答消息包括相應的狀態資訊，它們供請求的進程驗證請求是否被正確地執行。字元設備的域與此類似，但對各個不同的服務可能略有不同。例如發給時鐘任務的消息中包含時間，而發給終端任務的消息則包含一資料結構指標，該資料結構中保存終端的各種配置資訊，如行編輯按鍵、刪除鍵和抹行符等。

圖3－15 從檔案系統發送到塊設備驅動程式的消息的各個域，以及應答消息的各個域。

各設備驅動程式的任務是接收其他的進程發來的請求並執行之。通常情況下是來自檔案系統的請求。所有的塊設備進程都是首先獲取消息，然後執行和返回應答資訊，出於簡單起見，這樣的處理過程是完全串列的，沒有任何多道處理的成分。當發出一個硬體請求時，設備任務進程執行一個RECEIVE操作，指明它只接收中斷消息，新來的請求消息將被保持到當前的工作結束為止。終端任務略有不同，因為單個任務要為若干設備服務，所以可以在從串列線上讀數據的同時接收新的從健盤輸入的請求。但每一個設備都必須執行完上一個請求後才能開始另一個新的請求。

所有塊設備驅動程式中的主程序都如圖3－16所示。當系統啟動時，所有的驅動程式輪流進行內部資料結構的初始化，然後便阻塞等待接收消息。當收到一條消息後，先將調用者標識保存下來，然後開始為請求服務。其中對不同操作將調用不同的過程。當服務結束後，則向調用者發回應答，最後再繼續等待下一個請求。

message mess; /\* 訊息緩衝區 \*/

void io\_task() {

initialize(); /\* 只在系統初始化期間作一次 \*/

while (TRUE) {

receive(ANY, &mess); /\* 等待一個請求 \*/

caller = mess.source; /\* 消息來源 \*/

switch(mess.type) {

case READ: rcode = dev\_read(&mess); break;

case WRITE: rcode = dev\_write(&mess); break;

/\* 這裡還有其他分支，如OPEN、CLOSE及IOCTL \*/

default: rcode = ERROR;

}

mess.type = TASK\_REPLY;

mess.status = rcode; /\* 執行結果代碼 \*/

send(caller, &mess); /\* 向調用者發送應答消息 \*/

}

}

圖3－16 一個I/O任務主過程的框架。

設備驅動程式中的每一個dev\_xxx過程執行一種該驅動程式能做的操作。它返回一個狀態碼，該狀態碼包含在應答消息中的REP\_STATUS域中。若其值為負，則表示這是一個錯誤碼，為零或正值時指明傳輸的位元組數。該值有可能與請求的位元組數不等。如遇到檔案結尾時，所得到的位元組數數就可能小於請求的位元組數。在終端上，最大只能返回一行的字元數。

3.4.3 MINIX中與設備無關的I/O軟體

在MINIX中所有與設備無關的軟體都包含在檔案系統進程中。因為I/O系統與檔案系統的聯繫非常緊密，所以也可以合併到一個進程中。檔案系統執行的功能示於圖3－5中，其中不包括對獨佔設備的申請和釋放。目前的MINIX中不包含對獨佔設備的支援，但以後在需要時可以很容易地加到相關設備的驅動程式中。

檔案系統的工作包括與驅動程式、緩衝、資料塊分配等的介面，同時也處理對i-節點、目錄、當前被安裝檔案系統的保護和管理等，檔案系統將在第5章詳細討論。

3.4.4 MINIX中使用者級I/O軟體

本章前邊給出的框架在這裡也適用，系統提供了必要的庫常式來執行系統調用和所有POSIX標準要求的C函數，例如格式化輸入和輸出函數printf和scanf。標準的MINIX配置包括一個Spooling精靈進程lpd，它處理所有lp命令提交的列印檔案。標準的MINIX發行套裝軟體含許多支援網路功能的精靈進程，網路操作需要一些本書中的MINIX不提供的作業系統支援，但MINIX可以很容易被重新編譯，從而加入網路服務進程，它與記憶體管理進程和檔案系統進程一樣作為使用者進程運行。

3.4.5 MINIX的鎖死處理

MINIX繼承了UNIX的鎖死處理辦法：僅僅簡單地忽略它。儘管將一台標準DAT磁帶機裝上使用不會帶來任何問題，但是MINIX還是不支援任何獨佔的I/O設備。簡言之，鎖死可能發生的唯一環節是在使用隱含的共用資源時，例如進程表項、i-節點表項等等，沒有一種已知的鎖死演算法能夠解決這種非顯式申請資源的問題。

實際上，MINIX在幾個地方都採取了非常謹慎的措施來避免鎖死發生，主要的一點是檔案系統與記憶體管理程式之間的交互。在執行EXEC系統調用時，記憶體管理程式通過向檔案系統發送消息來讀入可執行檔。如果此時檔案系統正忙，則記憶體管理程式將阻塞；如果檔案系統隨後試圖向它發送消息時，檔案系統也因此而阻塞，則鎖死將發生。

其解決辦法是不允許檔案系統向記憶體管理程式發“請求”消息，而只發“應答”消息。這其中有一點小的例外，即在檔案系統啟動時，它向記憶體管理器報告RAM盤的大小，而此時記憶體管理器一定正在等待該消息到達。

不通過作業系統支援也可以對設備和檔加鎖，檔案名可以當作一個真正的全域變數，其存在與否可以被所有其他進程感知。在MINIX和多數UNIX系統中都存在一個特殊的目錄/usr/spool/locks供進程創建加鎖檔，以標誌其正在使用的資源。MINIX檔案系統也支持POSIX風格的建議鎖機制。但是MINIX和POSIX的鎖都不是強制性的，這取決於進程的行為，而且沒有什麼可以阻止一個進程使用另一個進程加鎖的資源。這與資源的剝奪並不完全一樣，它並不阻止第一個進程繼續使用該資源，亦即不存在互斥訪問。進程的這種不正常行為可能會搞得一片混亂，但不會導致鎖死。

3.5 MINIX中的塊設備

下面我們回到本章的重點－設備驅動程式。首先將討論塊設備共性的部分，隨後將詳細討論RAM盤、硬碟和軟碟。RAM盤是一個好的開端，因為它具有塊設備的所有共性，同時又省略了真正的I/O操作－－它只是記憶體的一個部分。硬碟展示了一個真實的磁片驅動程式的特徵。有人會誤認為軟碟比硬碟容易一些，但事實上不是這樣。我們不打算討論軟碟的完整細節，但將指出其中的複雜之處。

在塊設備之後將討論其他類型的設備驅動程式。時鐘是所有系統都必備的，而且與其他設備截然不同，它還有一個有趣的例外：它既不屬於塊設備，也不屬於字元設備。最後是終端驅動程式，它在所有的系統中都很重要，而且是典型的字元設備。

以下每一部分都將描述相關的硬體、驅動程式的有關軟體原理、實現的概述、以及代碼，這種安排將使這部分內容對那些不願閱讀代碼的讀者也頗有價值。

3.5.1 MINIX中塊設備驅動程式概述

我們提到過所有I/O服務進程的主框架都有類似的結構。MINIX至少有三種設備驅動程式（RAM盤、軟碟、硬碟），此外還可以將CD－ROM和一個SCSI設備驅動程式加進來。儘管每一種設備驅動程式都作為獨立進程運行，但由於它們都是整個可執行核心代碼的組成部分，所以有可能使它們共用相當數量的代碼，尢其是實用常式。

塊設備驅動程式首先要作一些初始化，RAM盤驅動程式要預留一些記憶體空間、硬碟驅動程式要確定硬碟參數等等。所有磁片驅動程式在相應的硬體初始化時都被調用，但隨後它們將調用包含一公共主迴圈的函數，該迴圈將一直執行下去，不會返回到調用者。主 迴圈執行的工作是：接收一條消息，執行相應的操作，然後發回一條應答消息。

各驅動程式所公用的主迴圈並不是每個驅動程式都擁有一份拷貝，在MINIX二進位碼中只存在一份拷貝。它使用的技術是各驅動程式向主迴圈傳遞一個參數，它是一個指向一個表的指標，該表中包含了完成各具體操作的函數指標，於是以後就可以間接調用這些函數。這種技術也使得各驅動程式能夠共用函數。圖3－17示出了主迴圈的框架。其中如下語句：

code = ( \*enty\_points -> dev\_read) (& mess)；

是間接函式呼叫，儘管各驅動程式執行同一個主迴圈，但它們調用不同的dev\_read函數。不過有些操作，例如CLOSE非常簡單，可供多個設備共用。

message mess; /\* 訊息緩衝區 \*/

void shared\_io\_task(struct driver\_table \* entry\_points) {

/\* 每個任務調用本過程之前先完成初始化 \*/

while (TRUE) {

receive(ANY, &mess);

caller = mess.source;

switch(mess.type) {

case READ: rcode = ( \*entry\_points->dev\_read) (&mess); break;

case WRITE: rcode = ( \*entry\_points->dev\_write)(&mess); break;

/\* 這裡還有其他分支，如OPEN、CLOSE及IOCTL \*/

default: rcode = ERROR;

}

mess.type = TASK\_REPLY;

mess.status = rcode; /\* 執行結果代碼 \*/

send(caller, &mess);

}

}

圖 3－17 一個使用間接調用的共用I/O任務。

這種共用一個主迴圈的方法很好地解釋了前邊提到的進程概念。記憶體中只有一個執行代碼的拷貝，但它同時為多個獨立的進程執行，在一給定時刻，各進程可能位於代碼的不同點上，但它們使用各自的資料和堆疊。

所有設備驅動程式都有6種可能的操作，它們對應於圖3－15所示的消息結構m.m\_type域的各種可能值，它們分別是：

1 OPEN

2 CLOSE

3 READ

4 WRITE

5 IOCTL

6 SCATTERED\_IO

有程式設計經驗的讀者對這些操作可能很熟悉。在設備驅動程式層，大多數操作與同名的系統調用相關。例如READ和WRITE的意義就非常清楚，READ操作將從設備讀取一個資料塊到調用進程的記憶體區域；WRITE則正好相反。調用READ時進程在資料傳輸完成前將一直阻塞；而對於WRITE，作業系統有可能將資料暫存在緩衝區中，隨後再真正將其傳送到設備，這時WRITE系統調用將很快就返回調用進程。這對調用進程很有利，因為它隨後可以再次使用這次暫存了寫出資料的緩衝區。OPEN和CLOSE的含義與檔的OPEN和CLOSE類似：OPEN 操作將驗證設備是否可用，當不可用時則返回一條錯誤消息，CLOSE將確保把先前延遲寫的資料真正寫到設備上。

對IOCTL可能不大熟悉。許多I/O設備都有一些指令引數，經常要對這些參數進行檢查，也可能要修改。IOCTL的任務就是做這些工作。常見的一個例子是改變通信線路的傳輸速率和同位方式。塊設備IOCTL的操作不大常用，在MINIX中，檢查和改變磁片設備的分區是用IOCTL完成的（儘管也可以通過讀寫資料塊完成）。

SCATTERED\_IO操作無疑是最少見的，除了個別非常快的磁片設備（如RAM盤）外，如果每次只請求讀寫一塊，那麼其I/O性能很難提高。一個SCATTERED\_IO請求允許檔案系統讀寫多個塊。對於READ來說，要求多讀的塊可能並不是由執行該調用的進程所申請的，而只是作業系統試圖預測將來對資料的請求。這種請求並不一定由設備驅動程式實現。對每一個塊的請求可以被一個標誌位元修改，它通知驅動程式該請求是可選的。實際上檔案系統可以聲明：“最好將這些資料全部都拿來，但實際上並非馬上就要用”。設備可以據此相機行事。例如軟碟驅動程式將返回整條磁軌上的資料，相當於：“我把這些資料給你，但移動到另一條磁軌上操作太費時了，在你需要時再通知我。”

當寫資料時，則不存在上述問題。但作業系統可能會暫存許多寫請求隨後一次寫出，這比每次處理單個請求的效率要高。在SCATTERED\_IO請求中，申請讀寫塊的請求被排序，這比隨機地處理要高效。而且一次調用驅動程式傳輸多個塊減少了需要發送的消息數。

3.5.2 公用塊設備驅動程式軟體

塊設備驅動程式需要的定義都放在driver.h中。其中最重要的是driver結構（9010～9020行）。其中保存了各驅動程式執行具體I/O操作的函數位址。該檔中還定義了device結構（9031～9034行）。其中保存了與分區相關的最主要資訊：基底位址和長度，它們都以位元組為單位。採用這種格式使得對基於記憶體的設備（RAM盤）無需作任何轉換，由此最大程度地提高了回應速度。而對於真正的磁片，由於有很多因素影響其存取速度，因而轉換到磁區位址並不增添很多麻煩。

所有塊設備驅動程式共用的主迴圈及其他函數都在driver.c中，在硬體執行完必要的初始化後，驅動程式都調用driver\_task，同時向其傳入一個driver結構作為參數。在獲得一個供DMA操作使用的緩衝區位址後進入主迴圈（9156～9199行）。該迴圈將一直執行下去，不返回到調用進程。

檔案系統是唯一向設備驅動任務發送消息的進程。第9165到9175行的switch語句正是對此進行檢查。硬體發出的多餘中斷被忽略，所有錯誤地發過來的消息將在屏慕上列印一條警告資訊。這看起來似乎沒什麼副作用，但顯然發送了錯誤消息的進程有可能一直阻塞以等待應答。在主迴圈中的switch 語句中，前三種消息 DEV\_OPEN、DEV\_CLOSE、DEV\_IOCTL將使用driver結構傳來的位址進行間接調用，而DEV\_READ、 DEV\_WRITE和SCATTERED\_IO消息將直接調用do\_rdwt和do\_vrdwt。但driver資料結構被switch中的所有調用作為一個參數傳遞，不論是直接還是間接，所以所有的被調用常式在需要時都可以進一步使用它。

在按消息中的請求進行處理後，根據設備本身的特性需要作一些清理操作。例如對於軟碟，可能會啟動一個計時器，以便在一段時間內下一個請求未達到的情況下關閉驅動器的電機。間接調用也可被用於此種目的。在清理操作之後，隨之構造一個應答消息並傳送給調用者（9194到9198行）。

在進入主迴圈後，各任務執行的第一個操作是調用init\_buffer（9205行）。它為DMA操作分配一個緩衝區，所有的驅動程式任務假如執行DMA操作的話，都使用這同一個緩衝區，但有些驅動程式不使用DMA。除了第一次初始化之外的其他的初始化都是多餘的，但並沒有副作用。編寫一段測試代碼來確定初始化是否應被跳過將更複雜。

這段初始化操作之所以必要是由於最初的IBM PC機硬體的一個古怪之處所引起的，該PC機硬體要求DMA緩衝區不得越過64K邊界。也就是說，一個1K的DMA緩衝區應從64510開始，而不是64514。因為後者正好越過65536的64K邊界。

這種討厭的規則源於IBM PC使用一種老的DMA控制器晶片－Intel 8237A，8237A含有一個16位的計數器。由於DMA使用絕對位址，而不是相對段寄存器的偏移位址，所以需要一個更大的計數器。在老式的只能定址1M位元組的機器上，DMA位址的低16位被裝入8237A，高4位則被裝入一個4位的鎖存器。新一點的機器使用8位的鎖存器，並能定址16M位元組。當8237A由0xFFFF變到0x0000時，並不向鎖存器進位，導致DMA地址突然減掉了64K。

一個可移植的C程式不能為一資料結構指定絕對的記憶體位址，所以無法防止編譯器將緩衝區放在一個不可用的位置。解決辦法是分配一片大小為所需緩衝區2倍的記憶體buffer（9135行）。並保留一個指標tmp\_buf（9136行）供訪問這片記憶體使用。init\_buffer首先嘗試著將tmp\_buf指向buffer的開頭；然後檢查它在遇到64K邊界之前是否能夠提供足夠的空間。如果不行，則tmp\_buf就遞增真正所需的空間的位元組數。這樣在buffer的某一端總會造成一些位址的浪費，但同時保證了不會由於64K邊界的限制而造成緩衝區失敗。

更新的IBM PC系列電腦使用更好的DMA控制器，所以這部分代碼可以簡化。而且如果所用的機器肯定不存在上述問題時，可以只申請一小部分記憶體便可滿足要求。如果確實這樣做了，那麼現在讓我們考慮一下假定在你判斷有誤的情況下，這個錯誤發生的有關情況。對於1K的DMA緩衝區，在使用老式的DMA控制器時，出錯的概率是64分之一。每次核心代碼被修改，導致編譯後的核心大小發生變化時，這樣的概率同樣存在。當這個錯誤在下一個月或次年發生時，很可能歸咎於最後一次修改的代碼。類似的不可預知的硬體“特徵”使我們可能需要花幾周的時間來查找那些隱藏很深的錯誤（對於那些與本例類似的，在技術手冊中從未提到的錯誤，可能會花更多的時間）。

do\_rdwt是driver.c中的下一個函數。它又可能調用三個與設備相關的函數。driver結構中的dr\_prepare、dr\_schedule和dr\_finish分別指向這三個函數。下面我們將按照C語言的記法，用\*function\_pointer表示function\_piinter指向的函數。

在檢查請求中的位元組計數是正數後，do\_rdwt調用\*dr\_prepare。該操作一定成功，因為只有當OPEN操作指定了一個非法設備時\*dr\_prepare才會失敗。接下來將填充一個標準的iorequest\_s結構（在include/minix/type.h中3194行定義）。然後是一個間接調用\*dr\_schedule。正如我們將要在磁片硬體一節中即將看到的那樣，簡單地按照接收到請求的順序來處理它們是不夠的。\*dr\_schedule的功能就是允許一個特定設備按照最符合其硬體特性的方式來處理請求。這裡的間接調用遮罩了各種設備的差異。對於RAM盤，dr\_schedule指向一個常式，由該常式來執行真正的I/O操作，而下一個間接調用\*dr\_finish什麼也不作。對於真正的磁片，dr\_finish指向一個常式，該常式處理自上次\*dr\_finish以來積累的所有\*dr\_schedule調用所發出的資料傳輸請求。但後面將會看到，在有些情況下調用\*dr\_finish可能並不傳送所請求的所有資料。

在執行真正資料傳輸的常式中，iorequest\_s結構中的io\_nbytes域將被改變。返回值為負表明出錯；返回值為正表明請求的位元組數和成功傳送的位元組數之間的差值。即使沒有發生位元組傳送也並不表示一定是出錯，它只是表明到了設備的結尾。當返回主迴圈時，如果出錯，則錯誤碼放在應答消息的REP\_STATUS域；若成功則將剩餘待傳輸位元組數從初始消息的COUNT域中減去（9249行），結果（真正傳送的位元組數）放在由driver\_task返回的應答消息的REP\_STATUS域中。

下一個函數do\_vrdwt處理所有散佈的I/O請求。請求一個散佈I/O的消息使用ADDRESS域來指向一個iorequest\_s類型的資料結構陣列。其中的每一項包含一次I/O操作所需的所有資訊，包括：緩衝區位址、在設備上的偏移、位元組數、請求是讀還是寫等等。一個請求中的全部操作或者都是讀，或者都是寫，而且將按照它們在設備上的資料塊順序排序。這裡需要作的工作不僅僅是do\_rdwt所作的簡單的讀或者寫，因為該請求陣列必須被拷貝到核心空間。而一旦拷貝完成，將間接地調用\*dr\_prepare、\*dr\_schedule和\*dr\_finish這三個與設備無關的常式。區別在於對\*dr\_schedule的調用是一個迴圈，它每收到一個請求就迴圈一次，或者直到發生錯誤（9288-9290行）。在迴圈結束後將調用\*dr\_finish。然後請求陣列被拷貝回原處。陣列中每一項的io\_nbytes域將被修改以反映傳送了多少位元組。儘管總數並未在driver\_task構造的返回消息中直接返回，但調用常式自己可以從該請求陣列中自行計算出來。

在一個散佈的I/O讀請求中，當最終調用\*dr\_finish時，並不一定所有調用\*dr\_schedule的傳送請求都已實現。在iorequest\_s結構中的io\_request域中含有一個標誌位元，通知設備驅動程式對某一資料塊的請求是否是可選的。

在driver.c中的隨後幾個常式都是為以上操作提供通用的支援。\*dr\_name用於返回一個設備的名字。對於沒有給定名字的設備，no\_name函數從任務表中檢索設備名。有些設備可能不需要一個特定的服務，如RAM盤在回應DEV\_CLOSE請求時並不要求作任何操作，這時就使用do\_nop函數，它僅僅根據請求的類別返回不同的碼值。接下來的函數nop\_finish和nop\_cleanup則分別用作那些不需要\*dr\_finish和\*dr\_cleanup服務的設備的啞常式。

有些磁片設備需要延時，例如等待一個軟盤機的步進電機加速。為此driver.c中包含了提供相應功能的常式clock\_mes。clock\_mess用來向時鐘任務發送消息。該常式的參數包括：需等待的時鐘滴答數，以及等待時間到之後需調用函數的位址。

最後，do\_diocntl（9364行）執行塊設備的DEV\_IOCTL請求。DEV\_IOCTL請求只能是讀（DIOGETP）或寫（DIOSETP）分區資訊，除此之外均出錯。do\_diocntl調用設備的\*dr\_prepare來驗證設備是否合法，並得到一個指向device結構的指標。該結構描述了按位元組計數的分區基址和大小。對於讀請求，它調用設備的\*dr\_geometry函數來獲得該分區的最後柱面號、磁頭及磁區資訊。

3.5.3 驅動程式庫

文件drvlib.h 和drvlib.c包含一些與系統有關的代碼，這些代碼支持IBM PC及兼容機的磁碟分割。

分區機制允許單個的外存設備被分成若干子設備，它主要用於硬碟，但MINIX還提供對軟碟的分區。對磁碟分割的理由是：

1 大容量磁片單位價格便宜。如果有兩個或更多的作業系統在使用不同的檔案系統，則將一個大硬碟分區比為各個作業系統安裝各自的硬碟更經濟。

2 作業系統能夠處理的設備的大小可能有限。

3 一個作業系統可能使用兩個或更多的檔案系統。例如，一個標準的檔案系統用於普通檔，同時一個不同結構的檔案系統用作虛存的交換區。

4 將一個系統的一部分檔放在一個獨立的邏輯裝置上可能方便一些。將MINIX根檔案系統放在一個小的設備上將使其更便於備份，而且有助於在引導時將其拷貝到一個RAM盤中。

磁碟分割的支持是平臺有關的，這種特定性與硬體沒有關係。分區支援獨立於設備。但如果一台設備上有多個作業系統運行，則它們必須對分區表的格式達成一致。在IBM PC上該標準由MS－DOS的fdisk命令確定。其他作業系統，如MINIX, OS/2以及Linux也使用該命令，以便與MS－DOS共存。當MINIX被移植到另一種機器時，應該使用在這種新硬體上運行的作業系統所採用的分區表格式。所以在MINIX中支援IBM電腦分區的源碼部分被放在drvlib.c中，而不是放在driver.c中，目的是使其能更容易地被移植到其他硬體上。

從固件設計人員繼承過來的基本資料結構定義在include/ibm/partition.h中，它通過#include 語句被包含在drvlib.h中。其中包含有每個分區的柱面－磁頭－磁區格式的資訊，以及標識檔案系統類型的代碼和一個分區是否可引導的標誌。對檔案系統進行了檢查之後，其中多數資訊便不再需要。

當第一次打開一個塊設備時，將調用partition函數（drvlib.c，9521行）。其參數包括一個driver結構（這樣它便可以調用與設備相關的函數）、一個初始的次設備號、以及另一個參數，它標識分區類型是軟碟、主要磁碟分割或次分區。partition函式呼叫設備相關的\*dr\_prepare函數來驗證設備合法，並獲得一個device結構的基底位址和大小。然後它調用get-part-table來判斷分區表是否存在。若存在則將其讀入，如果分區表不存在則操作結束。否則用最初調用時使用的分區編號規則計算第一個分區的次設備號。在使用主要磁碟分割的情況下，分區表是經過排序的，所以分區的順序與其他作業系統所使用的相同。

在這裡將第二次調用\*dr\_prepare，這次使用剛計算出來的第一個分區的設備號。如果子設備合法，則對表中所有項執行一個迴圈，每次檢查從設備上的表中讀出的值是否超越先前獲得的整個設備的基址和範圍。如果出現不一致，則記憶體中的表格被修改以取得一致。這似乎不大合適，但由於分區表可能被不同的作業系統修改，使用另一個作業系統的程式師可能很聰明地試圖利用分區表作一些不可預知的事情，或者由於別的原因磁片上的分區表中可能會有垃圾資訊。所以我們更相信使用MINIX計算出來的數值。這裡的思想是安全總比出錯要好。

還是在這個迴圈中，對於設備上所有標識為MINIX的分區，遞迴呼叫partition函數以獲取其子分區資訊。如果一個分區被標識為一個擴展分區，則調用drvlib.c中的下一個函數extpartition。

extpartition（9593行）實際上與MINIX無關。所以這裡不進行詳細討論。MS－DOS使用擴展分區，它實際上是另外一種創建子分區的機制。為了支援讀寫MS－DOS檔的MINIX命令，我們需要知道這些子分區。

get-part-table（9642行）調用do\_rdwt來從駐留分區表的設備（或子設備）上獲取一個磁區。如果它用來獲取一個主要磁碟分割則偏移參數為0；如果是次分區則偏移為非0。它檢查分區表的魔數（0xAA55），並根據是否找到了合法的分區表而分別返回真或假。若找到一個分區表，則將其拷貝到由輸入參數指定的表位址處。

最後，sort函數（9676行）按最低磁區將分區表中的表項排序。被標識為不包含分區的表項則不參與排序，所以被排在最後，哪怕其最低磁區號為0。這種排序只是簡單的冒泡排序，對於一個只有四項的表，沒必要使用特別優化的演算法。

3.6 RAM盤

現在我們回到各個設備驅動程式，並對其中的幾個進行詳細研究。我們將研究的第一個是RAM盤驅動程式，利用它可以訪問記憶體的任何部分，它的主要用途是保留一部分記憶體，並象普通磁片一樣來使用它。RAM盤並不提供永久存儲，但是一旦檔被拷貝到這一區域，就可以極快的速度進行訪問。

MINIX被設計成在僅有一個軟碟的電腦上也可以運行，在一個這樣的系統中，RAM盤還有一個優點，那就是通過把根檔設備放在RAM盤上，就可以隨意地安裝和卸下軟碟，從而支援了可移動介質。因為不能卸下根設備，所以如果把根檔案系統放在軟碟上，就不能將檔存在軟碟上。另外把根檔案系統放到RAM盤可以使系統具有很大的靈活性：軟碟和硬碟的任意組合都能安裝上去。雖然除了在嵌入系統中使用的電腦外，現在大多數的電腦都有硬碟，但是，在MINIX準備好使用硬碟之前的安裝過程中，或者當需要臨時使用MINIX而不進行完整的安裝時，RAM盤是很有用的。

3.6.1 RAM 盤硬體和軟體

RAM盤的思想很簡單。塊設備是具有兩個操作命令的存儲介質：即寫資料塊和讀資料塊，通常這些資料塊存儲於旋轉存放裝置上，例如軟碟和硬碟。RAM盤則簡單得多，它使用預先分配的主存來存儲資料塊。RAM盤具有快速存取的優點（沒有尋道和旋轉延遲），適於存儲需要頻繁存取的程式和資料。

有些系統支援可安裝的檔案系統，有些則不支援（例如MS－DOS，WINDOWS），這裡簡略地指出它們的區別。對於支援可安裝檔案系統的系統，根設備總是位於固定的位置，可移動的檔案系統（即磁片）可以安裝到檔樹上從而構成一個統一的檔案系統。一旦安裝完畢，用戶就不必關心一個檔在哪個設備上。

相反地，對於象MS－DOS這樣的系統，使用者必須指定每個檔的位置，要麼象B:\DIR\FILE一樣顯式指出，要麼使用一定的預設值（當前設備、目前的目錄等等）。在僅有一兩個軟碟機時，管理的負擔還可以承受。但對於擁有幾十個磁片的大型電腦系統而言，要在任何時刻跟蹤所有的設備，其負擔是無法忍受的。請注意：UNIX系統可以運行在從IBM－PC、工作站和超級電腦，一直到Cray－2這樣的系統；而MS－DOS僅在小系統上運行。

圖3－18 給出了實現RAM盤的思想。根據為RAM盤分配記憶體的大小，RAM盤被分成n塊，每塊的大小和實際磁片塊的大小相同。當驅動程式接收到一條讀寫一個資料塊的消息時，它只計算被請求的塊在RAM盤存儲區的位置，並讀出或寫入該塊，而不讀寫軟碟或硬碟。資料的傳輸通過調用一個組合語言過程來實現，該過程以硬體所能實現的最高速度把資料拷貝到使用者程式或從使用者程式拷出。

圖3－18 一個RAM盤。

一個RAM盤驅動程式可以支援將記憶體中的若干區域當作RAM盤來使用，每個RAM盤用次設備號來區分。一般情況下這些存儲區相互分開，但正如下一節我們將看到的，在某些情況下，使它們相互重疊可能會更方便。

3.6.2 MINIX中的RAM盤驅動程式概述

RAM盤驅動程式實際上是將四個緊密相關的RAM盤驅動程式放在一起，傳給驅動程式的每條消息都要指定下列次設備之一：

0：/dev/ram 1：/dev/mem 2：/dev/kmem 3：/dev/null

其中第一個檔/dev/ram是真正的RAM盤，驅動程式內部既不指定它的大小，也不指定它的起始位址，而是在MINIX啟動時由檔案系統確定。缺省情況下將創建一個與根檔案系統設備大小相同的RAM盤，以便將根檔案系統拷貝到RAM盤。可以通過啟動參數來指定一個容量大於根檔案系統的RAM盤；如果不把根檔案系統拷貝到RAM盤，則在為系統操作留下足夠的記憶體的前提下，RAM盤的容量可以取記憶體可容納的任意值。一旦RAM盤大小確定，則系統就尋找一塊足夠大的記憶體並將其從記憶體池中移出，該操作甚至發生在記憶體管理器開始工作之前。這種策略使得不必重新編譯作業系統即可增加或減少RAM盤的容量。

隨後的兩個次設備分別用於讀寫實體記憶體和核心記憶體。當打開/dev/mem並進行讀操作時，讀出的是起始於絕對位址零的內容（真實模式下的中斷向量）。普通使用者程式永遠不會執行這個操作，但與系統調試有關的程式可能會使用這種功能。打開並對/dev/mem執行寫操作將修改中斷向量。顯然，只有那些對該操作的結果十分清楚的的熟練用戶才可非常謹慎地執行這個操作。

設備檔/dev/kmem和/dev/mem相似，只是其第0位元組是核心資料存儲區的第0位元組，而其絕對位址隨MINIX核心代碼的大小而變。同樣地，它也是主要用於調試以及非常特殊的程式。注意這兩個次設備覆蓋的存儲區是重疊的，如果你精確地知道核心在記憶體中如何放置，則你可以打開/dev/mem檔，將檔指標定位到內核資料區的起點，你會發現這裡的資料和從檔/dev/kmem開頭讀出的資料相同。但如果你重新編譯核心並改變其大小，或者如果在MINIX的後續版本中內核被放在記憶體的其他位置，那麼你將不得不把指標定位到不同的位置才能發現和從檔/dev/kmem開頭讀出的一樣的資料。對這兩個檔必須採取保護措施以保證超級用戶之外的其他用戶不能使用它們。

最後一個檔 /dev/null 是一個接收資料並把資料拋棄掉的設備檔。在執行shell命令時，如果程式產生的結果不再需要，則可以使用它，例如：

a.out >> /dev/null

將執行a.out但丟棄輸出結果。RAM盤驅動程式對它採用一種高效的處理方式：將其長度置為零，這樣便沒有資料對它拷入或拷出。

處理/dev/ram、/dev/mem 和/dev/kmem的代碼是相同的，區別僅在於它們各自對應不同的存儲區，該存儲區由陣列ram\_origin和ram\_limit指示，兩陣列均由次設備號進行索引。

3.6.3 MINIX中的RAM盤驅動程式實現

和其他的磁片驅動程式一樣，RAM盤驅動程式的主迴圈在檔driver.c中，和設備相關的支援記憶體設備的代碼在memory.c中。陣列 m\_geom（9721行）存放了四個記憶體設備的基底位址和大小。9733至9743行的driver 結構m\_dtab定義了將在主迴圈中執行的記憶體設備調用。該表中的四個入口在driver.c中都屬於幾乎沒執行什麼操作的常式，這也證實了RAM盤操作並不複雜。主過程mem\_task（9749行）調用一個函數來進行一些初始化，然後調用主迴圈。主迴圈完成取消息、將其分派到相應的過程、然後發出應答。執行完畢後並不返回到mem\_task。

執行讀或寫操作時主迴圈執行三個調用：一個準備設備，一個調度 I/O操作，一個結束操作。對於記憶體設備，首先調用m\_prepare，它檢查請求的次設備是否合法，然後返回一個結構的位址，該結構中存放著所請求RAM區的基底位址和長度。第二個調用的是 m\_schedule（9774行），它執行所有的工作。對於記憶體設備，這個函數名起的並不恰當。根據定義，在RAM中任何位置都可以和其他位置一樣訪問，因此沒有必要做任何調度，只有對於有移動臂的磁片才需要。

RAM盤的操作非常簡單和快捷，所以根本不會出現需要延遲一個請求的情況。它所做的第一件事就是清除由離散I/O調用設置的一個比特，該比特用來指示一個操作的完成為可選。在消息中傳進來的目標位址指向調用者存儲區的一個位置，9792行到9794行的代碼將這個位址轉換成記憶體中的絕對位址，並檢查它是否合法。9818行到9820行的代碼執行實際的資料傳輸，它直接把資料從一處拷貝到另一處。

一個記憶體設備不需要第三步來結束一個讀寫操作，在m\_dtab中相應的內容為調用nop\_finish。

m\_do\_open（9829行）打開一個記憶體設備，其主要任務是調用m\_prepare來檢查引用的設備是否合法。當引用/dev/mem 和/dev/kmem時，將調用enable\_iop（在文件protect.c中）來改變當前CPU的特權級。當訪問記憶體時，這是不需要的。這是處理另外一個問題的技巧。請注意奔騰型的CPU實現了四個特權級，使用者程式運行在最低特權級。Intel處理器還具有一個在許多其他系統中不存在的體系結構特性：一套獨立的定址I/O埠的指令集。在這類處理器中，I/O埠和記憶體是分開處理的。通常，使用者進程試圖執行一條訪問I/O埠的指令將引起一個通用保護異常。然而對於MINIX，存在一些理由應該允許使用者編寫訪問埠的程式，特別是在小系統中。於是使用enable\_iop改變CPU的I/O保護級別（IOPL）位從而允許訪問埠，其結果是賦予允許打開/dev/mem 或/dev/kmem的進程訪問I/O埠的特權。在把I/O設備作為記憶體來訪問的系統結構中，這些設備的rwx位元自動地覆蓋了對I/O的訪問。如果隱藏了這個特性，則可能被認為是安全方面的缺陷，不過現在你已經知道了這一點。如果你計畫使用MINIX來控制銀行的安全系統，你可能希望重新編譯核心以去掉這一函數。

下一個函數m\_init（9849行）僅當第一次調用mem\_task時被調用一次，它設置/dev/kmem的基底位址和長度，並根據MINIX是運行在8088、80286還是80386模式下，將/dev/mem的長度分別設置為1MB、16MB、或4GB－1。這些長度值是MINIX支持的最大長度，和機器上安裝的RAM多少無關。

RAM盤支援函數m\_ioctl（9874行）中的幾個IOCTL操作。MIOCRAMSIZE是檔案系統設置RAM盤大小的簡便方法。MIOCSPSINFO被檔案系統和記憶體管理器用來將進程表中它們對應部分的位址設置在psinfo表中。實用程式ps使用MIOCSPSINFO來取出這些資料，ps是一個標準的MINIX實用程式，MINIX的微內核結構使它變得很複雜，這是因為這種結構將ps所需要的進程表資訊放在幾個不同的地方。IOCTL系統功能調用是處理這個問題的一種簡便方法。否則每次編譯MINIX的新版本時都必須編譯一個新版本的ps。

memory.c中的最後一個函數是m\_geometry（9934行），記憶體設備沒有機械驅動器的柱面、磁軌和每條磁軌上的磁區等幾何結構，但出於RAM盤一旦被詢問到這些參數的考慮，所以也要模擬提供這些參數。

3.7磁片

利用RAM盤來介紹磁片驅動程式很合適（因為它很簡單），不過實際磁片有一些我們尚未接觸到的特性。下面幾節中我們將首先簡單討論一下磁片硬體，然後籠統地分析一下磁片驅動程式，並重點分析MINIX的硬碟驅動程式。我們不詳細討論軟碟驅動程式，但是將述及軟碟驅動程式和硬碟驅動程式工作方式的不同之處。

3.7.1磁片硬體

所有實際的磁片都組織成許多柱面，一個柱面上的磁軌數和垂直放置的磁頭個數相同。磁軌又被分成許多磁區，每條磁軌上磁區數目典型的範圍是：對軟碟每條磁軌8至32磁區，在某些硬碟上則可多達幾百磁區。最簡單的設計是每條磁軌具有相同的磁區數，每個磁區包含相同數目的位元組。然而，略加思索就可以知道物理上靠近磁片外邊沿的磁區比靠近磁片內邊沿的磁區要長一些。不過讀寫每個磁區的時間是一樣的。很明顯在最裡面的柱面上的資料密度要高一些，一些磁片的設計要求讀寫內部磁軌是改變磁頭的驅動電流，這由磁碟控制卡硬體處理，對使用者（或作業系統的實現者）不可見。

內圈磁軌和外圈磁軌資料密度的不同意味著會犧牲一些磁片容量，也意味著存在更複雜的系統。有人嘗試過設計一種當磁頭處於外部磁軌時磁片旋轉速度更快的軟碟，這樣外圈磁軌就可以具有更多的磁區，從而增加磁片的容量。目前安裝MINIX的系統還不支援這種磁片。然而，現代大容量硬碟中外圈磁軌具有的磁區數比內圈磁軌更多，這就是IDE（Integrated Drive Electrinics）驅動器，它內置的電子器件進行的複雜的處理遮罩了具體細節，對於作業系統來說，它們仍然呈現出簡單的幾何結構，每條磁軌具有相同的磁區。

驅動器與控制器電子器件和機械硬體同等重要。插到電腦主機板上的控制卡的主要元件是一塊專用的積體電路，實際上是一個小的微型電腦。硬碟驅動卡的控制線路可能比軟碟的還要簡單，這是因為硬碟驅動器本身具有一個功能強大的電子器件控制器。對於磁片驅動程式有重要意義的一個設備特性是：控制器可以同時控制兩個或多個驅動器進行尋道，這稱為重疊尋道（overlapped seeks）。當控制器和軟體等待一個驅動器完成尋道時，控制器可以啟動另一個驅動器進行尋道。許多控制器可以在對一個或多個其他驅動器尋道的同時在一個驅動器上進行讀寫操作，但是軟碟控制卡不能同時讀寫兩個驅動器（讀寫資料要求控制器在毫秒級的時間內傳輸位元流，所以一個傳輸就基本佔用了所有的計算能力）。對於具有集成控制器的硬碟則情況不同，在一個有多個硬碟的系統中，這些驅動器可以同時操作，至少在磁片和控制器緩衝區之間的資料傳輸可以並行進行。然而，在控制器和系統記憶體之間只能同時執行一個傳輸，同時執行兩個或多個操作的能力極大地降低了平均讀寫時間。

圖3－19對用作最初IBM－PC標準存儲介質的雙面雙密度軟碟的參數和一個可能在奔騰電腦上發現的典型中等容量硬碟驅動器的參數。MINIX使用1K的資料塊，因此，對於這兩種磁片，軟體使用的資料塊由兩個連續的磁區組成，它們總是作為一個單元進行讀寫。

圖3－19 初始IBM－PC 360K軟碟和Western Digital WD AC2540 540－MB硬碟的磁片參數。

在閱讀現代硬碟規格時需要注意的一點是驅動程式軟體所指定和使用的幾何參數可能與物理格式不同。例如，圖3－19所描述的硬碟由推薦設置參數指定為1048柱面、16磁頭、每條磁軌63磁區。磁片上的電子器件將作業系統提供的邏輯磁頭和磁區參數轉換成磁片使用的物理參數。這也是為了相容舊系統（對老的固件）而採取妥協的另一個例子。最初的IBM－PC的設計者僅為BIOS ROM 中的磁區數域分配了六個比特，這樣每條磁軌的磁區數超過63的磁片必須工作在一套類比的邏輯磁片參數上。在這種情況下，廠商提供的規格指出實際上有四個磁頭，所以如圖中所示，實際上每條磁軌有252個磁區。因為這類磁片最外面的磁軌比最裡面的磁軌有更多的磁區，所以這裡實際上已經是簡化了。圖中所描述的磁片確實有四個物理磁頭，但是柱面數比3000略多一些。柱面被分成了十二個組，最內部的磁軌有57個磁區，最外部的磁軌有105個磁區，這些參數在磁片的規範中是沒有的。驅動器的電子器件完成了這個轉換，用戶便無需瞭解這些細節。

3.7.2 磁片軟體

本節將籠統地流覽一下和磁片驅動程式有關的內容。首先考慮讀寫一個磁片塊需要多長時間。需要的時間由下面三個因素決定：

1 尋道時間（將磁頭臂移動到相應的柱面所需的時間）。

2 旋轉延遲（相應的磁區旋轉到磁頭下面所需的時間）。

3 實際的資料傳輸時間。

對於大多數磁片，和另外兩個時間參數相比，尋道時間要大得多，因此減小平均尋道時間將極大地提高系統性能。

磁片設備可能會出錯。所以常將某種錯誤校驗 － 校驗和或者迴圈冗餘校驗的資訊和各磁區的資料同時記錄在盤上，甚至磁片格式化時記錄的磁區位址也有校驗資料。當檢測到出錯時，軟碟控制卡可以報告出錯資訊，但是必須由軟體決定出錯後該怎麼辦。硬碟控制器則通常承擔大部分出錯處理工作。

尤其對於硬碟，一條磁軌上連續磁區的傳輸是非常快的，因此讀取比請求資料更多的資料並把它緩衝在記憶體中對加速磁片存取非常有效。

磁片臂調度演算法

如果磁片驅動程式每次接收一個請求並按照接收循序執行，即先來先服務（FIFS），則很難優化尋道時間。然而當磁片負載很重時，可以採用其他策略。很有可能當磁片臂為一個請求尋道時，其他進程也產生了其他磁片請求。許多磁片驅動程式都維護一張表格，按柱面號索引，每一柱面的全部請求用一個連結錶鏈在一起，鏈表頭指標存放在表格相應的表目中。

有了這種資料結構，我們就可以改進先來先服務的調度演算法。為了說明如何實現，考慮一個具有四十個柱面的磁片。假設一個請求到達請求讀柱面11上的一個資料塊，當對柱面11尋道時，又順序到達了新請求要求尋道1 、36 、16 、34 、9 和12，則它們被安排進入請求等待表，每一個柱面對應一個單獨的鏈表。圖3－20顯示了這些請求。

圖3－20 最短尋道（SSF）磁片調度演算法。

當前請求（請求柱面11）結束後，磁片驅動程式需要選擇下一個請求。若使用FCFS，則首先選擇柱面1，然後是 36 ，依次類推。這個演算法中磁片臂分別需要移動 10 、35 、20 、18 、25和3個柱面，總共需要移動111個柱面。

為了減少尋道時間，也可以總是選擇和磁片臂最接近的柱面請求，對於圖3－20所示的請求，被選擇請求的順序如圖3－20下方折線所示，依次為12 、9 、16 、1 、34和36。按照這個順序，磁片臂分別需要移動1 、3 、7 、15 、33和2個柱面，總共需要移動61個柱面。這個演算法 － 最短尋道演算法Shortest Seek First（SSF）和FCFS演算法相比，把柱面移動數減小了一半。

不幸的是，SSF演算法存在一個問題。假設當圖3－20所示的請求正在被處理，有其他請求不斷到達。例如，磁片臂移到16柱面以後，到達一個對柱面8的請求，那麼它的優先順序將比柱面1要高。如果接著又到達了一個對柱面13的請求，磁片臂將移到柱面13而不是柱面1。對於一個負載很重的磁片，磁片臂趨向於大部分時間停留在柱面中部區域，對兩端極端區域請求的處理不得不等待，直到由於負載的統計波動使得中部區域沒有請求為止。遠離柱面中部區域的請求得到的服務很差。這裡獲得最小回應時間的目標和公平性之間存在衝突。

高層建築也得進行這種折衷處理，高層建築中的電梯調度問題和調度磁片臂很相似。電梯請求不停地到來，隨機地呼喚電梯到各個樓層，控制電梯的微處理器能夠很容易地跟蹤使用者按下按鈕的順序，並使用FCFS演算法為他們提供服務，也可以使用SSF演算法。

然而，大多數的電梯使用另一種演算法來調和效率和公平性的衝突。電梯保持按一個方向運動，直到在那個方向上沒有更遠的請求為止，然後改變方向。這個演算法在磁片領域和電梯領域都被稱為電梯演算法（elevator algorithm），它需要軟體維護一個二進位位元：即當前的方向：向上或是向下。當一個請求結束之後，磁片或電梯的驅動程式檢查該位元，如果是向上，磁片臂或電梯艙移至下一個更高的等待請求。如果更高的位置沒有請求，就翻轉方向位。如果方向位設置為向下，同時存在一個低位置的請求，則移向該位置。

圖3－21顯示了與圖3－20相同的七個請求使用電梯演算法的情況。假設初始方向位為向上，則各柱面獲得服務的順序是 12 、16 、34 、36 、9和1，磁片臂分別移動1 、4 、18 、2 、27和8個柱面，總共移動60個柱面。在該例中，電梯演算法比SSF演算法稍微好一點，儘管通常它不如SSF。電梯演算法的一個優良特性是對任意的一組請求，移動磁片臂次數的上界是固定的：正好是柱面數的兩倍。

對這個演算法進行略微改進可以進一步減小回應時間，方法是總是按一個方向移動磁片臂，處理完編號最高柱面上的請求後，磁片臂移動到具有讀寫請求的編號最低的柱面，然後繼續向上移動。這實際上等於將最低柱面看作是最高柱面之上的相鄰柱面。

一些磁碟控制卡提供了供軟體檢查當前磁頭下方磁區號的方法。對於這種磁碟控制卡，還可進行另一種優化。如果有兩個或多個讀寫同一柱面的請求正在等待處理，驅動程式可以請求讀寫下一個要通過磁頭的磁區。注意當一個柱面有多條磁軌時，因為選擇磁頭既不需要移動磁片臂也沒有旋轉延遲，控制器可以快速選擇任意磁頭，所以連續的讀寫請求可以申請不同的磁軌。

圖3－21 調度磁片請求的電梯演算法。

對於現代的硬碟，其資料傳輸率比軟碟快得多，所以必須採用某種自動高速緩衝機制。典型地，任何讀一個磁區的請求將使得這個磁區和同一磁軌上的其他磁區都被讀進來，讀進多少磁區取決於控制器中有多少可用的高速緩衝空間。圖3－19中描述的540M磁片具有64K或 128K的高速緩衝空間。磁碟控制卡動態地決定高速緩衝的使用。在最簡單的模式中，緩衝空間被分成兩部分：一部分用於讀，另一部分用於寫。

當有多個驅動器時，每個驅動器都有一個單獨的請求等待表。一旦任何一個驅動器空閒，則啟動一個尋道請求將磁片臂移到下一個請求的柱面（假設控制器允許重疊尋道）。當前資料傳輸結束後，將檢查是否有一驅動器位於正確的柱面上，如果存在一個或多個這樣的驅動器，則啟動下一個資料傳輸。如果沒有，則驅動程式對剛結束資料傳輸操作的驅動器發出新的尋道命令並等待，直到下一次中斷到來時檢查哪一個磁片臂首先到達了目標位置。

出錯處理

RAM盤不需要考慮尋道和旋轉優化：在任意時刻，不做任何物理移動就可以讀寫所有的資料塊。RAM盤比實際磁片簡單的另一個地方是出錯處理。RAM盤永遠正常工作，而實際磁片卻不能，它們可能會出現各種各樣的錯誤。常見的錯誤有：

1 程式性錯誤（例如請求讀寫不存在的磁區）。

2 暫時性校驗和錯（例如由磁頭上的灰塵引起）。

3 永久性校驗和錯（例如磁片塊的物理損壞）。

4 尋道出錯（例如磁片臂應定位在第6柱面，但卻到了第7柱面）。

5 控制器錯（例如控制器拒絕接受命令）。

磁片驅動程式應盡可能地處理這些錯誤。

當驅動程式通知控制器對不存在的磁區進行尋道、使用不存在的磁頭、或者對不存在的記憶體傳送資料時，將發生程式性錯誤。多數控制器檢查發送給它們的參數，並在參數非法時給出資訊。理論上這些錯誤不會發生，但是當控制器指出發生了這些錯誤時，驅動程式如何處理呢 ？對於家用系統，最好的辦法是停止運行，列印一條類似“求助於程式師”的消息，以便對錯誤進行跟蹤並加以修正。對於一個在世界上數以千計的地方使用的商用軟體產品，這種方法並不合適，也許唯一能做的是將當前磁片請求以出錯結束，並且希望這種錯誤不會發生得太頻繁。

暫時性校驗和錯因空氣中的灰塵進入盤面和磁頭之間引起。多數情況下，可以通過重複操作幾次來消除這種錯誤。如果它一直存在，則只能將相應的資料塊標誌為壞塊（bad block），並且不再使用。

避免壞塊的一種方法是寫一個非常特殊的程式，它將壞塊清單作為輸入，並小心地創建一個檔，使之包含所有的壞塊。一旦這個檔創建完畢，磁片分配程式就認為這些資料塊已被佔用並永遠不會把它們分配出去。只要不去讀壞塊文件，就不會發生任何問題。

永遠不讀壞塊檔說起來容易做起來難。許多磁片採用一次將一條磁軌的內容拷貝到後備磁帶或磁片上的方法進行備份。如果使用這種方式，則壞塊就會引起麻煩。如果後備程式知道壞塊檔的檔案名並且不拷貝它，那麼按一次拷貝一個檔的方式雖然要慢一些，但可以解決這個問題。

使用壞塊檔不能解決的另一個問題是：如何處理必須位於磁片中固定位置的檔案系統資料結構中存在的壞塊。幾乎每一檔案系統至少都有一個資料結構的位置必須是固定的，以便很容易地找到它。在一個可分區的檔案系統中，通過重新分區可以避開壞道，但軟碟或硬碟開始幾個磁區中的永久性錯誤一般意味著整個磁片無法使用。

“智慧”控制器保留了幾條使用者程式一般不能使用的磁軌。當格式化磁片的時候，控制器可以確定哪些是壞塊，並用保留的磁軌代替損壞的磁軌。存放壞道和保留磁軌之間對應關係的表格保存在控制器的內部記憶體和磁片中。這種替換對驅動程式透明，只是對於精心設計的電梯演算法，如果每次請求讀寫柱面3時，控制器實際使用的是柱面800，那麼其性能會大大降低。磁片記錄表面的製造技術已經比過去有所提高，但還不是盡善盡美，不過對用戶隱藏缺陷的技術也得到了提高。對於圖3－19中所示的硬碟，控制器也管理在使用過程中發生的新錯誤，並在錯誤不能恢復時永久地分配一個替換塊。對於這樣的磁片，驅動程式軟體很少會感覺到壞塊的存在。

尋道錯由磁片臂的機械問題引起。控制器在內部跟蹤磁片臂的位置。為了尋道，控制器向磁片臂電機發出一系列脈衝，每個脈衝移動一個柱面，從而將其移動到新的柱面。當磁片臂到達目標柱面後，控制器讀取實際的柱面號（在驅動器格式化時被寫上去），如果磁片臂處於不正確的位置，就發生尋道錯。

大多數的硬碟自動糾正尋道錯，但許多軟碟控制卡（包括IBM － PC）僅設置一個出錯位，而把其他工作留給驅動程式。驅動程式處理這個錯誤的方法是發出一條RECALIBRATE命令，把磁片臂移到最外面，並將使控制器內部將當前柱面號復位為0，如果不能解決問題，驅動器就必須進行修理了。

正如我們看到的：控制器實際上是一個專用的小電腦，它有完備的軟體、變數、緩衝區，偶而還出現故障。有時一個非常的事件序列，例如一個驅動器上的中斷和另一個驅動器上的RECALIBRATE命令同時發生可能引發一個故障，致使控制器陷入一個迴圈或無法接續正在做的工作。控制器設計者通常考慮到最壞的情況，在晶片上提供了一個引腳，當觸發該引腳時，強迫控制器忘記當前的工作並自行復位。如果其他努力都無法奏效，磁片驅動程式可以設置一個比特來觸發該信號對控制器重定。如果還不成功，那麼驅動程式就只能列印出錯資訊並放棄。

每次一道（Track at a time）緩衝

對一個新柱面尋道所花的時間通常比旋轉延遲大得多，而且往往比傳輸時間長很多，也就是說，一旦驅動程式需要把磁片臂移到某個位置，那麼讀一個磁區還是讀一條磁軌都差不多。特別是對於控制器提供旋轉檢測（rotational sensing）的情況，這種效應就更加明顯。因為這時驅動程式能夠知道哪個磁區正位於磁頭下面，並啟動一個對下一磁區的請求，從而使得在一次旋轉中讀一條磁軌成為可能（通常平均花旋轉半周的時間加上一個磁區的時間僅僅讀一個磁區）。

有些磁片驅動程式通過維護一個秘密的每次一道的高速緩衝來充分利用這一特性，並且不被獨立於設備的軟體所感知。如果需要的磁區位於高速緩衝裡，則不需要磁片資料傳輸。每次一道緩衝的一個缺點（除了軟體複雜性和需要緩衝空間）是：從高速緩衝到調用程式之間的資料傳輸必須由CPU使用一個迴圈完成，而不是讓DMA硬體來做這個工作。

一些控制器將這個過程更進一步，在它們自己內部的記憶體中實現每次一道緩衝，而對驅動程式透明，這樣在控制器和記憶體之間的資料傳輸就能使用DMA。如果控制器按這種方式工作，就沒必要讓磁片驅動程式也做這項工作。注意在控制器和驅動程式中由一個命令讀寫整條磁軌是合適的，但不能放在設備無關軟體中，因為它將磁片看成資料塊的線性序列，而不考慮它如何劃分成磁軌和柱面。

3.7.3 MINIX中的硬碟驅動程式概述

硬碟驅動程式是我們已經看到MINIX不得不處理各種類型硬體的第一部分。在開始討論驅動程式細節以前，我們簡單考慮一下一些硬體的不同可能引起的問題。“IBM－ PC”的的確確是一個不同電腦的家族。在不同的家族成員中不僅使用的處理器不同，而且在基本硬體上也有一些很大的差別。家族中最早的成員 － 最初的PC 和PC－XT 使用8位元匯流排，與8088處理器8位元的外部介面相配合。下一代 PC－AT 使用16位元匯流排，該匯流排設計得很巧妙，先前的8位外部設備還可使用，但新的16位的外部設備一般不能用在老的PC－XT系統中。AT匯流排原先是為使用80286處理器的系統設計，並且許多基於 80386、80486 和奔騰的系統也使用AT匯流排。但因為這些新的處理器具有32位元介面，所以，現在有幾種不同的32位元匯流排系統，例如 Intel的PCI匯流排。

對每一類匯流排，都有一個家族的I/O適配器（I/O adapters）供插到系統主機板上。為某一類匯流排設計的所有外設都必須與標準相容，但沒有必要和以前的匯流排相容。和其他的電腦系統家族一樣，在IBM－PC家族中，與匯流排設計同時配套的還有基本I/O系統（BIOS） ROM中的固件，其目的是在作業系統和硬體特殊性之間建立一個聯結二者的橋樑，有些外設在自己的外設板上提供擴充BIOS的ROM晶片。作業系統的實現者面臨的困難是：在IBM類型的電腦（當然指早期的）中的BIOS是為MS－DOS作業系統設計的，不支援多道程序設計，運行於16位真實模式，這是80X86 CPU家族可以使用的各種操作模式中最普通的一種模式。

於是，為IBM－ PC設計新作業系統的實現者面臨幾個選擇：首先，是使用BIOS中的設備驅動程式支援還是重頭編寫新的驅動程式。由於BIOS在許多方面不適於MINIX的需要，所以在MINIX的初始設計中作出選擇並不困難。當然，為了啟動，MINIX的引導監控程序仍要使用BIOS實現系統的初始裝載 － 無論是從硬碟還是從軟碟，實際上這是無法選擇的。一旦系統被裝入，包括我們的設備驅動程式，我們就可以比BIOS做得更好。

面臨的第二項選擇是：沒有BIOS的支援，我們如何使我們的設備驅動程式適用於不同系統中的各種硬體。為了使討論更具體，考慮存在至少四個基本類型的硬碟控制器，這些類型我們可以在一個適用於MINIX的系統上找到：最初的8位XT類型控制器、16位AT類型控制器，以及IBM PS/2系列電腦中兩種不同類型的控制器。有幾種不同的方法解決這個問題。

1 為需要支援的每種硬碟控制器重新編譯一個作業系統版本。

2 在核心中編譯幾個不同的硬碟驅動程式，由核心在啟動時自動決定使用哪一個。

3 在核心中編譯幾個不同的硬碟驅動程式，提供一種方法使用戶決定使用哪一個。

正如我們將看到的，這些方法並不相互排斥。

第一種方法從長遠來看是最好的。對於使用特定配置的系統，沒有必要為從來不使用的驅動程式浪費磁片和記憶體空間，然而，對軟體發行者而言這卻無法接受。準備四套不同的開機磁片，並告訴使用者如何使用這些碟片非常昂貴也十分困難。因此，其他方案是值得考慮的，至少對於初始安裝而言是這樣。

第二種方法是由作業系統檢測外部設備，通過讀寫每塊卡上的ROM或者寫和讀I/O埠來識別每一塊卡。這在一些系統中可行，但是在IBM類型的系統中卻工作得不是很好，因為存在太多的非標準I/O設備。有些情況下，讀寫I/O埠來對設備進行識別可能啟動其他的設備，使其獲得控制權而導致系統無法工作，這種方法也使每個設備的啟動代碼複雜化，並且還是不能工作得很好。使用這種方法的作業系統一般必須提供某種重載機制，典型的如我們在MINIX中使用的機制。

第三種辦法是MINIX使用的方法，它允許編譯多個驅動程式，其中一個是缺省驅動程式。MINIX引導監控程序允許在啟動時讀各種引導參數（boot parameters），它們可以手工輸入，也可以永久存放於磁片上。在啟動時如果發現引導參數格式為：

hd=xt

則強制使用XT硬碟驅動程式，如果沒有發現hd引導參數，使用缺省驅動程式。

為了減少支援多個硬碟驅動程式所導致的問題，MINIX還做了其他兩件事。第一是提供了一個支援在MINIX和ROM BIOS硬碟之間介面的驅動程式，這個驅動程式幾乎可以保證在所有的系統下都可以工作，通過使用啟動參數

hd=bios

來選擇，不過一般這是可求助的最後一種辦法。在80286或更高級的處理器上，MINIX運行於保護模式，但是BIOS永遠運行於真實模式（8086）。無論何時當BIOS中的常式被調用時，切換出保護模式，重新回到先前模式是很慢的。

MINIX處理驅動程式的另一策略是盡可能推遲初始化工作，這樣如果在某些硬體設定中沒有硬碟驅動程式可用，則仍然可以從軟碟啟動，完成一些有用的工作。只要不訪問硬碟，MINIX不會有任何問題。在用戶友好方面這算不上一個大的突破，但是考慮這種情況：由於我們不恰當地配置了一些從不使用的設備，如果系統引導時所有的驅動程式立即初始化，系統將全部處於停滯狀態。把設備初始化推遲到需要的時刻，則當用戶試圖去解決問題時，系統可以使用任何可以工作的部件繼續運行。

另一方面，我們在以困難的方式學習這門課：早期的MINIX版本一啟動就試圖初始化硬碟，如果沒有硬碟，系統就掛起。這是很不幸的。但是，雖然存儲能力要受限制，性能也要降低，MINIX完全可以在沒有硬碟的系統中順暢地運行。

在本節和下一節的討論中，我們選擇AT類型的硬碟驅動程式為例，在標準的MINIX發佈中這是缺省設備驅動程式。這是一個多功能的設備驅動程式，可以處理從早期80286系統中使用的硬碟控制器到能夠處理上吉（G）位元組存儲量的EIDE（Extended Integrated Drive Electronics － 擴展的集成驅動電子線路）控制器。我們在本節所討論的硬碟操作的一般內容也適用於系統所支援的其他硬碟驅動程式。

硬碟任務的主迴圈是我們已經討論過的相同的共用代碼，可以執行六種標準請求。因為在硬碟上進行了分區和子分區，所以DEV\_OPEN請求需要執行大量的工作。當設備打開時（第一次被訪問時），這些必須首先讀進來。一些硬碟控制器也支援CD－ROM驅動器，它們具有可移動介質，並且當執行DEV\_OPEN時，必須檢查介質是否存在。對於CD－ROM，DEV\_CLOSE操作還有以下含義：它需要打開光碟機彈出光碟。可移動介質的其他複雜性在軟盤機中更普遍，所以將在下一節討論。對於硬碟，DEV\_IOCTL操作設置當執行DEV\_CLOSE時介質應被彈出的標誌。這個特性對CD－ROM也是有用的。正如我們曾經提到過的，它也可以用來讀寫分區表。

象先前看到的那樣，每一個DEV\_READ 請求、DEV\_WRITE請求和SCATTERED\_IO請求都分成三個階段來處理：準備、調度和結束。和記憶體設備不同，在硬碟的調度階段和完成階段之間有明確的界限。硬碟驅動程式不使用SSF演算法或電梯演算法。但進行了一種更為有效的調度：把對連續磁區的請求集中起來。請求一般來源於MINIX檔案系統，並且請求許多塊，每塊1024位元組，但驅動程式也能夠處理512位元組的磁區的任意倍的請求。如果請求的磁區緊挨著前一次請求的磁區，則請求就被加入到一個請求鏈表中。這個表被組織成一個陣列，當它滿的時候或者請求的是一個不連續的磁區的時候，就調用結束常式。

在一個簡單的DEV\_READ和DEV\_WRITE請求中，可能會請求不止一塊，但是每一個調度常式的調用緊接著一個結束常式的調用，以保證滿足當前的請求列表。對於SCATTERED\_IO請求，在調用結束常式以前，可能會多次調用調度常式。 如果是對連續的資料塊的請求，請求鏈會擴展到陣列滿為止。回想在SCATTERED\_IO請求中，有一個標誌標識對一個特殊塊的請求是可選的。硬碟驅動程式象記憶體驅動程式一樣忽略OPTIONAL標誌，傳輸所有請求的資料。

硬碟設備驅動程式執行的基本調度和當請求連續的資料塊時推遲資料傳輸應被看成是三步調度過程中的第二步。檔案系統本身通過使用SCATTERED\_IO可以實現類似Teory版的電梯演算法。回想在SCATTERED\_IO請求中，請求鏈表按塊號排序。第三步調度發生在類似圖3－19中描述的現代硬碟控制器中。這些控制器具有“智慧”，能夠緩衝大量的資料，使用內部的程式演算法來按最有效的順序獲取資料，而不是按照請求的到達次序。

3.7.4 MINIX中的硬碟驅動程式實現

微機上使用的小型硬碟有時稱為“溫徹斯特”硬碟。關於這個名字的來源有幾個不同的故事。很明顯它是IBM一個開發磁片技術工程的代號，在這項工程中，讀/寫磁頭飛行於薄薄的空氣墊上，當磁片停止旋轉時落在記錄介質上。這個名字的一種解釋是一個早期的型號有兩個資料模組：30M位元組固定硬碟和30M位元組可移動硬碟。可以設想它使開發人員想起了溫徹斯特30 － 30 火槍 － 一個在美國西部傳說中的角色。但無論名字來源何處，其基本技術是一樣的，儘管今天典型的微電腦上的磁片比起14英寸磁片體積要小得多，容量要大的多 － 14英寸磁片是七十年代開發“溫徹斯特”技術時典型的磁片。

文件wini.c用於對內核的其他部分隱藏實際的硬碟驅動程式。這使我們能夠實現前一節討論的策略：即在一個內核映象中編譯幾個硬碟驅動程式，並在啟動時選擇一個。以後可以重新編譯使用者安裝程式使得它僅包含一個使用者需要的設備驅動程式。

wini.c包含一個資料定義，hdmap（10013行），它是一個使名字和函數位址相聯繫的陣列。編譯器對該位址初始化，使得需要多少個硬碟驅動程式陣列中就有多少項，這標識在檔include/minix/config.h中。函數winchester\_task使用該陣列，winchester\_task是一個在task\_tab表中的名字，當內核第一次初始化時使用這個函數。當winchester\_task（10040行）函數被調用時，它使用和普通C程式機制相似的內核函數試圖尋找一個hd環境變數，讀MINIX啟動監視程式創建的環境變數，如果hd值沒有定義，就使用陣列中的第一個元素。否則，在陣列中尋找匹配名字，然後間接調用相應的函數。在本節餘下的部分，我們將討論at\_winchester\_task，它是發佈的標準MINIX中hdmap陣列中的第一個元素。

AT風格的驅動程式在at\_wini.c（10100行）中，這是一個面向高級設備的複雜驅動程式，有好幾頁巨集定義了控制器寄存器、狀態位元和命令、資料結構和原型。象其他的塊設備驅動程式一樣，一個driver結構，w\_tab（10274行 到 10284行）被初始化為指向實際完成這項工作的函數指標。這些指針大多數定義在at\_wini.c中，但是由於硬碟不需要特殊的清除操作，所以dr\_cleanup指向了普通的在driver.c中的nop\_cleaanup，和其他不需要特殊清除操作的設備驅動程式共用了該函數。入口函數at\_winchester\_task（10294行）調用一個由硬體決定的初始化過程，然後調用driver.c中的主迴圈。這個迴圈永遠運行，把調用分派到driver表中指向的各種函數。

因為我們正在處理實際的機械電子存放裝置，所以要做一定量的工作來初始化硬碟驅動器。有關硬碟的各種參數保存于定義在10214行至10230行的陣列wini中，作為推遲初始化策略的一部分，由於在必須使用設備以前對設備初始化可能會失敗，所以在內核初始化時調用的init\_params（10307行）並不做任何訪問磁片的工作。它做的主要工作是把有關磁片的邏輯配置資訊拷貝到wini陣列中。這些資訊是ROM BIOS從CMOS記憶體中提取的，“奔騰”類電腦在這些記憶體中存放配置資訊。當機器第一次接通電源時，在MINIX第一部分的裝載過程開始以前，執行BIOS中的功能。取不出這項資訊並不是致命的，如果磁片是現代化的磁片，該資訊可以從磁片直接得到。

調用通用的主迴圈以後，直到試圖訪問硬碟以前，不做任何事情。然後，收到一個請求DEV\_OPEN操作的消息，再間接調用w\_do\_open函數（10355行）。w\_do\_open調用w\_prepare來確定請求的設備是否合法，然後調用w\_identify來確定設備的類型，並初始化在陣列wini中的其他一些參數。最後，使用在陣列wini中的計數器檢測從MINIX啟動以來是否第一次打開設備。檢測完畢後，計數器加一。如果是第一次DEV\_OPEN操作，就調用partition函數。

下一個函數w\_prepare（10388行）接收一個設備的次設備號或使用分區的整型參數device，並返回一個指向device結構的指標，指出設備的基址和大小。在C中，使用一個識別字來命名一個結構並不妨礙使用相同的名字來命名一個變數。一個設備是驅動器分區還是子分區可以由次設備號確定。一旦w\_prepare完成了它的工作，其他任何讀寫磁片的函數都不需關心它們和分區的關係。就象我們看到的那樣，當執行DEV \_OPEN請求時，將調用w\_prepare，該函數也是所有資料傳輸中使用的準備/調度/完成迴圈的一個步驟。在這裡，把w\_count初始化為零是很重要的。

各種軟體相容的AT型磁片已經使用了很長一段時間，w\_identify（10415行）函數不得不把這段時間內引入的各種不同的設計區分開來。第一步是首先檢查在所有這一類磁碟控制卡都應有的一個位址是否存在一個可讀寫的埠（10435至10437行）。如果這個條件滿足，在中斷描述表中安裝硬碟中斷處理常式位址並開放中斷控制器以回應中斷。然後向硬碟控制器發ATA\_IDENTIFY命令。如果返回結果為OK，就取出各項資訊，包括一個標識磁片模型的字串以及設備的物理柱面、磁頭和磁區參數（報告的“物理”配置可能並不是真正的物理配置，但我們只能接受驅動器報告的參數）。磁片資訊也指出了磁片是否允許進行線性塊訪問（Linear Block Addressing － LBA）。如果能，則驅動程式可以忽略柱面、磁頭和磁區參數，使用絕對的磁區號訪問磁片從而簡化訪問磁片操作。

我們以前已經提到，init\_params函數有可能沒有從BIOS表中復原邏輯磁碟設定資訊，如果確實如此，那麼在10469至10477行的代碼將試圖根據它從驅動器本身讀的參數來創建一合適的參數集，其思路是根據原始BIOS資料結構中對應域所允許的位數，把最大的柱面、磁頭和磁區數分別取值為 1023、255和63。

如果ATA\_IDENTIFY命令失敗，可能僅僅意味著磁片為過時的模型，不支援這個命令。在這種情況下，我們僅僅可以得到以前init\_params所讀出的邏輯配置值。如果這些值有效，則把它們拷貝到wini中的物理參數域，否則返回出錯，磁片不能使用。

最後，MINIX使用一個u32\_t變數來對位址按位元組計數。如果柱面數×磁頭數×磁區數的計算結果太大的話（10490行），那麼必須限制以磁區數表示的驅動程式所能處理的設備的大小。雖然在編寫這段程式碼時，在有可能會使用MINIX的機器上，不大可能出現容量超過4G的設備，但經驗告訴我們，應該編寫能經受這些極限參數測試的軟體，而不僅僅是能經受編寫軟體時設備參數的測試。整個設備驅動程式的基址和大小存放在wini陣列中。並調用w\_specify函數來向磁碟控制卡傳遞參數，如果有必要可以再調用一次。最後，在控制台上列印設備的名字（由w\_name確定）和由w\_identify確定的識別字串（如果是現代設備的話）或者列印BIOS報告的柱面磁頭磁區數（如果是以前的設備的話）。

w\_name（10511行）返回一個指向設備名的字串指標，它們是“at\_hd0”、“at\_hd10”或“at\_hd15"之一。w\_specify（10531行）除了向控制器傳參數外，還通過對零柱面尋道來重新校準驅動器（如果是以前的設備的話）。

現在，我們將討論在完成資料傳輸請求時被調用的函數。首先調用的是w\_prepare，這個函數在前面討論過，它把變數w\_count初始化為零工作是很重要的。在資料傳輸過程中調用的下一個函數是w\_schedule（10567行），它建立起基本參數：包括資料從哪裡來、資料到哪裡去、傳輸資料的位元組數（必須為磁區大小的倍數，在10584行進行這項檢查）、傳輸資料是讀數據還是寫資料。在SCATTERED\_IO請求中指示的可選資料傳輸位在向控制器傳輸的操作中復位（10595行），但是注意該位還被保留在iorequest\_s結構的io\_request域中。對於硬碟，驅動程式將試圖滿足所有的請求，但是就象我們看到的那樣，如果出錯的話，驅動程式可能會決定不這樣做。建立基本參數的最後一件事是檢查請求的範圍是否超出了設備上的最後一個位元組，如果超出的話，就減小請求傳輸的資料量。這裡可以計算出所要讀的第一個磁區。

從10602行調度過程開始，如果有等待的請求（通過測試w\_count大於零來檢測）而且如果下一個要讀的磁區和上一個請求的磁區是不連續的，則調用w\_finish函數來結束前一個操作。否則，更新w\_nextblock變數，在這個變數中存放的是下一個磁區號。執行10611行至10640行的迴圈把磁區請求加入請求陣列中。直到在允許的最大範圍以內，到達了允許的請求數目（10614行），該範圍保存在變max\_count中，我們以後將看到，這樣做對我們能夠修改該範圍是有幫助的。這裡再一次可能調用w\_finish。

我們已經看到，在函數w\_prepare中，有兩個地方調用了w\_finish。通常w\_prepare沒有調用w\_finish就終止了。但是無論是否在w\_prepare中被調用，w\_finish（10649行）都要在文件driver.c中的主迴圈中被調用。如果它已被調用過，那末它什麼也不做。所以，在 10659行需檢測這種情況。如果在請求陣列中還有請求，就進入w\_finish的主要部分。

就象我們所估計的那樣，可能會有許多請求，所以w\_finish的主要部分是一個在10664和10761行之間的迴圈。在進入主迴圈以前，通過預先設置變數r為出錯來實施強制重新復位控制器。如果對函數w\_specify的調用成功地完成了command結構中的命令，就初始化cdm來執行資料傳輸。command結構用來把所有需要的參數傳給實際控制磁碟控制卡的函數。一些控制器使用cdm.precomp來補償當磁頭從外柱面向內柱面移動時，由於磁頭下面介質移動的速度不同而引起的磁記錄介質性能的不同。對於特定的驅動器，它是永遠不變的，許多驅動器都忽略該參數。cdm.count 接收傳輸的磁區數，使其為8位元位元組的整數倍，因為驅動器的所有命令和狀態狀態寄存器都是8位元位元組。在10675行至10689行之間的代碼指定傳輸的第一個磁區，或者以28位元邏輯塊號的形式表示（10676行至此0679行），或者以柱面、磁頭和磁區的形式表示（10681行至此0688行），在這兩種情況下都使用了cdm結構中的同一域。

最後，裝入命令本身，並在10692行調用com\_out初始化資料傳輸。如果控制器沒有準備好或者在預先設置的時間範圍內沒有準備好，那麼調用com\_out可能會失敗。在這種情況下，出錯計數加一。如果出錯計數到達了MAX\_ERRORS，則放棄初始化工作，否則在10697行的continue語句使迴圈從10665行重新執行。

如果控制器接收了在調用com\_out時傳遞的命令，可能過一段時間才能得到資料，因此（假定命令為DEV\_READ）在10706行調用了w\_intr\_wait。以後我們將詳細討論這個函數，但是現在僅說明一下它調用了receive函數，這樣磁片任務將被封鎖。

一段時間以後，對w\_intr\_wait的調用返回，時間的長短取決於是否涉及到尋道。雖然有些控制器支援了DMA，但這個驅動程式並未使用，而是使用了程式I/O方式。如果w\_intr\_wait正確返回而且沒有出錯，組合語言函數port\_read將從控制器的資料埠傳輸SECTOR\_SIZE位元組的資料到目的地址，該位址位於檔案系統塊緩衝區中。然後根據成功的資料傳輸調整各種位址和計數以記錄該次成功的資料傳輸。最後，如果當前請求的位元組計數減少為零，就把指向請求陣列的指標向下移動以指向下一個請求（10714行）。

在DEV\_WRITE命令的情況下，第一部分設置命令參數並把命令送入控制器。在命令參數中除了命令操作碼以外和讀操作是一樣的。但對寫操作而言，後續事件的順序是不同的。首先要等待控制器發信號通知它準備好接收資料（10724行）。wait\_for是一個宏，一般很快即可返回。我們在後面還要進一步討論它。現在我們僅僅說明等待最終會超時，但我們希望很少有長時間的超時。然後使用port\_write函數（10729行）把資料由記憶體傳輸到控制器的資料埠，這時要調用w\_intr\_wait函數，阻塞磁片任務。當中斷到來時，磁片任務被喚醒並記錄薄記（10736至10739行）。

最後，如果在讀寫時出錯，必須對出錯進行處理。如果控制器告訴驅動程式由於磁區損壞而出錯，那麼就沒有必要重試了。但對其他類型的出錯至少在某種程度上值得重試一次。這裡所說的程度由出錯計數確定，如果出錯計數達到了MAX\_ERROR就放棄重試。當達到MAX\_ERROR/2時，調用w\_need\_reset實現當重試時強制重新初始化。不過，如果請求是可選的（由SCATTERED\_IO請求確定），不進行重試。

無論w\_finish無錯終止還是出錯終止，變數w\_command總要設置為CMD\_IDLE。這使得其他的函數可以確定不是因為在試圖完成某個操作後由於機械或電子的故障引起出錯而產生了一個中斷。

通過一組寄存器控制磁碟控制卡，在某些系統中，這組寄存器可以被映射為一段存儲區，但在IBM相容的機器中為I/O埠。標準IBM－AT類的硬碟控制器使用的寄存器示於圖3－22中。

這是我們第一次遇到I/O硬體，說明一下I/O埠和記憶體位址的幾點不同對我們是有幫助的。一般地，具有相同I/O位址的輸入輸出埠並不是相同的寄存器，因此寫入某一特定位址的資料不能在以後由讀操作取出來。例如，對於圖3－22中的最後一個寄存器位址，當對其讀時為磁碟控制卡狀態，當對其寫時為向磁碟控制卡發命令。一般讀寫I/O設備寄存器會引起一個獨立於資料傳輸細節動作的發生。AT磁碟控制卡中的命令寄存器就是這樣的。在使用時，通過把資料寫入低編號的寄存器來選擇讀出或寫入的磁片位址，然後把操作碼寫入命令寄存器。把操作碼寫入命令寄存器將啟動具體的操作。

圖3－22（a）IDE硬碟控制器的控制寄存器。括弧中的數是在LBA模式下每一個寄存器選擇的邏輯塊位址位元。（b）驅動器/磁頭 寄存器的選擇域。

也有這種情況，對於不同的操作模式，寄存器或寄存器中的某些域的使用是不同的。在表中給出的例子中，向第六寄存器的第六位LBA位寫入0或寫入1分別選擇使用CHS（柱面－磁頭－磁區模式）或LBA（線性塊定址）模式。向寄存器3 、4、 5寫入的資料或從寄存器3 、4、 5讀出的資料以及第六寄存器的低四位元資料由於LBA位元的不同其解釋是不同的。

現在我們通過調用com\_out（10771行）來研究以下命令是如何發送到控制器的。在改變任何寄存器的內容以前，通過讀狀態寄存器來確定控制器不忙。這個工作是通過檢查STATUS\_BSY位而完成的。在這裡速度是重要的。一般磁碟控制卡總是準備好的或在很短的時間內即可準備好，所以，採用了忙等待的方法。在10779行調用了waitfor來測試STATUS位元，為了提高回應速度，waitfor是一個巨集，它在10268行定義。它使所需的測試只執行一次，以避免當磁片準備好時調用函數產生的開銷。在極少數情況下，當需要等待時，就調用w\_waitfor，w\_waitfor執行迴圈測試直至條件為真或者預定義的超時時間到，因此，如果控制器準備好，那麼將會在最短時間內返回真。如果是暫時性失效，則經過一段時間的延遲後返回真，如果在超過限定時間後還沒有準備好，則返回假。當討論w\_waitfor時，我們將進一步討論超時問題。

一個控制器可以控制多個驅動器，所以一旦確定控制器準備好，通過向控制器寫一個位元組來選擇驅動器、磁頭和操作模式（10785行），然後再次調用waitfor。磁碟機執行命令時，有時會失敗或不能正常地返回一個出錯代碼。畢竟驅動器是機械設備，內部有可能發生各種機械故障。所以作為一項保險措施，要向時鐘任務發送一條消息以安排一個對喚醒常式的調用。然後通過首先向各種寄存器寫入參數再向命令寄存器寫入命令代碼來發出命令。下一步和接下來的對變數w\_command 和w\_status的修改部分為臨界區，因此通過調用lock 和unlock（10801行至10803行）把這個執行序列括起來，這兩個函數用於禁止和打開中斷。

下面幾個函數很短。我們注意到，當出錯計數計到MAX\_ERRORS的一半時，w\_finish函式呼叫了w\_need\_reset函數（10813行）。當等待磁片中斷或準備好時，如果發生超時，也將調用它。w\_need\_reset的工作僅僅是對wini陣列中每個驅動器的state變數做標誌，使得下一次訪問時強制進行初始化。

w\_do\_close（10828行）對常規的硬碟幾乎不作任何工作。當支援的設備為CD\_ROM或其他可移動的設備時，這個常式需要擴展為產生一個命令以打開驅動器或彈出CD，具體如何執行取決於硬體支援。

調用com\_simple來向控制器發出不需要傳輸資料，立即就會結束的一些命令。屬於這一類的命令包括取磁片標識，設置一些參數和重新校準。

當com\_out調用時鐘任務準備在磁碟控制卡失敗以後恢復程式運行時，它使用的一個參數是w\_timeout（10858行）的位址，為當超時發生時，時鐘任務將喚醒該函數。但是一般情況下磁片將完成所請求的操作，於是當超時發生時，將發現w\_command的值為CMP\_IDLE，這意味著磁片完成了它的操作，於是w\_timeout終止。如果命令沒有完成，並且操作是讀請求或是寫請求，那麼減小I/O請求的大小可能會有幫助。這項工作分兩步完成，首先把可以請求的最大磁區數目減小為8，再減小到1，對於所有的超時，列印一條消息，在下一次試圖訪問磁片時，調用w\_need\_reset強制重新初始化所有的驅動器，調用interrupt向磁片任務發一條消息，並類比產生一個應該在磁片操作結束時發生的硬體中斷。

當需要重定時，就調用w\_reset（10889行）。這個函數利用時鐘驅動程式提供的函數milli\_delay。經過使驅動器從以前的操作中恢復過來的初始延遲以後，選通磁碟控制卡中的某一位，也就是把它提到邏輯電平1一段時間，然後恢復為邏輯電平0。執行這個操作以後，調用w\_waitfor給驅動器一段合理的時間以便發信號使其準備好。如果重定不成功，就列印一條消息，返回出錯狀態，下一步如何做將由調用者決定。

發往磁片的有關資料傳輸的命令一般通過產生一個中斷而終止，該中斷向磁片任務發送一消息。事實上，當每一個磁區被讀出或寫入時，都產生一個中斷，因此發出一個命令以後，總要調用w\_intr\_wait（10925行）。接著，w\_intr\_wait在一個迴圈中調用receive，忽略掉每條消息的內容，等待一個把w\_status設置為非忙的中斷。如果接收到了一個這樣的消息，就檢查請求的狀態，這也是一個臨界區，因此，調用lock 和unlock來保證不發生新中斷，也防止了在相關的每一修改步驟完成以前對status的修改。

我們已經在幾個地方見到調用巨集waitfor來在磁碟控制卡狀態寄存器的某一位元上執行忙等待。經過初始測試以後，waitfor宏調用w\_waitfor（10955行），w\_waitfor調用milli\_start來啟動一個計時器，然後進入一個迴圈，輪流檢測狀態寄存器和計時器。如果發生超時，則調用w\_need\_reset，它設置標誌表示下一次請求磁片服務時，需對磁碟控制卡進行復位操作。

w\_waitfor使用的TIMEROUT參數在10206行定義為32秒。另一個相似的參數是WAKEUP，其值取為31秒（10193行），時鐘任務用它安排喚醒事件。考慮到普通進程在其被迫放棄CPU以前僅僅可以運行100ms，那麼這些參數對於忙等待而言，是很長的一段時間，但是，這些數值是基於已公佈的AT類電腦硬碟介面標準的，這些標準指出了磁片旋轉到一定的速度所允許的最長時間為31秒，當然實際上，這是最壞情況下的規範，在大多數的系統中，僅僅在剛加電時或在很長時間不活動以後，才需要啟動旋轉加速。MINIX處於發展之中，當增加支援CD\_ROM或其他經常旋轉的設備時，可以採用新的處理超時的方法。

w\_handler（10976行）是一個中斷處理常式。當硬碟任務第一次被啟動時，w\_identify把這個中斷處理常式的位址送入中斷描述表中。當磁片中斷發生時，磁碟控制卡的狀態寄存器被拷貝到w\_status中，然後調用內核中的interrupt函數，它重新調度硬碟任務。當這些發生時，由於初始化硬碟操作後，w\_intr\_wait調用了receive，所以硬碟任務肯定處於阻塞狀態。

at\_wini.c中的最後一個函數是w\_geometry，它返回被選中的硬碟設備的最大柱面數、磁頭數和磁區數。其值為實際值，而不是象RAM盤驅動程式那樣構造出來的值。

3.7.5軟碟處理

和硬碟驅動程式相比，軟碟驅動程式更長，也更複雜。因為軟碟的結構和硬碟相比要簡單，所以這好象有點不合情理。但是，簡單的機構具有一個簡單的控制器，因此，作業系統就必須考慮更多的內容。可移動介質這一事實也增加了複雜性。在這一節，我們討論一些在處理軟碟時必須考慮的問題。不過我們將不討論MINIX軟碟機的細節，其重要的部分和硬碟是相似的。

對於軟碟驅動程式，我們不必關心的一件事是支持多種控制器類型，而對硬碟，我們是不得不處理的。雖然，初始的IBM－ PC不支援當前使用的高密度軟碟，但是單一的軟體驅動程式可以支援IBM家族中各種電腦上的軟碟控制卡。這種和硬碟形成鮮明對照的情形可能因為對於軟碟而言，沒有象硬碟那樣提高性能的壓力。在電腦系統中，軟碟很少作為工作介質，和硬碟相比，其速度和容量太有限了。不過對於新軟體的發佈和後備，軟碟還是很重要的，因此幾乎所有的小電腦系統都配備至少一個軟盤機。

軟碟驅動程式不使用SSF演算法或電梯演算法，它是嚴格循序執行的。在接收另一個請求以前，要完成已經接收的請求。在初始MINIX的設計中，由於MINIX是為個人電腦而設計的，大多數時間裡僅僅有一個活躍的進程，所以設計者感到一個磁片請求正在處理時，到達另一個磁片請求的可能性很小。把各個請求排成一個佇列需要顯著地增加軟體複雜性，但又不會帶來什麼益處。現在，軟碟除了被配有硬碟的系統用來傳入傳出資料之外很少使用，這使其益處進一步降低。

這就是說，即使在驅動程式軟體中沒有對請求重新排序的支援，軟碟和其他塊設備一樣也能處理分散I/O請求。而且和硬碟驅動程式一樣，軟碟驅動程式搜集放在一個陣列中的請求，當請求順序磁區時，就繼續搜集這些請求。然而，對軟碟驅動程式而言，請求陣列比硬碟驅動程式要小，最大值為軟碟機上每條磁軌的磁區數。另外軟碟驅動程式在處理分散I/O時要考慮OPTION標誌，如果所有當前的請求是可選的，就不會處理一個新的磁軌。

軟碟機的簡單性使得軟碟驅動程式複雜化。廉價的、緩慢的、低容量的軟碟機不值得配置硬碟所使用的複雜的集成控制器，因此驅動程式軟體就不得不處理一些在硬碟中被隱藏於硬碟驅動器的操作。作為一個由軟碟機簡單性而引起的複雜性的例子，考慮在尋道過程中如何把讀寫磁頭定位到特定的磁軌。沒有硬碟會要求驅動器軟體顯式地調用尋道操作。對於硬碟而言，對程式師可見的柱面、磁頭、磁區等幾何結構和物理幾何結構並無對應關係。事實上物理結構是很複雜的。外部柱面比內部柱面要包含更多的磁區，然而，這對於用戶而言是不可見的。作為對磁片按柱面、磁軌、磁區定址的另一種方法，硬碟可以接收按磁片上絕對磁區號編址的邏輯塊位址（LBA）。即使採用柱面、磁軌、磁區編址，由於磁片的集成控制器計算把磁頭移到何處，如果需要執行尋道操作，只要不訪問任何不存在磁區，就可以使用任何幾何結構。

然而對於軟碟，尋道（SEEK）操作需要顯式地程式設計，如果尋道失敗，必須提供一個常式來執行重校準（RECALIBRATE）操作，強迫磁頭回到零柱面。這使得控制器有可能經過一定的步數把磁頭移到需要尋道的位置。對於硬碟也要相同的操作，但硬碟控制器執行了這些操作，而不需要驅動程式軟體的指導。

使軟碟驅動程式複雜化的一些因素是：

1 可移動介質

2 多種磁片格式

3 電機控制

一些硬碟控制器也支援可移動介質，CD\_ROM驅動器就是一個例子，但是即使沒有設備驅動程式軟體的支援，驅動器控制器一般也能處理一些複雜的問題。然而對於軟碟而言沒有這些內嵌的支援，實際上軟碟更需要這些支援。軟碟最常用的用處是安裝軟體和後備檔，經常需要從驅動器中取出軟碟或把軟碟插入驅動器中。把想寫入某張軟碟的資料寫入了另一張軟碟是令人傷心的。設備驅動程式應盡它最大的努力來防止這種情況的發生，雖然有時這是不可能的。因為並不是所有的驅動器硬體都支援檢測從上次訪問磁片以後驅動器是否被打開過。可移動介質可能引起的另一個問題是如果系統試圖訪問一個沒有插入軟碟的驅動器，系統可能掛起。如果能檢測驅動器門是否打開就可解決這個問題。然而，並不總是能進行這種檢測，所以如果對軟碟操作不能在一個合理的時間內終止，就必須提供一些措施實施超時處理和返回出錯。

可移動介質可以被其他的介質所代替。對於軟碟，有許多不同的格式，MINIX支持3.5寸和5.25寸軟碟機。軟碟可以按多種格式格式化，從360KB到1.2MB（5.25寸軟盤機）或1.44MB（3.5寸軟盤機）。MINIX支持七種不同的軟碟格式。對於不同格式引起的問題，有兩種解決方法，MINIX支持這兩種方法。一種方法是把每一種可能的格式看成一個特殊的驅動器，並為設備提供多個次設備號，MINIX就是這樣實現的。在設備目錄中，你將會發現十四個不同的設備，從第一個驅動器的/dev/pc0、360KB、5.25寸軟碟到第二個驅動器的/dev/ps1、1.44MB、3.5寸軟碟。要注意各種不同的組合是很累贅的，因此，MINIX提供了第二種方法。當用第一個/dev/fd0或第二個/dev/fd1第一次訪問軟盤機時，軟碟驅動程式測試在驅動器中所訪問的軟碟以確定其格式。一些格式的柱面比較多，另一些格式每道的磁區數比其他的格式多，通過逐步讀更大的磁軌和磁區數就可確定軟碟的格式。通過一個淘汰過程軟碟格式即可確定下來。當然這需要時間。如果軟碟上存在損壞的磁區還可能出錯。

軟盤機的最後一個複雜問題是電機控制。如果不旋轉，軟碟就不能讀寫。硬碟被設計成連續運行幾千小時也不會失效，但是電機永遠旋轉會使軟盤機和軟碟很快失效。如果訪問驅動器時電機未開，就需要發出命令啟動電機，然後在等待半秒後再試圖讀寫資料。開關電機是很慢的，所以MINIX每次使用完驅動器後，都繼續使電機開幾秒鐘，如果在這段時間內再次使用，則把計時器再延長幾秒。如果在這段時間內沒有使用軟盤機，那麼就關閉電機。

3.8 時鐘

由於許多原因，時鐘（也稱為計時器）對於任何分時操作的系統都是很重要的。它維護每天的時間，並且防止某一進程獨佔CPU而不讓其他實體使用CPU。雖然時鐘既不是象磁片一樣的塊設備，也不是象終端一樣的字元設備，但是時鐘軟體卻可以以設備驅動程式的形式工作。我們對時鐘的討論和前面幾節一樣，首先一般地討論一下時鐘硬體和軟體，然後討論這些想法如何應用於MINIX。

3.8.1 時鐘硬體

在電腦中通常使用兩種類型的時鐘，兩者都和人們使用的鐘和表有很大的不同。簡單的時鐘連接到110或220V的電力線上，以50或60HZ的頻率在每個電壓週期產生一次中斷。

另一種時鐘如圖3－23中所示，由三個元件組成：晶振、計數器和一個保持寄存器。當把石英晶體進行合適的切削並安裝於一定的壓力之下時，它會產生非常精確的週期信號。根據所選晶體的不同，其典型範圍在5至100M之間。在任何電腦系統中，都至少可以發現一個這樣的電路向電腦的各種電路提供同步信號。把這個信號送入計數器並使其遞減計數至零，當計數至零時，產生一個中斷。

典型的可程式設計時鐘有幾種操作模式。在單觸發模式（One-shot Mode）中，當一個時鐘啟動時，它把保持寄存器的值拷貝到計數器中，然後每從晶振來一個脈衝，對計數器值減一，當計數器為零時，產生一個中斷，並停止工作直到再次被軟體啟動為止。在方波模式（Square-wave Mode）中，每次計數至零並引起中斷以後，保持寄存器自動拷貝到計數器，整個過程不斷重複重複執行。這些週期性的中斷稱為時鐘滴答（Clock Tick）。

圖3－23 可程式設計時鐘。

可程式設計時鐘的優點是它的中斷頻率可以由軟體控制。如果使用1M的晶體，那麼計數器每微秒接收到一個脈衝，對於十六位寄存器，中斷可以程式設計為按1微秒至65536微秒的間隔而發生。可程式設計時鐘晶片一般包含兩至三個可程式設計時鐘，並有許多其他的選項（如向上計數還是向下計數，中斷遮罩等）。

為了防止當電腦掉電時丟失當前時間，大多數的電腦有一個電池供電的時鐘，採用使用在數位手錶中的低功耗電路。可以在啟動時讀電池時鐘，如果不存在後備時鐘，軟體可以詢問使用者當前的日期和時間。對於網路系統，存在一個從遠端主機獲取時間的協定。在UNIX和MINIX中，時間被轉換成從通用標準定時（Universal Coordinated Time （UTC））（格林威治平時 － Greenwich Mean Time）1970年1月1日上午12點開始的時鐘滴答數，當然也可以採用其他的時間基準。每一次時鐘滴答，實際時間加1。一般實用程式可以手工設置系統時鐘或後備時鐘，並使其同步。

3.8.2 時鐘軟體

硬體所做的工作僅僅是按已知時間間隔產生中斷。其他和時間有關的工作都必須由軟體驅動程式來實現。在不同的作業系統中，時鐘驅動程式的任務也不同，但一般包括以下內容：

1 維護日期時間。

2 防止進程的執行時間超出其允許的時間。

3 對CPU使用進行記費。

4 處理使用者進程提出的時間鬧鐘系統調用。

5 對系統某些部分提供監視計時器。

6 支援長條圖監視和統計資訊搜集。

第一項功能維護日期時間（也稱為實際時間）並不困難。它僅需要象我們前面提到的那樣，每次時鐘滴答增加一次計數器。需要注意的是日期時間的位數。對於60HZ的時鐘，32位元的計數器剛剛超過兩年就會溢出。顯然系統無法用32位元存儲1970年1月1日開始的滴答數來存儲實際時間。

有三個方法解決這個問題，第一種方法是使用64位元數目器。因為1秒以內需執行多次維護計數器的工作，因此提高了維護時鐘的代價。第二種辦法使用一個輔助計數器來對滴答計數直至累計一秒為止，因為2 32秒超過136年，這種方法直至22世紀都會工作得很好。

第三種方法按滴答計數，但是相對系統啟動時間，而不是相對一個確定的外部時刻。當讀後備時鐘或使用者輸入實際時間時，根據當前的時間計算系統啟動時間並以一種方便的格式存儲於系統中。以後當詢問每天的時間時，存儲的時間加上計數器中的時間就是當前時間。所有這三種方法都顯示在圖3－24中。

圖3－24維護每天時間的三種方法。

時鐘的第二項功能是防止進程運行太長的時間。無論何時啟動一個進程，調度程式都應該用一個以時鐘滴答計算的進程時間片的值初始化一個計數器。當每次時鐘中斷時，中斷驅動程式把時間片計數器減一，當它為零時，時鐘驅動程式調用調度程式來建立其他的進程。

第三項功能為CPU記帳。完成這項工作的最方便的方法是無論何時啟動一個進程就啟動一個和主系統計時器不同的第二個計時器，當進程暫停時，讀出該計時器的內容看一看進程運行了多長時間。為了使記帳正確，當中斷發生時，保存第二個計時器的內容，當中斷結束時，恢復第二個計時器的內容。

一個不太精確但比較簡單的記帳方法是：用一個全域變數維護一個指向當前運行進程在進程表中入口的指標。在每個時鐘滴答，增加當前進程入口表中的一個域的值。通過這種方法，每個時鐘滴答都被滴答時刻所運行的相應進程所“支付”。這種方法的一個小問題是如果在一個進程運行過程中，發生了多次中斷，即使進程沒有做多少工作，該進程也支付了一個完整的滴答。由於對中斷期間CPU記帳的合適的方法代價太高，因此從未使用過。

在MI NIX和許多其他的系統中，一個進程可以請求作業系統在一定的間隔後對它報警。報警通常是信號、消息或其他類似的東西。一個需要這些報警的應用是網路。在網路中，對在一段時間內沒有應答的包必須重傳。另一個應用是電腦輔助教學，當學生在一段時間內沒有提供回應時，就告訴他答案。

如果時鐘驅動程式有足夠的時鐘，它可以為每一個請求設置一個時鐘。但實際情況不是這樣。它必須使用單一物理時鐘模擬多個虛擬時鐘。一種辦法是：設置一張表格和一個變數，在表中保存所有等待計時器的信號時間，變數則給出下一次發送信號的時間。無論何時更新時間，驅動程式都要檢查是否到了最近的信號發送時間。如果時間到，就在表格中搜索要發送的下一個信號。

圖3－25 用一個時鐘類比多個計時器。

如果需要許多信號，那麼象圖3－25中所示那樣，把所有的時鐘請求在一個鏈表中連接起來並按時間排序，以此類比多個時鐘將更加有效。鏈表中的每個入口指出前一信號發生以後需要等待多少個滴答才引發下一個信號。在這個例子中，信號發生在 4203、4207、4213、4215、4216。

在圖3－25中，下一個中斷發生在三個滴答以後。每次滴答，next\_signal減一，當它為零時，就引發鏈表中第一項的信號，並把這一項從鏈表中移走，然後next\_signal設置為鏈表中第一個元素的值，在這個例子中為4。

注意在時鐘中斷期間，時鐘驅動程式有幾件事要做 － 遞增實際時間、遞減時間片並檢查是否為零、CPU記帳、遞減報警計數器。因為這些工作每秒都要重複多次，每項工作都必須仔細安排以加快速度。

作業系統的有些部分也需要計時器，這些是所謂的監視計時器（watch-dog Timer）。當研究硬碟驅動程式的時候，我們看到，每次向硬碟控制器發命令時，都要安排喚醒調用，以便當命令執行完全失敗時，可以試圖進行恢復。我們也注意到軟碟驅動程式不得不等待電機加速到一定的速度，並且如果在一段時間內沒有任何活動，就關掉電機。一些具有可移動頭的印表機可以每秒列印120個字元（8.3ms/字元），但是不能在8.3ms內把打印頭返回最左邊。當列印回車時，終端驅動程式必須延時。

時鐘驅動程式處理監視計時器的機制和使用者信號是一樣的。唯一的不同是當計時器時間到時，時鐘驅動程式將調用一個調用者提供的過程而不是產生一個信號，這個過程是調用者代碼的一部分。因為所有的驅動程式處於同一位址空間，因此時鐘驅動程式可以以任何方式調用它們。被調用的過程可以做需要做的任何工作，甚至可以引起一個中斷，雖然在內核中，中斷是不方便的，信號也不存在。這就是為什麼要提供監視計時器機制的原因。

我們所列的最後一件事是長條圖統計。一些作業系統提供一種機制使使用者程式可以要求系統構造程式計數器長條圖，以便分析時間在程式的各部分是如何分配的。當有可能統計長條圖時，在每一時鐘滴答，驅動程式都要檢查是否對當前程式進行長條圖統計，如果進行長條圖統計，則先計算當前程式計數器值所在的位址區段，然後把相應的計數器加一。這種機制也能用來對系統本身進行長條圖統計。

3.8.3 MINIX時鐘驅動程式概述

MINIX時鐘驅動程式包含在檔clock.c中，時鐘任務接收六個帶有如下所示參數的消息類型：

1. HARD\_INT

2. GET\_UPTIME

3. GET\_TIME

4. SET\_TIME（以秒計的新時間）

5. SET\_ALARM（進程號，調用過程，延遲）

6. SET\_SYN\_AL（進程號，延遲）

HARD\_INT是當發生時鐘中斷並有工作可做時發往驅動程式的消息，例如當必須發出一個報警或一個進程已經運行了很長時間的情況。

GET\_UPTIME用來取從啟動開始後以滴答計數的時間。GET\_TIME返回從1970年1月1日上午12：00開始以秒計的當前時間。SET\_TIME設置實際時間，它可以被超級用戶啟動。

時鐘驅動程式內部，按圖3－24（c）的方法記錄時間。當設置時間時，驅動程式計算何時系統被啟動，由於驅動程式維護當前的實際時間，它也知道系統已經運行了多少滴答數，所以它可以計算系統是何時啟動的。系統在一個變數中存放啟動的實際時間，以後當調用GET\_TIME時，把滴答計數的當前值轉換成秒，並把它和存儲的啟動時間相加。

SET\_ALARM允許一個進程設置一個計時器，該計時器經過一個指定滴答計數的時間間隔後，將引起某些事件的發生。當使用者進程執行ALARM調用時，它向存儲管理器發一條消息，管理器再把消息發往時鐘驅動程式，當報警發生時，時鐘驅動程式向存儲管理器發回一條消息，然後由它向相關進程發回一個信號。

需要啟動監視計時器的任務也使用SET\_ALARM，當計時器到達時，就簡單地調用提交的過程。但驅動程式並不知道調用的過程做什麼。

SET\_SYN\_AL和SET\_ALARM相似，但是用來設置同步鬧鐘（Synchronous Alarm）。同步鬧鐘發送一條消息到進程而不是產生一個信號或調用一個過程。同步報警任務負責向各個需要消息的進程分派消息，在後面將仔細討論同步報警。

時鐘任務並沒有使用主要的資料結構，但使用了幾個變數來保存時間。只有一個全域變數lost\_ticks，它定義在檔glo.h中（5031行）。如果未來加入到MINIX系統中的驅動程式，由於遮罩中斷時間太長以至於丟失一個或多個時鐘滴答的話，可以使用這個變數來補償。現在並未使用這個變數，但是如果編寫了這樣一個設備驅動程式，那麼驅動程式可以通過增加lost\_ticks來補償在遮罩中斷期間丟失的時間。

很明顯，時鐘中斷發生的頻率很高，對時鐘中斷的快速處理是很重要的。MINIX通過對大多數的時鐘中斷進行最少的處理來實現這個目標。一旦接收到中斷，處理常式把區域變數ticks設置為lost\_ticks+1，然後使用這個值更新統計時間和pending\_ticks（11079行），並把lost\_ticks復位為零。pending\_ticks是一個PRIVATE類型的變數，聲明在所有函式定義之外，但僅對定義於clock.c檔中函數可見。另一個PRIVATE類型的變數sched\_ticks1用於跟蹤執行時間，每次滴答減一，如果報警時間到或允許時間片用完，中斷處理常式就向時鐘任務發送一條消息。這種方法使得大多數中斷處理常式可以立即返回。

當時鐘任務接收到任一消息時，它把pending\_ticks加到變數realtime（11067行）上，然後置變數pending\_ticks為零。變數realtime和變數boot\_time（11068行）用以計算每天的當前時間，它們都是PRIVATE類型的變數。所以對於系統的任何部分唯一的取時間的方法是向時鐘任務發一消息。雖然在任意瞬間，realtime可能是不精確的，但這種機制能保證當需要的時候，時間永遠是精確的。如果你的手錶當你看它的時候是準時的，而當你不看它的時候，它卻不準時，這有關係嗎？

為了處理鬧鐘，next\_alarm記錄了下一個信號或監視計時器調用發生的時刻。驅動程式在這裡必須很謹慎，這是因為在信號發生以前，請求信號的進程可能已經終止或被殺死。當信號發生時，要檢查一下是否需要該信號，如果不需要，也就沒有必要處理了。

每一個使用者進程只允許有一個未完成的鬧鐘計時器。當計時器還在工作時，執行一個ALARM調用會終止上一個計時器，因此，一種簡便地保存計時器的方法是在每一個進程的進程表項中為計時器保留一個字。對於各個任務，被調用的函數必須在某處存放，所以設置了陣列watch\_dog。一個相似的陣列syn\_table用來存儲指示對於每一個進程是否等待接收同步鬧鐘的標誌。

時鐘驅動程式的整個邏輯和磁片驅動程式具有相同的模式。主程序是一個無限迴圈，重複執行取消息，再根據消息類型進行處理，然後發送一個應答（CLOCK\_TICK除外）。每種類型的消息由不同的過程處理，它們按我們的標準命名約定命名，即對於從主迴圈調用的過程，都命名為do\_XXX的形式，當然，各個XXX是不同的。順便說一句，不幸的是，許多連接程式把各個過程名削減為7或8個字元，所以名字do\_set\_time和do\_set\_alarm有可能發生衝突，因此後一名字換名為do\_setalarm，在整個MINIX中，這個問題都可能發生，一般通過更改名字來解決。

同步鬧鐘任務（Synchronous Alarm Task）

在這一部分，我們討論另一類任務：同步鬧鐘任務（Synchronous Alarm Task）。同步鬧鐘和鬧鐘很相似，但是當計數時間到時，它既不發信號，也不調用監視計時器函數，而是向鬧鐘任務發一條消息。到達的信號或者調用的監視計時器任務可能和當前正在執行的任務部分毫無關係，所以所有這些類型的鬧鐘都是非同步（Asynchronous ）的。與此相反，只有在接收方執行了receive調用後，才接收消息。

同步鬧鐘機制加入到MINIX中是為了支援網路服務器，它就象存儲管理器和檔案伺服器一樣，是作為一個獨立的進程運行的。當進程因等待輸入而被阻塞時，經常要設置一個等待時間的極限。例如，在網路中，在一定的時間內，可能由於傳輸失敗而沒有收到資料包的應答，網路服務器可以在試圖接收消息並阻塞以前設置一個同步鬧鐘。因為同步鬧鐘是作為一條消息遞交的，所以如果沒有從網路中收到消息，最後它也將使伺服器解除阻塞。如果收到了消息，伺服器必須首先重定鬧鐘，然後檢查消息的類型和來源，由此可以確定是資料包到達還是由於超時而解除阻塞。如果是後者，伺服器一般通過重新發送最後一個沒有確認的包來試圖恢復。

同步鬧鐘比發送信號的鬧鐘要快，後者需要幾條消息和一定量的處理，監視計時器函數也很快，但僅適用於和時鐘任務編譯進同一空間的任務。當進程在等待消息時，同步鬧鐘比信號或監視計時器更合適也更簡單，而且僅僅需要一些附加操作即可很方便地處理。

時鐘中斷處理常式

就象前面討論的，當時鐘中斷發生時，並不是立即更新realtime。中斷常式維護變數pending\_ticks計數器，並且完成象把當前的滴答計入到進程中並減小當前的時間片等簡單工作，僅當必須完成更複雜的任務時，才向時鐘任務發消息。這是和MINIX任務通過消息通信思想的一種妥協，但是它是對消耗CPU時間的一種讓步。在一個慢速的機器上，人們發現，和在每個時鐘中斷向中斷任務發消息的工作方式相比，按這種方式工作可以提高15％的速度。

毫秒定時

作為對實際情況的另一種妥協，在clock.c中提供了幾個常式用於實現毫秒定時。有許多設備需要不超過一毫秒的延時，使用鬧鐘和消息傳遞介面沒有辦法解決這個問題。在這裡提供的函數是被任務直接調用的，使用的是最古老和最簡單的I/O技術：輪詢法。按最快的速度直接讀用於產生時鐘中斷的計數器，並把計數值轉換為毫秒。調用者重複執行該過程直至經過了所要求的時間。

時鐘服務小結

圖3－26總結了clock.c提供的各種服務。某些服務是所有的進程都可得到的，其結果通過消息返回。可以從內核或任務發出的函式呼叫獲得時間，這種方法避免了消息機制的開銷。使用者進程可以請求一個鬧鐘。其結果是產生了一個信號，任務也可請求一個鬧鐘，其結果是啟動監視計時器函數。這兩種機制伺服器系統都不能使用，但伺服器可以請求同步鬧鐘。任務或內核可以通過mill\_delay函數請求一個延遲，也可以在輪詢常式中調用mill\_elapsed，例如當等待從埠輸入時就是如此。

3.8.4 MINIX時鐘驅動程式的實現

當MINIX啟動時，將調用所有的驅動程式。大多數驅動程式試圖取一條消息並阻塞。時鐘驅動程式clock\_task（11098行）也是這樣做的。但是它首先要調用init\_clock來把可程式設計時鐘的頻率初始化為60HZ。當收到其他消息時，它把pending\_ticks加到real\_time上，然後在做其他任何事以前把pend\_ticks復位。這個操作可能會和時鐘中斷發生衝突，因此調用了lock和unlock兩個函數來避免衝突（11115行至11118行）。時鐘中斷處理常式除主迴圈以外的其他部分和其他的驅動程式是相同的：接收一條消息，調用一個完成所需要工作的函數，然後回送一個應答消息。

圖3－26 支援和時間有關服務的時鐘代碼。

在時鐘的每次滴答，並不都調用do\_clocktick（11140行），因此其名字並不是其函數功能的準確描述。當中斷處理常式確定有一些重要的工作需要做時才調用它。首先檢查是否有信號或監視計時器事件到。如果其中之一發生，則檢查在進程表中所有的鬧鐘項。因為時鐘滴答並不是每個單獨處理的，所以在一遍對進程表的掃描中可能會發現多個鬧鐘時間到。也有可能接收下一個鬧鐘的進程已經終止。當發現一個進程其鬧鐘時刻比當前時間要小，但不為零時，則檢查和該進程相對應的在陣列watch\_dog中的項。在C語言中，數值型的值也具有一個邏輯值，因此如果在watch\_dog陣列中存放一個有效位址，11161行的測試將返回TRUE，並在11163行間接調用對應的函數。如果發現為空指標（NULL pointer）（在C中表示為數值零），其測試結果將為FALSE，並調用cause\_sig來發送一個SIGALRM信號。當需要同步鬧鐘時，也使用監視器項，在這種情況下，存放的位址為cause\_alarm而不是屬於特定任務的監視器函數的位址。對於發送一個信號，當然可以存放cause\_sig位址，但是然後將必須編寫不同的cause\_sig函數。該函數不需要參數，它從一個全域變數中取得目標進程號。另外，我們也可以要求所有的監視器進程希望一個它不需要的參數。

在以後的章節裡，當我們討論系統任務時，我們將討論cause\_sig。它的工作是發一條消息到存儲管理器。這裡需要檢查是否存儲管理器正在等待該消息。如果確實如此，它發一條消息通知鬧鐘時間已到。如果存儲管理器忙，那麼在第一次時將做一個標記來通知它。

當迴圈檢查進程表中每個進程的p\_alarming值時，更新next\_alarm。在啟動迴圈以前，它被設置為一個很大的數（11151行），然後對於每一個在發送了鬧鐘或信號後其鬧鐘值為非零的進程，將其鬧鐘值和next\_alarm進行比較，後者被設置為較小的值（11171行和11172行）。

處理完鬧鐘以後，do\_clockticks繼續執行，檢查是否到了調度另一進程的時刻。執行時間片的值保存在PRIVATE類型的變數shed\_clocktick中，一般在每次時鐘中斷處理常式中將其減1。然而，在那些do\_clocktick被啟動的嘀噠中，中斷處理常式並不減小它的值，而是讓do\_clocktick自己執行這項工作並測試結果是否為零（11178行）。當調度新進程時並不對sched\_ticks重定（因為允許檔案系統和存儲管理程式完成這項工作），而是每次SCHED\_RATE嘀噠後對其復位。11179行中的比較是用來保證當進程在被剝奪CPU以前確實至少運行了一個完整的調度滴答。

下一個過程do\_getuptime（11189行）只有一行，它把realtime的當前值（從啟動開始經過的滴答數）存入返回消息的正確域中，任何進程都可以通過這種方式取得已經過的時間。但是，對於存取時間的任務，消息傳遞的代價是很大的，因此提供了一個相關的函數get\_uptime（11200行），它可以被任務直接調用。因為不是通過發送給時鐘任務的消息啟動其運行，所以它必須把耽誤的時鐘滴答加到當前的realtime上。在這裡，lock和unlock也是必須使用的，它們用來防止當訪問pend\_ticks時發生時鐘中斷。

為了取得當前的時間，do\_get\_time（1121行）根據realtime和boottime（以秒計的從系統啟動開始計算的時間）計算當前時間。do\_set\_time（11270行）則與do\_get\_time相反，它根據給定的當前時間和從啟動開始計數的滴答數為boot\_time計算一個新值。

過程do\_setalarm（11242行）和do\_setsyn\_alarm（11269行）非常相似，因此放在一起討論。二者都從消息中提取參數，一個參數說明要向其發送信號的進程，另一個參數說明等待消息的時間。do\_setalarm也需要提取一個調用函數的參數。如果目標進程是使用者進程而不是任務，調用函數可以取空指標。在前面我們已經說明在do\_clocktick中對該指標進行檢測來確定目標進程應當得到一個信號還是調用一個監視計時器函數。這兩個函數也計算鬧鐘需等待的時間（以秒計）並把計算結果設置在返回的消息中。然後二者都調用common\_setalarm完成相應的操作。如果是do\_setsyn\_alarm，傳給common\_setalarm的函數參數將永遠是cause\_alarm。

common\_setalarm（11291行）完成上面所討論的兩個函數所啟動的工作，然後把鬧鐘時間存放在進程表中，把指向監視計時器過程的指標存在watch\_dog陣列中（也可能是指向cause\_alarm的指標或是空指標），然後它象do\_clocktick一樣掃描整個進程表尋找下一個鬧鐘。

cause\_alarm（11318行）很簡單，它把syn\_table陣列中對應於同步鬧鐘目標的項設置為TRUE。如果同步鬧鐘任務沒有處於活躍狀態，就發送一條消息喚醒它。

同步鬧鐘任務的實現

同步鬧鐘任務syn\_alarm\_task（11333行）和所有任務的模型是一樣的。它首先初始化，然後進入一個迴圈重複接收或發送消息。初始化工作包括：首先通過設置syn\_al\_alive為真宣佈任務為活躍狀態，然後通過設置在syn\_table中所有的項為FALSE宣佈它無事可作。對於進程表每一項，在syn\_table中都有一對應項。通過宣佈已完成了工作，開始其外層迴圈，然後進入內層迴圈，其中檢查在syn\_table中每一項，如果發現一項指示等待一個同步鬧鐘，則首先把該項復位，然後發送一個類型為CLOCK\_INT的消息至相應進程，並宣佈其工作尚未完成。在外層迴圈的底部，除非設置了work\_done標誌，否則將不等待任何新消息。因為cause\_alarm直接寫入了syn\_table中，所以並不需要一個新消息來告訴還需要做其他工作。僅當完成了所有工作以後才需要一條消息將它喚醒。其效果是只要有鬧鐘需發送就以很快的速度迴圈處理。

事實上，MINIX的發佈版本並不使用這個任務。然而，如果重新編譯MINIX來增加網路功能，網路服務器將會使用它，這是因為如果希望接收的包沒有接收到，那麼需要需要這種機制，以便加快進行超時處理。除了對速度的要求以外，使用這種機制的另一個原因是伺服器應當不停地永遠工作，而大多數信號的預設動作是撤銷目標進程，所以不能向伺服器發信號。

時鐘中斷處理的實現

時鐘中斷處理常式的設計是做很少的工作（因此使得處理時間很短）和完成較多的工作之間的妥協，後者使得費時的時鐘任務的啟動不太頻繁。它僅僅改變和測試一些變數。clock\_handler（11374行）先對系統記帳進行處理。MINIX既記錄使用者執行時間，也記錄系統執行時間。如果時鐘滴答時某進程正在運行，則對其用戶時間記帳。如果檔案系統或存儲管理器正在運行，則對系統時間記帳。變數bill\_ptr永遠指向被調度的最後一個使用者進程（不對兩個伺服器記帳），11447和11448行執行記帳，計算完畢後，把由clock\_handler維護的最重要的變數pending\_ticks加一（11450行）。為了測試是喚醒tty還是向時鐘任務發送一條消息，必須知道實際的時間。但是，由於更新realtme要使用鎖機構，所以其代價比較高。為了解決這個問題，中斷服務程式計算自己的實際時間，並存放在區域變數now中。有可能其結果暫時是錯誤的，但結果並不嚴重。

驅動程式的其他工作取決於各種測試。它需要不停地喚醒終端和印表機。tty\_timeout是一個由終端任務維護的全域變數，在這個變數中存放的是終端任務下一次被喚醒的時刻。對於印表機，需要檢查幾個印表機模組中PRIVATE類型的變數，再調用pr\_restart對這些變數進行測試，在印表機掛起這種最壞的情況下，該函數也能很快返回。如果鬧鐘時間到或到了應該進行任務調度的時候，就執行11455至11458行的測試以啟動時鐘任務。後面的測試比較複雜，它是三個簡單測試的邏輯與。11459行的代碼

interrupt（clock）

引起發送一個HARD\_INT消息到時鐘任務。

當討論do\_clocktick時，我們注意到它對sched\_ticks減一併測試是否為零以檢查執行時間片是否用完。測試sched\_ticks是否等於一是我們前面提到的複雜測試的一部分，如果時鐘任務沒有被啟動，則有必要在中斷處理常式中繼續對sched\_ticks執行減一操作，如果減至零，則對時間片復位。如果這發生了，注意這也是當前進程一個新的時間片的開始，這是通過在11466行把當前bill\_ptr的值賦值為prev\_ptr來完成的。

時間實用程式

最後，clock.c包含了一些提供各種支援的函數。它們中有許多是和硬體相關的。當把MINIX移植到非Intel的硬體時，必須替換這些程式。我們只討論它們功能而不詳細討論其內部細節。

時鐘任務第一次運行時調用init\_clock（11474行），它設置計時器晶片的模式和時間延遲以產生每秒60次的時鐘滴答中斷。儘管人們在為PC做的廣告上看到CPU的速度從初始的IBM－PC的4.77MHZ上升到在當今系統中200MHZ以上。但是，無論MINIX運行在哪種PC模型，初始化計時器的常數TIMER\_COUNT都是不變的。每一個IBM兼容機，無論運行多快，都為各種需要基準時間的設備提供了一個14.3MHZ的信號。串列通信線和視頻顯示都需要這樣一個基準。

和init\_clock相對應的是clock\_stop（11481行），它並不是必須的，而是對MINIX使用者有時希望啟動另一個作業系統這一事實的讓步。它僅僅把計時器晶片的參數重定為預設操作方式，這正是MS－DOS或其他作業系統第一次開始運行時期望ROM BIOS提供的

mill\_delay（11502行）是為需要極短延遲的任務而提供的。它用C語言編寫，沒有引入任何硬體相關性，但是使用了一種人們只有在低級組合語言中找到的技術。它把計數器初始化為零，然後對其快速輪詢直到到達指定的值。在第二章裡，我們說過這種忙等待技術一般應該避免，但是，實現的必要性要求不能遵循一般的規則。下一個函數mill\_start實現計數器的初始化（11516行），它只是簡單地把兩個變數清零。通過調用最後一個函數mill\_elapse（11529行）實現輪詢，它訪問計時器硬體。被檢查的計數器就是用來對時鐘滴答下降計數的那個計數器，因此它可能下溢並在需要的延遲以前被重定為最大值。mill\_elapse會糾正這個錯誤。

3.9終端

每台通用電腦都有一個或多個用來和它通訊的終端。終端有大量不同的型號，需要終端驅動程式來遮罩這些細節，從而作業系統和使用者程式的設備獨立性部分對於不同型號的終端不必重新編寫。在下面幾節，我們首先一般地討論終端硬體和軟體，然後討論MINIX軟體。

3.9.1終端硬體

從作業系統的觀點來看，根據作業系統如何和終端通信，終端被分成三類：第一類為存儲映射終端，包括鍵盤和顯示器，二者都直接與電腦相連。第二類為使用RS－232標準串列通訊線，一般還經由數據機構成序列介面的終端。第三類為通過網路連接到電腦上的終端。

存儲映射終端

圖3－27中的第一類終端是存儲映射終端。這些終端是電腦整體的一部分。存儲映射終端使用稱為視頻RAM（Video RAM）的特殊記憶體，視頻RAM是電腦位址空間的一部分，通過和其他位址空間一樣的方式對它進行訪問。

圖3－27 終端類型。

視頻存儲卡上有一個晶片稱為視訊控制器（Vedio Controller）。這個晶片從視頻RAM中取出字元，產生用於驅動顯示器（監視器）的視訊訊號。監視器產生水準掃描螢幕的電子束在螢幕上劃線，典型的螢幕有480至1024行，每行有640至1200點。這些點稱為象素（Pixel）。視訊控制器調節電子束，決定一個象素是亮的還是黑的。彩色監視器有三個電子束，分別對應紅色、綠色和藍色，它們各自獨立調節。

圖3－28 存儲映射終端直接寫入視頻RAM。

一個簡單的單色顯示器可能把一個字元顯示為寬度為9個象素，高度為14個象素（包括字元間的空白），共顯示25行，每行80個字元。這些顯示器有350行掃描線，每行掃描線有720個點，每幀每秒重畫45至70次。視訊控制器被設計成首先從視頻RAM中取80個字元，產生14行掃描線，再取80個字元，再產生14行掃描線，這樣一直工作下去。事實上，大多數視訊控制器顯示每個字元的每行掃描線時，都取一次字元以便在控制器中不需要緩衝。每個字元的9列寬14行高的位模保存在視訊控制器的視頻ROM中（也可以使用RAM以支援使用者字體）。ROM按12位元編址，8位元來自字元代碼，4位元指定掃描線。ROM中每個位元組的8位元控制8個象素，字元間的第九個象素永遠為空。因此螢幕上的每行文本需14×80＝1120次記憶體訪問，也需訪問相同次數的字元發生器。

IBM－PC有幾種螢幕模式，在最簡單的模式中，控制台使用一個字元映射顯示器。在圖3－29（a）中，我們看到了視頻RAM的一部分。在圖3－29（b）中螢幕上的每個字元在RAM中占兩個位元組，低位元組是顯示字元的ASCII碼，高位元組為屬性位元組，用於指定顏色、反顯、閃爍等等。在這種模式下，滿屏25行80列字元需4000位元組的視頻RAM。

圖3－29（a）IBM單色顯示器的視頻RAM圖像。 （b）和（a） 相對應的螢幕。×為屬性位元組

除了每個象素是獨立控制之外，位元映射終端使用相同的原理，對於單顯這種最簡單的配置，每個象素對應視頻RAM中的一位元，對於最複雜的配置，每一象素用24位的數來表示，紅色、綠色和藍色各8位。一個24位元象素的768×1024的彩色顯示器需2MB的RAM來存放圖像。

對於記憶體映射顯示器，鍵盤是與顯示器分開的，它可能通過一個串列口或並行口和電腦相連。對於每一個鍵動作，產生CPU中斷，鍵盤中斷程式通過讀I/O口取得鍵入的字元。

在IBM－PC中，鍵盤包括一個內嵌的微處理器，通過特殊的串列口和主機板上的一個控制晶片通訊。任何時刻擊鍵或釋放鍵，都產生一個中斷，而且鍵盤僅僅提供鍵碼，而不是ASCII碼。當擊A鍵時，鍵碼（30）被存放於I/O寄存器。輸入字元是大寫、小寫、CTRL－A、ALT－A、CTRL－ALT－A還是其他的組合則由驅動程式確定。因為驅動程式知道哪些鍵被按下還沒有釋放，因此它有足夠的資訊完成這項工作。雖然鍵盤介面把全部工作都交給了軟體，但這也提供了很大的靈活性。例如，使用者程式可能對一個數位是來源於最上面一行的鍵還是旁邊的數位小鍵盤感興趣。原則上，驅動程式可以提供這項資訊。

RS－232終端

RS－232終端是包括一個鍵盤和一個顯示器的設備，通過一次傳輸一位元的串列口與電腦通訊（參見圖3－30）。這些終端使用9針或25針的連接器，其中一針為發送資料，一針為接收資料，一針為地，其他各針可用於各種控制功能，實際上大多數並未使用。為了向RS－232終端發一個字元，電腦必須一次傳輸一位元，在字元前面加一起始位元，後接一個或兩個終止位元為字元定界。可以在終止位前插入一提供基本校驗的奇偶位，但是通常僅在與主機系統通訊時才需要這種技術。一般傳輸速率為9600、19200或38400 bps。RS－232終端通常用於和使用遠端電腦通訊，兩者之間使用數據機及電話線。

圖3－30一個RS－232終端通過一條一次傳輸一位元的通訊線路和電腦通訊。電腦和終端完全獨立。

電腦和終端在內部都是對整個字元進行操作的，但又必須通過串列線路以一次傳輸一位元的方式來通訊，因此開發了晶片來實現字元到串列口和串列口到字元的轉換。它們被稱為UART（通用非同步收發器 － Universal Asynchronous Receiver Transmitter）。UART通過象示於圖3－31那樣把RS－232介面板插入匯流排和電腦相連。RS－232終端正逐漸消失，而由PC和X終端代替，但是它們用於一些老的大型系統中，特別是銀行、飛機定票等類似的系統中。

為了顯示出一個字元，終端驅動程式把字元寫入介面卡，這個字元被緩衝在介面卡中，然後通過UART在串列線上一位元一位地發送出去。即使傳輸速率為38400bps，也需要250個毫秒來發送一個字元。由於傳輸速率很低，驅動程式一般向RS－232卡輸出一個字元後就阻塞，等待字元傳輸完畢以後產生中斷。UART能同時接收另一個字元。象其名稱所指出的，UART可以同時發送和接收字元。當接收到一個字元時一般也產生中斷，UART能夠緩衝少量的字元。當接收到中斷時，終端驅動程式必須檢查一個寄存器以確定中斷源。一些介面卡有CPU和記憶體，並能操縱多條線路，從而承擔了許多原先由CPU完成的I/O工作。

正如前邊提到的，RS－232終端可以進一步分成幾類。最簡單的是硬拷貝終端。通過鍵盤鍵入的字元傳輸至主機，主機傳出的字元列印在紙上。這些終端已過時並很少見到。

啞CRT終端也按這種方式工作，僅僅用螢幕代替了紙，因為在功能上和硬拷貝是一樣的，這些終端常被稱為“玻璃tty"（術語tty是TeleType的縮寫，TeleType是一個公司，過去該公司是電腦終端企業的先驅；tty的含義為任何終端）。玻璃終端也已過時。

智慧終端機事實上是微縮的專用電腦，它們有CPU、記憶體和軟體，軟體一般在ROM中。從作業系統的觀點，玻璃終端和智慧終端機的不同在於後者可以理解特殊的轉義字元序列，例如通過發送ASCII ESC字元（033）後接各種其他的字元可以把游標移至螢幕上任何位置，在螢幕中間插入文本等等。

X終端

智慧終端機最高檔的一種是包含和主機CPU一樣強大的CPU的終端，同時包含幾百萬位元組的主存、鍵盤和滑鼠。這種類型的一種常用的終端就是X終端，在其上運行MIT的X視窗系統。一般X終端通過乙太網和主機通訊。

一個X終端是運行X軟體的電腦。一些產品只能運行X，其他的為通用電腦，把X作為和其他程式一樣的程式來運行。無論哪種方式，X終端都有一個大的位元映射螢幕，一般960×1200或更高，黑色 、灰色或彩色，一個完整的鍵盤，一個滑鼠。滑鼠一般有三個按鈕。

X終端內收集從鍵盤或滑鼠來的輸入並接收遠端電腦命令的一個程式稱為X伺服器，它通過網路和運行在遠端主機上的X客戶通訊。使X伺服器運行於終端內，客戶程式運行于遠端主機上顯得很奇怪，但是X伺服器的工作是位元顯示，因此使它靠近用戶是有益的。客戶和伺服器的管理示於圖3－31中。

圖3－31 M.I.T X視窗系統中的客戶和伺服器。

X終端的螢幕包含一些視窗，每個視窗都是長方形象素網格形式，在其頂部有一標題條，左邊有一捲軸，在右上角有一改變視窗大小的小方框。一個X客戶是一個稱為視窗管理器（Window Manager）的程式，它的工作是控制在螢幕上創建、刪除和移動視窗。為了管理視窗，它向X伺服器發一命令，告訴它做什麼。這些命令包括劃點、劃線、劃矩形、劃多邊形、填充矩形、填充多邊形等等。

X伺服器的工作是調度來源於滑鼠、鍵盤和X客戶的輸入並更新顯示。它要跟蹤當前選中了哪個視窗（滑鼠所指的），所以它知道鍵盤輸入的內容送給哪個客戶。

3.9.2終端軟體

鍵盤和顯示器幾乎是獨立的設備，因此我們分別討論它們（它們並不是完全獨立的，因為鍵入的字元必須在顯示器上列印出來）。在MINIX中，鍵盤和顯示器是同一任務的不同部分，在其他的系統中，它們可能分成不同的驅動程式。

輸入軟體

鍵盤驅動程式基本的工作是收集從鍵盤輸入的資訊，當使用者程式讀終端時，把它傳輸給使用者程式。對於鍵盤驅動程式的設計可以採用兩種思想：第一，驅動程式的工作僅僅是接收輸入，並且不經任何修改就向上層傳遞。一個從終端讀的程式得到純粹的ASCII碼序列（為使用者程式提供鍵碼太原始，也過分依賴於機器）。

這種思想非常適用於象emacs這樣複雜的螢幕編輯器的需要。emacs允許使用者把任意動作捆綁到任意字元或字元序列上。然而，這意味著如果用戶鍵入了dste而不是date，然後鍵入三個退格建和ate鍵來對此進行修正，最後鍵入一個回車鍵，那麼使用者程式將收到鍵入的全部11個ASCII碼。

大多數程式不需要這麼多細節，它們僅僅希望得到正確的輸入，而不是如何生成它的完整的序列。基於這一點，產生了第二種思想：驅動程式處理行內的編輯，僅僅向使用者程式傳輸正確的一行。第一種思想是面向字元的，第二種思想是面向行的。我們把它們分別稱為原始模式（Raw Mode）和熟模式（Cooked Mode）。POSIX標準使用了一個不太形象的術語規範模式（Canonical Mode）來描述面向行的模式。在大多數系統中規範模式指的是定義好的配置。非規範模式（Uncanonical Mode）指的是原始模式。POSIX相容的系統提供了幾個庫函數以支援選擇哪種模式和改變終端配置。在MINIX系統中，IOCTL系統調用支援這些函數。

鍵盤驅動程式的第一個任務是收集字元。如果每次擊鍵產生一個中斷，驅動程式可以在中斷過程中獲得字元。如果中斷被低層次的軟體變成了消息，可以把最新獲得的字元放入消息中，也可以放在記憶體的一個小緩衝中，再由消息告訴驅動程式某件事情發生。如果僅僅把消息送到等待進程，那麼就使鍵盤驅動程式還有機會忙於處理前一字元，因此後一種方法比較安全。

一旦驅動程式接收到了字元，它必須開始處理它。如果鍵盤傳過來的是鍵碼而不是應用軟體使用的字元代碼，驅動程式必須使用一個表格對其進行轉換。並不是所有的IBM“兼容機”都使用標準鍵碼，所以如果驅動程式希望支援這些機器，必須利用不同的表格進行不同的映射。一種簡單的辦法是在驅動程式中編輯一個表格以進行鍵盤提供的代碼和ASCII碼（美國資訊交換標準代碼）之間的映射，但是這對於非英語的用戶是不令人滿意的。在不同的國家，鍵盤的安排是不同的，即使對西半球的大多數人而言，ASCII碼集也是不夠的。西班牙語、葡萄牙語和法語需要英語中不使用的標點符號和重音字元。為了滿足鍵盤配置的靈活性以提供支援不同語言的需要，許多作業系統提供了可裝載的鍵位表（Keymap）或內碼表（Code Page），從而使得選擇在鍵碼和傳輸給應用程式的代碼之間的映射成為可能。這可以在系統啟動時實現，也可以在啟動後實現。

如果終端工作於規範模式（熟模式），必須保存字元直至累積到一行，這是因為用戶以後可能決定刪除其中一部分。即使終端工作在原始模式，程式可能還未請求輸入，所以為了支持預輸入也必須對字元實施緩衝（不允許使用者提前輸入的系統設計者應該被塗以焦油，用羽毛裝飾甚至強迫使用自己的系統）。

一般有兩種字元緩衝的方法。對於第一種方法，驅動程式包含一個集中的緩衝集區，每個緩衝中大約可存放10個字元。和每個終端相關的是一資料結構，在該結構中除了其他內容外，包含一個緩衝鏈指標，而緩衝鏈中的各緩衝存放從該終端上輸入的各字元。使用者鍵入的字元越多，用於輸入的緩衝就越多，掛在鏈上的緩衝也越多。當字元傳送給使用者程式時，把緩衝移走放回緩衝集區中。

另一種方法直接在終端資料結構中實施緩衝，沒有集中的緩衝集區。用戶輸入一條命令，該命令需運行一段時間（例如編譯），因此通常用戶再預輸入幾行。出於安全性的考慮，驅動程式應該為每個終端分配大約200個位元組的緩衝。在一個具有100個終端的大規模分時系統中，固定分配20K用於預輸入顯然太多了，而在集中緩衝集區中有5K一般也足夠了。另一方面，為每個終端指定一個緩衝使驅動程式比較簡單（沒有連結表管理），在具有一兩個終端的電腦上，使用者更偏愛這種方法。圖3－32顯示了這兩種方法的不同。

儘管鍵盤和顯示器在邏輯上是分立的設備，但是許多使用者已習慣於在螢幕上看見他們剛剛鍵入的字元。一些過時的終端把鍵入的內容自動顯示出來，這不僅對於輸入口令是很令人討厭的，而且極大限制了複雜編輯器和應用程式的靈活性。幸運的是，大多數的現代終端在輸入時不顯示任何內容，而由軟體來顯示輸入的內容，這個過程稱為回顯（Echoing）。

當使用者擊鍵時，程式可能正在寫螢幕，這使得回顯過程複雜化了。至少，鍵盤驅動程式要計算出新輸入字元的顯示位置，使其不被應用程式的輸出所覆蓋。

在每行80個字元的終端上，鍵入超過80個字元時，回顯也變得複雜了。依賴于應用程式，轉入下一行可能是合適的。有些驅動程式通過把一行中第80個字元以後的字元去掉來解決這個問題。

圖3－32 （a）中央緩衝集區。 （b）每個終端指定的緩衝。

另一個問題是TAB鍵的處理。大多數鍵盤都有一個TAB鍵，但是幾乎沒有終端能處理TAB鍵的輸出。驅動程式需要既考慮到應用程式的輸出，又考慮到回顯的輸出，從而計算出把游標定位到何處，也要計算回顯正確的空格字元數。

現在我們討論設備等效性問題。從邏輯上說，在每一文本行的末尾，需要一個回車鍵把游標移到第一列，然後一個換行鍵進到下一行。需用戶在每一行末尾鍵入回車和換行兩鍵是不合適的（雖然有些終端有產生這兩個字元的鍵，但應用程式僅有50％的可能需要它）。驅動程式通常把輸入的內容轉換成作業系統使用的內部標準。

如果標準格式僅僅存儲換行（MINIX格式），那麼回車應被轉換為換行。如果內部格式存儲回車和換行，那麼驅動程式應該在鍵入回車時生成一分行符號，鍵入換行時生成一回車符。無論在內部如何轉換，為了使螢幕能正確更新，終端可能要求回顯回車和換行。因為一台大型電腦可能和大量不同的終端相連，需要鍵盤驅動程式把不同的回車/換行組合轉換成內部系統標準並正確處理回顯。

一個相應的問題是回車換行的定時問題。在一些終端上，顯示一個回車或換行比顯示字元或數位的時間長。如果終端內的微處理機不得不拷貝一大塊文本來實現滾動一行，換行可能會慢一些。如果機械打印頭不得不回到打印紙的左邊，回車會慢一些。在這兩種情況下，都需要驅動程式向輸出流中插入填補字元（Filler Character）（假的空字元）或者等待足夠長的時間以使終端跟得上。延遲時間一般與終端的速度有關，例如，4800bps或更低可能不需要延遲，但在9600或更高的速度，需要一個填補字元。如果硬體支援TAB，特別是硬拷貝輸出TAB後也需要一個延遲。

當操作在規範模式時，許多輸入字元有特殊的含義。圖3－33顯示了POSIX需要的特殊字元和MINIX識別的附加字元。其預設值都是和程式使用的代碼和文本輸入不相衝突的控制字元，但如果需要，除最後兩個以外，它們都可以由tty命令設置。老版本的UNIX對這些字元的大多數使用不同的預設值。

圖3－33在規範模式下特殊處理的字元。

ERASE字元使使用者擦除剛剛輸入的字元。在MINIX中，這是後退字元（CTRL－H）。並不把這個字元插入到字元佇列，而是從佇列中移走前一字元。為了從螢幕上去掉一個字元，它應順序回顯三個字元：退格、空格和退格。如果前一字元為TAB，把它擦除需要跟蹤在TAB前游標在何處。在大多數系統中，退格僅僅擦除當前行的字元，不能擦除回車回到上一行。

當使用者在輸入的一行中的起始部分發現一錯誤，刪除整行重新輸入是很方便的。KILL字元（在MINIX中CTRL－U）刪除當前行。MINIX使刪除的行從螢幕上消失，但是在一些系統中，由於某些使用者喜歡看到從前的一行，因此其處理是在後面放一回車和換行並回顯之。所以如何處理KILL字元是一個喜好的問題。關於ERASE字元，一般不可能退回到離開當前行。當刪除很多字元時，如果使用了緩衝，那麼驅動程式是否把緩衝退還給緩衝集區可能值得，也可能不值得。

有時ERASE或KILL字元必須作為普通資料登錄，LNEXT字元起轉義字元（Escape Character）的作用。在MINIX中，其預設值為CTRL－V。作為一個例子，在老版本的MINIX中，經常使用@作為KILL，但是互連網郵件系統使用形式為linda@cs.washington.edu的位址，對於老習慣感覺比較舒服的人可能會重定義KILL為@，但是在e-mail位址中需要輸入@，這可以通過鍵入CTRL－V@來實現。CTRL－V自身可以通過鍵入CTRL－V CTRL－V來輸入。發現CTRL－V以後，驅動程式設置一個標誌，宣佈對下一個字元不進行特殊處理。LNEXT字元自身不輸入到字元序列中。

為了使使用者能使螢幕圖像停下來不致滾出視窗以外，提供了使用者凍結螢幕和以後重啟動畫面的控制碼。在MINIX中，分別是STOP（CTRL－S）和START（CTRL－Q），它們不被存儲，但用來清除和設置終端資料結構中的一個標誌，無論何時試圖輸出，檢查一下這個標誌，如果設置，不執行輸出。一般一起禁止回顯和程式輸出。

經常需要殺死一些失去控制的正在調試的程式，使用INTR（DEL）和QUIT（CTRL－\）的目的即在於此。在MINIX中，DEL引起發送一個SIGINT信號到從該終端啟動的所有進程。實現DEL很有技巧性，困難在於從驅動程式取得資訊，然後送到處理信號的系統部分，但它並沒有要求這種資訊。CTRL－\和DEL是相似的，但是發送SIGINT信號，如果該信號未被忽略或捕獲，則強制進行核心轉儲。無論鍵入哪個鍵，驅動程式應回顯一個回車換行，丟掉所有的輸入以允許重新啟動。INTR的預設值通常為CTRL－C而不是DEL，這是因為許多程式內部把DEL作為編輯的退格建。

另一特殊字元為EOF（CTRL－D），在MINIX中，即使緩衝為空，也使等待從緩衝讀的請求得到滿足，緩衝中有什麼就讀出什麼。在一行的開始鍵入CTRL－D使應用程式讀出零位元組，按慣例這被解釋為讀到了檔案結尾，使應用程式就象在輸入檔中看到了檔案結尾一樣進行操作。

一些終端驅動程式允許比我們前面討論的更加令人喜歡的行內編輯。這些終端有特殊的控制字元用於刪除一個字、向前或向後跳幾個字或字元、回到輸入行的行首或行尾等等。把所有這些功能加到終端驅動程式會使它變得很大，而且當使用工作於原始模式的螢幕編輯器時這也是一種浪費。

為了使程式控制終端參數，在標準庫中POSIX需要幾個函數，最重要的兩個函數為tcssetattr和tcgetattr。tcgetattr取回一個如圖3－34中顯示的資料結構的一份拷貝，該結構為termios，它包含用來改變特殊字元、設置模式和修改終端其他特性的所有資訊。程式可以檢查當前的設置，當需要時對這些設置進行修改，然後調用tcsetattr把這個結構寫回終端任務中。

圖3－34 termios結構。在MINIX中 tc\_flag\_t的類型為short，speed\_t的類型為 int ，cc\_t的類型為 char 。

POSIX並未規定這些需求是通過庫函數實現還是通過系統調用實現，MINIX提供了一個系統調用IOCTL，通過以下的形式調用：

ioctl（descriptor，request，argp）

它用於檢查和修改許多設備的配置。用這個調用實現tegetattr和tcsetattr。變數request指出了是讀還是寫termios結構，在後一種情況下，該請求是立即起作用還是等到當前輸出佇列中的內容全部輸出時才起作用。變數是argp指向一個在調用程式中termios結構的指標。這種在應用程式和驅動程式之間的特殊通訊方式是為了UNIX相容性而不是為了固有的美感。

順序討論一下termios結構。四個標誌提供了很大的靈活性。在c\_iflag中的各位控制處理輸入的各種方法。例如ICRNL位元使在輸入中的CR字元轉換為NL字元，這個標誌在MINIX中設置為預設值。c\_oflag存放影響輸出處理的各位，例如OPOST位允許輸出處理。MINIX也設置了該位和ONLCR位，這兩位元使輸出的NL字元轉換為CR NL序列。c\_cflag是控制標誌，MINIX的預設設置使一條線路接收8位元字元，而且如果使用者在一條線路註銷，則使數據機掛起。c\_lflag標誌是局部模式標誌域，ECHO位元允許回顯（登錄時可以關閉以提供登錄時的安全性）。最重要的位元為ICANON位元，它使終端工作於規範模式，在關閉ICANON位元情況下，還存在幾種可能的設置。如果其他的設置都是預設值，則進入常規的cbreak模式。在這種模式下，輸入的字元不必等待滿行就傳給了應用程式，但是INTR、QUIT、START和STOP還起作用。通過在標誌中對相應的位元復位，所有這些都可以禁止，其結果為常規的原始模式。

各種可以改變的特殊字元，包括那些MINIX擴充的字元，都存放在c\_cc陣列中，該陣列中也存放了兩個使用在非規範模式的參數，存儲在c\_cc[VMIN]中的數值MIN規定了READ調用讀出的最少字元數。c\_cc[VTIME]中的值TIME設置了這些調用的時間極限。MIN和TIME組合起來的使用示於圖3－35，顯示的是請求N個位元組的調用。當TIME＝0同時MIN＝1時，其行為類似於常規的原始模式。

圖3－35 MIN和TIME決定在非規範模式下當調用讀時如何返回，N是請求的位元組數。

輸出軟體

輸出比輸入簡單，但是，RS－232終端的驅動程式和存儲映射終端的驅動程式基本上是不同的。一般用於RS－232的方法是為每一終端設置緩衝，緩衝可以來源於同時作為輸入緩衝的緩衝集區，也可以是象輸入一樣指定的專用緩衝。當程式向終端寫時，輸出被首先拷貝到緩衝。同樣，回顯輸出也拷貝到緩衝。當所有的輸出拷貝到緩衝以後（或緩衝滿），輸出第一個字元，驅動程式睡眠，當中斷發生時輸出下一個字元，如此進行下去。

對於存儲映射終端，一個更加簡單的方案是可行的。需要列印的字元在某一時刻從用戶空間中取出，直接放入視頻RAM中。對於RS－232終端，每一字元通過傳輸線送至終端，對於存儲映射終端，一些字元需特殊處理，其中包括退格、回車、換行和響鈴（CTRL－G）。一個存儲映射終端的驅動程式必須在軟體中跟蹤在視頻RAM中當前位置，以便在那裡列印可列印字元，然後向前移動輸出位置。空格、回車、換行都需要相應地更新位置。

特別是當在螢幕底部行的末尾輸出換行時，螢幕必須上滾。為了分析滾動如何實現，看一下圖3－29，如果視訊控制器永遠從0XB8000開始讀RAM，滾動螢幕的唯一辦法是拷貝24×80個字元到0XB0000（每個字元需兩個位元組，這是很費時間的）。

幸運的是，在這裡，硬體提供了一些説明。大多數的視訊控制器包含一個寄存器，這個寄存器控制在視頻RAM中從何處開始取螢幕上的最頂行的位元組，通過設置該寄存器指向0XB00A0而不是0XB0000，以前的第二行移到了最頂行，這個螢幕上滾一行。驅動程式必須做的其他工作僅僅是拷貝新的最底行的內容，當視訊控制器指向了RAM的最高端後，它僅僅返回到最低位址並開始繼續取資料。

驅動程式必須處理的另一個問題是存儲映射終端的游標位置，硬體一般也用一個寄存器來提供幫助，該寄存器告訴游標去向何處。最後還有一個響鈴問題，這通過向揚聲器送一正弦波或方波來產生。揚聲器是電腦中和視頻RAM不同的部分。

值得注意的是，存儲映射終端驅動程式所面臨的許多問題（滾動、響鈴等等）RS－232終端中的微處理機也會遇到。從微處理機的觀點來看，它是帶有存儲映射顯示器系統中的的主處理機。

螢幕編輯和許多其他的複雜程式需要比僅僅把文本從底部上滾的方式更加複雜的更新螢幕的方式。為了支援這些功能，許多終端驅動程式支援大量的轉義序列。雖然許多終端驅動程式支援各別的轉義序列集，但是有一個標準使一個系統的軟體能適應另一個系統是有益的。美國國家標準委員會（ANSI）定義了一套標準的轉義序列，MINIX支援ANSI標準的轉義序列的一個子集，該子集示於圖3－36，對於許多公共操作，這已經足夠了。當驅動程式看到開始轉義序列時，它設置一個標誌並等待轉義序列的其他部分進入系統。當全部進入以後，驅動程式必須在軟體中實現轉義序列指定的操作。插入和刪除文本需要在RAM中移動大量字元。硬體除了滾動和顯示游標外不能提供任何幫助。

圖3－36 終端輸出驅動程式接受的ANSII碼轉義序列。ESC表示ASCII碼轉義字元（0X1B）。n，m和s為可選的數值參數。

3.9.3 MINIX中終端驅動程式概述

終端驅動程式包含在四個檔中（如果支援RS－232和偽終端，則為六個檔），它們構成了MINIX中最大的驅動程式。終端驅動程式既處理鍵盤，也處理顯示器，二者都很複雜，同時又包括兩個可選擇終端，因此終端驅動程式是分成幾部分來解釋的。和調度程式比較，終端驅動程式要比其大30倍，這對於大多數人而言，一定感到非常吃驚（看一看大量關於作業系統的書籍，這些書籍中關於調度程式的內容相當於把所有I/O的內容加在一起的30倍，這一定會使這些人更吃驚）。

終端驅動程式接收七種消息類型：

1 從終端讀（來自FS，它代表使用者進程）

2 向終端寫（來自FS，它代表使用者進程）

3 為IOCTL設置終端參數（來自FS，它代表使用者進程）

4 上一次時鐘滴答時發生I/O（來自時鐘中斷）

5 終止上一個請求（當信號發生時，來自檔案系統）

6 打開設備

7 關閉設備

除了不需要POSITION域外，讀寫消息和示於圖3－15中的消息具有相同的格式。對於磁片，程式必須指定它想讀那一塊，對於終端，卻沒有這項選擇。程式永遠取鍵入的下一個字元，終端不支援尋道。

使用IOCTL系統調用的POSIX函數tcgetattr和tcsetattr檢查和修改終端屬性（特性），好的程式設計實踐是使用它們和其他在include/termiios.h中的函數，讓庫函式呼叫IOCTL系統調用。有一些MINIX需要的控制操作在POSIX中沒有提供，裝入可選擇的鍵位表就是一個例子。對於這些問題，程式師必須顯式地使用IOCTL。

通過IOCTL系統調用發往驅動程式的消息包括一個功能請求代碼和一個指標。對於tcsetattr函數，執行的是一個具有TCSETS、TCSETSW或TCSETSF請求類型和一個指向類似圖3－34的termios結構指標的IOCTL調用。所有這些調用把當前的屬性集換成一個新的屬性集，它們區別在於：TCSETS請求立即起作用，TCSETSW請求直到所有的輸出被輸出後才起作用，TCSETF等待輸出完成並拋棄掉還沒有讀的輸入。tcgetattr轉換為一個具有TCGETS請求類型的IOCTL調用，並返回調用者一個填入termios結構的消息，因此，可使用該調用檢查當前設備的狀態。有一些IOCTL調用並不完全對應于POSIX定義的函數，它傳輸的是另外幾種結構的指標，例如，KIOCSMAP請求用於裝入一個新鍵位表，於是需傳輸的是一個指向keymap\_t的指標，這是一個1536位元組的結構（128鍵×6修飾符，每項16位）。圖3－43概括了標準的POSIX調用是如何轉換成IOCTL系統調用的。

終端驅動程式使用的一個主要的資料結構為tty\_table，它是一個tty結構的陣列，每個終端一個。標準PC僅有一個鍵盤和顯示器，但MINIX可支援多達8個虛擬終端，這取決於顯示卡卡的存儲空間大小。這允許使用控制台的使用者登錄多次，並且顯示輸出和鍵盤輸入可從一個使用者切換到另一個用戶。對於虛擬控制台，按ALT－F2選擇第二個用戶，ALT－F1返回第一個，也可使用ALT和方向鍵。除此之外，串列線路也支援通過RS－232和數據機相連的兩個遠端用戶，也支援使用者通過網路相連接的偽終端。驅動程式被編寫成很容易增加附加終端。在書中的源碼中，支持兩個虛擬控制台，不支援串列線和偽終端。

在tty\_table中的每個tty結構既跟蹤輸入，也跟蹤輸出。對於輸入，它存放了一個所有已經鍵入但還沒有被程式讀出字元的佇列，它也存放了請求讀但還未接收到字元的資訊，以及超時資訊，因此如果沒有鍵入字元也能請求輸入，而且可以使任務不永久阻塞。對於輸出，它存放了沒有完成的寫請求的參數，其他的域存放著各種通用變數，例如，前面討論的termios結構，該結構影響輸入輸出的許多特性。在tty結構中還有一個指向特殊類設備需要的資訊，但在tty\_table項中並不是每個設備都需要。例如，控制台驅動程式和硬體有關的部分需要在螢幕上和視頻RAM中的當前位置，以及當前顯示的屬性位元組，但對於RS－232傳輸線，則不需要這些資訊。每種類型設備的私有資料結構也與緩衝處於同一位置，該緩衝存放來自插斷服務常式的資料。慢速設備，例如鍵盤不需要象快速設備所需要的那麼大的緩衝。

終端輸入

為了更好地理解終端驅動程式如何工作，讓我們首先看一下在終端上鍵入的字元是如何從系統到需要它們的程式的。

當使用者在控制台上登錄時，系統為其創建一個shell，它用/dev/console作為標準輸入、標準輸出和標準出錯。shell啟動並試圖通過調用庫過程read讀標準輸入，這個過程發送一條包含檔描述符、緩衝位址和位元組數的消息到檔案系統。這條消息就象圖3－37（1）所示的那樣。發送消息以後，shell阻塞，等待應答（使用者進程僅僅執行SEND\_REC原語，這個原語把一個SEND和一個從發往進程的RECEIVE結合起來）。

檔案系統取消息，並找到對應檔描述符的i結點，這個i結點對應一個字元設備設備檔/dev/console，並且包含終端的主次設備號，終端的主設備號為4，對控制台，次設備號為0。

檔案系統在設備映照圖dmap中查找終端任務號。如圖3－37（2）所示，向終端任務發一條消息。通常使用者至此還沒有鍵入，因此終端驅動程式不能滿足這個請求。它立即發回一個應答使檔案系統解除阻塞，報告不能得到任何字元，這正如圖3－17（3）所示。

圖3－37 當還未輸入字元時來自終端的讀請求。FS是檔案系統，TTY是終端任務。當輸入字元時，終端的中斷處理常式把輸入字元排成一個佇列，但是在這裡由時鐘中斷處理常式喚醒TTY。

檔案系統在tty\_table的控制台資料結構中記錄一個進程等待終端輸入，然後去處理下一個請求。使用者shell繼續阻塞，直至請求字元到達之後。

當字元最後在鍵盤上鍵入以後，它引起兩個中斷，其中之一是當按下鍵的時候，另一個是當釋放鍵的時候。這個規則對於象CTRL和SHIFT這樣的修飾鍵也是適用的，這些鍵本身並不傳輸任何資料，但每個鍵仍要引起兩個中斷。鍵盤中斷為IRQ1，在彙編代碼文件mpx386.s中的\_hwint01啟動kbd\_hw\_int（13123行），然後調用scan\_keyboard（13432行）從鍵盤硬體取得鍵碼。如果鍵碼是普通字元，並且中斷是由按下鍵而產生的，那麼把它放入輸入佇列ibuf中，但是如果中斷是由於釋放鍵而引起，則忽略它。對於兩種類型中斷，CTRL和SHIFT這類修飾鍵的代碼也放在佇列中，但可以通過一個僅僅當鍵釋放時設置的位區分開。注意在這點上，接收和存儲在ibuf中的代碼並不是ASCII碼，它們僅僅是IBM鍵盤產生的掃描碼。然後kbd\_hw\_int設置一個標誌tty\_events（tty\_table中的有關鍵盤的部分），調用force\_timeout，最後返回。

不象其他的插斷服務常式，kbd\_hw\_int不發送消息喚醒終端任務。在圖中虛線（4）表示調用force\_timeout，這些不是消息。它們設置對插斷服務常式通用的位址空間中的變數tty\_timeout。在下一次時鐘中斷，clock\_handler發現tty\_timeout指明現在是調用tty\_wakeup的時刻，於是它向終端任務發送一條消息（5）。注意雖然tty\_wakeup的源碼在tty.c中，它卻根據時鐘中斷來運行，因此我們說時鐘中斷向終端任務發消息。如果輸入到達得很快，大量的字元代碼按這種方式排成一個佇列，這就是為何在圖中顯示了多次調用force\_timeout。

一旦接收到了喚醒消息，終端任務檢查每一個終端設備的tty\_events標誌。對於每一設置了該標誌的終端，調用handle\_events（12256行）。tty\_events標誌指出了各種活動（雖然大多數為輸入），因此，handle\_events永遠調用和設備有關的用於讀寫的函數。對於從鍵盤輸入，將調用kb\_read（13165行），該函數跟蹤指示CTRL、SHIFT和ALT鍵的按下和釋放的鍵盤代碼，把鍵盤代碼轉換為ASCII碼。kb\_read轉而調用in\_process（12367行），該函數處理ASCII碼，它要考慮特殊字元和可能設置的不同標誌，包括規範模式是否有效，雖然一些代碼，例如BACK SPACE有其他的功能，但是其一般處理結果為向在tty\_table中的控制台輸入佇列中加入字元。通常in\_process還要啟動向顯示器回顯ASCII碼。

當輸入了足夠多的字元時，中斷服務程式調用組合語言常式phys\_copy把資料拷貝到shell請求的位址中去。這個操作也不是通過消息傳遞實現的，正因為如此，在圖3－37中通過虛線（6）表示。這樣的線顯示了若干條，這是因為在用戶請求完全滿足以前，可能有多個這樣的操作。當這個操作最後完成的時候，終端驅動程式向檔案系統發一條消息，告訴它工作已完成（7）。檔案系統回應這條消息，向shell發一條消息來使其接觸阻塞。

怎樣才算輸入了足夠多的字元呢？這取決於終端模式。在規範模式，當收到回車、行結束或檔案結尾代碼時，請求就完成了。為了對輸入進行合適的處理，輸入的一行不能超過輸入佇列的大小。在非規範模式，讀可以請求大量的字元，in\_process可能不得不在返回檔案系統一條消息告訴它操作完成以前傳輸多次字元。

注意終端驅動程式把實際的字元直接從它自己的位址空間中拷貝到shell空間中，它並不首先通過檔案系統。對於塊I/O，資料確實通過了檔案系統，以維護一個最近使用資料塊的緩衝。如果請求塊正好在緩衝中，請求可以直接由檔案系統滿足，不需做任何磁片I/O。

對於終端I/O，緩衝毫無意義，而且檔案系統到磁片驅動程式的請求總可以在幾百個毫秒內得到滿足，因此使檔案系統等待沒有什麼損失。終端I/O可能需幾個小時才能完成，甚至永遠不能完成（在規範模式，可能等待的時間更長，這取決於MIN和TIME的設置），因此，使檔案系統阻塞直至終端請求滿足是不合適的。

此後，使用者可能預鍵入了一些字元，這些字元在請求以前就已準備好，在這種情況下，事件1、2、6、7和8在讀請求以後相繼順序發生，3則不發生。

如果當時鐘中斷時，終端任務碰巧正在運行，因為它還沒有等待消息，因此不能向它發消息。然而，為了在終端任務忙時能平穩地輸入輸出，要檢查幾次所有終端設備的tty\_events標誌，例如，緊接著消息的處理和回應立即進行一次這種檢查，於是沒有來自時鐘的喚醒的説明，字元也能加入到控制台佇列中。如果在終端驅動程式完成它正在做的工作以前，發生了兩次或多次時鐘中斷，所有的字元都存儲在ibuf中，並重複設置tty\_flags，最後終端任務得到一條消息，其他的消息就丟失了。但是，因為所有的字元都安全地存儲在緩衝中，輸入並未丟失。甚至有可能在終端任務接收到消息時，輸入已經完成，應答也發往了使用者進程。

在這種設計中，存在一個當中斷常式向忙進程發消息時，如果是在一個無緩衝的消息系統中如何處理的問題。對於大多數的設備，例如磁片，中斷是由於響應驅動程式發出的命令而發生的，因此在任何時刻，僅可能等待一個中斷。其自身產生中斷的設備是時鐘和終端（當允許時，網路也產生）。通過對沒有處理的滴答進行計數來處理時鐘，因此如果時鐘任務沒有從時鐘中斷得到消息，以後的處理會對其補償。對終端的處理方法是使中斷常式在緩衝中積累字元同時增加標誌，指示已收到了字元的方法處理終端，如果終端任務正在運行，在其睡眠以前檢查這些標誌，如果有其他工作要做就推遲睡眠。

終端任務並不被終端中斷直接喚醒，因為那樣做負擔太重。每一次終端中斷之後的下一次滴答時鐘向終端任務發一個中斷。以每分鐘100個字的速度，打字員每秒的輸入不超過10個字元，即使對於快速的打字員，對於在鍵盤上輸入的每個字元，也會向終端任務發一條消息，雖然有些消息可能丟失。如果在緩衝中沒有空間時仍有字元到來，就把該字元丟棄。但是經驗顯示，對於鍵盤，32個字元的緩衝就足夠了。對於其他輸入裝置，可能具有更高的資料速率 － 一個連到28800 bps數據機的串列口可能比打字員快1000倍或更多。在這樣的速度，在兩個時鐘滴答之間，數據機大約收到48個字元，但是考慮到在數據機鏈路上的資料壓縮，連到數據機上的串列口必須能處理至少兩倍的字元。對於串列線，MINIX提供1024個字元的緩衝。

我們有一些遺憾。終端任務沒有完全按通用設計原理實現，而是採取了一種折衷方案，但我們使用的方法沒有過多地增加軟體複雜性，性能也無損失。另一種明顯的方法是拋棄會合原則，使系統緩衝所有沒有被等待的消息，但這將更複雜，速度也更慢。

實際系統的設計者必須經常面對兩種技術間的折衷，一種是採用通用的方案，這種方案在所有時候都很完美，但有時速度較慢；另一種是採用簡單技術，一般速度很快，但是在一兩種情況下，需要一些技巧以使其正常工作。在給定的情況下，哪種方式最好，經驗是唯一真正的指導。Lampson （1984）和brooks（1975）總結了大量設計作業系統的經驗，雖然有點過時，但是這些內容還是頗具經典性的。

在結束終端輸入概述時，總結一下當終端任務被第一個讀請求啟動和接收到鍵盤輸入以後被再次啟動時發生的事件（參見圖3－38）。在第一種情況下，當消息進入終端任務請求從鍵盤讀字元時，主過程tty\_task（11817行）調用do\_read（11891行）來處理這個請求，如果沒有足夠多的緩衝字元，do\_read把調用參數存入tty\_table的鍵盤項中。

然後它調用in\_transfer（12303行）取得任何已在等待的輸入，再調用handle\_events（12256行），該函數接著調用kb\_read（13165行），並再次調用in\_transfer試圖再取得幾個字元。kb\_read調用了幾個在圖3－38中沒有顯示的常式來完成它的工作，其結果是把立即可得到的都拷貝給使用者，如果什麼也得不到，就什麼也不拷貝。如果in\_transfer或handle\_events完成了讀操作，當所有字元已被傳輸時向檔案系統發一消息，因此檔案系統可以使調用者解除阻塞。如果讀還未完成（沒有字元或無足夠多的字元），do\_read報告檔案系統，告訴它或者掛起初始調用者，或者對對非阻塞請求，取消該讀請求。

圖3－38的右邊總結了鍵盤中斷以後終端任務被喚醒時發生的事件，當鍵入一個字元時，中斷過程kb\_hw\_int把接收到的字元碼放入鍵盤緩衝，設置一個標誌指示控制台設備經歷了一個事件，然後安排在下一次時鐘滴答時發生超時。時鐘任務向終端任務發一條消息，告訴它事件發生了。一旦接收到這條消息，tty\_task檢查所有設備的事件標誌，為每一個標誌已設置的設備調用handle\_events。如果是鍵盤，就象接收到初始讀請求一樣，handle\_events調用kb\_read和in\_transfer。在FS處接到第一條消息後，圖右側所示的事件可能發生若干次，直到有足夠的字元能滿足do\_read所接收的請求為止。如果FS在第一個請求結束之前試圖啟動同一設備的另一個請求以讀取更多字元，則將返回一個錯誤。當然，各個設備都是獨立的；在遠端終端機上的用戶的讀請求和控制台上的用戶的讀請求是分開進行處理的。

圖3-38 終端驅動程式的輸入處理。樹的左分枝表示處理一個讀字元請求。右分枝表示向驅動程式發送了一條“字元已被輸入”消息。

圖3-38中未給出的由kb\_read調用的函數包括：把硬體產生的鍵碼（掃描碼）轉換為ASCII碼的map\_key，和跟蹤修飾鍵（如SHIFT鍵）狀態的make\_break，以及處理各種複雜情況如使用者試圖用退格覆蓋錯誤輸入，其他特殊字元，和不同輸入模式下的可用選項等情況的in\_process。in\_process也調用echo（第12531行），所以鍵入的字元也會顯示在螢幕上。

終端輸出

一般來說，控制台輸出比終端輸入簡單，這是因為作業系統擁有控制權，不需要考慮意外的輸出請求。另外，由於MINIX控制台是記憶體映象顯示器，所以向控制台輸出就顯得特別簡單。輸出的基本操作是把資料從一個記憶體區拷貝到另一個記憶體區，不需要任何中斷。另一方面，顯示管理的所有細節工作，包括轉義序列的處理，都必須由驅動軟體處理。和上一節講述鍵盤輸入時一樣，我們將跟蹤向控制台發送字元時涉及的每個步驟。我們假設在例子中正被寫入的是當前活動顯示器；如果考慮虛擬控制台設備，情況將有一點複雜。我們將在以後討論這種情況。

一個進程在試圖顯示時一般要調用printf。printf調用WRITE向檔案系統發送一條消息。該消息包含一個指向待顯示字元序列的指標（不是字元序列本身）。接著檔案系統向終端驅動程式發送一條消息，終端驅動程式取出字元並拷貝到視頻RAM中。圖3-39繪出了輸出時涉及到的主要過程。

圖 3－39 終端輸出中使用的主要過程。虛線表示cons\_write把字元直接拷貝到ramqueue。

當終端任務收到一條請求在螢幕上輸出的消息時，它調用do\_write（第11964行）來存儲tty\_table中控制台的tty結構參數。然後handle\_events（只要tty\_events標誌被置位元，就調用這個函數）被調用。每次調用時這個函數都會調用由它的參數指定的設備的輸入和輸出常式。在討論控制台顯示時，這意味著任何處於等候狀態的鍵盤輸入將被首先處理。在有輸入等待時，則待回顯字元被加到任何正在等待輸出的字元後面。然後執行一個對cons\_write的調用（第13729行）。cons\_write是一個記憶體映象顯示過程，它使用phys\_copy把字元塊從使用者進程拷貝到局部緩衝區。由於局部緩衝區只有64位元組，這個過程和以下各步驟可能要重複若干次。當局部緩衝區滿時，每個8位元位元組被傳送到另一個緩衝區ramqueue中。ramqueue是一個16位元的字陣列，交替間隔地用螢幕屬性位元組的當前值填充。螢幕屬性位元組決定了字元的前景色、背景色以及其他屬性。如果可能，字元就被直接傳送到ramqueue中；但某些字元如控制字元或轉義序列中的字元，需要經過特殊處理；當一個字元的螢幕位置超出了螢幕的實際寬度，或者ramqueue已滿時，也需要進行特殊處理。在這些情況下就要調用out\_char（第13809行）來傳送字元並執行各種相應的附加動作。例如，在定位螢幕的最後一行時，如果接收到一個分行符號，那麼scroll\_screen（第13869行）將被調用；而對於一個轉義序列，系統將調用parse\_escape來處理這些字元。一般地，out\_char調用flush（第13951行）把ramqueue中的內容拷貝到視頻顯示記憶體，其中使用了一個名為mem\_vid\_copy的組合語言常式。在最後一個輸出字元被送入ramqueue中後也要調用flush以確保所有要輸出的字元都被顯示出來。Flush的最終執行結果是使6845視訊控制器在正確的位置上顯示游標。

從邏輯上講，從使用者進程取出的字元可以在每次迴圈反覆運算中逐個寫入視頻RAM。然而，在Pentium類處理器的保護存儲環境中，把在ramqueue中等待的字元用mem\_vid\_copy成塊地拷入視頻RAM的效率要高得多。有趣的是，這項技術卻是在早期運行在沒有存儲保護的處理器上的MINIX版本中引入的。早期的mem\_vid\_copy必須考慮定時問題——老式的視訊卡只允許在CRT電子束垂直回掃螢幕為空時寫視訊記憶體，以避免螢幕上出現雜亂無章的“雪花”現象。由於性能犧牲過大，MINIX現在已不再支援這些過時的設備。不過，成塊拷貝ramqueue的思想卻使現代的MINIX在其他方面得到了益處。

控制台可以使用的視頻RAM區由console結構中的c\_start和c\_limit域限定。游標的當前位置存放在c\_column和c\_row域中。座標原點（0，0）位於螢幕的左上角，硬體由這裡開始填充螢幕。每次視頻掃描從c\_org中指定的位址開始，連續掃描80×25個字元（4000個位元組）。也就是說，6845晶片從視頻RAM的偏移c\_org處取出一個字顯示在螢幕左上角（0，0）處，由屬性位元組控制字元的顏色和閃爍等；然後6845取出下一個字顯示在（1，0）處。這個過程一直持續到到達（79，0）處，然後從螢幕的第二行（0，1）處重新開始。

電腦剛啟動時，螢幕被清空，寫入視頻RAM的輸出字元從視頻RAM區的偏移c\_start處開始存放；c\_org被賦予和c\_start相同的初值。這樣第一行就顯示在螢幕的頂部。當第一行滿或out\_char檢測到一個分行符號，需要開始一個新行時，輸出被寫到偏移量等於c\_start加80處。當25行都被寫滿時，就需要進行卷屏操作。有些程式也需要進行向下卷屏操作。例如一個文字編輯器在游標已位於最頂部並仍然向上運動時，就需要把整個文字區塊向下移動。

卷屏的方式有兩種。在軟體卷屏方式下，顯示在（0，0）位置上的字元永遠被寫入視訊記憶體的第一個位置，c\_start指向相對位置為0的字的位置。通過賦予c\_org和c\_start相同的值使得視頻控制晶片從這個位置開始顯示。當螢幕需要捲動時，視頻RAM中相對位置80處的字，也就是第二行的第一個字元，被拷入相對位置0處；相對位置81處的字被拷入相對位置1處；等等。而掃描順序沒有改變，仍然把視訊記憶體位置0處的資料顯示在螢幕位置（0，0）處。這樣螢幕上就產生了向上捲動一行的效果。這種方式下CPU的開銷是需要移動80×24=1920個字元。在硬體卷屏方式下，資料本身並不在記憶體中移動，而是通過改變視訊控制器的起始掃描位置使它從視頻RAM中的不同的位置開始顯示，比如可以設置為從第80個字開始顯示。設置的過程是把c\_org的內容加上80，把所得結果保存起來，並把這個值寫入視頻晶片中相應的寄存器。這種方式要求要麼控制器有足夠的智慧在視頻RAM區中捲動，在到達底端（由c\_limit指定的位址）時能回卷到視頻RAM的起始位址（由c\_start指定）讀取資料；要麼擁有比存儲正好一屏內容所必需的80×2000個字更多的視頻RAM。老式的顯示介面卡顯示記憶體較少，但控制器能夠回捲進行硬體卷屏；較新的介面卡一般有比一屏文本的容量大得多的顯示記憶體，但控制器不能回卷。這樣，一塊擁有32768位元組顯示記憶體的介面卡就可以存儲204行每行160位元組的資料，並且在遇到不能回卷的問題前可以執行179次硬體卷屏。但是最終仍然需要一次記憶體拷貝操作把最後的24行的資料移到視頻記憶體的位置0處。不管採用那種方法，都要向視頻RAM中拷入一空行以保證在螢幕最底部的新行是空行。

如果配置了虛擬控制台，那麼一塊介面卡上可用的記憶體將被各個控制台平均分配，每個控制台設備使用的範圍由各自的c\_start和c\_limit域設定。配置虛擬控制台將對卷屏產生影響。對於擁有足夠的記憶體支援虛擬控制台的介面卡，儘管從概念上講仍然可以使用硬體卷屏，但實際上使用更多的是軟體卷屏。每個控制台可用的記憶體越少，軟體卷屏就使用得越頻繁。如果配置了最大數目的虛擬控制台，就達到了極限狀態。這時所有的卷屏操作都由軟體完成。

相對於視頻RAM起始位址的游標位置可以由c\_column和c\_row計算出來。但單獨為游標位置設置一個c\_cur域會更快些。當需要顯示一個字元時，就把它寫入視頻RAM的c\_cur處，然後更新c\_cur和c\_column。圖3-40總結了console結構中影響當前位置和起始顯示原點的域。

圖 3－40 console結構中與當前螢幕位置相關的域。

對影響游標位置的字元（如分行符號，後退字元等）進行處理時需要調整c\_column，c\_row 和c\_cur的值。這些工作由flush在結束前執行一個名為set\_6845的調用來完成。這個調用將設置視訊控制器中相應的寄存器。

終端驅動程式支援轉義序列使得編輯器和其他互動式程式可以用靈活的方式來更新螢幕。所支持的序列是ANSI標準的一個子集，這個子集應該是完備的，使得為其他硬體和作業系統編寫的程式能方便地移植到MINIX上。轉義序列分為兩類：一類不包含可變參數，另一類則可能包含參數。第一類中MINIX支持的唯一的代表是ESC M，它反向搜索螢幕，並且把游標上移一行；如果游標已經位於第一行，就把螢幕下卷一行。另一類轉義序列可以包含一個或兩個數值參數。這一類中所有的序列都以ESC [開頭，其中“[”是控制序列引入符。圖3-36給出了ANSI標準定義的可由MINIX識別的轉義序列表。

解析轉義序列並不簡單。在MINIX中有效的轉義序列可以只有兩個字元，如ESC M；而一個可以接受二個兩位數的數值參數的序列則可以有8個字元長，如ESC [20;60H，它使得游標移到第20行，第60列。在一個接受一個參數的序列中，參數可能被省略；而在接受兩個參數的序列中，可能會省略一個參數，或者兩個都省略掉。當有一個參數被省略或超出了有效範圍時，就用預設值替換。預設值等於最低有效值。

以下是構造一個把游標移到螢幕左上角的轉義序列的幾種方法：

1. ESC [H是可以接受的，因為在沒有輸入參數時，就用最低有效值代替。

2. ESC [1;1H正確地把游標移動到第1行，第1列處。

3. ESC[1;H和ESC [;1H都省略了一個參數，和第一個例子中一樣，用默認 值為1替換。

4. ESC [0;0H也具有同樣的作用，因為每個參數都小於最小有效值，用最小值代替。

這裡給出這些例子並不是推薦任意地使用無效參數，而是想說明解析這些序列的代碼並不簡單。

MINIX使用了一個有限狀態自動機解析轉義序列。console結構中的c\_esc\_state變數在正常狀態下值為0。當out\_char檢測到一個ESC字元時，c\_esc\_state的值被置為1，後面的字元由parse\_escape（第13968行）進行處理。如果後繼字元是控制序列引入符，就進入狀態2；否則認為序列已結束，調用do\_escape（第14045行）。在狀態2，只要後繼字元是數值字元，就把參數計算過程中上一次的參數值（初始值為0）乘以10，加上當前字元的數值作為當前參數值。參數值保存在一個陣列中，如果檢測到一個分號，處理過程就移到陣列的下一個單元中（MINIX中的陣列只有兩個元素，但原理是一樣的）。如果遇到一個不是分號的非數值字元，就判斷序列已結束，接著同樣調用do\_escape。在do\_escape的入口處，根據當前字元決定採取什麼動作和如何解釋參數，是使用預設值還是字元流中輸入的參數。圖3-48詳細地繪出了整個過程。

可裝載的鍵位映射表

IBM PC鍵盤並不直接產生ASCII碼。從標準PC鍵盤的左上角開始，每個鍵依次用一個數來標識——“ESC”鍵為1，“1”鍵為2，依次類推。這樣每個鍵都被賦予一個數，包括修飾鍵，如左SHIFT鍵為42，右SHIFT鍵為54。當按下一個鍵時，MINIX把接受到的對應的數作為掃描碼。當鬆開一個鍵時也產生一個掃描碼，但鬆開時產生的掃描碼最高位元被置位（即該鍵對應的數加上128）。這樣就可以分辨出一個鍵被壓下和鬆開。通過檢測修飾鍵是否被壓下並且仍未鬆開可以產生大量的複合鍵。在一般的場合，兩鍵的組合對於雙手打字來說最容易操作，如SHIFT-A或CTRL-D。但在某些特殊場合也使用三鍵組合（或更多）。比如CTRL-SHIFT-A或眾所周知的PC使用者用來重定並重新開機系統的CTRL-ALT-DEL複合鍵。

PC鍵盤十分複雜，為使用者提供了很大的靈活性。一個標準鍵盤定義了47個普通字元鍵（26個字母，10個數位，11個標點）。如果我們願意使用類似CTRL-ALT-SHIFT這樣的三鍵組合，那麼我們可以支援有376（8×47）個成員的字元集。但這是在不加任何限制的情況下，現在我們假定不區分左右修飾鍵，也不使用數位鍵盤和功能鍵。實際上，我們並不只限於使用CTRL，ALT和SHIFT作為修飾鍵；如果需要編寫一個支援這種系統的驅動程式，我們可以從普通鍵集合中去掉一些鍵，把它們作為修飾鍵。

支援這種鍵盤的作業系統根據按下的鍵和有效的修飾鍵，用鍵位映射表（keymap）決定把什麼字元碼傳給程式。MINIX的鍵位映射表可以看作一個128行，8列的陣列，行代表可能的掃描碼值（選擇這個行數是為了容納日文鍵盤；美國和歐洲鍵盤沒有這麼多鍵），各列分別代表無修飾鍵，SHIFT鍵，Control加SHIFT鍵，左ALT鍵，右ALT鍵，和ALT與SHIFT鍵的組合。用這種方式可以產生720個（（128-6）×6）字元編碼，能定義一個完備的鍵盤。這就要求表中的每個表目都是16位長。對於美國鍵盤，ALT列和ALT2列是一樣的。在其他語言的鍵盤上，ALT2也被稱為ALTGR，許多鍵位映射表使用這個鍵作為修飾鍵支援3符號的鍵。

在編譯MINIX內核時包含了一個標準鍵位映射表（由keyboard.c中的行

#include keymaps/us-std.src

決定），但也可以用

ioctl(0, KIOCSMAP, keymap)

把其他不同的鍵位映射表裝入到內核位址keymap處。一個完整的鍵位映射表佔用1536個位元組（128×6×2）。其他附加的映射表以壓縮格式存儲。一個程式可以調用genmap來生成一張新的壓縮的映射表。在編譯時，genmap為某個特定的映射表包含keymap.src代碼，這樣這張映射表就被編譯到genmap中。正常情況下，genmap在編譯之後立即執行，把生成的映射表以壓縮格式輸出到一個檔中；然後二進位的genmap被刪除。命令loadkeys讀入一個壓縮的映射表，把它在記憶體中展開，然後調用IOCTL把映射表傳送到內核存儲區。MINIX可以在啟動時自動執行loadkeys，也可以由用戶在任何時刻調用它。

圖 3－41 一個鍵位映射表原始檔案中的若干表目。

一個映射表的原始程式碼中定義了一個很大的已初始化的陣列。為了節省篇幅，這裡不能用源碼形式列出一個映射表檔。圖3-41用表格形式給出了src/kernel/keymap/us-std.src中若干行的內容，說明了映射表的若干方面。IBM PC鍵盤上沒有鍵產生值為0的掃描碼。掃描碼為1，即ESC鍵對應的表目說明壓下SHIFT和CTRL鍵時返回值不變，但ALT和ESC同時壓下時將返回一個不同的值。編譯到表中各行裡的值是由include/minix/keymap.h中定義的宏決定的：

#define C(c) ((c)&0x1F) /\*Map to control code\*/

#define A(c) ((c) | 0x80) /\*Set eight bit(ALT)\*/

#define CA(c) A(C(c)) /\*CTRL\_ALT\*/

#define L(c) ((c) | HASCAPS) /\*Add "Caps Lock has effect" attribute\*/

前三個巨集對被引用字元的掃描碼中的位元進行操作，把生成的所需的掃描碼返回給應用程式。最後一個巨集在16位元值的高位元組中設置HASCAPS位元。這個標誌位元指示需要檢測大寫鎖定變數，並且掃描碼在返回前可能需要修改。在圖中，掃描碼為2，13和16的表目說明了典型的數位，標點和字母鍵是如何被處理的。對掃描碼28可以觀察到一個不同的特徵——ENTER鍵一般產生掃描碼CR（0x0D），這裡用C(`M`)表示。由於在UNIX檔中新行符的編碼是LF（0x0A），並且有時需要直接輸入新行符，所以這個映射表提供了一個CTRL-ENTER組合來產生這個編碼，C(`J`)。

掃描碼29對應一個修飾鍵碼，並且無論是否還壓下了其他什麼鍵它都應該能被識別出來，因此總是返回CTRL。按下功能鍵不返回普通的ASCII碼值。掃描碼59對應的行中用符號（在include/minix/keymap.h中定義）給出了F1鍵和其他修飾鍵組合時的返回值。這些值是F1: 0x0110，SF1: 0x1010，AF1: 0x810，ASF1: 0x0C10和CF1: 0x0210。表中的最後一個表目對應掃描碼127，在映射表陣列末尾附近的表目中十分典型。絕大多數在歐洲和美洲使用的鍵盤沒有那麼多的鍵對應所有可能的掃描碼，表中的這些表目都用0填充。

可裝載字體

早期的PC只能用存儲在ROM中的字元點陣在螢幕上顯示字元。現代PC系統中使用的顯示器的視頻顯示介面卡上則提供了可以裝入定制的字元發生器（character generator）的RAM。MINIX提供了一個IOCTL操作

ioctl(0, TIOCSFON, font)

來支援定制的可裝入字元發生器。MINIX支援80行×25列的顯示模式，字體檔包含4096個位元組。每個位元組代表一行8個圖元，如果某一位元為1，對應的圖元就被點亮，映射一個字元需要16個這樣的行，即16個位元組。不過視頻介面卡中使用了32個位元組來映射一個字元，用來在MINIX目前還不支援的模式中提供更高的解析度。MINIX提供了命令loadfont把這些字體檔轉換到由IOCTL調用引用的8192位元組的font結構中。和鍵位映射表一樣，字體可以在啟動時載入，也可以在正常操作中的任何時候載入。不過，任何視訊卡都有一個存儲在ROM中的標準字體作為可使用的預設字體。MINIX並不需要把一個字體編譯到它本身裡，內核中唯一需要的對字體的支援是進行TIOCSFON IOCTL操作的代碼。

3.9.4 設備無關終端驅動程式的實現

在這一節中我們將要仔細地研究終端驅動程式的原始程式碼。在研究塊設備時，我們已經看到支援幾個不同設備的多個任務可以共用一套共同的基本軟體。終端設備的情況也是類似的，不同的是這裡一個終端任務要支援幾個不同類型的終端設備。現在我們從設備無關代碼開始。後面幾節我們再研究鍵盤和存儲映象控制台顯示的設備相關代碼。

終端任務資料結構

文件tty.h中包含了在實現終端驅動程式的C檔中使用的定義。這個檔中定義的絕大多數變數都具有首碼tty\_。在glo.h中還說明了一個EXTERN變數tty\_timeout，由時鐘和終端中斷處理常式使用。

tty.h中定義的標誌O\_NOCTTY和O\_NONBLOCK在檔include/fcntl.h中已經定義過了。在這裡再重寫一次是為了避免需要再包含一個檔。devfun\_t和devfunarg\_t類型用來定義指向函數的指標，以提供和我們在磁片驅動程式代碼的主迴圈中看到的相類似的間接調用機制。

tty.h中最重要的部分是tty結構的定義（第11614行到11668行）。每個終端設備都有一個這樣的結構與之對應（控制台顯示和鍵盤看作一個單一的終端）。tty結構中的第一個變數tty\_events被用作一個標誌，在某個中斷引起變化需要終端任務來處理設備時被置位元。在設置這個標誌時，系統還對全域變數tty\_timeout進行操作以通知時鐘中斷處理常式在下一個時鐘滴答喚醒終端任務。

tty結構中的其餘部分分別按照處理輸入、輸出、狀態和關於未完成操作的資訊等功能組織在一起。在輸入部分，tty\_inhead和tty\_intail定義了緩衝接收到的字元的佇列。Tty\_incount對佇列中的字元個數計數，tty\_eotct對字元的行數計數，這將在下面解釋。除了用來建立間接調用指標的終端初始化常式以外，所有針對特定設備的調用都是間接調用。tty\_devread和tty\_icancel域中保存了指向執行讀和輸入取消操作的設備指定代碼的指標。tty\_min用來和tty\_eotct比較，當後者與前者一致時，就認為一個讀操作結束。在規範輸入模式中，tty\_min被置為1，用tty\_eotct對輸入的行數計數。在非規範輸入模式中，用tty\_eotct對字元計數，tty\_min則由termios結構中的MIN域決定。這樣，根據所處的模式，通過比較兩個變數可以判斷出一行是否準備好或者何時達到最小字元計數。

Tty\_time保存計時器的值，這個值決定了終端任務應該何時被時鐘中斷處理常式喚醒，tty\_timenext是用來把活動的tty\_time域連結成一個鏈表的指標。只要設置了計時器，這個鏈表就被重新排序，因此每次時鐘中斷處理常式只需要檢查表頭即可。MINIX可以支持許多遠端終端機，在任意時刻可能只對幾個終端設置了計時器。比起檢查tty\_table中的每個表目，使用活動計時器鏈表可以使時鐘處理常式的工作簡單一些。

由於輸出排隊是由設備指定代碼處理的，所以tty的輸出部分沒有變數說明，全部由指向寫、回顯、送中斷信號和取消輸出等設備指定函數的指標組成。在狀態部分，標誌tty\_reprint、tty\_escaped和tty\_inhibited說明最後一個字元具有特殊含義；例如，當遇到一個CTRL-V（LNEXT）字元時，tty\_escaped被置為1指示下一個字元的任何特殊含義應被忽略。

結構中的下一部分保存了DEV\_READ，DEV\_WRITE和DEV\_IOCTL操作的進度資料。每個操作中包含了兩個進程。管理系統調用的伺服器（正常時為FS）在tty\_incaller（第11644行）中指定。伺服器為另一個需要進行I/O操作的進程調用tty任務，這個客戶進程在tty\_inproc（第11645行）中指定。如圖3-37所示，在執行一個READ時，字元直接由終端任務傳送到初始調用者的記憶體空間中的緩衝區。緩衝區由tty\_inproc和tty\_in\_vir定位。接下來的兩個變數tty\_inleft和tty\_incum對仍須傳送和已傳送的字元計數。一個WRITE系統調用也需要類似的變數集。對於IOCTL操作可能有在請求進程和任務之間的直接資料傳送，因此需要定義一個虛擬位址，但不需要標誌操作進度的變數。一個IOCTL請求可能被延遲，比如直到當前輸出完成；但在時間合適時，請求一次操作即可完成。最後，tty結構中還包含了一些不在其他集中的變數，包括指向在設備級處理DEV\_IOCTL和DEV\_CLOSE的函數的指標，一個POSIX格式的termios結構，一個支援面向視窗螢幕顯示的winsize結構。結構的最後部分提供了存儲輸入佇列本身的陣列tty\_inbuf。注意這是一個u16\_t陣列，而不是8位元char型字元陣列。雖然應用程式和設備使用8位元編碼的字元，C語言卻需要輸入函數getchar能使用更大的資料類型以便能返回在所有的256個可能的值之外的符號值EOF。

tty\_table，即tty結構的陣列，是用EXTERN巨集說明的（第11670行）。每個被include/minix/config.h中的定義NR\_CONS，NR\_RS\_LINES和NR\_PTYS使能的終端都有一個元素與之對應。在本書討論的配置中，有兩個控制台被使能，但MINIX可以被重新編譯以增加到兩條串列線和64個偽終端。

tty.h中還有一個EXTERN定義。tty\_timelist（第11690行）是由計時器使用的保存tty\_time域鏈表頭的指針。許多文件包含了tty.h標頭檔，tty\_table和tty\_timelist的存儲空間在編譯table.h時分配，這與glo.h中定義的EXTERN變數是類似的。

在tty.h的最後定義了兩個宏buflen和bufend。終端任務代碼經常使用這兩個巨集把資料拷入和拷出緩衝區。

設備無關終端驅動程式

主要的終端任務和設備無關的支援函數都在tty.c中定義。由於任務支援許多不同的設備，所以在一個特定的調用中需要使用次設備號來辨別那一種設備是被支援的，這些都定義在11760行到11764行中。接下來的是一些巨集定義。如果一個終端沒有初始化，那麼指向該設備的設備指定函數的指標將被C編譯器置為0。這樣就可以定義一個tty\_active宏（第11774行），在發現一個空指標時返回FALSE。當然，如果設備的初始化代碼的一部分工作是初始化指標使間接存取成為可能，那麼它本身是不能被間接存取的。11777行到11783行定義的條件宏使得對RS-232或偽終端設備的初始化調用在沒有配置這些設備時都同樣地調用一個空函數。在這個部分do\_pty也可以類似地被禁止。這樣，如果不需要某些設備，就可以把這些設備對應的代碼完全省略掉。

由於每個終端可配置的參數很多，並且在一個網路系統中可能有許多終端，所以第11803到11810行說明了一個termios\_defaults結構，並用預設值初始化。當需要初始化或重新初始化一個終端時，就向該終端的tty\_table中拷貝一個這樣的結構。圖3-33中給出了特殊字元的預設值。圖3-42給出了各種標誌的預設值。接下來的行類似地說明了winsize\_defaults結構。它應由C編譯器全部初始化為0。這是一個合適的默認動作，因為它表示“視窗大小未知，使用/etc/termcap。”

圖 3－42 預設的termios標誌值。

終端任務的入口點為tty\_task（第11817行）。在進入主迴圈之前，對每個已配置的終端調用一次tty\_init（在第11826行的迴圈中），然後顯示MINIX啟動資訊（第11829行到11831行）。雖然從源碼中看到的是對printf的調用，但在編譯時使用了把對printf庫常式的調用轉化為對printk的調用的宏。printk使用了控制台驅動程式中的一個叫作putk的常式，所以沒有涉及到FS。這條消息只發往主控制台顯示器，不能被重定向。

第11833行到11884行的主迴圈原理上與其他任務的主迴圈是一樣的——它接收一條消息，根據消息類型執行一條switch語句調用適當的函數，並產生一條返回消息。但是這裡有一點複雜。首先，許多工作是由低層的中斷常式完成的，特別是在處理終端輸入時。在上一節中我們看到，從鍵盤輸入的各個字元要被接收並緩衝，而不是為每個字元向終端任務發一條消息。這樣，在試圖接收一條消息之前，主迴圈總是掃描整個tty\_table，檢查每個終端的tp->tty\_events標誌，如果必要就調用handle\_events（第11835行到11837行）來處理未完成的事務。只有在沒有需要立刻處理的事件時才產生一個調用來接收消息。如果接收到的消息來自硬體，就執行一條continue語句跳過這一次迴圈，重新檢查事件。

其次，這個任務可能為幾個設備服務。如果接收到的消息來自硬體中斷，那麼通過檢查tp->tty\_events標誌可以確認需要服務的設備。如果消息不是來自硬體中斷，那麼消息中的TTY\_LINE域可以用來確定是由哪個設備發出的。次設備號通過一系列的比較被解碼，借助於設備號tp指向tty\_table中正確的表目（第11845行到11864行）。如果該設備是一個偽終端，那麼就調用do\_pty（在pty.h中）並且重新開始主迴圈。在這種情況下，do\_pty生成應答消息。當然，如果偽終端沒有被使能，那麼對do\_pty的調用將使用前面定義的空的宏。人們希望試圖存取不存在設備的情況不會發生，但加上一個檢查過程要比確保系統中任何地方都不會出錯容易得多。在設備不存在或沒有配置時，就產生一條ENXIO錯誤消息，並且控制返回到迴圈的頂部。

任務的其餘部分和我們在其他任務的主迴圈部分看到的相類似，根據消息的類型進行一個switch選擇（第11874行到11883行），調用請求對應的函數，如do\_read，do\_write等等。每個被調用的函數產生回答消息，而不是把構造消息所需要的資訊傳回主迴圈。只有在一個有效的消息類型沒有被收到時，主迴圈的結束部分才產生一條EINVAL錯誤資訊。因為回答消息是在終端任務內部許多不同地方發出的，所以可以調用一個公共的常式tty\_reply來處理構造回答消息的細節。

如果tty\_task收到的消息是一個有效的消息類型而不是一個中斷的結果，也不是來自一個偽終端，那麼主迴圈最後的switch將把它分發給函數do\_read，do\_write，do\_ioctl，do\_open，do\_close和do\_cancel中的一個。每個函式呼叫的參數是指向tty結構的指標tp和消息的位址。在分別考察這些函數之前，我們要先進行一些一般性的討論。由於tty\_task可能要為多個終端設備服務，所以這些函數必須很快地返回以便主迴圈能繼續回應其他請求。然而，do\_read，do\_write和do\_ioctl可能無法立即處理完所有的請求。為了允許FS為其他調用服務，需要立即回應。如果請求不能立即結束，那麼就在回答消息的狀態域中返回一個SUSPEND碼。這對應於圖3-37中的消息（3），它掛起開始調用的進程，同時釋放FS。圖中消息（7）和（8）在操作結束後被發送。如果請求可以被完全滿足，或者發生了一個錯誤，那麼就在返回給FS的消息的狀態域中返回傳送的位元組數或錯誤碼。在這種情況下FS立即向進行原始調用的進程發送一條消息，喚醒該進程。

讀終端與讀磁片設備是完全不同的。磁片驅動程式向磁片硬體發出一條指令，資料就被逐漸返回，除非發生機械或電子故障。電腦可以在螢幕上顯示一條提示資訊，但卻無法強迫一個人坐在鍵盤前開始輸入。因此，電腦根本無法保證有人坐在那兒。為了能夠儘快得到返回，在有輸入到達時，do\_read（第11891行）一開始就會保存使得請求可以稍後被完成的資訊。首先要進行一些錯誤檢查。當設備仍然在等待輸入以滿足前一個請求，或消息中的參數無效時，將發生一個錯誤。如果通過了檢查，第11911行到11915行就把關於請求的資訊拷貝到設備的tp->tty\_table表目中合適的域。下一步把tp->tty\_inleft設置為所需的字元數非常重要。這個變數用來決定讀請求何時得到滿足。在規範模式中，tp->tty\_inleft隨著每個返回的字元逐漸減少，直到接收到一行的結束時，這個值突然減少為0。在非規範模式中，處理有所不同，但在任何情況下，只要調用被滿足tp->tty\_inleft就被置為0，不論是由於超時還是接收到了要求的最小字元數。當tp->tty\_inleft達到0時，一條回答消息被送出。如同我們將要看到的那樣，回答消息可以在好幾個地方產生。有時還需要檢查一個讀進程是否仍在等待一個回答，非0時的tp->tty\_inleft可以用作這個目的。

在規範模式中，一個終端設備一直處於等待直到已接收到調用中需要的字元數，或達到了一行或一個檔的末尾。第11917行通過檢查termios結構中的ICANON位元判斷終端是否處於規範模式。如果這一位沒有置位，就檢查termios的MIN和TIME值以決定採取什麼動作。

在圖3-35中我們可以看到MIN和TIME是怎樣相互作用以提供一個讀調用可能採取的不同動作。TIME在第11918行中檢測。0值對應於圖3-35中的左邊一列，在這種情況下不需要進一步的檢測了。如果TIME不為0，那麼就檢測MIN。如果為0，就調用settimer啟動計時器，即使沒有收到一個位元組，在延遲一段時間後也將結束DEV\_READ請求。這裡tp->tty\_min被置為1，使得在超時前如果接收到一個或幾個位元組就立即終止調用。這裡還沒有對可能的輸入進行檢查，所以可能有不止一個字元在等待滿足請求。在這種情況下，發現輸入後將立即返回READ調用中指定個數的字元。如果TIME和MIN都不為0，那麼計時器將有不同的含義。在這種情況下，計時器用來作為字元間的計時器。它只在收到第一個字元後啟動，並在接收到每個後繼字元之後重新開機。tp->tty\_eotct在非規範模式下對字元計數，如果在第11931行中為0，那麼就還沒有接收到字元，並且位元組間計時器被禁用。lock和unlock用來保護所有對settimer的調用，防止在settimer運行時發生時鐘中斷。

在任何情況下，第11941行都調用in\_transfer把所有已在輸入佇列中的位元組直接傳送到讀進程中。接著進行一個handle\_events調用，這個調用可能向輸入佇列中送入更多的資料，然後再次調用in\_transfer。這裡需要解釋一下這個顯得重複的調用。雖然到目前為止所有的討論都從鍵盤輸入的角度來考慮，但do\_read是在代碼的設備無關部分，也要為通過串列線連接的遠端終端機的輸入服務。上一次的輸入可能已經填滿了RS-232輸入緩衝區從而導致輸入被禁止。對in\_transfer的第一次調用不能重新啟動輸入流，但對handle\_events的調用可以起到這個作用。由此引起的對in\_transfer的第二次調用並沒有實質性的作用，主要是為了確保遠端終端機再次被允許發送。這些調用都可能滿足請求並向FS回答消息。tp->tty\_inleft被用作一個標誌來判斷回答是否已發送；如果在11944行中它仍為0，那麼do\_read就自己產生並發送回答消息。這些由第11949行到11957行完成。如果原始調用指定了一個不可阻塞讀，FS將被通知向原始調用發送一個EAGAIN錯誤碼。如果調用是一個普通的可阻塞讀，FS將收到一個SUSPEND碼，釋放FS但通知它保持原始調用者阻塞。在這種情況下，終端的tp->tty\_inrepcode域被置為REVIVE。如果READ稍後被滿足，這個值將被放在給FS的回答消息中以指示原始調用被置為睡眠態需要喚醒。

Do\_write（第11964行）和do\_read很相似，但更簡單一些，因為在處理一個WRITE系統調用時需要考慮的情況較少。用和do\_read中相類似的檢查來檢測上一次寫是否不在進行以及參數是否有效，然後把請求的參數拷貝到tty結構中。接著調用handle\_events，並檢查tp->tty\_outleft以判斷是否完成（第11991行到11992行）。如果是，那麼handle\_events已經發送了一條回答消息，不需要再做什麼。如果沒有，那麼就產生一條回答消息，消息參數依賴於原始的WRITE調用是否是不可阻塞模式。

圖 3－43 POSIX調用和IOCTL操作。

下一個函數do\_ioctl（第12012行）很長，但不難理解。do\_ioctl的函數體是兩條switch語句。第一個確定由請求消息中的指標指向的參數的大小（第12033行到12064行）。如果大小不為0，就檢驗參數的有效性。這裡不能檢測內容，但可以檢查一個在指定位址處所需大小的結構是否能裝入它被指定放置的段。函數的餘下部分是根據IOCTL操作請求類型進行選擇的switch語句（第12075行到12161行）。不幸的是，用IOCTL調用支持POSIX要求的操作意味著必須構造IOCTL操作的名字而不是複製POSIX要求的名字。一個TCGETS操作為使用者的tcgetattr調用服務，並且只是簡單地返回終端設備的tp->tty\_termios結構的一份拷貝。下面的四個請求類型共用代碼。TCSETSW、TCSETSF和TCSETS請求類型對應使用者調用POSIX定義的函數tcsetattr，並且都有把一個新的termios結構拷貝到一個終端的tty結構中這樣的基本動作。拷貝動作用一個phys\_copy調用從使用者進程取得資料，接下來在12098行到12099行調用setattr。對於TCSETS調用，拷貝動作立即執行；對於TCSETSW和TCSETSF調用，則可能在輸出結束後執行。如果調用tcsetattr時使用了一個修飾符請求把動作延遲到當前輸出完畢後執行，並且如果在第12084行中對tp->tty\_outleft的檢測表明輸出沒有結束，那麼就把請求的參數放在終端的tty結構中供以後處理。tcdrain被翻譯為一個TCDRAIN類型的IOCTL調用，它掛起一個程式直到輸出結束。如果輸出已經結束，那麼它不再執行什麼動作。如果沒有，它也必須把資訊保存在tty結構中。

POSIX的tcflush函數根據參數丟棄未讀入的輸入和/或未發送的輸出。它被直接翻譯為IOCTL調用，由一個對為所有終端服務的函數tty\_icancel和/或由tp->tty\_ocancel指向的設備指定函數的調用組成（第12102行到12109行）。tcflow也被類似地直接翻譯為一個IOCTL調用。為了掛起或重置輸出，它向tp->tty\_inhibited中寫入一個TRUE或FALSE，然後置位元tp->tty\_events標誌。為了掛起或重置輸入，它使用由tp->tty\_echo（第12120行到12125行）指向的設備指定回顯常式向遠端終端機發送適當的STOP（正常為CTRL-S）或START（CTRL-Q）碼。

餘下的由do\_ioctl處理的大部分操作都由一行代碼調用一個合適的函數處理。對於KIOCSMAP（裝入鍵位映射表）和TIOCSFON（裝入字體）操作要進行檢驗以保證該設備確實為控制台，因為這些操作不應用於其他終端。如果使用了虛擬終端，那麼所有的控制台都將使用相同的鍵位映射表和字體，硬體不允許用其他的方法。視窗大小操作在使用者進程和終端任務之間拷貝winsize結構。不過要注意TIOCSWINSZ操作對應的代碼下面的注釋。當一個進程改變了它的視窗大小時，在某些版本的UNIX下，內核應該向進程組發送一個SIGWINCH信號。POSIX標準中並不要求這個信號。但是任何想要使用這些結構的人應該考慮在這兒添加代碼初始化這個信號。

do\_ioctl的後兩種情況支援POSIX要求的tcgetpgrp和tcsetpgrp函數。這兩種情況沒有對應的動作，只是永遠返回一個錯誤。這並沒有什麼毛病。這些函數支援作業控制（job control），能夠從鍵盤上掛起和重啟一個進程。POSIX不要求作業控制，在MINIX中也不支持。不過，即使不支援作業控制，POSIX也需要這些函數以保證程式的可攜性。

do\_open（第12171行）執行的是一個簡單的基本動作——它增加設備的tp->tty\_openct變數的值以便能夠檢驗是否被打開。不過，在這之前首先要進行一些檢測。對於普通終端，POSIX指定第一個打開終端的進程為會話主導進程。當一個會話領導終止時，將收回該組中的其他進程對該終端的存取權。監控程序需要能夠輸出出錯資訊，並且如果錯誤資訊輸出沒有被重定向到一個檔中，那麼它應當被送到一個不能被關閉的顯示器上。為此MINIX中存在一個稱為/dev/log的設備。在物理上它與dev/console是相同的設備，但它由一個獨立的次設備號標識，處理方式也不同。它是一個隻寫設備，這樣如果do\_open試圖以讀方式打開它將返回一個EACCESS錯誤（第23283行）。do\_open執行的另一個檢測是測試O\_NOCTTY標誌。如果該標誌沒有被置位元並且設備不是/dev/log，則該終端成為一個進程組的控制終端。這是通過把調用方的進程號放入tty\_table表目的tp->tty\_pgrp域完成的。然後，增加tp->tty\_openct變數的值並發送回答消息。

一個終端設備可能被打開不止一次，而下一個函數do\_close（第12198行）除了減少tp->tty\_openct變數的值外不執行其他動作。第12198行的檢測防止了該函數試圖關閉設備/dev/log。如果這個操作是最後一次關閉操作，那麼就調用tp->tty\_icancel取消輸入。由tp->tty\_ocancel和tp->tty\_close指向的設備指定常式也被調用。然後tty結構中的各個域被置回他們的預設值並發送回答消息。

最後一類消息由do\_cancel處理（第12220行）。當一個試圖讀或寫的被阻塞的進程接收到一個信號時將引起這個動作。這時需要檢查三種狀態：

1. 終止時進程可能在讀。

2. 終止時進程可能在寫。

3. 進程可能被tcdrain掛起直到它的輸出結束。

對每種情況進行檢測，並根據需要調用通用的tp->tty\_icancel或由tp->tty\_ocancel指向的設備指定常式。在最後一種情況下，唯一要做的就是重置tp->tty\_ioreq標誌來指示IOCTL操作現在已經完成。最後，tp->tty\_events標誌被置位元並發送一條回答消息。

終端驅動程式支援代碼

現在我們已經研究了tty\_task中主迴圈調用的頂層函數，下面將要學習支援他們的代碼。我們將從handle\_events開始（第12256行）。前面提到，每執行一次主迴圈，都要檢查每個終端設備的tp->tty\_events標誌，如果需要處理某個特定設備的事件就調用handle\_events。do\_read和do\_write也調用handle\_events。這個常式必須工作得很快。它重置tp->tty\_events標誌，然後調用由指標tp->tty\_devread和tp->tty\_devwrite（第12279行到12282行）指向的設備指定常式讀和寫。這些都是被無條件地調用的，因為這裡無法檢驗一次讀或寫是否會引起標誌的置位元——這兒有一個設計上的選擇，即對每個設備檢查兩個標誌的代價要高於每次一個設備被啟動時做兩次調用的代價。另外，在大部分時間裡從終端接收到的字元是需要回顯的，所以兩個調用都是需要的。如同在討論處理do\_ioctl調用的tcsetattr時所提到的那樣，POSIX可能會推遲對設備的控制操作直到當前的輸出結束，所以在調用設備指定函數tty\_devwrite後立即處理ioctl操作是一個很好的時機。這些在第12285行中完成：如果有一個等待的控制請求，就調用dev\_ioctl。

由於tp->tty\_events標誌由中斷置位元，並且字元可能由一個快速設備以很快的速度到達，因此可能出現在對設備指定的讀和寫常式以及dev\_ioctl的調用結束時，另一個中斷又使得標誌置位元的情況。因此將輸入資訊由緩存中中斷常式放置的初始位置移出被賦予較高的優先權。這樣只要在迴圈結束處（第12286行）檢測到tp->tty\_events標誌置位元，handle\_events就重複地調用設備指定常式。當輸入流停止時（也可以是輸出，但輸入更可能這樣重複地請求），就調用in\_transfer把字元從輸入佇列中傳送到調用一個讀操作的進程的內部緩衝區中。如果傳送結果結束了請求，無論是傳送了請求的最大字元數還是達到了一行的末尾（在規範模式中），in\_transfer本身就發送一條回答消息。如果是這樣，那麼在返回到handle\_events時tp->tty\_left的值為0。這裡還對傳送的字元數是否達到請求的最小數目進行進一步檢查並發送一條回答資訊。對tp->tty\_inleft的檢測防止了重複消息的發送。

下一步我們研究in\_transfer（第12303行），這個函數把資料從任務存儲空間中的輸入佇列中移動到請求輸入的使用者進程的緩衝區中。然而，直接拷貝是不行的。輸入佇列是一個環型緩衝區，並且需要檢查字元以判斷是否已達到檔末尾，或者檢查是否處於規範模式下，傳送只持續到一行結束。另外，輸入佇列中各單元大小為16位元，而接收緩衝區是一個8位元字元的陣列。這樣就需要使用一個中間局部緩衝區。字元在放入局部緩衝區時被逐個檢查，當緩衝區被填滿或者輸入佇列被移空時，就調用phys\_copy把局部緩衝區中的內容移到接收進程的緩衝區中（第12319行到12345行）。

tty結構中有三個變數tp->tty\_inleft，tp->tty\_eotct和tp->tty\_min用於確定in\_transfer是否需要完成一些任務，前兩個變數用於控制主迴圈。前面已經提到，tp->tty\_inleft一開始被設置為一個READ調用所請求的字元數。正常情況下，每傳送一個字元，它就被減一。但是當到達一行的結束時，它會突然減少到0。只要tp->tty\_inleft為0，就向讀進程發送一條回答資訊，因此該變數也可以作為一個標誌指示是否已經發送回答消息。所以，在第12314行中，檢測到tp->tty\_inleft為0可以作為不發送回答消息就中止in\_transfer的執行的充分理由。

檢測的下一部分比較tp->tty\_eotct和tp->tty\_min兩個變數。在規範模式中，這兩個變數與完整的輸入行有關；而在非規範模式中，它們與字元有關。只要一個“行中止”字元或一個位元組被放入輸入佇列，就增加tp->tty\_eotct的值。這樣tp->tty\_eotct就可以對已被終端任務接收但還未傳給一個讀進程的行或位元組計數。tp->tty\_min則指示了滿足一個讀請求所必須傳送的最小的行數（規範模式）或字元數（非規範模式）。在規範模式中它的值總是為1，在非規範模式中，它可以是從0到MAX\_INPUT（MINIX中為255）之間的任何一個值。行12314中的第二部分檢測使得在規範模式下如果沒有接收到一個整行，in\_transfer就立即返回。傳送並不馬上執行，而是等到一行結束之後，這樣佇列的內容就可以被修改。例如，如果用戶在按下ENTER之前接著輸入ERASE或KILL字元，那麼將丟棄佇列中的內容。在非規範模式中，如果無法取得所需的最小字元數，就立即返回。

在幾行之後，tp->tty\_inleft和tp->tty\_eotct被用來控制in\_transfer的主迴圈。在規範模式中，傳送一直持續到佇列中不再有完整的行。在非規範模式中，tp->tty\_eotct用來對等待的字元計數。tp->tty\_min用來控制何時進入迴圈，但不用來決定何時結束。一旦進入迴圈，或者傳送所有的字元，或者就傳送原始調用所需的字元數，這取決於那一個更小。

圖 3－44 字元放入輸入佇列時字元碼中的域。

輸入佇列中的字元大小為16位元。實際傳給使用者進程的是低8位元。圖3-44顯示了高位位元組是怎樣使用的。三位元被用來標誌字元是否被轉義（CTRL-V）、是否表示檔結束，或是否是表示行結束的幾個代碼之一。四位元用來計數字元回顯時佔據的螢幕空間。第12322行檢查IN\_EOF位（圖中D）是否被置位。檢測是在內迴圈頂部進行的，因為檔結束符本身不傳給使用者進程，也不計入字元計數。在傳送每個字元時，用一個遮罩使高8位為0，只有低8位中的ASCII值被送到局部緩衝區（第12324行）。

標誌輸入結束的方法不止一種，但一般使用設備指定常式確定接收到的字元是否為分行符號，CTRL-D，或者其他這樣的字元。in\_transfer只需要在第12340行檢測IN\_EOT位（圖3-44中的N）這個標誌。如果檢測到，就減少tp->tty\_eotct的值。在非規範模式中，每個字元在放入輸入佇列中時都這樣被計數，並且每個字元都在這時候被標記IN\_EOT位元，所以tp->tty\_eotct實際上就是佇列中未移走的字元的計數。in\_transfer的主迴圈在兩種不同的模式下的唯一區別在第12343行。這裡如果發現一個字元標誌為行中止符，則tp->tty\_inleft被置為0，但只有在規範模式下才這樣做。這樣當控制返回到迴圈的頂部時，在規範模式中迴圈在一個行中止符之後終止，但在非規範模式中行中止符被忽略。

當迴圈中止時，一般會有一個部分被填滿的局部緩衝區等待傳送（第12347行到12353行）。然後如果tp->tty\_inleft達到0，那麼就發送一條回答消息。在規範模式下總是這種情況，但如果處於非規範模式並且傳送的字元數少於全部的請求，那麼將不發送回答。如果你還記得我們在調用in\_transfer時所看到的（在do\_read和handle\_events中），當in\_transfer在返回時傳送了多於tp->tty\_min所指定的字元數時，在in\_transfer調用之後的代碼將發送一條回答消息，那麼你會覺得很迷惑，因為現在就是這種情況。在我們討論下一個函數時，你將會看到in\_transfer為什麼不無條件地發送回答消息。這個函數在一個不同的環境中調用in\_transfer。

這個函數就是in\_process（第12367行）。它由設備指定軟體調用來執行對所有輸入都要進行的共同處理。它的參數是一個指向tty結構的指標，一個指向待處理的8位元字元陣列的指標和一個計數器。計數器被返回給調用者。in\_process是一個很長的函數，但它的動作並不複雜。它把16位元的字元加入到輸入佇列中，這些字元稍後將由in\_transfer處理。

in\_transfer中提供了幾種不同的處理方法。

1. 普通字元被加入到輸入佇列中，並擴展為16位。

2. 影響以後處理的字元，通過修改標誌位元來表示影響，但不放入佇列。

3. 控制回顯的字元立即生效，不放入佇列。

4. 具有特殊標記的字元在放入緩衝區時把編碼如EOT位元加入它們的高位元組。

讓我們先來看一下在一個完全正常的環境中，如何從一個設置為標準MINIX預設屬性的終端上輸入的一條短句中間的一個普通字元，例如“x”（ASCII碼0x87）。從輸入裝置接收到時，這個字元佔用了圖3-44中的0到7位。在第12385行，如果ISTRIP被置位元，那麼該字元的最高有效位元第7位被重置為0，但MINIX中的預設值是不清除這一位元，允許輸入全部的8位元。不過這不會影響我們的“x”。MINIX的預設值是允許輸入的擴充處理，所以能夠通過對tp->tty\_termios.c\_flag（第12388行）中IEXTEN位的檢測。但接下來的檢測在我們的假設之下不能通過：沒有字元轉義（第12391行）。輸入本身不是字元轉義字元（第12397行），並且該輸入不是REPRINT字元（第12405行）。

下面幾行的檢測判斷出字元既不是特殊的\_POSIX\_VDISABLE字元，也不是CR和NL字元。最後，一個正的結果：處於規範模式，這是正常的預設值（第12424行）。然而，我們的“x”既不是ERASE字元，也不是KILL、EOF（CTRL-D）、NL或EOL字元之一，所以直到第12457行，仍然沒有對它進行任何處理。這裡檢測到IXON被默認置位，允許使用STOP（CTRL-S）和START（CTRL-Q）字元，但在接下來的測試中卻沒有發現匹配。在第12478行發現ISIG位被默認置位，允許使用INTR和QUIT字元，但同樣找不到匹配。

實際上，對一個普通字元第一個值得注意的處理發生在第12491行，這裡對輸入佇列是否滿進行檢測。如果已滿，那麼字元將在這裡被丟棄，因為這時處於規範模式下，使用者看不到它回顯在螢幕上（continue語句丟棄該字元，因為它重新開始外迴圈）。然而，由於我們在說明時假設了完全正常的情況，所以我們假定緩衝區還沒有滿。下一個檢測判斷是否需要一個特殊非規範模式處理（第12497行），如果失敗，就向前跳到第12512行。這裡由於tp->tty\_termios.c\_flag中的ECHO位是默認置位的，所以調用echo顯示字元。

最後，在第12515行到12519行，字元被放入輸入佇列。這時要增加tp->tty\_incount的值，但是由於該字元為普通字元，沒有EOT位元標誌，所以tp->tty\_eotct不變。

如果剛送入佇列的字元填滿了佇列，那麼迴圈中的最後一行就調用in\_transfer。然而，在一般條件下，對於這個例子我們假設in\_transfer即使被調用也將不執行任何動作，這是因為（假設佇列被正常服務，並且當前一行結束時，前一次輸入被接受）tp->tty\_eotct為0，tp->tty\_min為1，在in\_transfer開頭的檢測（第12314行）使得它立即返回。

在一個普通字元的情況下執行了一遍in\_process後，讓我們回到in\_process的開頭看一看在特殊一點的情況下將會如何。首先我們討論轉義字元，它允許一個普通字元把一種特殊影響傳遞給使用者進程。如果轉義字元有效，tp->tty\_escaped將被置位，當檢測到這個標誌時（在第12391行），該標誌立即被重置，並且IN\_ESC位，即圖3-44中的V位，被加到當前的字元中。這使得字元在回顯時被特殊處理——轉義控制字元用“^”加字元的形式顯示使它們可見。IN\_SEC位元也防止字元被識別為其他特殊字元。下面幾行處理處理轉義字元本身，LNEXT字元（預設為CTRL-V）。當檢測到LNEXT字元時，tp->tty\_escaped標誌被置位元，並且調用兩次rawecho輸出一個“^”後面接一個後退字元。這使得使用者想起在轉義字元有效的鍵盤上，當後繼字元被回顯時將覆蓋“^”。LNEXT字元是影響後繼字元的字元的一個例子（在這裡，是緊接著的字元）。它不放在佇列中，並且在兩次對rawecho的調用後重新啟動迴圈。這兩次檢測的次序很重要，這使得可以在一行中輸入LNEXT兩次而把第二個拷貝送給一個進程。

in\_process處理的下一個特殊字元是REPRINT字元（CTRL-R）。當in\_process發現接下來是一個reprint調用時（第12406行），將重新顯示當前的回顯輸出。REPRINT本身被丟棄，對輸入佇列沒有影響。

考慮每個特殊字元的處理細節是很冗長乏味的，並且in\_process的原始程式碼是很簡單明瞭的。我們將只提幾個地方。一個是在放入輸入佇列的16位元值的高位使用特殊位元使得區分有相似作用的一類字元顯得很容易。比如，EOT（CTRL-D），LF和可選的EOL字元（缺省時沒有定義）都由EOT位元標誌，即圖3-44中的D位（第12447行到12453行），這使得後面的識別更容易了。最後，我們將說明前面提到的in\_transfer的特別行為。雖然我們在前面看到在in\_transfer調用中似乎在返回時總是產生一條回答消息，但實際上回答消息並不是每次中止時都產生。回憶當輸入佇列滿時（第12522行），in\_process對in\_transfer的調用在規範模式下沒有作用。但如果在非規範模式中，第12499行將為所有的字元打上EOT標誌，這樣每個字元都在第12519行中被tp->tty\_eotct計數。在返回時，這將引起進入由於在非規範模式中輸入佇列滿而引起的in\_transfer調用的主迴圈。在這種場合當in\_transfer終止時不應該發送消息，因為在返回in\_process後很可能讀入了過多的字元。實際上，雖然在規範模式中單個READ的輸入由輸入佇列的大小限制（MINIX中為255個字元），但是在非規範模式下，一個READ調用必須能夠傳送POSIX要求的\_POSIX\_SSIZE\_MAX個字元。在MINIX中這個值為32767。

tty.c中的下面幾個函數支援字元輸入。ECHO（第12531行）對一些字元進行特殊處理，但對大多數字元只是在和輸入使用的同一個設備的輸出方顯示出來。在輸入被回顯的同時，一個進程可能正好要向同一個設備輸出，這時如果使用者正試圖從鍵盤退格，就有可能引起混亂。為了處理這種情況，在產生正常的輸出時，設備指定輸出常式總是把tp->tty\_reprint標誌設置為TRUE，這樣處理退格的函數就能夠判斷出是否產生了混合輸出。由於echo也使用設備輸出常式，所以tp->tty\_reprint的當前值在回顯時由區域變數rp（第12552行到12585行）保存起來。不過，如果剛開始了一個新的輸入行，rp將被置為FALSE而不是保持原值以保證tp->tty\_reprint在echo終止時被重置。

你可能已經注意到echo將返回一個值，例如，在in\_process中的第12512行的調用中：

ch = echo(tp,ch)

echo返回的值包括回顯字元所用的螢幕上的空格數，如果是TAB字元，這個值可以一直到8。這個值被存放在圖3-44中的cccc域。普通字元在螢幕上佔據一格，但一個控制字元（除TAB，NL，CR或DEL（0x7F）外）回顯時，該字元被顯示為“^”加一個可列印的ASCII字元的形式，在螢幕上佔用兩個位置。而另一方面NL或CR不佔用空間。當然，實際的回顯必須由設備指定常式完成，並且不論何時一個字元必須被送往設備時，就如同在第12580行中對普通字元那樣，用tp->tty\_echo執行一個間接調用，。

下一個函數rawecho用來旁路由echo進行的特殊處理。它檢測ECHO標誌是否被置位元，如果是，就不經任何特殊處理把字元送往由tp->tty\_echo指定的設備指定常式。這裡使用了一個區域變數rp以防止rawecho自己對輸出常式的調用改變tp->tty\_reprint的值。

當in\_process發現一個後退字元，下一個函數backover（第12607行）將被調用。它對輸入佇列進行操作，如果在佇列中可以回溯，那麼就刪除佇列中的前一個字元。如果佇列為空或上一個字元為行中止符，那麼就不能回溯。這裡也檢測在討論echo和rawecho時提到的tp->tty\_reprint標誌。如果為TRUE，那麼就調用reprint（第12618行）在螢幕上放置一個輸出行的純拷貝。然後查詢上一個顯示字元的len域（圖3-44中的cccc域）來確定需要在顯示器上刪除多少個字元，並且對每個字元，都由rawecho發出一個退格-空格-退格字元序列從螢幕上刪除不需要的字元。

下一個函數是reprint。除了被backover調用外，用戶也可以通過按下REPRINT鍵（CTRL-R）來引用該函數。第12651行到12656行的迴圈在輸入佇列中逆向查找最後一個行中止符。如果找到的是位於佇列的最末尾的位置，那麼reprint什麼也不做就返回。否則就回顯CTRL-R，即在螢幕上顯示兩字元序列“^R”，然後移到下一行，重顯佇列中從上一個行中止符到末尾的部分。

現在我們已經討論到out\_process函數（第12677行）。和in\_process一樣，它也由設備指定輸出常式調用，但要簡單一些。它由RS-232和偽終端設備指定輸出常式調用，但不能被控制台常式調用。out\_process工作在環形緩衝區上，但不從緩衝區中刪除字元。它對緩衝區陣列所做的唯一變動是如果tp->tty\_termios.oflag中的OPOST（使能輸出處理）和ONLCR（映射NL為CR-NL）位被置位，就在緩衝區中的NL字元前插入一個CR字元。MINIX中這兩位都默認為置位。out\_process的任務是保持設備tty結構中的變數tp->tty\_position更新。定位字元和後退字元使得情況要複雜一些。

下一個常式是dev\_ioctl（第12763行）。當用TCSADRAIN或TCSAFLUSH選項調用dev\_ioctl時，它都支援執行tcdrain函數和tcsetattr函數。在這些情況下，dev\_ioctl在輸出沒有結束時不能立即完成動作，所以延遲IOCTL操作的資訊被保存在tty結構的一部分中。無論handle\_events何時運行，它都要在調用設備指定輸出常式後檢查tp->tty\_ioreq域，並且如果有操作不能確定，就調用dev\_ioctl。dev\_ioctl檢查tp->tty\_outleft判斷輸出是否結束，如果是，就執行和dev\_ioctl在不延遲立即執行時相同的動作。為tcdrain服務的唯一的動作就是重置tp->tty\_ioreq域並向FS發送回答消息，通知它喚醒原始調用進程。tcsetattr在TCSAFLUSH選項下調用tty\_icancel取消輸入。對tcsetattr的兩個選項，termios結構被拷貝到設備的tp->tty\_termios結構中，termios結構的位址已在對IOCTL的原始調用中被傳遞。然後調用setattr，接下來和tcdrain一樣，通過發送一條回答消息來喚醒被阻塞的原始調用。

下一個過程是setattr（第12789行）。如同我們已經看到的，它由do\_ioctl或dev\_ioctl調用來改變一個終端設備的屬性，而do\_close則調用它來把屬性重置為預設設置。setattr總是在把一個新的termios結構拷貝到一個設備的tty結構中後被調用，這是因為只拷貝參數是不夠的。如果被控制的設備正處於非規範模式，那麼第一個動作就是當前輸入佇列中的所有字元都標上IN\_EOF位元，如同在處於非規範模式下這些字元剛被輸入到佇列中時所要做的一樣。直接這樣做（第12803行到12809行）要比檢測字元是否有該位元容易一些。因為沒有辦法可以知道哪個屬性剛被改變，哪個還保持原值。

下一個動作是檢測MIN和TIME值。在規範模式中tp->tty\_min總是為1；這在第12818行置位。在非規範模式中，兩個值的組合允許四種不同的操作，如圖3-35所示。在第12823行到12825行tp->tty\_min首先被設置為tp->tty\_termios.cc[VMIN]中傳過來的值，然後，如果它為0並且tp->tty\_termios.cc[VTIME]不為0，那麼它將被修改。

最後，setattr確保如果XON/XOFF控制被禁止，輸出就不停止，如果輸出速度被設置為0，就發送一個SIGHUP信號，並且執行一個由tp->tty\_ioctl指向的設備指定常式的間接調用，來完成只能在設備層完成的工作。

下一個函數tty\_reply（第12845行）在前面的討論中已經被多次提到。它的動作十分簡單：構造並發送一條消息。如果因為某種原因回答失敗，就會發生混亂。接下來的函數也同樣簡單。sigchar（第12866行）請求MM發送一個信號。如果NOFLSH標誌沒有置位元，已在排隊的輸入將被刪除——接收到的字元或行的計數被置為0，佇列的頭指標和尾指針重合。這是默認動作。當即將捕獲一個SIGHUP信號時，NOFLSH可以被置位元，以在捕獲信號後恢復輸入和輸出。tty\_icancel（第12891行）根據sigchar描述的方法無條件地丟棄等待的字元，並且另外再調用由tp->tty\_icancel指向的設備指定函數取消可能留在設備本身裡或被低層代碼緩衝的輸入。

對每個設備tty\_init（第12905行）在tty\_tasks第一次啟動時被調用一次。它的設置為預設值。開始時，一個指向tty\_devnop的指標和一個什麼也不做的空函數被設置到變數tp->tty\_icancel、tp->tty\_ocancel、tp->tty\_ioctl和tp->tty\_close中。然後tty\_init根據不同類型的終端（控制台，串列線，或偽終端）調用設備指定初始化函數。它們設置間接調用設備指定函數的實際的指標。前面講過，如果根本就沒有配置某一特定類型的設備，那麼將生成一個立即返回的巨集，這樣就不需要為沒有配置的設備編譯代碼了。對scr\_init的調用初始化控制台驅動程式，並且也調用鍵盤的初始化常式。

tty\_wakeup（第12929行）雖然短，但在終端任務的功能中卻十分重要。只要時鐘中斷處理常式運行，也就是說，對每個時鐘滴答，全域變數tty\_timeout（在glo.h第5032行定義）都被檢查是否包含小於當前時間的值。如果是，那麼就調用tty\_wakeup。tty\_timeout被終端驅動程式的插斷服務常式置為0，所以在每個終端設備中斷的下一個時鐘滴答喚醒操作都被強迫執行。當一個終端設備為一個非規範模式下的READ調用服務並且如同我們馬上就要看到的那樣需要設置一個超時，那麼tty\_timeout也會被settimer改變。當tty\_wakeup運行時，它首先把tty\_timeout置為TIME\_NEVER，一個未來的很遠的時間，以禁止下一次喚醒。然後它掃描按最早存入的最先被喚醒排序的計時器值的鏈表，直到達到比當前時間遲的那個計時器。這就是下一個將要執行的喚醒，它被放入tty\_timeout中。tty\_wakeup還把該設備的tp->tty\_min設置為0，以保證即使沒有收到字元下一個讀操作也能成功，把設備的tp->tty\_events標誌置位元以保證當終端任務下一次運行時得到處理，並且把該設備從計時器鏈表中刪除。最後，它調用interrupt向任務發送喚醒消息。如同在討論時鐘任務時提到的，tty\_wakeup邏輯上是時鐘中斷服務代碼的一部分，因為它只從那兒被調用。

下一個函數settimer（第12958行）設置計時器以確定在非規範模式中何時從一個READ調用中返回。它的調用參數為一個指向tty結構的指標tp，和一個值為TRUE或FALSE的整數on。首先掃描由timelist指向的tty結構鏈表，尋找與tp參數匹配的已存在的表目。如果找到，就把它從表中刪除（第12968行到12973行）。如果settimer被調用來取消設置一個計時器，那麼這些就是它所要做的全部工作。如果它被調用來設置一個計時器，那麼設備tty結構中的tp->tty\_time元素被設置為當前時間加上在設備的termios結構的TIME值中指定的以十分之一秒為單位的增量。然後把匹配表目按排好的次序放入表中。最後，比較剛放入表中的超時和全域變數tty\_timeout的值，如果新的超時較早，就用其取代後者。

最後，tty.c中還定義了一個tty\_devnop（第12992行），它是一個無操作函數，被間接地定位在一個不需要服務的設備中。我們已經看到在tty\_init中tty\_devnop被用作在調用設備的初始化常式之前置入各個函數指標的預設值。

3.9.5 鍵盤驅動程式的實現

現在我們回到支援由一個IBM PC鍵盤和一個記憶體映象顯示器組成的MINIX控制台的設備無關代碼。支援它們的物理設備是完全獨立的：在一個標準的桌面系統中顯示器使用一塊插在底板上的介面卡（至少由半打不同的基本型號），而鍵盤由設計在主機板上的電路支援，通過該電路與鍵盤內部的一個8位元單片機介面。兩個子設備需要完全獨立的軟體支援，即檔keyboard.c和console.c。

作業系統把鍵盤和控制台看作同一個設備/dev/console的兩個部分。如果顯示卡上有足夠的記憶體，就可以把虛擬控制台（virtual console）支援編譯進去，這樣除/dev/console外，還可以有其他的邏輯裝置，如/dev/ttyc1，/dev/ttyc2等。輸出只在任何給定的時刻送往顯示器，並且對任何被啟動的控制台都只有一個鍵盤用來輸入。在邏輯上鍵盤對於控制台是有用的，但只有兩個相對次要的方面說明了這一點。首先，tty\_table包含了一個控制台的tty結構，為輸入和輸出提供了單獨的域，例如，keyboard.c和console.c中的函數指標tty\_devread和tty\_devwrite在啟動時被賦值。不過，只有一個tty\_priv域，並且它只指向控制台的資料結構。其次，在進入主迴圈前，tty\_task調用每個邏輯裝置一次以進行初始化。為/dev/console調用的常式位於console.c中。從那裡調用鍵盤的初始化代碼。不過這種隱含的層次關係也可能被倒過來。在處理I/O設備時我們總是先看輸入再看輸出，現在我們仍然保持這種模式，在這一節中討論keyboard.c，而把console.c的留到後面討論。

和我們看到的大部分原始檔案一樣，keyboard.c的開頭也是幾條#include語句。不過其中有一條比較特殊。keymaps/us-std.src（在第13014行中包含）不是一個普通的標頭檔；這是一個C原始檔案，它使得默認鍵位映射表被編譯為keyboard.o中的一個已初始化的陣列。由於篇幅所限，鍵位映射表原始檔案沒有在本書的末尾列出，但圖3-41中示出了一些代表性的表目。在#include語句之後的是定義各種常數的宏。第一組用於和鍵盤控制器的低層交互。其中許多是在這些交互中有意義的I/O埠位址或位元組合。下一組包含了一些特殊鍵的符號名。由於IBM硬體只支援一個鍵盤，因此巨集kb\_addr（第13041行）總是返回一個指向kb\_lines陣列中第一個元素的指標。下一行中用符號定義了鍵盤輸入緩衝區的大小為KB\_IN\_BYTES，其值為32。接下來的11個變數用來保存在解釋一個按鍵時所需要記憶的各種狀態。它們的使用各不相同。例如，每次當Caps Lock鍵按下時，capslock標誌的值（第13046行）在TRUE和FALSE之間切換。當Shift鍵被按下時，shift標誌被置為TRUE，當Shift鍵被鬆開時，該標誌被置為FALSE。當接收到一個轉義掃描碼時，esc變數被置位元。在收到後繼字元時，它總是被重置。

第13060行到13065行中的kb\_s結構用來跟蹤輸入的掃描碼。在這個結構中，掃描碼被放在陣列ibuf中，該陣列為一個環形緩衝區，大小為KB\_IN\_BYTES。每個控制台說明了這些類型的一個陣列kb\_lines[NR\_CONS]，但實際上只使用了第一個，這是因為kbaddr宏總是被用來確定當前kb\_s結構的位址。不過，我們一般使用一個指向該結構的指標來引用kb\_lines[0]中的變數，例如kb->ihead，這樣我們就可以和處理其他設備的方法保持一致，並且使在文本中的引用與列出的原始程式碼一致。當然，由於有許多陣列元素未使用，要浪費一小部分空間。不過，如果有人製造出支援多個鍵盤的PC，MINIX也能支持，只需要對kbaddr宏進行一些修改。

map\_key0（第13084行）被定義為一個宏。它返回一個掃描碼對應的ASCII碼，忽略修飾符。這等價於鍵位映射表陣列中的第一列。它的兄弟函數是map\_key（第13091行），該函數進行掃描碼到ASCII碼的完全映射，包括處理和普通字元同時按下的（多重）修飾鍵。

鍵盤插斷服務常式是kbd\_hw\_int（第13123行），當一個鍵被壓下或鬆開時被調用。它調用scan\_keyboard從鍵盤控制器晶片獲取掃描碼。當一個鍵被鬆開引起中斷時，其掃描碼的最高有效位元被置位元，在這種情況下該鍵被忽略除非它是修飾鍵之一。如果中斷由按下一個鍵引起，或鬆開一個修飾鍵，則原始的掃描碼被放在環形緩衝區中，如果還有空間的話；當前控制台的tp->tty\_events標誌被置位元（第13154行），然後調用force\_timeout被調用以保證時鐘任務在下一個時鐘滴答啟動終端任務。圖3-45給出了在緩衝區中的一短行字元的掃描碼，其中包含兩個大寫字元，每個的前導是壓下一個shift鍵的掃描碼，而後繼是鬆開shift鍵的掃描碼。

圖 3－45 在輸入緩衝區中的由鍵盤輸入的一行文本的掃描碼，下面一行是對應的按下的鍵。其中，L+，L-，R+和R-分別代表按下和鬆開左、右Shift鍵。鬆開一個鍵的掃描碼比按下同一個鍵時大128。

當時鐘中斷發生時，就運行終端任務自身，當發現控制台的tp->tty\_events標誌被置位元時，就使用控制台tty結構中tp->tty\_devread域中的指標調用設備指定常式kb\_read（第13165行）。kb\_read從鍵盤的環形緩衝區中取出掃描碼，並把ASCII碼放在它的局部緩衝區中，緩衝區應足夠大以保存來自數位鍵盤的某些掃描碼產生的轉義序列。然後它調用硬體無關代碼中的in\_process把字元放入輸入佇列。第13181行到13183行使用了lock和unlock來防止kb->icount被一個可能在同一時刻到達的鍵盤中斷減少。對make\_break的調用以整數形式返回ASCII碼。特殊鍵，例如小鍵盤和功能鍵，他們的值大於0xFF。在HOME和INSRT之間的掃描碼（0x101到0x10C，在include/minix/keymap.h中定義）由按下數位鍵盤產生，並且通過numpad\_map陣列被轉換為圖3-46所示的三字元轉義序列。這些序列接著被傳送到in\_process中（第13196行到13201行）。更高的掃描碼不傳給in\_process，但要檢測ALT-LEFT-ARROW，ALT-RIGHT-ARROW和ALT-F1到ALT-F12這些掃描碼，並且如果發現其中之一，就調用select\_console切換虛擬控制台。

圖3－46 由數位鍵盤產生的轉義編碼。普通鍵的掃描碼被轉換為ASCII碼，而特殊鍵被轉換為大於0xFF的“偽ASCII”碼。

make\_break（第13222行）把掃描碼轉換為ASCII碼然後更新跟蹤修飾鍵狀態的變數。首先，它檢測PC用戶都知道的用來在MS-DOS下強制重啟動的神奇的CTRL-ALT-DEL組合。不過由於希望的是順序地關機，所以要向init，所有進程的父進程，發送一個SIGABRT信號，而不是試圖啟動PC BIOS常式。init應該在返回到可以完全重啟系統或重啟MINIX的啟動監控程序之前接收這個信號，把它解釋為開始一個按次序關機的進程的命令。當然，希望這個過程每次都能完成是不現實的。大部分的用戶清楚在某些地方真正出了問題並且系統已不能正常控制之前不按CTRL-ALT-DEL而突然關機的危險。這時系統可能變得如此的混亂以致按次序向另一個進程發送信號是不可能的。這就是在make\_break中為什麼有一個靜態變數CAP\_count的原因。大多數系統崩潰時，中斷系統仍能工作，所以鍵盤輸入仍然可以被接收，時鐘任務仍可以保持終端任務運行。這裡MINIX利用了電腦使用者的可預期的行為，即在某些東西看起來不能正常工作時重複地重擊鍵盤。如果試圖向init發送SIGABRT失敗，並且用戶按了CTRL-ALT-DEL兩次以上，就直接調用wreboot返回到監控程序而不經過調用init。

make\_break的主要部分是不難看懂的。變數make記錄掃描碼由按下還是放開一個鍵產生，接著被調用的map\_key把ASCII碼返回給ch。接下來是根據ch的一個switch語句（第13248行到13294行）。我們要考慮兩種情況，一種是普通鍵，一種是特殊鍵。對於普通鍵沒有可以匹配的情況，在缺省情況下也不做任何處理（第13292行），這是因為普通鍵只在按下和放開的產生（按下）階段被接收。如果由於某種原因一個普通鍵在放開時被接收，那麼就用-1代替，而調用者kb\_read將忽略這個鍵。而一個特殊鍵，例如CTRL，會在switch語句中合適的地方被檢測出來，對於CTRL是在第13249行。相應的變數，在這種情況下是control，記錄make的狀態，用-1代替要返回的字元掃描碼（被忽略）。對ALT，CALOCK，NLOCK和SLOCK鍵的處理要複雜一些，但是對所有這些特殊鍵的效果是相似的：一個變數記錄了當前的狀態（對只在按下時有效的鍵）或保持前一個狀態（對鎖定鍵）。

另外還要考慮一種情況，就是EXTKEY掃描碼和esc變數。它不會和鍵盤上的ESC鍵混淆，返回的ASCII碼是0x1B。從鍵盤上按任何一個鍵或複合鍵都是無法產生單獨的EXTKEY掃描碼的；它是PC 鍵盤的擴展鍵首碼（extended key prefix），是一個兩位元組掃描碼的第一個位元組，它說明了一個不是初始PC擴充鍵的一部分但具有相同的掃描碼的鍵被按下了。在很多情況下軟體對這兩個鍵的處理是等同的。例如，對普通的“/”字元和數位鍵盤上灰色的“/”鍵，幾乎總是這樣的情況，在其他情況下，則可能需要區別這些字元。例如，許多非英語鍵盤配置對左右ALT鍵作不同的處理，來支援必須產生不同字元掃描碼的鍵。兩個ALT鍵都產生相同的掃描碼（56），但當右ALT按下時前面要加上EXTKEY碼。當返回EXTKEY碼時，esc標誌被置位元。在這種情況下，make\_break從switch內部返回，在正常返回前旁路了在每種情況下把esc置為0（第13295行）這最後一步。這使得esc只對緊接著收到的掃描碼有效。如果你熟悉PC鍵盤在普通應用下的各種複雜情況，那麼你也會對這種情況感到熟悉，但也會有一點奇怪，因為PC BIOS不允許讀一個ALT鍵的掃描碼並且為擴展掃描碼返回一個不同的值，而MINIX卻允許這樣做。

set\_leds（第13303行）打開或關閉指示PC鍵盤上的Num Lock、Caps Lock或Scroll Lock鍵是否被按下的燈。一個控制位元組LED\_CODE被寫到輸出埠通知鍵盤寫到埠的下一個位元組是指示燈控制位元組，三個指示燈的狀態用該位元組中的三位元進行編碼。下面兩個函數支援這個操作。kb\_wait（第13327行）被調用來決定鍵盤是否已準備好接受一個命令序列；kb\_ack（第13343行）被調用來檢驗該命令是否被回應。這些命令都使用忙等待，即持續地讀直到出現一個期望的掃描碼。對於處理大多數的I/O操作，這並不是一種值得推薦的技術。但是打開或關閉鍵盤上的指示燈並不經常發生，所以如果效率稍低一些並不會浪費很多時間。注意這兩個函數也都可能會失敗，如果發生了這種情況，可以從函數的返回碼判斷出來。但是設置鍵盤上的燈並不是很重要的工作，並不值得這兩個調用去檢查它的返回值，所以set\_leds只是直接繼續執行。

因為鍵盤是控制台的一部分，所以它的初始化常式kb\_init（第13359行）從console.c的scr\_init中被調用，而不是直接從tty.c的tty\_init中調用。如果使能了虛擬控制台（即include/minix/config.h中的NR\_CONS大於1），則對每個邏輯控制台調用一次kb\_init。在第一次之後，kb\_init中唯一對於附加控制台必不可少的部分就把kb\_read的位址設置到tp->tty\_devread中（第13367行），但重複函數的其他部分並不會有害。kb\_init的餘下部分初始化一些變數，打開鍵盤上的指示燈，並掃描鍵盤以確保沒有殘餘的已讀入的擊鍵。一切都初始化完畢後，它調用put\_irq\_handler，然後接著調用enable\_irq，所以當一個鍵按下或鬆開時將執行kbd\_hw\_int。

下面三個函數都非常簡單。kbd\_loadmap（第13392行）幾乎微不足道。它由tty.c的do\_ioctl調用把一個鍵位映射表從使用者進程空間拷貝到內核空間，覆蓋由在keyboard.c開頭包含的鍵位映射表原始檔案編譯成的默認映射表。

func\_key（第13405行）從kb\_read中調用以判斷是否按下了意味著局部處理的特殊鍵。圖3-47總結了這些鍵和它們的作用。調用的代碼可以在幾個檔中找到。F1和F2啟動dmp.c中的代碼，我們將在下一節中討論。F3啟動console.c中的toggle\_scroll，也在下一節討論。而CF7、CF8和CF9掃描碼將引起對tty.c中sigchar的調用。當在MINIX中加入網路工作時，一個附加的case，用來檢測F5掃描碼，被添加進來以顯示乙太網的狀態。除此以外，還有大量的其他掃描碼可以用來從控制台啟動其他的調試資訊或特殊事件。

scan\_keyboard（第13432行）在硬體介面層工作，從I/O埠讀寫位元組。從第13440行到13442行的序列通知鍵盤控制器讀入了一個字元，該序列讀入一個位元組，把它的最高有效位置為1再寫一次，再把該位重置為0再重寫一次。這樣就防止了後繼的讀操作讀入相同的資料。在讀鍵盤時沒有狀態檢測，但在任何情況下都應該沒有問題，因為scan\_keyboard只在中斷時被調用，來自kb\_init的異常調用會清除任何不需要的東西。

圖 3－47 func\_key()檢測的功能鍵。

keyboard.c中的最後一個函數是wreboot（第13450行）。如果是由於系統混亂被引用，它為用戶提供了用功能鍵顯示調試資訊的機會。第13478行到13487行的迴圈是使用忙等待的另一個例子。它重複地讀鍵盤直到鍵入了一個ESC。當然沒有人可以斷言在崩潰後等待命令重啟時需要一種更有效的技術。在迴圈內部，func\_key被調用以提供獲得有助於分析崩潰原因的資訊的可能。我們將不再討論返回到引導監控程序的細節。這些細節非常依賴於硬體並且與作業系統沒有多少關係。

3.9.6 顯示驅動程式的實現

如果有足夠的記憶體，IBM PC顯示器可以被配置為幾個虛擬終端。這一節中我們將探討控制台的設備相關代碼。我們還要研究使用鍵盤和顯示器低層服務的調試轉儲常式（debug dump routine）。這些常式支援與控制台使用者有限的交互，即使在MINIX系統的其他部分不工作時。它們還能在系統幾乎完全崩潰之後提供有用資訊。

對控制台輸出到PC記憶體映象螢幕的依賴於硬體的支援在console.c中。console結構在第13677行到13693行中定義。在某種意義上，這個結構是tty.c中定義的tty結構的擴展。在初始化時，控制台的tty結構的tp->tty\_priv域被賦予一個指向自己的console結構的指標。console結構中的第一項是一個指回對應的tty結構的指標。console結構中的元素對應於一個視頻顯示器：記錄游標的行列位置的變數，顯示記憶體的起始位址和界限，由控制器晶片的基指標指向的記憶體位址，和游標的當前位址。其他的變數用來管理轉義序列。因為字元最初被接收時是8位元位元組的形式，必須和屬性位元組組合以16位元字的形式傳送到視頻記憶體，所以在c\_ramqueue中建立了一個待傳送的塊，這個塊是一個足夠大的陣列，用來存放一整行80列16位元的字元-屬性對。每個虛擬控制台需要一個console結構，存放在陣列cons-table中（第13696行）。和處理tty與kb\_s結構一樣，我們一般通過一個指標，例如cons->c\_tty，來引用console結構中的元素。

cons\_write函數的位址存放在每個控制台的tp->tty\_devwrite入口處（第13729行）。它只從一個地方被調用，即tty.c的handle\_events中。console.c中的其他大部分函數的存在都是為了支援這個函數。當它在一個客戶進程進行了一個WRITE調用後被第一次調用時，待輸出的資料位於客戶進程的緩衝區中，可以用tty結構中的tp->tty\_outproc和tp->out\_vir域找到。tp->tty\_outleft域記錄了還有多少字元需要傳送，tp->tty\_outcum被初始化為0，表示什麼還沒有傳送。這是在進入cons\_write時的一般情況，因為在正常情況下，一旦該函數被調用，它就傳送原始調用請求的所有資料。不過，如果使用者希望進程能慢下來，以便在螢幕上察看資料，那麼他可以鍵入一個STOP（CTRL-S）字元，這將引起tp->tty\_inhibited標誌置位元。cons\_write在該標誌置位元時立即返回，即使WRITE還沒有結束。在這種情況下，handle\_events將繼續調用cons\_write，當tp->tty\_inhibited最終被重置時，用戶通過鍵入START（CTRL-Q）可以讓cons\_write繼續中斷的傳送。

cons\_write的唯一的參數是一個指向特定控制台tty結構的指標，所以第一件必須做的事就是初始化cons，這是一個指向這個控制台的console結構的指標（第13741行）。然後，因為handle\_events總是調用cons\_write，所以第一個動作就是檢測是否真的有事要做。如果沒有就快速返回（第13741行）。接下來就進入了第13751行到13778行的主迴圈。這個迴圈在結構上與tty.c中in\_transfer的主迴圈很相似。通過調用phys\_copy從客戶進程的緩衝區中獲得資料，並填入一個可以保存64個字元的局部緩衝區，指向資料來源的指標和計數器被更新，然後局部緩衝區中的每個字元連同稍後將由flush傳送到螢幕的屬性位元組一起被傳送到cons->c\_ramqueue陣列中。可以用不止一種方法來執行傳送，如圖3-39所示。可以調用out\_char來傳送每個字元，但可以預見如果字元是一個可見字元，沒有一個轉義序列，沒有超過螢幕寬度，並且cons->c\_ramqueue陣列沒有滿，那麼out\_char的特殊服務就變得多餘。該字元連同屬性位元組（由cons->c\_attr獲得）被直接放入cons->c\_ramqueue中，並且增加cons->c\_rwords（佇列的索引）、cons->c\_column（跟蹤螢幕上的當前列）和指向緩衝區的指標tbuf。這個把字元直接放入cons->c\_ramqueue中的動作對應於圖3-39中左邊的虛線。如果需要，就調用out\_char（第13766行到13777行）。它進行所有的登記，另外還在必要時調用flush最終把字元傳送到螢幕記憶體。只要tp->tty\_outleft指示仍有字元等待傳送並且tp->tty\_inhibited仍然沒有置位元，從使用者進程緩衝區到局部緩衝區再到佇列傳送就重複進行。當傳送停止時，不論是因為WRITE操作結束還是tp->tty\_inhibited被置位，就再次調用flush把佇列中餘下的字元傳送到螢幕記憶體。如果操作完成（檢測tp->tty\_outleft是否為0），就調用tty\_reply（第13784行到13785行）發送一條回答消息。

除了從handle\_events調用cons\_write外，待顯示的字元還由終端任務的硬體無關部分的echo和rawecho送往控制台。如果控制台是當前的輸出設備，就由tp->tty\_echo指標調用下一個函數cons\_echo（第13794行）。cons\_echo通過調用out\_char然後再調用flush完成所有的工作。來自鍵盤的輸入逐個字元地到達，而輸入的人希望看到回顯沒有可察覺的延遲，所以把字元送入輸出佇列並不能滿足要求。

接下來我們討論out\_char（第13809行）。它檢測是否有一個轉義序列，如果有，就調用parse\_escape並立即返回。否則，就進入一個switch語句，檢查與特殊情況的匹配：空字元，退格，響鈴符，等等。對這些情況的處理大都容易理解。換行和定位字元是最複雜的，因為它們涉及到游標在螢幕上的位置變化，並且可能需要卷屏。最後要檢測ESC碼。如果有，就置位元cons->c\_esc\_state標誌（第13871行），對out\_char的調用轉換為對parse\_escape的調用，直到序列結束。最後，對可列印字元進行缺省處理。如果超出了螢幕寬度，則螢幕可能需要卷屏，flush將被調用。在一個字元被放入輸出佇列前，要檢測佇列是否已滿，如果是，就調用flush。把一個字元放入佇列需要進行我們前面在cons\_write中看到的記帳。

下一個函數是scroll\_screen（第13896行）。scroll\_screen處理一般的在螢幕最後一行滿時的向上卷屏，以及向下卷屏，在游標定位命令試圖把游標移到螢幕最頂行之外時發生。對每個方向的捲動，有三種可能的處理方法。它們要用來支持不同的視訊卡。

我們將研究向上捲動的情況。開始時，chars被賦予螢幕尺寸減去一行的值。軟體卷屏由一個單獨的對vid\_vid\_copy的調用完成，把chars個字元向記憶體低端移動，移動的距離是一行中的字元數。vid\_vid\_copy是可回卷的，即如果請求移動的區塊溢出了賦予視頻顯示記憶體的上邊界，就從區塊的低端取出溢出部分，把它移動到高於被移到低端部分的位址處，即把整個塊看作一個環形陣列。這個調用看起來簡單，但執行卻相當慢。即使vid\_vid\_copy是定義在klib386.s中的組合語言常式，這個調用也需要CPU移動3840個位元組，就算對於組合語言也是一項繁重的工作。

軟體卷屏永遠不會作為默認方法；只有在硬體卷屏不能工作或由於某種原因不能採用時才選擇軟體方式。一個原因可能是因為希望使用screendump命令把螢幕記憶體存入一個檔。當使用硬體卷屏時，screendump可能會產生不可預知的結果，因為螢幕記憶體的起始位址和顯示器可見部分的開始位置可能不一致。

在第13917行中wrap變數被檢測，它是一系列複合檢測的第一部分。對於較老的支援硬體卷屏的顯示器wrap為真，如果檢測失敗，那麼第13921行將進行簡單的硬體卷屏，視頻控制晶片使用的原始指標被更新為指向顯示器左上角顯示的第一個字元。如果wrap為FALSE，複合檢測將繼續測試卷屏操作將要移動的區塊是否溢出了分配給該控制台的記憶體界限。如果是，就再次調用vid\_vid\_copy執行一個回卷移動把區塊移動到控制台分配記憶體的起始處，並且原始指針被修改。如果沒有重疊，控制就被傳遞給舊的視訊控制器一直使用的簡單硬體卷屏方法。這一步包括調節cons->c\_org，然後把新的原點放入控制器晶片對應的寄存器中。這個調用後面還要調用一次，因為有一個清空螢幕最後一行的調用。

向下卷屏的代碼與向上卷屏很相似。最後，調用mem\_vid\_copy清空由new\_line指向的螢幕最後（或最頂部）一行。然後調用set\_6845把新的原點從cons->c\_org中寫到對應的寄存器中，flush將確保所有的改動在螢幕上可見。

我們已經好幾次提到flush（第13951行）。它使用mem\_vid\_copy把佇列中的字元傳送到視頻記憶體中，更新某些變數，並保證行和列的數值是合理的，例如，如果一個轉義序列試圖把游標移到一個負的列數的位置上，flush將調整它們。最後將計算游標應處的位置，並與cons->c\_cur比較。如果它們不一致，並且如果現在正在處理的視頻記憶體屬於當前的虛擬控制台，就調用set\_6845把控制器的游標寄存器設置為正確的值。

圖3-48繪出了對轉義序列的處理如何被表示為一個有限狀態自動機。這是由parse\_escape（第13986行）實現的，如果cons->c\_esc\_state不為0，它就在

圖 3－48 處理轉義序列的有限狀態機。

out\_char的開頭被調用。out\_char本身檢測到一個ESC並且設置cons->c\_esc\_state為1。當接收到下一個字元時，parse\_escape就在指向參數陣列起始位置的指標cons->c\_esc\_intro中放入一個‘\0’，把cons->c\_esc\_parmv[0]放入cons->c\_esc\_parmp中，並且參數陣列本身都置為0，準備進一步的處理。接著檢查ESC後面的第一個字元——有效的值為“[”或“M”。在第一種情況下，“[”被拷貝到cons->c\_esc\_intro中，狀態轉換到2。在第二種情況下，do\_escape被調用來執行這個動作，並且轉義狀態被重置為0。如果ESC之後的第一個字元不是有效值中的一個，它將被忽略，後繼的字元仍然正常顯示。

當檢測到一個ESC [序列時，輸入的下一個字元由狀態2的代碼處理。這裡有三種可能性。如果該字元是一個數值字元，那麼它的值被取出並與由cons->c\_esc\_parmp所指向的當前位置上的值乘以10相加，初始值為cons->c\_esc\_parmv[0]（被初始化為0）。轉義狀態沒有變化。這使得可以輸入一系列十進位數字並累積為一個很大的數值參數，雖然MINIX現在最大能識別的值為80，由把游標移到任意位置的序列使用（第14027行到14029行）。如果該字元為分號，那麼指向參數字串的指標就向前移動，這樣後繼的數位值可以在第二個參數中累積（第14031行到14033行）。如果想要改變MAX\_ESC\_PARMS來為參數分配一個大一些的陣列，也不需要改變這段代碼以在輸入額外的參數後累積額外的數位值。最後，如果字元既不是一個數位也不是分號，就調用do\_escape。

儘管MINIX能識別的轉義序列相對來說並不多，do\_escape（第14045行）卻是MINIX系統原始程式碼中較長的一個函數，。不過，這段代碼並不難理解。在執行了一個對flush的初始調用以保證視頻顯示被完全更新之後，有一個簡單的if選擇，這個選擇取決於緊接著ESC之後的字元是否是一個特殊的 控制序列引入符。如果不是，那麼只有一個有效的動作，即如果該序列是ESC M就把游標上移一行。注意對“M”的檢測是在一個switch中的缺省動作中完成，作為有效性檢查並等待其他ESC [格式的序列的附加部分。在許多轉義序列中檢查cons->c\_row變數來決定是否需要卷屏這個動作很典型。如果游標已經位於第0行，就用SCROLL\_DOWN來調用scroll\_screen；否則游標被上移一行。後者由對cons->c\_row減一併調用flush完成。如果發現了一個控制序列引入符，那麼就執行第14069行在else之後的代碼。這裡要對MINIX現在唯一能識別的控制序列引入符“[”進行檢測。如果序列有效，那麼在轉義序列中的第一個參數，或在沒有輸入參數時為0，被賦給value（第14072行）。如果該序列無效，那麼除了跳過接下來的switch（第14073行到14272行）並且在從do\_escape返回之前把轉義狀態重置為0外，什麼也不做。在我們更感興趣的序列有效的情況中，將進入switch語句。我們不討論所有的情況；我們只研究轉移序列決定的動作中具有代表性的幾種。

開始的五個沒有數值參數的序列由IBM PC鍵盤上的四個方向鍵和Home鍵產生。頭兩個ESC [A和ESC [B與ESC M很相似，除了它們可以接受一個數值參數並且可以向上或向下移動多於一行的距離，如果參數指定的移動超過了螢幕的界限，它們並不捲動螢幕。在這些情況下，flush捕獲移出界限的請求並把移動限制在對應的最後一行或第一行。下面兩個序列，ESC [C和ESC [D把游標向左右移動，它們也類似地受到flush的限制。當由方向鍵產生時，沒有數值變數，這樣就進行預設的移動一行或一列。

下一個序列ESC [H可以有兩個數值變數，例如，ESC [20;60H。這兩個參數指定的是絕對位置而不是相對位置，並且被從1計數的數值轉換為從0計數的數值。Home鍵產生默認的序列（無參數）把游標移動到（1，1）處。

下麵的兩個序列ESC [sJ和ESC [sK，清除全部螢幕或當前行的一部分，這將取決於輸入的參數。每種情況下都計算字元計數。例如，對於ESC [1J，count中保存著從螢幕的開始到游標位置的字元數，這個計數和一個可以是螢幕的起始位置cons->c\_org，或當前的游標位置cons->c\_cur的位置參數dst，被用作調用mem\_vid\_copy的參數。用一個參數調用這個過程將使得它用當前背景色填充指定的區域。

下面的四個序列插入或刪除在游標處的行或空格，它們的動作不需要詳細的解釋。最後一種情況ESC [nm（注意n代表一個數值參數，但“m”是一個文字）對cons->c\_attr起作用，即當寫到視訊記憶體時交錯存放在字元碼之間的屬性位元組。

下一個函數set\_6845（第14280行）被用來更新視頻控制晶片。6845具有16位的內部寄存器，一次可以對8位程式設計，寫一個寄存器需要四次I/O埠寫操作。Lock和unlock調用用來禁止中斷，如果允許序列被中斷的話可能會產生問題。圖3-49中給出了6845視頻控制晶片的部分寄存器。

圖 3－49 部分6845寄存器。

beep函數（第14300行）在必須輸出CTRL-G字元時被調用。它利用了PC內建的對聲音的支援，向揚聲器發送方波發出聲音。聲音的初始化代碼大多數是只有組合語言程式員才喜愛的I/O埠操作，要注意過程中有一段臨界區不能被中斷。代碼中更令人感興趣的部分是使用了時鐘任務設置鬧鐘的功能，這可以用來啟動一個函數。下一個常式stop\_beep（第14329行）的位址被放在送往時鐘任務的消息中。它在給定的時間後停止發聲並且重置用來防止其他對發聲常式的調用的beeping標誌。

scr\_init（第14343行）由tty\_init調用NR\_CONS次。每次調用的參數是一個指向tty結構的指標，它是tty\_table的一個元素。在第14354行和14355行，line被用作cons\_table陣列的索引， 它首先被計算出來，然後進行有效性檢測，如果有效，就用來初始化指向當前控制台表表目的指標cons。這裡可以用指向設備的主tty結構的指標來初始化cons->c\_tty，並且tp->tty\_priv可以指向該設備的console\_t結構。接著，kb\_init被調用來初始化鍵盤，然後建立指向設備指定常式的指標，tp->tty\_devwrite指向cons\_write，tp->tty\_echo指向cons\_echo。第14368行到14378行取出CRT控制器基址寄存器的I/O位址，決定視頻記憶體的位址和大小，並根據使用的視訊控制器的類型設置wrap標誌（用來確定如何卷屏）。第14382行到14384行在通用描述元表中初始化視頻記憶體的段描述符。

接下來是虛擬控制台的初始化。每次調用scr\_init時，參數是不同的tp值，這樣第14393行到14396行就用不同的line和cons為各虛擬控制台提供可用視頻記憶體的各自的部分。接著每個螢幕被清空，準備好開始，最後控制台0被選為第一個活動控制台。

console.c中餘下的常式都很短並且簡單，我們將很快地說明一下。putk（第14408行）前面已經提過了。它為任何連結到內核映象裡需要服務的代碼列印一個字元而不用通過FS。toogle\_scroll（第14429行）的動作與它的名稱一樣，它切換確定使用硬體還是軟體卷屏的標誌。它還在當前游標位置顯示一條資訊來確認已選擇的模式。cons\_stop（第14442行）重新初始化控制台為重啟監控程序指定的狀態，優先于關機或重啟。cons\_org0（第14456行）只有在F3鍵強制切換卷屏模式或準備關機時使用。select\_console（第14482行）選擇一個虛擬控制台。它用新的索引調用，並調用set\_6845兩次使視訊控制器顯示視頻記憶體的正確部分。

最後兩個常式非常依賴於硬體。con\_loadfont（第14497行）把一個字體裝入圖形卡，以支持IOCTL TIOCSFON I/O操作。它調用ga\_program（第14540行）執行一系列對I/O埠的寫使得一般不可被CPU定址的視訊卡字體記憶體變為可見。然後調用phys\_copy把字體資料拷貝到這塊記憶體區，並且引用另一個序列把圖形卡設置回一般操作模式。

調試轉儲

在終端任務中我們最後要討論的一組過程原來只是為了在調試MINIX時暫時使用的。當不再需要時，它們可以被刪除，但很多用戶發現保留它們很有用。在修改MINIX時，它們特別有用。

如同我們已經看到的，在kb\_read的開始調用func\_key來檢測用來控制和調試的掃描碼。在檢測到F1和F2鍵時調用的轉儲常式位於dmp.c中。首先，p\_dump（第14613行）為所有的進程顯示基本的處理資訊，包括在按下F1鍵時顯示的記憶體使用資訊。第二，map\_dump（第14660行）在按下F2時提供更詳細的記憶體使用資訊。proc\_name（第14690行）通過檢查進程名支援p\_dump。

由於這段代碼完全包含在內核二進位碼本身之中。並且不作為一個使用者進程或任務運行，所以即使在一個嚴重的系統崩潰之後，它也常常能繼續正確地工作。當然，這些常式只能從控制台存取。轉儲常式提供的資訊不能被重定向到一個檔或其他設備中，所以硬拷貝或經由網路連接使用並不是可選項之一。

我們建議向試圖擴充MINIX的第一步最好是擴展轉儲常式以在你想要擴充的方面提供更多的資訊。

3.10 MINIX中的系統任務

把檔案系統和存儲管理器服務進程放在內核以外的一個後果就是有時它們擁有內核需要的資訊。不過這種結構禁止了它們把資訊寫入到內核表中。例如，系統調用FORK由存儲管理器處理。當生成一個新進程時，為了能夠進行調度，內核必須知道新進程的資訊。存儲管理器是如何通知內核的呢？

解決這個問題的辦法是使用一個內核任務通過消息機制與檔案系統和存儲管理器通信，而且它對所有內核表擁有存取權。這個任務被稱為系統任務，處於圖2-26中的第二層，它的作用與我們在本章中研究過的任務相似。唯一的區別在於它不控制任何I/O設備。但是，和I/O任務一樣，它實現了一個介面，但在這種情況下不是面向外部世界，而是面向系統中大部分的內部元件。它和I/O任務具有相同的特權，並和它們一起被編譯到內核映象中，在這裡研究它比在其他章節中更合適。

系統任務接受19種消息，如圖3-50所示。系統任務的主程序sys\_task（第14837行）與其他任務具有相似的結構。它接收一條消息，分發給合適的服務過程，然後發送一條回答消息。我們將觀察每一條消息和它的服務過程。

SYS\_FORK消息被存儲管理器用來通知內核產生了一個新的進程。內核需要瞭解這條消息以進行調度。該消息包含進程表內部的對應于父親和孩子的插槽號，存儲管理器和檔案系統也有進程表，在三個表中表目k指的都是同一個進程。這樣，存儲管理器就可以只指定父親和孩子插槽號，而內核會知道指的是那些進程。

圖 3－50 系統任務接收的消息類型。

過程do\_fork（第14877行）首先檢查存儲管理器送給內核的是否是無用的東西。檢測使用了一個宏isok-susern，它在proc.h中定義，用來檢測父進程和子進程表表目是否有效。在system.c中絕大部分服務過程都要進行類似的檢測。這稍有一點偏執狂，但保持內部檢查的一致性並沒有壞處。do\_fork把父進程的進程表表目拷貝到子進程的槽中。這裡需要進行一些調整。子進程不受父進程的任何未確定信號的影響，並且子進程不繼承父進程的跟蹤狀態。當然，所有子進程的帳號資訊被置為0。

在一個FORK調用之後，存儲管理器為子進程分配記憶體。內核必須知道子進程位於記憶體何處以在運行子進程時能正確設置段寄存器。SYS\_NEWMAP消息允許存儲管理器傳給內核任何進程的存儲映象。該消息也可以在一個BRK系統調用改變了映象之後使用。

這條消息由do\_newmap（第14921行）處理，它必須首先把新的映象從存儲管理器的位址空間中拷貝出來。映象本身並不包含在消息中因為它太大了。在理論上，存儲管理器可以通知內核映象位於位址m處，而m會是一個非法位址。實際上存儲管理器並不這樣做，但內核仍然要檢查。映象被直接拷貝到取得新映象的進程的進程表表目的p\_map中。對alloc\_segments的調用將從映象中提取出資訊並把它裝入保存段寄存器的p\_reg中。這並不複雜，但細節將依賴於處理器，因此被分離在一個獨立的函數中。

SYS\_NEWMAP消息更經常地在MINIX系統的普通操作中使用。另一條類似的消息SYS\_GETMAP只在檔案系統初始啟動時使用。這條消息需要從內核向存儲管理器反向傳送進程映象資訊。它由do\_getmap（第14957行）執行。兩個函數的代碼很相似，只是在調用phys\_copy時源和目的參數交換了位置。

當一個進程執行EXEC系統調用時，存儲管理器為它建立一個新的包含參數和環境的堆疊。它使用SYS\_EXEC把堆疊指標傳給內核，該消息由do\_exec（第14990行）處理。在對一個有效進程的一般檢測後，要檢查消息中的PROC2域。這裡該域被用作指示進程是否被跟蹤的標誌，與標誌一個進程無關。如果正處於跟蹤之下，就調用cause\_sig發送一個SIGTRAP信號給進程。這裡該信號的效果

與一般情況下不一樣，正常情況下它將終止一個進程並引起一個內核的轉儲顯示。在存儲管理器中，所有送往被跟蹤進程的信號除SIGKILL外都被截獲並暫停被跟蹤的進程以便偵錯工具能控制它下一步的執行。

EXEC調用會引起一些輕微的不正常。產生該調用的進程向存儲管理器發送一條消息並且阻塞自己。在其他的系統調用中，回答消息將釋放進程。但EXEC調用沒有回答，因為新裝入的內核映象不需要回答消息。因此，do\_exec在第15009行自己釋放進程。下一行用防止可能競爭條件的lock\_ready函數使新存儲映象準備運行。最後，保存命令字串使得在使用者按下F1鍵顯示所有進程狀態時該進程可以被識別。

MINIX中可以用一個EXIT系統調用向存儲管理器發送一條消息來退出一個進程，也可以通過一個信號終止一個進程。在兩種情況下，存儲管理器通過SYS\_XIT通知內核。這項工作由do\_xit（第15027行）完成，它比你想像的要複雜得多。處理記帳資訊的方法很直接。如果有鬧鐘計時器，可以在它的頂部存入一個0來終止它。因為這個原因時鐘任務在一個計時器超時時總是檢查是否仍有需要處理的事件。do\_xit中比較富於技巧的部分是當進程被終止時它可能處於發送或接收排隊狀態。第15056行到15076行的代碼檢測這種可能性。如果在其他進程的訊息佇列中發現了已存在的進程，那麼它將被很小心地刪除。

和前一條稍稍複雜一些的消息比起來，SYS\_GETSP完全就是微不足道的了。它被存儲管理器用來找出某些進程的的當前堆疊指標的值。BRK和SBRK系統調用需要該值來判斷資料段和堆疊段是否衝突。這些代碼在do\_getsp中（第15089行）。

現在我們研究幾種只用於檔案系統的消息類型之一，SYS\_TIMES。實現TIMES系統調用時需要這條消息，它向調用者返回計帳時間。do\_times（第15106行）的所有任務就是把請求的時間放入回答消息。對lock和unlock的調用用來防止在存取時間計數器時可能發生的競爭。

存儲管理器或檔案系統都有發現一個錯誤以致於無法繼續執行操作的可能。例如，如果在首次啟動時檔案系統發現根設備的超級塊有致命的損壞，它將陷入混亂狀態並向內核發送一條SYS\_ABORT消息。超級用戶也可能強迫返回到啟動監控程序和/或使用reboot命令調用REBOOT系統調用重新開機。在任何一種情況下，系統任務執行do\_abort（第15131行），把指令拷貝到監控程序中，然後調用wreboot完成處理。

信號處理的大部分工作由存儲管理器完成，如果信號的發送者有權這樣做的話，它檢測接收信號的進程是否被允許捕獲或忽略信號，以及諸如此類的工作。不過，存儲管理器並不能真正引發需要向進程的堆疊中壓入一些資訊的信號。

POSIX之前的信號處理有一些問題，因為捕獲一個信號時恢復了對信號的預設響應。如果需要連續地處理後繼信號，那麼程式師將不能保證可靠性。信號是非同步的，並且第二個信號可能正好在處理被重新使能之前到達。POSIX風格的信號處理解決了這個問題，但付出的代價是較複雜的機制。舊的方法可以通過作業系統向進程的堆疊中推入一些資訊來實現，和一個中斷推入資訊很相似。這樣程式師就可以寫一個以返回指令結束的處理常式，彈出所需資訊來恢復執行。當接收到一個信號時，POSIX保存比上述做法多的資訊。然後在進程能繼續正在處理的工作前要執行一項另外的工作。這樣存儲管理器就必須向系統任務發送兩條消息來處理一個信號。這樣的好處是信號處理的可靠性高。

當一個信號被送往一個進程時，SYS\_SENDSIG消息被送往系統任務。該消息由do\_sensig處理（第15157行）。處理POSIX風格信號所需的資訊在一個包含處理器寄存器內容的sigcontext結構和一個包含信號如何被進程處理的sigframe結構中。這些結構都需要一些初始化，但do\_sensig的基本工作只是把所需的資訊放在進程的堆疊中，並調整進程的程式計數器和堆疊指標以便在下次調度程式允許進程執行時將能執行信號處理代碼。

當一個POSIX風格的信號處理常式完成了它的工作後，和在舊的處理風格中一樣，它不彈出被中斷進程將恢復執行的位址。編寫處理常式的程式師寫一條return指令（或等效的高階語言），但SENDSIG調用對堆疊的操作將引起return執行一個SIGRETURN系統調用。然後存儲管理器向系統任務發送一條SYS\_SIGRETURN消息。該消息由do\_sigreturn處理（第15221行），它把sigcontext結構拷貝回內核空間中，然後恢復進程的寄存器。被中斷的進程將在下次調度程式允許它運行時從被中斷的位址恢復執行，保留以前建立的任何特殊信號處理。

和本節中討論的大部分其他調用不一樣，SIGRETURN系統調用在POSIX中不是必須的。這是一個MINIX約定，它可以方便地在一個信號處理常式結束時初始化所需的處理。程式師不應該使用這個調用；它不能被其他作業系統識別，並且在任何情況下都不需要明確地提到它。

有些信號來自內核映象內部，或者在它們被送往存儲管理器前由內核處理。這其中包括原來來自任務的信號，如來自時鐘任務的鬧鐘，或由終端任務檢測到的由按鍵引起的信號，還有CPU檢測到的由異常（比如被0除或非法指令）引起的信號。原來來自檔案系統的信號也首先由內核處理。SYS\_KILL消息被檔案系統用來請求產生一個這樣的信號。這個名字可能會讓人有一點誤解。它與KILL系統調用的處理無關，而是用於普通進程的信號發送。這條消息由do\_kill（第15276行）處理，對消息的有效來源進行一般的檢查，然後調用cause\_sig實際把信號傳給進程。起源於內核的信號也由對該函數的調用傳送，它通過向存儲管理器發送一條KSIG消息來初始化信號。

無論何時存儲管理器完成了這些KSIG類型之一的信號處理，它將向系統任務送回一條SYS\_ENDSIG消息。該消息由do\_endsig（第15294行）處理，它將減少等待信號的計數，如果該計數減少到0，就重置進程的SIG\_PENDING位元。如果沒有其他標誌被置位元指示進程不能運行，就調用lock\_ready允許進程再次運行。

SYS\_COPY消息是最常用的一條消息。它被用來允許檔案系統和存儲管理器從使用者進程拷貝資訊。

當一個使用者進程執行READ調用時，檔案系統查看它的緩存中是否有所需的塊。如果沒有，它就向適當的磁片任務發送一條消息把該塊裝入緩存。然後檔案系統向系統任務發送一條消息通知它把該塊拷貝到使用者進程。在最壞的情況下，讀一個塊需要7條消息；在最好的情況下，需要4條消息。圖3-51畫出了這兩種情況。這些消息是MINIX系統開銷的來源，是高度模組化設計的代價。

圖 3－51 (a)讀一個塊的最壞情況需要七條消息。(b)讀一個塊的最好情況需要四條消息。

在沒有保護機制的8088中，可以很容易的欺騙檔案系統使它把資料拷貝到調用者的位址空間，但這就違背了設計原則。任何使用這種舊機器的對提高MINIX性能感興趣的人必須仔細研究這種機制以判斷為了獲得性能上的提高必須忍受多少不合適的行為。當然，這意外著在具有保護機制的Pentium類機器上這種提高性能的方法是不可能的。

處理SYS\_COPY請求是很直接的。它由do\_copy（第15316行）完成，由提取消息參數和調用phys\_copy以及其他的一些動作組成。

處理消息傳遞機制效率較低的一種方法是把多個請求包裝到一條消息中。SYS\_VCOPY消息完成這個工作。這條消息的內容是一個指向指定在記憶體中移動的多個資料塊的向量的指標。函數do\_vcopy（第15364行）執行一個迴圈，提取源和目的地址和塊長度並重複調用phys\_copy直到所有的拷貝結束。這與磁片設備基於一個請求處理多個傳輸的能力是相似的。

系統任務接受的消息類型還有另外幾種，大部分都很簡單。其中的兩個一般在系統啟動時使用。檔案系統發送一條SYS\_GBOOT消息請求啟動參數。在include/minix/boot.h中說明的一個結構bparam\_s允許MINIX在啟動之前可以指定系統組態的各個方面。do\_gboot（第15403行）函數完成這個操作，它只是從記憶體的一部分拷貝到另一部分。存儲管理器也在啟動時向系統任務發送一系列SYS\_MEM消息來請求可用區塊的基址和大小。do\_mem（第15424行）處理這個請求。

SYS\_UMAP消息被一個非內核進程使用來請求計算給定虛擬位址的實體記憶體位址。do\_umap（第15445行）通過調用umap執行這個動作，而umap是從內核中調用的處理這個轉換的函數。

我們討論的最後一條消息類型是SYS\_TRACE，它支援用於調試的PTRACE系統調用。調試不是一個基本的作業系統功能，但作業系統的支援可以使它變得簡單。在作業系統的説明下，一個偵錯工具可以檢查和修改由一個被檢查進程使用的記憶體，以及在被偵錯工具不運行時存儲在進程表中的處理器寄存器的內容。

一般地，一個進程一直運行到被阻塞以等待I/O或用完一定的時間。大部分CPU的設計都提供了使得一個進程可以只執行一條指令的方法，或者通過設置一個中斷點（breakpoint），使得進程只有在到達某一條指令時才被執行的方法。利用這些功能可以對程式進行仔細的分析。

使用PTRACE可以進行11種操作。其中只有一小部分可以完全由存儲管理器完成，對於大部分操作存儲管理器向系統任務發送一條SYS\_TRACE消息，系統任務再接著調用do\_trace（第15467行）。這個函數根據跟蹤操作實現了一條switch語句。這些操作一般都很簡單。在進程表中的一個的P\_STOP位被MINIX用來識別是否處於調試中，由停止進程的命令（case T\_STOP）設置，或者重置以重新開機該進程（case T\_RESUME）。調試依賴於硬體的支援，在Intel處理器中是由CPU的標誌寄存器中的一位元控制的。當該位被置位時，處理器只執行一條指令，然後就產生一個SIGMAP異常。和前面提到的一樣，當向一個進程發送一個信號時，存儲管理器就停止一個被跟蹤的程式。這個TRACEBIT由T\_STOP和T\_STEP命令操縱。可以用兩種方法來設置中斷點：或者用T\_SETINS命令用一條產生一個SIGTRAP的特殊代碼來替換一條指令，或者用T\_SETUSER命令來修改特殊的中斷點寄存器。對任何MINIX可以移植的系統，都可以用相似的技術來實現一個調試器，但移植這些函數需要研究特定的硬體。

do\_trace執行的大部分命令返回或修改在被跟蹤進程的正文或資料空間中的值，或者是它的進程表表目中的值，並且代碼的實現是很直接的。允許修改某些寄存器和CPU的標誌位元實在是太危險了，所以在處理T\_SETUSER命令的代碼中有許多檢查防止發生這種操作。

在system.c的最後是可以在內核的各個部分使用的幾個實用過程。當一個任務需要引發一個信號時（例如，時鐘任務需要引發一個SIGALRM信號，終端任務需要引發一個SIGINT信號），它就調用cause\_sig（第15586行）。這個過程在進程的進程表表目的p\_pending域中設置某一位元，然後檢查存儲管理器現在是否在等待來自ANY的消息，即它是否空閒並在等待下一個請求來處理。如果空閒，就調用inform通知存儲管理器處理該信號。

如上所述，inform（第15627行）只有在檢測到存儲管理器不忙時才被調用。除了從cause\_sig調用外，只要存儲管理器被阻塞並且有未處理的內核信號時，它就從mini\_rec中（在proc.c中）被調用。inform建立一個KSIG類型的操作，並把它發送到存儲管理器。調用cause\_sig的任務或進程在消息被拷貝到存儲管理器的接收緩衝區中後立即繼續運行。它不等待存儲管理器的處理；如果在普通的發送機制中，這將引起發送者被阻塞。不過，在它返回之前，inform調用lock\_pick\_proc，由它調度存儲管理器運行。由於任務具有比伺服器更高的優先順序，所以存儲管理器直到所有的任務被滿足之後才被運行。當任務結束時，將進入調度程式。如果存儲管理器是優先順序最高的可運行進程，那麼它將被運行並處理信號。

過程umap（第15658行）是一個通用的把虛擬位址映射為物理位址的過程。如同我們已經注意到的，它由為SYS\_UMAP消息服務的do\_umap調用。它的參數是一個指向虛擬位址待映射的進程或任務的進程表表目的指標，一個指定正文、資料、或堆疊段的標誌，虛擬位址本身，以及一個位元組計數器。這個計數器很有用因為umap要檢查以確保從虛擬位址開始的全部緩衝區都處於進程的位址空間中。為此，它必須知道緩衝區的大小。位元組計數器並不用於映射本身，而只用於這個檢查。所有從/向使用者進程空間拷貝資料的任務都使用umap計算緩衝區的物理位址。對於設備驅動程式如果能夠使用進程號作為參數而不是指向進程表表目的指標就可以得到umap的服務將更為方便。numap（第15697行）完成這個功能。它調用proc\_addr來轉換它的第一個參數，然後調用umap。

在system.c中定義的最後一個函數是alloc\_segments（第15715行）。它由do\_newmap調用。在初始化時它也被內核的main常式調用。這個定義十分依賴於硬體。它取出記錄在進程表表目中的段賦值並操作Pentium處理器使用的寄存器和描述符以在硬體層支援段保護。

3.11 小結

輸入/輸出經常被忽視，但它是一個重要的課題。任何作業系統中都有相當重要的部分與I/O操作有關。我們由研究I/O硬體開始，分析了I/O設備和I/O控制器的關係，這是軟體必須處理的問題。然後我們研究了I/O軟體的四個層次：中斷常式，設備驅動程式，設備無關I/O軟體，以及在使用者空間運行的I/O庫和周邊同作。

下面我們研究了鎖死的問題，以及如何處理它。鎖死發生在一組進程都擁有對某些資源的互斥存取權，並且每個進程還要求仍屬於該組中另一個進程的資源時。所有進程都被阻塞，沒有一個可以再次運行。鎖死可以通過結構化系統來使得它不會發生。例如，在任何時刻一個進程只允許擁有一個資源。通過檢測每個資源請求是否可能導致鎖死（不安全狀態）並否決或延遲那些可能引起麻煩的請求也可以避免鎖死。

MINIX中的設備驅動程式是作為嵌入在內核中的進程來實現的。我們研究了RAM磁片驅動程式，硬碟驅動程式，時鐘驅動程式，以及終端驅動程式。同步鬧鐘任務與系統任務不是設備驅動程式但在結構上非常相似。這些任務都有一個主迴圈取出請求並進行處理，逐漸送回回答消息報告發生的事件。所有的任務都位於相同的位址空間中。RAM磁片，硬碟，軟碟驅動器任務都使用一個相同的主迴圈拷貝，並共用相同的函數。不過，每個任務仍然是獨立的進程。幾種不同的使用系統主控台，串列線和網路連接的終端都由一個單一的終端任務支援。

設備驅動程式和中斷系統有許多不同的關係。能很快完成工作的設備如RAM磁片和記憶體映象顯示器根本不使用中斷。硬碟驅動器任務在任務代碼本身中完成了大部分工作，中斷處理常式只返回狀態資訊。時鐘中斷處理常式本身執行了許多登記操作，只有在處理常式不能處理某項工作時才向時鐘任務發送一條消息。鍵盤中斷處理常式緩衝輸入並且從不向它對應的任務發送消息，而是改變一個由時鐘中斷處理常式檢測的變數，時鐘中斷處理常式將向終端任務發送一條消息。

習 題

1 假設晶片技術的發展可以將完整的控制器放在一個便宜的晶片上，其中包含所有的匯流排存取邏輯。這對圖3-1的模型有什麼影響？

2 如果一個磁碟控制卡沒有內部緩衝區，一從磁片上收到位元組就將它們寫到記憶體中，請問這時交叉存取有用嗎？試討論之。

3 根據磁片的旋轉速率和幾何尺寸，對於軟碟和硬碟，在磁片本身與控制器的緩衝區間傳送的比特速率各為多少？這與其他形式的I/O（串列線和網路）如何進行比較？

4 一個雙交叉編址的磁片如圖3-4(c)所示，每磁軌有8個512位元組的磁區，旋轉速率為300rpm，假設磁頭臂已被正確地定位，並且把磁區0置於磁頭下需要旋轉1/2圈，那麼它要花多長時間才能依次讀完一條磁軌上的所有磁區？資料傳輸率是多少？同樣條件下，對於具有相同特性但無交叉編址的磁片上述問題如何回答？因為交叉編址而使資料傳輸速率降低多少？

5 多年前用於PDP—11的DM—11終端多工器，以七倍於串列傳輸速率的速度在每一終端線（半雙工）上採樣來檢查輸入的數位是0還是１，線上上每次採樣時間5.7微秒，則DM—11能夠支持多少條1200波特的終端線？

6 一個局域網以如下方式被使用：使用者發出一條系統調用，請求將資料包寫到網上，然後作業系統將資料拷貝到一個內核緩衝區中，再將資料拷貝到網路控制器板上，當所有資料都安全存儲于控制器中時，把它們在網上以10百萬位元/秒的速率傳送，在每一位被發送後一微秒，接收的網路控制器將它保存。當最後一位元到達時，目標CPU被中斷，內核將新到達的資料包拷貝到內核緩衝區中進行檢查。一旦它指明該資料包是發送給哪個使用者進程的，內核就將資料拷貝到該使用者進程空間。如果我們假設每一個中斷及其相關處理過程費時1毫秒，資料包有1024位元組（忽略頭標），拷貝一個位元組費時1微秒，將資料從一個進程注入另一個進程的最大速率是多少？假設發送者一直被阻塞直到接收端的工作完成並且返回一條確認消息。為簡便起見，假設返回確認消息的時間可以忽略不計。

7 什麼叫設備無關性？

8 以下的工作各在四個I/O軟體層的哪一層完成？

(a)為一個磁片讀操作計算磁軌、磁區、磁頭。

(b)維護一個最近使用的塊的緩衝。

(c)向設備寄存器寫命令。

(d)檢查使用者是否有權使用設備。

(e)將二進位整數轉換成ASCII碼以便列印。

9 為什麼印表機的輸出檔在列印前通常都周邊同作輸出在磁片上？

10 考慮圖3-8。假設在步驟(o)中C需要S而不是R，這將會導致鎖死嗎？如果既要求S也需要R呢？

11 仔細觀察圖3-11(b)。如果Suzanne再多請求一個單位，將會導致一個安全的狀態還是不安全的狀態？如果請求來自Marvin而不是Suzanne呢？

12 圖3-12中所有的軌跡都是垂直的或是水準的，你能想像出可能出現的對角線軌跡的情況嗎？

13 假設圖3-13中的進程A請求最後一台磁帶驅動器。這會導致鎖死嗎？

14 一台電腦有６台磁帶機被n個進程競爭，每個進程可能需要兩台磁帶機，那麼n為多少時，系統沒有鎖死的危險？

15 一個系統可以處於既不鎖死也不安全的狀態嗎？如果可以，舉出例子；如果不可以，請證明所有狀態均處於鎖死或安全兩種狀態之一。

16 一個使用信箱的分散式系統有兩條IPC原語：SEND和RECEIVE。後一個原語指定從哪個進程接收，如果該進程沒有可用消息，那麼即使消息可以等待其他進程。該原語也阻塞。不存在共用資源，但是進程因為其他原因需要經常通信。請討論可能會發生鎖死嗎？

17 在一個電子轉帳系統中，有數百個相同進程以如下的方式工作：每一進程讀取一個輸入行，指定一定數目的款項、貸方帳號、借方帳號，然後鎖定兩個帳號，傳送這筆錢，完成後在解鎖。由於許多進程並行運行，所以存在這樣的危險：鎖定x將無法鎖定y，因為y已被一個正在等待x的進程鎖定。設計一個方案來避免鎖死。在沒有完成交易處理前不要釋放帳號記錄。（換言之，鎖定一個帳號時如果另一個帳號被鎖定就立即釋放它的處理方法是不允許的。）

18 銀行家演算法在一個有m個資源類和n個進程的系統中運行。在m和n都很大的情況下，為檢查狀態是否穩定而進行的操作次數正比於manb。a和b的值為多少？

19 Cinderella和Prince要離婚，為分割財產，他們商定了以下演算法：每天早晨各人發函給對方律師要求財產中的一項。由於郵遞信件需要一天的時間，他們商定如果發現在同一天兩人要求了同一項財產，第二天他們將發信取消這一要求。他們的財產包括狗Woofer、Woofer的狗屋、金絲雀Tweeter和Tweeter的鳥籠。由於這些動物喜愛它們的房屋，所以又商定任何將動物和它們房屋分開的方案都無效，整個分配從頭開始。Cinderella 和Prince都非常想要Woofer。於是他們分別去度假，並且每人都用一台個人電腦處理這一談判工作。當他們度假回來，發現電腦仍在談判，為什麼？可能發生鎖死嗎？可能發生饑餓（永遠等待）嗎？

20 圖3-15的消息格式被用來向塊設備的驅動程式發送請求消息。如果可以，消息中的哪些欄位在字元設備中可以省略？

21 磁片請求以10、22、20、2、40、6、38柱面的次序到達磁碟機。尋道時每個柱面移動需要6毫秒，計算以下尋道時間：

(a)先到先服務。

(b)下一個最鄰近柱面。

(c)電梯演算法（起始移動向上）。

　　所有情況下磁頭臂起始都位於柱面20。

22 一個個人電腦銷售商在訪問阿坶斯特丹西南部的一所大學時，竭力推銷他的產品，聲稱他的公司投入很大的努力使他們的UNIX版本速度非常快。例如，他指出他們的磁片驅動程式使用電梯演算法，並將一個柱面內的多個請求按磁區次序排隊。學生Harry Hacker被打動並買了一台。回家後他寫了一個程式隨機地讀分佈在磁片上的10000個塊，令他驚訝的是，他測試的結果與先到先服務方式得出的結果一樣。商人是否在騙人？

23 一個UNIX進程包含兩部分 — 使用者部分和核心部分，核心部分類似於子常式還是共行常式（coroutine）？

24 一個電腦的時鐘中斷處理常式每一滴答要佔用2毫秒（包括所有的切換開銷），時鐘以60Hz的頻率運行，那麼CPU時間用於時鐘處理的部分是多少？

25 教材中給出了兩個監視計時器的例子：定時軟碟馬達的啟動和允許硬拷貝終端上使用回車，請給出第三個例子。

26 為什麼RS232終端是中斷驅動，而記憶體映象終端不是中斷驅動？

27 考慮一個終端如何工作：驅動程式輸出一個字元，然後阻塞。當字元被列印完畢後，發生一條中斷並且傳送一條消息給阻塞的驅動程式，它輸出下一個字元然後再次阻塞。如果傳送一條消息，輸出一個字元，然後阻塞所需時間為4毫秒，那麼這一方法在串列傳輸速率為110的傳輸線上能很好地工作嗎？在4800波特的傳輸線上呢？

28 一個位映象終端包含1200\*800個象素。為了滾動一個視窗，CPU（或者控制器）必須移動上面的文本的所有行，這通過將其所有比特從視頻RAM的一個部分拷貝到另一個部分實現。如果一個視窗高66行，寬80個字元，（共5280個字元），每個字元寬8象素，高12象素。假設將字元輸出到螢幕需要50微秒，如果以每位元組500納秒的速率進行拷貝，則滾動整個視窗需要多長時間？如果每行都是80個字元，那麼終端的串列傳輸速率是多少？對於每象素占4比特的彩色終端，計算其串列傳輸速率（假設現在將一個字元輸出到螢幕需要200微秒）。

29 作業系統為什麼提供轉義字元，例如MINIX中的CTRL-V。

30 MINIX驅動程式在接收到一個DEL（SIGKILL）字元後終端當前排隊的所有輸出，為什麼？

31 許多RS232終端具有這樣的轉義序列：刪除當前傳輸線並將其下邊的所有線上移，你認為在終端內部如何實現這一特性？

32 在最初IBM PC的彩色顯示器上，除了CRT電子束垂直回掃之後的任何時間向視頻RAM中寫資料都會導致螢幕上的“雪花”。一個螢幕為25x80字元，每個佔用8x8象素。每行的640象素在電子束的一次水準掃描中繪出，包括水準會掃，該過程歷時63.6微秒，螢幕每秒鐘刷新60次，每次刷新均需要一個垂直回掃以使電子束回到螢幕頂端。該過程中可供寫視頻RAM的時間占多少？

33 為IBM彩色顯示器或其他位元映象顯示器寫一個圖形驅動程式，它應該接收如下命令：設置並清除單個象素、移動螢幕上的矩形、以及你認為有趣的其他特性。使用者程式與驅動程式的介面方式為打開/dev/graphics檔並向其中寫入命令。

34 修改MINIX軟碟驅動程式以完成每次一道高速緩衝。

35 實現一個軟碟驅動程式使其作為一個字元設備，而非塊設備來工作，以便旁路掉檔案系統的塊高速緩衝。在此方式下，使用者可從磁片讀大量資料，這些資料通過DMA方式直接傳到使用者空間，以大幅度提高性能。該驅動程式主要針對那些需要不通過檔案系統，而是機械地從磁片讀取比特的程式，例如檔案系統檢查工具。

36 實現MINIX中缺少的UNIX PROFIL系統調用。

37 修改終端驅動程式，使得除了一個用來刪除前一個字元的特殊鍵外，另外再有一個鍵能夠刪除前一個單詞。

38 MINIX系統中已經加入了一種新的支援可移動介質的硬碟設備。該設備在更換介質時必須旋轉加速，而且這個時間相當長。預期在系統運行過程中介質更換很頻繁。這樣at\_wini.c檔中的waitfor常式便無法滿足需要。請設計一個新的waitfor常式，若所等待的比特模式在1秒鐘的忙等待後仍未出現，則進入一個階段，該階段下磁片任務將睡眠1秒鐘，測試埠，並回去繼續睡眠1秒鐘，直到或者出現相應的比特模式，或者到達預先設置的TIMEOUT超時。

第四章 記憶體管理

記憶體是一種必須仔細管理的重要資源。雖然現在一台普通家用電腦的記憶體容量可能是60年代早期全世界最大的電腦IBM 7094的五倍以上，但是程式長度的增長速度和記憶體容量的增長一樣快。用類似于帕金森定律的話來說： “記憶體有多大，程式就會有多長”。在這一章中我們將討論作業系統是怎樣管理記憶體的。

在理想的情況下，每個程式師都會喜歡的是無窮大、快速並且內容不易變（即掉電後內容不會丟失）的記憶體，同時又希望它是是廉價的。但不幸的是當前的技術沒有能夠提供這樣的記憶體，因此大部分的電腦都有一個記憶體層次結構，它由少量的非常快速、昂貴、易變的的快取記憶體（cache），若干百萬位元組的中等速度、中等價格、易變的主記憶體（RAM），和數百兆或數千百萬位元組的低速、廉價、不易變的磁片組成。作業系統的工作就是協調這些記憶體的使用。

作業系統中管理記憶體的部分稱為存儲管理器，它的任務是跟蹤哪些記憶體正在被使用、哪些記憶體空閒，在進程需要時為它分配記憶體，使用完畢後釋放記憶體，並且在主存無法容納所有進程時管理主存和磁片間的交換。

在這一章中我們將討論許多不同的記憶體管理方案，從非常簡單的到高度複雜的。我們從最簡單的存儲管理系統開始，然後逐步過渡到更加精密的系統。

4.1基本的記憶體管理

存儲管理系統可以分為兩類：在運行期間將進程在主存和磁片之間移動的系統（交換和分頁）和不移動的系統。後一種比較簡單，因此我們首先討論。隨後在本章的後半部分我們將討論交換和分頁。在本章中讀者應該自始至終清醒地認識到：交換和分頁在很大程度上是由於缺少足夠的主存以同時容納所有的進程而引入的。隨著主存越來越便宜，選擇某種記憶體管理方案的理由也許會變得過時，除非程式變大的速度比記憶體降價的速度還要快。

4.1.1沒有交換和分頁的單道程式

最簡單的記憶體管理方案是同一時刻只運行一道程式，應用程式和作業系統共用記憶體。這種方案的三個變種如圖4-1所示。作業系統可以位於主存最低端的隨機存取記憶體（RAM）中，如圖4-1(a)所示；它也可以位於主存最高端的唯讀記憶體（ROM）中，如圖4-1(b)所示；還可以讓設備驅動程式位於記憶體高端的ROM中而讓作業系統的其他部分位於低端的RAM中，如圖4-1(c)所示。例如小型的MS-DOS系統使用的就是最後一種模型。在IBM PC電腦中，系統位於ROM中的部分稱為基本輸入輸出系統（BIOS）。

圖4-1 在一個作業系統一個使用者進程時三種簡單的記憶體組織方法，其他方法也是存在的。

當這樣組織系統時，同一時刻只能有一個進程在記憶體中運行。一旦使用者輸入了一個命令，作業系統就把需要的程式從磁片拷貝到記憶體中並執行它；在進程運行結束後，作業系統顯示出一個提示符並等待新的命令。當收到新的命令時它把新的程式裝入記憶體，覆蓋掉原來的程式。

4.1.2固定分區的多道程序

雖然單道程式常常用於帶有簡單作業系統的小型電腦上，但我們往往更加希望同時能有多個進程同時運行。在分時系統中，允許多個進程同時在記憶體中，這意味著當一個進程因為等待I/O結束而阻塞時，其他的進程可以利用CPU，因而提高了CPU的利用率。即使在個人電腦上，同時運行兩個或多個進程也常常是很有用的。

實現多道程序的最容易的辦法是把主存簡單地劃分為n個分區（可能不相等），分區的劃分可以在系統啟動時手工完成。

當一個作業到達時，可以把它放到能夠容納它的最小的分區的輸入佇列中。因為在這種方案中分區大小是固定的，一個分區中未被作業使用的空間就白白浪費掉了。圖4-2(a)所示的就是這種固定分區、各分區有自己獨立的輸入佇列的系統。

圖4-2 (a)各分區具有獨立輸入佇列固定記憶體分區。(b)僅有單個輸入佇列的固定記憶體分區。

把輸入作業排成多個佇列時，在大分區的佇列為空而小分區的佇列為滿的情況下，其缺點就變得很明顯，如圖4-2(a)中的分區1和3所示。另一種方法如圖4-2(b)所示，只使用一個佇列，當一個分區空閒時，選擇最靠近佇列頭可以被該分區容納的作業裝入其中運行。因為我們不希望為了一個小作業而浪費一個大分區，所以另一個策略是搜索整個輸入佇列找出該分區能容納的最大的作業，這種演算法認為不值得給小作業一個完整的分區，然而通常我們更加期望給小作業（假設是交互作業）最好的服務，而不是最差的。一個解決方法是至少保留一個小分區，這樣就不必為了使小作業運行而為其分配大的分區。另一個方法是採用這樣一個規則：一個可運行的作業最多只能夠被跳過k次。一個作業每被跳過一次就得一分，當它得到k分時它就不能再被跳過了。

這種由操作員在早晨設置好隨後就不能再被改變的固定分區的系統曾在IBM大型機的OS/360中使用了許多年，它被稱為MFT（具有固定數目任務的多道程序，或OS/MFT），它易於理解也易於實現：輸入作業被送入佇列排隊直到有合適的分區可用，隨後作業被裝入分區運行直到它結束。現在只有極少數作業系統，如果還有的話，支援這種模式。

重定位和保護

多道程序引入了兩個必須解決的問題—重定位和保護。從圖4-2可以清楚地看出不同的作業將在不同的位址區間運行。在一個程式被連結時（即把主程序、使用者編寫的常式、庫常式結合到同一個位址空間中），連結器必須知道程式將在主存的什麼位址開始運行。

例如，假設程式的第一條指令是調用在連結器產生的二進位檔案中絕對位址為100的一個過程。如果程式被裝入分區1，這條指令跳轉的目的地址將是絕對位址100，這會造成混亂，因為該位址在作業系統的內部。其實真正應該被調用的地址是100K+100。如果程式被裝入分區2，它就應該去調用200K+100，等等。這就是重定位問題。

一個可能的解決方法是在程式裝入主存時直接修改指令，裝入分區1的程式在每個位址上加100K，裝入分區2的程式在每個位址上加200K，等等。為了在裝入時能這樣重定位，連結器必須在二進位程式中包含點陣圖或鏈表，由他們指明那些程式字是需要進行重定位的位址，那些是操作碼、常數和其他不能進行重定位的元素。OS/MFT就是這樣工作的，一些微機也是這樣工作的。

在裝入時重定位並沒有解決保護問題，一個惡意的程式總可以生成一條新指令並跳轉到這條指令執行。因為在這個系統中使用的是絕對位址而不是相對於某個寄存器的位址，沒有辦法能阻止程式生成讀或寫主存任何位置的指令。在多使用者系統中，我們不希望一個進程讀寫屬於另一個使用者的主存空間。

IBM採用的保護360機器的辦法是將主存劃分為2K位元組的塊並為每個塊分配4位的保護碼。PSW中包含一個4位的金鑰，若運行進程試圖對保護碼不同於PSW中金鑰的主存進行訪問，則由硬體引起一個陷入。因為只有作業系統能夠修改保護碼和金鑰，這種辦法能有效地阻止使用者進程干涉其他進程或作業系統本身。

另一個既針對重定位又針對保護問題的解決方法是在機器中設置兩個專門的寄存器，稱為基址和界限寄存器。在一個進程被調度到時，它的分區的起始位址被裝入基址寄存器，分區的長度被裝入界限寄存器。進程產生的每一個位址在訪問主存前被自動加上基址寄存器的內容，因此如果基址寄存器是100K，不用修改指令，一條 CALL 100 指令就被有效地轉換為一條CALL 100K+100指令。指令還被自動地用界限寄存器進行檢查以確保他們沒有試圖訪問當前分區以外的位址。基址和界限寄存器受到硬體保護，以防止使用者程式修改他們。

CDC 6600—世界上第一台巨型機—使用了這個方案。用於初期IBM PC的Intel 8088 CPU使用了這個方案的一個較弱的版本—有基址寄存器，但沒有界限寄存器。從286開始，採用了一種更好的方案。

4.2交換

在批次處理系統中，把主存組織為固定的分區是簡單而且高效的，每個作業在排到佇列頭時被裝入一個分區，它停留在主存中直到運行完畢。只要有足夠的作業能被保持在主存中以使CPU始終處於忙的狀態，那麼就沒有理由使用任何更加複雜的方案。

但在分時系統或面向圖形的個人電腦中情形就不同了，有時會沒有足夠的主存以容納所有當前活動的進程，多出的進程必須被保存在磁片上並動態地調入主存運行。

在硬體支援下，有兩個通用的記憶體管理方法可以使用。最簡單的策略稱為交換（swapping），它把各個進程完整地調入主存，運行一段時間，再放回到磁片上；另一種策略稱為虛擬記憶體（virtual memory），它使進程在只有一部分在主存的情況下也能運行。下面我們將先討論交換系統，在4-3中我們將討論虛擬記憶體。

交換系統的操作如圖4-3所示，開始時只有進程A在主存，隨後進程B和C被創建或從磁片上被調入，在圖4-3(d)中A結束了或被交換到了磁片上，然後D進入，接著B離開，最後E進入。

圖4-3 記憶體分配情況隨著進程進出記憶體而變化，陰影表示的區域是未使用的記憶體。

圖4-2所示的固定分區與圖4-3所示的可變分區的主要區別是：在後者中分區的數量、位置、大小隨著進程的出入是動態變化的，而在前者中他們是固定不變的。這種可變分區不再受固定分區可能太大或太小的約束，從而提高了主存的利用率，但它也使記憶體的分配、釋放和對各個區塊的跟蹤更加複雜。

當交換在主存中生成了多個空洞時，可以把所有的進程向下移動至相互靠緊，從而把這些空洞結合成一大塊，這種技術稱為記憶體緊縮（memory compaction）。我們通常不進行這個操作，因為它需要大量的CPU時間，例如在一個有32M主存，每微秒可以拷貝16個位元組的電腦上把全部記憶體緊縮一次需要兩秒鐘。

一個值得關心的問題是在一個進程被創建或換進時應該為它分配多大的記憶體。如果進程創建時的大小是固定的並且不會改變，那麼分配是很簡單的：完全根據需要的大小進行分配。

然而如果進程的資料段可以增長，例如在許多程式設計語言中都允許動態地從堆中分配記憶體，那麼進程一旦試圖增長時問題就出現了，如果該進程鄰接著一個空洞就可以把這個空洞分配給它；然而如果進程鄰接的是另一個進程，則需要增長的進程或者不得不被移動到記憶體中一個足夠大的空洞中去，或者必須把一個或多個進程交換出去以生成一個足夠大的空洞。如果一個進程不能在記憶體中增長並且磁片上的交換區已經滿了，那麼這個進程必須等待或被殺死。

如果大部分進程在運行時都要增長，那麼為了減少進程因為所在的記憶體區域不夠而引起的交換和移動所帶來的開銷，可以採用的一種方法是：在進程被換進或移動時為其分配一點額外的記憶體。當然，在進程被換出到磁片上時應該只交換進程實際使用的記憶體中的內容，將額外的記憶體交換出去純粹是浪費。在圖4-4(a)中我們可以看到一個已經為兩個進程分配了增長空間的記憶體配置。

圖4-4 (a)為能夠增長的資料段預留空間。(b)為能夠增長的資料段和堆疊段預留空間。

如果進程有兩個可增長的資料部分，例如一個供動態分配和釋放的變數使用的作為堆的資料段和一個存放普通區域變數和返回位址的棧段，那麼可以使用另一種安排，如圖4-4(b)所示。在這個圖中我們可以看到所示進程的棧段在進程所占記憶體的頂端並向下增長，緊接在正文段後面的資料段向上增長，處於這兩個段之間的記憶體，他們都可以使用，如果用完了，則這個進程或者必須被移動到足夠大的空洞中，或者交換出記憶體直到記憶體中有足夠的空間，或者被殺死。

4.2.1使用點陣圖的記憶體管理

動態分配的記憶體必須由作業系統管理。一般來說有兩種方法跟蹤記憶體的使用情況：點陣圖和自由鏈表。本節和下一節將逐個討論這兩種方法。

在使用點陣圖方法時，記憶體被劃分為可能小到幾個字或大到幾千位元組的分配單位，每個分配單位對應於點陣圖中的一位，0表示空閒1，表示佔用（或者反過來）。圖4-5表示出了一個記憶體片段和對應的點陣圖。

圖4-5 (a)一段有五個進程和三個空洞的記憶體，刻度表示記憶體分配的單位，陰影表示空閒區域（在點陣圖中用0表示）。(b)對應的點陣圖。(c)用清單表示的同樣的資訊。

分配單位的大小是一個重要的設計因素。分配單位越小，點陣圖越大，但是即使分配單位只有4個位元組大小，32位元的記憶體也只需要點陣圖中的1位，32n位元的記憶體只需要n位元的點陣圖，因此點陣圖只佔用了1/33的記憶體。如果分配單位選的比較大，需要的點陣圖就比較小，但是如果進程的大小不是分配單位的整數倍，那麼最後一個分配單位中相當數量的記憶體就可能被浪費掉。

因為點陣圖的大小僅僅取決於記憶體和分配單位的大小，它提供了一個簡單的使用固定大小記憶體就能對記憶體使用情況進行跟蹤的方法。它的主要問題是當它決定把一個占k個分配單位的進程調入記憶體時，記憶體管理器必須搜索點陣圖以找出一串k個連續的0。在點陣圖中查找指定長度的連續0串是一個緩慢的操作（因為串可能跨越字邊界）。這是反對使用點陣圖的一個理由。

4.2.2使用鏈表的記憶體管理

跟蹤記憶體使用的另一個方法是維持一個已分配和空閒的記憶體段的鏈表，這裡，一個段或者是一個進程，或者是兩個進程間的一個空洞。圖4-5(a)的記憶體可以用圖4-5(c)所示的段鏈表來表示。表中的每一個表項都包含下列內容：指明是空洞(H)還是進程(P)的標誌、開始位址、長度、和指向下一個表項的指標。

在這個例子中，段鏈表是按照位址排序的，這樣作的好處是在進程結束或被換出時更新鏈表十分直觀。一個要結束的進程一般有兩個鄰居（除非它是在記憶體的最低端或最高端），他們可能是進程也可能是空洞，這導致了圖4-6所示的四種組合。在圖4-6(a)中更新鏈表需要把P替換為H；在圖4-6(b)和4-6(c)中兩個表項被合併成為一個，鏈表變短了一個表項；在圖4-6(d)中三個表項被合併為一個，兩個表項被從表中刪除。因為結束進程的進程表表項中通常含有指向對應于它的段鏈表表項的指標，因此這個鏈表使用雙鏈表可能要比圖4-5(c)所示的單鏈表更方便，這樣更易於找到上一個表項以檢查是否可以合併。

圖4-6 進程X終止時四種與鄰居合併的方式。

當進程和空洞按照位址順序存放在鏈表中時，好幾種演算法都可以用來為新創建和換進的進程分配空間，這裡我們假設存儲管理器知道要分配的記憶體的大小。最簡單的演算法是首次適配演算法，存儲管理器沿著記憶體段鏈表搜索直到找到一個足夠大的空洞，除非空洞大小和要分配的空間大小剛好一樣，否則的話這個空洞將被分為兩部分，一部分供進程使用，另一部分是未用的記憶體。首次適配演算法是一種快速的演算法，因為它盡可能地少搜索。

首次適配的一個較小變形是下次適配，它的工作方式和首次適配相同，區別是每次找到合適的空洞時都記住當時的位置，在下次尋找空洞時從上次結束的地方開始搜索，而不是每次都從頭開始。Bays(1977)的模擬指出下次適配的性能略低於首次適配。

另一個大家熟知的演算法是最佳適配演算法，它搜索整個鏈表以找出夠用的最小的空洞。最佳適配演算法試圖找出最接近實際需要的大小的空洞，而不是把一個以後可能會用到的大空洞先使用。

作為首次適配和最佳適配演算法的例子，讓我們再觀察圖4-5，假如需要一個大小為2的塊，首次適配將分配在位置5的空洞，而最佳適配將分配在位置18的空洞。

由於最佳適配演算法每次被調用時都要搜索整個鏈表，它要比首次適配演算法慢，有點出乎意料的是它還會導致比首次適配更多的記憶體浪費，因為它傾向于生成大量沒用的很小的空洞，而總的來說首次適配演算法生成的空洞更大。

為了避免最接近適配的空洞會分裂出極小空洞的問題，大家可能會想到最差適配，即總是分配最大的空洞，以使分裂出來的空洞比較大從而可以繼續使用，但模擬說明最差適配也同樣不是一個好主意。

如果把進程和空洞放在不同鏈表中，那麼這四個演算法的速度都能得到提高，這樣他們就能只檢查空洞而不是進程。但這種分配速度的提高的一個不可避免的代價就是複雜度提高和記憶體釋放速度變慢，因為一個釋放的記憶體段必須從進程鏈表中刪除並插入空洞鏈表。

如果進程和空洞使用不同的鏈表，空洞鏈表可以按照大小排序以提高最佳適配的速度。在最佳適配演算法搜索由小到大排列的空洞鏈表時，當它找到一個合適的空洞時它就知道這個空洞是能容納這個作業的最小的空洞，因此是最佳的，不需要象在單個鏈表的情況那樣繼續進行搜索。當空洞鏈表按大小排序時，首次適配與最佳適配一樣快，而下次適配則毫無意義。

在空洞被保存在不同於進程的鏈表中時我們可以作一個小小的優化：不用單獨的資料結構存放空洞鏈表，取而代之用空洞自己。每個空洞的第一個字可以是空洞大小，第二個字指向下一個空洞，於是圖4-5(c)中三個字加一位(P/H)的那些鏈表結點就不再需要了。

還有一種分配演算法叫做快速適配，它為一些經常被用到長度的空洞設立單獨的鏈表。例如，它可能有一個n個項的表，這個表的第一個項是指向長度為4K的空洞的鏈表的表頭的指標，第二個項是指向長度為8K的空洞的鏈表的指標，第三個項指向長度12K的空洞鏈表，等等。象21K這樣的空洞既可以放在20K的鏈表中也可以放在一個專門的存放大小比較特別的空洞的鏈表中。快速適配演算法尋找一個指定大小的空洞是十分迅速的，但它有一個所有將空洞按大小排序的方案所共有的一個缺點，即在一個進程結束或被換出時尋找它的鄰接塊以查看是否可以合併是非常費時間的。如果不作合併，記憶體會很快分裂成大量的進程無法使用的小空洞。

4.3 虛擬記憶體

許多年以前人們第一次遇到了太大以至於記憶體容納不下的程式，通常採取的解決方法是把程式分割成多個叫做覆蓋塊的片段，覆蓋塊0首先運行，在它結束時它將調用另一個覆蓋塊。有一些覆蓋系統非常複雜，允許多個覆蓋塊同時在記憶體中。覆蓋塊存放在磁片上，在需要時由作業系統動態地換入換出。

雖然覆蓋塊換入換出的實際操作都是由系統完成的，但是必須由程式師把程式分割成片段。把一個大程式分割成小的、模組化的片段是非常費時和枯燥的。不久就有人找到了一個把全部工作都交給電腦的辦法。

這個方法(Fotheringham, 1961)被稱作虛擬記憶體（virtual memory）。虛擬記憶體的基本思想是程式、資料、堆疊的總的大小可以超過可用物理記憶體的大小，作業系統把程式當前使用的那些部分保留在記憶體中，而把其他部分保存在磁片上。例如一個16M的程式，通過仔細地選擇在各個時刻將哪4M內容保留在記憶體中，並在需要時在記憶體和磁片間交換程式的片段，那麼就可以在一個4M的機器上運行。

虛擬記憶體也可以工作在許多程式的片段同時存放在記憶體的多道程序系統中。當一個程式等待它的一部分被調入時，它是在等待I/O操作而不能運行，因此CPU可以象在任何其他多道程序系統中一樣，交給另一個進程使用。

4.3.1分頁

大部分虛擬記憶體系統都使用了一種稱為分頁（paging）的技術，我們這裡將討論它。在任何一台電腦上，都存在一個程式能夠產生的記憶體位址的集合。當程式執行這樣一條指令時：

MOVE REG, 1000

它把位址為1000的記憶體單元的內容複製到REG中（或者反過來，這取決於電腦的型號）。位址可以通過索引、基址寄存器、段寄存器和其他方式產生。

這些由程式產生的位址被稱為虛位址（virtual addresses），他們構成了一個虛位址空間（virtual address space）。在沒有虛擬記憶體的電腦上，虛位址被直接送到記憶體匯流排上，使具有同樣位址的物理記憶體字被讀寫；而在使用虛擬記憶體的情況下，虛位址不是被直接送到記憶體匯流排上，而是送到記憶體管理單元(MMU)，它由一個或一組晶片組成，其功能是把虛位址映射為物理位址，如圖4-7所示。

圖4-7 MMU的位置和功能。

圖4-8是一個非常簡單的的例子，它演示這個映射如何工作。在這個例子中，我們有一台可以生成16位元位址的電腦，位址變化範圍從0到64K，這些位址是虛位址。然而這台電腦只有32K的物理記憶體，因此雖然可以編寫64K的程式，他們卻不能被完全調入記憶體運行。在磁片上必須有一個可以大到64K的程式的完整內核映象，以保證程式片段在需要時能被調入。

圖4-8 虛位址與實體記憶體位址之間的關係由頁表給出。

虛位址空間被劃分成稱為頁（pages）的單位，在物理記憶體中對應的單位稱為頁框（page frames），頁和頁框總是同樣大小的，在這個例子中是4K，但在現有的系統中常用的頁的大小是從512位元組到64K。對應於64K的虛位址空間和32K的物理記憶體，他們分別包含有16個虛頁和8個頁框。記憶體和磁片之間的傳輸總是以頁為單位的。

當程式試圖訪問位址0時，比如使用這條指令：

MOVE REG, 0

虛地址0將被送往MMU，MMU看到虛地址落在頁0範圍內（0到4095），根據它的映射這個頁對應的是頁框2（8192到12287），因此MMU把位址變換為8192並把位址8192送到匯流排上。記憶板對MMU一無所知，它只看到一個對位址8192讀或寫的請求並且執行它。MMU從而有效地把所有從0到4095的虛位址映射到了8192到12287的物理位址。

同樣地，指令：

MOVE REG, 8192

被有效地轉換為：

MOVE REG, 24576

因為虛位址8192在虛頁2中並且這個虛頁被映射到了物理頁框6（物理位址從24576到28671）。第三個例子，虛地址20500在虛頁5（虛位址20480到24575）距開頭20位元組處，被映射到物理位址12288 + 20 = 12308。

通過適當地設置MMU，它可以把16個虛頁映射到8個頁框中的任何一個，但是這種能力本身並沒有解決虛位址空間比物理記憶體大的問題。在圖4-8中我們只有8個物理頁框，於是只有8個虛頁被映射到了物理記憶體中，其他在圖中用叉號表示的頁沒有被映射。在實際的硬體中，每個虛頁都有一個Present/absent位指出該頁是否被映射了。

在程式試圖訪問未映射的頁時，例如通過下列指令，會發生什麼 ？

MOVE REG, 32780

虛頁8（從32768開始）的第12個位元組所對應的物理位址是什麼呢？MMU注意到這個頁沒有映射（在圖中用叉號表示），於是使CPU陷入作業系統，這個陷入稱為缺頁故障。作業系統找到一個很少使用的頁框並把它的內容寫入磁片，隨後把需引用的頁取到剛才釋放的頁框中，修改映射，然後重新開機引起陷入的指令。

例如，假設作業系統決定放棄頁框1，那麼它將把虛頁8裝入物理位址4K，並對MMU作兩處修改：首先，它要標記虛頁1為未映射，以使以後任何對虛位址4K到8K的訪問都引起陷入；隨後把虛頁8對應表項的叉號改為1，因此在引起陷入的指令重新開機時，它將把虛位址32780映射為物理位址4108。

現在讓我們看看MMU的內部結構以瞭解它是怎麼工作的、以及為什麼我們選用的頁面大小是2的次冪。在圖4-9中我們可以看到一個虛位址的例子，虛位址8196（二進位是0010000000000100）用圖4-8所示的MMU的映射進行映射，輸入的16位虛位址被分為4位的頁號和12位的偏移量，4位的頁號可以表示16個頁面，12位元的偏移可以為一頁內的全部4096個位元組編址。

圖4-9 在16個4K頁的情況下MMU的內部操作。

頁號用作頁表的索引，以得出對應於這個虛頁的頁框號。如果Present/absent位元是0，則將引發一個作業系統的陷入；如果這個位是1，在頁表中查到的頁框號將被複製到輸出寄存器的高三位中，加上輸入虛位址中的12位偏移，就構成了15位元的物理位址。輸出寄存器的內容隨即被作為物理位址送到記憶體匯流排。

4.3.2頁表

從理論上來講，虛位址到物理位址的映射就象我們剛才描述的那樣。虛位址被分成虛頁號（高位）和偏移（低位元）兩部分，虛頁號被用做頁表的索引以找到該虛頁對應的分頁表項目，從分頁表項目中可以找到頁框號（如果有的話）。隨後頁框號被拼接到偏移的高位端，形成送往記憶體的物理位址。

頁表的目的是把虛頁映射為頁框。從數學角度而言，頁表是一個函數，它的參數是虛頁號，結果是物理頁框號。通過這個函數可以把虛位址中的虛頁域替換成頁框域，從而形成物理位址。

在這個簡單的描述之外，我們還必須面對兩個主要問題：

1. 頁表可能會非常大。

2. 位址映射必須十分迅速

第一點是因為現代電腦使用的虛位址最少也有32位，在頁長為4K時32位的位址空間有1百萬個頁，64位的位址空間則更要多得多。虛位址空間中的1百萬個頁需要有1百萬個表項的頁表，並且每個進程都需要有自己的頁表。

第二點是因為虛位址到物理位址的映射在每次記憶體訪問時都要進行。一條典型的指令有一個操作字，通常還有一個記憶體運算元，因此每條指令都要進行一次、兩次或多次的頁表引用。假如執行一條指令需要10納秒，則頁表的查找必須在幾個納秒內完成以避免成為一個主要的瓶頸。

對大而快速的頁映射的需要是電腦構造方式的一個重要限制，這個問題不僅在最高檔的電腦中是最重要的，在強調性能/價格比的低檔機中也是一個重要的問題。在本節和下幾節中我們將詳細研究頁表的設計和已經實際使用的一些解決方法。

最簡單的設計（至少在理論上）是使用一個由一組快速的硬體寄存器組成的頁表，虛頁號作為索引，每個虛頁一個表項。在啟動一個進程時，作業系統把位於主存中的進程頁表裝入寄存器，在進程運行期間不用因為頁表而訪問記憶體。這一方法的優點是直觀並且在映射時不需要訪問記憶體，它的一個缺點是非常昂貴（如果頁表很大的話），而且在每次進程切換時都要裝入位於主存中的頁表，這也會降低性能。

另外一個極端是把頁表全部放在主存中，這時，需要的全部硬體僅僅是一個指向頁表起始位址的寄存器，在上下文切換時只要重新裝入這個寄存器就可以改變記憶體映射。當然，它的一個嚴重缺點是在執行每條指令時都需要一次或多次訪問記憶體讀取分頁表項目，因此這個這種方法很少以它最單純的方式使用，下面我們將研究它一些性能要好得多的變種。

多級頁表

為了避免把龐大的頁表始終保存在記憶體中，許多電腦使用了多級頁表，圖4-10是一個簡單的例子。在圖4-10(a)中，32位的虛地址劃分為10位的PT1域、10位的PT2域和12位的偏移。因為偏移是12位，所以頁長是4K，頁面共有220個。

圖4-10 (a)一個有兩個頁表域的32位位址。(b)二級頁表。

多級頁表的秘密是避免把所有的頁表一直保存在記憶體中，特別是那些不需要的就不應該保留。比如一個需要12百萬位元組的進程，最低端是4兆程式正文，後面是4兆資料，頂端是4兆堆疊，在資料頂端上方和堆疊底端之間的上吉（千兆）位元組的空洞根本沒有使用。

讓我們看看圖4-10(b)的例子中二級頁表是怎樣工作的。在左邊是頂級頁表，它具有1024個表項，對應於10位的PT1域。當一個虛位址被送到MMU時，MMU首先抽出虛位址的PT1域並把它作為訪問頂級頁表的索引。整個4G虛位址空間（32位）已經被切成1024塊，每塊4百萬位元組，這1024個表項中每個表項都表示其中的一塊。

通過索引得到的頂級分頁表項目包含有二級頁表的位址或頁框號，頂級頁表的表項0指向程式正文的頁表、表項1指向資料的頁表、表項1023指向堆疊的頁表。其他的表項（用陰影表示的）未用。PT2域被用作訪問選定的二級頁表的索引以找到該虛頁的頁框號。

舉一個例子，虛位址0x00403004（十進位4,206,596）位於資料部分12,292位元組處。它對應的PT1=1、PT2=3、偏移=4。MMU首先用PT1作為索引訪問頂級頁表得到表項1，它對應的地址範圍是4M到8M；隨後它用PT2作為索引訪問剛剛找到的二級頁表並得到表項3，它對應的位址範圍是在它的4M塊內的12288到16383（即絕對位址4,206,592到4,210,687）。這個表項含有虛地址0x00403004所在頁的頁框號。如果這個頁不在記憶體中，Present/absent位元將是0，一個頁面故障將被引發；如果這個頁在記憶體中，從二級頁表中得到的頁框號將和偏移(4)結合構成物理位址並通過匯流排送到記憶體。

在圖4-10中值得注意的是雖然位址空間超過一百萬個頁，實際上只需要四個頁表：頂級頁表，0到4M、4M到8M和頂端4M的二級頁表。頂級頁表中有1021個表項的Present/absent位都被設為0，使得訪問他們時強制產生一個頁面故障。如果這種情況發生了，作業系統將注意到進程在試圖訪問一個不期望被訪問的位址並採取適當的行動，比如向進程發出一個信號或殺死進程。在這個例子中各種長度我們選擇的都是整的數（正文、資料、堆疊都是4M長），並且選擇PT1與PT2等長，但在實際中其他的值當然也是可能的。

圖4-10的二級頁表可以擴充為三級、四級或更多級。更多的級別帶來了更多的靈活性，但頁表超過三級所帶來的複雜性是否值得令人懷疑。

現在讓我們把注意力從頁表的總體結構轉移到單個分頁表項目的細節上來，表項的結構是與機器密切相關的，但不同機器的表項存儲的資訊大致相似。在圖4-11中我們給出了分頁表項目的樣本，不同電腦分頁表項目的大小不一樣，但32位是一個一般的尺寸。最重要的域是頁框號，畢竟頁映射的目的是找到這個值；其次是存在/不在(Present/absent)位，這一位是1時這個表項是有效的可以被使用，如果是0，表示這個表項對應的虛頁現在不在記憶體中，訪問這一位為0的頁會引起一個頁面故障。

圖4-11 一個典型的分頁表項目。

保護位指出這個頁允許什麼樣的訪問。在最簡單的形式下這個域只有一位，0表示讀寫，1表示唯讀。一個更先進的安排是使用三位，各位分別指出是否允許讀、寫、執行這個頁。

修改(Modified)和引用(Referenced)位跟蹤頁的使用。在一個頁被寫入時硬體自動設置修改位元，此位元在作業系統重新分配頁框時是非常有用的，如果一個頁已經被修改過（即它是“髒”的），則必須把它寫回磁片，否則只用簡單地把它丟棄就可以了，因為它在磁片上的拷貝仍然是有效的。此位有時也被稱為髒(dirty)位，因為這反映了該頁的狀態。

引用位在該頁被引用時設置，不管是讀還是寫。它的值被用來説明作業系統在發生頁面故障時選擇淘汰的頁，不在使用的頁要比在使用的頁更適合於被淘汰。這個位在我們即將討論的好幾個頁面替換演算法中都將扮演重要的角色。

最後一位可以禁止該頁被緩存，這個特性對那些映射到設備寄存器而不是常設記憶體的頁面是非常重要的。假如作業系統正在緊張地迴圈等待某個I/O設備對它剛發出的命令作出回應，保證硬體是不斷地從設備中讀取而不是讀取一個舊的被緩存的拷貝是非常重要的，通過這個位可以禁止緩存。具有獨立的I/O空間而不使用記憶體映射I/O的機器不需要這個位。

值得注意的是當一個頁面不在記憶體時保存該頁的磁片位址不是頁表的一部分，原因很簡單，頁表只保存硬體把虛位址轉換為物理位址時所需要的資訊，作業系統在處理頁面故障時需要的資訊保存在作業系統內部的軟體表格中。

4.3.3 TLBs—翻譯後援記憶體(Translation Lookaside Buffers)

由於頁表尺寸較大，因此許多分頁方案都只能把它保存在記憶體中，但這種設計對性能有很大的影響。例如一條把一個寄存器的內容複製到另一個寄存器中指令，在不使用分頁時只需要訪問記憶體一次取指令，而在使用分頁時需要額外的記憶體訪問去讀取頁表。因為系統的運行速度一般都是被CPU從記憶體中取得指令和資料的速率限制的，在每次訪問記憶體時都要去訪問兩次頁表會使機器的性能降低2/3，這樣的系統是沒有人會願意用的。

電腦設計者們很早就意識到了這個問題並提出了一個解決方案，這個方案基於這樣的觀察：大部分程式傾向於對較少的頁面進行大量的訪問。因此只有一小部分分頁表項目經常被用到，其他的很少被使用。

採取的解決方法是為電腦裝備一個不需要經過頁表就能把虛位址映射成物理位址的小的硬體設備，這個設備叫做TLB (翻譯後援記憶體，Translation Lookaside Buffer)，有時也叫做相聯記憶體（associative memory），如圖4-12所示。它通常在MMU內部，條目的數量較少，在這個例子中是8個，一般很少超過64個。每個條目包含一個頁的資訊，主要有虛頁號、在頁被修改時將被設置的一個位、保護碼（讀/寫/執行允許）、和頁所在的物理頁框號，這些域和頁表中的域是一一對應的，還有另外一位指出這個條目是否有效。

圖4-12 一個用於加速分頁操作的TLB。

生成圖4-12所示的TLB的一個可能的例子是一個正在執行某個迴圈的進程，該迴圈跨越虛頁19、20、21，因此這些頁面的保護碼是讀和執行。正在使用的主要資料（比如一個正在被處理的陣列）在頁面129和130，頁140包含了陣列演算中用到的索引。最後，堆疊在頁面860和861。

現在讓我們看一看TLB是怎樣工作的，當一個虛位址被送到MMU翻譯時，硬體首先把它和TLB中的所有條目同時（並行地）進行比較，看它的虛頁號是否在TLB中，如果找到了並且這個訪問沒有違反保護位，它的頁框號將直接從TLB中取出而不用去查頁表。如果虛頁號在TLB中但當前指令試圖寫一個唯讀的頁面，這時將發生一個保護故障，與直接訪問頁表時相同。

有趣的是當虛頁號不在TLB中時會發生什麼？如果MMU發現在TLB中沒有命中，它將隨即進行一次常規的頁表查找，然後從TLB中淘汰一個條目並把它替換為剛剛找到的分頁表項目。因此如果這個頁很快會再被用到的話，第二次訪問時它就能在TLB中直接找到。在一個TLB條目被淘汰時，被修改的位元被複製回在記憶體中的分頁表項目，其他的值則已經在那裡了。當TLB從頁表中裝入時，所有的域都從記憶體中取得。

軟體TLB管理

直到現在我們一直假設，每個帶有頁式虛擬記憶體的機器都有能被硬體識別的頁表外加一個TLB，在這個設計中TLB的管理和TLB故障的處理都完全由MMU硬體完成，只有一個頁不在記憶體時才會陷入作業系統。

在過去，這個假設是對的，但是在一些現代的RISC機中，包括MIPS、Alpha、HP PA，幾乎全部的這種頁管理工作都是由軟體完成的。在這些機器中，TLB條目是由作業系統顯式地裝入的，在TLB沒有命中時，MMU不是到頁表中找到並裝入需要的頁面資訊，而是產生一個TLB故障把問題交給作業系統，作業系統必須找到頁面，從TLB中淘汰一個條目，裝入一個新的，然後重新開機產生故障的指令。當然，所有這些都必須用很少幾條指令完成，因為TLB不命中發生的頻率遠比頁面故障大得多。

令人驚奇的是，如果TLB的尺寸取一個合理的較大值（比如64個條目）以減少不命中的頻率，那麼軟體管理的TLB效率可能相當高。這裡主要的收益是一個簡單得多的MMU，它在CPU晶片上為快取記憶體和其他能提高性能的部件讓出了相當大的面積。Uhlig等詳細地討論了軟體TLB管理(1994)。

人們已經使用了很多方法來提高使用軟體管理TLB機器的性能，有一個方法既能減少TLB的不命中率又能減少在TLB不命中確實發生時的開銷（Bala等，1994）。為了減少TLB的不命中，作業系統有時可以用它的直覺來指出那些頁可能將被使用並把他們預裝入TLB中。例如當一個客戶進程向位於同一台機器上的伺服器進程發出一個RPC請求時，伺服器很可能即將運行。知道了這一點，在客戶進程因執行RPC陷入時，系統就可以找到伺服器的代碼、資料、堆疊的頁面，並在TLB中提前為他們建立映射，以避免TLB故障的發生。

無論是硬體還是軟體，處理TLB不命中的一般方法是對頁表執行索引操作找出所引用的頁。用軟體執行這個搜索的一個問題是保存頁表的頁面本身可能就不在TLB中，這將在處理過程中再一次引發一個TLB故障。這種故障可以通過保持一個大的（比如4K）TLB條目的軟體快取記憶體而得到減少，這個快取記憶體保存在固定位置，它的頁面總是保持在TLB中，作業系統通過首先檢查軟體快取記憶體可以大大減少TLB不命中的次數。

4.3.4逆向頁表

到目前為止，在我們描述的各種類型的傳統頁表中每個虛頁都需要一個分頁表項目。如果位址空間是232位元組，每頁4096位元組，則需要大於一百萬的分頁表項目，頁表最低限度也有4百萬位元組長，在較大的系統上這也許是可行的。

然而，隨著64位元電腦越來越普遍，情況徹底地改變了。如果位址空間是264位元組，每頁4K，我們需要超過1015位元組的空間存放頁表，用一百萬吉伽位元組的記憶體僅僅來存放頁表是絕對不可行的，無論是現在還是未來的幾十年內。因此，64位元的位址空間需要一種不同的解決方法。

一種解決方法就是逆向頁表，在這個設計中物理記憶體的每個頁框對應一個分頁表項目，而不是虛位址空間中的每個虛頁對應一個分頁表項目。在具有64位元虛位址、4K大小頁面、32M RAM的系統上，一個逆向頁表只需要8192個表項，每個表項跟蹤誰（進程，虛頁）現在在該表項對應的物理頁框中。

儘管逆向頁表至少在虛位址空間遠大于物理記憶體時能節省大量的空間，但是它有一個嚴重的弱點：虛位址到物理位址的變換變得十分困難。當進程n引用虛頁p時，硬體不能再用p作為索引查頁表得到物理位址，取而代之的是在整個逆向頁表中查閱資料表項(n,p)。更為嚴重的是這種搜索必須在每次記憶體訪問時進行，而不只是發生頁面故障時，在每次記憶體訪問時搜索一個8K的表不可能讓你的機器能和原來一樣快。

擺脫這個困境的方法是使用TLB。如果TLB能包括所有常用的頁面，位址轉換就能象傳統頁表一樣快。然而，在TLB不命中時必須對逆向頁表進行搜索，通過用雜湊表作為逆向頁表的索引可以使這個搜索是速度相當快。逆向頁表現在被用在一些IBM和HP工作站上，隨著64位機的普及它將變得更普遍。

處理很大虛存的其他方法可以參閱下列文獻：Huck和Hays, 1993; Talluri和Hill, 1994; Talluri等, 1995。

4.4頁面替換演算法

在發生頁面故障時，作業系統必須從記憶體中選擇一個頁刪除掉以便為即將被調入的頁讓出空間。如果要被刪除的頁在記憶體期間已經被修改過，就必須把它寫回磁片以更新該頁在磁片上的拷貝；如果這個頁沒有被修改過（例如一個包含程式正文的頁），那麼它在磁片上的拷貝已經是最新的了，因此不需要寫回，要讀入的頁直接覆蓋被淘汰的頁就可以了。

在每次發生頁面故障時儘管我們可以隨機選擇一個頁替換，但是選擇不常使用的頁會使系統性能好得多。如果一個經常使用的頁被淘汰掉了，極有可能它很快又要被調回來，引起不必要的額外開銷。人們已經在理論上和實踐上對頁面替換演算法進行了深入的研究，下面我們將描述幾個最重要的演算法。

4.4.1最優頁面替換演算法

最好的頁面替換演算法描述非常容易但卻不可能實現，它是這樣工作的：在頁面故障發生的瞬間，有些頁在記憶體中，其中的一個頁將被緊接著的下一條指令訪問（包含那條指令的哪個頁），其他的頁可能要到10、100、或1000條指令後才會被訪問，每個頁都可以用在該頁面首次被引用前所要執行的指令數進行標記。

最優頁面替換演算法只是簡單地規定，標記最大的頁應該被淘汰掉。如果一個頁在八百萬條指令內不會被使用，另外一個頁在六百萬條指令內不會被使用，則淘汰掉前一個，從而把因需要取回這個頁發生的頁面故障推到將來，越遠越好。電腦也象人一樣，希望把不愉快的事情盡可能往後拖。

這個演算法唯一的一個問題是它是無法實現的，在頁面故障發生時，作業系統根本沒有辦法知道各個頁面下一次被訪問是什麼時候（在短作業優先調度演算法中我們曾遇到同樣的情況—系統怎麼能知道哪個作業是最短的呢？）。當然，通過首先在模擬器上運行程式，跟蹤所有頁面的引用情況，在第二次運行時利用第一次運行時收集的資訊是能夠實現最優頁面替換演算法的。

我們可以用這個辦法對可實現演算法和最好演算法的性能進行對比。如果一個作業系統達到了只比最優演算法差百分之一的性能，那麼尋找更好的演算法的努力最多只能換來百分之一的提高。

為了避免可能的混淆，讀者必須清楚這個頁面訪問情況的記錄只涉及到剛剛被測試的一個程式，因此從中匯出的頁面替換演算法也只是針對該程式的。雖然這個方法對評價頁面替換演算法很有用，但它在實際系統中沒有用處。下面我們將研究可以在實際系統中使用的演算法。

4.4.2 最近未使用頁面替換演算法

為了使作業系統能夠收集關於哪些頁正在被使用和哪些頁沒有被使用的統計資訊，在大部分具有虛擬記憶體的電腦中每個頁都有兩個與其相關的狀態位元。R在頁面被引用（讀或寫）時設置；M在頁面被寫入（即被修改）時設置。這些位包含在分頁表項目中，如圖4-11所示。這些位元一定要在每次記憶體訪問時都進行更新，因此由硬體來設置它們是必不可少的。一旦一個位被設置成1，它就一直保持1直到作業系統用軟體把它重定為0。

如果硬體沒有這些位元，我們可以這樣進行模擬：在一個進程啟動時，把它所有的頁面都標記成不在記憶體，一旦一個頁被訪問到就會引發一個頁面故障，這時作業系統就可以設置R位元（在它的內部表格中）、修改分頁表項目使它指向正確的頁、屬性為唯讀、並重新開機指令。如果這個頁隨後被寫入就又會引發一個頁面故障，允許作業系統設置這個頁的M位元並把它的屬性改為讀寫。

R位和M位可以用來構造這樣一個簡單的換頁演算法：在一個進程啟動時，它的所有頁的兩個位都由作業系統設置成0，R位被定期地（比如在每次時鐘中斷時）清零，以把最近沒有被訪問的頁和被訪問了的頁區別開來。

當發生頁面故障時，作業系統檢查所有的頁面並根據它們當前的R位元和M位的值把分為四類：

第0類：沒有被訪問，沒有被修改。

第1類：沒有被訪問，已被修改。

第2類：已被訪問，沒有被修改。

第3類：被訪問，被修改。

儘管第1類乍看起來似乎是不可能的，但是一個第3類的頁在它的R位元被時鐘中斷清除後就成了第1類。時鐘中斷不清除M位元是因為這個資訊在決定一個頁是否需要寫回磁片時將要用到。

NRU（最近未使用）演算法隨機地從編號最小的非空類中挑選出一個頁淘汰。這個演算法隱含著的意思是淘汰一個在最近一時鐘週期中（典型時間是20毫秒）沒有被訪問的已修改頁要比淘汰一個被頻繁訪問的乾淨頁好。NRU吸引人的地方是易於理解和有效地實現，並切它的性能儘管不是最好的，但常常是夠用的。

4.4.3先進先出頁面替換演算法

另一種低開銷的頁面替換演算法是FIFO（先進先出）演算法，為了解釋它是怎樣工作的，讓我們設想一個超市，它的貨架恰好能展示k種不同的商品。有一天，某家公司介紹了一種新的方便食品—快速、冷凍乾燥的可以用微波爐加工的乳酸酪，這個產品非常成功，所以我們有限的超市必須撤掉一種舊商品以便能夠存放它。

一種可能的解決方法就是找到超市中銷售時間最長的商品撤掉（比如某個120年以前就已經開始賣的商品），理由是已經沒有人喜歡它了。這實際上相當於超市有一個按照引入時間排列的所有商品的鏈表，新的商品加到鏈表的尾部，鏈表頭上的商品被撤掉。

同樣的思想也可以用在頁面替換演算法中。由作業系統維持一個所有當前在記憶體中的頁的鏈表，最老的頁在頭上，最新來的頁在表尾。當發生頁面故障時淘汰表頭的頁並把新調入的頁加到表尾。當FIFO用在超市時，它可能會淘汰須蠟，但它也可能淘汰掉麵粉、鹽、或黃油。當它用在電腦上時也會引起同樣的問題，由於這個原因，單純形式的FIFO很少使用。

4.4.4 第二次機會頁面替換演算法

為了避免FIFO可能會把經常使用的頁替換出去的問題，我們可以對它做一個簡單的修改，對最老頁面的R位元進行檢查。如果R位是0，那麼這個頁既老又沒用，應該被立刻替換掉；如果是1，就清除這個位，把這個頁放到頁鏈表的尾端，修改它的裝入時間讓它就象剛裝入的一樣，然後繼續搜索。

這個演算法的操作叫做第二次機會，如圖4-13所示。在圖4-13(a)中我們看到頁A到H按照抵達記憶體的時間的順序保存在鏈表中。

圖4-13 第二次機會演算法的操作。(a)頁面按先進先出的順序排列。(b)在時間20發生頁面故障並且A的R位元已經設置時的頁面鏈。

假設在時間20發生了一個頁面故障，這時最老的頁是在進程啟動的時間—時間0抵達的A。如果A的R位是0，它將被淘汰出記憶體，或者把它寫回磁片（如果它已經被修改），或者只是簡單地放棄（如果它是乾淨的）；在另一方面，如果R位已經被設置，則A將被放到鏈表的尾部並且“裝入時間”被重新設置為當前時間（20），R位將被清除，對合適頁面的搜索將從B開始繼續進行。

第二次機會演算法所做的是尋找一個從上一次對它檢查以來沒有引用過的頁面，如果 所有的頁都被引用了，它就降級為純粹的FIFO演算法。特別地，讓我們設想假如圖4-13(a)中所有的頁的R位都被設置了，作業系統將一個接一個地把各個頁移到鏈表的尾部並清除被移動的頁的R位。最後演算法又將回到頁A，這時它的R位已經被清除了，因此A將被淘汰，所以這個演算法總是可以結束的。

4.4.5時鐘頁面替換演算法

儘管第二次機會演算法是一個較合理的演算法，但它經常要在鏈表中移動頁面，既降低了效率，又是不必要的。一個更好的辦法是把所有的頁面保存在一個類似鐘錶面的環形鏈表中，如圖4-14所示，有一個錶針指向最老的頁面。

在發生頁面故障時，演算法首先檢查錶針指向的頁面，如果它的R位元是0就淘汰掉這個頁，並把新頁插入這個位置，然後把錶針前移一個位置；如果R位是1就清除R位並把錶針前移一個位置，重複這個過程直到找到一個R位為0的頁為止。瞭解了這個演算法的工作方式我們就不奇怪為什麼它被叫做時鐘了，它和第二次機會演算法的區別僅僅是實現不同。

圖4-14 時鐘頁面替換演算法。

4.4.6最久未使用頁面替換演算法

對最優演算法的一個很好的近似是基於這樣的觀察的：在上面幾條指令中被頻繁用到的頁面極有可能在下面幾條指令中也將被頻繁用到，反過來說，已經很久沒有使用的頁面很有可能在未來較長的一段時間內都不會被用到。這個思想向我們提示了一個可以實現的演算法：在頁面故障發生時，淘汰掉沒有使用的時間為最長的頁。這個策略叫做LRU（最久未使用）換頁。

雖然LRU在理論上是可以實現的，但代價很高。為了完全實現LRU，需要維持一個在記憶體中的所有頁的鏈表，最近使用的頁在表頭，最久未使用的頁在表尾。困難的是這個鏈表必須在每次記憶體訪問時都進行更新，在鏈表中找到被訪問的頁，將它移動到表頭是一個非常費時的操作，即便是用硬體實現也是如此（假設有這樣的硬體）。

然而，還是有其他一些用特殊硬體實現LRU的方法，我們先考慮一個最簡單的。這個方法要求硬體有一個64位元數目器C，它在每條指令執行完後自動加1，每個分頁表項目必須有一個足夠容納這個計數器值的域。在每次記憶體訪問後，當前的C值被存到被訪問頁的分頁表項目中。當發生頁面故障時，作業系統檢查頁表中所有的計數器的值以找出最小的，這個頁就是最久未使用的頁。

現在讓我們看一看第二個硬體LRU演算法，在一個有n個頁框的機器中，LRU硬體可以維持一個n×n位元的矩陣，開始時所有的位都是0。在頁k被訪問到時，硬體首先把k行的位都設置成1，再把k列的位都設置成0。在任何時刻，二進位值最小的行就是最久未使用的，第二小的行是下一個最久未使用的，依此類推。這個演算法的工作情況我們可以用圖4-15所示的實例說明，該實例中有4個頁框，他們的引用次序為

0 1 2 3 2 1 0 3 2 3

在頁0被訪問過後的情況示於圖4-15(a)，依此類推。

圖4-15 使用矩陣的LRU演算法。

4.4.7用軟體類比LRU

雖然前面兩種LRU演算法在理論上都是可以實現的，但是如果有的話也只有非常少的電腦有這種硬體，因此對為沒有這種硬體的機器開發作業系統的設計者而言，他們是沒有價值的。我們需要的是一個能用軟體實現的解決方案。一種可能的方案就是稱作NFU（不常使用）的演算法 ，它使每個頁都和一個軟體計數器相聯繫，計數器的初值是0。在每次時鐘中斷時由作業系統對在記憶體中的頁進行掃描，把每個頁的R位（它的值是0或1）加到它的計數器上。實際上這個計數器是試圖跟蹤各個頁被訪問的頻繁程度，當發生頁面故障時，計數器值最小的頁被選中替換掉。

NFU的主要問題是它從來不忘記任何事情。比如在一個多次掃描編譯器中，在第一遍掃描中被頻繁用到的頁，在程式進入第二遍掃描時計數器值可能仍然很高。實際上，如果第一次掃描的執行時間恰好是各次掃描中最長的，含有以後各次掃描代碼的頁的計數器可能總是比含有第一次掃描代碼的頁小，結果是作業系統將刪除掉有用的頁而不是不再使用的頁。

幸運的是只用對NFU做一個小小的修改就能讓它相當好地模擬LRU。修改分兩部分：第一是計數器在R位被加進來之前右移一位；第二是R位加到計數器的最左端而不是最右端。

修改以後的演算法稱為老化演算法，圖4-16解釋了它是怎樣工作的。假設在第一個時鐘週期後頁0到頁5的R位值分別是1、0、1、0、1、1，換句話說，在時鐘週期0到在時鐘週期1期間，頁0、2、4、5被訪問到，他們的R位被設置為1，而其他頁的R位仍然是0，對應的六個計數器在經過移位並把R位插入其左端後的值如圖4-16(a)所示。圖中後面的四列是在下四個時鐘週期後的六個計數器的值。

圖4-16 用軟體類比LRU的老化演算法。圖中所示是6個頁面在5個時鐘週期的情況，5個時鐘週期分別由(a)-(e)表示。

在發生頁面故障時，計數器值最小的頁將被淘汰掉。如果一個頁在前面四個週期中都沒有被訪問過，它的計數器最前面應該有四個連續的0，因此它的值肯定要比在前面三個週期中都沒有訪問過的頁的計數器小。

這個演算法在兩個方面和LRU不同。讓我們看圖4-16(e)中的頁3和5，他們都連續兩個週期沒有被訪問了，而在兩個週期之前的週期中他們都被訪問過。根據LRU，如果有一個頁必須被淘汰掉，我們應該在這兩個頁中選一個。然而現在的問題是我們不知道在時鐘週期1到時鐘週期2期間這兩個頁中的哪一個後被訪問到，在每個時鐘週期中只記錄一位元使我們無法區分一個週期內較早和較晚的訪問，我們所能作的只是淘汰掉頁3，因為頁5在再往前的週期中也被訪問過而頁3沒有。

LRU和老化演算法的第二個區別是老化演算法的計數器只有有限位元，在這個例子中是8位。如果兩個頁的計數器都是0，我們只能在兩個頁中隨機選一個。在實際中，有可能一個頁上次被訪問是9個週期以前，另一個頁是1000個週期以前，而我們卻無法看到這些。在實踐中，如果時鐘週期是20毫秒，8位一般是夠用的。假如一個頁已經有160毫秒沒有被訪問，它可能並不是那麼重要。

4.5分頁系統中的設計問題

在前面幾節中我們解釋了分頁系統是怎樣工作的，並給出了幾個基本的頁面替換演算法，但僅僅知道基本的機制是遠遠不夠的。為了設計一個系統，你必須知道更多的東西以使它工作得好。這兩者之間的差別就象知道了怎樣移動象棋的各種棋子和成為一個好棋手之間的差別。在下面的章節中我們將討論為了使分頁系統達到較好的性能，作業系統設計者必須仔細考慮的一些其他問題。

4.5.1 工作集模型

在單純的分頁系統中，當進程啟動時它的頁面沒有一個在記憶體中，在CPU試圖取第一條指令時就會引發一個頁面故障，使作業系統裝入含有第一條指令的頁，其他的由訪問全域資料和堆疊引起的頁面故障通常會緊接著發生。一段時間以後，進程需要的大部分頁已經在記憶體了，進程開始在頁面故障較少的情況小運行。這個策略稱為請調（demand paging），因為頁是在需要時調入，不是預先裝入。

儘管我們可以很容易地編寫一個測試程式，系統地讀一個大的位址空間中的所有頁，產生大量的頁面故障，使得記憶體無法把這些頁面都容納下，但幸運的是大部分進程不是這樣工作的，他們都表現出一種訪問的局部性（locality of reference），意思是在進程運行的任何階段，它都只訪問它的頁面中較小的一部分。例如在一個多次掃描編譯器中，各次掃描時只引用所有頁中的一小部分，並且是不同的部分。

一個進程當前使用的頁的集合叫做它的工作集（working set）（Denning, 1968a; Denning, 1980）。如果整個工作集都在記憶體中，在進入下一個運行階段（比如編譯器的下一次掃描）之前進程的運行不會引起很多頁面故障。如果記憶體太小無法容納整個工作集，進程運行將引起大量的頁面故障並且速度十分緩慢，因為執行一條指令通常只需要幾個納秒，而從磁片上讀入一個頁面一般需要幾十個毫秒。一個20毫秒執行一到兩條指令的程式不知道需要多長時間才能運行完。一個每隔幾條指令就發生一次頁面故障的程式被稱為是在顛簸（Denning, 1968b）。

在分時系統中，進程經常會被轉移到磁片上（即它所有的頁面都被從記憶體中刪除），從而讓其他的進程有機會佔有CPU，問題是當一個進程被再次調回來以後應該怎樣處理？從技術的角度上來看，什麼都不用做，這個進程將一直產生頁面故障直到它的工作集全部被裝入。但是，在每次裝入一個進程時都產生20、50甚至100個頁面故障是非常慢的，並且因為CPU處理一個頁面故障需要幾個毫秒的時間，這也浪費了相當多的CPU時間。

因此，許多分頁系統都試圖跟蹤進程的工作集，並保證在進程運行以前它的工作集就已經在記憶體中了。這樣一種方法稱為工作集模型（Denning, 1970），它旨在大大減少頁面故障。在進程運行之前預先裝入頁面也叫做預調。

為了實現工作集模型，作業系統必須對哪些頁面屬於工作集一直保持跟蹤。監視這個資訊的一個辦法就是使用上面討論過的老化演算法，所有計數器的高n位含有1的頁都被認為是工作集的成員；如果一個頁連續n個時鐘週期沒有被訪問到，它將被從工作集中刪除。對於不同的系統，參數n必須通過試驗確定，但系統的性能通常對精確的數值不是特別地敏感。

有關工作集的資訊可以用來提高時鐘演算法的性能。本來如果錶針指向的頁的R位是0，這個頁就將被淘汰掉，改進之處是檢查這個頁是不是當前進程工作集的一部分，如果是就跳過這個頁。這個演算法叫做wsclock。

4.5.2 局部與全域分配策略

前面的幾節中，我們討論了好幾個在發生頁面故障時選擇被淘汰頁的演算法。與這個選擇相聯繫的一個主要問題（到目前為止我們一直在小心地回避這個問題）是怎樣在相互競爭的可運行進程間分配記憶體。

讓我們看一看圖4-17(a)，在圖中三個進程A、B、C構成了可運行進程的集合。假如A發生了頁面故障，頁面替換演算法在尋找最久未使用的頁時是只考慮分配給A的六個頁呢，還是考慮所有在記憶體中的頁？如果只考慮分配給A的頁，年齡值最小的頁是A5，於是我們將得到圖4-17(b)所示的情況。

圖4-17 局部與全域替換演算法。(a)原先配置。(b)局部頁面替換。(c)全域頁面替換。

在另一方面，如果我們要淘汰年齡值最小的頁而不管它是誰的，那麼頁B3將被選中，我們將得到圖4-17(c)所示的情況。圖4-17(b)的演算法被稱為局部頁面替換演算法，而圖4-17(c) 被稱為全域演算法。局部演算法實際上對應於為每個進程分配固定的記憶體片段；全域演算法在可運行進程之間動態地分配頁框，因此分配給各個進程的頁框數是隨時間變化的。

在通常情況下，尤其是當工作集的大小會在進程的運行期間發生變化時，全域演算法工作得比局部演算法好。如果使用局部演算法，那麼即使還有大量的空閒頁框存在，工作集的增長會導致顛簸；如果工作集收縮了，局部演算法又會浪費記憶體。在使用全域演算法時，系統必須不斷地確定應該給各個進程分配多少頁框。一個辦法是監視由年齡位元指出的工作集的大小，但這個辦法並不足以阻止顛簸。工作集的大小可以在幾微秒內改變，而年齡位只是一個跨越所經過時鐘週期數的粗略的量度。

另一種途徑是使用一個為進程分配頁框的演算法。一種辦法是定期地確定運行進程的數目並為他們分配相等的份額。例如在有475個空閒（即非作業系統用）頁框和10個進程時，每個進程將獲得47個頁框，剩下的5個放入一個公用池中在發生頁面故障時使用。

這個演算法貌似公平，但是給一個10K的進程和一個300K的進程分配同樣大小的記憶體是完全不合理的。取而代之我們可以按照進程大小的比例為他們分配頁面，這樣300K的進程將得到10K進程的30倍的份額。一個可能比較明智的做法是為每個進程都規定最小的頁框數，這樣可以使不管多麼小的進程都可以運行，例如在某些機器上，一條指令的指令自己、源運算元和目的運算元可能都會跨越頁邊界，這樣的一條指令需要多達六個頁面，如果只分配了5個頁面，包含這樣的指令的程式根本就無法運行。

無論是平均分配還是按比例分配都不是直接處理顛簸問題的，一個更直接的控制顛簸的方法是使用頁面故障率(Page Fault Frequency)，或者叫做PFF分配演算法。我們曾經討論過，包括LRU在內的一大類頁面替換演算法的故障率隨著分配的頁框數的增加而減少，如圖4-18所示。

圖4-18 頁面故障率是分配的頁框數的函數。

虛線A對應的是一個過高的故障率，發生故障的進程將被分配更多的頁框以減少故障率；虛線B對應的是一個過低的故障率，我們可以得出結論分配給這個進程的記憶體太多了，可以收回一些頁框。可以看出，PFF試圖把頁面故障率保持在一個可以接受的範圍內。

如果發現記憶體中進程太多以至不可能使所有進程的故障率都低於A，那麼我們就必須從記憶體中移去某些進程，把他們的頁框分給餘下的進程或放進一個空閒頁框池中供後面發生頁面故障時使用。從記憶體中移去一個進程的決定實際上是一種負載控制，它表明即使在分頁系統中交換仍然是需要的，不同的只是現在交換是為了降低潛在的對記憶體的需求，而不是為了立刻使用收回的區塊。

4.5.3頁面大小

頁面大小經常是一個作業系統可以選擇的參數。例如，即使硬體設計只支援512位元組的頁，作業系統可以很容易地通過總是分配兩個連續的頁把0和1、2和3、4和5等當作1K的頁。

選擇最優的頁面大小需要在幾個互相衝突的因素之間進行折衷。首先，隨便找一個正文、資料或堆疊段，它不會恰好裝滿整數個頁，在平均的情況下，最後一頁中有一半是空的。多餘的空間就被浪費掉了，這種浪費叫做內零頭（internal fragmentation）。在記憶體中有n個段、頁長為p位元組時，np/2位元組將被內零頭浪費。這個推理要求小的頁面。

如果考慮一個程式，它分成8個階段循序執行，每階段需要4K記憶體，那麼另一個要求小頁面的原因就很清楚了。如果頁面大小是32K，那就必須始終給該進程分配32K記憶體；如果頁面大小是16K，它就只需要16K的；如果頁面大小是4K或更小，在任何時刻它只需要4K記憶體。總的來說，大的頁面要比小的頁面使更多的沒有用的程式保留在記憶體中。

在另一方面，小的頁面意味著程式需要更多的頁面，這又意味著更大的頁表。一個32K的程式只需要4個8K的頁，卻需要64個512位元組的頁。記憶體與磁片之間的傳輸一般是一次一個頁，因為大部分時間都花在了尋道和旋轉延遲上，傳輸一個小的頁面所用的時間和傳輸一個大的頁面基本上是相同的。裝入64個512位元組的頁可能需要64×15毫秒，而裝入4個8K的頁可能只需要4×25毫秒。

在一些機器上，每次CPU從一個進程切換到另一個進程時都必須把頁表裝入硬體寄存器中。在這些機器上頁面越小意味著裝入頁寄存器的時間越長，而且頁表佔用的空間隨著頁的減小而增大。

這最後一點可以數學地進行分析，假設平均進程大小是s個位元組，頁面大小是p個位元組，每個分頁表項目需要e個位元組，那麼進程需要的頁數大約是s/p，佔用了se/p個位元組的頁表空間，由於內零頭在最後一頁浪費的記憶體是p/2。因此，由頁表和內零頭損失造成的全部開銷是：

開銷 = se/p + p/2

在頁面比較小時第一項（頁表大小）大；在頁面比較大時第二項（內零頭）大。最優值一定在中間某個地方，通過對p求導並令其等於零，我們得到方程：

-se/p2 + 1/2 = 0

從這個方程我們可以得出最優頁面大小的公式（只考慮碎片浪費和頁表所需的記憶體），結果是：

對於s = 128K和每個分頁表項目e = 8個位元組，最優頁面大小是1448位元組。考慮到其他因素（比如磁片速度），在實際中將使用1K或2K。大部分商品電腦使用的頁面尺寸在512位元組到64K之間。

4.5.4虛擬記憶體介面

到目前為止，我們一直假設虛擬記憶體對進程和程式師是透明的，他們所能看到的全部是在一個帶有較小的物理記憶體的電腦上的一個大的虛位址空間。在許多系統上這是對的，但在一些先進的系統上，程式師對記憶體映射有一定程度的控制並且能用非傳統的方式使用它。在這一節中，我們將簡略地介紹其中的少數幾個。

給予程式師對記憶體映射某些控制的一個原因是為了允許兩個或多個進程共用同樣的記憶體。如果進程可以命名它的記憶體區，一個進程就有可能把自己的一塊記憶體區的名字告訴另一個進程，使另一個進程也可以把它映射進來。兩個（或多個）進程共用同樣的頁面使高頻寬的共用成為可能—一個進程寫入共用記憶體區，另一個進程從中讀出。

頁面共用還可以用來實現高性能的消息傳遞系統。一般情況下在要傳遞消息時，資料是被從一個位址空間複製到另一個位址空間，代價相當高。如果進程可以控制他們的頁面映射，消息傳遞可以這樣實現：發送消息的進程把包含消息的頁面從它的頁面映射中刪除掉，而由接受進程把映射進來。這裡只需要複製頁面的名字，而不是複製全部資料。

還有一種先進的記憶體管理技術是分佈共用記憶體（distributed shared memory）（Feeley等，1995；Li和Hudak，1989；Zekauskas等，1994）。它的思想是允許在網路上的多個進程共用一套頁面，這些頁面可以，但不是必須，構成一個共用的單一線形位址空間。當一個進程引用一個當前沒有被映射的頁時就會引發一個頁面故障。頁面故障處理器，它可以在內核中也可以在用戶空間，隨即找到保存這個頁面的機器，並向這個機器發消息讓它刪除對這個頁的映射並把它從網路上發送過來。在頁面到達後，它將被映射進來，故障進程將被重新開機。

4.6 分段

到目前為止我們討論的虛擬記憶體都是一維的，虛位址從0到最大位址，一個位址跟著另一個位址。對許多問題來說，有兩個或多個獨立的位址空間可能比只有一個要好得多。比如一個編譯器在編譯過程中會建立許多表格，其中可能包括：

1. 保存的供列印清單用的源碼正文。

2. 符號表，包含變數的名字和屬性。

3. 包含整形和浮點常數的表。

4. 語法樹，包含程式的語法分析的結果。

5. 編譯器內部程序呼叫使用的堆疊。

前四個表隨著編譯的進行不斷地增長，最後一個在 編譯過程中以一種不可預計的方式增長和縮小。在一維記憶體中，這五個表只能被分配到虛位址空間中鄰接的塊中，如圖4-19所示。

圖4-19 在一維位址空間中，當有多個動態增長的表時，一個表可能會撞上另外一個。

讓我們考慮一下如果一個程式有異常大數量的變數時會發生什麼。位址空間中分給符號表的塊可能會被裝滿，但這時其他表中還有大量的空間。編譯器當然可以簡單地列印出一條資訊說由於變數太多編譯不能繼續進行，但在其他表中還有空間時這樣作似乎並不恰當。

另外一種可能的方法就是羅賓漢的劫富濟貧，從空閒空間過多的表中拿出一些空間給空間很少的表。這種處理是可以做得到的，但是它和自己管理覆蓋一樣—在最好的情況下也是一件另人討厭的事，在最壞的情況下則是一大堆沉悶並且毫無回報的工作。

我們真正需要的是一個能夠把程式師從管理表的收縮和擴張的工作中解放出來的辦法，就象虛擬記憶體使程式師不用再為怎樣把程式劃分成覆蓋塊而擔心一樣。

一個直觀並且非常通用的方法是在機器上提供多個互相獨立的稱為段(Segment)的位址空間，每個段由一個從0到最大的線性位址序列構成。各個段的長度可以是0到允許的最大值之間的任何一個值，不同的段的長度可以不同，而且通常也都不同。段的長度在運行期間可以改變，堆疊段的長度在資料被壓入時會增長，在資料被彈出時又會減小。

因為每個段都構成了一個獨立的位址空間，他們可以獨立地增長或減小而不會影響其他的段。如果一個在某個段中的堆疊需要更多的空間來增長，因為在它的位址空間中沒有任何其他東西阻擋，它就可以立刻得到它。段當然有可能被裝滿，但因為段通常很大，所以這種情況發生的可能性很小。在這種分段的或二維的記憶體中指明一個位址必須提供一個段號和一個段內位址。圖4-20圖示了用於前面討論過的編譯表格的分段記憶體。

圖4-20 分段的記憶體允許各個表獨立地增長或縮減。

我們要強調的是段是一個邏輯實體，程式師知道這一點並把它用作一個邏輯實體。一個段可能容納一個過程、或者一個陣列、或者一個堆疊、或者一些數值變數，但它一般不會同時包含多種不同的內容。

分段的記憶體除了能簡化對長度經常變動的資料結構的處理以外還有其他一些優點。如果每個過程都位於一個獨立的段中並且起始位址是0，把單獨編譯好的過程連結起來的操作就能大大簡化。當組成一個程式的所有過程都被編譯和連結好以後，一個對在段n的過程的程序呼叫將使用由兩個部分組成的位址 (n, 0) 轉到地址字0（入口點）。

如果隨後位於段n的過程被修改並重新進行了編譯，即使新版本的程式比老的要大，也不需要對其他的過程進行修改（因為沒有修改他們的起始位址）。在一維位址中，過程被一個挨一個緊緊地放在一起，中間沒有空隙，結果是修改一個過程的大小會影響其他無關過程的起始位址，而這又需要修改所有引用了被移動過程的過程，以使他們的引用指向這些過程的新地址。在一個有數百個過程的程式中這個操作的開銷可能是相當大的。

分段也有助於在幾個進程之間共用過程和資料，這方面一個常見的例子就是共用庫。運行先進視窗系統的現代工作站常常要把非常大的圖形庫編譯進幾乎每個程式。在分段系統中，可以把圖形庫放到一個單獨的段中由各個進程共用，從而不再需要在每個進程的位址空間中都有一份。雖然在純的分頁系統中也可以有共用庫，但是它要複雜得多，並且這些系統實際上是通過類比分段來實現的。

因為段是一個為程式師所知道的邏輯實體，比如一個陣列、一個堆疊，不同的段可以有不同種類的保護。一個過程段可以被指明為只允許執行，從而禁止對它的讀出和寫入；一個浮點陣列可以被指明為允許讀寫但不允許執行，任何試圖向這個段內的跳轉都將被俘獲。這樣的保護有助於抓住程式的錯誤。

你應該盡力理解為什麼保護在分段記憶體中有意義而在一維的分頁記憶體中則沒有。在分段記憶體中用戶知道每個段中包含了什麼，例如一般來說，一個段中不會既包含一個過程又包含一個堆疊，而是只包含其中的一個或另外一個。正是因為每個段只包含一種類型的物件，這個段就可以有針對這種特定類型的合適的保護。圖4-21對分段和分頁進行了比較。

考慮的因素

分 頁

分 段

需要程式師知道正在使用這種技術嗎？

不需要

需要

有多少個線性位址空間？

1

多個

全部位址空間可以超過物理記憶體大小嗎？

可以

可以

資料和過程可以被區別開並被分別保護嗎？

不能

能

能夠很容易地容納長度經常變化的表嗎？

不能

能

有助於在用戶間共用過程嗎？

不

是

為什麼發明這個技術？

為了得到大的線性位址空間而又不必去買更多的物理記憶體

為了允許程式和資料能被劃分為邏輯獨立的位址空間以説明實現共用和保護

圖4-21 分頁與分段的比較

頁的內容在某種程度上是隨機的，程式師甚至覺察不到分頁的事實。儘管在頁表的每個表項中放入幾位說明對應頁的存取權限是可能的，然而為了利用這一點程式師必須對他的位址空間中頁的界限保持跟蹤，當初正是為了避免這一類的管理工作，人們才發明分頁的。在分段系統中，因為使用者有所有的段都一直在記憶體中的幻覺－－也就是說他可以就象所有這些段都在記憶體中一樣去訪問－－他可以分別地保護各個段，不需要關心對他們覆蓋的管理工作。

4.6.1 純分段系統的實現

分段和分頁的實現從根本上是不相同的：頁是定長的而段不是。圖4-22(a)所示的是開始時包含了5個段的物理記憶體，現在讓我們考慮當段1被淘汰、比它小的段7放進它的位置後會發生什麼？這時的記憶體配置如圖4-22(b)所示，在段7與段3之間是一個未用區域，即一個空洞。隨後段4被段5代替，如圖4－22(c)；段3被段6代替，如圖4－22(d)。在系統運行一段時間後記憶體被劃分為許多塊，一些包含段，一些包含空洞，這種現象被稱為跳棋盤現象或外零頭（external fragmentation）。它在空洞上浪費了記憶體，而這可以通過緊縮來解決，如圖4-22(e)所示。

圖4-22 (a)-(d)棋盤形碎片的形成。(e)通過緊縮消除棋盤形碎片。

4.6.2 分段和分頁結合：MULTICS

如果一個段比較大，把它整個保存在記憶體中可能是不方便甚至是不可能的，這導致了對它進行分頁的想法，這樣只有哪些真正需要的頁才需要被調進來。有幾個著名的系統支援對段進行分頁，這一章我們將介紹第一個：MULTICS，在下一章我們將介紹一個更新的：Intel的Pentium。

MULTICS運行在Honeywell 6000電腦和它的一些派生機上，它為每個程式提供了最多218個段（超過250,000個），每個段最長65536個字長（36位）的虛位址空間。為了實現它，MULTICS的設計者決定把每個段看作一個虛擬記憶體並對它進行分頁，以結合分頁的優點（統一的頁面大小和在只使用段的一部分時不用把它全部調入記憶體）和分段的優點（易於程式設計、模組化、保護和共用）。

每個MULTICS程式都有一個段表，每個段對應一個描述符。因為段表可能會有大於25萬個表項，段表自己也是一個段並被分頁。一個段描述符包含了一個段是否在記憶體的標誌，只要一個段的任何一部分在記憶體這個段就被認為是在記憶體中，並且它的頁表也將在記憶體中。如果一個段在記憶體中，它的描述符將包含一個18位的指向它的頁表的指標〖參看圖4-23(a)〗。因為物理位址是24位元並且頁是按照64位元組的邊界對齊的（這隱含著頁位址的低6位是000000），所以在描述符中只需要18位元來存儲頁表位址。描述符還包含了段大小、保護位和一些其他的條目。圖4-23(b)是一個MULTICS段描述符。段在二級記憶體中的位址不在段描述符中，它在系統故障處理常式使用的另一個表中。

圖4-23 MULTICS的虛擬記憶體。(a)描述符段指向頁表。(b)一個段描述符，其中的數字是各個域的長度

每個段都是一個普通的虛位址空間，用與本章前面討論過的不分段的頁式記憶體相同的方式進行分頁。一般的頁面大小是1024位元組（儘管有一些MULTICS自己使用的段不分頁或以64位元組為單位進行分頁以節省記憶體）。

一個MULTICS的位址有兩部分構成：段和段內地址。段內位址又進一步分為頁號和頁內的記憶體字，如圖4-24所示。在進行記憶體訪問時，執行下面的演算法。

1. 根據段號找到段描述符。

2. 檢查該段的頁表是否在記憶體中，如果在就找到它的位置，如果不在就發出一個段故障。如果訪問違反了段的保護要求就發出一個故障（陷入）。

3. 檢查所請求虛頁的分頁表項目，如果頁面不再記憶體中則發出一個頁面故障，如果在記憶體中就從分頁表項目中取出這個頁在主存中的起始位址。

4. 把偏移位址加到頁的起始位址上，得到要訪問的字在主存的位址。

5. 最後進行讀或寫操作。

圖4-24 34位的MULTICS虛地址。

這個過程如圖4-25所示，為了簡單起見，我們忽略了描述符段自己也要分頁的事實。實際的過程是通過一個寄存器（描述符基址寄存器）找到描述符段的頁表，這個頁表指向描述符段的頁面。一旦找到了所需段的描述符，下面的定址過程就如圖4-25所示。

正如你現在肯定已經猜想到的，如果對於每條指令，上面所述的演算法都由作業系統來運行，那麼程式就不可能運行得很快。實際上，MULTICS硬體包含了16字高速TLB，對給定的關鍵字它能並行地搜索所有的表項，如圖4-26所示。當一個位址被送到電腦時，定址硬體首先檢查虛位址是不是在TLB中，如果在，就直接從TLB中取得頁框號生成要訪問的字的實際位址，而不必到描述符段或頁表中去查找。

圖4－25 兩部分組成的MULTICS位址到主存位址的轉換。

圖4－26 簡化的MULTICS TLB。兩種頁面長度的存在使實際的TLB更加複雜。

TLB中保存著16個最近訪問的頁。工作集小於TLB容量的程式將隨著整個工作集被裝入TLB中而逐漸達到穩定，開始高效地運行。如果頁不在TLB中，描述符和頁表才會被引用以找出頁框位址，並更新TLB使它包含這個頁，最久未使用的頁被淘汰出TLB。年齡域對哪個表項是最久未訪問過的保持跟蹤。之所以使用TLB是為了並行地和所有表項的段號和頁號進行比較。

4.6.3 分段和分頁結合：Intel 的 Pentium

Pentium（和Pentium Pro）在許多方面都與MULTICS類似，其中包括既有分段機制又有分頁機制。MULTICS有256K個獨立的段，每個段最長可以有64K個36位字，Pentium有16K個獨立的段，每個段最多可以容納十億個32位字。這裡雖然段的數目較少，但是大的段尺寸要重要得多，原因是幾乎沒有程式需要1000個以上的段，但是有很多程式需要一個段能容納數百萬位元組的資料。

Pentium虛擬記憶體的核心是兩張表，LDT（局部描述符表）和GDT（通用描述元表）。每個程式都有自己的LDT，但是同一電腦上所有的程式共用一個GDT。LDT描述局部於每個程式的段，包括代碼、資料、堆疊等，GDT描述系統段，包括作業系統自己。

為了訪問一個段，Pentium程式必須把這個段的選擇符(Selector)裝入機器的六個段寄存器中的某一個中。在運行過程中，CS寄存器保存程式碼片段的選擇符，DS寄存器保存資料段的選擇符，其他的段寄存器不太重要。每個選擇符是一個16位數，如圖4-27所示.

圖4-27 一個奔騰的選擇符。

選擇符中的一位元指出這個段是局部的還是全域的（即，它是在LDT中還是在GDT中），其他的13位是LDT或GDT的入口號，因此這些表長度被限制在最多容納8K個段描述符，還有兩位和保護有關，我們將在後面討論。描述符0是禁止使用的，它可以被安全地裝入一個段寄存器中用來表示這個段寄存器目前不可用，如果使用會引起一次陷入。

在選擇符被裝入段寄存器時，對應的描述符被從LDT或GDT中取出裝入微程式寄存器中，以便於快速地訪問。一個描述符由8個位元組構成，包括段的基底位址、長度和其他資訊，如圖4-28所示。

圖4-28 奔騰的程式碼片段描述符，資料段有輕微的差別。

選擇符的格式經過了聰明的挑選，定位描述符十分方便。首先根據第二位元選擇LDT或GDT；隨後選擇符被拷貝進一個內部寄存器中並且它的低3位被清0；最後LDT或GDT表的位址被加到它上面，得出一個直接指向描述符的指標。例如選擇符72指向GDT的第九個入口，它位於GDT＋72。

現在讓我們跟蹤一個（選擇符，偏移）對被轉換為物理位址的過程。只要微程式知道哪個段寄存器正在被使用，它就能從內部寄存器中找到對應于這個選擇符的完整的描述符，如果段不存在（選擇符為0）或已被換出，則會發生一次陷入。

它隨後檢查偏移量是否超出了段的結尾，如果是也會發生一次陷入。從邏輯上來說，在描述符中應該簡單地有一個32位的域給出段的長度，但實際上只有20位可以使用，因此Pentium採用了一種不同的方案。如果Gbit（細微性）域是0，則Limit域是精確的段的長度，最大1MB；如果是1，Limit域以頁而不是位元組為單位給出段的長度。Pentium頁的長度是固定的4K位元組，因此20位元組足夠最大232位元組的段使用。

假設段在記憶體中並且偏移也在範圍內，Pentium接著把描述符中32位的基底位址和偏移量相加形成所謂的線性位址（linear address），如圖4-29所示。為了和只有24位基底位址的286相容，Pentium基底位址被分為三片分佈在描述符的各個位置。實際上，基址域允許各個段的起始位址在32位元線性位址空間的任何位置。

圖4-29 （選擇符、偏移）到線形位址的轉換。

如果分頁被禁止（通過一個全域控制寄存器中的一位元），線性位址就被解釋為物理位址並被送往記憶體用於讀寫。因此在分頁被禁止時，我們就得到了一個純的分段方案，各個段的基址在它的描述符中。順便提一句，段允許互相覆蓋，這可能是因為驗證所有的段都互不重疊太麻煩太費時間的緣故。

在另一方面，如果分頁被允許，線性位址將被解釋成一個虛擬位址並通過頁表映射成為物理位址，很象我們前面講過的例子。這裡唯一真正複雜的是在32位虛位址和4K頁的情況下，一個段可能包含多達一百萬個頁，因此Pentium使用了一種兩級映射以在段較小時減小頁表尺寸。

每個運行程式都有一個由1024個32位表項組成的頁目錄（page directory），它的位址由一個全域寄存器指出。目錄中的每個表項都指向一個也包含1024個32位表項的頁表，分頁表項目指向頁框，這個方案如圖4-30所示。

圖4-30 線形位址到物理位址的映射。

在圖4-30(a)中我們看到線性位址被分為三個域：Dir、Page、和Off。Dir域被作為索引在頁目錄中找到指向正確的頁表的指標；隨後Page域被用作索引在頁表中找到頁框的物理位址；最後，Off被加到頁框的位址上得到需要的位元組或字的物理位址。

每個分頁表項目是32位，其中20位是頁框號，其餘的位元包含了由硬體設置供作業系統使用的訪問位元和修改位、保護位、和一些其他有用的位。

每個頁表有描述1024個4K頁框的表項，因此一個頁表可以處理4M的記憶體。一個小於4M的段將有一個只有一個表項的頁目錄，這個表項指向一個也是唯一的頁表。通過這種方法，短的段的開銷只是兩個頁，而不是一級頁表時的一百萬個分頁表項目。

為了避免重複的記憶體訪問，Pentium和MULTICS一樣，也有一個TLB把最近使用過的頁目錄號-頁號組合映射為頁框的物理位址，只有在當前的組合不在TLB中時圖4-30所示的機制才被真正執行並更新TLB。

略微思考一下我們就會發現在使用分頁時，讓描述符的基址域不為0確實是毫無意義的。基址的全部作用就是使小的偏移量使用頁目錄中間的表項而不是開頭，之所以包含基址域是為了允許純的分頁和與總是禁止分頁的286相容（即，286只有純的分段，沒有分頁）。

還有一點值得注意的是如果一個應用程式不需要分段，只需要單個分頁的32位位址空間，這也是可以的。這時所有的段寄存器都被設置為同一個選擇符，它的描述符基址是0，長度是最大，這樣只有一個位址空間被使用，指令偏移將成為線性位址，實際上就是一般的分頁。

不管怎麼說，我們不得不稱讚Pentium的設計者，向他們提出的是互相衝突的目標：實現純的分頁、純的分段、和段頁式管理，還要與286相容，而他們高效地實現了所有的目標，最終的設計非常簡潔乾淨。

儘管我們已經簡單地討論了Pentium虛擬記憶體的全部體系機構，關於保護我們還是值得再說幾句的，因為它和虛擬記憶體聯繫很緊密。Pentium的保護機構和虛擬記憶體一樣非常象MULTICS模型，它支援四個保護級，0級許可權最高，3級最低，如圖4－31所示。在任何時刻，運行程式都處在由PSW中的兩位所指出的某個保護級上，系統中的每個段也有一個級別。

圖4-31 奔騰中的保護。

只要程式只使用與它同級的段，一切都會很正常。對更高級別資料的存取是允許的，而對更低級別的資料的存取是非法的並會引起陷入。調用不同級別（更高或更低）的過程是允許的，但是要通過一種被嚴格控制著的方法。為執行越級調用，CALL指令必須包含一個選擇符而不是位址，選擇符指向一個稱為調用門（call gate）的描述符，由它給出被調用過程的位址，因此要跳轉到任何一個不同級別的程式碼片段的中間都是不可能的，只有正式指定的入口點可以使用。保護級和調用門的概念來自MULTICS，在那裡他們被稱為保護環（protection rings）。

這個機構的一種典型的應用如圖4-31所示。在0級是作業系統內核，處理I/O、記憶體管理、和其他關鍵的操作；在1級是系統調用處理常式，使用者程式可以通過調用這裡的過程執行系統調用，但是只有一些特定的和受保護的過程可以被調用；在2級是庫過程，它可能是由很多正在運行的過程共用的，使用者程式可以調用這些過程，讀取他們的資料，但是不能修改他們；最後，使用者程式運行在級別3上，受到的保護最少。

陷入和中斷使用了一種和調用門類似的機構，他們引用的也是描述符而不是絕對位址，這些描述符指向將被執行的特定的過程。圖4-28中的Type域用於區別程式碼片段、資料段、和各種門。

4.7 MINIX記憶體管理概覽

MINIX的記憶體管理是非常簡單的：既不分頁也不交換。存儲管理器保存著一張按照記憶體位址排列的空洞清單，當由於執行系統調用FORK或EXEC需要記憶體時，系統將用首次適配演算法對空洞清單進行搜索找出一個足夠大的空洞。一旦一個程式被裝入記憶體，它將一直保持在原來的位置直到運行結束，它永遠不會被換出或移動到記憶體的其他位置去，為它分配的空間也不會增長或縮小。

對這個方案應該進行一些解釋，它是考慮了三個因素後的結果：(1) MINIX是用於個人電腦的，而不是大型的分時系統；(2) 希望MINIX在所有的IBM PC上運行；(3) 希望系統的實現很直觀以利於在其他小型電腦上實現。

第一個因素意味著一般來說，運行進程的數目將比較少，在典型的情況下記憶體足夠容納所有的進程，並且還會有一些空餘。在這裡交換是不需要的，它只會增加系統的複雜性，不要交換使得代碼很簡單。

讓MINIX在所有IBM PC兼容機上運行的願望也對記憶體管理的設計有很大的影響。在這個家族中最簡單的系統使用8088處理器，它的記憶體管理結構十分原始，不支援任何形式的虛擬記憶體，並且甚至不檢查堆疊溢位。這個缺點對進程在記憶體中佈局的方案有很大的影響。這些限制在使用80386，80486或Pentium的處理器時都不再存在，但是使用這些特性將使MINIX和許多目前仍然可以使用或在使用的低端機器不相容。

對相容性的考慮也要求記憶體管理方案儘量簡單。如果MINIX使用了分頁和分段，要把它移植到沒有這些特性的機器上將是十分困難的，如果不是不可能的話。通過對硬體所能提供的支援作最少的假設可以使MINIX能移植的機器數量大大增加。

MINIX另一個不同尋常的方面是它的記憶體管理實現的方法。它不是內核的一部分，而是由一個運行在用戶空間、使用標準的消息機構和內核通訊的管理進程完成的。處於伺服器級別的記憶體管理器的位置如圖2-26所示。

把記憶體管理器移到內核外面是把決策和機構分開的一個例子，哪個進程應該被放在記憶體中哪個位置的決定（決策）是由記憶體管理器作出的，而具體的為進程設置記憶體映射（機構）的操作是由在內核中的系統任務完成的。這個劃分使得修改記憶體管理策略（演算法等）比較容易實現，不需要修改作業系統底層。

大部分記憶體管理器的代碼是用來處理和記憶體管理有關的系統調用的，主要是FORK和EXEC，而不是僅僅操作進程和空洞的表。在下一節中我們將看到記憶體的佈局，在再下一節中我們將鳥瞰記憶體管理器是怎樣處理記憶體管理系統調用的。

4.7.1 記憶體佈局

簡單的MINIX進程使用結合的I和D空間，這時進程所有的部分（正文、資料、和堆疊）共用一個區塊，它是作為一個塊申請和釋放的。進程也可以被編譯為使用分開的I和D空間。為了清楚起見，我們先討論簡單模型的記憶體分配。使用獨立的I和D空間的進程能更有效地利用記憶體，但利用這個特徵的優點會使事情變得複雜。我們將在勾畫完簡單的之後討論複雜的。

在MINIX中有兩種情況需要分配記憶體。第一種是在一個進程執行fork時，為子進程分配所需要的空間；第二種是在一個進程通過EXEC系統調用修改它的記憶體映象時，老的映象被作為空洞送到空閒表，需要為新的映象分配記憶體。新的映象可能會在記憶體中不同於被釋放的記憶體的地方，它的位置取決於在什麼地方能找到合適的空洞。在進程因為執行exit或被信號殺死而結束時記憶體也將被釋放。

圖4-32示出了這兩種分配記憶體的方法。在圖4-32(a)中我們看到在記憶體中有兩個進程，A和B。如果A執行了fork，我們將得到圖4-32(b)所示的情況，子進程是A的一個精確拷貝。如果子進程執行了檔C，記憶體看起來將如圖4-32(c)所示，子進程的映象被C取代。

圖4-32 記憶體分配。(a)原來的情況。(b)A執行FORK以後。(c)A的子進程執行了EXEC以後。陰影區域是未用記憶體。這個進程是一個普通的I和D結合的進程。

需要注意的一點是用於子進程的老的記憶體是在用於C的新的記憶體分配之前被釋放的，因此C可以使用子進程的記憶體。這樣，一系列的FORK和EXEC對（比如shell設置一個管道線）將會使所有的進程都是鄰接的，中間沒有空洞。如果我們在老的記憶體被釋放之前就分配新記憶體，就有可能產生空洞。

由FORK或EXEC系統調用引起而分配記憶體時，一定數量的記憶體將給予新進程。在前一種情況下，分配的數量和父進程的相同；在後一種情況下，記憶體管理器將分配被執行檔的頭部所指明的數量。一旦分配完畢，決不會再給進程分配一絲一毫的記憶體。

前面所討論的適用於被編譯為具有結合的I和D空間的程式。具有獨立的I和D空間的程式可以利用增強方式記憶體管理的一個叫做共用正文的優點。當這樣的一個進程FORK時，只需要分配為新進程做一個堆疊段和資料段拷貝所需數量的記憶體。父進程和子進程將共用已經由父進程使用的執行代碼。當一個這樣的進程執行EXEC時，系統將查找進程表看是否有另外一個進程已經在使用需要的執行代碼，如果找到了就只為資料和堆疊分配新記憶體，已經在記憶體的正文將被共用。共用正文使得進程的結束複雜化了。在一個進程結束時它總是要釋放它的資料和堆疊佔用的記憶體，但是只有在搜索了進程表並發現沒有其他進程在使用正文段後，才釋放正文段所佔用的記憶體。因此如果一個進程在啟動時裝入了自己的正文，而在結束時它的正文正在被一個或多個其他進程共用，那麼這個進程啟動時分配的記憶體就會比結束時釋放的多。

圖4-33 (a)一個保存在磁片檔中的程式。(b)一個進程的內部記憶體佈局。在兩圖中最低的磁片或記憶體位址都是在底端，最高的位址都是在頂端。

圖4-33所示的是，一個程式怎樣作為磁片檔保存和怎樣被轉換為MINIX進程的記憶體佈局。磁片檔的頭部包含了進程映象各部分的大小以及總的大小的資訊。在具有共同的I和D空間的程式頭部，有一個域指出正文和資料部分的總長度，這些部分被直接拷貝到記憶體映象中。映象中的資料部分增擴了頭部bss域指出的數量，擴大的部分被清0，用於未初始化的靜態資料。總共分配的記憶體數量是由檔頭中的“total”域說明的。假如一個程式有4K正文段、2K的資料與bss、和1K堆疊，檔頭中說明的需要分配的記憶體總量是40K，那麼在堆疊段和資料段之間的未用記憶體將是33K。在磁片上的程式檔中還可以包含一個符號表，它僅用於調試，不裝入記憶體。

如果程式師知道，程式 a.out 的資料和堆疊段增長所需要的記憶體總共最多是10K，他可以使用下面的命令：

chmem = 10240 a.out

這個命令將修改檔頭，使得在EXEC的時候記憶體管理器將為這個程式分配比初始的正文和資料段長度的和多10240位元組的空間。在上面的例子中，當哪個文件隨後被EXEC時將為它分配16K的記憶體，其中高端的1K用於堆疊，9K是空隙，供資料區或堆疊或者兩者共同增長用。

對於使用獨立的I和D段的的程式（由檔頭中連接器設置的一位指出），檔頭中的total域只對結合的資料段和堆疊段有用。一個有4K正文、2K資料、1K堆疊，total域為64K的程式將被分配68K的空間（4K指令空間，64K資料空間），留出61K空間供資料段和堆疊在運行時使用。資料段的界限只有通過BRK系統調用才能修改，BRK所作的全部工作就是檢查新的資料段是否超越了當前的堆疊指標，如果沒有，就修改一些內部表格以反映這個變動。這個操作涉及的只是當初分配給進程的記憶體區，作業系統不會分配額外的記憶體。如果新的資料段闖入了堆疊，這個調用將以失敗告終。

採用這個策略是為了能在使用8088處理器的IBM PC機器上運行MINIX。在這種處理器中，硬體不檢查堆疊溢位，一個使用者程式可以在作業系統不知道的情況下把任意多個字推入堆疊。在具有更先進的記憶體管理硬體的電腦上，開始時堆疊被分配給一定數量的記憶體，如果堆疊試圖增長超過這個數量就會引起一個作業系統陷入，然後如果可能，系統就為堆疊再分配一些空間。然而在8088上這個陷入不存在，因為堆疊會在不發出警告的情況下快速增長，所以讓堆疊段與除了一大塊未用記憶體之外的任何物件相鄰都是十分危險的。MINIX的設計使得當它在一個具有更好的記憶體管理的電腦上實現時，修改MINIX記憶體管理器是非常直觀的。

這裡應該指出一個容易混淆的地方，當我們使用“段”這個詞時，我們指的是作業系統定義的一個記憶體區域。Intel的80x86有一套內部的“段寄存器”和（在更高級的處理器上）“段描述符表”，他們為“段”提供硬體支援。Intel硬體設計者的段的概念類似於，但又不總是和MINIX定義和使用的段相同。在本書中所有對段的引用都應該被解釋為對由MINIX資料結構所描述的記憶體區域的引用，在談到硬體時我們將明確地使用段寄存器或段描述符。

這個警告可以推而廣之。硬體設計者經常試圖為期望在他們機器上運行的作業系統提供支援，因此用來描述寄存器和處理器結構其他方面的術語通常會反映對這些特徵將來用途的設想。這些特徵對作業系統的實現者經常是有用的，但是使用的方法可能和硬體設計者預見的不同，同樣的詞在描述作業系統的一個方面和底層的硬體時有不同的含義就有可能導致誤解。

４.７.２ 消息處理

象MINIX所有其他部分一樣，記憶體管理器是消息驅動的。在系統初始化完成之後，記憶體管理器就進入它的主迴圈，包括等待消息、執行消息中包含的請求、和發送應答。圖4-34中的列表給出了合法的消息類型、他們的輸入參數、和在應答消息中送回的值。

圖 4-34 與記憶體管理器通訊時使用的消息類型、輸入參數、應答值。

FORK、EXIT、WAIT、WAITPID、BRK、和EXEC很明顯地與記憶體分配和釋放緊密相關；KILL、ALARM、PAUSE、SIGACTION、SIGSUSPEND、SIGPENDING、SIGMASK、SIGRETURN等都與信號有關，因為當一個信號殺死進程時將釋放進程的記憶體，所以他們也有可能影響記憶體；REBOOT對整個作業系統都有影響，但它的第一個任務是以一種受控制的方法發送信號結束所有的進程，因此記憶體管理器是恰當的對其進行處理之處；七個GET/SET與記憶體管理根本沒有任何關係，他們與檔案系統也沒有任何關係。但是因為任何一個系統調用都是由兩者中的一個處理的，而且檔案系統已經足夠大了，因此只能把他們放在記憶體管理器中；PTRACE用於調試，它也是由於同樣的原因放到了這裡。

最後一條消息KSIG不是系統調用，它由內核用來向記憶體管理器發送SIGINT、SIGQUIT、SIGALRM等源自內核的信號。

系統提供庫常式sbrk，但並無系統調用SBRK。該庫常式把參數說明的增量或減量加到當前長度上，並以此作為參數調用BRK。同樣地，不存在獨立的系統調用geteuid和getegid，調用GETUID和GETGID既返回有效標識又返回真實標識。同樣地，GETPID返回調用進程及其父進程的pid。

用於消息處理的一個關鍵資料結構是在table.c中（16515行）說明的call\_vec表。它包含了指向處理不同類型消息的過程的指標。當一條消息來到記憶體管理器時，主迴圈抽出消息類型並把它放在全域變數mm\_call中，隨後這個值被作為call\_vec的索引以找到處理新到消息的過程。緊接著調用這個過程執行系統調用。它返回的值被放到應答消息中送回檔用者，以報告調用是成功還是失敗。這個機制與圖1-16的類似，差別在於它在用戶空間而不是在內核。

4.7.3 記憶體管理器資料結構和演算法

記憶體管理器有兩個關鍵資料結構：進程表和空洞表。我們將逐個討論。

在圖2-4中我們看到進程管理需要進程表的一些域，記憶體管理需要另外一些域，檔案系統又需要其他一些域。在MINIX中，作業系統的這三個部分都有自己的進程表，每個部分的表僅包含了自己需要的域。為了簡單起見，表項是精確對應的，因此記憶體管理器的表項k和檔案系統的表項k對應的是同一個進程。為了保持同步，在進程創建或結束時，這三個部分都要更新他們的表以反映新的情況。

記憶體管理器的進程表叫做mproc，定義在/usr/src/mm/mproc.h中。它包含了與進程記憶體分配有關的全部域和一些附加的資訊。最重要的域是mp\_seg陣列，它有三個表項，分別用於正文、資料、和堆疊段。各個表項是一個由虛位址、物理位址、和段長度組成的結構。他們都是用塊（click）而不是位元組來量度的。塊的大小是依賴於實現的，在標準的MINIX中是256位元組。所有的段必須開始於塊邊界並且佔據整數個塊。

記錄記憶體分配的方法如圖4-35所示，該圖中的一個進程有3K正文、4K資料、1K的空隙、和隨後的2K堆疊，分配的全部記憶體是10K。假設這個進程沒有獨立的I和D空間，在圖4-35(b)中我們看到這三個段各自的虛位址、物理位址、和長度域。在這個模型中，正文段總是空的，資料段包含了正文和數據。當進程引用虛位址0時，不管是跳轉到它還是讀它（即，在指令空間還是在資料空間），將使用物理位址0x32000（十進位是200K），這個位址位於塊0x320。

圖4-35 (a)記憶體中的一個進程。(b)這個進程在I和D空間非獨立時的記憶體表示。(c)I和D空間獨立時的記憶體表示。

值得注意的是堆疊的起始虛位址取決於分配給進程的記憶體總量。如果為了提供更大的動態空間（在資料段和堆疊之間更大的間隙）而用chmem命令修改了檔頭，那麼在下一次檔執行時堆疊將從一個更高的虛位址開始。如果堆疊增長了一塊，那麼堆疊的表項應該從三元組(0x20, 0x340, 0x8)變成三元組 (0x1F, 0x33F, 0x9)。

8088硬體沒有段界限陷入，在32位處理器上，MINIX定義堆疊的方法使得在堆疊段已經覆蓋資料段之前不會激發陷入。因此，在下一次BRK系統調用之前不會對堆疊表項做出上面的修改。那時，作業系統會專門讀取SP並重新計算段表項。在具有堆疊陷入的機器上，一旦堆疊超出它所在的段，就可以立即更新堆疊段表項。然而在32位Intel處理器上的MINIX並沒有這樣做，下面下面我們將討論其原因。

在前面我們提到，硬體設計者的努力並不總是能恰好產生軟體設計者所需要的。即使是在Pentium的保護模式下，當堆疊增長超過它的段時MINIX也不會發生陷入。儘管處於保護方式的Intel硬體檢測超出段界限（由段描述符定義，例如圖4-28中的那個）的記憶體訪問，但是在MINIX中資料段描述符和堆疊段描述符總是一樣的。MINIX定義的資料和堆疊各自使用這個空間的一部分，因此兩者都可以擴展進入位於他們之間的空隙中。在這裡，只有MINIX可以管理這些，因為對硬體來說間隙既是資料段也是堆疊段合法的一部分，CPU根本無法檢測涉及間隙的錯誤。當然，硬體能夠檢查非常大的錯誤，例如對超出結合的資料-間隙-堆疊區域的訪問。這能夠保護一個進程免受另外一個進程錯誤的影響，但卻不足以保護一個進程免受自己錯誤的影響。

在這裡我們作出了一個設計決定。我們意識到在放棄共用的硬體定義的段方面存在不同意見，這種段允許MINIX動態地重分配間隙區域。另一種選擇是用硬體定義不重疊的堆疊和資料段，這樣作可以在某種程度上提供更高的安全性，以避免某些錯誤，但卻使得MINIX記憶體的使用方面更加緊張。任何想評估一下另一種方法的人都可以得到它的源碼。

圖4-35(c)所示的是4-35(a)的記憶體佈局在具有獨立的I和D空間情況下的段表項。這裡，正文和資料段的長度都不為零。圖4-35(b)和圖4-35(c)中的mm\_seg陣列主要是用於把虛位址映射成物理位址。給定一個虛位址和它所屬的空間，那麼確定虛位址是否合法（即它是否落在一個段的內部）以及在合法時它對應的物理位址是一件很簡單的事情。例如，內核過程umap就為I/O任務和用戶空間與內核空間間的拷貝實現這種映射。

圖4-36 (a)一個具有獨立的I和D空間進程的記憶體映射，與上圖相同。(b)第二個進程啟動後的記憶體佈局，他們通過共用正文執行相同的程式映射。(c)第二個進程的記憶體映射。

一個進程的資料和堆疊區域的內容可能會隨著進程的執行而改變，但正文不會改變。幾個進程同時執行同一個程式拷貝的情況是很常見的，例如，幾個用戶可能會同時執行相同的shell，使用共用正文能提高記憶體的利用率。當EXEC要裝入一個程式時，它首先打開保存著該程式磁片映射的檔並讀入檔頭，如果進程使用的是獨立的I和D空間，系統將搜索每個mproc表項的mm\_dev、mm\_ino、和mm\_ctime域，這些域保存著相應進程正在執行的映射所在的設備、i結點號和修改時間。如果找到了一個正在執行相同程式的進程，那麼就沒有必要為正文的另一個拷貝分配記憶體，只要把新進程記憶體映射的mm\_seg[T]部分初始化為指向已經裝入正文段的位置，只有資料和堆疊部分被設置成新分配的記憶體。這個過程如圖4-36所示。如果程式使用的是結合的I和D空間或合適的進程沒有找到，記憶體將象圖4-35那樣分配，新進程的正文和資料將從磁片上拷貝進來。

除了段資訊之外，mproc還保存著進程自己的進程號(pid)和父進程的進程號、使用者號(uid)和組號(gid)（真實和有效的）、關於信號的資訊、以及在進程已經結束而父進程還沒有執行對它的WAIT時的終止狀態。

記憶體管理器另外一個主要的表是空洞表，定義在alloc.h中，它按照記憶體位址遞增的順序列出了記憶體中的各個空洞。資料和堆疊段之間的空隙不認為是空洞，他們已經被分配給進程了，所以他們沒有被包含在空洞表中。每個空洞表項有三個域：以塊為單位的空洞的基底位址、以塊為單位空洞的長度、和一個指向表中下一個入口的指標。這個表是單向連結的，所以從任何一個空洞開始找到下一個空洞很容易，但是如果要找上一個就必須從頭開始搜索直到找到給定的空洞。

用塊而不是位元組為單位記錄段和空洞的各種資訊的原因非常簡單：在16位元狀態模式下，記憶體位址用16位元整數記錄，在塊為256位元組時可以支援最大16M的記憶體；在32位元模式下，位址域可以引用高達240位元組的位址，即1024G。

在空洞表上的主要操作是分配一塊指定大小的記憶體和歸還一塊已經分配的記憶體。在分配記憶體時，首先從最低的位址開始搜索空洞表，直到找到一個足夠大的空洞（首次適配）；隨後，從空洞中減去段需要的空間，或在極少的大小恰好相等的情況下從空洞表中刪除空洞來為段分配空間。這個方案簡單快速，但是存在著內零頭（最後一個塊可能會浪費多達255位元組空間，因為分配的總是整數個塊）和外零頭的問題。

在進程結束並被清理完後，它的資料和堆疊記憶體區歸還給空洞表。如果進程使用的是結合的I和D空間，它的全部記憶體都將被釋放；如果進程使用的是獨立的I和D空間並且搜索進程表發現沒有其他進程在共用其正文，那麼它的正文區也將被釋放。因為在共用正文時正文和資料區域未必相鄰，所以也可能會歸還兩塊記憶體。對於每個被歸還的記憶體區域，如果它的任何一側或者兩側鄰接的區域也是空洞，則將他們合併，因此決不會出現鄰接的空洞。這樣，空洞的數量、位置和大小在系統運行期間將不斷變化。一旦所有的使用者進程都終止了，那麼全部可用記憶體將又一次可供分配。然而這未必意味著只有一個空洞，因為實體記憶體可能會被作業系統不可用的區域截斷，就象在IBM兼容機中那樣唯讀記憶體（ROM）和為I/O傳輸保留的記憶體把可用記憶體分為640K以下和1M以上兩塊。

4.7.4 FORK、EXIT、和WAIT系統調用

在創建和撤銷進程時必須分配或釋放記憶體、必須更新進程表，包括由內核和FS保存的部分。這些操作都由記憶體管理器協調。進程的創建是由FORK完成的，這時將執行圖4-37所示的一系列步驟。

1. 檢查進程表是否滿了。

2. 試為子進程的資料和堆疊分配記憶體。

3. 把父進程的資料和堆疊複製到子進程的記憶體中。

4. 找到一個空閒的進程表項並把父進程的表項複製進去。

5. 在進程表中輸入子進程的記憶體映射。

6. 為子進程選擇一個進程號。

7. 告訴內核和檔案系統子進程的情況。

8. 向內核報告子進程的記憶體映射。

9. 向父進程和子進程發送應答資訊。

圖 4-37 執行FORK系統調用需要執行的步驟。

半途中斷FORK調用是困難的也是不方便的，所以為了比較容易地判斷進程表是否有空閒位置，記憶體管理器總是保存著當前存在的進程的數目。如果進程表沒有滿，就嘗試為子進程分配記憶體，對於具有獨立的I和D空間的程式只需要為新的資料和堆疊段分配足夠的空間。只要這一步成功了，FORK就保證能成功。隨後填寫新分配的記憶體、找到一個進程表項並填寫、選擇pid、通知系統的其他部分一個新進程已經被創建。

在下列兩個事件都已經發生的情況下進程才會完全終止：(1)進程自己已經退出（或已經被一個信號殺死），(2)它的父進程已經執行了WAIT系統調用以觀察發生了什麼。已經退出或被殺死而它的父進程還沒有為它執行WAIT的進程將進入某種掛起狀態，有時被稱為僵死狀態(Zombie State)，這種進程不再參與調度，它的報警時鐘被關閉（如果原來是開的），但它仍將留在進程表中。它的記憶體被釋放。僵死是一種臨時狀態，很少會持續較長的時間，當父進程最後執行WAIT時，將釋放進程表項，並通知檔案系統和內核。

這樣就出現了一個問題，如果要結束進程的父進程已經死了怎麼辦？如果不採取特殊處理，要結束的進程將永遠處於僵死狀態。對於這種情況，系統將修改進程表使這種孤兒進程成為init的子進程。在系統啟動時，init讀取/etc/ttytab檔以得到所有終端的清單，並為每個終端fork出一個註冊進程，隨後阻塞，等待進程結束。通過這個方法可以使孤兒很快被清理掉。

4.7.5 EXEC系統調用

當從終端輸入一條命令被時，shell fork 出一個新進程，然後由它執行請求的命令。本來讓一條系統調用一次完成FORK和EXEC兩個操作是可以的，但是由於下述理由：“為了易於實現I/O重定向”，他們被作為兩條不同的調用提供。在shell執行fork時，如果輸入是重定向的，子進程在執行命令前先關閉標準輸入再打開新的標準輸入，這樣新創建的進程將繼承重定向過的標準輸入，對標準輸出的處理方法相同。

EXEC是MINIX中最複雜的系統調用，它用新的記憶體映射替換當前的記憶體映射，包括設置新的堆疊。它通過圖4-38所示的一系列步驟完成其工作。

1. 檢查許可權—檔是否可執行？

2. 讀取檔頭得到各段長度和總長度。

3. 從調用者處取參數和環境。

4. 分配新記憶體和釋放舊記憶體。

5. 把堆疊複製到新的記憶體映射中。

6. 把資料（可能還有正文）段複製到新的記憶體映射中。

7. 檢查處理setuid、setgid位。

8. 設置進程表項。

9. 告訴內核進程現在是可運行的。

圖 4-38 執行EXEC系統調用需要執行的步驟。

每一步都是由更小的步驟組成的，其中有些可能會失敗，例如記憶體可能不夠。執行檢測的順序經過了仔細的選擇，以保證直到確定EXEC肯定會成功時才會釋放舊的記憶體映射，以避免出現既不能生成新的記憶體映射又沒有舊的記憶體映射可以回去的窘迫的情形。通常EXEC 不會返回，但是如果它失敗了，調用進程必須再次得到控制，並得到一個錯誤指示。

對圖4-38中的幾個步驟應該作更多的注釋。首先是是否有足夠空間的問題。首先需要檢查新進程是否可以共用其他進程的正文記憶體，以確定它所需的記憶體大小，然後搜索空洞表以檢查在釋放老的記憶體前是否有足夠的物理記憶體—如果老的記憶體先被釋放了而記憶體又不夠，我們將很難再恢復老的記憶體映射。

然而，這個測試是過於嚴格了，它有時會拒絕有些實際上可以成功的EXEC調用。例如，假設執行EXEC系統調用的的進程佔用了20K，並且它的正文段沒有被其他進程共用。再假設有一個30K的空洞，而新的映射需要50K。通過在釋放以前的測試，我們將發現只有30K空間可用，因此會拒絕這個調用。但是如果我們先釋放，這個調用也許會成功，取決於新的20K空洞是否與30K的空間相鄰因而可以合併。稍複雜一點的實現可以更好地處理這種情況。

如果EXEC後的進程使用獨立的I和D空間，則另一個可能的改進是搜索兩個空洞，一個用於正文段一個用於資料段，兩個段不需要鄰接。

一個更微妙的問題是可執行檔是否能放得進虛位址空間。這個問題的原因在於記憶體是以256位元組的塊而不是位元組為單位分配的。由於整個記憶體管理都是以塊為單位的，每個塊必須只屬於一個段，不允許出現象一半資料一半堆疊這樣的情形。

為了看到這個限制怎樣會帶來麻煩，請注意16位元系統（8086和80286）的位址空間是限制在64K的，它可以被分為256個塊。假設一個具有獨立I和D空間的程式有40000位元組的正文、32770位元組的資料和32760位元組的堆疊。資料段佔用了129個塊，其中最後一塊只用了一部分，然而整個塊還是資料段的一部分。堆疊段是128塊。儘管此時資料段和堆疊段需要的位元組數之和沒有超出虛位址空間，但他們加起來超過了256塊，所以並不能一起放入虛位址空間。在理論上，這個問題在所有的塊大於1位元組的機器上都存在，但是在實際上，因為Pentium系列處理器允許大的（4GB）的段，這個問題很少出現。

另一個重要的問題是如何設置初始堆疊。通常使用庫函數execve來調用EXEC，其形式是：

execve(name, argv, envp);

這裡name是指向將被執行的檔案名的指標、argv是指向一個指標陣列的指標，陣列的每個成員指向一個參數、envp是指向一個指標陣列的指標，陣列的每個成員指向一個環境串。

如果僅僅把上面三個指標放在一條消息中傳給記憶體管理器，而讓它自己去取檔案名和兩個陣列來實現EXEC是非常簡單的。但是，記憶體管理器不得不一次一個地取各個參數和串，因為記憶體管理器無法事先知道各個參數或串的長度，所以對每個參數或串至少需要向系統任務發一條甚至更多的消息。

為了避免用很多消息讀取所有的參數和環境串，MINIX採用了一種完全不同的策略。由execve庫過程在它的內部構造完整的初始堆疊，並把它的基底位址和長度傳給記憶體管理器。因為在使用者空間構造新的堆疊時，對參數和串的引用都是都是局部記憶體訪問，不存在對其他位址空間的訪問，所以效率非常高。

圖4-39 (a)傳給execve的陣列。(b)execve構造的棧。(c)經記憶體管理器重新定位後的棧。(d)在main開始運行時的棧。

為了更清楚地解釋這個機制，讓我們來看一個例子。當用戶向shell鍵入了

ls -l f.c g.c

以後，shell將解釋它並隨後調用庫過程

execve("/bin/ls", argv, argp);

兩個指標陣列的內容如圖4-39(a)所示。位於shell位址空間中的過程execve現在將構造初始堆疊，這如圖4-39(b)所示。這個堆疊在處理EXEC調用的過程中最終將原封不動地複製到記憶體管理器中。

在堆疊最後被拷貝到使用者進程中時，它將不是被放在虛位址0，而是放在分配到的記憶體空間頂端，而該存儲空間的大小是由可執行檔頭中的總長度域決定的。例如，讓我們隨意假設總長度是8192位元組，因此程式可以訪問的最後一個位元組位於位址8191。記憶體管理器負責在把堆疊放在新的位址時重新定位堆疊中的各個指標。這時的堆疊如圖4-39(c)所示。

在EXEC調用結束，程式開始運行時的堆疊與圖4-39(c)所示的完全相同，初始棧指針是8136。然而還有一個問題需要處理，被執行檔的主程序可能是這樣說明的：

main(argc, argv, envp);

對C編譯器來說，main只是一個普通的函數，它並不知道main是特別的，於是認為三個參數將按C的調用習慣（即最後一個參數在最前面）被傳遞進來，並按此編譯訪問這三個參數的代碼。一個整數和兩個指標，這三個參數應該佔用返回地址前的三個字，圖4-39(c)的堆疊看起來當然不是這個樣子。

解決的方法是讓程式不從main開始，而是把稱為C運行起始過程的一小段彙編常式，crtso，總是被連接到正文位址0，由它首先得到控制。它的任務是把三個字推入堆疊然後用標準的調用指令調用main，這使得在main開始運行時的堆疊如圖4-39(d)所示，因此main被欺騙了，認為它是以通常的方式被調用的（實際上這並不是真正的欺騙，它就是那樣被調用的）。

圖4-40 C運行時初啟過程的關鍵部分。

如果程式師在main結束時忽略了調用exit，在main結束後控制會回到運行起始過程。跟前面一樣，編譯器只是把main看作一個普通的函數，在最後一個語句後生成通常的從函數中返回的指令。所以，main將返回到它的調用者—C運行起始過程。它隨後調用exit退出。32位crtso的大部分代碼如圖4-40所示。需要指出的一點是省略的代碼只是那些裝入被壓棧的寄存器和設置指示浮點處理器存在或不存在的標誌的指令。

4.7.6 BRK系統調用

庫過程brk和sbrk用來調整資料段的上限。前者的參數是絕對長度（以位元組為單位），並以此調用BRK；後者的參數是相對于當前長度的正的或負的增量，並由此計算出新的資料段長度，然後調用BRK。不存在真的SBRK系統調用。

一個有趣的問題是：“sbrk是怎樣跟蹤當前的長度以便求出新長度的呢？”答案是有一個變數，brksize，總是保存著當前長度，所以sbrk可以得到這個值。這個變數被初始化為一個編譯器生成的給出正文加數據（非獨立的I和D）或僅僅資料（獨立的I和D）的初始長度的符號。實際上這個名字和這樣的符號是否存在都是依賴於編譯器的，所以在原始檔案目錄的任何標頭檔中都找不到它的定義，它被定義在庫中，並包含在在檔brksize.s中。它的具體位置依賴於系統，但將在和crtso.s相同的目錄中。

執行BRK對記憶體管理器來說是非常簡單的，需要做的全部工作只是檢查各個部分是否仍能放進位址空間、調整表格、並通知內核。

4.7.7 信號處理

在第一章中信號被描述成一種把資訊傳遞給一個未必在等待輸入的進程的機制。在系統中有一個預定義的信號集合，每個信號都有一個缺省動作—或者殺死接受信號的進程、或者忽略這個信號。如果只有這兩種可能，那麼信號處理將是很容易理解和實現的。然而，進程可以通過系統調用改變這些回應。進程可以要求忽略任一信號（除了特別的SIGKILL信號），而且進程還可以準備捕獲一個信號（再一次除了SIGKILL信號），其方法是要求接收到該信號時啟動進程內部的一個信號處理過程而不是執行缺省動作。因此對於程式師來說作業系統處理信號似乎有兩個不同的時間：一個準備階段，這時進程可以修改它對未來信號的回應；和一個回應階段，這時信號發生了並且相應的動作被執行。動作可能是執行使用者編寫的信號處理常式，實際上還有第三個階段。在用戶編寫的處理常式結束時，一個特殊的系統調用清理並恢復接受信號進程的正常操作。程式師實際上不需要知道這第三個階段，他編寫處理常式就象寫其他函數一樣，由作業系統負責處理激發與結束處理常式和管理堆疊。

在準備階段有幾個進程可以在任何時間執行以修改它對信號的回應的系統調用。其中最通用的是SIGACTION，用它可以說明進程將忽略某些信號、捕獲某些信號（用執行位於進程內部、使用者定義的信號處理常式替換缺省處理）、或者恢復某些信號的缺省處理。另一個系統調用SIGPROCMASK可以阻塞一個信號，它使得一個信號被暫時保存起來，只有當進程在後來某個時候解除對這個信號的阻塞時才會回應這個信號。可以在任何時候使用這些調用，甚至可以在信號處理過程內部。在MINIX中，因為需要的資料結構都在記憶體管理器中，所以信號處理的準備階段完全由記憶體管理器處理。每個進程都有幾個sigset\_t變數，每個可能的信號由他們中的一個位表示。一個這樣的變數定義要忽略的信號集，另一個定義要捕獲的信號集，依此類推。每個進程還都有一個sigaction結構的陣列，每個信號對應一個陣列元素，每個sigaction結構元素都包含兩個變數：一個變數保存相應信號指定處理過程的位址，另一個變數sigset\_t用於指定在該處理過程執行時應阻塞的信號，保存處理過程位址的域也可以保存指出信號將被忽略或將以缺省的方式處理的特殊值。

在信號發生時，MNIX的多個部分都將被涉及。回應從記憶體管理器開始，它使用剛剛說明的資料結構，指出哪個進程應該得到這個信號。如果信號應該被捕獲，就必須把它傳遞給目的進程，這需要保存有關進程狀態的資訊以使進程可以恢復正常運行。這些資訊被保存在接收信號進程的堆疊上，因此必須檢查是否有足夠的堆疊空間。因為這是記憶體管理器的領地，所以這個檢查由它執行，隨後記憶體管理器調用在內核中的一個系統任務把這些資訊放到堆疊上。系統任務還處理進程的程式計數器，使進程能夠執行信號處理過程的代碼。在信號處理過程結束時將執行一個SIGRETURN系統調用，通過這個調用記憶體管理器和內核共同恢復進程的信號上下文和寄存器，使進程可以恢復正常運行。如果信號不需要捕獲，缺省的動作將被執行，這可能涉及調用檔案系統生成一個core檔（把進程的映射寫到一個檔裡，它可能會被調試器檢查），或者殺死進程，這涉及到記憶體管理器、檔案系統、內核。最後，因為一個信號可能需要傳送給一組進程，所以記憶體管理器可能使這些動作重複多次。

圖4-41 POSIX和MINIX定義的信號。(\*)標記的信號依賴於硬體的支援。(M)標記的信號POSIX沒有定義，而MINIX為了與老程式相容定義了。一些過時的名字和同義詞沒有給出。

MINIX知道的信號定義在/usr/include/signal.h中，這是POSIX標準所要求的一個檔。他們列在圖4-41中。所有POSIX需要的信號在MINIX中都定義了，但目前並不是所有的信號都得到了支援。例如，POSIX需要一些與作業控制，即把一個正在運行的程式放到後臺和再放回來的能力，有關的信號，MINIX不支援作業控制，但是可能產生這些信號的程式可以移植到MINIX上。這些信號如果產生將被忽略。MINIX還定義一些非POSIX信號和一些為了與舊的源碼相容的POSIX名的同義詞。

信號可以通過兩種方式產生：由KILL系統調用和由內核產生。由MINIX內核產生的信號總是包括SIGINT、SIGQUIT、SIFALRM，其他的內核信號依賴於硬體支援。例如，在8086/8088處理器上不支援檢測非法指令操作碼，但286及以上的處理器有這個能力，在他們上面執行非法指令時會發生陷入。這種服務是硬體提供的，作業系統的實現者必須提供產生一個信號的代碼以作為對這種陷入的響應，我們在第二章看到在/kernel/exception.c中包含了針對幾個不同條件完成上述操作的代碼。因此當MINIX運行在286或更高的處理器上時，作為對非法指令的回應它將產生SIGILL信號，但當MINIX運行在8088處理器上時則永遠不會產生這個信號。

僅僅是硬體在某種條件下能夠陷入並不意味著作業系統實現者能夠充分地利用這個能力。例如，在286以上的Intel處理器上有好幾種違反記憶體完整性的操作都會導致異常，位於/kernel/exception.c中的代碼把這些異常翻譯成SIGSEGV信號。違反硬體定義的堆疊段的界限和違反其他段的界限會產生不同的異常，因為他們可能需要進行不同的處理。然而MINIX使用記憶體的方式使得硬體對可能發生的所有這些錯誤都無法檢測。硬體為每個段都定義了一個基址和一個長度。硬體定義的資料段基址和MINIX的資料段基址相同，但是硬體定義的資料段界限要比MINIX用軟體強制的界限高。換句話說，硬體定義的資料段是在不存在堆疊的情況下，MINIX可能用於資料的最大數量的記憶體。同樣地，硬體定義的堆疊段是在不存在資料區的情況下，MINIX堆疊能用的最大數量的記憶體。儘管硬體能夠檢測到某些違法的訪問，但是因為對硬體和描述符來說資料區和堆疊區域是重疊的，所以它無法檢測堆疊增長進入了資料區域這種最可能發生的堆疊訪問違法。

在內核中可以加入一些代碼，在每個進程每次得到機會運行後由這些代碼檢查進程的寄存器，如果檢測到違反MINIX定義的資料段或堆疊段完整性的情況，就發出一個SIGSEGV信號。但是這樣作是否值得還不清楚，硬體陷入能立刻俘獲一個違法訪問，而軟體檢查可能要到執行了幾千條指令以後才有機會進行，到了這個時候信號處理過程已經幾乎不能為從錯誤中恢復做什麼了。

不論信號來自什麼地方，記憶體管理器都以同樣的方式處理。對每個信號欲傳向的進程，首先將進行一系列的檢查以確定這個信號是否可行。只有當信號發送者是超級使用者，或者它的真實或有效使用者號等於接受信號進程的真實或有效使用者號時，它才能發信號給接受信號的進程。除此之外還有幾個條件能阻止一個信號被發送，比如僵死的進程不能接收信號。如果一個進程已經調用SIGACTION忽略了某個信號或調用SIGPROCMASK阻塞了某個信號，那麼就不能向它遞交信號。阻塞一個信號與忽略不同，接收到的阻塞的信號將被記錄下來，當接收進程取消對這個信號的阻塞時就將該信號遞交給它。最後，如果接收信號的進程堆疊空間不夠它將被殺死。

如果所有的條件都滿足，信號將被發送。如果進程沒有設置捕獲這個信號，那麼就不需要向進程傳遞任何資訊。在這種情況下記憶體管理器將執行這個信號的缺省動作，通常是殺死進程，有時需要產生一個core檔，少數信號的缺省動作是忽略這個信號。在圖4-41中標記為“不支援”的信號是POSIX要求定義的，他們在MINIX中將被忽略。

捕獲一個信號意味著執行進程自己的信號處理代碼，它的位址保存在進程表的sigaction結構中。在第二章中我們看到了一個進程表項中的棧框是怎樣在該進程被中斷時接收重啟它所需的資訊的。通過修改將接收信號進程的棧框，可以使得在該進程被允許執行時，運行信號處理常式。通過修改進程在使用者空間的堆疊，可以使得在信號處理常式結束時執行一個SIGRETURN系統調用。這個系統調用永遠不會從使用者代碼中發出，它是通過內核把它的位址放到堆疊上，使得在信號處理常式結束時成為從堆疊中彈出的返回位址而被執行的。SIGRETURN恢復接受信號進程的棧框，使它能夠從被信號中斷的地方恢復運行。

雖然發送信號的最後一個階段是由系統進程完成的，但是由於資料是從記憶體管理器傳到內核的，所以在這裡總結一下執行的過程是很恰當的。在信號處理過程結束時進程應該象什麼都沒有發生一樣繼續執行，因此捕獲一個信號需要某種與進程上下文切換很類似的東西，而上下文切換是在一個進程停止運行、另一個進程開始執行時發生的。然而在進程表中只有一個地方，其中可以存放將進程恢復到原來狀態所需的所有CPU寄存器的內容。解決這個問題的方法如圖4-42所示。圖的(a)部分是一個進程在中斷發生後剛剛停止運行時的簡化的進程堆疊和部分進程表項的視圖。在進程掛起時，CPU所有的寄存器都被拷貝到進程表內核部分的進程表項的棧框結構中，因為信號是由不同於接收者的進程或任務產生的，所以這就是信號產生時的情形。

圖 4-42 一個進程在處理信號不同階段的棧（上面），和它在進程表中的棧框（下面）。(a)進程停止運行時的狀態。(b)信號處理過程開始運行時的狀態。(c)執行SIGRETURN時的狀態。(d)SIGRETURN執行完之後的狀態。

在準備處理信號的過程中，進程表中的棧框被作為一個sigcontext結構複製到進程自己的堆疊中，從而將它保存起來。隨後一個sigframe結構被放到堆疊上，這個結構包含了在信號處理過程結束後SIGRETURN將使用的資訊。它還包含了調用SIGRETURN的庫過程的位址 ret addr1， 和被中斷的進程將來恢復運行的位址 ret addr2。然而，後一個位址在正常運行時是不使用的。

雖然信號處理過程是程式師寫的一個普通過程，它卻不是通過call指令被調用的。位於進程表棧框中的指令指標（程式計數器）被改變，使得當restart把接收信號的進程投入運行時，將執行信號處理過程。圖4-42(b)所示是這個準備工作，已經完成信號處理過程開始運行時的情況。記住，信號處理過程是一個普通的過程，在它結束時 ret addr1 將被彈出，SIGRETURN將被執行。

(c)部分所示是SIGRETURN正在執行時的情況。sigframe的其他部分現在是SIGRETURN區域變數。SIGRETURN的部分工作是調整它自己的堆疊指標使得在它象一個普通函數那樣結束時，它將用 ret addr2作為它的返回位址。然而，SIGRETURN實際上並不是這樣結束的。它結束得和其他系統調用一樣，允許內核中的調度程式決定重啟哪個進程。最後，接收信號的進程將被重新調度到，並從存放在進程原來棧框中的位址重新啟動。把這個位址放在堆疊上的原因是使用者可能想要用調試器跟蹤程式，當一個信號處理過程被跟蹤時這樣做能欺騙調試器並給它一個關於堆疊的合理的解釋。在每個階段，堆疊看起來都象一個普通進程的堆疊，區域變數在返回位址之上。

SIGRETURN的實際工作是把各個部分恢復成他們接收信號以前的狀態，並進行清理。最重要的是通過使用保存在接收信號進程堆疊中的拷貝，進程表中的棧框被恢復到它原來的狀態。SIGRETURN結束時的情況如圖4-22(d)所示，從中可以看出等待恢復執行的進程的狀態與它剛被中斷時的狀態相同。

大部分信號缺省動作是殺死接收信號的進程。記憶體管理器負責處理滿足下列條件的各種信號：對他們的缺省動作不是忽略、並且接收進程沒有處理、阻塞、或忽略他們。如果父進程正在等待，被殺死的進程將被清理並從進程表中刪除；如果父進程沒有等待，它將進入僵死狀態。對特定的信號（如SIGQUIT），記憶體管理器還要在進程的目前的目錄下寫一個core檔。

一個信號被發往一個因用READ讀一個目前沒有輸入的終端而被阻塞的進程，這樣情形是很常見的。如果進程沒有說明這個信號將被捕獲，系統將簡單地按照通常的方法殺死該進程。但是如果信號被捕獲了，這裡就出現了一個問題：在信號中斷被處理完之後應該怎麼辦？進程應該繼續等待還是執行下一個語句？

MINIX是這樣處理的：系統調用將被終止並返回錯誤代碼EINTR，讓進程可以知道調用是被一個信號中斷的。確定一個進程是否被阻塞並不是一件很簡單的事，記憶體管理器必須請求檔案系統進行檢查。

POSIX建議可使用這種處理方法，但並不要求，它也允許READ在接到一個信號時返回它在接到信號時已經讀到的位元組數。返回EINTR使設置一個時鐘報警並捕獲SIGALRM成為可能，這是一種簡單的實現超時的方法，例如，如果用戶在指定的時間內沒有反應就中止login，並掛斷數據機。使用同步時鐘進程可以用較小的開銷做同樣的事情，但它是MINIX的發明，沒有使用信號移植性好而且只有伺服器進程可以使用，使用者進程不能使用。

4.7.8 其他系統調用

記憶體管理器還處理一些更簡單的系統調用。庫函數getuid和geteuid都調用GETUID系統調用，在它的返回資訊中包括兩個值。同樣地，GETGID系統調用返回真實和有效值，以供getgid和getegid庫函數使用。GETPID也以同樣的方式工作，它返回進程號和父進程號。SETUID和SETGID可以在一個調用中同時設置真實和有效值。在這一組調用中還有兩個調用，GETPGRP和SETSID，前者返回進程組號，後者把它設置為當前的進程號。

PTRACE和REBOOT系統調用也由記憶體管理器處理，前者支援程式的調試，後者影響到系統的很多方面。把它放在記憶體管理器中是合適的，因為它的第一個動作就是殺死除了init以外的所有進程，然後調用檔案系統和系統任務完成它的工作。

4.8 MINIX中記憶體管理的實現

在一般地流覽了記憶體管理器的工作情況以後，讓我們看看代碼本身。記憶體管理器完全用C寫成，代碼是直觀的並且包含有很多注釋，所以大部分不需要長的或深入的說明。我們將首先簡單地看一看標頭檔，然後是主程序，最後是上面討論過的各個系統調用組的檔。

4.8.1 標頭檔和資料結構

記憶體管理器來源程式目錄中的幾個頭檔和內核目錄中的檔有相同的名字，這些名字還將在檔案系統中再一次見到。這些檔在他們自己的上下文中有相似的函數。設計並行結構的目的是為了易於理解整個MINIX系統的組織。記憶體管理器還有一些具有獨特名字的標頭檔。與系統的其他部分一樣，在記憶體管理器版本的table.c被編譯時，全域變數的存儲空間將被保留。這一節我們將看一看所有的標頭檔和table.c。

與MINIX其他主要部分一樣，記憶體管理器有一個主標頭檔mm.h（15800行）。每個檔編譯時都包含它，它自己則包含了每個目的模組都需要的位於/usr/include和它的子目錄中的所有系統範圍的標頭檔。大部分包含在/kernel/kernel.h中的檔也被包含到了這裡。記憶體管理器還需要include/fcntl.h和include/unistd.h中的定義。記憶體管理器自己版本的const.h、type.h、glo.h也包含在裡面。

const.h（15900行）定義了記憶體管理器用的常量，其中有些是特別為16位機器設置的。其中還包含下列符號定義語句：

#define printf printk

這使對printf的調用被編譯成對函數printk的調用。這個函數與我們在內核中看到的那個類似，定義也是為了同樣的原因，使記憶體管理器能在不請求檔案系統説明的情況下顯示錯誤和調試資訊。

type.h目前沒有使用，它只是以一個框架的形式存在，使記憶體管理器和MINIX其他部分有同樣的組織。proto.h（16100行）收集了在整個記憶體管理器內需要的函數原形。

記憶體管理器的全域變數在glo.h中（16200行）說明。內核中用到的關於EXTERN的技巧也用到了這裡，即除了在文件table.c中，EXTERN是一個擴展為extern的普通的宏。而在文件table.c中，它變成一個空串，使得用EXTERN說明的變數被分配空間。

這些變數中的第一個，mp，是指向一個mproc結構的指標，它是某進程表的MM部分，而該進程的系統調用正在被處理。第二個變數，dont\_reply，在每一個新的請求到來時被初始化為FALSE，但在調用的執行過程中如果發現沒有應答資訊應該發送，它就被設置為TRUE。例如成功的EXEC就沒有應答發送。第三個變數，proc\_in\_use，跟蹤當前使用了多少個進程表項，使確定FORK是否可行非常簡單。

訊息緩衝區mm\_in和mm\_out是分別用於請求和應答消息的。who是當前進程的索引，它與mp的關係是

mp = &mproc[who];

當一條消息到達的時候，系統調用號被抽取出來放到mm\_call中。

三個變數err\_code、result2、和res\_ptr用來保存在應答消息中返回給調用者的值。其中最重要的是變數err\_code，在調用沒有錯誤而結束時，它將被設置為OK。最後兩個變數在發生問題時使用。在進程非正常結束時，MINIX把進程的映射寫到一個core檔中，core\_name定義了這個檔將取的名字，core\_sset是一個定義哪些信號應該產生core轉儲檔的點陣圖。

進程表的記憶體管理器部分在另一個檔mproc.h中（16300行）。大部分的域都由注釋做了恰當的描述。幾個域和信號處理有關，mp\_ignore、mp\_catch、mp\_sigmask、mp\_sigmask2、和mp\_sigpending是點陣圖，每個位元代表一個可以送往進程的信號。類型sigset\_t是32位元整數，因此MINIX可以容易地支援32個信號，但當前只定義了16個，信號1是最低（最右）位。在任何情況下，POSIX都要求提供標準的函數，以添加或刪除這些點陣圖表示的信號集成員，因此所有必須的操作都可以在程式師不知道細節的情況下完成。陣列mp\_sigact對處理信號是非常重要的。對每個信號類型都有一個陣列元素，每個元素是一個sigaction結構（定義在 include/signal.h 中），每個sigaction結構由三個域組成：

1. sa\_handler域定義信號是按照缺省的方式處理、忽略、還是由專門的處理過程處理。

2. sa\_mask域的類型是sigset\_t，它定義了在這個信號正在被使用者的處理過程處理時，哪些信號將被阻塞。

3. sa\_flags是一些信號處理用的標誌。

這個陣列使得對信號的處理能夠有很大的靈活性。

如檔案結尾部所示，mp\_flags域用來保存一些位。這個域是一個不帶正負號的整數，在低端CPU上它是16位，在386和以上的CPU上則是32位。因為只用了9位，所以它有足夠的擴充空間，即使在8088上也是如此。

進程表中的最後一個域是mp\_procargs。在一個新進程啟動時，系統將建立一個類似圖4-39的堆疊，指向新進程argv陣列起始位址的指標被存放在這裡，它被ps命令使用。對於圖4-39的例子，值8164將被保存在這裡，如果在ls活動時執行ps命令，那麼它將顯示出命令列：

ls -l f.c g.c

下一個檔是param.h（16400行），它包含了許多用於請求資訊中的系統調用參數的宏和四個用於應答消息中域的巨集。當下麵這一行：

k=pid;

出現在任何包含了param.h的檔中時，在把它送往編譯器之前前置處理器將其轉換成：

k=mm\_in.m1\_i1;

在介紹執行代碼之前讓我們看一看table.c（16500行），它的編譯將為我們在glo.h和mproc.h中看到的各種EXTERN變數和結構保留空間。語句：

#define \_TABLE

使EXTERN成為空串，這與我們在內核代碼中看到的方法一樣。

table.c另一個重要的成份是陣列call\_vec（16515行）。當一個請求消息到達時，其中的系統調用號將被取出作為call\_vec的索引，以找到處理這個系統調用的過程。不是合法調用的系統調用號都會引起執行no\_sys，它只是返回一個錯誤代碼。值得注意的是雖然\_PROTOTYPE宏用在了call\_vec的定義中，這個定義並不是一個原形定義，而是一個初始化的陣列的定義。但是因為它是一個函數的陣列，使用\_PROTOTYPE巨集是使程式與經典C（Kernighan & Ritchie）和標準C都相容的最簡單的辦法。

4.8.2 主程序

記憶體管理器是獨立於內核和檔案系統編譯和連接的，所以它有自己的主程序。它的主程序在內核初始化自己之後被啟動。主程序在main.c中（16627行），在通過調用mm\_init完成自己的初始化之後，記憶體管理器進入16636行的迴圈。在這個迴圈中，它調用get\_work等待到來的請求消息，然後通過call\_vec調用某個do\_XXX過程執行請求，最後如果需要的話發送應答。這種結構現在應該很熟悉了，它和I/O任務所用的一樣。

過程get\_work（16663行）和reply（16676行）分別處理實際的接收和發送。

這個檔的最後一個過程是mm\_init，它初始化記憶體管理器。在系統開始運行之後它將不再被使用。16730行對sys\_getmap的調用得到核心的記憶體使用資訊，16734到16741的迴圈為任務和伺服器初始化所有的進程表項，隨後的幾行準備init的進程表項。在16749行記憶體管理器等待檔案系統給它發消息，正如在對MINIX中鎖死處理的討論中提到的，這是唯一的一次檔案系統向記憶體管理器發送請求消息。該消息說明多少記憶體被用於RAM磁片。16755行對mem\_init的調用通過調用系統任務初始化空洞表。在這以後，正常的記憶體管理就可以開始了。這個調用還填充total\_clicks和free\_clicks變數。此後，列印、顯示記憶體總量、核心記憶體的使用、RAM磁片的大小和空閒記憶體情況。在列印完資訊之後，向檔案系統發送一個應答（16764行），使它能夠繼續。最後，為了方便ps命令，進程表的記憶體管理器部分的位址被送給記憶體任務。

4.8.3 FORK、EXIT、和WAIT的實現

FORK、EXIT、和WAIT系統調用是由檔forkexit.c中的do\_fork、do\_exit、do\_wait過程實現的。過程do\_fork（16832行）根據圖4-37所示的步驟執行。要注意的是對procs\_in\_use的第二次調用（16847行）為超級使用者保留最後幾個進程表項。在計算子進程需要多大的記憶體時，資料段和堆疊之間的空隙也被包括在內，但不包括正文段。這時或者父進程的正文被共用，或者進程有結合的I和D 空間，它的正文段長度為零。計算完成之後，調用alloc\_mem以得到記憶體。如果成功了，子進程和父進程的基底位址將被從塊轉換為絕對位元組，然後調用sys\_copy發送消息到系統任務由其完成拷貝工作。

現在在進程表中已經找到了一個空位，早先涉及proc\_in\_use的測試保證了肯定存在一個空位。然後在其中填入各種內容，首先把父進程的表項複製到這裡，然後修改mm\_parent、mm\_flags、mm\_seg、mm\_exitstatus、和mm\_sigstatus。這些域中有些需要特殊處理，因為子進程不繼承跟蹤狀態，所以mm\_flags中的TRACED位被置零。mm\_seg域是一個包含用於正文、資料、和堆疊段的元素的陣列，如果是具有獨立的I和D空間可以共用正文的程式，那麼它的正文部分將指向父進程的正文段。

下一步是為子進程指定進程號。變數next\_pid跟蹤將被指定的下一個進程號。這時在理論上可能會發生下面所述的問題：在把一個進程號，比如20，賦給一個非常長壽的進程後，可能會有30000多個進程被創建和撤銷，next\_pid可能又再次回到20。指定一個仍然在使用的進程號將是一場災難（設想隨後某個進程向進程20發信號的情形），所以我們要搜索整個進程表以確定將被指定的進程號沒有被使用。

對sys\_fork和tell\_fs的調用分別通知內核和檔案系統新進程已經被創建，使他們能夠更新他們的進程表。（所有以sys\_開頭的過程都是向內核中的系統進程發送一條消息請求圖3-50中一個服務的庫常式。）進程的創建和撤銷都是從記憶體管理器開始，並在結束時傳播到內核和檔案系統。

對子進程的應答資訊是在do\_fork的結尾處發送的。對父進程的應答包含有子進程的進程號，它象對普通請求的應答一樣是由main中的迴圈發出的。

下一個由記憶體管理器處理的系統調用是EXIT。過程do\_mm\_exit（16912行）接收這個調用，但大部分工作是調用mm\_exit（16927行）完成的。這樣劃分工作是因為mm\_exit()也被用來處理被信號終止運行的進程。兩者工作相同，但參數不同，所以這樣劃分是很方便的。

mm\_exit做的第一件事情是，如果進程有一個計時器在運行就停止它；其次，通知內核和檔案系統這個進程不再可以運行（16949和16950行）。對庫常式sys\_xit的調用發一條消息給系統任務告訴它標誌該進程不可運行，使它不會再被調度到。然後是釋放記憶體。對find\_share的調用確定正文段是否正在與另一個進程共用，如果沒有，就調用free\_mem釋放它。緊接著調用同一個過程釋放資料和堆疊。確定能否在一次free\_mem調用中釋放所有的記憶體是不值得的。如果父進程在等待，mm\_exit就調用cleanup釋放進程表項。如果父進程沒有等待，就讓進程進入僵死狀態。這由mm\_flags中的HANGING位指示。無論進程是被徹底消滅還是進入僵死狀態，mm-exit的最後一個動作都是搜索進程表尋找剛才終止進程的子進程（16975行到16982行），如果找到就把它們變成init的子進程。如果init正在等待並且一個子進程進入了HANGING狀態，它將調用cleanup處理這個子進程。這個方法可以處理圖4-43(a)所示的這類情況。在圖中我們看到進程12要結束了，它的父進程7正在等待它，這時cleanup將被調用以除掉12，因此52和53將成為init的子進程，如圖4-43(b)所示。現在的情況是已經結束的進程53成為一個正在執行WAIT的進程的子進程，所以它也將被清理。

圖4-43 (a)進程12要結束時的情況。(b)進程12結束以後的情況。

當父進程執行WAIT或WAITPID時，控制轉移到16992行的過程do\_waitpid。兩個調用的參數不同，期望的動作也不同，但在17009到17011行通過設置內部變數使do\_waitpid能夠完成兩個調用中任何一個的動作。從17019到17041行的迴圈掃描整個進程表，檢查這個進程是否有子進程，如果有則檢查他們是否處於現在可以被清理的僵死狀態，如果找到了一個僵死狀態的（17026行），do\_waitpid將清理掉它並返回。設置dont\_reply是因為對父進程的應答是從cleanup內部而不是main中的迴圈中發出的。如果找到了一個被跟蹤的子進程，do\_waitpid將送回一個指示這個進程已經停止的應答並返回，這裡也同樣把dont\_reply位設置為真，以阻止main發送第二個應答。

如果執行WAIT的進程沒有子進程，它將簡單地得到一個錯誤返回（17053行）；如果它有子進程，但是其中沒有一個處於僵死或被跟蹤的狀態，系統將檢查在do\_waitpid被調用時是否設置了一個標誌父進程不想等待的位元。如果沒有設置（通常的情況），17047行將設置一個位指示它在等待，父進程將被掛起直到一個子進程結束。

當一個進程已經結束並且它的父進程在等待它時，不管這些事件發生的次序如何，過程cleanup都將被調用執行最後的操作。這時要做的工作不多，把父進程從WAIT或WAITPID中喚醒並給它終止子進程的進程號，和退出及信號狀態。這時檔案系統已經釋放了子進程的記憶體，內核已經停止了對它的調度，因此現在內核要做的全部工作就是釋放子進程在進程表中的表項。

4.8.4 EXEC的實現

EXEC代碼按圖4-38中的工作步驟編寫，它被包含在過程do\_exec（17140行）中。在幾個簡單的合法性檢查之後，記憶體管理器從使用者空間取到要執行的檔的名字。在17172行它向檔案系統發一個特殊的消息以切換到使用者的目錄，使剛剛取到的路徑被解釋為相對於使用者而不是記憶體管理器的工作目錄的路徑。

如果該檔存在並且可執行，記憶體管理器將讀入其檔頭並抽出各段長度。隨後從使用者空間取來堆疊（17188和17189行）、檢查新進程是否能與已經在運行的進程共用正文（17196行）、為新的記憶體映射分配記憶體（17199行）、修改指針[參見圖4-39(b)和(c)的區別]、和讀入正文段（如果需要）和資料段（17221到17226行）。最後，處理setuid和setgid位元、更新進程表項、告訴內核它已經結束工作，因此這個進程可以再被調度。

儘管對所有步驟的控制都在do\_exec中，但是許多細節都是由exec.c中的許多輔助過程完成的。例如read\_header（17272行）不僅讀取檔頭返回各段長度，它還驗證該檔對指定的CPU類型是有效的MINIX執行檔，這是通過在記憶體管理器被編譯時，條件編譯適當的測試語句實現的。read\_header還驗證所有的段能否放入虛位址空間。

過程new\_mem（17366行）檢查是否有足夠的記憶體用於新的記憶體映射。如果正文是共用的它就只查找足夠容納資料和堆疊的空洞，否則查找足夠容納結合的正文、資料和堆疊段的單個空洞。這裡一個可能的改進是查找兩個分開的空洞，一個用於正文，另一個用於資料和堆疊，原因是這兩個區域不需要鄰接，而在MINIX的早期版本中這是需要的。如果找到了足夠的記憶體，舊的記憶體將被釋放，新的將被分配。如果沒有足夠的記憶體，EXEC將失敗。在新的記憶體分配後，new\_mem將更新記憶體圖（在mp\_seg中），並通過調用庫常式sys\_newmap把它通知內核。

new\_mem餘下部分涉及到把bss段、空隙和堆疊段清零（bss段是資料段中包含所有未初始化的全域變數的那一部分）。許多編譯器會產生將bss段清零的代碼，但這樣做使MINIX與不生成這種代碼的編譯器也能合作。資料段和堆疊段之間的空隙也被清零，因此當資料段被BRK系統調用擴充時，新獲得的記憶體將全部是零。這不僅是為了方便程式師，他可以指望新變數初值為零，在多使用者作業系統上這還是一個安全措施，以前佔用這塊記憶體的進程可能在其中存放了不該被其他進程看到的資料。

下一個過程是patch\_ptr（17465行），它把圖4-39(b)指標重新定位成圖4-39(c)的形式。這種工作很簡單：檢查堆疊，找到所有的指標並把基底位址加到每個指針上面。

每次EXEC調用過程load\_seg（17498行）一次到兩次，完成可能需要的裝入正文段和總是需要的裝入資料段的工作。我們在這裡使用了一個技巧，以允許檔案系統把整個段直接裝入用戶空間，而不是一塊一塊地從檔中讀入然後拷貝到使用者空間。實際上，這個調用被檔案系統以一種略微特殊的方式解碼，使它看起來就像是使用者進程自己讀取一個整段。只有檔案系統讀常式的頭幾行知道這裡面作了特殊的處理，這個策略使裝入略微得到加速。

exec.c的最後一個過程是find\_share（7535行），它把要被執行的檔的i結點、設備、和修改時間與現有進程的進行比較，尋找可以共用正文段的進程，這只是對mproc中合適域進行的一個簡單搜索。當然，必須忽略為了它進行這次搜索的那個進程。

4.8.5 BRK的實現

正如我們剛剛看到的，MINIX使用的記憶體模型是相當簡單的：每個進程在創建時獲得一塊用於資料和堆疊的連續的區塊，它永遠不會在記憶體中被移動、永遠不會被交換出記憶體、永遠不會增長、也永遠不會縮小。唯一能夠發生的是資料段可能會從低端吃掉一些空隙，以及堆疊段可能會從高端吃掉一些空隙。在這些情況下，位於break.c中的BRK調用的實現是特別簡單的，它首先檢查新的大小是否可行，隨後更新表格以反映這些變化。

頂層的過程是do\_brk（17628行），但大部分工作是由adjust（17661行）完成的。後者檢查資料段和堆疊段是否衝突。如果衝突，BRK調用就不能執行，但進程並不是立刻被殺死。在進行測試前，一個安全係數SAFETY\_BYTES被加到資料段的頂上，所以雖然可以作出堆疊過度增長的結論，但同時在堆疊中仍會有足夠供進程繼續運行一小段時間。它取回控制（帶著一個錯誤資訊），這樣它可以列印出合適的錯誤資訊，然後優美地結束。

值得注意的是SAFETY\_BYTES是在過程的中間（17693行）用一個#define語句定義的。這種用法是很不同尋常的，在通常情況下這種定義出現在檔的開頭或獨立的標頭檔中。相關的注釋指出，程式師發現確定安全因數的大小是非常困難的，這樣定義毫無疑問是為了引起注意，也許還是為了鼓勵進一步的實驗。

資料段的基址是一個常數，所以如果adjust必須調整資料段，它只需要更新長度域。堆疊從一個固定的終點向下增長，因此如果adjust注意到作為參數傳遞給它的堆疊指標已經增長超出了堆疊段（到了更低的位址），那麼長度和起點都將被更新。

最後一個過程size\_ok()（17736行）以塊和位元組為單位檢查段的長度是否能被位址空間容納下。為了說明為什麼它被寫成了一個單獨的函數，用於16位元機器的條件碼被保留在清單中，在用於32位的MINIX上把它作為一個單獨的函數是沒有什麼意義的。它只在兩個地方被調用，並且用17765行替換調用將使代碼更加緊湊，因為調用要傳遞幾個在32位實現上根本不用的參數。

4.8.6 信號處理的實現

與信號處理有關的系統調用有八個。他們摘要示於圖4-44中。這些系統調用和信號自己都是在signal.c中處理的。還有一個系統調用REBOOT，因為它用信號終止所有的進程，所以也是在這個檔中處理的。

系統調用

目的

SIGACTION

修改對未來信號的回應

SIGPROCMASK

改變阻塞信號集合

KILL

向另一個進程發信號

ALARM

經過延遲後向自己發ALRM信號

PAUSE

掛起自己直到收到未來的信號

SIGSUSPEND

改變阻塞信號集合，然後PAUSE

SIGPENDING

檢查未處理的（阻塞的）信號

SIGRETURN

在信號處理過程後進行清理

圖 4-44 與信號處理有關的系統調用

SIGACTION調用支援sigaction和signal函數，他們使進程能改變自己對信號的回應方式。sigaction是POSIX要求的，大多數情況下它是首選的調用，而signal庫函數是標準C要求的，要移植到非POSIX系統上的程式應該用它來寫。do\_sigaction的代碼（17845行）開始於檢查信號號碼的合法性，和驗證這個調用沒有試圖改變對SIGKILL信號的回應（17851到17852行）（忽略、俘獲、或阻塞SIGKILL是不允許的，SIGKILL是使用者能夠控制自己的進程和管理員能夠控制使用者的最後手段）。SIGACTION被調用時帶有指向一個sigaction結構的指標sig\_osa，它接收在調用執行以前的有效的舊的信號處理屬性，和另一個這樣的結構sig\_nsa，它包含了新的屬性。

第一步是調用系統任務把當前的屬性拷貝到sig\_osa所指的結構中。把NULL指標放在sig\_osa 中調用SIGACTION，這可以檢查舊的信號處理屬性而不修改他們，在這種情況下do\_sigaction會立刻返回（17860行）。如果sig\_nsa不是NULL，定義新的信號動作的結構將被複製到記憶體管理器的空間中。17867到17877行的代碼根據新的動作是忽略信號、使用缺省處理、或者是捕獲信號修改點陣圖mp\_catch、mp\_ignore、和mp\_sigpending。儘管這些都是可以用簡單的宏實現的直觀的位元操作，這裡還是使用了庫函數sigaddset和sigdelset。這些函數是POSIX標準的要求，目的是使使用他們的程式容易移植，甚至是移植到信號數目大於一個整數可用的比特數的系統上。通過使用庫函數使MINIX自己能更容易地移植到不同結構的系統上。

最後，進程表記憶體管理器部分其他與信號有關的域被填入。每個信號都有一個點陣圖sa\_mask，它定義了當這個信號的處理常式正在執行時那些信號將被阻塞；每個信號也都有一個指標sa\_handler，它包含指向信號處理常式的指標，或指出信號將被忽略或按缺省方式處理的特殊值。在處理常式結束時調用SIGRETURN的庫常式的位址保存在mp\_sigreturn中，這個位址是記憶體管理器收到的消息中的一個域。

POSIX允許進程修改自己的信號處理方式，甚至在信號處理過程中也是這樣。這可以用來改變在處理一個信號的過程中進程對隨後信號的響應，然後再恢復普通的回應。下一組系統調用支援這些信號操作特性。SIGPENDING由do\_sigpending處理，它返回點陣圖mp\_sigpending，使進程能判斷它是否有等待的信號；SIGPROCMASK由do\_sigprocmask處理，它返回當前被阻塞的信號集合，它還可以以來改變集合中單個信號的狀態或把整個集合替換成一個新的。一個信號被解除阻塞的時刻是一個很好的檢查等待信號的時機，這是通過在17927和17933行調用check\_pending完成的。do\_sigsuspend（17949行）執行SIGSUSPEND系統調用，它掛起一個進程直到收到一個信號為止，與我們在這裡討論過的其他函數一樣，它也要操作點陣圖。它還設置mp\_flags中的SIGSUSPENDED位，這就是它為阻止進程運行所作的全部。同時這裡也是調用check\_pending的一個很好的時機。最後，do\_return處理SIGRETURN，它用來從一個用戶指定的處理過程中返回，它恢復在進入處理過程時就存在的信號上下文，然後在17980行再次調用check\_pending。

有些信號是源自內核的，如SIGINT，這些信號的處理方式與使用者進程調用kill產生的信號的處理方式類似。do\_kill（17983行）和do\_sig（17994行）這兩個過程在概念上是類似的。他們都使記憶體管理器發出一個信號。對KILL的單個調用可能需要向一組進程發送信號，do\_kill只是調用check\_sig，由它在整個進程表中查找夠資格的接收者。當內核的消息到達時將調用do\_ksig。在消息中含有一個點陣圖，使得內核可以在一條消息中產生多個信號。與KILL相同，這些信號中的每一個都可能需要傳給一組進程。這個點陣圖在18026到18048行的迴圈中被一次一位地處理。一些內核信號需要特別注意：在有些情況下進程ID被修改以便使信號傳遞給一組進程（18030行到18033行），如果沒有請求過，SIGALRM將被忽略。除了這個例外，每個位集都將導致調用check\_sig，這與do\_kill中相同。

ALARM系統調用由do\_alarm（18056行）控制。它調用下一個函數set\_alarm發消息給時鐘任務，讓它啟動計時器。把set\_alarm（18067行）作為一個獨立的函數是因為在進程結束而計時器仍然打開著時，還要用它關閉計時器。當計時器時間到時，內核通過向記憶體管理器發一個類型為KSIG的消息來宣佈這個事件，正如上面所討論的，這將使do\_ksig運行。如果沒有被捕獲，SIGALRM的缺省動作是殺死進程。要捕獲SIGALRM必須用SIGACTION安裝一個處理過程。具有使用者指定處理過程的SIGALRM信號完整的事件序列如圖4-45所示，這裡有三條消息序列。在消息(1)、(2)、(3)中，使用者通過向記憶體管理器發消息執行一個ALARM調用，管理器向時鐘發請求，時鐘應答；在消息(4)、(5)、(6)中，時鐘任務向記憶體管理器發送告警，記憶體管理器調用系統任務為執行信號處理過程準備使用者進程的堆疊（與圖4-42(b)中一樣），系統任務應答；消息(7)調用SIGRETURN，它在信號處理過程執行結束時發生。作為回答，記憶體管理器向系統任務發送消息(8)讓它完成清理工作，然後系統任務用消息(9)應答。消息(6)自己並不引起信號處理過程執行，但事件的序列將被保持，因為根據MINIX的優先順序調度演算法，系統任務作為一個任務將被允許完成它的工作，信號處理過程作為使用者進程的一部分只有在系統任務完成它的工作後才會執行。

圖4-45 與計時器有關的消息。其中最重要的有：(1)用戶執行ALARM。(4)時間到信號被發出。(7)處理過程結束，調用SIGRETURN。詳見文中描述。

do\_pause處理PAUSE系統調用（18115行）。這裡需要作的只是設置一個位並且不作應答，從而使調用者被阻塞，甚至都不需要通知內核，因為它知道調用者被阻塞了。

signal.c中處理的最後一個系統調用是REBOOT（18128行）。這個調用只被超級使用者執行的專門程式使用，但它提供了一個重要的功能，它保證所有進程有序地結束，以及在調用內核中的系統任務關機之前，檔案系統被同步。進程的終止是通過check\_sig向除init之外的所有進程發SIGKILL完成的，這也是REBOOT被包含在這個檔中的原因。

前面我們提到過signal.c中的幾個支援函數，現在我們將詳細地討論他們。到目前為止最重要的是sig\_proc（18168行），它實際地發送一個信號。首先作一些檢查，向死的（18190到18192行）或掛起的（18194到18196行）進程發送信號都是嚴重問題，會產生一個系統panic。正在被跟蹤的進程在接到信號時將被停止（18198到18202行）。如果信號將被忽略，sig\_proc的工作在18204行結束。這是一些信號的缺省動作，例如POSIX要求但MINIX不支援的一些信號。如果信號被阻塞，唯一需要作的是在進程的mp\_sigpending點陣圖中設置一個位。關鍵的測試（18213行）是區分已經允許捕獲信號的進程和還沒有允許的。到此為止，所有其他的特殊情況都已經消除，不能俘獲信號的進程將結束。

能被捕獲的信號在18214到18249行處理。首先構造一條消息發往內核，它的一些部分是進程表記憶體管理器部分中資訊的拷貝。如果接收信號的進程先前被用SIGSUSPEND掛起，那麼在掛起時被保存的信號遮罩將被包含在消息中，否則當前進程的遮罩將被包含（18213到18217行）。消息中包含的其他條目是接收信號進程空間中的幾個位址：信號處理過程的位址、在處理過程結束時將被調用的sigreturn庫常式的位址、和當前堆疊指標。

下一步是在進程的堆疊上分配空間。圖4-46所示是將被放在堆疊上的結構。把sigcontext部分放在堆疊上保存是為了以後恢復，因為進程表中對應的結構在準備執行信號處理過程時將被修改。sigframe部分提供了信號處理過程的返回位址，和SIGRETURN在處理過程結束時完成進程狀態恢復所需的資料。返回位址和堆疊指標實際上並沒有被MINIX任何部分使用，他們的作用是在有人用調試器跟蹤信號處理過程時愚弄調試器。

圖4-46 為信號處理過程作準備而推入棧的sigcontext和sigframe結構。處理器的寄存器組是上下文切換時使用的棧框的一個拷貝。

將被放到接收信號進程的堆疊上的結構是相當大的。18225到18226行的代碼為它保留空間，隨後調用adjust檢查進程堆疊上是否有足夠的空間。如果沒有，程式將用很少使用的goto語句（18228行到18229行）跳轉到標號doterminate，然後殺死進程。

對adjust的調用有一個潛在的問題。在對BRK實現的討論中我們提到過，如果堆疊與資料段的間隙長度小於SAFETY\_BYTES，那麼adjust將返回錯誤。為錯誤提供一個餘量是因為合法性檢查只能由軟體間歇地進行，在當前的例子中因為處理信號所需的堆疊空間是確切知道的，所以這個錯誤餘量可能是太多了，多出的空間只有信號處理過程需要，而它一般是一個較簡單的函數。一些進程可能會因為對adjust的調用失敗而毫無必要地終止，這當然要比讓進程在某些時候神秘地失敗強，當然更加精細的調節也許是可能的。

如果堆疊上有足夠的空間，另外兩個標誌將被檢驗。SA\_NODEFER標誌指出在處理信號時是否阻塞後來的相同類型的信號，SA\_RESETHAND標誌指出在接到這個信號時信號處理過程是否被重定。（這忠實地模擬了老式的signal調用，儘管這個“特徵”經常被認為是老式調用的一個缺陷，支援老式的特徵需要同時支援他們的缺陷）。隨後用庫常式sys\_sendsig（18242行）通知內核。最後，指示有一個信號在等待的位元被清除，unpause被調用以結束任何進程可能在上面掛起的系統調用。當接收信號的進程下一次運行時，信號處理過程將會執行。

下面我們看一看由標號doterminate（18250行）標記的結束代碼，這個標號和goto語句是處理可能的adjust調用失敗的最簡單的方法。這裡被處理的信號由於某種原因不能或不應該被捕獲。如果對這個信號適合，處理動作可能包括生成一個core檔，最後總是通過調用mm\_exit（18258行）使進程結束。

check\_sig（18265行）是記憶體管理器檢查是否有需要發送的信號的地方。調用

kill(0, sig);

使指定的信號被發給調用者組的所有進程（即所有從同一個終端啟動的所有進程），源自內核的信號和REBOOT也可能影響多個進程。由於這個原因，check\_sig在18288到18318行的迴圈中掃描整個進程表以找出所有應該接收信號的進程。迴圈中包括了大量的測試。信號只發給通過了所有測試的進程，發送通過在18315行調用sig\_proc完成。

check\_pending（18330行）是另一個在我們討論過的代碼中被多次調用的函數。它迴圈檢查do\_sigmask、do\_sigreturn、do\_sigsuspend引用的進程mp\_sigpending點陣圖中所有的位，查看被阻塞的信號中是否有已經不再阻塞的。它調用sig\_proc發送找到的第一個不再阻塞的等待信號。因為所有信號處理過程最終都將引起do\_sigreturn被執行，這樣作足以保證把所有掛起的信號最終都發送出去。

過程unpause（18359行）處理髮往掛起在PAUSE、WAIT、READ、WRITE或SIGSUSPEND調用上的進程的信號。PAUSE、WAIT和SIGSUSPEND可以通過查詢進程表的記憶體管理器部分檢查出來，如果都沒有發現就必須讓檔案系統用它自己的do\_unpause函數檢查可能的在READ和WRITE上掛起的情況。在各種情況下動作都是相同的：向等待的調用發送一個錯誤回答，重定對應於進程等待原因的標誌位元，使進程能夠恢復執行處理信號。

這個檔中最後一個過程是dump\_core（18402行）。它把核心映射寫到磁片上。核心映射包括一個含有進程各段長度的頭、從內核進程表中獲得的這個進程的所有狀態資訊的一個拷貝、和各個段的記憶體映射。調試器能解釋這些資訊以説明程式師確定在程式運行過程中什麼發生了錯誤。寫文件的代碼是很直觀的，但前面章節中提到過的潛在的問題又出現了，而且更難處理。為了保證記錄到核心映射上的堆疊段是最新的，在18428行調用了adjust，這個調用可能會因為安全餘量的原因失敗。dump\_core不檢查這個調用是否成功，所以無論如何核心映射都將被寫入，但是檔中有關堆疊的資訊可能會不正確。

4.8.7 其他系統調用的實現

文件getset.c包含了一個過程do\_getset（18515行），它執行餘下的七個記憶體管理器調用，如圖4-47所示。他們都是如此簡單以至不值得為每個寫一個單獨的過程。GETUID和GETGID調用同時返回真實和有效用戶號或組號。

系統調用

描述

GETUID

返回真實和有效用戶號

GETGID

返回真實和有效組號

GETPID

返回進程和它父親的進程號

SETUID

設置調用者的真實和有效用戶號

SETGID

設置調用者的真實和有效組號

SETSID

創建新的會話，返回進程號

SETPGRP

返回進程組標識

圖4-47 mm/getset.c中支援的系統調用

設置用戶號和組號要比讀他們稍微難一點，這裡必須檢查調用者是否有許可權設置組號或用戶號。如果調用者通過了檢查，則必須把新的用戶號或組號通知檔案系統，因為檔案保護依賴於他們。SETSID調用創建一個新的會話，已經是進程組組長的進程不允許執行這個操作，18561行的測試完成這個檢查。檔案系統完成把一個進程變成沒有控制終端的進程組組長的操作。

命令

描述

T\_STOP

停止進程

T\_OK

允許這個進程被父進程跟蹤

T\_GETINS

從正文（指令）空間返回值

T\_GETDATA

從資料空間返回值

T\_GETUSER

從使用者進程表返回值

T\_SETINS

在正文（指令）空間設置值

T\_SETDATA

在資料空間設置值

T\_SETUSER

在使用者進程表中設置值

T\_RESUME

恢復執行

T\_EXIT

退出

T\_STEP

設置跟蹤位

圖 4-48 mm/trace.c中支援的調試命令

PTRACE系統調用提供的最小的調試支援包含在檔trace.c中。共有十一個命令可以作為參數傳遞給PTRACE系統調用，如圖4-48所示。在記憶體管理器中，do\_trace處理其中的四個：enable、exit、resume、step。允許和退出跟蹤的請求在這裡完成，其他的命令都被傳遞給系統任務，它能夠操作進程表的內核部分。這是通過在18669行調用庫函數sys\_trace完成的。在trace.c尾部定義了兩個支援函數，stop\_proc用來在被跟蹤的進程收到信號時停止它，findproc通過在進程表中搜索要被跟蹤的進程支援do\_trace。

4.8.8 記憶體管理器工具

餘下的檔中包含了工具常式和表格。文件alloc.c是系統對記憶體的那些部分在被使用那些部分空閒保持跟蹤的地方，它有三個入口：

1. alloc\_mem - 請求一塊給定大小的記憶體。

2. free\_mem - 歸還不再需要的記憶體

3. max\_hole - 計算最大可用空洞的長度

4. mem\_init - 在記憶體管理器開始運行時初始化空閒表。

正如我們前面說過的，alloc\_mem（18840行）在按照記憶體位址排列的空洞表中使用首次適配演算法查找。如果找到的塊太大，它就取出它需要的，把其他的留在空閒表上並從其原長度中減掉取走的。如果整個空洞都需要，它將調用del\_slot（18926行）從空閒表中刪除這個空洞的入口。

free\_mem的工作是檢查一塊新釋放的記憶體能否與任何一方的鄰居合併，如果能就調用merge（18949行）合併空洞並更新空閒表。

max\_hole（18985行）掃描空洞表並返回它找到的最大的條目。mem\_init（19005行）構造由所有可用記憶體組成的初始空閒表。

下一個檔是utility.c，它包含了在記憶體管理器不同部分的幾個雜過程。過程allowed（19120行）檢查一個對檔的操作是否允許。比如do\_exec需要知道檔是否是可執行的。

過程no\_sys（19161行）應該永遠不被調用，提供它只是為了處理用戶用非法的或不是由記憶體管理器處理的系統調用號調用記憶體管理器的情況。

panic（19172行）只有在記憶體管理器檢測到一個它無法恢復的嚴重錯誤時才會被調用。它向系統任務報告錯誤，系統任務緊急停止系統。它不該被輕易調用。

utility.c中最後一個是tell\_fs，當記憶體管理器處理的事件需要通知檔案系統時，它構造一條消息並發給檔案系統。

儘管與前面的有很大不同，檔putk.c中的兩個過程也是工具過程。對printf的調用被一次又一次插入到記憶體管理器中，主要是為了調試，panic也要調用printf。前面已經說過，printf實際上是一個宏，它被定義為printk，所以對printf的調用不使用把消息送到檔案系統的標準I/O庫過程。printk調用putk直接與終端任務通訊，這種操作對普通用戶是禁止的。在內核代碼中我們也見到過一個同名的常式。

4.9 小結

在本章中討論了一般的和MINIX的記憶體管理。我們看到最簡單的系統根本不用交換和分頁，一個程式一旦裝入記憶體就一直保持在那個地方直到結束為止。有些作業系統同一時刻只允許一個進程在記憶體中，另外一些支援多道程序。

再向前一步是交換。在使用了交換以後，系統可以處理比記憶體所能容納的更多的進程，得不到空間的進程將被換出到磁片上。記憶體和磁片上的空閒空間可以用點陣圖或空洞表跟蹤。

更先進的電腦通常都有某種形式的虛擬記憶體。在最簡單的情況下，每個進程的位址空間都被劃分為同樣大小的稱為頁的塊，它可以被放到記憶體中任何可用的頁框中去。有許多頁面替換演算法，最著名的兩個是第二次機會和老化演算法。為了使分頁系統很好地工作，僅僅選擇一個好的演算法是不夠的，還需注意確定工作集、記憶體分配策略、頁面大小等問題。

分段有助於處理在運行中要改變大小的資料結構、簡化連結和共用，它還有助於為不同的段提供不同的保護。有時分段和分頁結合起來構成兩維的虛擬記憶體，MULTICS系統和Intel的Pentium支援分段和分頁。

MINIX的記憶體管理是很簡單的。記憶體在進程執行FORK或EXEC系統調用時被分配。只要進程還存在，這樣分配的記憶體就永遠不會增長或減小。在Intel處理器上MINIX使用的記憶體模型有兩種，小程式可以把指令和資料放到同一個記憶體段中，大程式使用獨立的指令和資料空間（獨立的I和D）。具有獨立I和D空間的進程可以共用他們記憶體中的正文部分，所以在執行FORK時只有資料和堆疊的記憶體是必須分配的。這一點在執行EXEC時也可能成立，其條件是有另外一個進程已經在使用EXEC所需要的正文了。

記憶體管理器的大部分工作與使用一個空洞表和首次適配演算法的跟蹤空閒記憶體的工作沒有關係，卻是執行與記憶體管理有關的系統調用。一些系統調用支援POSIX風格的信號，因為大部分信號的缺省動作是結束接受信號的進程，所以在記憶體管理器中處理他們是比較合適的。所有進程的終止都由它發起。幾個與記憶體沒有直接關係的系統調用也由記憶體管理器處理，因為記憶體管理器比檔案系統小，所以把他們放在這裡是最方便的。

習 題

1. 一個電腦系統有足夠的空間在它的主存中放四個程式，這些程式有一半的時間在空閒等待I/O操作，多大比例的CPU時間被浪費掉了？

2. 在一個使用交換的系統中，按位址排列的記憶體中的空洞大小是：10K，4K，20K，18K，7K，9K，12K，和15K。對於連續的段請求：

(a) 12K

(b) 10K

(c) 9K

使用首次適配演算法那個空洞將被取出？對最佳適配、最差適配、下次適配回答同樣的問題。

3. 物理位址和虛位址之間的區別是什麼？

4. 使用圖4-8的頁表，指出對應於下列虛位址的物理位址：

(a) 20

(b) 4100

(c) 8300

5. Intel 8086處理器不支援虛擬記憶體，然而一些公司曾經出售過含有沒作任何改動的8086 CPU的分頁系統。對他們是如何做到這一點的做一個合理的猜想。（提示：想一想MMU的邏輯位置）

6. 如果一條指令需要1微秒，一個頁面故障需要另外n微秒，給出在頁面故障每k條指令發生一次時指令的實際執行時間。

7. 一個機器有32位元位址空間和8K頁面，頁表完全用硬體實現，每個入口一個32位字。在進程啟動時，頁表被以每個字100納秒的速度從記憶體拷貝到硬體中。如果每個進程運行100毫秒（包含裝入頁表的時間），多少比例的CPU時間被用來裝入頁表？

8. 一個32位元位址的電腦使用兩級頁表，虛位址被分為9位的頂級頁表域，11位的二級頁表域和偏移，頁面長度是多少？在位址空間中一共有多少個個頁？

9. 以下是一小段用於一個頁面長度512位元組的電腦的組合語言程式。程式位於位址1020，堆疊指標位於8192（堆疊向0增長）。給出這個程式產生的頁面引用序列。每個指令佔用4個位元組（1個字），對指令和資料的引用都應該被包括在引用序列中

load word 6144 into register 0

push register 0 onto stack

call procedure at 5120, stacking the return address

substract the immediate constant 16 from stack pointer

compare the actual parameter to immediate constant 4

jump if equal to 5152

10. 假設一個32位虛位址被分成a、b、c、d四個域，前三個用於一個三級頁表系統，第四個域d是偏移。頁面數與這四個域的大小都有關係嗎？如果不是，與那些有關、那些無關？

11. 一台電腦的處理器的位址空間有1024個頁面，頁表保存在記憶體中。從頁表中讀取一個字的開銷是500納秒。為了減小開銷，這個電腦使用了TLB，它有32個（虛位址物理位址）對，能在100納秒內完成查找。把平均開銷降到200納秒需要的命中率是多少？

12. VAX上的TLB沒有R位，為什麼？

13. 一台機器有48位虛位址和32位元物理位址，頁面是8K，在頁表中需要多少個入口？

14. 一台電腦有4個頁框，裝入時間、上次存取時間、和每個頁的R和M位如下所示（時間以時鐘滴答為單位）：

頁 裝入 上次引用 R M

0 126 279 0 0

1 230 260 1 0

2 120 272 1 1

3 160 280 1 1

(a) NRU將替換那個頁？

(b) FIFO將替換那個頁？

(c) LRU將替換那個頁？

(d) 第二次機會將替換那個頁？

15. 如果FIFO頁面替換演算法被用到4個頁框和8個頁面上，當頁框初始為空，引用序列為0172327103時會發生多少次頁面故障？如果使用LRU呢？

16. 一台小型電腦有四個頁框。在第一個時鐘週期時R位元是0111（頁0是0，其他是1）。在隨後的時鐘週期中這個值是1011、1010、1101、0010、1010、1100、0001。如果使用帶有8位元數目器的老化演算法，給出最後一個週期後四個計數器的值。

17. 把一個64K的程式從平均尋道時間30毫秒、旋轉時間20毫秒、每道32K的磁片上裝入，這需要多長時間？

(a) 在頁長為2K時

(b) 在頁長為2K時

頁隨機地分佈在磁片上。

18. PDP-1是最早的分時電腦之一，有4K 18位元字的記憶體。在每個時刻它在記憶體中保持一個進程。當調度程式決定運行另一個進程時，在記憶體中的進程將被寫到一個換頁鼓上，鼓的表面有4K個18位字。鼓可以從任何位址開始讀寫，你認為為什麼要選這個鼓？

19. 一台電腦為每個進程提供65536位元組的位址空間，劃分為4K的位元組的頁。一個特定的程式有32768位元組的正文、16386位元組的資料、和15870位元組的堆疊。這個程式能裝入位址空間嗎？如果頁面長度是512位元組，能放下嗎？別忘了一個頁不能同時包含兩個不同段的成分。

20. 人們已經觀察到在兩次頁面故障之間執行的指令數與分配給程式的頁框數直接成比例。如果可用記憶體加倍，頁故障間的平均間隔也加倍。假設一條普通指令需要1微秒，但是如果發生了頁面故障就需要2001微秒（即兩毫秒用於處理故障）。如果一個程式運行了60秒，期間發生了15000次頁面故障，如果可用記憶體是原來的兩倍這個程式運行需要 多長時間？

21. Frugal電腦公司的一組作業系統設計員正在思考在他們的新作業系統中減少對後備記憶體數量的需求的方法，有人建議根本不要把程式正文保存在交換區中，而是在需要的時候直接從二進位檔案中調頁進來，這種方法有什麼問題嗎？

22. 解釋內零頭和外零頭的區別。那個發生在分頁系統中？那個發生在使用純的分段的系統中？

23. 當分段和分頁同時使用時，就象在MULTICS中那樣，首先必須查找段描述符，然後是頁描述符。TLB也是這樣以兩級查找的方式工作的嗎？

24. 為什麼在MINIX的記憶體管理模式下必須有一個象chmem這樣的程式？

25. 修改MINIX使得僵死進程一旦進入僵死狀態就釋放它的記憶體，而不是一直等到它的父進程執行wait。

26. 在當前MINIX的實現中，當執行EXEC系統調用時，記憶體管理器檢查當前是否有足夠容納新的記憶體映射的空洞，如果沒有就拒絕這個調用。一個更好的演算法會看在當前的記憶體映射被釋放後是否有足夠大的空洞，實現這個演算法。

27. 當執行一條EXEC系統調用時，MINIX使用了一條技巧使檔案系統立刻讀入整個段。設計並實現一個類似的技巧以允許用相似的方式來寫核心轉儲（core dump）。

28. 修改MINIX以實現交換。

29. 4.7.5節中指出對於一條EXEC調用，通過在釋放當前進程的記憶體之前測試足夠的“空洞”可以得到一個次最優的實現。請重新對該演算法程式設計以獲得更好的性能。

30. 4.8.4節指出最好分別對正文段和資料段搜索“空洞”，請實現這一改進。

31. 請重新設計adjust，以避免被發送信號的進程由於過於嚴格的堆疊空間檢查而被不必要地撤銷的問題。

第五章 檔案系統

所有的電腦應用程式都要存儲資訊和檢索資訊。進程在運行時可以在自己的位址空間中存儲一定量的資訊。但其存儲容量只限於虛擬位址空間的大小，對有些應用程式，這已經足夠了；但對於其他的應用程式，例如飛機定票系統、銀行系統或者公司記錄保存系統，這個存儲空間又顯得太小了。

在進程的位址空間內保存資訊的第二個問題是：當進程終止時，資訊隨之丟失。對許多應用（例如資料庫）而言，它們的資訊必須保存幾周、幾個月、甚至一直保留。資訊隨著使用進程的終止而丟失是不可接受的。此外，即使系統突然崩潰，這些資訊也應該設法保存下來。

第三個問題在於經常有多個進程同時存取訊號（或者部分資訊）。如果我們在某個進程的位址空間記憶體有線上電話簿，只有那個進程可以對它進行存取。解決這個問題的方法是使資訊本身獨立於任何進程。

由此，對於長期的資訊存儲，我們有以下三個基本要求：

1. 必須能夠存儲大量的資訊。

2. 在使用資訊的進程終止時，資訊必須保存下來。

3. 多個進程可以併發地存取訊號。

解決所有這些問題的常用方法是把資訊以一種單元，即通常所說的檔（file）的形式存儲在磁片或其他外部介質上。然後，在需要時進程可以讀取這些資訊或者寫入新的資訊。存儲在檔中的資訊必須是永久性的，也就是說，它不會因進程的創建和終止而受到影響。只有當用戶顯式地刪除它時，檔才會消失。

檔是通過作業系統來管理的。檔的結構以及命名、存取、使用、保護和實現方法都是作業系統設計的主要內容。總體上，作業系統中處理檔的那部分被稱為檔案系統（file system）。檔案系統是本章的主要論題。

從用戶的觀點來看，檔案系統中最重要的方面是檔案系統如何呈現在他們面前。即一個檔由什麼組成、檔如何命名、如何保護檔，以及對檔可以進行哪些操作等等。而一些細節，比如是用連結表還是用點陣圖來記錄空閒存儲區，以及在一個邏輯塊中有多少個磁區等則無關緊要，雖然這些方面對檔案系統設計者相當重要。因此我們把本章分成幾節，前兩節分別講述了檔和目錄的使用者介面，接下來是有關檔案系統實現的詳細討論，然後研究檔案系統中的安全和保護機制。最後，我們介紹MINIX的檔案系統。

5.1 文件

這一節，我們從用戶角度來研究檔，即檔是怎麼使用的，它有些什麼特性。

5.1.1 檔命名

檔是一個抽象機器制，它提供了一種把資訊保存在磁片上而且便於以後讀取的方法。它必須這樣來實現，使使用者不必瞭解資訊存儲的方法、位置以及磁片實際運作方式等細節。

抽象機器制最重要的特徵或許是被管理物件的命名方法。因此，我們從檔命名開始討論檔案系統。在創建一個檔時，進程給出檔案名。進程終止後，檔仍然存在，其他進程使用該檔案名可以對它進行存取。

各種系統的檔命名規則略有不同，但所有作業系統都允許用一到八個字母組成的字串作為合法檔案名。因而，andrea、bruce和cathy都可以用作檔案名。數位和一些特殊字元也可用於檔案名之中，所以象2、urgent!和Fig.2-14也經常用作檔案名。許多檔案系統支援長達255個字元的檔案名。

有些檔案系統區分大小寫，而另一些則不加區分。UNIX屬於前一類，MS-DOS屬於後者。因此在UNIX系統中，barbara、Barbara、BARBARA、BARbara和BarBaRa等是不同的檔，而在MS-DOS中，他們都表示同一個檔。

許多作業系統支援兩部分檔案名，兩部分之間用句號加以分隔，比如prog.c。在句號後面的部分稱作檔副檔名(file extension)，它通常給出了與檔有關的一些資訊。在MS-DOS中檔案名由1-8個字元和1-3個字元的可選副檔名組成。在UNIX中，如果使用副檔名，則其長度完全由用戶決定，甚至一個檔之中可以含兩個或多個部分的副檔名。例如，prog.c.Z中，.Z通常表明檔prog.c已經使用Ziv\_Lempel壓縮演算法壓縮過。一些常用的文件副檔名及他們的含義如圖5-1。

圖5-1 一些典型的文件副檔名。

在某些時候，檔副檔名僅僅是一種慣例，並不強迫使用。以file.txt命名的檔可能是文字檔，但這個檔案名也主要用於提醒使用者，而並不是給電腦傳送特別的資訊。然而，C編譯器可能要求供其編譯的檔以.c結尾，否則它將拒絕編譯。

當某個程式可以處理幾種不同的檔時，這種慣例特別有用。比如C編譯器，可以編譯、連結多種檔，其中有些是C檔，有些是組合語言檔。這時副檔名顯得很有必要，利用它編譯器能夠區分哪些是C文件，哪些是彙編文件，哪些是其他文件。

5.1.2 檔結構

檔可以按幾種不同方式構成。圖5-2給出了三種常用的方式。圖5-2(a)中的檔是一個無結構位元組序列。事實上，作業系統不知道也不關心檔中有些什麼內容，它所見到的都是位元組。任何含義都只能加在使用者級程式中。UNIX和MS-DOS均使用這一方法。除此之外，WINDOWS 95基本上使用了MS-DOS的檔案系統，只是做了一些語法方面的改進(例如長檔名)，因此本章中關於MS-DOS的論述幾乎都適用於Windows 95。可是，WINDOWS NT的檔案系統完全不同。

圖5-2 三種文件。 (a)位元組序列。 (b)記錄序列。 (c)樹。

作業系統把檔看成位元組序列提供了很大的靈活性。使用者程式可以在檔中加入任何內容，並且以任何方便的形式來命名。這時，處理檔結構的任務由使用者完成，作業系統無法提供任何説明，但也不會設置障礙。對於需要做特殊操作的使用者來說，下面的檔結構是非常重要的。

在位元組序列結構上的第一步改進見圖5-2(b)。在這個模型中，檔是一個固定長度記錄的序列，每條記錄都有內部結構。把檔作為記錄序列的中心思想是：讀操作返回一條記錄，而寫操作重寫或追加一條記錄。在過去幾年中，當80列的穿孔卡片還在廣泛使用的時候，許多作業系統把他們的檔案系統建立在由80個字元的記錄組成的檔基礎之上。這些系統也支援由132個字元的記錄組成的檔，這種檔用於行式印表機(當時是132列的行式印表機)。程式以80個字元為單位讀取資料，以132個字元為單位輸出。當然，最後52個字元有可能都是空格。

CP/M就是一個把檔看作固定長度記錄序列的(老式)系統，它使用長度為128個字元的記錄。雖然，把檔視作為固定長度的記錄序列一度曾成為標準，現在這種想法已經過時了。

檔結構的第三種組成方式如圖5-2(c)所示。這種方式中，檔由一棵記錄樹構成，並非每條記錄都具有同樣的長度。在記錄的固定位置包含一個關鍵字域。記錄樹按關鍵字域進行排序，這便於對特定關鍵字進行快速查找。

在這種檔結構中，基本操作並不是取"下一條"記錄，而是獲取具有特定關鍵字的記錄，雖然取"下一條"記錄也是可以的。例如就圖5-2(c)中的檔zoo而言，使用者可以要求系統取關鍵字為pony的記錄，而不必關心記錄在檔中的確切位置。此外，在檔中能夠添加新記錄。這時，由作業系統而不是使用者來決定把記錄放在檔的什麼位置。這種檔結構和UNIX與MS-DOS中使用的無結構位元組流顯然不同，它廣泛使用在一些用於商業資料處理的大型電腦之中。

5.1.3 檔案類型

許多作業系統支援幾種類型的檔。例如，UNIX和MS-DOS中都有正規檔和目錄。UNIX還有字元設備檔和塊設備檔。正規檔(regular file)中包含有使用者資訊。圖5-2中的所有檔都是正規檔。目錄(directory)是管理檔案系統結構的系統檔。我們把目錄放在以後研究。字元設備檔(character special file)和輸入/輸出有關，用於模仿串列I/O設備。例如終端、印表機、網路等等。塊設備檔(block special file)則用於模仿磁片。本章我們主要討論正規檔。

一般來說，正規檔是ASCII檔或者二進位檔案。ASCII檔由多行正文組成。在某些系統中，每行用回車符結束。其他系統則用到分行符號。有時，回車符和分行符號同時使用。各行不一定要有同樣的長度。

ASCII檔的最大優點是可以原樣地顯示和列印，也可以用通常的文字編輯器進行編輯。此外，如果許多程式都以ASCII檔作為輸入和輸出，那麼我們就很容易把一個程式的輸出作為另一個程式的輸入，如同shell管道一樣。(管道方式來實現進程間通信實現起來並非更容易，但如果採用一種公認的標準來表示它的話，對資訊的解釋將會更容易，比如用ASCII碼。)

除ASCII檔外，另一種正規檔是二進位檔案，把二進位檔案印在印表機上得到的是一張晦澀難懂的表，裡面全是些亂七八糟的字元。二進位檔案往往具有一定的內部結構。

例如，在圖5-3(a)中，可以看到一個簡單的可執行的二進位檔案，它取自一個早期版本的UNIX。雖然技術上，這個檔只是一個位元組序列，但只有在檔有正確格式時，作業系統才會執行這個檔。這種檔有五段：檔頭、正文、資料、重定位位元和符號表。檔頭以所謂的魔數(magic number)開始，表明該檔是一個可執行檔(防止非此格式檔的偶然運行)。接下來是一些16位元的整數，給出了檔不同部分的長度、執行的起始位址和一些標誌位元。在檔頭之後是程式本身的正文和資料，在他們裝載到記憶體中時，使用重定位位進行定位。符號表則用於調試。

圖5-3 (a)可執行檔。 (b)存檔文件。

二進位檔案的第二個例子是一個UNIX存檔檔，它由許多編譯過但還沒有連結的庫過程(模組)組成。每個模組以模組頭開始，其中給出了模組名、創建日期、擁有者、保護代碼和模組長度等等。如同可執行檔一樣，模組頭中也充斥著二進位數字字，把他們在印表機上輸出同樣毫無意義。

所有的作業系統都必須識別一種檔案類型，即他們自己的可執行檔。有些作業系統則可以識別多種檔案類型。老式TOPS-20系統甚至可以檢查運行檔的創建時間，然後找到相應的原始檔案，看它在二進位檔案生成之後是否被修改過。如果修改過，作業系統自動重新編譯這個原始檔案。按照UNIX的用語，在shell中嵌入了make程式。這時，作業系統要求使用者使用強制的檔副檔名，以便確定二進位程式是由哪個原始檔案生成的。

類似地，當WINDOWS用戶按兩下檔案名時，以該檔作為參數，作業系統調用一個合適的程式運行。根據檔副檔名，作業系統可以決定運行哪個程式。

如果使用者執行系統設計者未曾預料的操作，這樣的強類型檔可能會引起麻煩。舉個例子，假設在一個系統中，程式輸出檔為dat類型(資料檔案)，如果使用者編寫一個程式格式化器，讀取.pas檔，進行轉換(例如把該檔轉換成標準的行首縮進風格)，再把轉換後的檔輸出，輸出檔具有.dat類型。如果用戶想用Pascal編譯器來編譯這個檔，但因為檔副檔名不符，Pascal編譯器將拒絕編譯。試圖把file.dat拷貝到file.pas也是徒勞，系統會認為這種拷貝是無效的(防止用戶誤操作)。

儘管這種"用戶友好性"對初學者有利，卻使得一些有經驗的用戶大傷腦筋。他們不得不花很大的精力來適應作業系統對合理操作和不合理操作的區分。

5.1.4 文件存取

早期的作業系統只提供了一種檔存取方式：順序存取(sequential access)。在這些系統中，進程可以從檔開始處順序讀取檔中所有位元組或者記錄，但不能夠略過某些內容，也不能夠非順序讀取。順序存取檔可以重繞，只要需要，可以多次讀取該檔。當存儲媒體是磁帶，而不是磁片時，用順序存取檔是非常方便的。

用磁片存儲檔後，我們可以非順序地讀取檔中的位元組或記錄，或者根據關鍵字而不是位置來存取記錄。能夠以任何順序讀取的檔叫做隨機存取文件(random access file)。

對許多應用程式來說，隨機存取檔是必不可少的，例如資料庫系統。如果一個飛機乘客打來電話，想要預訂特定航班的機票。訂票程式必須能夠直接存取該航班的記錄，而不必首先讀出成千上萬條其他航班的記錄。

有兩種方法指明從哪兒開始讀取檔。第一種是，每次READ操作都給出檔中開始讀的位置。另一種方法是，提供一個特殊的SEEK操作來設置當前位置，此後，從這個新的當前位置開始順序地讀取檔。

有些早期的主機作業系統中，檔在創建時，就指定為是順序存取檔或者隨機存取檔。對這兩類檔，系統使用不同的存儲技術。現代作業系統則不加區分，創建後，所有檔自動成為隨機存取檔。

5.1.5 檔案屬性

每個檔都有檔案名和資料。此外，所有作業系統還給檔賦以其他資訊，比如，檔創建日期、檔長度等等。我們把這些額外的項稱為檔案屬性(attribute)。不同系統的屬性差別很大。圖5-4的表中列出了一些可能的屬性，但其他的屬性也存在。沒有一個系統具有全部這些屬性，然而，每種屬性都在某個系統中使用。

圖5-4 一些可能的檔案屬性。

開始四個屬性與檔案保護有關，它給出了誰可以存取這個檔，誰不能存取這個檔。存在著各種不同的檔案保護方案，其中的一些我們會在以後討論。在一些系統中，使用者必須給出口令才能存取檔，這時，口令也是檔案屬性之一。

標誌是一些位元或者短域，用來抑制或者允許某些特定性質。例如隱藏檔使其不出現在檔的顯示清單中。存檔標誌位元記錄檔是否備份過。備份程式清除該標誌位元，在檔修改後，作業系統設置存檔標誌位元。這樣，備份程式可以區分哪些檔需要備份。臨時標誌位元表示在創建該檔的進程終止後，它被自動刪除。

記錄長度、關鍵字位置和關鍵字長度等域只出現在那些能夠用關鍵字查找記錄的檔之中，他們提供了查找關鍵字所需資訊。

各個時間域記錄了檔的創建時間、最近存取時間以及最近修改時間等等。他們可用於不同目的。例如，在相應的目的檔案生成後修改過的原始檔案需要重新編譯，這些域提供了必要的資訊。

當前長度域給出了文件當前的大小。在一些主機作業系統中，檔創建時，需要指明檔最大長度，以便作業系統事先保留一定的存儲空間。工作站和個人電腦的作業系統則無需指明這一點。

5.1.6 檔操作

檔用於存儲資訊便於以後檢索。不同系統提供了不同的操作進行存儲和檢索。下面是一些與檔有關的最常用的系統調用：

1. CREATE。創建沒有任何資料的檔。該調用的目的是聲明檔存在，並且設置一些屬性。

2. DELETE。當檔不再需要時，必須刪除它以釋放磁碟空間。DELETE系統調用可用於實現這一目的。

3. OPEN。在使用檔之前，進程必須打開檔。OPEN調用的目的是：將檔案屬性和磁片位址表載入主存，便於以後系統調用的快速存取。

4. CLOSE。當存取結束後，檔案屬性和磁片位址就不再需要了，這時應該關閉檔以釋放內部表空間。許多系統限制進程的打開檔數，鼓勵用戶關閉不再使用的檔。磁片以塊為單位寫入，關閉檔可以迫使檔最後一塊寫回磁片，儘管這一塊可能還沒有寫滿。

5. READ。從檔中讀取資料。一般，讀出的資料來自當前位置。調用者必須指明需要讀取多少資料，並且提供存放這些資料的緩衝區。

6. WRITE。向檔中寫入資料，寫操作一般也是從當前位置開始。如果當前位置是檔末尾，檔長度增加。如果當前位置在檔中間，則現有資料被重寫，並且永遠丟失了。

7. APPEND。該調用是WRITE的限制形式，它只能在檔末尾添加資料。只提供最小系統調用集的系統通常沒有APPEND。許多系統對同一操作提供了多種實現方法，這些系統中往往有APPEND調用。

8. SEEK。對於隨機存取檔，需要指定從哪兒開始讀寫資料，常用的方法是用SEEK系統調用把當前位置指標指向檔中特定位置。此後，從該位置開始讀寫資料。

9. GET ATTRIBUTES。進程往往需要讀取檔案屬性，例如，UNIX中make程式常用於管理由多個原始檔案組成的軟體發展項目。在調用make時，檢查所有原始檔案和目的檔案的修改時間，安排最小數目的編譯，使得所有檔都為最新版本。為實現該目的，需要查找檔的某些屬性，如修改時間。

10. SET ATTRIBUTES。有些屬性是使用者可以設置的，在檔創建之後，能夠修改他們。SET ATTRIBUTES系統調用使之成為可能。保護模式資訊是一個很明顯的例子，大多數標誌亦屬此類。

11. RENAME。用戶常常要改變現有檔的檔案名。系統調用RENAME可以實現這一目的。嚴格說來，這個調用並非必要，因為我們可以先把檔拷貝到新檔案名的檔中，然後刪除原來的檔。

5.2 目錄

為了記錄檔，檔案系統通常有目錄。在許多系統中，目錄本身也是檔。本節討論目錄、它的組成、特性和可以進行的操作。

5.2.1 層次目錄系統

目錄通常包含有許多目錄項，每個目錄項代表一個檔。圖5-5(a)中是一種可能情況，其中每個目錄項包含檔案名、檔案屬性和檔資料在磁片上的位址等等。另一種情況見圖5-5(b)，這裡，目錄項中含有檔案名和指向另一個資料結構的指標，檔案屬性和磁片位址就放在這個資料結構之中。以上兩種系統都得到了廣泛的應用。

圖5-5 (a)屬性放在目錄項中。 (b)屬性放在其他地方。

在打開檔時，作業系統查找目錄，直到找到要打開檔的檔案名。然後從目錄項或者所指向的資料結構中取得檔案屬性和磁片位址，放入記憶體的相應表中。之後，對該檔的所有引用均使用主存中的資訊。

每個系統的目錄數目各不相同。最簡單的設計方案是維護一個單獨的目錄，其中包含所有使用者的全部檔，如圖5-6(a)。如果有很多用戶，他們使用相同的檔案名(如mail和games)，這時會出現衝突和混亂，使得系統無法正常工作。這一系統模型用在第一代微機作業系統中，而現在已經很少見到了。

圖5-6 三種檔案系統設計。 (a)所有使用者共用一個目錄。(b)每個使用者擁有一個目錄。 (c)每個用戶有一棵任意樹。圖中字母表示目錄或檔擁有者。

對於整個系統中使用單獨一個目錄管理所有檔的想法的改進是，每個使用者擁有一個目錄[參見圖5-6(b)]。這種設計消除了不同用戶之間的檔案名衝突，但仍然難以使那些有許多檔的用戶感到滿意。使用者常常需要把他們的檔按某種邏輯方式組織起來。例如，一個教授，可能有許多檔一起構成了他正在為某門課程編寫的一本書，另一些檔包含了學生交上來的另一門課程的程式，此外，他的第三組檔是他正在構造的高級編譯器編寫系統的代碼，第四組檔則包含有資助申請表，同時，他還有一些其他的檔，如電子郵件、會議記錄、論文、遊戲等等。這時，需要某種方法，使得用戶可以按自己選擇的方式把檔組織起來。

我們需要的是一般的層次結構(即目錄樹)。使用層次結構，每個使用者可以擁有所需的多個目錄，以便自然地組織他們的檔。這種方法如圖5-6(c)所示。圖中，根目錄包含有目錄A、B、C，他們分別屬於不同的用戶，其中有兩個用戶為他們正在做的專案創建了子目錄。

5.2.2 路徑名

使用目錄樹來組織檔案系統時，需要某種方法指明檔案名。通常用到的方法有兩種。第一種是，每個檔都賦予一個絕對路徑名(absolute path name)，它由從根目錄到檔的路徑組成。例如，路徑/usr/ast/mailbox表示根目錄中含有子目錄usr，而usr中又包含子目錄ast，檔mailbox就放在目錄ast下。絕對路徑名總是從根目錄開始，並且是唯一的。在UNIX中，路徑各部分之間用"/"分隔。在MS-DOS中，分隔符號是"\"。在MULTICS中是">"。不管使用哪個分隔符號，如果路徑名的第一個字元是分隔符號，那麼這個路徑就是絕對路徑。

另一種檔案名是相對路徑名(relative path name)。它常和工作目錄(也稱作目前的目錄)的概念一起使用。使用者可以指定一個目錄作為當前的工作目錄。這時，所有的路徑名，如果不是從根目錄開始，都是相對於工作目錄的。例如，如果當前的工作目錄是/usr/ast，則絕對路徑名為/usr/ast/mailbox的檔可以簡單地用mailbox來引用。換句話說，如果工作目錄是/usr/ast，則UNIX命令

cp /usr/ast/mailbox /usr/ast/mailbox.bak

和

cp mailbox mailbox.bak

具有相同的含義。相對路徑往往更加方便，但是，它實現的功能和絕對路徑完全一樣。

有些程式需要存取某個特定檔，而不管當前的工作目錄是什麼。這種情況下，應該使用絕對路徑名。比如，一個拼寫檢查程式要讀取檔/use/lib/dictionary，而它不知道當前的工作目錄，就須使用完整的絕對路徑名。無論當前的工作目錄是什麼，絕對路徑名總是可以使用的。

當然，如果這個拼寫檢查程式需要從目錄/usr/lib中讀取很多檔，它可以採用另一種方法，即執行一個系統調用，把工作目錄切換到/usr/lib，然後只需用dictionary作為open的第一個參數。通過顯式地改變工作目錄，程式可以知道它在目錄樹中的確切位置，因此可以使用相對路徑名。

大多數系統中，每個進程都有自己的工作目錄，因而，在進程改變它的工作目錄並退出後，其他的進程不會受到影響，在檔案系統中也不會留下改變的痕跡。對進程來說，工作目錄的切換是安全的，因此只要需要，它就可以改變當前的工作目錄。可是，如果庫過程改變了工作目錄，在結束前不改為原來的目錄，那麼程式其他部分可能無法正常運行，因為他們關於目前的目錄的假設已經無效了。所以，在庫過程中，很少改變工作目錄，如果非改不可的話，總是在返回之前改回到原來的工作目錄。

大多數支援層次目錄結構的作業系統，在每個目錄中有兩個特殊的目錄項"."和".."，通常讀作"dot"和"dotdot"。dot指目前的目錄，dotdot指其父目錄。要瞭解他們是如何使用的，我們可以考慮圖5-7中的UNIX檔樹。某個進程的工作目錄是/usr/ast，它可以使用".."沿樹向上到達其父目錄/usr。例如，它可以使用shell命令

cp ../lib/dictionary .

把檔/usr/lib/dictionary拷貝到自己的目錄下。第一個路徑告訴系統上溯(到usr目錄)，然後向下到達lib目錄，找到dictionary檔。

圖5-7 一棵UNIX目錄樹。

第二個參數指定為目前的目錄。當cp命令用一個目錄名(包括".")作為它的第二個參數時，它把所有的檔拷貝到該目錄中。當然，對於上述拷貝，更普遍的方法是鍵入

cp /usr/lib/dictionary .

這裡，用戶使用"."避免了第二次鍵入dictionary。

5.2.3 目錄操作

相對於檔的系統調用而言，各個系統中用於管理目錄的系統調用差別更大。為了讓讀者對這些系統調用及其工作方式有一個印象，我們下面將給出一個例子(取自UNIX)。

1. CREATE。創建目錄。除了目錄項"."和".."之外，目錄內容為空。目錄項"."和".."是系統自動放在目錄中的(有時通過mkdir程式)。

2. DELETE。刪除目錄。只有空目錄可以被刪除。只含有目錄項"."和".."的目錄都認為是空目錄，這兩個目錄項是不能被刪除的。

3. OPENDIR。目錄內容可被讀取。例如，為了列出目錄中的所有檔，清單程式必須先打開該目錄，然後讀取其中所有檔的檔案名。同打開和讀取檔一樣，在讀目錄之前，必須打開目錄。

4. CLOSEDIR。讀目錄結束後，應該關閉該目錄以釋放內部表空間。

5. READDIR。系統調用READDIR返回打開目錄的下一個目錄項。以前我們也使用通常的READ系統調用來讀目錄，但這種方法有一個缺點：程式師必須瞭解目錄的內部結構。相反，不管使用哪一種目錄結構，READDIR總是以標準格式返回一個目錄項。

6. RENAME。在很多方面，目錄和檔相似。檔可以換名，目錄亦然。

7. LINK。連結技術允許檔出現在多個目錄中。這個系統調用指定一個存在的檔和一個路徑名，並建立從檔到路徑所指定的名字的連結。這樣，同一檔可以在多個目錄中出現。

8. UNLINK。刪除目錄項。如果被解鏈的檔只出現在一個目錄中(正常情況)，它從檔案系統中被刪除。如果它出現在多個目錄中，只刪除指定的路徑名，其他路徑名依然保留下來。在UNIX中，刪除檔的系統調用(前面已有論述)實際上就是UNLINK。

以上列出了最主要的系統調用。但還有一些其他調用，比如管理與目錄相關的保護資訊的系統調用。

5.3 檔案系統的實現

現在我們從用戶角度轉到實現者角度來研究檔案系統。用戶關心的是檔是如何命名的、可以進行哪些操作、目錄樹是什麼樣的以及類似的介面問題。而實現者感興趣的是檔和目錄是如何存儲的、磁碟空間是怎樣管理的以及如何使系統有效而可靠地工作等等。在下面幾節中，我們將研究檔案系統的實現中出現的一些問題以及如何來解決這些問題。

5.3.1 實現檔

或許在實現檔存儲中最重要的問題是記錄各個檔分別用到哪些磁片塊。不同作業系統採用不同的方法。這一節，我們將討論其中一些方法。

連續分配

最簡單的分配方案是把每個檔作為連續資料塊存儲在磁片上。因此，在具有1K大小塊的磁片上，50K的檔要分配50個連續的塊。這一分配方案有兩大優點。首先，簡單、容易實現，記錄每個檔用到的磁片塊僅需記住一個數字即可，也就是第一塊的磁片位址。其次，性能較好，在一次操作中，就可以從磁片上讀出整個檔。

不幸的是，連續分配方案也有兩個很大的不足。首先，除非在檔創建時就知道了檔的最大長度，否則這一方案是行不通的。不知道檔的最大長度，作業系統也就無法確定要保留多少磁碟空間。但是，在那些檔資料一次性寫入的系統中，連續分配的優點可以得到充分利用。

第二個不足之處是：該分配方案會造成磁片碎片。原本可以使用的空間被浪費了。磁片壓縮的代價往往很高，儘管這可以在深夜，當系統空閒的時候進行。

連結表分配

存儲檔的第二種方法是為每個檔構造磁片塊的連結表，如圖5-8所示。每個塊的第一個字用於指向下一塊的指標，塊的其他部分存放資料。

圖5-8 按磁片塊連結表來存儲檔。

與連續分配方案不同，這種方法中每個磁片塊都被利用了。不會因為磁片碎片而浪費存儲空間(最後一塊的內零頭除外)。同樣，在目錄項中，只需要存放第一塊的磁片位址。檔的其他塊可以根據這個位址來查找。

然而，在連結表分配方案中，儘管順序讀取檔非常方便，但是隨機存取卻相當緩慢。此外，因為指標占去了一些位元組，每個磁片塊存儲資料的位元組數不再是2的冪，雖然這個問題並不足以致命，但它確實降低了系統的運行效率，因為大多數程式都是以長度為2的冪來讀寫磁片塊的。

使用索引的連結表分配

如果取出每個磁片塊的指標字，把它放在記憶體的表或索引中，就可以消除上述連結表

的兩個不足。就圖5-8的例子而言，記憶體中表的內容如圖5-9所示。這兩個圖中，有兩個文件。檔A依次使用了磁片塊4、7、2、10和12，檔B依次使用了磁片塊6、3、11和14。利用圖5-9的表，我們可以從第4塊開始，順著鏈到底，找到檔A的所有磁片塊。同樣，從第6塊開始，順著鏈到底，也能夠找出檔B的所有磁片塊。

圖5-9 使用記憶體表的連結表分配。

這樣組織的話，整個塊都可以存放資料。此外，隨機存取也容易得多。要查找檔給定偏移位置，仍然要順著鏈進行，但是整個鏈表都存放在記憶體中，不需要訪問磁片。同以前的方法一樣，不管檔有多大，在目錄項中只需記錄一個整數(起始塊號)，根據它可以找到檔的所有塊。MS-DOS就使用這種方法進行磁片分配。

這種方法的主要缺陷是我們必須把整個鏈表都存放在記憶體中。對於大磁片，例如具有500000個1K大小塊(500M)的磁片，其鏈表中含有500000項，每項至少3個位元組，為了提高查找速度，有時需要4個位元組。這樣，對於空間或時間的不同優化方案，這張表要佔用1.5或2M記憶體。儘管MS-DOS使用這種分配方案，它在大磁片上使用大的塊(長達32K)，從而避免了使用很大的表。

i- 節點

記錄各個檔分別包含哪些磁片塊的最後一個方法是給每個檔賦予一張稱為i-節點(索引節點)的小型表，其中列出了檔案屬性和各塊在磁片上的位址，參見5-10。

圖5-10 i-節點。

開始幾個磁片位址存放在i-節點內，因此對於小檔，所需資訊均在i-節點中。在打開檔時，這些資訊從磁片讀入主存。稍大一些的檔，在i-節點中有一個稱為一次間接塊的磁片塊的位址，這個磁片塊中含有附加的磁片位址。如果檔再擴大，可以使用i-節點中的另一個位址，即所謂的二次間接塊的位址，二次間接塊包含有許多一次間接塊的位址，而每個一次間接塊又指向幾百個資料塊。如果這還不夠的話，我們也可以使用三次間接塊。UNIX中使用了這種分配方案。

5.3.2 實現目錄

在讀檔前，必須先打開檔。打開檔時，作業系統利用使用者給出的路徑名找到相應目錄項，目錄項中提供了查找檔磁片塊所需的資訊。由於系統不同，這些資訊可能是整個檔的磁片位址(連續分配方案)、第一個塊的塊號(對於兩種連結表分配方案)或者是i-節點號。無論怎樣，目錄系統的主要功能是把ASCII檔案名映射成查找檔資料所需的資訊。

與此密切相關的問題是在哪兒存放檔案屬性。一種較明顯的方法是把檔案屬性直接存放在目錄項中。許多系統確實是這樣實現的。對於使用i-節點的系統，還存在另一種可能，即把檔案屬性存放在i-節點中。以後我們將看到，這種方法比把屬性存放到目錄項中更為優越。

CP/M中的目錄

讓我們先從一個特別簡單的例子，CP/M(Golden and Pechura，1986)，來研究目錄。CP/M的目錄項如圖5-11中所示。在這個系統中只有一個目錄，因此要查找檔案名，檔案系統所要做的是查找這個唯一的目錄。當找到對應的目錄項後，也就知道了檔的磁片塊號。同所有檔案屬性一樣，檔的磁片塊號也存放在目錄項中。如果檔的磁片塊數多於一個目錄項中所能容納的數目，就為這個檔分配額外的目錄項。

圖5-11 目錄項中包含每個檔的磁片塊號。

圖5-11中各個域的含義如下。使用者碼域記錄了檔擁有者。在查找過程中，只檢查那些屬於當前登錄使用者的目錄項。接下來兩個域給出了檔案名和副檔名。多於16塊的檔佔有多個目錄項，這時使用範圍域。根據這個域，我們可以知道哪個目錄項是檔的第一個目錄項，哪個是第二個，等等。塊數域給出了目錄項中16個可能的磁片塊中實際使用的塊數。最後16個域包含磁片塊號本身。最後一個塊可能沒有寫滿，因而系統無法確切地知道檔的位元組數(即它是以磁片塊，而不是位元組為單位來記錄檔長度的)。

MS-DOS中的目錄

我們現在考慮層次目錄系統的一些例子。圖5-12是一個MS-DOS的目錄項。它總共32個位元組長，其中包含了檔案名、檔案屬性和第一個磁片塊的塊號。根據第一個磁片塊的塊號，順著圖5-9中的鏈，我們可以找到檔的所有塊。

圖5-12 MS-DOS目錄項。

在MS-DOS中，目錄可以包含其他目錄，從而形成層次檔案系統。通常在MS-DOS中，每個應用程式在根目錄下創建一個目錄，把它的所有檔都放在這個目錄下，因此不同的應用程式不會發生衝突。

UNIX中的目錄

UNIX中使用的目錄結構非常簡單，如圖5-13所示，每個目錄項只包含一個檔案名及其i-節點號。有關檔案類型、長度、時間、擁有者和磁片塊等所有資訊都放在i-節點中。有些UNIX系統有不同的佈局，但無論如何，目錄項中最終要包含一個ASCII字串和一個i-節點號。

圖5-13 UNIX目錄項。

在打開檔時，檔案系統必須根據給出的檔案名找到它所在的磁片塊。讓我們分析如何來查找路徑名/usr/ast/mbox。儘管是以UNIX作為例子，但是這個查找演算法對所有的層次目錄系統基本上都一樣。首先檔案系統找到根目錄。在UNIX中，根目錄的i-節點位於磁片上的固定位置。

然後在根目錄中查找路徑的第一部分，usr，從而也就獲得了檔/usr的i-節點號。因為每個i-節點都位於磁片的固定位置，所以根據i-節點號找到i-節點是很直接的。利用這個i-節點，檔案系統找到目錄/usr，並接著查找下一部分ast。當找到ast目錄項後，得到目錄/usr/ast的i-節點。從而找到目錄/usr/ast並在該目錄中查找檔mbox。接著，檔mbox的i-節點被讀入記憶體，並保存在記憶體中，直至關閉該檔。這個查找過程見圖5-14。

圖5-14 查找/usr/ast/mbox的過程。

相對路徑名的查找也類似。只是，相對路徑名從工作目錄，而不是根目錄開始查找。每個目錄在創建時都含有"."和".."項。"."項給出目前的目錄的i-節點號，".."項給出了其父目錄的i-節點號。因而，查找../dick/prog.c的過程僅僅是在工作目錄中查找".."項，找到父目錄的i-節點號，並在父目錄中查找到dick目錄。我們不需要特別的機制來處理這些檔案名，就目錄系統而言，和其他的檔案名一樣，他們都僅僅是一些ASCII字元。

5.3.3 磁碟空間管理

檔通常存放在磁片上，所以磁碟空間的管理是系統設計者要考慮的一個主要問題。存儲n個位元組的檔可以有兩種策略：分配n個位元組的連續磁碟空間，或者把檔分成許多個(並不一定要)連續的塊。在記憶體管理器系統中，單純段式和分頁也要進行同樣的權衡。

按連續位元組序列來存儲檔有一個明顯問題，當檔擴大時，可能需要在磁片上移動檔。記憶體中分段也具有同樣的問題。不同的是，相對於把檔從磁片的一個位置移動到另一個位置，段在記憶體中的移動操作要快得多。由於這個原因，幾乎所有的檔案系統都把檔分割成固定大小的塊來存儲，各塊不必相鄰。

塊大小

一旦決定把檔按固定大小的塊來存儲，就會有一個問題：塊大小應為多少呢？根據磁片組織方式，磁區、磁軌和柱面顯然都可以作為分配單位。在頁式系統中，頁面大小也是主要選項之一。

如果分配單位大，比如以柱面為分配單位，這時每個檔，甚至是1個位元組的檔，都要佔用整個一個柱面。研究(Mullender和Tanenbaum，1984)表明，UNIX環境下的平均檔長度為1K。因而分配32K的柱面將浪費31/32或者說97％的磁碟空間。另一方面，小的分配單位意味著每個檔由許多塊組成。每讀一個塊都要有尋道延遲和旋轉延遲，因此，讀取由許多小塊組成的檔非常緩慢。

舉一個例子，假設磁片每道有32768個位元組，其旋轉時間為16.67毫秒，平均尋道時間為30毫秒。以毫秒為單位，讀取一個k個位元組的塊所需時間是尋道延遲、旋轉延遲和傳送時間之和：

30+8.3+(k/32768)\*16.67

圖5-15的實線顯示一個磁片的資料讀取速率與塊大小之間的關係。如果我們粗略地假設所有檔都是1K位元組(所測檔平均長度)，則磁碟空間的利用率如圖5-15中虛線所示。不幸的是，從圖中我們看到，隨著磁碟空間利用率的提高，讀取磁片資料的速率降低。反之，在資料讀取速率提高時，磁碟空間利用率隨之下降。時間效率和空間效率本質上相互衝突。

圖5-15 實線(左邊標度)給出磁片資料率，虛線(右邊標度)給出磁碟空間效率， 所有檔均為1K。

常見的折衷辦法是把塊大小選為512、1K或2K位元組。如果在磁區大小為512位元組的磁片上選擇1K大小的磁片塊，檔案系統通常讀寫兩個連續的磁區，並把他們看成是一個不可分割的單元。不管作了何種選擇，我們都必須定期地重新計算塊大小。這是因為，同電腦技術的其他方面一樣，用戶可使用的資源越來越多，但是，他們的需求也越來越高。有系統管理員聲稱，在他所管理的大學系統中，檔的平均長度近年來緩慢地增長，在1997年，學生的平均檔長度增至12K，而教師的則增至15K。

記錄空閒塊

一旦選定了塊大小，接下來的問題就是如何記錄空閒塊。如圖5-16所示，兩種方法得到了廣泛使用。第一種使用磁片塊的連結表。每個塊中包含盡可能多的空閒磁片塊號。對於1K大小的塊和32位元的磁片塊號，空閒塊鏈表中每個塊包含有255個空閒塊塊號(我們需要使用一個項存放指向下一塊的指標)。200M的磁片最多需要804個塊的空閒鏈表來存放所有200K個磁片塊的塊號。通常情況下，我們使用空閒塊存放空閒塊鏈表。

圖5-16 (a)將空閒表存放在連結表中。 (b)點陣圖。

空閒磁碟空間管理的另一種方法是使用點陣圖。n個塊的磁片需要n位元點陣圖。在點陣圖中，空閒塊用1表示，分配塊用0表示(或者反之)。200M的磁片需要有200K位元來映照，即只需25個塊。毋庸置疑，點陣圖模型所需空間要少於鏈表模型所需磁碟空間，在鏈表模型中每個塊用到32位元，而在點陣圖模型中每塊僅需1位。只有在磁片快滿時連結表方案需要的塊比點陣圖更少。

如果有足夠的記憶體來存放點陣圖，那麼這種方法無疑是非常好的。但是，如果只有一個塊的記憶體可以用於記錄空閒磁片塊，並且空閒磁片塊很少，這時選擇連結表方案更佳。假設記憶體中只存放一個塊的點陣圖，那麼可能在這個點陣圖塊中根本找不到空閒塊，這時，我們不得不從磁片上調入其他點陣圖塊。而將一個空的連結表塊裝入記憶體後，在從磁片上讀取連結表的下一塊之前，我們可以進行多達255次分配。

5.3.4 檔案系統的可靠性

比起電腦的損壞，檔案系統的破壞往往要糟糕得多。如果由於火災、閃電電流或者一杯咖啡潑在鍵盤上而弄壞了電腦，確實讓人傷透腦筋，而且又要花上一筆錢，但一般來說，更換非常方便。求助於分銷商，便宜的個人電腦在短短幾個小時之內就可以修復(當然，如果這發生在大學裡面，發出購買定單需征得3個委員會的同意，並蓋5個公章，總共要花90天的時間，那麼情況就不一樣了)。

不管是硬體或軟體的故障，或者是老鼠咬壞了軟碟，如果電腦的檔案系統被破壞了，恢復所有資訊將是一件困難而又費時的工作，有些時候，這根本是不可能的。對於那些其程式、文檔、客戶檔、稅收記錄、資料庫、市場計畫或者是其他資料丟失的使用者來說，這不啻為一次大的災難。儘管檔案系統無法防止設備和媒體的物理損壞，它至少應能保護資訊。這一節，我們討論與檔案系統保護有關的一些問題。

正如我們在第三章中所指出的那樣，磁片中可能有壞塊。在出廠時，軟碟通常是完好無損的，然而在使用過程中，卻可能出現壞塊。溫徹斯特磁片常常一開始就有壞塊：要把它做得完美無缺，成本實在是太高了。事實上，在老式磁片上往往保留一個磁區作為壞塊表，在出廠時其中記錄著測試到的壞塊。在控制器第一次初始化時，它讀取壞塊表，選擇一個備用塊(或磁軌)取代有缺陷的塊，並把這一資訊記錄在壞塊表中。以後，所有對於壞塊的請求都使用備用塊。在發現新的壞塊時，可以進行低級別的格式化來修改這個表。

製造技術已有了穩步的提高，壞塊不再象以前那麼普遍了，但還是會出現。我們在第三章說過，現代磁碟機的控制器非常複雜。在這些磁片上，磁軌要比所需的至少多一個磁區，於是在一條磁軌上出現壞塊就不會造成任何問題，而只需要在讀取資料時跳過它。 在每個柱面上還有一些備用的磁區，以便控制器在注意到讀寫某個磁區的複執次數大於某值時，能夠自動將其重新映射至一個備用磁區。因而用戶往往不會意識到壞塊的存在以及壞塊的管理。可是，當一個現代IDE或SCSI磁片失效時，這往往是很可怕的，因為這時所有的備用塊都已用光。SCSI磁片在重映射磁片塊時， 給出"恢復時出錯"資訊。如果驅動器發現該錯誤資訊，則將列印出一條消息，如果使用者看到這條消息經常出現，就會知道他應該購買一個新磁片了。

此外，對於壞塊問題，還可以用軟體方法加以解決，這種方法適合於老式磁片，它要求使用者或檔案系統精心地構造一個包含全部壞塊的檔。這種技術能將壞塊從空閒表中刪除，使其不會出現在資料檔案之中。只要不對壞塊進行讀寫操作，檔案系統就不會出現任何問題。在磁片備份時，需要注意避免讀取這個檔。

備份

即使有再好的處理壞塊的策略，我們也需要經常地備份檔案。畢竟，在一些關鍵的資料塊損壞之後，自動切換到備用塊，無異於亡羊補牢。

備份軟碟上的檔案系統很簡單，只需把整個磁片拷貝到一張空軟碟上。小型溫徹斯特磁片上的檔案系統可以轉儲到磁帶上。現代技術中使用了150M的磁帶和8G的Exabyte或DAT磁帶。

對於大型溫徹斯特磁片(例如10GB)，把整個驅動器的內容轉儲到磁帶上是一件可怕而費時的工作。一種容易實現、但浪費了一半存儲空間的策略是，每台電腦備有2個驅動器，每個驅動器都分成兩部分：資料區和備份區，每天晚上，驅動器0中的資料被拷貝到驅動器1的備份區，反之，驅動器1中的資料拷貝到驅動器0的備份區，如圖5-17。這樣，即使有一個驅動器完全損壞了，也不會丟失任何資訊。

圖5-17 將一個驅動器中的資料備份到另一個驅動器上浪費了一半存儲空間。

另一種轉儲整個檔案系統的方法是增量轉儲(incremental dump)。最簡單的增量轉儲形式是：定期地做一次全量轉儲，比如每週一次或者每月一次，此後，每天只存儲自上次全量轉儲以後修改過的檔，或者可以採用一種更好的方案，即每天只轉儲那些自上次增量轉儲以來修改過的檔。

要實現這一方法，必須在磁片上保存一張表，記錄每個檔的轉儲時間。轉儲程式檢查磁片上的每個檔，如果這個檔在上次轉儲之後修改過，則再次轉儲，並且把它的"上次轉儲時間"改為當前時間。如果以月為週期進行轉儲，這種方法需要31盤日常轉儲磁帶(每日一盤)，和足夠多的磁帶用於每月一次的全量轉儲。我們有時也用到其他更複雜但需要磁帶更少的方法。

檔案系統一致性

影響檔案系統可靠性的另一個問題是檔案系統的一致性。許多檔案系統讀取磁片塊，進行修改後，再寫回磁片。如果在修改過的磁片塊全部寫回之前，系統崩潰，那麼檔案系統可能出現不一致。如果一些未被寫回的塊是i-節點塊、目錄塊或者包含空閒表的磁片塊時，這個問題尤為嚴重。

為了解決檔案系統的不一致問題，許多電腦都帶有一個實用程式，檢驗檔案系統的一致性。系統初啟時，特別是在崩潰之後重新開機，可以運行該程式。下面我們講述在UNIX和MINIX中，這個實用程式是如何工作的。在其他系統中，其工作原理類似。這些檔案系統檢驗程式可以獨立地檢驗各個檔案系統(磁片)的一致性。

一致性檢查分為兩種：塊的一致性檢查和檔的一致性檢查。在檢查塊的一致性時，該程式建立兩張表。每張表中，每塊對應有一個計數器，初始值設為0。第一張表的計數器記錄了每塊在檔中出現的次數，第二張表的計數器記錄了每塊在空閒塊鏈表(或空閒塊點陣圖)中出現的次數。

檢驗程式讀取所有的i-節點，從i-節點開始，可以建立相應檔中使用的所有塊的塊號表。每當讀到一個塊號時 ，該塊在第一張表中的計數器加1。接著這個程式檢查空閒塊鏈表或點陣圖，查找所有未使用的塊。每當在空閒表中找到一個塊時，它在第二張表中的計數器加1。

如果檔案系統一致，則每個塊要麼在第一張表中為1，要麼在第二張表中為1，如圖5-18(a)所示。可是系統崩潰後，這兩張表可能如圖5-18(b)。在圖中，磁片塊2不出現在任何一張表中，這時報告塊丟失。儘管塊丟失不會造成損害，但卻浪費了磁碟空間，減少了磁片容量。解決塊丟失問題是很直觀的：檔案系統檢驗程式只需要把他們加到空閒表中。

圖5-18 檔案系統狀態： (a)一致 (b)塊丟失 (c)空閒表中有重複塊 (d)重復資料塊

另一種可能出現的情況見圖5-18(c)。這裡，我們看到，磁片塊4在空閒表中出現了2次(只有在空閒表是一張真正意義上的鏈表時，才會出現重複，在點陣圖中，這種情況不會發生。)它的解決方法也是很簡單的：我們只需要重新建立空閒表。

最糟糕的情況是，同一個資料塊在兩個或多個檔中出現，如圖5-18(d)中的磁片塊5。如果刪除任何一個檔，磁片塊5會加到空閒表中，導致一個磁片塊同時出現在檔和空閒表中。兩個檔都刪除後，這個磁片塊會在空閒表中出現兩次。

檔案系統檢驗程式可以這樣來處理，先分配一個空閒塊，把磁片塊5中的內容拷貝到空閒塊中，然後把它插到其中一個檔之中。這樣，檔中的內容未改變(雖然我們幾乎可以肯定這些內容是不正確的)，而檔案系統的結構保持了一致。這一錯誤應該報告出來，以便用戶檢查。

除了檢查每個磁片塊外，檔案系統檢驗程式還檢查目錄系統。這時也要用到一張計數器表，每個計數器對應於一個檔。檢驗程式從根目錄開始，沿著目錄樹遞迴下降，檢查檔案系統中的每個目錄。對每個目錄中的檔，其i-節點對應的計數器加1(參見圖5-13中的目錄項佈局)。

當全部檢查完成後，得到一張表，對應於每個i-節點號，表中給出了指向這個i-節點的目錄數目，然後，檢驗程式把這些數位與存儲在檔i-節點中的連結數目相比較。在一致的檔案系統中，這兩個數目相吻合。但是，有可能出現兩種錯誤，i-節點中的連結數太大或太小。

如果i-節點的連結數大於指向i-節點的目錄項個數，這時，即使所有的檔都被刪除，檔連結數仍然為非0值，檔i-節點不會被刪除。這一錯誤並不嚴重，可是卻浪費了磁碟空間。我們可以把i-節點中的檔連結數設置成正確的值來改正這一錯誤。

另一種錯誤則是一種潛在的災難。如果兩個目錄項都連結到同一個檔，但其i-節點的檔連結數隻為1，如果刪除任何一個目錄項，i-節點連結數變為0。檔案系統將該i-節點標誌為"未使用"，並釋放該檔的所有磁片塊。這將導致一個目錄指向一個未使用的i-節點，而其磁片塊很可能馬上分配給其他檔。同樣，糾正方法是把i-節點中的連結數設置為目錄項的實際數目。

由於效率的原因，檢查磁片塊和檢查目錄的操作常常結合在一起(即僅需對i-節點掃描一遍)。當然還存在著一些其他的檢查方法。例如，目錄項有明確的格式：i-節點號和ASCII檔案名，如果某個i-節點號大於磁片中i-節點的總數，顯然這個目錄被破壞了。

此外，每個i-節點都包含有一個模式項。有些模式是合法的，但值得懷疑，比如0007，它不允許檔主及其所在用戶組的成員進行訪問，而其他用戶卻可以讀寫和執行此檔。這種情況下，我們有必要讓系統把其他使用者許可權高於檔主許可權這一情況報告出來。擁有1000個目錄項的目錄也很可疑。放在使用者目錄下，但為超級塊用戶所擁有，並且設置了SETUID位的文件，也可能有安全問題。稍加努力，我們可以列出一長串特殊情況，這些情況儘管合法，但卻有必要報告給用戶。

以上我們討論了保護使用者檔不因系統崩潰而破壞的問題，某些檔案系統也防止用戶自身的誤操作，如果用戶想輸入

rm \*.o

刪除所有以.o結尾的檔(編譯器生成的目的檔案)，但不幸鍵入的是

rm \* .o

(請注意，星號後面有空格)，那麼rm命令將刪除目前的目錄中的所有檔，然後報告說不能找到檔.o。在MS-DOS和一些其他系統中，刪除檔僅僅是設置目錄或i-節點的某一位元，標誌檔被刪去，在實際需要前，磁片塊並不返回到空閒表中，因此，如果使用者馬上發現了操作錯誤，他可以運行一個特定的實用程式，恢復被刪除的檔。在WINDOWS 95中，被刪除的檔放在recycled目錄下，便於使用者檢索，在這些檔從該目錄中刪除之前，並不回收其存儲空間。

5.3.5 檔案系統性能

訪問磁片要比訪問記憶體慢得多。在記憶體中讀取一個字往往需要幾十納秒，而從硬碟上讀取一個塊則需要50多個微秒，此外還需要加上10到20毫秒的尋道時間，然後再等待要讀取的磁區移到讀寫頭下麵。如果唯讀一個字，記憶體訪問要比磁片快100000倍，正因為此，許多檔案系統在設計時都儘量減少磁片訪問次數。

減少磁片訪問次數最常用技術是塊快取記憶體(block cache或者buffer cache)。在這裡，快取記憶體是一些塊，他們邏輯上屬於磁片，但基於性能的考慮而保存在記憶體中。

在管理快取記憶體時，用到了不同的演算法。一個常用演算法是：檢查所有的讀請求，看看所需的塊是否在快取記憶體中。如果在，無須訪問磁片便可進行讀操作。如果塊不在快取記憶體中，首先要把它讀到快取記憶體，再拷貝到所需的地方。之後，對該塊的讀寫請求都通過快取記憶體完成。

如果快取記憶體已滿，此時要調入新的塊，需要把原來的某一塊調出快取記憶體，如果要調出的塊自上次調入以後作過修改，則需要把它寫回磁片。這種情況與分頁非常相似，因此，第四章說過的所有的分頁演算法，例如FIFO演算法、第二次機會演算法、LRU演算法等等都適用於快取記憶體。分頁和快取記憶體的不同之處在於，快取記憶體引用相對要少，因此我們可以把所有塊按精確的LRU順序用鏈錶鏈接起來。

不幸的是，這裡有一個使人兩難之處。在這種情況下，雖然我們可以使用精確的LRU演算法，但它卻又帶來了問題，這與前一節討論過的系統崩潰和檔一致性有關。如果一個關鍵塊，比如說i-節點塊，讀入到快取記憶體並作過修改，但是沒有寫回磁片，這時，系統崩潰將導致檔案系統的不一致。如果我們把i-節點塊放在LRU鏈表的尾部，在它到達鏈首並寫回磁片前，可能需要相當長的一段時間。

此外，某些磁片塊，比如兩次間接塊，在一個短的時間內，很少被引用兩次。基於這些考慮，我們需要修改LRU方案，並注意以下兩點：

1 是否這一塊不久要重新用到?

2 是否這一塊關係到檔案系統的一致性?

就這兩個問題而言，可以把塊分為i-節點塊、間接塊、目錄塊、全資料塊、部分資料塊等幾類。最近可能不再需要的塊放在鏈表前端，這樣，他們的緩衝區很快又可使用。不久又有可能使用的塊，比如正在寫的部分資料塊，放在鏈表的尾部，以便他們在快取記憶體中保存一段較長的時間。

第二個問題與前一個相獨立。如果某塊關係到檔案系統的一致性(除資料塊之外，其他塊基本上都是這樣)，它被修改後，不管是否放在鏈表尾部，都應該立即寫回磁片。把關鍵的塊迅速寫回磁片，我們大大減少了電腦崩潰後，檔案系統被破壞的可能性。

儘管這種方法可以保證檔案系統的一致性不受破壞，我們也不希望將資料塊放在快取記憶體中很久之後才寫入磁片。設想某人用個人電腦編寫一本書。即使作者定期要求編輯器把正在編輯的檔寫回磁片，他輸入的內容只放在快取記憶體中的可能性還是非常大。這時如果系統崩潰，檔案系統的結構並不會遭到破壞，但是他這整天的工作都將毀於一旦。

這種情況即使只發生幾次，便足以讓我們感到不愉快。系統採用兩種方法來解決這一問題。UNIX系統中有一個系統調用，SYNC，使所有修改過的塊立即寫回磁片。在系統啟動時，一個稱作update的程式在後臺運行，它處於無休止的迴圈之中，不斷執行SYNC調用，在兩次調用之間睡眠30秒。這樣，即使系統崩潰，丟失的也只是30秒內的工作。

在MS-DOS中，塊被修改時，該塊同時寫回磁片。如果修改過的塊立即寫回磁片，我們稱這樣的快取記憶體為直接寫入式快取。和非直接寫入式快取相比，直接寫入式快取需要更多的磁片輸入/輸出。當一個程式欲寫滿1K的塊，每次寫一個字元時，我們可以看到這兩種方法的區別。UNIX把所有字元保存在快取記憶體中，每30秒把這個塊寫回磁片，或者當這一塊從快取記憶體中刪除時，寫回磁片。在MS-DOS中，每寫入一個字元，訪問一次磁片。當然，大多數程式都有內部緩衝，因此通常情況下，在執行系統調用WRITE時，並不是逐個字元寫入的，而是以行或者更大的單位寫入。

這兩種不同的快取記憶體策略的結果是：在UNIX系統中，未調用SYNC就移走(軟)磁片，往往會導致資料丟失，有時還會破壞檔案系統。而在MS-DOS中，則不會出現這種情況。之所以選擇不同的策略，是因為UNIX是在這樣的環境下開發出來的：所有的磁片都是硬碟，不可移動。而MS-DOS則是從軟碟世界發展起來的。隨著硬碟成為標準，甚至在小型微機中，UNIX的快取記憶體方案由於其更高的效率，必將會成為我們當然的選擇。

快取記憶體並不是改善檔案系統性能的唯一方法，另一種重要技術是把那些有可能順序存取的塊放在一起，最好是同一個柱面上，從而減少磁片臂的移動次數。當寫一個輸出檔時，若該檔需要新的塊，檔案系統就為它分配一塊。如果空閒塊用點陣圖來記錄，並且整個點陣圖放在主存中，我們很容易選擇與前一塊最近的空閒塊。如果採用空閒鏈表，並且鏈表有一部分內容存放在磁片上，找到這樣的空閒塊就要困難得多了。

然而，即使是採用空閒鏈表，也可以使用塊群技術。這裡用到一個小技巧，即使用連續塊群，而不是塊，來記錄磁片存儲區。如果每道由64個磁區組成，每個磁區有512個位元組，系統可能使用1K大小的塊(2個磁區)，但卻按每2個塊(4個磁區)為一個單位來分配磁片存儲區。這和2K的塊並不一樣，它在快取記憶體中依然使用1K大小的塊，磁片與記憶體資料傳送也是以1K為單位進行，但在一個不是很空閒的系統上順序讀取檔，其尋道次數可以減少一半，從而大大地改善檔案系統的性能。

若考慮到旋轉定位，我們可以得到這種方案的一個變體。當分配塊時，系統儘量把檔的連續塊存放在同一柱面上，但加以交叉以獲取最大吞吐率。這樣，如果磁片的旋轉延遲為16.67毫秒，並且使用者進程需花4毫秒來請求並讀取一塊資料，則每個資料塊的位置應距離前一塊至少四分之一磁軌。

在使用i-節點或者與i-節點等價結構的系統中，另一個性能瓶頸在於，即使讀取一個很短的檔也要訪問兩次磁片：一次是讀取i-節點，另一次是讀取檔塊。通常情況下，i-節點的放置如圖5-19(a)所示，圖中，所有i-節點都靠近磁片頭部，因此i-節點和相應塊之間的平均距離是柱面總數的一半，這需要很長的尋道延遲。

圖5-19 (a)i-節點放在磁片開始位置。 (b)磁片分為柱面組，每組有自己的塊和i-節點。

一個簡單的改進方法是把i-節點放在磁片中部。這時，在i-節點和第一塊之間的平均尋道時間減為原來的一半。另一種想法是：把磁片分成多個柱面組，每個柱面組有自己的i-節點、資料塊和空閒表，參見圖5-19(b)(Mckusick et al.，1984)。在創建檔時，可以選取任一個i-節點，然而分配塊時，在該i-節點所在的柱面組上進行查找，如果該柱面組中沒有空閒的資料塊，就查找與之相鄰的柱面組。

5.3.6 日誌結構的檔案系統

技術的改進使得當前的檔案系統面臨著很大壓力。特別地，CPU速度越來越快，磁片容量不斷增大，成本不斷降低(但是存取速度沒有很大的提高)，記憶體容量呈指數增長，然而有一個參數沒有得到迅速改進，這就是磁片尋道時間，所有這些因素結合在一起，表明在許多檔案系統中正出現著一個瓶頸。Berkely學院設計了一種全新的檔案系統，即日誌結構的檔案系統(log-structured file system)，試圖減輕這個問題。本節我們簡要地講述一下LFS的工作原理，要瞭解更詳細的內容，請參見(Rosenblum和Ousterout，1991)。

促成LFS設計的想法是：CPU越來越快，RAM記憶體越來越大，磁碟快取的容量迅速增加。因此，無需訪問磁片，從檔案系統的快取記憶體中就可能滿足所有讀請求。將來大多數磁片訪問是寫操作。某些檔案系統使用的預讀機制，即把資料塊在實際需要前調入記憶體，對檔案系統性能的改進不再那麼重要了。

更糟糕的是，在大多數檔案系統中，寫操作都是以小塊為單位進行的，效率很差，因為在50微秒的寫磁片之前，需要有10毫秒的尋道延遲和6毫秒的旋轉延遲。由於後者的存在，使得磁片效率還不足1%。

要看看是什麼引起小塊寫，我們考慮在UNIX系統中創建一個新檔。為了寫這個檔，必須對目錄的i-節點、目錄塊、檔的i-節點和檔本身執行寫操作。儘管這些寫操作可以延遲進行，但是延遲寫在系統崩潰時，很容易使檔案系統產生嚴重的一致性問題。因此，i-節點一般都是立即寫入的。

基於上述推理，LFS的設計人員決定重新設計UNIX的檔案系統，他們希望即使在有大量小塊隨機寫的情況下，也能獲得磁片的全部頻寬。基本思想是把整個磁片作為日誌。所有寫操作都存放在記憶體的緩衝區中，並定期地收集到一個單獨的段中，作為日誌尾部的相鄰段寫回磁片。因此，每個段都含有i-節點、目錄塊、資料塊等，並且是他們的混合體。在每段起始位置還有一個摘要，給出了該段中的內容。如果段的平均長度大約為1MB，那麼幾乎所有的磁片頻寬都能利用。

在這一設計中，i-節點依然存在，並且和UNIX中的i-節點具有相同結構。但是這些i-節點並不放在磁片的固定位置，而是分散在日誌之中。一旦找到i-節點，可以用通常的方法找到相應塊。毫無疑問，現在查找一個i-節點要困難得多。我們不能象UNIX那樣，根據i-節點號通過簡單計算來得到i-節點的位置。為了查找i-節點，需要維護一張i-節點映照表，它以i-節點號為下標，其中第i項指向磁片上第i個i-節點。這張映照表存放在磁片中，但它同樣使用緩存機構，所以在大多時候，最常用的部分將保存在記憶體中。

我們小結一下LFS的工作方式。所有寫的資料開始時都存放在緩衝區中，並且定期地把這些緩衝區中的資料以一個段的形式寫到磁片中，放在日誌的尾部。打開一個檔首先要在i-節點映照表中查找該檔的i-節點，一旦找到了i-節點， 也就知道了相應塊的位址，所有的塊也存放在段中，即日誌的某個地方。

如果磁片容量無限大，上面的描述無疑非常完美。但是，真正的磁片都有有限的容量，最終日誌將占滿整個磁片，這時新段不能被寫到日誌中。幸好，許多現有段都包含一些不再使用的塊。例如，如果檔被重寫，其i-節點將指向新塊，而原來的塊仍然佔有以前段的空間。

為了解決這兩個問題，LFS中有一個清理工(cleaner)執行緒迴圈地流覽和壓縮磁片。它首先讀取日誌中的第一個段的摘要，找出其中的i-節點和檔。接著查找當前的i-節點映照表，檢查i-節點是否還在使用以及檔塊是否還在使用。若沒有使用，這些資訊將被丟棄。還在使用的i-節點和塊讀入記憶體中，以便寫到下一個段中。原來的段標記為空閒，以便日誌用於存放新資料。這樣，清理工執行緒沿日誌向前移動，刪除舊段，把有效資料讀入記憶體，寫入下一新段。這樣，磁片是一個迴圈的緩衝區，寫執行緒不斷地在前面添加新段，而清理工執行緒不斷地從後面刪除舊段。

這裡，簿記工作比較麻煩，當某檔塊寫回到一個新段時，先要在日誌中找到該檔的i-節點，對其進行修改，然後把它放在記憶體中，以便寫回下一段。i-節點映照也必須進行相應修改以指向新的拷貝。但是總的來說，這種管理是可取的，性能研究表明這種複雜性是值得的。前面提到的論文測試顯示，在進行大量小塊寫的情況下，LFS的性能明顯超過了UNIX。而在讀數據和大塊寫資料時，其性能也同於，甚至要好於UNIX。

5.4 安全性

檔案系統往往包含有使用者非常寶貴的資訊。因此，如何保護這些資訊不被未授權使用是所有檔案系統的一個主要內容。下面幾節中，我們討論與安全和保護有關的一些問題。這些問題既適用於分時系統，也適用於通過局域網連接到共用伺服器的個人電腦網路。

5.4.1 安全環境

安全和保護這兩個術語經常交替使用。然而，我們有必要區分兩類不同的問題，其中一類是確保未被授權使用者無法讀取或修改某些檔，它包括技術、管理、法律和政治等方面。而另一類則指用於提供安全的特定作業系統機制。為避免混淆，我們用安全性(security)指這些綜合的問題，用保護機制(protection machanism)來指用於保護電腦資訊的特定作業系統機制。然而，他們之間的界線並不非常明確。我們首先來看看安全性。在這章的後面部分我們再討論保護機制。

安全性有許多方面，其中較重要的兩個方面是資料丟失和入侵者。造成資料丟失的原因往往是：

1. 災禍：火災、洪水、地震、戰爭、暴亂或者是老鼠咬壞磁帶或軟碟等等。

2. 硬體或軟體故障：CPU誤操作、不可讀取的磁片或磁帶、遠端通訊故障、程式故障等等。

3. 人的失誤：不正確的資料登錄、磁帶或磁片安裝故障、程式運行錯誤、磁帶或磁片丟失或者一些其他的錯誤。

這些資料丟失問題大多數可以通過保存足夠的備份而解決，最好是將備份資料放在與來源資料相隔較遠的地方。

一個有趣的話題是如何對付入侵者。入侵者可以分為兩類：消極的入侵者只想讀取未授權的文件。積極的入侵者則懷有惡意，他們在未獲授權的情況下試圖修改檔資料。在設計一個安全的系統時，必須記住要防止哪一類入侵者。下面是較常見的幾類入侵：

1. 非技術人員的偶然窺視。大多數人在自己的工作臺上都擺放著一台分時系統終端或者聯網的個人電腦。不管他們的本性如何，如果我們不設置障礙的話，某些人會讀取別人的電子郵件或其他檔。在大多數UNIX系統中，所有檔的缺省屬性均為可讀的。

2. 入侵者的窺探。學生、系統程式師、操作人員和其他一些技術人員常常以突破局部電腦系統的安全性為個人能力的一次挑戰。他們往往都技術嫺熟，並且願意花大量的時間來從事這一努力。

3. 明確的偷竊企圖。一些銀行程式師試圖突入銀行系統，從中竊取金錢。他們的陰謀五花八門，包括刪除利息據為己有、盜用多年未使用的帳戶、甚至敲詐勒索("趕快給我一筆錢，否則我就刪除銀行所有記錄。")

4. 商業或軍事間諜活動。間諜活動指的是由競爭對手或者外國政府資助的、正規的、高投入的活動，主要是竊取對方的程式、商業秘密、專利、技術、電路設計和市場計畫等等。他們常常竊聽對方的通道消息甚至樹起天線截獲對方電腦的電磁輻射。

我們應該清楚：防止敵對的外國政府竊取軍事秘密和防止學生在系統中插入一條有趣的"今日要聞"是完全不同的兩碼事。人們花在安全和保護上的勞動顯然取決於他們所設想的對手是誰。

安全問題的另一方面是隱私權，即保護個人資訊不被濫用。這牽涉到許多法律和道德上的問題：政府是否有權為了捕捉X騙子而搜集個人檔案呢？這裡X可以是"財富"或者是"稅收"，這主要取決於你的政策了。員警是否可以對某些人進行搜查來阻止有組織的犯罪活動呢？雇主和保險公司是否有這種權力呢？當這些權力和個人權利相悖時，會出現什麼情況呢？所有這些問題都非常重要，但他們都超出了本書的範圍。

5.4.2 著名的安全缺陷

正如運輸界有泰坦尼克等事故一樣，電腦安全專家往往也容易忽略某些細節。在本節中，我們介紹在三種不同的作業系統：UNIX，TENEX和OS/360中發生的一些有趣的安全問題。

UNIX中實用程式lpr用於列印一個檔，在列印完成後，允許用戶選擇是否刪除該檔。在UNIX的早期版本中，任何人都可以使用lpr進行列印，從而他們都可以從系統中刪除口令檔。

突破UNIX的另一種方法是將目前的目錄下的core檔連結到口令檔上，然後，入侵者促使對某個SETUID程式進行core轉儲。這樣，使用者可以用一個包含自由選取的字串檔(即命令參數)來代替口令文件。

在UNIX中，另一個缺陷與命令

mkdir foo

有關。mkdir是一個根使用者擁有的SETUID程式，它首先使用系統調用MKNOD為目錄foo創建一個i-節點，然後把foo的擁有者從有效uid(即根)改為實際uid(用戶的uid)。當系統很慢時，使用者有可能迅速地刪除foo目錄的i-節點，而以foo為名字建立一個到口令檔上的連結。如果時機選擇得好，上述操作可以在mkdir執行MKNOD之後，執行CHOWN之前完成。這樣，執行CHOWN後，用戶就變為口令檔的擁有者。在shell腳本中放入一些必要的命令，用戶可以反復運行這些命令，直至目的得逞。

TENEX作業系統以前廣泛運行在DEC-10機器上，如今它不再使用了，但是由於以下的設計缺陷，它的"名聲"將永遠留在電腦的安全領域。TENEX支持分頁，為了使用戶能監控程序的行為，他們可以要求系統在頁錯誤時，調用自己的函數進行相應處理。

TENEX也使用口令來保護檔，要存取一個檔。程式中必須給出正確的口令。作業系統在檢測口令時，每次檢查一個字元，一旦發現口令錯誤，立即停止執行。要闖入TENEX，入侵者可以精心地安排口令，如圖5-20(a)所示，把第一個字元放在一頁的末尾，而剩下部分放在下一頁的開頭。

圖5-20 TENEX口令問題。

接下來，要確保第二頁不在記憶體之中。通過多次引用其他頁面，我們可以保證第二頁被調出記憶體。現在，程式可以使用這個精心排列的口令試著打開受害者的檔。如果真實口令中的第一個字元不是A，系統立即停止檢查並報告"口令非法"。可是，如果口令確實是以A開始，系統繼續讀下面的字元，得到頁錯誤，這一錯誤也會報告給入侵者。

如果口令不是以A開始，入侵者修改口令如圖5-20(b)所示，並重複上述整個過程，看看口令是否以B開始。要確定口令的第一個字元，最多只需嘗試128次，即檢查整個ASCII字元集。

假設第一個字元為F，那麼圖5－20(c)所示的記憶體佈局可使入侵者檢測FA、FB等形式的字串。採用這種方法，猜出一個n個字元的ASCII口令最多僅需要128n次，而不是128n次。

我們想介紹的最後一個缺陷與OS/360有關。下面的描述略微簡單，但保持了缺陷的本質。在該系統中，我們可以啟動磁帶讀操作，並在磁帶驅動器將資料傳送到使用者空間時，繼續執行運算。這裡我們用到一個小技巧，精心地啟動讀磁帶操作，然後執行一個要求使用者輸入資料結構的系統調用，例如，檔及其口令。

作業系統首先檢查給定檔的口令正確無誤，然後返回，再次讀取檔案名以訪問之(它原本可以將檔案名在內部保存起來，但是卻沒有這樣做)。可惜，就在系統第二次讀取檔案名前，檔案名被磁帶驅動器重寫了。這樣系統讀到的是一個新檔，而這個檔無需用戶輸入口令。掌握時機可能需要動些腦筋，但這並不困難。事實上，電腦所擅長的無疑便是反復地進行同一操作。

除了這些例子之外，近年來還出現了許多其他的安全問題和攻擊方法。人們經常會聽到特洛伊木馬(Trojan horse)，特洛伊木馬是一個貌似無辜並廣泛流傳的程式，它執行人們意料不到的操作，例如竊取使用者資料並用電子郵件傳送到某台遠距離的電腦上。

在如今工作無保障的年代裡，另一個安全問題是邏輯炸彈(logic bomb)。這是公司(當前雇傭)的程式師編寫的一段代碼，程式師把它悄悄地插入到生產性的作業系統中。只要程式師每天輸入口令，則一切正常。如果他突然被解雇，那麼第二天，這個邏輯炸彈無法得到口令，它就將爆炸。

一旦爆炸，系統可能清空磁片、隨機地刪除檔、對關鍵程式作難以察覺的修改或者加密一些必需的檔等等。其後，公司不得不在下列兩者之間作出艱難的選擇：報警(即使過了幾個月，也不能保證定罪)或者向勒索者讓步，以天文數字的薪水重新雇傭他為"顧問"來糾正這個問題(並且希望他在糾正時不要再放置一個新的邏輯炸彈)。

或許，歷史上最大的電腦安全犯罪發生在1988年11月2日。當時康納爾大學的研究生，Robert Tappan Morris，把一個蠕蟲程式放在Internet上，最終使得全世界成千上萬台機器陷於崩潰。

這個蠕蟲由兩個程式組成，即引導程式和真蠕蟲程式。引導程式是一個名為l1.c的99行程式，在所攻擊的系統中編譯、執行。運行時，它連接到它原來所在的機器上，然後下載真蠕蟲程式並執行。在設法將自己隱藏起來後，蠕蟲接著查閱這台主機的路由表，看看哪些機器連接到這台主機，並試圖把引導程式傳播到那些機器。

進入某台機器後，蠕蟲試著猜出用戶的口令。Morris並沒有化太多精力做這件事，他的父親是美國國家安全局---美國政府絕密代碼破譯機構---的安全專家，十年前在貝爾實驗室工作時，曾和Ken Thompson合作寫過一篇關於安全問題的經典文章(Morris和Thompson，1979)。Morris只需向他父親要一份這篇文章的副本，其中每個破譯出的口令允許蠕蟲在使用這些口令的用戶擁有帳號的任何機器上登錄。

Morris被捕是因為，有一天，他的一個朋友與紐約時報的電腦報導員John Morkoff交談，並試圖讓Morkoff相信這一事件純屬偶然，蠕蟲程式毫無惡意以及程式編制者非常難過等等。第二天，這一消息刊登在報紙的顯著位置，甚至超過了三天后的總統選舉。聯邦法院對Morris進行了審判，對其處以10000美元的罰款，3年緩行以及400小時的公眾服務。訴訟費可能高達15,000美元。

這一判罰引起了很大的爭論，電腦界的許多人認為Morris是一個十分聰明的研究生，他只不過搞了一次惡作劇，蠕蟲中沒有證據證明他試圖竊取或破壞如何東西。而另一些人則認定Morris是一個罪犯，應該被判處監禁。

這一事件導致了電腦緊急事件反應組的建立。CERT為人們提供了一個報告非法入侵的場所，其中，有一群專家專門分析安全問題以及設計糾正方法。這無疑是一次大的進步，但仍然有不足之處。CERT收集有關系統攻擊方法及彌補對策的資訊，它把這些資訊通過Internet傳送給成千上萬個系統管理員，這就意味著某些壞蛋可以獲取這些資訊，在漏洞補好之前幾個小時(甚至幾天)大搞破壞活動。

5.4.3 一般的安全性攻擊

上述的缺陷都早已修復了。但是，一般的作業系統還會有其他問題。測試系統安全性的常用方法是雇傭一群專家組成猛虎組(tiger teams)或者叫滲透組(penetration teams)，看看他們是否能夠闖入系統之中。Hebbard et al.(1980)和他的研究生做了同樣的工作。許多年來，這些滲透小組發現在許多方面系統容易出現安全問題。下面我們列出一些常常奏效的攻擊方法。當你在設計一個系統時，需要防止類似的攻擊：

1. 請求記憶體頁、磁碟空間和磁帶並讀取其內容。許多系統在分配時並不刪除以前的內容，而其中往往有很多其他使用者寫入的重要資訊。

2. 嘗試非法的系統調用、或者是帶非法參數的合法系統調用、或者是帶有合法而不合適的參數的合法系統調用。許多系統很容易引起混淆。

3. 在登錄過程中途鍵入DEL、RUBOUT或BREAK。有些系統中，口令檢查程式被終止，認為登錄成功。

4. 試圖修改保存在用戶空間內的、複雜的作業系統結構(如果有的話)。在某些系統(特別是一些大型機)中，在打開檔時，程式創建一個大的資料結構，其中包含有檔案名和許多其他參數，並把它傳給系統。在讀寫檔時，系統有時修改這些結構。修改這些域可能造成安全性的嚴重破壞。

5. 寫一段程式來哄騙使用者，程式在螢幕上列印"login："。許多用戶在登錄時鍵入的用戶名和口令被程式詳細地記錄下來。

6. 仔細查閱手冊中關於"請不要做XXX"的說明，然後試驗各種變通方法。

7. 說服系統程式師修改系統，使得在使用者以你的用戶名登錄時，跳過某些重要的安全性檢查，這種攻擊方法稱為暗門(trapdoor)。

8. 在其他方法都無效時，滲透者有可能找到計算中心主任的秘書，給他一筆數量可觀的賄賂，而這個秘書或許有權訪問各種重要的資訊，但薪水很低，可千萬不要低估了這些職員帶來的問題。

這些以及其他一些攻擊方法在Linde(1975)中都有討論。

病毒

一類特殊的攻擊是電腦病毒。病毒已經成為許多電腦使用者的主要問題。病毒(virus)是一個程式段，它附在合法的程式中，並企圖感染其他程式。和蠕蟲不同的是，病毒附在現有程式中，而蠕蟲本身是一個完整的程式。病毒和蠕蟲都試圖廣泛傳播，他們都可能造成嚴重的後果。

病毒往往是這樣流行的：病毒編制者首先寫一個有用的程式，比如是MS-DOS下的一個遊戲，這個程式中潛伏著病毒。然後，病毒編制者把遊戲上載到公告牌系統中，或者以軟碟的形式免費或廉價地提供給用戶。接著，這個程式被大肆宣揚，許多人下載並使用它。編制病毒並非一件容易的事，因此這些病毒編寫者往往是一些很聰明的程式師，他們編寫的遊戲或者其他程式都非常優秀。

病毒程式啟動後，它立即檢查硬碟上的所有的二進位檔案，檢查他們是否已被感染。如果找到一個未被感染的檔，病毒代碼被加在檔案結尾部，並且把第一條指令換成跳躍陳述式，轉而執行病毒代碼。在病毒代碼執行完畢後，接著執行原來程式的第一條指令，然後跳到第二條指令處繼續執行。這樣，每次被感染的程式運行時，它總是感染更多的程式。

除了僅僅感染程式之外，病毒程式還可以做一些其他事情。例如刪除、修改或者加密檔。有的程式甚至在螢幕上顯示一條勒索資訊，要該使用者送500美元現金到巴拿馬的某個郵政局信箱中，否則他就將遭受資料或硬體的巨大損失。

病毒程式還可能感染硬碟的引導區，導致電腦無法啟動，這樣的病毒可能會要求使用者輸入口令，而病毒的編制者往往廉價地提供這種口令。

和刪除病毒相比，預防病毒要容易得多。最安全的辦法是從可信賴的商店裡購買精裝軟體。從公告牌系統中下載軟體或者是用軟碟拷貝盜版軟體無疑是自找麻煩。一些商用殺毒軟體也有銷售，但是，其中大多數只能查找已知的病毒。

更一般的方法是重新格式化整個硬碟，包括引導區，再安裝所有可靠軟體，對每個檔計算一個校驗碼。採用何種演算法無關緊要，只是，校驗碼要有足夠多的位(至少32位)。把這些(檔，校驗碼)對存放在一個安全地方。你可以存放在軟碟中或者加密後存放到硬碟上。這樣當系統啟動時，計算所有檔的校驗碼並與原來的校驗碼相比較。任何檔，如果兩者的比較結果不一致，都值得懷疑。儘管這種辦法無法防止檔被病毒感染，至少它可以較早地發現病毒。

如果使目錄中的二進位檔案對普通使用者不可寫，則感染這些檔要困難得多。這種技術使得病毒難以修改二進位檔案。雖然它可以用在UNIX中，但對MS-DOS卻不適用，因為後者的目錄根本不可能設置為不可寫。

5.4.4 安全性的設計原則

大多數情況下，病毒發生在桌面系統中。在更大的系統中，會出現其他問題，需要一些其他方法來解決。Saltzer和Schroeder(1975)列出了幾個用於指導安全系統設計的一般原則，下面簡要地介紹他們的想法(基於MULTICS的經驗)：

首先，系統設計必須公開，假設入侵者不知道系統的工作方式無疑是自欺欺人。

其次，缺省屬性應該是不可訪問。合法訪問被拒絕的錯誤，會比未授權訪問被容許出現的錯誤更快地報告出來。

第三，檢查當前許可權。 系統不應該先檢查許可權，確定該訪問合法，然後把這個資訊保留以備後用。許多系統是在打開檔時檢查許可權，而不是在以後。 這意味著一個用戶打開檔幾個星期後，依然有權存取該檔，即使檔主早就修改了檔的許可權。

第四，給每個進程賦予一個最小的可能許可權。如果一個編輯器只有權存取它所編輯的檔(在編輯器啟動時指定)，這時，即使它帶有特洛伊木馬病毒，也不會造成很大的損害。這個原則蘊含著一個小顆粒度的保護方案。我們後面還會談到這樣的保護方案。

第五，保護機制要力求簡單一致、嵌入到系統的底層。要想在不安全的系統上改進安全性幾乎是不可能的。同正確性一樣，安全性也不是一個附加的特徵。

第六，採取的方案必須可以接受。如果用戶覺得保護其檔太麻煩，他們根本就不會去保護，然而，在出現問題時他們會不停地抱怨系統設計者，這時對他們說："這是你自己的問題"，用戶一般是不會接受的。

5.4.5 用戶驗證

許多保護方案都假定系統知道使用者的身份。當用戶登錄時，檢驗其身份的問題叫做用戶認證(user authentication)。大多數驗證方法都基於以下的檢驗，即用戶知道些什麼、用戶擁有什麼或者用戶是什麼等等。

口令

使用最廣泛的用戶認證形式是用戶輸入口令。口令保護容易理解，也容易實現。在UNIX中口令是這樣工作的：登錄程式要求使用者輸入用戶名和口令，這時口令立即加密，然後登錄程式讀取口令檔，通常是許多行的ASCII檔，直至找到包含用戶登錄名的那一行，如果這行(加密)的口令與計算得到的加密口令相符，則允許登錄，否則不予登錄。

口令認證也很容易被突破。人們經常可以聽到一群高中生，甚至是初中生，借助於他們的家庭電腦，闖入到某大公司或者政府機構的絕密系統中去。一般來說，他們總是想辦法猜出用戶名和口令組合。

儘管近年來在口令安全方面進行著越來越多的研究，然而經典的工作還是Morris和Thampson(1979)在UNIX系統中所做的。他們搜集並編成了一張可能的口令表，其中包括姓名、街名、城市名、中型字典中的單詞(包括反過來拼寫的單詞)、門牌號碼以及很短的隨機字元組成的字串。然後他們使用已有的加密演算法對這些可能的密碼進行加密，檢查任何加密口令是否與他們的表中某一項相匹配。超過86%的口令都在他們的表中出現。

如果口令由7個字元組成，每個字元隨機取自95個可列印的ASCII字元，那麼我們需要搜索957種組合，約等於7\*1013。

即使以每秒加密1000項的速度，也需要2000多年才能建立起一張這樣的表。何況，這張表要耗費20億盤磁帶。甚至要求口令包含一個小寫字母、一個大寫字母、一個特殊字元以及至少7到8個字元，也是對使用者自由選擇口令的一個極大地改進。

儘管要求每個用戶都合理地挑選口令在實際上是做不到的，Morris和Thompson還是提出了一種使得他們自己的攻擊方法(事先對大量的口令加密)失效的方法。他們的想法是給每個口令賦予一個n位亂數字。當口令改變時，亂數隨之改變。亂數以明碼的形式存放在口令檔中，任何人都可以讀取。這時，我們不單單加密口令，而是把口令和亂數連接起來一起加密，將這個加密的結果存儲在口令檔中。

設想一個入侵者試圖建立一張可能的口令表，對他們進行加密，把這個結果存放在檔f中，以便從中查找加密口令。如果入侵者猜測到口令可能是Marilyn，這時他不僅僅對Marilyn加密，把結果放在檔f中。事實上，他必須加密2n個字串，如Marilyn0000、Marilyn0001、Marilyn0002等等，並把他們全部放到f中。這種方法把檔f的長度增大了2n倍。UNIX也使用這種方法，其n值為12。這種方法也稱為鹽漬口令檔。一些版本的UNIX使得口令檔本身不可讀取，但提供了一個程式來查找，這樣就大大地減慢了破譯速度。

雖然這種方法可以防止入侵者預先計算一張大的加密口令表，它卻無法保護用戶David使用David作口令的情況。一種鼓勵使用者挑選好的口令的方法是讓電腦提供一些建議。有些電腦程式生成容易拼讀的、無意義的單詞作為口令，例如fotally、garbungy或bipitty(最好其中有一些大寫字元和一些特殊字元)。

有的電腦則要求使用者定期改變其口令，從而減少口令洩漏可能造成的損失。最極端的方法是使用一次性口令。這種情況下，使用者備有一本口令書，其中記載著一長串口令，每次登錄時都使用書中下一個口令。如果入侵者偶然破譯出口令，也沒有什麼好處，用戶下一次就會使用不同的口令。不過使用者需要好好保管他的口令書。

毫無疑問，在使用者鍵入口令時，電腦不應該把敲入的字元顯示出來，以防旁邊有人窺視。此外，口令決不要以未加密的形式存放，甚至是電腦中心的管理部門也不應該有未加密的口令拷貝。口令未加密就存放起來，簡直是自找麻煩。

口令思想的一個變通形式是給每個用戶提供一長串的問題，答案以加密形式存放起來。這些問題應該儘量為用戶所熟悉，使用者根本無需把他們記錄下來。下面是一些典型的問題：

1. 誰是Marjolein's的姐姐?

2. 你的小學在哪條街上?

3. Woroboff小姐教哪門課程?

在登錄時，電腦隨機地選擇一條詢問用戶，並檢查用戶的回答是否正確。

口令思想的另一個變體是查問-回答。使用查問-回答時，用戶在註冊時選擇好一個演算法，比如x2。當使用者登錄時，電腦顯示某個參數，例如7，這時用戶需鍵入49。這種演算法可以經常變動，早晚不同、每天不同、並且在不同的終端上也不相同。

物理鑒定

另外一個完全不同的認證方法是檢查用戶是否有某些"證件"，通常是含有磁條的塑膠卡。將卡片插入終端中，系統可以查出卡片所有者。這種方法可以和口令一起使用。因此用戶要成功登錄，必須(1)擁有卡片，並且(2)知道口令。自動提款機就是這樣工作的。

還有一種方法是測量那些難以偽造的物理特徵。例如，終端上的指紋或者聲波紋讀取機可以認證用戶身份。(如果電腦不是將給出的指紋與整個資料庫相比較，而是根據使用者的名字進行查找，則整個過程要快得多。)直接視覺辨認現在還不可行，但終究有一天會實現。

另一種技術是簽名分析。使用者使用與終端相連的特製筆簽名後，電腦與線上存儲的已知樣本進行比較。更好的方法是不必比較簽名，而是比較簽名時筆的移動情況，一個優秀的模仿者或許可以模仿你的簽名，但是，至於你在在簽名時行筆的確切順序，他就不清楚了。

指長分析也非常實際。當採用這種方法時，每個終端都有一個如圖5-21的設備。使用者把手插入到這個設備中，他的各個手指的長度被記錄下來，並與資料庫中存放的資料比較。

圖5-21 測量指長的設備。

我們還可以舉出更多的例子。但是，下面兩個例子說明了一個關鍵問題。貓和其他的動物會在他們的邊界周圍撒上一泡尿，這樣他們可以很容易彼此區分。假設有人提供能當場進行尿液分析的小設備，然後，每個終端都配置這樣一個設備，旁邊寫著："登錄時，請在此排放尿液"。這或許是一個絕對不可突破的系統，但是使用者顯然難以接受。

下面的方案同樣也很難讓用戶接納：系統由圖釘和小型攝譜儀組成，登錄時，用戶將拇指按在圖釘上，提取一滴血液讓攝譜儀分析。至關重要的是，任何一種認證方案都必須為用戶所接受。指長測量可能不會引起任何問題，但是，即使是一些非侵入性方法，比如線上存儲指紋，許多人也不願接受。

對策

往往在入侵者闖入並製造極大的損壞之後，電腦裝置才會對安全性引起嚴重關注，他們常常採取很多方法來增加未授權訪問的難度。例如，每個用戶只允許在指定的某一台終端上登錄，並且只能在一周的某幾天或者一天中的某幾個小時中登錄。

撥號電話可按如下方式工作：任何人都可以撥號和登錄，但一旦成功登錄，系統立即中止連接，並且以某個協商好的號碼回呼用戶。使用這一方法，入侵者不能從隨意一根電話線上進入系統。要進入系統，他只能通過使用者的(家庭)電話。並且，不論是否回呼，系統應該至少用10秒鐘來檢查用戶在撥號線上鍵入的口令。當幾次連續的登錄失敗後，這個時間還要增加，以減少入侵者非法闖入的可能性。在三次登錄失敗後，電話線被中斷10分鐘，並通知安全人員。

所有登錄資訊都應該記錄下來。當使用者登錄時，系統告知他上次登錄的時間和終端，以便他檢測出可能的非法入侵。

我們還可以設置陷阱來捕獲入侵者，例如採用帶簡單口令的特殊登錄名(如登錄名guest，口令guest)。當發現任何人以這一名字登錄時，立即通知安全專家。其他的陷阱可以是容易發現的作業系統故障或類似的東西，他們是專門為捕獲入侵者而設計的。Stoll就曾寫過一篇有關陷阱的有趣報導(1989)，利用這陷阱，他捕獲了一個潛入某大學電腦系統中竊取軍事秘密的間諜。

5.5 保護機制

在前面的幾節中，我們看過許多潛在的問題。其中有些是技術性的，有些是非技術性的。下面幾節，我們主要討論作業系統中保護檔和其他資訊的一些技術，所有這些技術把策略(誰的資料需要保護，以及禁止誰訪問)和機制(如何實現保護)明確地區分開來。策略和機制的分開在(Levin et al.，1975)中討論。這裡，我們的重點主要放在機制，而不是策略上。要瞭解更詳細的材料，請參看(Sandhu，1993)。

某些系統使用一個叫做引用監視器的程式來實現保護。每次訪問被保護資源時，系統請求引用監視器檢查訪問是否合法。 引用監視器查找策略表，作出相應的決定。下面我們介紹引用監視器的運行環境。

5.5.1 保護域

電腦系統中包含有許多有待保護的"物件"，這些物件有可能是硬體(例如CPU、記憶體段、磁碟機或者印表機)，也有可能是軟體(例如進程、檔、資料庫或者信號)。

每個物件都有一個唯一的名字，用戶通過這個名字來引用物件。並且，物件還有一個有限的、允許進程執行的操作集。例如對檔有READ、WRITE操作，對信號可進行UP、DOWN操作等等。

很明顯，我們需要某種方法禁止未授權進程訪問物件。此外，保護機制還應該可以限制進程只執行合法操作的子集。比如，進程A有權讀取檔F，但不能寫F。

為了討論不同的保護機制，我們有必要引入域的概念。域是(物件，許可權)對的集合，每個對標明了一個物件和一個可執行操作的子集，這裡許可權表示允許執行的操作。

圖5-22顯示了三個域，並給出了每個域的物件，以及對這些物件允許的操作[讀、寫、執行]。注意，Printer1同時屬於兩個域。雖然我們在圖中沒有畫出，但是同一物件可出現在多個域中，並且在不同域中的許可權也可以不同。

圖5-22 三個保護域。

無論何時，進程都運行在某個保護域中。換句話說，它可以訪問某些物件，對每個物件有一定的許可權。進程在執行時，可以從一個域切換到另一個域，切換的規則完全獨立於系統。

為了使保護域的思想更加具體，我們來看看UNIX。在UNIX中，進程的域用uid和gid來定義，給出一個(uid，gid)對，可以建立一張完整的表，列出所有可以訪問的物件(檔，包括用設備檔表示的I/O設備等)以及是否可以讀、寫、執行這些物件。有相同(uid，gid)對的兩個進程有完全相同的訪問對象集。具有不同(uid，gid)對的進程可能訪問不同的物件集，儘管多數情況下，這些物件集是相互重疊的。

此外，UNIX中每個進程有兩個部分：使用者部分和內核部分。當進程執行系統調用時，從使用者部分切換到內核部分。內核部分和使用者部分有不同的訪問物件集。例如，內核部分可以存取實體記憶體中的所有頁面，進而訪問整個磁片以及其他保護資源。因此，系統調用導致域切換。

當進程對設置了SETUID或SETGID位的檔執行EXEC操作時，它獲得新的有效uid或gid。對於不同的(uid，gid)對，有不同的檔集和有效操作。運行一個有SETUID或SETGID的進程也是域切換，因為這些可用的許可權並不相同。

這裡會遇到一個重要問題：系統如何記錄某個物件屬於哪個域呢？至少，你可以想像用一個矩陣來表示，矩陣的行表示域，列表示物件。圖5-22對應的矩陣如圖5-23所示，給定該矩陣及當前域號，系統可以給出是否能夠按特定方式從某指定域中訪問物件。

圖5-23 一個保護矩陣。

如果意識到域本身也是物件，我們很容易用ENTER操作把域切換包含在矩陣模型中。圖5-24重新顯示了圖5-23，只是增加了三個域作為對象。域1中的進程可以切換到域2，然而，一旦切換後，它不能重新回到域1。在UNIX中，這種情況類比執行一個SETUID程式。在這個例子中，其他的域切換都不允許。

圖5-24 保護矩陣把域作為對象。

5.5.2 存取控制表

實際上，我們很少存儲象圖5-24中那樣的矩陣，這個矩陣非常大並且有很多空項。大多數域都只存取很少的物件，因此存儲一個大而空的矩陣浪費了大量的磁碟空間。可是，我們可以使用兩種方法：按行或列來存儲矩陣，而且只存儲非空元素。這兩種方法有很大的不同。本節中，我們介紹按列存儲。下一節，我們將討論如何按行存儲這個矩陣。

在第一種存儲技術中，每個物件被賦予一張排序的列表，其中列出了可以訪問該物件的所有域，以及如何進行訪問。這張表也稱為存取控制表(access control list)，或者叫ACL。如果在UNIX中實現這種技術，我們可以把每個檔的ACL放在一個單獨的磁片塊中，並在檔的i-節點中包含這個磁片塊的塊號。因為只存儲了非空項，全部ACL所需的存儲空間要比存儲整個矩陣所需的空間少得多。

為了說明ACL的工作原理，我們還是來想像一下它在UNIX中的應用。在UNIX中，域通過(uid，gid)對指定。事實上，ACL在UNIX的作用模型MULTICS中的使用或多或少與我們下面例子中描述的有些類似。因而，這個例子並不是憑空想像的。

假設有四個用戶(即uid)Jan，Els，Jelle和Maaike。他們分別屬於用戶組system，staff，student和student。另外我們還假定一些檔的存取控制表如下：

File0：(Jan，\*，RWX)

File1：(Jan，system，RWX)

File2：(Jan，\*，RW-)，(Els，staff，R--)，(Maaike，\*，R--)

File3：(\*，\*，R--)

File4：(Jelle，\*，---)，(\*，\*，R--)

括弧中的每個ACL項均指定了uid、gid以及對檔允許的訪問(讀、寫、執行)。星號代表所有的uid或gid。這個例子中，File0可以被uid為Jan，gid為任意值的進程讀寫和執行；File1只能被uid為Jan，gid為system的進程訪問。uid等於Jan而gid等於staff的進程可以存取檔File0，但無權訪問File1。File2允許uid等於Jan，gid為任意值的進程讀寫，此外，uid為Els而gid為staff的進程，以及uid等於Maaike而gid為任意值的進程都可以讀取該檔。任意進程都能夠讀取File3。File4比較有趣，只要uid是Jelle的進程都無權訪問File4，但是，所有其他進程均可以讀取它。利用ACL，我們可以禁止特定的用戶或使用者組訪問某個物件。

UNIX的具體實現方法略有不同。在UNIX中，每個檔都為檔主、檔主所在的用戶組以及其他用戶提供了三個位rwx，這種方案也屬於ACL，只是將其壓縮到9位。這是和物件相關聯的一張表，給出哪些使用者可以存取該物件，以及如何存取該物件。儘管9位元的UNIX方案遠遠沒有上述ACL系統那樣全面，實際上已經相當適用了，何況，其實現要簡單、方便得多。

物件的擁有者隨時都能改變物件的ACL。他可以禁止原先允許的訪問。唯一問題在於，改變ACL很可能不會影響當前正在使用該物件的使用者(即當前打開該檔的用戶)。

5.5.3 許可權

此外，我們也可以按行來分割矩陣。這時，每個進程都賦予一張該進程可以訪問的物件表以及對每個物件允許的操作，即域。這張表稱為許可權表(capability list)，其中的每一項叫做許可權(Dennis和Van Horn，1966；Fabry，1974)。

圖5-25是一張典型的許可權表。在每個許可權中，類型域給出了物件的類型，許可權域是一個點陣圖，表明對這種類型的物件允許執行的合法操作，物件域則是指向物件本身的指標(i-節點號)。許可權表本身也是物件，可以從其他的許可權表中引用，從而很容易實現子域的共用。許可權常常按他們在許可權表中的位置來引用。某個進程可能要求："在許可權2指向的檔中讀出1K資料。"這種形式的定址與UNIX中的檔描述符類似。

圖5-25 圖5-23中域2的許可權表。

顯然我們應該防止許可權表，或者常常被稱作C-表，被用戶篡改。有三種方法可用於保護許可權表。第一種方法需要帶特徵位元的體系結構。在硬體設計時，每個記憶體字有一個額外(或特徵)位元給出了該記憶體字是否包含有許可權。 該特徵位元不參與算數運算和比較運算，也不用於類似的普通指令中，它只能被運行在核心模式下的程式(即作業系統)修改。

第二種方法是把C-表保存在作業系統內部，進程引用許可權時給出許可權的槽口號，就象我們上面說到的那樣。Hydra(Wulf et al.，1974)就是這樣實現的。

第三種方法是把C-表加密後，直接保存在用戶空間內。這種方法尤其適用於分散式作業系統。它廣泛使用在Amoeba(Tanenbaum et al.，1990)中。

除了依賴於物件的特殊許可權，比如讀、寫和執行，許可權往往還有一些適用於所有物件的一般許可權。下面就是一般許可權的例子：

1. 拷貝許可權：為同一物件創建一個新許可權。

2. 拷貝物件：創建一個複製物件，使它具有新的許可權。

3. 刪除許可權：從C-表中刪除一項，相應對象不受影響。

4. 刪除對象：永久地刪除對象及其許可權。

最後要提的是，在許可權系統中，撤銷對某物件的訪問是很困難的。系統很難找出所有權限並將其收回。事實上，這些許可權存儲在遍及磁片的許可權表中。一種方法是把每個許可權指向一個間接物件，而不是指向物件本身。由於間接物件指向真正物件，系統總能中斷他們之間的聯繫，從而使得許可權失效(當一個指向間接物件的許可權後來向系統提交時，使用者會發現這個間接物件如今指向空物件。)

另一種廢除的辦法是Amoeba用到的。在Amoeba中，每個物件包含一個很長的亂數，這個亂數同時出現在許可權中。在使用權限時，將上述兩個數字進行比較。只有在他們一致時才允許操作。物件的擁有者能夠修改該物件的亂數，從而使容量失效。這兩種方案都不允許選擇性撤銷，也就是說，它不能只收回某些用戶的許可權。

5.5.4隱藏通道

儘管使用了存取控制表以及許可權表，安全洩漏還常常發生。這一節，我們討論一個這方面的問題。它是Lampson(1973)提出的。

Lampson的模型中有三個進程，主要適用於大型分時系統。第一個進程是客戶進程，它向第二個進程，即伺服器進程，請求執行某項操作。客戶進程和伺服器進程並不完全相互信任。例如，伺服器進程説明客戶進程填寫稅務單。客戶進程擔心伺服器進程會偷偷地記錄下這些資料，然後把這些資料出售給別人。而伺服器進程可能懷疑客戶進程會竊取其稅務程式。

第三個進程是協調進程。協調進程和伺服器進程合謀竊取客戶進程的秘密資料。協調進程和伺服器進程往往屬於同一使用者。這三個進程顯示在圖5-26中。我們希望設計一個安全系統，使得伺服器進程不可能向協調進程洩漏從客戶進程那兒獲得的資訊。Lampson把它叫做限制問題。

圖5-26 (a)客戶、伺服器和協調進程。 (b)封裝後的伺服器進程仍然可通過隱藏通道向協調進程洩露資訊。

系統設計者的目標是封裝或限制伺服器進程，使得它無法向協調進程傳送資訊。使用保護矩陣，我們很容易保證伺服器進程與協調進程之間不可能通過檔進行通信，也就是說，伺服器進程不可能把資料寫到一個協調進程能夠讀取的檔中。此外，我們還可以保證兩個進程不可能通過系統的進程間通信機制實現通信。

不幸的是，他們可以採用其他巧妙的通道進行通信。比如，伺服器進程可用下列方式傳送二進位位元流。為了傳送1，在一段時間內，它不停地進行運算。而傳送0時，伺服器進程睡眠同樣的時間段。

然後，協調進程可以仔細地監視它自己的反映時間，從而檢測出伺服器進程傳送的位元流。一般說來，伺服器進程在傳送0時，協調進程的反映速度要比傳送1好得多。這種通信通道稱為隱藏通道(covert channel)，如圖5-26(b)所示。

當然，隱藏通道是一個噪音通道，它含有大量的外部資訊。但是通過錯誤校驗碼我們可以在噪音通道上可靠地傳送資訊(例如海明威碼，甚至是更複雜的錯誤校驗碼)。錯誤校驗碼的使用降低了隱藏通道原來就很低的頻寬，然而足以洩漏重要資訊。顯然，任何基於物件和域矩陣的保護模型都無法防止這類洩漏。

調節CPU的使用並不是唯一的隱藏通道。頁面率也可以進行調節(很多分頁錯誤表徵1，無分頁錯誤表徵0)。事實上，幾乎所有按時間降低系統性能的方式都可用作隱藏通道。如果系統提供鎖檔的方法，伺服器進程可以給某些檔加鎖以表示1，而釋放鎖表示0。在有些系統中，進程能夠檢測出檔的鎖狀態，儘管它可能沒有權力訪問該檔。

請求和釋放專用資源(磁帶驅動器、繪圖器等等)也可用作隱藏通道的信號。在發送1時，伺服器進程請求該資源。在發送0時，伺服器進程釋放該資源。在UNIX中，伺服器進程創建一個檔表示傳送1，刪除該檔表示送0；協調進程用ACCESS系統調用檢測該檔是否存在。儘管協調進程無權訪問該檔，ACCESS調用依然可行。非常遺憾，還存在著許許多多其他的隱藏通道。

Lampson還提到一種向伺服器進程的擁有者洩露資訊的辦法。假定伺服器進程有權通知其擁有者它為客戶進程服務的工作量，以便對客戶計費。打個比方，如果實際計算出的帳單為100美元，而客戶的收入為53K美元，則伺服器進程可以把100.53報告給其擁有者。

試圖找出所有的隱藏通道是相當困難的，更不要說去堵塞他們了。實際上，我們幾乎毫無辦法。引入一個隨機產生分頁錯誤的進程，或者是利用其他降低系統性能的方法來減少隱藏通道的頻寬也根本不可取。

5.6 MINIX檔案系統概述

象所有的檔案系統一樣，MINIX檔案系統也必須解決我們上面討論的問題。它必須為檔分配空間和釋放空間、記錄磁片塊和空閒空間、提供某種方法以防止檔被未授權使用等等。在本章的剩下部分，我們將仔細研究MINIX的檔案系統，看看它是如何實現這些目標的。

在本章的前半部分，為了更具有普遍性，我們反復地引用了UNIX，而不是MINIX作業系統。但兩者的外部介面實質上是相同的。現在我們把注意力集中到MINIX檔案系統的內部設計上來。要瞭解UNIX的內核，請參看Thompson(1978)、Bach(1987)、Lions(1996)和Vahalia(1996)。

MINIX檔案系統不過是在用戶空間中運行的一個大型C程式(圖2-26)。讀寫檔時，使用者進程向檔案系統發送一條消息，檔案系統進行相應處理後，返回結果。實際上，MINIX檔案系統可以看成是和調用進程在同一台主機上運行的網路檔服務程式。

如此設計有深刻的含意。一方面，檔案系統可以獨立於MINIX的其他部分進行修改、調試和測試。另一方面，我們可以很方便地把整個檔案系統移植到任何帶有C編譯器的電腦上，在那台機器上進行編譯，並把它作為一台獨立的類UNIX遠端檔案伺服器。我們要做的唯一修改是消息的發送和接收方式。在不同的系統中，消息的發送和接收方式往往並不相同。

在下面的幾節中，我們將縱觀一下檔案系統設計中的許多關鍵領域。我們特別要介紹消息、檔案系統佈局、點陣圖、i-節點、塊快取記憶體、目錄與路徑、檔描述符、檔鎖以及設備檔(管道)。在研究了上述內容之後，我們將給出一個簡單的例子。通過跟蹤使用者進程執行READ系統調用的過程，說明各個部分是如何結合在一起運行的。

5.6.1 消息

檔案系統接收39種消息請求。除兩種外都用於MINIX的系統調用。這兩個異常消息是MINIX的其他部分生成的。在用於系統調用的消息中，有31種是從使用者進程中接收到的，而另外6種消息用於這樣的系統調用，他們首先被記憶體管理器所處理，然後記憶體管理器調用檔案系統完成一部分工作。檔案系統還處理兩種異常消息。這些消息如圖5-27所示。

圖5-27 檔案系統消息。

檔案系統的結構基本上與記憶體管理器和所有的I/O任務一樣。檔案系統的主迴圈程式不斷地等待消息。當收到消息後，它首先提取消息的類型，然後以其為索引查找檔案系統中處理各類消息的過程指標表。隨後調用相應過程，進行處理後，返回狀態值。檔案系統再把回答消息發送給調用進程，然後回到迴圈的開始，等待下一條消息的到來。

5.6.2 檔案系統佈局

MINIX檔案系統是一個邏輯的、自包含的實體，它含有i-節點、目錄和資料塊。MINIX檔案系統可以存儲在任何塊設備中，例如軟碟或一個硬碟分區。MINIX的檔案系統都有相同的佈局。圖5-28所示是一個128個i-節點和1K塊的360K軟碟的佈局。更大的系統，或者是那些有不同的i-節點數和塊大小的系統，也同樣由這6部分順序組成，但是各部分之間的相對大小可能不一樣。

圖5-28 最簡單磁片的磁片佈局，該磁片為360K軟碟，有128個i-節點，塊大小為1K(即兩個連續的512位元組磁區作為一個塊)。

每個檔案系統都以啟動區開始，啟動區中包含有可執行代碼。啟動電腦時，硬體從引導設備將啟動區讀入記憶體，轉而執行其代碼。啟動區代碼開始作業系統本身的載入過程。一旦系統啟動之後，啟動區不再使用。並非每個磁碟機均可用作引導設備，但是為了保持結構的一致，每個塊設備都為啟動區代碼保留一塊。這種方法最多不過浪費了一個塊。為防止硬體從非啟動設備上啟動，在將可執行代碼寫入引導設備的啟動區中時，在啟動區的已知位置處寫入魔數。從一個設備上啟動時，硬體(實際上是BIOS代碼)首先檢測魔數是否存在。若不存在，則拒絕把啟動區載入記憶體，這樣可以防止把垃圾用作引導程式。

超級塊(super-block)中含有檔案系統的佈局資訊，如圖5-29所示，它的主要功能是給出檔案系統不同部分的大小。如果給定塊大小和i-節點數，我們很容易算出i-節點點陣圖的大小和存放i-節點所需的塊數。例如1K的塊，每個點陣圖塊有1K位元組(8K位元)，可以記錄8192個i-節點的狀態(實際上第一塊只能處理8191個i-節點，因為0號i-節點並不存在，但我們在點陣圖中也為它保留一位)。10000個i-節點，要用到兩個點陣圖塊。每個i-節點占64位元組，1K的塊中可以有16個i-節點。如果有128個可用的i-節點，則需要8個磁片塊來存放。

圖5-29 MINIX超級塊。

後面我們會詳細地解釋區段和塊之間的區別。但是此時，讀者只要瞭解磁片存儲區可以以區段為單位進行分配，而每個區段可以包含1、2、4、8個，或一般情況下，2n個磁片塊。區段點陣圖按區段，而不是塊來記錄空閒存儲區。MINIX用到的所有標準軟碟中，區段大小和塊大小是一樣的(均為1K)，因此在這些設備上，可以近似把區段看成是塊。在本章後面部分詳細討論存儲分配之前，在看到"區段"時，只要把它當作"塊"就可以了。

可以注意到，每個區段包含的塊數並沒有存放在超級塊中，我們並不需要這一資料。我們存放的是底為2，區段數除以塊數所得值的對數。根據它，可以知道從區段轉換成塊或者從塊轉換成區段要移位的次數。例如，每個區段中含有8個塊，log28=3，因此要找到包含第128塊的區段，我們可以把128右移3位，得到16。

區段點陣圖中只包含資料區段(即點陣圖和i-節點用到的塊並不在該點陣圖中)。第一個資料區段在點陣圖中用區段1表示，同i-節點點陣圖一樣，區段點陣圖中第0位也未使用，因此第一個區段點陣圖塊只能映射到8191個區段，以後的每塊可以映射到8192個區段。考察一下新格式化磁片的點陣圖，可以發現i-節點和區段點陣圖中均有2位為1。一位是不存在的0號i-節點和0號區段，而另一位元是設備根目錄使用的i-節點和區段，在檔案系統創建時，根目錄自動存在。

我們還可以注意到，超級塊中的資訊冗餘。由於我們有1K空間可用於存放超級塊的資訊，因此我們可以事先按不同的形式算出所需的資訊，而不必在使用時重新計算。例如，磁片上的第一個資料區段的區段號，可以從塊大小、區段大小、i-節點數以及區段數計算得到。但是，直接把它存放在超級塊中要方便得多。超級塊中的剩下部分總歸是要浪費的，我們還不如用它來存儲一些有用的資料。

在MINIX啟動時，根設備中的超級塊被讀入記憶體中，同樣，安裝其他檔案系統時，他們的超級塊也讀入記憶體。記憶體的超級塊表中有些域不出現在磁片上，其中包括指定設備打開方式和位元組順序的標誌等，此外，記憶體的超級塊表還含有指向點陣圖第一個空閒位的域，使用這個域可提高存取速度，以及表徵超級塊所屬設備的一個域。

在磁片用作MINIX檔案系統之前，必須具有圖5-28所示的結構。實用程式mkfs可用來創建檔案系統。我們可以通過象

mkfs /dev/fd1 1440

的命令列調用該程式，在驅動器1中的軟碟上創建1440個塊的空檔案系統，這一命令還在超級塊中寫入魔數，表明該檔案系統是一個有效的MINIX檔案系統。此外，我們也可以通過一個原型檔來調用mkfs，其中列出要包含在新檔案系統中的目錄和檔。MINIX檔案系統經過改進，有些方面(比如i-節點大小)與早期版本不同，魔數還可以表明創建檔案系統的mkfs的版本。從而處理他們之間的不同。MOUNT系統調用檢查超級塊中魔數和其他信息，可以拒絕安裝不是MINIX格式的檔案系統，例如MS-DOS的檔案系統。

5.6.3 點陣圖

MINIX用兩個點陣圖來記錄空閒i-節點和空閒區段(見圖5-29)。當檔被刪除時，很容易算出哪一個點陣圖塊包含了所釋放i-節點的相應位，利用通常的快取記憶體機制查找該塊，一旦找到，相應於被釋放i-節點的那一位清0。區段在區段點陣圖中的釋放也類似。

邏輯上，在創建檔時，檔案系統必須在點陣圖塊中查找第一個空閒i-節點，把它分配給這個新創建的檔。然而，超級塊在記憶體的拷貝中有一個域指向第一個空閒i-節點，因此不必進行查找，在該空閒i-節點分配使用後，就需要修改指標，使它指向下一個空閒i-節點，往往是下一個或者較近的一個節點。同樣地， i-節點被釋放後，檢查這個i-節點是否位於第一個空閒i-節點的前面，若是，則需要修改指向第一個空閒i-節點的指標。如果磁片上的所有i-節點全被使用，查找函數返回0，這也是0號i-節點未使用的原因(即它可以用於表明查找失敗)(在mkfs創建新檔時，它把0號i-節點清零，並把點陣圖中的最低位設置為1，防止檔案系統分配0號i-節點)。上面所講的同樣也適用于區段點陣圖。邏輯上，申請空間時，需要在區段點陣圖中查找第一個空閒區段，但是超級塊的記憶體拷貝保存了指向第一個空閒區段的指標，因而消除了很多順序查找點陣圖的麻煩。

有了這些知識後，現在我們可以解釋區段和塊之間的不同了。使用區段的目的是：確保同一檔的所有磁片塊都位於同一個柱面上，從而改進順序讀取檔的性能。我們採用了一次分配多個塊的方法。假設塊大小為1K，而區段大小為4K，區段點陣圖中記錄了區段，而非塊的使用情況。一個20M的磁片有5K個4K大小的區段，因此其區段點陣圖需要5K位元。

檔案系統的大多數部分都以塊為單位進行操作的。磁片每次傳送一塊，快取記憶體也按照塊進行處理，系統中只有記錄物理磁片位址的一小部分(例如區段點陣圖和i-節點)需要知道區段的存在。

在開發MINIX檔案系統的過程中，我們不得不做一些設計上的決策。1985年，當MINIX還處於構思階段的時候，磁片容量很小，許多使用者只有軟碟。在V1檔案系統中，我們決定把磁片位址限制在16位元，這樣能夠把大多數地址存放在間接塊中。16位的區段號和1K大小的區段只能定址64K個區段，從而將磁片容量限制為64M。那時，這可是一個相當大的容量，我們當時還考慮到，需要擴充磁片容量時，很容易把區段大小轉換為2K或者4K，而不必改動塊的大小。16位的區段號還容易使得i-節點的大小保持為32位元組。

由於MINIX不斷發展，同時大磁片更加普及，因此有必要對MINIX的檔案系統進行修改。許多檔長度小於1K，因此增加塊大小意味著浪費磁片頻寬和讀寫幾乎是空的塊，也意味著把它存放在快取記憶體時浪費了寶貴的主存。原本我們是可以增加區段大小的，但是大的區段浪費了更多的磁碟空間，何況我們還希望保持對小磁片操作的高效率。當然，另一種合理的解決方案是在不同大小的設備中使用不同大小的區段。

但最終我們決定把磁片指標的大小增加到32位。這使得在塊大小和區段大小均為1K時，MINIX V2版本的檔案系統可以處理容量達4TB的設備。作出這種決定的另外一個原因與i-節點的更改有關。由於i-節點中包含的內容增加了，為此我們把i-節點的大小也增加到64個位元組。

區段的使用也帶來了意料不到的問題。我們用一個簡單的例子來闡述，考慮4K的區段和1K的塊，假設檔長度為1K，這時檔被分配了一個區段，1K和4K之間的磁片塊中含有垃圾(以前使用者的殘留資料)，但是這不會對我們造成不利。因為在i-節點中檔大小清楚地標記為1K。事實上，包含垃圾的磁片塊根本不會讀到快取記憶體中，因為讀操作是以塊，而不是區段為單位進行的。超出檔案結尾的讀操作總是返回0，不包含任何資料。

假設現在有人將檔指標定在32768位元組並寫入1個位元組資料，檔長度變為32769位元組。隨後讀取1K後面的資料將返回該塊以前的內容，從而形成一個很大的安全缺口。

解決這一問題的方法是，執行寫操作時，檢查寫入位置是否超出了檔案結尾。若超出，則將檔的最後一個區段中所有還未分配的塊清空。儘管這種情況很少發生，然而我們的代碼必須進行處理，這使得MINIX檔案系統更加複雜。

5.6.4 i-節點

MINIX中i-節點的佈局如圖5-30所示，它幾乎與標準UNIX的i-節點相同。磁片區段指標是一些32位的指針，總共有9個這樣的指針：7個直接的，2個間接的。MINIX的i-節點占64個位元組，這也同標準UNIX的i-節點一樣。有一個未使用的空間可以用於第10個(三次間接)指標，我們在MINIX檔案系統的標準版本中還不支持這一指針。MINIX i-節點中的存取時間、修改時間以及i-節點的修改時間都和標準UNIX一樣。i-節點的修改時間在大多數檔操作時都要進行修改，只有讀檔除外。

圖5-30 MINIX i-節點。

在打開檔時，首先要找到檔的i-節點，並把它載入記憶體的inode表中，直至關閉前，它一直保存在記憶體中。記憶體的inode表中有一些域不出現在磁片上，例如，i-節點所在設備的設備號以及在該設備上的i-節點號，通過這兩個值檔案系統可以知道在記憶體中資料修改後，要將這些資料寫到何處。每個i-節點還有一個計數器，當檔多次打開時，在記憶體中只保存一個i-節點拷貝，但是每次打開該檔，計數器加1，每次關閉該文件，計數器減1。只有在計數器減到0時，才將i-節點從inode表中刪除。若i-節點自上次調入記憶體之後被修改過，則要將它寫回磁片。

檔i-節點的主要功能是給出檔資料塊所在的位置。前7個區段號就放在i-節點結構之中。對於MINIX標準發行版本，區段大小和塊大小均為1K，因此小於7K的檔不必使用間接塊。如果檔長度超過7K，就要使用間接區段。MINIX中採用了圖5-10所示的方案，只是它只用到了一次間接塊和兩次間接塊。若塊大小和區段大小均為1K，區段號為32位，則一次間接塊含有256項，可以表示1/4M的存儲區，兩次間接塊指向256個一次間接塊，因此可以存取長達64M的檔。MINIX檔案系統中檔的最大長度為1G，我們可以使用三次間接塊或修改區段大小來存取大於64M的檔。

i- 節點中還包含有模式資訊，它給出了檔的類型(正規檔、目錄、塊設備檔、字元設備檔或管道)以及保護標誌、SETUID位元和SETGID位。i-節點的連結數目域記錄了有多少個目錄項指向這個i-節點，因此檔案系統知道什麼時候該釋放檔的存儲區。我們不應當把它與打開檔計數器(只出現在記憶體的inode表中，不在磁片上)相混淆，後者指出了檔被打開的次數，而且往往是被不同的進程所打開。

5.6.5 塊快取記憶體

MINIX使用塊快取記憶體來改進檔案系統性能。快取記憶體用一個緩衝陣列來實現，其中每個緩衝區由包含指標、計數器和標誌的頭以及用於存放磁片塊的體組成。所有未使用的緩衝區均使用雙鏈表，按最近一次使用時間從近到遠的順序連結起來，這如圖5-31所示。

圖5-31 塊快取記憶體使用的連結表。

為了迅速判斷某一塊是否在記憶體中，我們使用了雜湊表。所有緩衝區，如果它所包含塊的雜湊代碼為k，在雜湊表中用第k項指向的單鏈錶鏈接在一起。雜湊函數提取塊號低n位作為雜湊代碼，因此不同設備的塊可以出現在同一雜湊鏈之中。每個緩衝區都在其中某個鏈中。在MINIX啟動，初始設定檔案系統時，所有緩衝區均未使用，並且全部在雜湊表第0項指向的單鏈表中。這時，雜湊表其他項均為空指針。但是一旦系統啟動完成，緩衝區將從0號鏈中刪除，放到其他鏈中。

檔案系統需要一塊時，它調用過程get\_block，計算該塊的雜湊代碼，搜索相應鏈。get\_block被調用時，有兩個參數：設備號和塊號。這兩個值與緩衝區鏈中對應域相比較，如果找到包含這一塊的緩衝區，則緩衝區頭中標誌塊使用次數的計數器加1，並返回指向該緩衝區的指標。如果在雜湊表中未找到這樣的塊，我們可以使用LRU鏈中的第一個緩衝區。LRU鏈中的緩衝區必然不在使用中，因而它包含的塊可以被換出記憶體，以釋放這個緩衝區。

如果選定某一塊調出記憶體，這時需要檢查塊頭中另一個標誌，看它在上次讀入記憶體之後是否修改過。若修改過，就重新寫回磁片，然後檔案系統向磁片任務發送一條消息，要求讀入新塊，之後檔案系統被掛起，直到塊讀入後才繼續運行，將指向該塊的指標返回給調用進程。

請求塊的過程完成任務後，它調用另一個過程put\_block釋放該塊。正常情況下，塊在讀入後立即被使用並釋放。但也有可能在它被釋放前出現其他對該塊的請求，因此put\_block僅僅是將使用計數器減1。當使用計數器減到0時，才將它放到LRU鏈中。否則，我們就讓它保留在那兒。

在put\_block的參數中，有一個給出被釋放塊的類型(例如i-節點、目錄、資料)。對於不同類型的塊，需要作不同的考慮：

1. 把該塊放在LRU鏈頭部還是尾部?

2. 是否把該塊(如果修改過)立即寫回磁片?

最近不可能使用的塊，比如超級塊，放在LRU鏈的前端。這樣，如果下次需要一個空閒緩衝區，可以立即使用之。所有其他塊都按真正的LRU方式放在LRU鏈的尾部。

修改的塊只在下面兩種情況下寫回磁片：

1. 它到達LRU鏈前端並被換出。

2. 執行了SYNC系統調用。

SYNC並不遍歷LRU鏈，相反，它查找快取記憶體中的緩衝區陣列。緩衝區即使未被釋放，如果它被修改過，SYNC也可以找到它，並修改它在磁片上的副本。

但是，有一種情況比較特殊。修改過的超級塊要立即寫回磁片。在早期的UNIX版本中，安裝檔案系統時要修改超級塊，我們把修改後的超級塊資訊立即寫回磁片，可以減少在系統崩潰時檔案系統被破壞的可能性。而現在的MINIX版本中，超級塊不再被修改，因此那些用於把超級塊立即寫回磁片的代碼已經過時了。在標準配置中，其他塊都不立即寫回。但是，若修改系統設定檔include/minix/config.h中ROBUST的缺省定義，然後重新編譯檔案系統，則可以實現i-節點、目錄、間接塊和點陣圖塊在釋放時立即寫回磁片。之所以這樣，是為了使檔案系統更加健壯；其代價是運行速度下降。這種處理是否有效，我們並不清楚。但是，如果斷電，不管丟失的是i-節點塊還是資料塊，總是件使人頭疼的事。

我們還可以看到，塊中表明塊已修改過的標誌是由檔案系統中請求和使用該塊的過程設置的。get\_block和put\_block過程只關心對鏈表的操作，他們不知道檔案系統的哪個過程需要該塊，更不知道它為什麼需要該塊。

5.6.6 目錄和路徑

在檔案系統中另一個重要的子系統是目錄和路徑名的管理。許多系統調用，例如OPEN，都以檔案名作參數。實際需要的是檔的i-節點，因此，檔案系統需要在目錄樹中查找檔，找到相應的i-節點。

MINIX目錄由包含16個位元組目錄項的檔組成。目錄項的頭兩個位元組構成16位元的i-節點號，剩下的14個位元組是檔案名，這一點和我們在圖5-13中見到的傳統UNIX目錄項相同。為了查找路徑/usr/ast/mbox。檔案系統首先在根目錄中查找usr，然後在/usr中查找ast，最後在/usr/ast中查找mbox。每次只查找路徑的一個部分，如圖5-14所示。

唯一麻煩的是遇到被安裝的檔案系統。在MINIX和其他類UNIX系統中，通常配置有一個小的根檔案系統，根檔案系統中含有用於啟動和進行基本維護的檔。此外，配置中還有很多其他檔，包括使用者的目錄，他們都存放在另一個設備上，該設備安裝至/usr目錄下。現在我們可以來介紹安裝是如何實現的了。當使用者在終端上鍵入命令

mount /dev/hd2c /usr

時，包含在硬碟第2個分區上的檔案系統被安裝到根檔案系統的/usr目錄下。安裝前和安裝後的檔案系統如圖5-32所示。

圖5-32 (a)根檔案系統。 (b)未被安裝的檔案系統。 (c) (b)的檔案系統安裝到

目錄/usr後的效果。

安裝成功後，在/usr的i-節點的記憶體拷貝中設置了一個標誌位元，表明有設備已安裝到這個i-節點上，這就是關鍵所在。MOUNT調用還把新安裝的檔案系統的超級塊調入記憶體中的超級塊表中，並設置其中兩個指針。此外，系統還把被安裝檔案系統的根目錄i-節點也放在記憶體的inode表中。

在圖5-29中，我們看到，被安裝檔案系統的超級塊中有兩個域。第一個是i-node-of-the-mounted-file-system域，指向被安裝檔案系統的根目錄i-節點。其次是i-node-mounted-on域，指向該檔案系統所安裝到的i-節點，在這裡是/usr的i-節點。這兩個指針把被安裝的檔案系統和根檔案系統連結起來[如圖5-32(c)中虛線所示]，使得被安裝的檔案系統能夠正常工作。

在查找路徑/usr/ast/f2時，檔案系統在/usr的i-節點中發現一個標誌，知道它需要在安裝到/usr的檔案系統的根目錄i-節點上繼續查找。問題在於，它如何能找到這個i-節點呢？

答案很顯然的。系統搜索記憶體中所有的超級塊，直至找到i-node-mounted-on域指向/usr的那一塊，它必然是檔案系統安裝到/usr上的超級塊。一旦它找到了這個超級塊，便很容易沿著另一個指標找到被安裝檔案系統的根目錄i-節點。現在檔案系統可以繼續查找了。在圖5-32所示的例子中，它在硬碟第2個分區的檔案系統的根目錄下繼續查找ast目錄。

5.6.7 檔描述符

文件打開時，文件描述符被返回給使用者進程。其後，READ、WRITE調用都使用檔描述符對檔進行操作。這一節，我們介紹一下檔案系統中是如何來管理文件描述符的。

象內核和記憶體管理器一樣，檔案系統也在其位址空間內維護進程表的部分內容。其中有三個域需要特別注意：前兩個是指向根目錄i-節點的指標和指向工作目錄i-節點的指標。象圖5-14中的路徑查找總是從其中一個目錄開始，它取決於路徑是絕對路徑還是相對路徑，執行CHROOT和CHDIR系統調用可以改變這兩個指標，使他們指向新的根目錄和新的工作目錄。

進程表中的第三個域是以檔描述符號為下標的陣列，根據檔描述符可以在該陣列中找到適當的檔。粗略地看，我們把陣列中的第k個項指向描述符為k的檔的i-節點就夠了。畢竟，在檔打開時，只需要把它的i-節點調入記憶體，並保存在其中直至檔關閉，所以說，這種方法肯定能奏效。

很不幸的是，這種簡便方法並不可行。事實上，在MINIX中(同UNIX一樣)，檔能夠共用。每個檔都有一個32位元的數字表徵下一個讀寫的位元組位置，這個數位叫做檔位置，每當執行系統調用LSEEK時進行修改。我們可以把問題簡單地描述成："檔位置應該如何存放呢？"

第一種可能是把它放在i-節點中。然而，如果兩個或更多的進程同時打開同一個檔，他們都應該有自己的檔指標。一個進程的LSEEK操作影響到另一個進程的讀操作是不可思議的。因此，檔位置不能存放在i-節點中。

那麼把它存放在進程表中又如何呢？為什麼我們不使用一個與檔描述符陣列相平行的陣列來給出檔當前位置呢？這個想法也不可取，但是原因就更微妙了。問題來自FORK系統調用的語義。當生成一個進程時，父進程和子進程必須共用打開檔的指標。

為了更好地理解這個問題，我們考慮一個其輸入重定位到某檔的shell腳本。當該shell生成第一個子進程時，其標準輸出檔的位置為0 。然後，子進程繼承該位置，輸出1K資料。在子進程結束之後，共用檔位置應該為1K。

假設shell讀取更多的shell腳本，並且生成另一個子進程，第二個子進程必須從shell中繼承1K的檔位置，這樣，第二個子進程接著在第一個子進程的位置之後輸出。如果shell不和子進程共用檔位置，那麼第二個子進程可能重寫第一個子進程的輸出，而不是在第一個子進程的輸出之後追加資料。

所以，我們也不能把檔位置放在進程表中，它必須真正地實現共用。在MINIX中，解決這個問題的方法是引入一個新的共用表filp，其中包含了所有的檔位置。filp的使用見圖5-33所示。通過真正地共用檔位置，FORK語義能正確地實現了，shell腳本也能正常工作了。

圖5-33 父進程與子進程共用檔位置。

儘管filp表中真正必須包含的項是共用檔的位置，我們也可以把指向檔i-節點的指標放在其中。這樣，進程表中的檔描述符陣列只需包含指向filp項的指標。每個filp項中還包含著檔模式(允許位元)、一些表明檔是否以特殊模式打開的標誌、以及使用這個filp項的進程計數器。檔案系統可以判斷最後一個使用它的進程是否終止，若終止，則釋放這個filp項。

5.6.8 文件鎖

在檔案系統的管理中，另外還有一種情況要使用專門的表結構，這就是檔鎖。MINIX支持POSIX建議性檔鎖(advisory file locking)的進程間通信機制，允許檔的任何一個或多個部分標記為已鎖。MINIX作業系統不強制進行鎖操作，但是進程在執行可能與其他進程相衝突的操作前，可以對檔鎖進行查詢。

我們之所以為鎖提供一個單獨的表結構，原因與上節講到的filp表相同。一個進程可以設置多個鎖，而一個檔的不同部分可以被不同進程上鎖(但是，不同的鎖不能重疊)。因此進程表和filp表均不適於記錄檔鎖。因為檔中可以有多個鎖，所以檔的i-節點也不宜用來記錄鎖。

MINIX用另外一張表，即file\_lock表，來記錄所有鎖。表中每一項都包含鎖類型，表明對該檔加的是讀鎖還是寫鎖。此外，file\_lock各項中還有加鎖進程的進程號、指向被鎖檔i-節點的指標以及加鎖範圍的第一個位元組和最後一個位元組在檔中的偏移。

5.6.9 管道和設備檔

管道和設備檔與普通檔有很大的不同。當進程讀寫一個磁片檔時，該操作最多在幾百個毫秒內就能完成。最糟糕時，也不過需要2到3次磁片訪問。而讀管道時，情況就不同了：如果管道為空，讀進程可能要等待另一個進程把資料寫到管道中，這可能要幾個小時。類似地，當從終端上讀取資料時，進程必須等待，直至使用者輸入。

因此，檔案系統通常的規則，即處理請求直至完成，就不再可行了。我們有必要把這些請求掛起，在以後再重新執行。當進程試圖從管道中讀寫資料時，檔案系統立即檢查管道的狀態，看該操作是否能夠完成。若能，就等待它執行完成。否則，檔案系統把系統調用的參數記錄在進程表中，以便在時機到來後繼續執行之。

檔案系統並不需要採取特殊方法掛起調用進程，它要做的無非是不發送回答信號，讓等待回答信號的調用進程自動阻塞。掛起進程後，檔案系統回到主迴圈中等待下一個系統調用。一旦別的進程修改了管道的狀態使得掛起進程能夠運行結束，則檔案系統設置某個標誌位元。這樣，在下一次迴圈時，檔案系統從進程表中提取掛起進程的參數，繼續執行前面的系統調用。

終端和其他字元設備檔略有不同。每個設備檔的i-節點中都含有兩個數位：主設備號和次設備號。主設備號給出了設備類型(例如RAM盤、軟碟、硬碟、終端)，它可用作檔案系統表的索引，而檔案系統表將主設備號映射到相應的系統任務號(即I/O驅動程式)。事實上，主設備號指定了調用哪個I/O驅動程式。次設備號作為參數傳遞給驅動程式，指明使用的設備，例如終端2或驅動器1。

在有些情況下，尤其是終端設備中，次設備號包含了一些設備資訊。例如，MINIX的主控制台/dev/console的主、次設備號分別為4、0。虛擬控制台也使用同樣的驅動程式處理，如/dev/ttyc1(主、次設備號為4、1)和/dev/ttyc2(主、次設備號為4、2)等。串列終端需要使用不同的底層軟體，這些設備，即/dev/tty00和/dev/tty01，的設備號為4，16和4，17。類似地，網路終端使用偽終端驅動程式，也用到不同的底層軟體。在MINIX中，網路終端設備包括ttyp0、ttyp1等，其設備號為4，128和4，129。這些偽設備每個都有一個與之關聯的設備，比如ptyp0、ptyp1等，與他們相應的主、次設備號分別是4，192和4，193等。之所以選擇這些數位是為了方便驅動程式調用每組設備所需要的底層函數。對任何人來說都不大可能在一個MINIX系統中配備192台以上的終端。

當進程要從設備檔中讀取資料時，檔案系統從檔的i-節點中提取主設備號及次設備號，並根據主設備號獲得相應的任務號。接著，檔案系統向該任務發送一條消息，其中以次設備號、要執行的操作、調用進程的進程號、緩衝區位址和要傳送的位元組數為參數。消息的格式與圖3-15中相同，只是沒有用到POSITION項。

如果驅動程式能夠立即執行該操作(例如，在終端上已輸入了一行)，它把資料從自己的內部緩存拷貝到使用者緩衝區中，然後向檔案系統發回回應訊息，告知執行完畢。檔案系統再向使用者發一個回答信號。這樣，這個調用結束。需要注意的是，驅動程式並不把資料拷貝到檔案系統中。從塊設備檔中讀取的資料要經過塊快取記憶體，但是從字元設備檔中讀取資料則沒有這個必要。

如果驅動程式不能進行這個操作，它把消息參數記錄在自己的內部表中，立即向檔案系統發送一個回應，說明該系統調用無法完成。這種情況同檔案系統發現某人試圖從空管道中讀取資料一樣。檔案系統記錄下進程被掛起的事實，隨後等待下一條消息。

在獲得足夠完成調用的資料後，驅動程式將這些資料傳送到被阻塞使用者程式的緩衝區中，向檔案系統發送消息報告它完成的操作。檔案系統再向使用者程式發送消息使其解除阻塞，並告知已傳送的位元組數。

5.6.10 一個例子：READ系統調用

不久我們會看到，檔案系統的很多代碼都用於系統調用的執行。因此，我們有必要作個小結。我們來看看READ這個最重要的系統調用是如何實現的。

當使用者程式執行語句

n=read(fd，buffer，nbytes)；

讀取普通檔時，庫過程read被調用。它首先創建一條消息，其中包含fd、buffer、nbytes等參數，以及表示READ類型的消息碼。然後將這條消息送給檔案系統，並阻塞以等待檔案系統的回應。檔案系統在收到消息後，以消息類型為下標查找過程表，調用相應過程處理讀請求。

該過程從消息中提取出檔描述符，由此找到相應的filp項以及要讀取檔的i-節點(參見圖5-33)。接著，讀請求被分成幾個段，每段對應一塊。例如，如果當前的檔位置為600位元組，要讀取的資料長度為1K位元組。那麼，讀請求將分成兩個部分，分別是從600到1023位元組和從1024到1623位元組(假定塊大小為1K位元組)。

對於上述各段，依次檢查他們的相關塊是否在快取記憶體中。如果不在，檔案系統選擇最久未使用的塊，把它調出記憶體並收回其緩衝區，如果這一塊在上次調入之後修改過，檔案系統向磁片任務發送一條消息，將其寫回磁片，然後，檔案系統還要請求磁片任務將所需的塊讀入。

如果要讀入的塊已在快取記憶體中，那麼檔案系統向系統任務發送一條消息，請求它把資料拷貝到使用者緩衝區中(即從600到1023位元組的資料拷貝到使用者緩衝區起始位置，而從1024到1623位元組的資料拷貝到從424位元組開始的使用者緩衝區中)。在拷貝之後，檔案系統向使用者程式送出回應訊息，告知拷貝的位元組數。

在使用者程式收到回應後，庫函數read提取回應代碼，作為函數值返回給調用進程。

這裡還有額外的一步，其實它並不是READ調用的一部分。 如果對塊設備執行的是讀操作，並且滿足一些其他條件，檔案系統在讀出資料，送迴響應後，將繼續讀取下一塊。順序讀取檔非常普遍，因此可以設想下一次讀操作將請求檔的下一塊，於是提前做這一操作，當實際需要時，所需的磁片塊就已經在快取記憶體中了。

5.7 MINIX檔案系統的實現

MINIX的檔案系統相對較大(多於100頁C代碼)，但是非常直觀。執行系統調用的請求到達後，進行相應的處理，然後送回回答信號。下面幾節中，我們逐個分析檔，指出其中的關鍵之處，代碼本身也含有許多注釋以便於讀者閱讀。

在分析MINIX其他部分的代碼時，我們首先分析進程的主迴圈，接著分析處理不同消息類型的函數。我們按照不同的方式來分析檔案系統：首先我們研究主要的子系統(快取記憶體管理、i-節點管理等等)，接著我們討論主迴圈和各種檔操作的系統調用，之後是關於目錄作業系統調用的研究，最後我們介紹其他的系統調用。

5.7.1 標頭檔和全域變數

同內核和記憶體管理器一樣，檔案系統中使用的資料結構和表都定義在標頭檔之中，有些資料結構放在目錄include/及其子目錄下的系統標頭檔中。例如，include/sys/stat.h定義了系統調用向其他程式提供i-節點資訊的格式，而include/sys/dir.h中定義了目錄項結構。這兩個檔是POSIX所需要的。此外，全域設定檔include/minix/config.h中的許多定義也會影響檔案系統。例如，ROBUST巨集表明是否將重要的檔案系統資料結構在修改後立即寫回磁片。NR\_BUFS和NR\_BUF\_HASH則控制塊快取記憶體的大小。

檔案系統標頭檔

檔案系統本身的標頭檔放在檔案系統的原始目錄/src/fs/中，在看過MINIX系統的其他部分後，許多檔案名我們都很熟悉了。檔案系統的主要標頭檔fs.h(19400行)和src/kernel/kernel.h及src/mm/mm.h相似，它包含了檔案系統的C來源程式中引用的其他標頭檔，如const.h、type.h、proto.h和glo.h等等。下面我們逐個分析這些標頭檔。

const.h(19500行)中定義了整個檔案系統中使用到的一些常數，例如表長度和標誌位元等等。MINIX有一定的開發歷史，其早期版本有不同的檔案系統。現在的MINIX同時支援V1版本和V2版本的檔案系統，因此用戶可以存取早期MINIX版本寫的檔。檔案系統超級塊中包含有魔數，根據魔數作業系統可以判別檔案系統的版本號。常數SUPER\_MAGIC和SUPER\_V2定義了這些魔數。對於老版本的支持並不只是理論上的研究，常常是軟體人員實現新版本時主要考慮的因素。系統設計者必須決定花多大的精力來方便老版本的用戶。我們會看到在MINIX檔案系統中，有幾個地方支持老版本是很麻煩的事。

type.h(19600行)同時定義了V1版本和V2版本的i-節點在磁片上的結構。V2版本i-節點的長度是V1版本的兩倍。V1版本的檔案系統是為不帶硬碟驅動器和360-KB磁片的系統而設計的。V2版本中提供了UNIX系統中有的三個時間域。V1版本的i-節點中只有一個時間域，但是可用STAT或FSTAT來模擬，STAT或FSTAT返回一個包含有這三個域的stat結構。支持這兩種版本的檔案系統稍微有些困難，這一點我們在19616行的注釋中提到。老版本的MINIX將gid\_t類型定義為八位的值，所以d2\_gid必須聲明為u16\_t類型。

proto.h(19700行)提供了K&R C編譯器或ANSI標準C編譯器所支援的函數原形。這是一個很長的檔，但並不很重要。只是有一點需要注意：因為檔案系統處理多種不同的系統調用，並且由於檔案系統的組織方式，不同的do\_xxx函數分散在許多檔之中。proto.h按檔進行組織，如果要查閱處理某個特定系統調用的代碼，可以在proto.h找到包含這個系統調用的檔。

最後，glo.h(19900行)定義了全域變數。接收和返回消息的緩衝區也定義在這個檔中。這裡我們還用到了EXTERN巨集，因而這些變數可以為檔案系統的各個部分所存取。同MINIX的其他部分一樣，在編譯檔table.c時，系統會保留一定的存儲空間。

進程表的檔案系統部分包含在檔fproc.h中(20000行)。fproc陣列也用EXTERN巨集聲明，它含有模式遮罩碼、指向根目錄i-節點和工作目錄i-節點的指標、檔描述符陣列、uid和gid和各個進程的終端號，進程的id以及進程組的id也包含在這裡。這些項中有一部分在內核和記憶體管理器的進程表中重複定義。

在系統調用被中途掛起，例如當從空管道中讀取資料時，需要保存調用參數，為此專門使用了幾個域。fp\_suspended域和fp\_revived域實際上只需要一位元，然而使用字元時編譯器可以生成更好的代碼。此外，有一個域為POSIX標準所要求的FD\_CLOEXEC位而設置，用於表明在執行EXEC系統調用時應該關閉檔。

現在我們來介紹檔案系統中定義其他表的檔。文件buf.h(20100行)定義了塊快取記憶體，這個結構也用EXTERN聲明。buf陣列中含有所有的緩衝區，每個緩衝區分為資料部分和頭部分。頭部分中含有指標、標誌和計數器等。資料部分說明為五種類型的聯合(20117行)，這樣我們可以把塊視作為字元陣列，或目錄等來引用。

如果把緩衝區3作為字元陣列，引用其資料部分的正確方法是buf[3].b.b\_\_data。這裡，buf[3].b表示整個聯合，我們使用的是其b\_\_data域。儘管正確，但這種方法略顯繁瑣。因此在20142行定義了宏b\_data，這樣上述引用可以用buf[3].b\_data來表示。請注意，b\_\_data(聯合中的域)中含有兩個底線。而為了區別，在b\_data(宏)中只使用了一個底線。訪問塊的其他巨集定義列在20143行至20148行。

緩衝區雜湊表buf\_hash定義在20150行。雜湊表中的每一項指向一個緩衝區列表，初始時，所有列表均為空。文件buf.h末尾的宏(20160行至20166行)定義了不同的塊類型。在塊使用完畢並返回到快取記憶體中時，快取記憶體管理進程可以根據這些值決定是否把塊放在LRU鏈的前端或尾部、以及是否將該塊立即寫回磁片。WRITE\_IMMED位表明塊被修改後，應該馬上寫回磁片。超級塊是檔案系統中唯一無條件設置WRITE\_IMMED的資料結構。那麼MAYBE\_WRITE\_IMMED是什麼意思呢？MAYBE\_WRITE\_IMMED定義在檔include/minix/config.h中，當ROBUST為真時，它等於WRITE\_IMMED，否則等於0，在MINIX的標準配置中，ROBUST定義為0。

文件buf.h的最後一行(20168行)利用檔include/minix/config.h中NR\_BUF\_HASH的值定義了HASH\_MASK。HASH\_MASK和塊號相與，根據結果，在buf\_hash相應項所指向的鏈表中開始查找該塊的緩衝區。

下一個檔dev.h(20200行)定義了dmap表，表本身在table.c中說明並置初值。dmap表的定義不能同時包含在幾個檔中，這就是為什麼使用dev.h檔的原因。這裡，dmap用extern而不是EXTERN聲明。dmap表提供了主設備號與相應任務之間的映射。

文件file.h中包含了中間表filp(20300行)(用EXTERN聲明)。filp表用於存放文件當前位置及i-節點指針(參見圖5-33)，此外，它還給出了打開的檔是否可以讀寫，以及有多少個檔描述符指向該項等。

檔鎖表file\_lock(用EXTERN聲明)放在檔lock.h(20400行)中。這個陣列的長度由NR\_LOCKS指定，NR\_LOCKS在檔const.h中定義為8。如果想要在MINIX系統上實現一個多使用者的資料庫，這個數位還要增加。

在文件inode.h(20500行)中定義了i-節點表inode(聲明為EXTERN)，用於保存當前使用的i-節點。我們前面說過，在打開檔時，檔的i-節點被讀入，並保存在記憶體中直至檔關閉。inode結構定義了保存在記憶體中，還未寫入到磁片i-節點上的資訊。請注意，這裡的i-節點只有一個版本，在將i-節點從磁片上讀入時，即處理了V1版本和V2版本的檔案系統的差別。檔案系統其他部分不必瞭解磁片上檔案系統的格式，至少在修改資訊被寫回磁片之前是這樣。

至此，多數域都是自解釋的，然而我們有必要說明一下i-seek。前面提到過，為了優化，在檔案系統順序讀取檔時，它會在請求前即將某些塊讀入到快取記憶體中。而隨機存取檔沒有預讀。在執行LSEEK系統調用時，i-seek域設置為禁止預讀。

文件param.h(20600行)與記憶體管理器中的同名檔相似。它為包含參數的消息域定義了名稱，因而代碼中可以引用buffer來代替m.ml\_pl，從訊息緩衝區m中選擇一個域。

在文件super.h(20700行)中，我們定義了超級塊表。系統啟動時，根設備的超級塊被載入。而安裝一個檔案系統時，其超級塊同樣被載入。同其他表一樣，super\_block表也用EXTERN加以聲明。

檔案系統存儲區分配

本節我們要討論的最後一個檔並不是標頭檔，但是同記憶體管理器一樣，在分析了所有的標頭檔之後，我們有必要介紹檔table.c，它在編譯時包含了上述標頭檔。同檔案系統的全域變數以及進程表的檔案系統部分一樣，我們前面提到的大多數資料結構，如塊快取記憶體、filp表等等，都用EXTERN宏聲明。在編譯table.c時即保留一定的存儲區，這一點也與MINIX系統的其他部分相似。table.c還包含兩個重要的初始化陣列。call\_vector是指標陣列，可用於在主迴圈中決定過程與系統調用號的對應關係，在記憶體管理器中，我們也看到過類似的表。

在20914行有一張新表，即dmap表。從0開始，每個主設備對應一行。當設備被打開、關閉和讀寫時，dmap表提供了執行相應操作所調用的過程名。所有這些過程都放在檔案系統的位址空間中，大多數過程什麼也不做，也有些程序呼叫某個任務，請求I/O。另外，dmap表還給出了每個主設備的相應任務號。

要在MINIX中增加一個新的主設備，必須在dmap表中增加一行，說明在設備被打開、關閉和讀寫時，執行何種操作(如果有的話)。例如，如果在MINIX中增加一個磁帶驅動器，當打開磁帶驅動器的設備檔時，可以調用表中的過程，檢查是否該驅動器還在使用中。為方便使用者重新配置時修改該表，我們定義了宏DT進行自動處理(20900行)。

每個可能的主設備在dmap表中均佔有一行，用宏的形式寫出。有些必需的設備，巨集中的enable參數置為1。如果驅動器還未準備或者已經移走，enable參數則置為0。對於可在文件include/minix/config.h中配置的設備，其對應項使用它的允許宏，例如20920行的宏ENABLE\_WIN。

5.7.2 表的管理

與塊、i-節點、超級塊等表結構相關的是包含了管理這些表的過程的檔。這些過程多次為檔案系統的其他部分調用，構成了表和檔案系統之間的主要介面。因此，我們應該先從表的管理開始來分析檔案系統的代碼。

塊管理

塊快取記憶體是通過檔cache.h中的過程進行管理的。這個檔包含有9個過程，如圖5-34所示。第一個過程get\_block(21027行)是檔案系統獲取資料塊的標準方式。當檔案系統過程需要讀取使用者資料塊、目錄塊、超級塊或其他任何塊時，它調用get\_block。在調用時，需要標明設備號和塊號。

圖5-34 用於塊管理的過程。

get\_block被調用時，它首先檢查塊快取記憶體，看看請求的塊是否在塊快取記憶體中。若在，返回其指針。否則必須把它讀入塊快取記憶體。塊快取記憶體中的塊用NR\_BUF\_HASH個連結錶鏈接在一起。NR\_BUF\_HASH和NR\_BUFS都是可調的參數，後者指定了塊快取記憶體的大小。這兩個參數都放在文件include/minix/config.h中。在本節後面部分我們會提到如何優化塊快取記憶體和雜湊表的大小。HASH\_MASK等於NR\_BUF\_HUSH-1，如果有256個雜湊表，則HASH\_MASK為255。因此每條鏈所有塊的最後8位元都是相同的字串，分別為00000000、00000001、......或11111111。

一般時候，get\_block都首先要在雜湊鏈中查找塊，但有一種情況比較特殊：在讀取稀疏檔的一個孔時，不必進行查找。這就是我們在21055行進行檢查的原因。下面的兩行將bp設置為指向某條鏈的開始，如果所查找的塊在快取記憶體中，它必然在這條鏈中，鏈的下標是HASH\_MASK與塊號相與的結果。接下來一行迴圈查找這條鏈，檢查是否能找到所請求的塊。若找到，並且該塊未被使用，則從LRU鏈中刪除之，若還在使用，則它根本不會在LRU鏈中。在21063行，將找到的塊的指標返回給調用進程。

如果該塊不在雜湊鏈中，它也就不在快取記憶體中。因此這時從LRU鏈中取出最久未使用塊，將其緩衝區從原雜湊鏈中刪除，因為它將獲得一個新的塊號，從而屬於不同的雜湊鏈。在從原雜湊鏈中刪除時，若該塊修改過，則在21095行把它寫回磁片。用flushall調用可以把同一設備上所有其他修改過的塊都寫到磁片中。當前正在使用的塊並不換出記憶體，事實上他們不在LRU鏈中。可是我們很難判斷一個塊是否還將使用，通常情況下在使用完成後，每個塊立即用put\_block釋放。

一旦分配了緩衝區，所有域，包括b-dev，都使用新參數進行修改(21099行至21104行)，然後把該塊從磁片上讀入。可是有兩種情況我們並不需要讀入磁片塊。當get\_block以參數only\_search調用時，表明這是預讀。在預讀時，查找一個可用的緩衝區，必要時需將原有內容寫回磁片，為找到的緩衝區賦予一個新的塊號，但是其b\_dev域置為NO\_DEV，表明該塊中還沒有有效的資料。在討論rw\_scattered函數時，我們會看到它的使用。only\_search也可以用來表徵需要重寫整個塊，這時沒有必要先把該塊讀入。在這兩種情況下，緩衝區中的參數都被修改，但是忽略讀磁片過程(21107行至21111行)。在讀入新塊後，get\_block將其指標返回給調用進程。

假設檔案系統在查找檔案名時臨時需要一個目錄塊，它調用get\_block請求該目錄塊，在找到檔案名後，調用put\_block把該塊返回到快取記憶體中(21119行)，其緩衝區可以用於滿足隨後的一個不同塊的需要。

put\_block把新返回的塊插入到LRU鏈中，在某些情況下，需要把資料寫回磁片。第21144行，根據block\_type參數判斷把該塊放在LRU鏈的前面還是後面。block\_type是調用進程提供的一個標誌，它給出了塊類型。最近不久可能要用到的塊放在LRU鏈的尾部，以便它在LRU鏈中可以保存一段時間，而最近很可能不會用到的塊放在LRU鏈的前端，這樣它的緩衝區可以重新分配。目前只有超級塊按照後面一種方式處理。

在塊被放到LRU鏈後，還要檢查該塊是否應立即寫回磁片(21172行和21173行)。在標準配置下，只有超級塊標記為立即寫回，而超級塊只有在系統初始化，改變RAM盤大小時才被修改，需要立即寫回。這種情況下，修改後的超級塊寫到RAM盤中，而它不大可能需要再次讀入，因此立即寫回很少使用。但是，用戶可以修改include/minix/config.h檔中的ROBUST巨集，使得i-節點、目錄塊以及關係到檔案系統本身正確運行的其他塊標記為立即寫回。

隨著檔的擴大，需要不時地給它分配區段以保存新資料。過程alloc\_zone(21180行)可用于分配新區段。它在區段點陣圖中查找空閒區段。如果要分配的區段是檔的第一個區段，那麼可以使用超級塊中指向設備上第一個可用區段的s\_zsearch域，沒有必要查找點陣圖。否則，為了使檔的各區段放在一起，需要在區段點陣圖中查找與檔最後一個區段相鄰的區段，這可以通過從檔最後一個區段開始搜索點陣圖實現(21203行)。點陣圖中的位號與區段號之間的映射在21215行處理，第1位元對應於第一個資料區段。

刪除檔時，需要把它的區段返回到空閒區段點陣圖中。free\_zone(21222行)用於回收不再使用的區段，它以區段點陣圖和位號為參數，調用free\_bit。free\_bit也用於回收空閒i-節點，但是後者以i-節點點陣圖為第一個參數。

管理快取記憶體需要讀寫塊，為了提供一個簡單的磁片介面，MINIX中定義了rw\_block過程(21243行)。rw\_block讀或寫一個塊。類似地，rw\_inode用於讀寫i-節點。

本檔中下一個過程是invalidate(21280行)。比如在卸下磁片時，調用此過程以便從快取記憶體中刪除屬於某個剛卸下的檔案系統的所有塊。如果未調用該過程，則重新使用該設備(用另外一張軟碟)時，檔案系統可能把原來的塊當成新塊使用。

系統調用SYNC調用flush\_all(21295行)刷新屬於指定設備並修改過的緩衝區，將它們全部寫回設備。每次卸下一個設備時，都需調用flash\_all。執行時，它將塊快取記憶體看成線性陣列，這樣，所有修改過的緩衝區都可以找到，即使是那些當前正在使用，不在LRU鏈中的緩衝區也可以找到。在快取記憶體中的所有緩衝區都掃描一遍，那些屬於待刷新的設備以及需要重新寫回的緩衝區則被加到dirty指標陣列中。這個陣列用static聲明，所以不會保存在棧中，之後，該陣列被傳給rw\_scattered函數。

rw\_scattered函數(21313行)以設備標誌符、緩衝區指標陣列的指標、陣列大小和一個讀寫標誌位元為參數。它首先對緩衝區指標陣列按塊號排序以提高讀寫操作的效率。只有上面提到的flushall函數以WRITING標誌調用rw\_scattered。這種情況下，塊號的來源很容易理解，他們實際上是一些修改過的資料緩衝區。唯一調用rw\_scattered進行讀操作的是檔read.c中的rahead函數。現在，我們只要知道：在調用rw\_scattered之前，已經以預取方式多次調用過get\_block，因此預留了一組緩衝區；這些緩衝區中含有塊號，但沒有設備參數，不過這不要緊，因為在調用rw\_scattered時會提供該參數。

設備驅動程式對rw\_scattered讀請求的處理方式與寫請求相比有很大的不同。寫多個塊的請求必定全部按請求處理，然而對讀取多個塊的請求，不同的驅動程式可能進行不同的處理，這要取決於如何才能最有效。rahead在調用過程rw\_scatterred時往往伴隨著對一簇塊的請求，而這些塊可能實際上並不需要，所以最好的回應是盡可能多地獲取那些很容易就能讀取的塊，而不是在設備上四處尋道來讀取所有塊，因為這可能花費大量的尋道時間。例如，軟碟驅動程式在磁軌邊界處停止，而許多其他驅動程式唯讀取連續的塊。當讀操作完成後，rw\_scattered過程標誌讀入的塊，在相應緩衝區中填入設備號。

圖5-34中最後一個函數是rm\_lru(21387行)。這個函數從LRU鏈中刪除一個塊，它只由本檔中的get\_block調用，因而聲明為PRIVATE，使得它對本檔外的過程不可見。

在結束塊快取記憶體的分析之前，我們談談如何優化之。NR\_BUF\_HASH必須是2的冪。如果它大於NR\_BUFS，那麼雜湊鏈的平均長度將不足1。若記憶體可以保存大量的緩衝區，則必然有足夠的空間來容納大量的雜湊鏈，因此通常把NR\_BUF\_HASH選為大於NR\_BUFS的最小的2的冪。本文定義了512個塊和1024個雜湊表。最佳大小取決於系統的使用情況。根據經驗，我們發現將緩衝區數目增加到1024以上，在重新編譯MINIX時並不能改進系統性能，因此，1024個緩衝區顯然足以容納在整個編譯過程中用到的二進位檔案。對於其他一些作業，較小的緩衝區可能就足夠了，但是也有可能使用較大的緩衝區能夠提高系統性能。

本書所附CD\_ROM中，MINIX系統的二進位檔案是使用一個小得多的塊快取記憶體編譯而成的，這樣它可以在更多的主機上運行。我們希望生成一個可以在只有2MB記憶體的系統上安裝的MINIX發行版本本，而用1024個塊快取記憶體編譯而成的系統需要多於2MB的RAM記憶體。這個二進位檔案中還包含有所有可能的硬碟驅動程式以及其他在特殊安裝時不會用到的驅動程式。大多數用戶寧願在安裝後編輯include/minix/config.h檔，重新編譯該系統，從中刪除不必要的驅動程式，並盡可能增加塊快取記憶體的數量。

談到塊快取記憶體，我們需要指出：因為在16位元的Intel處理器上記憶體段最大長度為64KB，這使得在這些機器上不可能使用很大的快取記憶體。我們可以設定檔系統使它將RAM盤作為輔助快取記憶體，存放所有調出主快取記憶體的塊。這種輔助快取記憶體在32位元的Intel系統中並不需要，因此我們不加討論；若允許的話，一個大的主快取記憶體將帶來很高的性能，然而對於那些檔案系統的虛擬位址空間不足以存放很多主快取記憶體的機器(例如286)來說，輔助快取記憶體非常有效。輔助快取記憶體要比常規RAM盤的性能優越。一個快取記憶體，如果它所包含的資料至少被用到一次，並且這個快取記憶體非常大，那麼它將極大地提高系統的性能。在這裡，什麼才是"足夠大"並不能在事先確定，我們只能試著增大快取記憶體的數量，觀察系統性能是否確實得到了改進，才能確定快取記憶體是否足夠大了。time命令測量運行一個程式所耗費的時間，可以説明我們對系統進行優化。

i-節點管理

塊快取記憶體並不是唯一需要支援過程的表，i-節點表同樣需要。很多管理i-節點表的過程與塊管理過程相類似，他們都列在圖5-35中。

圖5-35 用於i-節點管理的過程。

get\_inode過程(21534行)與get\_block相似。當檔案系統任何部分需要一個i-節點時，它調用get\_inode過程。get\_inode首先搜索inode表，判斷所請求的i-節點是否在記憶體中。若在，則將其使用計數器加1，並且返回其指針。這一搜索過程包含在21546行至21556行的代碼之中。如果i-節點不在記憶體中，則調用rw\_inode將它讀入。

當一個過程使用完i-節點之後，它調用put\_inode過程(21578行)把i-節點返回。put\_inode將i-節點的使用計數器i-count減1。如果i-count減到0，表明檔不再使用，此時i-節點可以從inode表中刪除，若它被修改過，則需要寫回磁片。

如果i-節點的i\_link域為0，表明沒有任何目錄項指向相應檔，因此它的所有區段均可被釋放。注意使用計數器為0和連結數為0是完全不同的兩碼事，他們的原因不同，所導致的結果也不同。如果i-節點用於管道，這時儘管其連結數目非0，我們也必須釋放管道的所有區段。這常常發生在讀進程釋放它所讀取的管道時，只為一個進程保留管道顯然是件荒謬的事。

在創建一個新檔時，必須調用alloc\_inode(21605行)為其分配i-節點。MINIX允許設備按唯讀模式安裝，因此要檢查設備的超級塊確保其可寫。對i-節點的處理和對區段的處理不同，系統儘量使一個檔的各個區段相鄰，而i-節點則無此必要。為了節省查找i-節點點陣圖的時間，我們可以利用超級塊中記錄第一個空閒i-節點的域。

在獲得某個i-節點後，alloc\_inode調用get\_inode過程將它讀入記憶體的inode表中。接著初始化這個i-節點的各個域。其中一部分初始化工作在21641行至21648行完成，而另一部分則調用wipe\_inode過程(21664行)完成。我們之所以這樣分工，是因為在檔案系統的其他地方還要使用wipe\_inode來清除i-節點中的某些域(不是所有域)。

刪除檔時，調用free\_inode過程(21684行)釋放它的i-節點。這個過程只是將i-節點點陣圖中的相應位置0，並且修改超級塊中記錄第一個空閒i-節點的域。

下面一個函數update\_times(21704行)從系統時鐘獲取時間，並且修改時間域。update\_times函數還在STAT和FSTAT兩個系統調用中使用，因而用PUBLIC聲明。

rw\_inode過程(21731行)與rw\_block相似，它的任務是從磁片上取得一個i-節點。其執行過程如下：

1. 計算包含所需i-節點的塊。

2. 調用get\_block讀入該塊。

3. 從該塊中提取i-節點並把它拷貝到inode表中。

4. 調用put\_block返回該塊。

rw\_inode過程實際上比我們上面列出的更複雜，因此需要一些附加函數。首先，獲取當前時間代價很大，因此在需要修改i-節點的時間域時，若i-節點在記憶體中，則只設置它的i-update域作為標誌。在i-節點要寫回磁片時如果發現i-update域非0，這時才調用update\_times過程。

其次，MINIX的開發歷史也增加了複雜性：原先V1版本檔案系統在磁片上的i-節點具有與V2版本不同的結構。函數old\_icopy(21774行)和new\_icopy(21821行)實現結構的轉換。前者在記憶體中的i-節點和V1檔案系統使用的i-節點之間進行轉換，後者在記憶體中的i-節點和V2檔案系統使用的i-節點之間進行轉換。這兩個函數僅在本檔中調用，所以聲明為PRIVATE。每個函數均處理兩個方向的轉換(從磁片到記憶體和從記憶體到磁片)。MINIX也在與Intel處理器位元組順序不同的一些系統上實現，每種實現都在磁片上使用本地位元組順序。超級塊中的sp->native域指出了使用的位元組順序。必要時，old\_icopy和new\_icopy都要調用conv2和conv4來交換位元組順序。

dup\_inode(21865行)僅僅增加i-節點的使用計數器，在打開一個已經打開的檔時被調用。第二次打開檔時，並不需要把檔的i-節點從磁片上讀入。

超級塊的管理

文件super.c中包含了管理超級塊和點陣圖的過程。這個檔中有5個過程，列於圖5-36中。

圖5-36 用於管理超級塊和點陣圖的過程。

當需要一個i-節點或區段時，調用alloc\_inode或alloc\_zone。如我們在前面所述，這兩個過程都調用alloc\_bit(21926行)搜索相關點陣圖。這個搜索過程有三個嵌套迴圈：

1. 最外層，對點陣圖的所有塊的迴圈。

2. 中間層，對塊的所有字的迴圈。

3. 最內層，對字的每一個位的迴圈。

中間層迴圈檢查當前字各位是否全1。如全1，則這個字中沒有空閒的i-節點或區段，這時接著檢查下一個字，若找到一個不是全1的字，則進入內層迴圈，查找空閒位(即0)。如果搜索完所有塊後都沒有找到0位，表明沒有空閒i-節點或區段，因此返回NO\_BIT碼(0)。這樣的搜索過程需要很多處理器時間，但是使用超級塊中指向第一個空閒i-節點或第一個空閒區段的域，把它作為origin參數傳給alloc\_bit過程，可以大大縮短搜索時間。

釋放位要比分配位簡單得多，它不需要搜索點陣圖。free\_bit過程(22003行)首先計算哪一個點陣圖塊包含了要釋放的位，接著調用get\_block將這個點陣圖塊讀入記憶體，相應位置0，最後調用put\_block把該塊寫回磁片。

下面一個過程get\_super(22047行)用於在超級塊表中搜索特定設備。例如，在安裝一個檔案系統時，需要檢查它是否已經安裝，這時可以調用get\_super查找該檔案系統的設備，如果沒有找到，則說明相應的檔案系統還未安裝。

下面的函數mounted(22067行)僅在關閉塊設備時調用。正常情況下，在設備關閉時，它的所有在快取記憶體中的資料均被丟棄。可是如果這個設備恰巧是被安裝設備，丟棄在快取記憶體中的資料是不可取的。mounted函式呼叫時，以一個指向設備i-節點的指標為參數，若該設備是根設備或是被安裝的設備，該函數返回真。

最後，我們看看read\_super過程(22088行)。它與rw\_block和rw\_inode有些類似，但只在讀超級塊時調用。系統的正常操作中並不需要寫超級塊。read\_super檢查所讀出的檔案系統的版本號，必要時進行相應的轉換。這樣儘管在磁片上超級塊結構不同，但是記憶體中超級塊的拷貝都是標準結構。

文件描述符的管理

MINIX中還有管理檔描述符和filp表的過程(見圖5-33)，這些過程放在檔filedes.c中。創建或打開檔時，需要分配一個空閒的檔描述符和一個空閒的filp項，這時可調用get\_fd過程(22216行)。但是分配到的檔描述符和filp項並不標誌為使用，因為在確信CREAT和OPEN成功之前，需要進行許多檢查。

get\_filp過程(22263行)用於檢查檔描述符是否在合適的範圍之內，若在，返回它的filp指針。

本檔中的最後一個過程是find\_filp(22277行)。當進程正向一個斷裂管道(即這個管道未被任何進程打開讀取)寫資料時，調用該過程。find\_filp搜索filp表，查找潛在的讀進程。若未找到，說明這個管道是斷裂管道，寫入失敗。

文件鎖

POSIX記錄鎖函數如圖5-37所示。在FCNTL系統調用中指定F\_SETLK或F\_SETLKW請求，可以在檔某部分加上讀寫鎖，或者只加寫鎖。利用F\_GETLK請求，我們很容易判斷檔的某一部分是否有鎖。

圖5-37 POSIX的建議性記錄鎖操作，這些操作通過系統調用FCNTL請求。

文件lock.c中只有兩個函數。lock\_op(22319行)為FCNTL所調用，調用時給出圖5-37中的一個操作碼。lock\_op檢查指定的鎖範圍是否有效，在設置鎖時，不能與現有的鎖衝突，在清除鎖時，現有的鎖不能分成兩個部分。清除任何一個鎖後，調用本檔中的另一個函數lock\_revive(22463行)，這個函數喚醒所有因等待鎖而阻塞的進程。我們之所以採取這一折衷方法，是因為要指出哪些進程正在等待釋放某個特定的鎖，需要編寫額外的代碼。那些還在等待加鎖檔的進程被喚醒後，會再次阻塞。這種策略假定鎖並不經常使用。如果要在MINIX系統上建立一個大型的多使用者資料庫，那麼就需要重新實現檔鎖這一部分。

加鎖檔被關閉時，也要調用lock\_revive過程。這種情況是可能發生的，例如，進程在結束使用加鎖檔之前被終止。

5.7.3 主程序

檔案系統的主迴圈包含在檔main.c中，其代碼從22537行開始。在結構上，檔案系統的主迴圈和記憶體管理器以及I/O任務的主迴圈非常相似。它調用get\_work等待下一條消息請求(除非先前在管道或終端上被掛起的進程如今能夠執行)。它還設置全域變數who為調用者的進程表槽口號，同時設置另一個全域變數fs\_call為待執行的系統調用的編號。

一旦控制回到主迴圈，設置三個標誌：fp指標指向調用進程的進程表項(fproc)，super\_user給出了調用進程是否是超級塊使用者，dont\_reply初始時設置為假。然後是關鍵部分，即執行系統調用的過程。用fs\_call作為過程指標陣列call\_vect的下標選定被調用的過程。

當控制重新回到主迴圈之後，如果dont-reply被設置，則禁止發送回答信號(例如，進程因試圖從空管道中讀取資料而阻塞)。否則送回一個回答消息。主迴圈中最後的一條語句檢測檔是否被順序讀取，並且在實際需要前把下一塊載入快取記憶體中，從而改進系統性能。

get\_work過程(22572行)檢查是否有以前阻塞的進程已醒，這些進程的優先順序高於新收到的消息。只有當檔案系統沒有內部操作時，才調用系統內核獲得消息(22598行)。

在系統調用完成之後，不管成功與否，都要用reply函數(22608行)向調用進程發送回答消息。調用進程有可能已被信號終止，因此從內核中返回的狀態碼被忽略，在這種情況下，不進行任何處理。

初始化函數

文件main.c的剩下部分由一些僅用於系統初啟的函數組成。在檔案系統進入主迴圈之前，它調用fs\_init(22625行)進行初始化。而fs\_init又調用幾個其他函數，初始化塊快取記憶體、獲得啟動參數、必要時裝載RAM盤以及載入根設備的超級塊等等。接下來，fs\_init為所有任務和伺服器初始化進程表的檔案系統部分(22643行至22654行)。最後，fs\_init對一些重要常數進行檢測，以確保其合理性。為了供ps程式使用，還要向記憶體任務發送一條消息，其中包含進程表的檔案系統部分的位址。

fs\_init調用的第一個函數是buf\_pool(22679行)，它建立塊快取記憶體使用的連結表。圖5-31顯示了塊快取記憶體的正常狀態，其中，所有塊都同時連結到LRU鏈和一個雜湊鏈中。為了更好地理解，我們來看看圖5-31的情況是如何發生的。在快取記憶體用buf\_pool過程初始化後，所有的緩衝區均在LRU鏈中，並且這些緩衝區都用0號雜湊鏈連結起來，如圖5-38(a)所示。當一個緩衝區被請求並在使用時，情況如5-38(b)，這時，我們看到一個塊從LRU鏈中刪除，放在另一個不同的雜湊鏈中。正常情況下，塊立即被釋放，返回到LRU鏈中。圖5-38(c)中顯示了該塊返回到LRU鏈後的情況。儘管此時沒有被使用，但是如果需要它還可以被存取，並提供相同的資料內容，所以把它保留在雜湊鏈中。在系統運行了一段時間後，差不多所有塊都被使用過，隨機地分佈在不同的雜湊鏈中。這時，LRU鏈看上去就象圖5-31一樣。

圖5-38 塊快取記憶體的初始化。 (a)在使用任何緩衝區之前。 (b)在請求一塊後。

(c)在該塊被釋放後。

接下來一個函數是get\_boot\_parameters(22706行)。它送一條消息到系統任務，向它請求啟動參數的一份拷貝，這些參數用在下面的load\_ram函數(22722行)中，load\_ram為RAM盤分配空間。如果啟動參數標明

rootdev=ram

從啟動區開始，根設備檔案系統被逐塊地從設備ramimagedev拷貝到RAM盤中，拷貝時對檔案系統的資料結構不作任何解釋。若ramsize啟動參數較根設備檔案系統小，則擴大RAM盤以容納之。如果ramsize大於根設備檔案系統的長度，則分配指定大小的空間，並調整RAM盤檔案系統，讓它使用所有這些空間(22819行至22825行)。22825行的put\_block調用是檔案系統中唯一對超級塊進行寫操作的調用。

若指定的ramsize非零，load\_ram為空RAM盤分配空間。這時，因為沒有拷貝檔案系統結構，該RAM設備不能用作檔案系統，除非它已使用mkfs命令進行初始化。除此之外，如果編譯的檔案系統提供相應支援，這樣的RAM盤還可以用作輔助快取記憶體。

文件main.c中的最後一個函數是load\_super(22832行)。它初始化超級塊表，並把根設備的超級塊讀入記憶體。

5.7.4 對單個檔的操作

本節我們逐個研究對檔的系統調用(同目錄相區別)。我們從打開、創建和關閉檔開始，然後詳細討論檔的讀寫機制，在這之後，我們再來分析管道及其操作與檔有什麼不同。

創建、打開和關閉文件

文件open.c中包含有6個系統調用的代碼，這6個系統調用是：CREAT、OPEN、MKNOD、MKDIR、CLOSE和LSEEK。我們首先將CREAT和OPEN結合起來考慮，然後逐個討論其他的系統調用。

在老版本的UNIX中，CREAT和OPEN調用用於不同的目的。打開一個並不存在的檔會出錯，而一個新檔必須用CREAT創建，CREAT也可以把一個存在檔的長度刪除為0。可是POSIX系統不再需要兩種不同的調用。在POSIX中，OPEN調用可以用於創建一個新檔和截斷一個老檔，因此CREAT調用實現的功能只是OPEN的一個子集，僅用於與原有程式相相容。處理CREAT和OPEN的過程分別是do\_creat(22937行)和do\_open(22951行)(檔案系統使用了和記憶體管理器相同的慣例，即系統調用xxx由過程do\_xxx執行)。打開或創建一個檔包括三步：

1. 找到i-節點(創建檔時需要進行分配和初始化)。

2. 找到或創建目錄項。

3. 為檔建立並返回一個描述符。

CREAT和OPEN都要做兩件事：獲取檔案名、並調用common\_open執行這兩個調用的共同操作。

common\_open過程(22975行)首先驗證空閒的檔描述符及空閒的filp項是否存在。若調用函數指定創建新檔(調用時設置O\_CREAT位)，則執行22998行的new\_node過程。如果目錄項存在，new\_node將返回指向現有i-節點的指標，否則創建目錄項和i-節點。在無法創建i-節點時，設置全域變數err\_code。錯誤碼並不總是表示錯誤，如果new\_node發現一個已有的檔，返回的錯誤碼表明檔存在，但是這種錯誤是可以接受的(23001行)。若O\_CREAT位沒有設置，需用另一種方法來搜索i-節點，即我們後面要討論的文件path.c中的eat\_path函數。這時要注意的是：如果沒有找到i-節點或者創建i-節點失敗，在執行到23010行前common\_open出錯並終止。否則繼續執行，對檔描述符賦值，並在filp表中申請一項。之後，如果新檔剛被創建，就跳過23017行到23094行之間的代碼。

對於現有檔，檔案系統需要檢查檔的類型、模式等，從而確定該檔是否能打開。第23018行的forbidden調用首先對rwx位進行一般性檢查。若是正規檔，並且調用common\_open時設置了O\_TRUNC位，則檔長度截為0，再次調用forbidden(23024行)以確保文件可寫。如果許可權允許，調用wipe\_inode和rw\_inode以重新初始化i-節點，並將它寫回磁片。其他檔案類型(目錄、設備檔、具名管道)也需進行適當的檢測。對於設備，在23053行(使用dmap結構)調用適當的過程打開它。若是具名管道，則調用pipe\_open(23060行)，並進行和管道有關的各種檢查。

common\_open的代碼，同檔案系統的其他許多過程一樣，含有大量檢查各種錯誤和非法組合的代碼，這些代碼對於設計一個無錯的、健壯的檔案系統是必不可少的。如果出現錯誤，以前分配的檔描述符和filp項被收回，i-節點被釋放(23089行至23101行)。在這種情況下，common\_open過程返回一個負數以表示出錯。若運行成功，返回一個正數，即檔描述符。

現在我們可以更詳細地討論過程new\_node(23111行)了。new\_node負責分配i-節點，並為CREAT和OPEN把路徑名加到檔案系統中。它還用在MKNOD及MKDIR兩個調用中，這兩個調用我們以後會介紹。第23128行的語句分析路徑名(即逐個部分查詢路徑)，直到最後一個目錄；後面第三行的advance調用檢查最後一個部分能否被打開。

例如，在調用

fd=creat("/usr/ast/foobar"，0755)；

時，last\_dir把/usr/ast的i-節點裝入記憶體i-節點表中，返回其指針。如果檔不存在，不久我們就要用到這個i-節點，以便把foobar加入目錄中。所有其他增加或刪除檔的系統調用，也都先用last\_dir打開路徑的最後一個目錄。

假設new\_node發現檔不存在，它就調用23134行的alloc\_inode過程分配並載入一個新的i-節點，返回指向該節點的指標。若磁片上沒有空閒i-節點，new\_node執行失敗，返回NIL\_INODE。

如果能夠分配i-節點，則繼續執行23144行的語句，填入某些域，把它寫回磁片，然後在最後目錄下加入檔案名(23149行)。這裡我們再次看到檔案系統需要經常地檢查錯誤，一旦遇到錯誤，精心地釋放它申請到的所有資源，比如i-節點和塊。在出現如i-節點耗盡等情況時，如果不取消當前調用的影響，給調用進程返回錯誤碼，而是讓MINIX陷於混亂，那麼我們的檔案系統要簡單得多。

我們前面也說到過，對於管道需要作特殊處理。如果一個管道沒有讀/寫對，pipe\_open（23176行）過程將掛起調用進程，否則它調用release函數在進程表中查找因該管道而阻塞的進程，並將找到的進程喚醒。

MKNOD調用在過程do\_mknod(23205行)中處理，這一過程與do\_creat類似，只是前者僅僅創建i-節點並為其分配一個目錄項。事實上，大多數工作都是由23217行的new\_node過程完成的。如果i-節點已存在，則返回一個錯誤碼。這個錯誤碼與common\_open中調用new\_node得到的錯誤碼相同。在後面一種情況下，這個錯誤碼可以接受，但是對於前者，錯誤碼被傳遞給調用進程，以便作相應處理。在此，我們不再逐行分析了。

MKDIR調用在do\_mkdir函數(23226行)中處理。同我們這兒討論的其他系統調用一樣，new\_node仍然起了重要作用。與檔不同的是，目錄通常有連結。此外，目錄不可能為空，在創建時，它至少包含兩個目錄項"."和".."，分別指向目前的目錄及其父目錄。我們將檔連結數限制為不超過LINK\_MAX(對於標準的MINIX檔案系統，LINK\_MAX在檔include/limits.h中定義為127)。因為子目錄對父目錄的引用也是到父目錄的一個連結，do\_mkdir首先檢查是否可以在父目錄中創建另外一次連結(23240行)。若可以，則調用new\_node過程。new\_node成功執行後，接著創建"."和".."兩個目錄項(23261行和23262行)。這一過程非常直觀，但可能出錯(例如磁片已滿)，因此，我們提供很多代碼，以便在操作無法完成時，恢復到進程的初始狀態。

關閉檔比打開檔要容易得多，它主要是在do\_close過程(23286行)中完成的。管道和設備檔需要特別注意。但是對於正規檔，我們只需要減少filp計數器並檢查該計數器是否為0。若為0，則調用put\_inode過程回收i-節點。最後刪除所有的鎖，喚醒所有因等待這個檔的鎖被釋放而掛起的進程。

請注意，回收一個i-節點意味著它在inode表中的計數器減1，這樣它將最終從inode表中刪除。這個操作與i-節點的釋放(即在點陣圖中設置一位表明這個i-節點可用)無關。只有在檔從所有目錄中刪除時，才要釋放i-節點。

文件open.c的最後一個過程是do\_lseek(23367行)。進行檔讀/寫定位時，調用這個過程設置新的檔位置。第23394行禁止預讀；順序存取檔時，不能設置檔位置。

讀文件

檔一旦被打開，就可以進行讀寫。許多函數既可用於讀檔，又可以用於寫檔。這些函數在檔read.c中。我們首先分析read.c，然後再來分析write.c中專門用於寫操作的代碼。讀寫操作有很大的不同，但是也有很多相似之處，因此，do\_read過程(23434行)執行時，僅需在調用公共過程read\_write時設置READING標誌即可。在後面一節我們還會看到，do\_write的代碼也同樣簡單。

read\_write過程從23443行開始。在23459行至23462行之間有一些特殊的代碼，這些代碼在檔案系統把記憶體管理器載入使用者空間時執行。正常調用的處理從第23464行開始，之後是有效性檢查(例如，讀取一個隻為寫而打開的檔)以及變數的初始化。從字元設備檔中讀取資料並不經過塊快取記憶體，因此在第23498行將它們篩選出來。

從23507行到23518行的檢測只適用於寫操作，它處理檔長度可能超出設備容量的情況，或者超出檔案結尾寫資料從而在檔中產生空洞的異常。我們在MINIX概述一節中說過，一個區段中使用多個塊會導致一些問題，需要特別處理。在這裡，我們還需要檢查管道。

讀檔的關鍵，至少對普通檔來說，是從23530行開始的迴圈。這個迴圈把讀請求分成若干小塊，每個小塊都在一個單獨的磁片塊中，它從當前位置開始，直到滿足下面的條件之一：

1. 所有位元組都讀完。

2. 遇到一個塊邊界。

3. 讀到文件末尾。

這些規則表明一個小塊不可能用到兩個磁片塊。圖5-39中的三個例子說明如何確定小塊的大小，圖中小塊分別為6、2和1個位元組。實際計算由23632行到23641行的代碼完成。

圖5-39 三個例子表明在10位元組檔中如何決定第一個小塊的長度，塊大小為8個位元組，請求的位元組數為6，小塊用陰影顯示。

事實上，讀取小塊是在rw\_chunk中完成的。在rw\_chunk執行後，各計數器及指標相應增加，接著進行下一個迴圈，當迴圈結束後，檔位置及一些其他變數(例如管道指標)也要作修改。

最後，如果要求預讀，則將要讀的i-節點和位置存儲在全域變數中，以便在向用戶送回回應訊息後，檔案系統能夠接著讀取下一個磁片塊。大多數情況下，檔案系統被阻塞，等待下一個磁片塊，這時，使用者程式可以處理剛剛收到的資料。這種安排使得計算和I/O併發執行，從而大大地提高了系統性能。

rw\_chunk過程(23613行)以i-節點和檔位置為參數，把他們轉換成物理磁片塊號，然後請求傳送該塊(或其中一部分)到用戶空間中。相對檔位置與物理磁片位址的映射是在read\_map函數中進行的。對於普通檔，23637行和23638行的變數b和dev分別包含了物理塊號及設備號。在23660行的get\_block調用查找塊，必要時將它讀入記憶體。

一旦我們獲得了該塊的指標，就可以調用23670行的函數sys\_copy將塊中的所需部分傳送到使用者空間中，隨後調用put\_block過程釋放該塊，以便在需要時，將該塊換出塊快取記憶體(在調用get\_block申請到某一塊後，該塊不會出現在LRU佇列中，如果塊頭中的計數器表明該塊還在使用，它也不會返回到LRU鏈中，因而可避免被換出。put\_block過程將計數器減1，在它減為0時，將這個塊放到LRU佇列中。)第23680行的代碼表明是否一次寫操作會填滿塊。然而，通過n傳遞給put\_block過程的值並不會影響如何將塊放入佇列中。所有塊一律放在LRU鏈的尾部。

read\_map過程(23689行)檢查i-節點，將邏輯檔位置轉換為物理塊號。對於靠近文件頭，在前面7個區段(即那些在i-節點中的區段)中的塊，通過簡單的計算便可知道要訪問哪個區段、哪個塊。而對於檔中更後面的塊，則需要讀取一個甚至多個間接塊。

讀取間接塊時，調用過程rd\_indir(23753行)。我們之所以將它作為一個單獨的過程，是因為資料在磁片上的存放格式不同，這主要視檔案系統的版本號及編寫的硬體而定。如果必要的話，在rd\_indir中進行繁瑣的轉換，使得檔案系統的其他部分只見到唯一一種格式的資料。

read\_ahead(23786行)將邏輯位置轉換成為物理塊號，調用get\_block確保塊在快取記憶體中(或將其調入)，之後立即回收該塊。read\_ahead對塊並不進行任何操作，只是增加它即將使用時，已調入記憶體的機會而已。

請注意，read\_ahead僅在main函數中調用。它並不作為READ系統調用的一部分而調用。我們還要意識到，read\_ahead 調用是在送迴響應之後進行的，所以，用戶可以接著進行其他操作，儘管檔案系統不得不等待讀取下一個磁片塊。

read\_ahead本身只是請求另一塊，它調用檔read.c中的最後一個函數rahead來完成該操作。rahead(23805行)的工作原理基於這樣的思想：如果稍多一點總比沒有強，那麼再多一點則更棒。因為磁片和其他的存放裝置在找第一塊時要花較長一段時間，而讀取相鄰塊所需時間則較短，稍加努力，我們就可以讀取更多的塊。預讀請求送給get\_block，後者申請塊快取記憶體以便一次接收多個塊。接著調用rw\_scattered。我們在前面已經討論過：當設備驅動程式被rw\_scattered調用時，各個驅動程式可以自由地回應那些能夠被高效地處理的請求。這聽起來很複雜，但對那些從磁片上讀取大量資料的應用，這種複雜的辦法可能顯著地加快速度。

圖5-40顯示了在讀取檔時用到的一些主要過程之間的關係，特別是他們之間的調用關係。

圖5-40 用於讀文件的一些過程。

寫文件

寫檔的代碼放在檔write.c中。寫檔與讀檔類似，do\_write過程(24025行)只是在調用read\_write時設置WRITING標誌。讀寫檔的主要不同在於寫檔需要分配新的磁片塊。write\_map(24036行)與read\_map相似，只不過它不是在i-節點及其間接塊中查找物理塊號，而是在其中添加新的物理塊號(精確地說，它添加的是區段號，而不是塊號)。

write\_map的代碼很長，它需要處理幾種不同情況。如果插入的區段靠近檔頭，我們只需要將它插在i-節點中(24058行)。

最糟糕的情況是，檔超過了一次間接塊所能處理的長度，這時需要使用二次間接塊。接下來，分配一個一次間接塊，並將其位址填入二次間接塊中。同讀檔一樣，這時要調用一個獨立的過程，wr\_indir。如果成功獲取二次間接塊，而這時磁片已滿，無法分配一次間接塊，就需要將前面獲得的二次間接塊返回，以免破壞點陣圖。

這時候，若我們僅僅讓系統陷於混亂並顯示嚴重故障資訊，那麼MINIX代碼就簡單許多。然而，從使用者觀點來看，在磁碟空間不足時，WRITE調用返回錯誤碼，比系統因此崩潰而造成檔案系統被破壞要好得多。

wr\_indir(24127行)調用conv2或conv4進行必要的資料轉換，並將新區段號寫入間接塊中。注意這個函數的名字，它同其他涉及讀寫操作的函數的名字一樣，字面含義並不一定很確切。真正往磁片中寫入資料是由管理塊快取記憶體的函數進行的。

文件write.c中的下一個過程是clear\_zone(24149行)，它負責清空突然出現在檔中部的某些塊。超出檔案結尾寫入資料時，就會出現這種情況。幸好，它並不經常發生。

若需要一個新塊，rw\_chunk程序呼叫new\_block(24190行)。圖5-41是順序檔擴大時的6個連續階段。在這個例子中，塊大小為1K，而區段大小為2K。

圖5-41 (a)-(f)塊(區段大小為2K，塊大小為1K)的連續分配。

第一次調用new\_block時，分配區段12(塊24及25)，接下來使用塊25，第三次調用時，分配區段20(塊40及41)等等。zero\_block過程(24243行)清空某個塊，刪除塊中以前的內容，我們上面的描述要比zero\_block的實際代碼長得多。

管道

在許多方面，管道和普通檔類似。這一節，我們主要討論他們之間的不同。我們要分析的代碼都在檔pipe.c中。

首先，創建管道的方式不同，管道並不由CREAT，而是由PIPE來創建。do\_pipe過程(24332行)處理PIPE調用，它為管道分配一個i-節點，並返回兩個檔描述符。管道為系統，而不是使用者所擁有，並放在指定的管道設備上(在檔include/minix/config.h中配置)。管道資料無需永久保存，所以可把RAM盤用作管道設備。

管道讀寫與檔的讀寫也略有不同，管道只有有限的容量。在管道已滿時，所有繼續向管道寫入資料的進程都被掛起。同樣，讀取空管道的進程也會被掛起。事實上，管道有兩個指針：當前位置指標(由讀進程使用)及長度指標(由寫進程使用)。這兩個指標決定讀寫管道資料的位置。

pipe\_check過程(24385行)進行各種檢查，以保證對管道的操作能夠完成，這些檢查都將導致調用進程被掛起。除此之外，pipe\_check還調用release過程，判斷能否喚醒那些因為沒有資料或有過多資料而被掛起的進程。對於正在睡眠的寫進程和讀進程，喚醒操作分別在24413行和24452行執行。在pipe\_check中，還要檢測往一個斷裂的管道(沒有讀進程)中寫資料的異常。

掛起一個進程是由suspend函數實現的(24463行)。該函數將調用參數保存在進程表中，並將dont\_reply標誌設為真，從而禁止檔案系統的回應訊息。

release過程(24490行)檢查是否某個因管道而掛起的進程可以繼續運行。如果找到這樣的進程，它調用revive設置一個標誌，以便於主迴圈處理。這個函數並不是系統調用，我們將它列在圖5-27(c)中，是因為它使用了消息傳遞機制。

文件pipe.c中的最後一個過程是do\_unpause(24560行)。在記憶體管理器試圖向某進程發送信號時，它必須檢查該進程是否正掛起在一個管道或者設備檔上(這種情況下，它必須用EINTR錯誤喚醒)。因為記憶體管理器並不能識別管道和設備檔，它向檔案系統發送一條消息詢問，這條消息在過程do\_unpause中處理。若進程被阻塞，do\_unpause將它喚醒。同revive一樣，do\_unpause有些類似於一個系統調用，儘管它並不是系統調用。

5.7.5 目錄和路徑

我們前面已經討論了檔的讀寫機制，下一步就來看看路徑名和目錄是如何處理的。

將路徑名轉換成i-節點

許多系統調用(例如OPEN、UNLINK和MOUNT)都以路徑名(即檔案名)作為參數。這些系統調用在開始調用前，大多數都要首先獲得指定檔的i-節點。路徑名如何轉換成i-節點是本小節研究的主要內容。在圖5-14中我們已經看過了大致的輪廓。

路徑名分析在檔path.c中進行。在path.c中，第一個過程是eat\_path(24727行)。該過程接受指向路徑名的指標，進行分析，並把它的i-節點調入記憶體，然後返回指向這個i-節點的指標。具體過程如下：它首先調用last\_dir獲得最後目錄的i-節點，隨後調用advance取得路徑的最後一個部分。如果搜索失敗，比如路徑中有一個目錄不存在，或者雖然存在但卻禁止搜索，那麼就返回NIL\_INODE。

路徑名可以是絕對路徑名或相對路徑名，可以含有任意多個部分，各部分之間以斜杠分隔。這些問題都在last\_dir函數(24754行)中處理。last\_dir首先檢查路徑名的第一個字元以判斷其為絕對路徑還是相對路徑(24771行)。對於絕對路徑，rip設置為指向根i-節點的指標，否則rip設置為指向當前工作目錄的指標。

現在，last\_dir已經知道路徑名以及指向某個目錄的i-節點，可以在這個i-節點中查找路徑的第一部分了。它進入24782行的迴圈，逐部分對路徑名進行分析。當到達尾部時，便返回指向最後目錄的指標。

get\_name實用過程(24813行)從字串中提取檔路徑名的各個部分。更有趣的是advance過程(24855行)，它以目錄指標和一個字串為參數，在目錄中查找該字串。如果找到，則返回指向相應i-節點的指標。在advance中還要仔細處理如何跨越至安裝的檔案系統中。

儘管advance處理字串查找。然而字串與目錄項的比較是在search\_dir(24936行)中實現的。search\_dir是檔案系統中唯一檢查目錄檔的過程，它含有兩個嵌套的迴圈：一個是針對目錄中各塊的迴圈，另一個是針對塊中各項的迴圈。在目錄中增加和刪除名字時也要調用search\_dir過程。圖5-42給出了用於查找路徑名的一些主要過程之間的關係。

圖5-42 用於查路徑名的一些過程。

安裝檔案系統

MOUNT和UMOUNT兩個系統調用影響著整個檔案系統。利用這兩個調用，不同次設備上的檔案系統可以連結在一起，形成一個無縫的命名樹。正如我們在圖5-32中看到的那樣，安裝是通過把被安裝檔案系統的根目錄i-節點及其超級塊讀入記憶體，並且設置超級塊中兩個指針實現的。其中一個指標指向被安裝至的i-節點，另一個指標指向被安裝檔案系統的根目錄i-節點。這兩個指標把不同的檔案系統連結在一起。

上述兩個指針是由檔mount.c中的do\_mount函數在25231行和25232行設置的。在指針設置前的兩頁代碼主要檢查在安裝檔案系統時可能出現的各種錯誤。其中的一些錯誤如下：

1. 給定的設備檔不是塊設備。

2. 設備檔是塊設備，但是已經安裝。

3. 被安裝檔案系統具有不正確的魔數。

4. 被安裝檔案系統無效(例如，沒有i-節點)。

5. 要被安裝至的檔不存在或者是設備檔。

6. 沒有空間以存放被安裝檔案系統的點陣圖。

7. 沒有空間以存放被安裝檔案系統的超級塊。

8. 沒有空間以存放被安裝檔案系統的根目錄i-節點。

或許我們不宜在此反復強調，然而在一個實用的作業系統中，確實需要有很大部分的代碼用於錯誤處理。如果某個使用者偶爾安裝一個錯誤的軟碟，例如每月一次，出現系統崩潰，檔被破壞，他肯定會懷疑系統的可靠性，因而責怪系統設計者，而不會從他自身找原因。

我們這裡可以引用湯瑪斯·愛迪生的一段論述。他認為"天才"是1%的靈感加99%的勤奮的結果。一個好的系統和一個平凡的系統的不同之處，並不是調度演算法是否優越，而在於是否考慮了所有的細節問題。

卸下檔案系統要比安裝容易得多，出錯的機會也要少。do\_unmount過程(25241行)處理卸載。卸載時，關鍵在於確保沒有進程在被卸下的檔案系統中具有打開檔或工作目錄。這個檢查過程是很直觀的：我們只需掃描整個i-節點表，檢查記憶體中是否有i-節點屬於要卸下的檔案系統(根目錄i-節點除外)。若有，UMOUNT調用失敗。

文件mount.c中的最後一個過程是name\_to\_dev(25299行)。name\_to\_dev以一個設備檔的路徑名作為參數，獲得它的i-節點並從中提取主設備號和次設備號。這些設備號存放在i-節點中原用於存放第一個區段的地方。該位置之所以能夠使用，是因為設備檔並沒有區段。

連結和解鏈檔

我們接下來分析檔link.c。link.c用於連結檔和解鏈檔。同do\_mount類似，do\_link(25434行)中幾乎所有代碼均用於錯誤檢查。在調用

link(file\_name，link\_name)；

時，可能發生的錯誤列舉如下：

1. file\_name不存在或者不能訪問。

2. file\_name已經有最大數目的連結。

3. file\_name是一個目錄(只有超級塊使用者才可以對目錄進行連結)。

4. link\_name已經存在。

5. file\_name和link\_name在不同的設備上。

若無錯誤發生，則創建一個新的目錄項，其檔案名為link\_name，而i-節點號為檔file\_name的i-節點號。在代碼中，name1對應於file\_name，而name2對應於link\_name。新目錄項實際上是由do\_link在25485行調用的search\_dir函數創建的。

解鏈可以刪除檔和目錄。系統調用UNLINK和RMDIR的工作都由do\_unlink(25504行)來完成。這裡，我們依然要進行各種檢查；檢測檔存在以及目錄不是安裝點的工作在do\_unlink的公共代碼中進行，然後根據所支援的系統調用的不同分別執行remove\_dir或unlink\_file。我們很快還要談到這些

文件link.c支援的另一個系統調用是RENAME。UNIX用戶所熟悉的shell命令mv最終調用了RENAME；其名稱還反映了這個系統調用的另一方面：它不僅可以在目錄中改變檔案名，還能夠將檔從一個目錄移至另一個目錄，這些都是自動完成的。具體工作由do\_rename(25563行)實現。在完成這個命令時，需要檢測一系列情況，其中有：

1. 原始檔案必須存在(25578行)。

2. 在目錄樹中，原路徑名不能是新路徑名上面的目錄(25596行至25613行)。

3. 原路徑名和新路徑名均不能是"."和".."(25618行)。

4. 他們的父目錄必須要在同一個設備上(25622行)。

5. 他們的父目錄均可寫、可查找，並且都在可寫設備上(25625行和25626行)。

6. 原檔案名和新檔案名都不能是檔案系統安裝至的目錄。

如果新檔案名已經存在，我們還要檢查另外一些情況，其中最重要的是，我們必須有權刪除以新檔案名為名字的檔。

在do\_rename的代碼中，有些例子可以說明我們是如何減少問題出現的可能性的。如果不首先刪除原來的檔，將檔案名換成一個已經存在的檔案名可能導致磁片滿，儘管最終不需要額外的空間，這就是25660行到25666行的代碼的目的。25680行是同樣道理，在同一目錄下創建新檔案名之前刪除原來的檔，以避免目錄申請額外的塊。然而，如果新檔案名和原檔名位於不同的目錄下，所考慮的與此毫不相關。在25685行，我們在刪除原檔案名前創建一個新檔案名(在不同目錄下)，因為從系統完整性的觀點來看，在系統崩潰時，有兩個檔案名同時指向一個i-節點，遠比有一個節點不被如何目錄項所指向的情況要好得多。在執行換名操作時空間耗盡的可能性很小，由此導致系統崩潰的可能性更小，但在這些情況下應付最壞情況並不耗費更多的資源。

文件link.c中剩下的函數為我們上面討論過的函數提供支援。此外，他們中的第一個函數truncate(25717行)還在檔案系統其他幾個地方被調用。它在i-節點中查找區段，釋放它所找到的所有區段，包括間接塊。remove\_dir(25777行)進行許多額外的檢測，確保目錄可被刪除，然後調用unlink\_file(25818行)。若沒有錯誤，目錄項被清空，i-節點中連結數減1。

5.7.6 其他系統調用

我們要討論的最後一組系統調用涉及狀態、目錄、保護、時間和其他服務等。

改變目錄和檔的狀態

文件stadir.c包含四個系統調用的代碼，他們是：CHDIR、CHROOT、STAT和FSTAT。在分析last\_dir時，我們看到路徑查找首先檢查第一個字元是否為分隔符號，根據結果，將指標指向工作目錄或根目錄。

從一個工作目錄(或根目錄)切換到另一個工作目錄(或根目錄)，只需要修改調用進程的進程表中的相應指標。這一修改在do\_chdir(25924行)和do\_chroot(25963行)中進行。這兩個過程都做一些必要的檢查，然後調用change(25978行)打開新目錄以取代原來的目錄。

在使用者進程調用CHDIR時，並不執行do\_chdir中的25935行至25951行的代碼。它是專門為記憶體管理器在處理EXEC調用時改變使用者目錄而編寫的。若某使用者想在他的工作目錄下執行一個檔，比如a.out，對記憶體管理器來說，切換到這個工作目錄要比指出它的位置容易得多。

本檔中的另外兩個系統調用，STAT和PSTAT，本質上是相同的，只是指定檔的方式不同：前者給出了路徑名，而後者給出的是打開檔的檔描述符。最上層的過程do\_stat(26014行)和do\_fstat(26035行)都調用stat\_inode。在調用之前，do\_stat還要打開檔，獲得其i-節點。這樣，do\_stat和do\_fstat都向stat\_inode傳遞一個i-節點指標。

stat\_inode過程(26051行)從i-節點中提取資訊，將它拷貝到緩衝區中。緩衝區很大，無法放在消息中，因而要調用26088行的sys\_copy函數將其顯式地拷貝到使用者空間。

保護

MINIX保護機制使用rwx位。每個文件都有三組rwx位，分別用於文件主、文件主所在的組及其他用戶。這些位元由系統調用CHMOD設置，而該系統調用由檔protect.c中的do\_chmod過程(26124行)執行。在進行一系列有效性檢查後，do\_chmod最終在26150行改變保護模式。

系統調用CHOWN和CHMOD有些相似，兩者都修改某檔內部i-節點的一個域。兩者的實現也類似，但是do-chown(26163行)在改變檔擁有者時只能為超級塊用戶調用。普通使用者可以使用這個系統調用改變其檔所在的組。

UMASK系統調用允許使用者設置遮罩標誌(保存在進程表中)，該標誌將遮罩此後的CREAT系統調用中的相應位元。完整的實現實際上只需要26209行一條語句，只是該調用需返回其原來的遮罩值。 這一點使得程式碼量翻了三番(26208行至26210行)。

系統調用ACCESS可由進程用來查找它是否能以某種特定方式訪問一個檔(例如讀檔)，它在do\_access函數(26217行)中實現，do\_access取得檔的i-節點，並調用內部過程forbidden(26242行)檢查訪問是否被禁止。forbidden檢查uid、gid以及i-節點的資訊。根據這些資訊，選取三組rwx中的一組，檢查是否允許指定的訪問。

read\_only(26304行)是一個內部過程，它給出了i-節點參數所在的檔案系統是安裝為唯讀還是讀寫，主要用於防止對安裝成唯讀的檔案系統進行寫操作。

時間

MINIX中有幾個涉及時間的系統調用，他們是UTIME、TIME、STIME和TIMES，都列在圖5-43中。儘管這些系統調用中很多與檔無關，但我們把他們都放在檔案系統中分析，因為時間資訊記錄在檔的i-節點中。

圖5-43 涉及到時間的四條系統調用。

每個檔都有3個32位元數字與其相關聯，其中兩個記錄了檔的最後存取時間和最後修改時間。第三個記錄了檔i-節點本身狀態最後一次修改的時間，該時間在幾乎所有的檔存取時都要修改，唯有READ和EXEC操作例外。這三個時間都保存在i-節點中。利用UTIME系統調用，檔主和超級塊用戶可以設置檔的存取時間以及修改時間。文件time.c中的do\_utime過程(26422行)執行這個系統調用，它取出檔的i-節點，並把時間保存其中。在26450行重置表明需要修改時間的標誌，這樣，系統就不會每次都花很高代價，去執行不必要的clock\_time調用。

即時時鐘由內核中的時鐘任務來維護，檔案系統並不維護即時時鐘。因此，獲取和設置即時時鐘的唯一方法是向時鐘任務發送一條消息。這實際上也是do\_time和do\_stime過程的內容。即時時鐘以秒為單位，從1970年1月1日開始計算。

記帳資訊亦由內核來維護。在每個時鐘週期，它在某個進程中增加一個時鐘週期。這個資訊可以通過向系統任務發送消息而得到，do\_tims過程(26492行)主要執行這一工作。該過程之所以不命名為do\_times，是因為大多數C編譯器都在外部變數前面加底線，並且很多連結程式把符號截為8個字元，這使得do\_time和do\_times無法區分。

其他

文件misc.c中包含了幾個系統調用中使用到的過程，這些過程都不宜放在上面說明。系統調用DUP複製檔描述符，換句話說，它創建一個新的檔描述指向其參數所指定的檔。DUP2是DUP的變體。這兩個系統調用都在過程do\_dup(26632行)中處理，他們現在都不再使用了，在MINIX中仍包含這種功能的目的是支援原來的二進位程式。在C來源程式中遇到其中任何一個時，當前版本的MINIX C庫函數將調用系統調用FCNTL。

在過程do\_fcntl(26670行)中處理的FCNTL是向一個打開檔請求進行某些操作的最佳方式。請求服務時使用圖5-44列出的POSIX定義的標誌。該過程被調用時，含有檔描述符、請求代碼以及特定請求需要的其他附加參數。例如，和原先的調用

dup2(fd，fd2)；

等價的是

fcntl(fd，F\_DUPFD，fd2)；

其中幾個請求設置或讀取標誌位元，他們的代碼很簡單，僅有幾行。例如，F\_SETFD 請求設置一個標誌位元，使得檔的擁有者進程執行EXEC調用時關閉該檔；F\_GETFD 請求則用於檢查在調用EXEC時是否要關閉檔；通過F\_SETFL和F\_GETFL請求可以設置標誌位元以表明某檔可用於非阻塞模式或者是追加操作。

圖5-44 系統調用FCNTL的POSIX請求參數。

do\_fcntl還處理文件鎖，帶有F\_GETLK、F\_SETLK或F\_SETLKW命令的調用被翻譯成對lock\_op的調用，我們在前面幾節中已經討論過lock\_op過程。

下一個系統調用是SYNC，它把自上次讀入記憶體後修改過的所有塊和i-節點寫回磁片。這個調用在過程do\_sync(26730行)中處理。它搜索所有表，找出修改過的項。i-節點應該首先處理，因為rw\_inode把結果保留在塊快取記憶體中。在所有修改過的i-節點寫入到塊快取記憶體之後，將所有修改過的塊都寫回磁片。

系統調用FORK、EXEC、EXIT和SET實際上都是記憶體管理器調用，但他們運行的結果也同樣在這裡作些說明。在生成一個進程時，內核、記憶體管理器以及檔案系統都應該知道這一點。這些"系統調用"並不來自使用者進程，而是來自記憶體管理器。do\_fork、do\_exit和do\_set把相關資訊記錄在進程表的檔案系統部分。do\_exec搜索所有標誌為closed-on-exec的檔，將他們關閉(調用do\_close過程)。

本檔中的最後一個函數並不是一個真正的系統調用，但我們將它按系統調用加以處理，這個函數是do\_revive(26921行)。一個任務原先無法完成檔案系統請求的工作，但是現在完成了該工作(例如為某使用者進程提供輸入資料)，此時調用本函數。檔案系統喚醒使用者進程並向它發送回應訊息。

5.7.7 I/O設備介面

MINIX中的I/O是向內核任務發送消息來完成的。檔案系統與這些任務的介面包含在檔device.c中。在需要設備I/O時，在read\_write中調用dev\_io(27033行)處理字元設備檔，而在rw\_block中調用dev\_io中處理塊設備檔。dev\_io創建一個標準的消息(見圖3-15)並將該消息發送到指定的任務。各個任務在

(\*dmap[major].dmap\_rw)(task，&dev\_mess)；

(27056行)中調用。上面的語句通過table.c中dmap陣列的指標調用相關函數，這些函數就放在檔device.c中。在dev\_io等待任務送迴響應時，檔案系統也等待。其內部並未實現多道程序機制。但是，通常情況下，這個等待過程非常短(例如50毫秒)。

在打開和關閉設備檔時，根據設備類型的不同，需要進行特別處理。dmap表也用於決定在打開和關閉各種不同的主設備時分別調用哪些函數。對於磁片設備，不論是軟碟、硬碟、或是基於記憶體的設備，都調用dev\_opcl過程(27071行)。代碼行

mess\_ptr->PROC\_NR=fp-fproc;

(27081行)計算調用進程的進程號。實際工作是通過向call\_task傳遞任務號和指向消息的指標完成的。我們在後面還會討論call\_task過程。關閉設備時，也調用dev\_opcl。事實上，在本函數層次上，打開設備和關閉設備的唯一不同在於：從call\_task返回之後繼續做什麼。

其他通過dmap結構調用的函數包括用於串列服務的tty\_open和tty\_close，以及為控制台服務的ctty\_open和ctty\_close。最後一個函數ctty\_close幾乎是一個啞函數，它無條件地返回狀態OK。

SETSID系統調用要求檔案系統完成某些任務，這個調用由過程do\_setsid(27164行)來執行。系統調用IOCTL主要放在檔device.c中處理，之所以這樣，是因為它和任務介面密切相關。在執行IOCTL時，調用do\_ioctl創建一條消息並將其發送給適當的任務。

為了控制終端設備，符合標準POSIX的程式應當使用在標頭檔terminal.h中定義的一個函數。C庫將把這些函數轉換為IOCTL調用。非終端設備的許多操作都要用到IOCTL，其中許多已在第三章中介紹。

下面一個函數find\_dev(27228行)是本檔中唯一的PRIVATE函數。這是一個小的輔助過程，它從一個完整的設備號中提取出主設備號和次設備號。

大多數設備的讀寫都在過程call\_task(27245行)中進行，call\_task調用sendrec將一條消息傳送給記憶體圖像中適當的任務。如果這個任務試圖喚醒一個進程以響應一個更早的請求，那麼這種傳送可能會失敗。這個進程很可能與當前請求為其而發送的進程不同。若接收到一個不合適的消息，call\_task過程將在控制台上顯示一條消息。在MINIX的正常操作中，這些消息通常不會出現，但在試圖開發一個新的設備驅動程式時可能出現。

設備/dev/tty在物理上並不存在。它只是一個虛擬裝置，多使用者系統中任何使用者都可以引用它，而無需明確指出使用哪一個真實終端。當發送一個引用了/dev/tty的消息時，下一函數call\_tty(27311行)找到正確的主、次設備號並在call\_task將該消息繼續傳遞下去之前在消息中將設備號替換掉。

本檔中的最後一個函數是no\_dev(27337行)，該函數從表中一個設備並不存在的項調用，例如，在一個沒有網路支援的機器中訪問網路設備時。no\_dev返回ENODEV狀態，因而防止在訪問不存在的設備時系統崩潰。

5.7.8 一般的實用程式

檔案系統中還包含有幾個一般目的的實用過程，這些實用過程收集在檔utility.c中，在許多地方都要用到這些過程。

第一個過程是clock\_time(27428行)，它向時鐘任務發送消息查詢當前即時時間。因為許多系統調用都以檔名作為參數，所以我們定義了過程fetch\_name(27447行)。如果檔案名較短，它被包含在使用者發送給檔案系統的消息之中。若檔案名很長，就把指向用戶空間中名字的一個指標放在消息中。fetch\_name檢查這兩種情況，獲得檔案名。

在檔案系統中，有兩個函數用於處理一般類型的錯誤。 在檔案系統接收到錯誤的系統調用時調用no\_sys函數。而panic則在出現致命性錯誤時列印一條消息，並要求內核停止操作。

最後兩個函數conv2和conv4用於處理Intel和Motorola處理器的不同位元組順序問題。在讀寫磁片資料結構，例如i-節點或點陣圖時調用這些過程。創建磁片的系統所採用的位元組順序記錄在超級塊中。如果它與本地處理器使用的位元組順序不同，則需要交換順序。這樣，檔案系統的其他部分就無需瞭解磁片上的位元組順序。

最後一個檔是putk.c，它包含有兩個過程，都用來列印消息。由於標準的庫過程需要向檔案系統發送消息，因此我們不能使用他們。這兩個過程向終端任務直接發送消息。在本檔的記憶體管理器版本中，我們已經看到兩個幾乎與他們完全一致的函數。

5.8 小結

從外部看來，檔案系統是一組檔和目錄，以及對檔和目錄的操作。檔可以被讀寫，目錄可以被創建和刪除，並且可將檔從一個目錄移到另一個目錄中。大多數現代作業系統都支援層次目錄系統，在層次目錄系統中，目錄中含有子目錄，如此迴圈，直至無窮。

而在內部看，檔案系統卻迥然不同。檔案系統的設計者必須考慮到存儲區如何分配以及系統如何記錄檔使用了哪些塊等等。我們還看到，不同的檔案系統具有不同的目錄結構。檔案系統的可靠性和性能也是一個重要問題。

檔案系統的安全和保護對用戶和設計者都至關重要。我們討論了早期系統中的一些安全缺陷以及大多數系統的共同問題。我們還討論了身份確認、存取控制表、許可權以及矩陣模型等等。

最後，我們詳細地研究了MINIX檔案系統。MINIX檔案系統很大，但並不複雜。它從使用者進程接收任務請求，索引過程指標表，接著調用相應過程執行所要求的系統調用。由於其模組結構以及處在核心之外，我們可以將它從MINIX中刪除，進行小小的修改後將其用作一個獨立的網路檔案伺服器。

在系統內部，MINIX將資料存放在塊快取記憶體中，並在順序存取檔時預讀。若快取記憶體足夠大，在反復存取某些程式，例如編輯時，大多數程式正文都可以在記憶體中找到。

習 題

1 請給出檔/etc/passwd的5種不同路徑名。(提示：考慮目錄項"."和"..")

2 在支援順序檔的系統中常常有檔回繞操作，支援隨機存取檔的系統是否也需要該操作？

3 某些作業系統提供系統調用RENAME給檔改名。同樣也可以通過將檔拷貝到新檔並刪除原文件，而實現檔更名。請問這兩種方法有何不同？

4 考慮圖5-7中的目錄樹，如果當前工作目錄是/usr/jim，那麼相對路徑名為../ast/x的檔的絕對路徑名是什麼？

5 正如書中所提到的，檔的連續分配會導致磁片碎片。請問這是內零頭還是內零頭？並將它與前一章的內容作比較。

6 若某作業系統僅支援單一目錄，但允許該目錄有任意多檔，並且檔案名可任意長。請問該作業系統能否類比一個層次檔案系統？如何模擬？

7 空閒磁碟空間可用空閒表或點陣圖來記錄。磁片位址需要D位元。某磁片有B個塊，其中F個空閒。在什麼條件下，空閒表使用的空間少於點陣圖？設D為16，請確定空閒磁碟空間的百分比。

8 有建議認為每個UNIX檔的第一部分最好和其i-節點放在同一個磁片塊中，這樣做有什麼好處？

9 檔案系統的性能取決於快取記憶體的命中率(即在快取記憶體中找到所需塊的概率)。從快取記憶體中讀取資料需要1毫秒，而從磁片上讀取需要40毫秒，若命中率為h，給出讀取資料所需平均時間的計算公式。並畫出h從0到1.0變化時的函數曲線。

10 一軟碟有40個柱面，尋道時移過每個柱面花6毫秒。若不採取措施儘量使檔的塊在磁片上緊靠存放，則邏輯上相鄰的塊平均間隔13個柱面。另一種情況是作業系統儘量將相鄰的塊放在一起，此時塊間的平均距離為2個柱面。設旋轉延遲為100毫秒，傳輸速率為每塊25毫秒，則在這兩種情況下傳輸一個100塊的檔各需要多長時間？

11 定期緊縮磁碟空間可能帶來什麼好處？

12 如何修改TENEX，使之避免書中所述的口令問題？

13 在取得學位後，你應聘為某大學電腦中心的主任。該中心剛剛將一種很老的作業系統轉移到UNIX。你負責這項工作，在開始工作後十五分鐘，你的助手沖進你的辦公室，說有些學生已經發現了系統使用的口令加密演算法，並將其公佈在BBS上。此時你應該作什麼？

14 Morris-Thompson保護機制使用n位亂數，可以防止入侵者事先對普通字串進行加密以破譯口令。這種機制對於防止一個學生猜測他機器上的超級塊用戶口令是否奏效？

15 某電腦系在局域網中連接了許多UNIX機器。任何主機上的使用者均可鍵入如下命令

machine4 who

並使之在主機machine4上運行，而用戶不必首先登錄到這台遠端主機上。這一功能可以由用戶的內核將命令和使用者uid傳送到遠端主機上實現。若內核可信(例如，含有硬體保護的分時小型機)，這種方案是否安全？若有些機器是學生的不帶硬體保護的PC機，則情況又如何？

16 刪除檔時，檔的塊常常返回到空閒鏈中，但不被清除。你認為作業系統是否應該在釋放塊之前將它清除？請從安全性和性能兩方面考慮，並解釋各自的影響。

17 我們討論了三種不同的保護機制：許可權、存取控制表以及UNIX的rwx位。對於下面的問題，分別適用於哪些機制。

（a）Ken希望除他的同事以外，任何人都能讀取他的文件。

（b）Mitch和Steve希望共用某些秘密檔。

（c）Linda希望公開她的一些文件。

對於UNIX，假設用戶被分為教職工、學生、秘書等。

18 考慮下面的保護機制：給每個物件和進程分別賦予一個號碼，只有當物件號大於進程號時，進程才可以存取物件。這一機制與書中提到的哪種方案相似？它和這種方案有什麼根本不同？

19 在一個用許可權保護的系統中，特洛伊木馬的攻擊能奏效嗎？

20 關於i-節點有兩種觀點，一種認為記憶體容量越來越大，價格越來越便宜，所以當打開檔時，直接為i-節點在記憶體i-節點表中建立一個新拷貝將更快、更方便，沒有必要搜索整個i-節點表來判斷它是否已經存在。另一方則不同意這種觀點。這兩種觀點哪個對？

21 病毒和蠕蟲之間有什麼區別？他們各自如何複製？

22 符號連結是間接地指向其他檔和目錄的檔。與當前MINIX實現的普通連結不同，符號連結有自己的i-節點，該i-節點指向一個資料塊，這個資料塊包含有被連結檔的路徑，而且符號連結的i-節點使得連結可以有與被連結檔不同的屬主和許可權。符號連結和它所指向的檔或目錄可以位於不同的設備。符號連結不是1990年POSIX標準的一部分，但可能在將來被加入到POSIX標準中。請在MINIX中實現符號連結。

23 你認為MINIX中定的64M位元組的檔大小限制滿足不了需要，請使用i-節點中的未用空間實現三次間接來擴展該檔案系統。

24 驗證設置ROBUST是否會使得檔案系統在系統崩潰時更加健壯。這個問題在當前版本的MINIX中還未曾研究，因此兩種情況都有可能。考慮修改的塊被換出快取記憶體，以及修改資料塊同時修改i-節點和點陣圖等。

25 filp表的長度當前作為常數NR\_FILPS定義在檔fs/const.h中。為了容納更多的網路使用者，你希望增大檔include/minix/config.h中的NR\_PROCS。如何把NR\_FILPS定義為NR\_PROCS的函數？

26 設計一種機制，增加對"外部"檔案系統的支持，這樣我們可以在MINIX檔案系統的目錄中安裝其他的檔案系統，比如MS-DOS的檔案系統。

27 假設技術有重大突破，出現了非揮發RAM，即使在掉電時它也能保存資料，而價格和性能等方面也不比常規RAM差。這一改進對檔案系統設計有何影響？

第六章 閱讀材料和參考文獻

前五章中我們已經對作業系統的許多內容作了介紹。本章的目的在於為那些希望對作業系統作進一步研究的讀者能提供一些幫助。6.1節羅列了向讀者推薦的閱讀材料，6.2節按照字母順序列出了本書中所引用的所有書籍和論文。

除了下面列出的參考文獻之外，ACM每年舉辦的作業系統原理專題研討會論文集《Proceedings of the n-th ACM Symposium on Operating Systems Principles》，以及IEEE每年舉辦的分散式運算系統國際會議論文集《Proceedings of the n-th International Conference on Distributed Computing System》都是查閱有關作業系統最新論文的好途徑。同樣，USENIX作業系統設計和實現專題研討會也是一個好的資訊源。更進一步，《ACM Transactions on Computer Systems》和《Operating Systems Review》這兩本學報也常常登載相關的文章。

6.1 推薦的進一步閱讀材料

6.1.1 介紹和概論

Brooks, The Mythical Man-Month: Essays on Software Engineering

一本機智、幽默和信息量很大的著作，關於如何避免象某些人那樣以一種很困難的方式來編寫作業系統，其中有很多好的建議。

Comer, Operation System Design. The Xinu Approach

一本關於Xinu 作業系統的書。Xinu作業系統運行在LSI－11電腦上，它包含有原始程式碼的詳細說明，其中包括使用C語言的完整程式。

Corbato, "On Building Systems That Will Fail"

Corbato被譽為分時系統之父，在他接受圖靈獎的頒獎演說中，他談到許多Brooks在其著作《The Mythical Man-Month》中所述及的問題。Corbato的結論是複雜系統將最終失敗，而且若想要成功，最重要的就是必須擯棄複雜性，盡力爭取設計的簡潔和優雅。

Deitel, Operating Systems, 2nd Ed.

一本作業系統的綜述性教材。本書在標準內容之外，添加了一些實例，如UNIX、MS－DOS、MVS、OS/2以及Mac OS。

Finkel, An Operating Systems Vade Mecum

關於作業系統的另一本綜述性教材。它面向實際應用，寫得很好，並且述及到很多本書中講述的問題。如果想找一本書從不同角度描述同一問題，則它是非常合適的。

IEEE, Information Technology - Portable Operating System Interface (POSIX),

Part 1: System Application Program Interface (API) [C Language]

這是一個標準。其中的有些部分非常易讀，尤其是附件B：“合理性和標注”，它對為什麼採取這樣的方法來解決問題作了許多說明。引用標準的好處之一是，從定義來講，它是絕對沒有錯誤的。即使是排版過程中導致的一個巨集定義名字錯，也不能算作錯誤，因為它是官方認可的標準。

Lampson, "Hints for Computer System Desing"

Buttler Lampson是世界上新型作業系統設計的領導人物之一。他彙集了多年實際經驗中的種種啟示、建議和指南，並將其集中在這篇睿智和信息量很大的文章中。與Brooks的書一樣，這是每一個優秀的系統設計人員的必讀材料。

Lewine, POSIX Programmer's Guide

這本書用一種比POSIX標準文檔更具可讀性的方式描述了POSIX標準，同時還包括一些討論，如怎樣將老的程式轉換到POSIX，以及如何在POSIX環境下開發新程式。書中有很多代碼實例，包括一些完整的程式。本書還描述了所有POSIX需要的庫函數和標頭檔。

Silberschatz and Galvin, Operating System Concepts, 4th Ed.

這是關於作業系統的另一本教材。它涵蓋了進程、記憶體管理、檔和分散式系統。其中包含兩個實例：UNIX和Mach。封面上畫滿了恐龍，不知道在九十年代這與作業系統有什麼關係。

Stallings, Operating Systems, 2nd Ed.

關於作業系統的又一本教材。它包含了通常作業系統所有的論題，也包括一小部分分散式系統的內容，附錄介紹了一些排隊理論。

Stevens, Advanced Programming in the UNIX Environment

本書教讀者如何使用UNIX系統調用介面和標準C庫來編寫C程式。所給的例子基於系統V版本4和4.4BSD UNIX。書中詳細討論了這些UNIX實現與POSIX之間的關係。

Switzer, Operating Systems, A Practical Approach.

該書與本書很類似。書中使用虛擬碼對理論概念給出了解釋，還包含有關TUNIX的大段C原始程式碼，TUNIX是一個模型化的作業系統。與MINIX不同，TUNIX並不在真實機器上運行，而是在一台虛擬機器上運行。在設備驅動程式方面TUNIX沒有MINIX那樣面向實際，但在其他方面它確實比MINIX走得更遠，比如虛擬記憶體。

6.1.2 進程

Andrews and Schneider, "Concepts and Notations for Concurrent Programming"

本文探討了進程和進程間通信，包括忙等待、信號量、管程、消息傳遞以及其他技術。同時也說明了這些概念如何嵌入在不同的程式設計語言中。

Ben-Ari, Principles of Concurrent Programming

這本小冊子專門討論進程間通信問題，其中有不同的章節討論互斥性、信號量、管程以及哲學家就餐問題，等等。

Dubois et al. "Synchronization, Coherence, and Event Ordering in Multiprocessors"

這是關於在共用記憶體的多處理機系統中的同步問題的指導性材料，但其中的某些思想對單處理機系統和分佈記憶體系統同樣有效。

Silberschatz and Galvi, Operating System Concepts, 4th Ed.

本書的第4到第6章討論了進程和進程間通信，包括調度、臨界區、信號量、管程以及經典進程間通信問題。

6.1.3 輸入/輸出

Chen et al., "RAID: High Performance Reliable Secondary Storage"

在高端系統中，使用多個磁碟機平行作業以獲得高速輸入/輸出是一個發展潮流。作者就此進行了討論並從性能、價格和可靠性等方面研究了不同的組織結構。

Coffman et al., "System Deadlocks"

本文簡短地介紹了鎖死，鎖死的產生以及如何預防和檢測鎖死。

Finkel, An Operating Systems Vade Mecum. 2nd Ed.

本書第5章討論了輸入/輸出硬體和設備驅動程式，重點針對終端和磁片。

Geist and Daniel, "A Continuum of Disk Scheduling Algorithms"

本文給出了一個通用的磁片臂調度演算法，並給出了詳細的模擬和實驗結果。

Holt, "Some Deadlock Properties of Computer Systems"

本文主要圍繞鎖死進行討論。Holt引入了一個可被用於分析某些鎖死情況的有向圖模型。

IEEE Computer Magazine, March 1994

該期雜誌包含8篇關於先進輸入/輸出系統的文章，其中涉及到模擬、高性能記憶體、快取記憶體、平行電腦的輸入/輸出以及多媒體。

Isloor and Marsland, "The Deadlock Problem: An Overview"

這是關於鎖死的指導性材料，其中重點講了資料庫系統，並描述了多種模型和演算法。

Stevens, "Heuristics for Disk Drive Positioning in 4.3 BSD"

本文詳細研究了在Berkeley UNIX中的磁片性能。正如多數電腦系統一樣，實際情況遠比理論預測複雜。

Wilkes et al., "The HP AutoRAID Hierarchical Storage System"

RAID（廉價冗餘磁碟陣列）是高性能磁片系統的一項重要進展，在RAID中，由若干小磁片構成的陣列一起工作，以構造一個高性能系統。本文中作者詳細描述了他們在HP實驗室裡開發的系統。

6.1.4 記憶體管理

Denning, "Virtual Memory"

本文是一篇關於虛擬記憶體的經典文章，其中論及了許多虛擬記憶體的許多方面。Denning是該領域的先軀之一，也正是他創立了工作集概念。

Denning, "Working Sets Past and Present"

本書很好地綜述了眾多的記憶體管理機制和調頁演算法，書後附有完整的參考文獻。

Knuth, The Art of Computer Programming Vol. 1

在本書中，集中討論和比較了最先適應演算法、最佳適應演算法和其他演算法。

Silberschatz and Galvin, Operating System Concepts, 4th Ed.

本書第8和第9章討論記憶體管理問題，包括對換、調頁和分段，其中提到很多頁面演算法。

6.1.5 檔案系統

Denning, "The United States vs. Craig Neidorf"

當一個年輕的“駭客”發現並發表了有關電話網系統的工作原理之後，他被指控為電腦詐騙。本文就講述了這樣一件事情，這中間牽涉到很多基本的問題，包括言論的自由。本文發表後引起了一些非議和反駁。

Hafner and Markoff, Cyberpunk

本書以一名紐約時報電腦記者的口吻講述了三個杜撰的故事和他自己的記者生涯。，在故事中，年輕的電腦“駭客”闖入了全世界範圍的電腦。而這名記者本身曾攻破了Internet蠕蟲病毒。

Harbron, File Systems

本書描述檔案系統的設計、應用和性能，其中對結構和演算法均作了介紹。

McKusick et al., "A Fast File System for UNIX"

UNIX的檔案系統在4.2BSD環境下被重新加以實現。本文描述了新檔案系統的設計，其中重點放在性能特性上。

Silberschatz and Galvin Operating System Concepts, 4th Ed.

本書第10和11章的內容是檔案系統。其中包括檔操作、檔訪問方式、一致性語義、目錄和保護等，此外還有其他主題的實現。

Stallings, Operating Systems, 2nd Ed.

本書第14章包含很多有關系統環境安全的內容，特別是關於電腦“駭客”、病毒和其他威脅。

6.2 按字母排序的參考文獻

ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D., and LEVY, H.M.:

"Scheduler Activations: Effective Kernel Support for the Uer-level Management of Parallelis, " ACM Trans. on Computer Systems, vol. 10, pp. 53-79, Feb. 1992.

ANDRES, G.R., and SCHNEIDER, F.B.:

"Concepts and Notations for Concurrent Programming," Computing Surveys, vol. 15, pp.3-43, March 1983.

BACH, M.J.:

The Design of the UNIX Operating Systems, Englewood Cliffs, NJ: Prentice Hall, 1987.

BALA, K., KAASHOEK, M.F., WEIHL, W.:

"Software Prefetching and Caching for Translation Lookaside Buffers," Proc. First Symp. on Operating System Design and Implementation, USENIX. pp. 243-254, 1994.

BAYS, C.:

"A Comparison of Next-Fit, First-Fit, and Best-Fit," Commun. of the ACM, vol. 20, pp. 191-192, March 1977.

BEN-ARI, M:

Principles of Concurrent Programming, Englewood Cliffs, NJ: Prentice Hall International, 1982.

BRINCH HANSEN, P.:

"The Programming Language Concurrent Pascal," IEEE Trans. on Software Engineering, vol. SE-1, pp. 199-207, June 1975.

BROOKS, F.P., jR.:

The Mythical Man-Month: Essays on Software Engineering, Anniversary edition, Reading, MA: Addison-Wesley, 1996.

CADOW, H.:

OS/360 Job Control Language, Englewood Cliffs, NJ: Prentice Hall, 1970.

CHEN, P.M., LEE, E.K., GIBSON, G.A., KATZ, R.H., and PATTERSON, D.A.:

"RAID: High Performance Reliable Storage," Computing Surveys, vol. 26, pp. 145-185, June 1994.

CHERITON, D.R.:

"An Experiment Using Registers for Fast Message-Based Interprocess Communication," Operating Systems Review, vol. 18, pp.12-20, Oct. 1984.

COFFMAN, E.G., F\ELPHICK, M.J., and SHOSHANI, A.:

"System Deadlocks," Computing Surveys, vol. 3, pp.67-78, June 1971.

COMER, D.:

Operating System Design, The Xinu Approach, Englewood Cliffs, N.J.: Prentice Hall, 1984.

CORBATO, F.J.:

"One Building Systems That Will Fail," Commun. of the ACM, vol. 34, pp. 72-81, June 1991.

CORBATO, F.J., MERWIN-DAGGETT, M., and DALEY, R.C:

"An Experimental Time-Sharing System," Proc. AFIPS Fall Joint Computer Conf., AFIPS, pp. 335-344, 1962.

CORBATO, F.J., SALTZER, J.H., and CLINGER, C.T.:

"MULTICS-The First Seven Years," Proc. AFIPS Spring Joint Computer Conf., AFIPS, pp.571-583, 1972.

CORBATO, F.J., and VYSSOTSKY, V.A.:

"Introduction and Overview of the MULTICS System," Proc. AFIPS Fall Joint Computer Conf., AFIPS, pp. 185-196, 1965.

COURTOIS, P.J., HEYMANS, F., and PARNAS, D.L.:

"Concurrent Control with Readers and Writers," Commun. of the ACM, vol. 10. 667-668, Oct. 1971.

DALEY, R.C., and DENNIS, J.B.:

"Virtual Memory, Process, and Sharing in MULTICS," Commun. of the ACM, vol. 11, pp. 306-312, May 1968.

DEITEL, H.M.:

Operating Systems, 2nd Ed., Reading, MA: Addison-Wesley, 1990.

DENNING, D.:

"The United states vs. Craig Neidorf," Commun. of the ACM, vol. 34, pp. 22-43, March 1991.

DENNING, P.J.:

"The Working Set Model for Program Behavior," Commun. of the ACM, vol. 11, pp. 323-33, 1968a.

DENNING, P.J.:

"Thrashing: Its Causes and Prevention," Proc. AFIPS National Computer Conf., AFIPS, pp. 915-922, 1968b.

DENNING, P.J.:

"Virtual Memory," Computing Surveys, vol. 2, pp. 153-189, Sept. 1970.

DENNING, P.J.:

"Working Sets Past and Present," IEEE Trans. on Software Engineering, vol. SE-6, pp. 64-84, Jan. 1980.

DENNING, J.B., and VAN HORN, E.C.:

"Programming Semantics for Multiprogrammed Computations," Commun. of the ACM, vol. 9, pp. 143-155, March 1966.

DIJKSTRA, E.W.:

"Co-operating Sequential Processes," in Programming Languages, Genuys, F. (Ed.), London: Academic Press, 1965.

DIJKSTRA, E.W.:

"The Structure of THE Multiprogramming System," Commun. of the ACM, vol. 11, pp. 341-346, May 1968.

DUBOIS, M., SCHEURICH, C., and BRIGGS, F.A.:

"Synchronization, Coherence, and Event Ordering in Multiprocessors," IEEE Computer, vol. 21, pp. 9-21, Feb. 1988.

ENGLER, D.R., KAASHOEK, M.F., and O'TOOLE, J.Jr.:

"Exokernel: An Operating System Architecture for Application-Level Resource Management," Proc. of the Fifteenth Symp. on Operating Systems Principles, ACM, pp. 251-266, 1995.

FABRY, R.S.:

"Capability-Based Addressing," Commun. of the ACM, vol. 17, pp. 403-412, July 1974.

FEELEY, M.J., MORGAN, W.E., PIGHIN, F.H., KARLIN, A.R., LEVY, H.M., and THEKKATH, C.A.:

"Implementing Global Memory Management in a Workstation CLuster," Proc. of the Fifteenth Symp. on Operating Systems Principles, ACM, pp. 201-212, 1995.

FINKEL, R.A.:

An Operating Systems Vade Mecum, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1988.

FOTHERINGHAM, J.:

"Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store," Commun. of the ACM, vol. 4, pp. 435-436, Oct. 1961.

GEIST, R., and DANIEL, S.:

"A Continuum of Disk Scheduling Algorithms," ACM Trans. on Computer Systems, vol. 5, pp. 77-92, Feb. 1987.

GOLDEN, D., and PECHURA, M.:

"The Structure of Microcomputer File Systems," Commun. of the ACM, vol. 29, pp. 222-230, March 1986.

GRAHAM, R.:

"Use of High-Level Languages for System Programming," Project MAC Report TM-13, M.I.T, Sept. 1970.

HAFNER, K., and MARKOFF, J.:

Cyberpunk, New York: Simon and Schuster, 1991.

HARBRON, T.R.:

File Systems, Englewood Cliffs, NJ: Prentice Hall, 1988.

HAUSER, C., JACOBI, C., THEIMER, M., WELCH, B., and WEISER, M.:

"Using Threads in Interactive Systems: A Case Study," Proc. of the Fourteenth Symp. on Operating Systems Principles, ACM, pp. 94-105, 1993.

HAVENDER, J.W.:

"Avoiding Deadlock in Multitasking Systems," IBM Systems Journal, vol. 7, pp. 74-84, 1968.

HEBBARD, B. et al.:

"A Penetration Analysis of the Michigan Terminal System," Operating Systems Review, vol. 14, pp. 7-20, Jan. 1980.

HOARE, C.A.R.:

"Monitors, An Operating System Structuring Concept," Commun. of the ACM, vol. 17, pp. 549-557, Oct. 1974; Erratum in commun. of the ACM, vol. 18, p. 95, Feb. 1975.

HOLT, R.C:

"Some Deadlock Properties of Computer Systems," Computing Surveys, vol. 4, pp. 179-196, Sept. 1972.

HOLT, R.C:

Concurrent Euclid, The UNIX System, and TUNIS, Reading, MA: Addison-Wesley, 1983.

HUCK, J., and HAYS, J.:

"Architectural Support for Translation Table Management in Large Address Space Machines," Proc. Twentieth Annual Int'l Symp. on Computer Arch., ACM. pp. 39-50, 1993.

IEEE:

Information technology - Protable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language], New York: Institute of Electrical and Electronics Engineers, Inc., 1990

ISLOOR, S.S., and MARSLAND, T.A.:

"The Deadlock Problem: An Overview," IEEE Computer, vol. 13, pp. 58-78, Sept. 1980.

KERNIGHAN, B.W., and RITCHIE, D.M.:

The C Programming Language, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1988.

KLEIN, D.V.:

"Foiling the Cracker: A Survey of, and Improvements to, Password Security," Proc. UNIX Security Workshop II, USENIX, Summer 1990.

KLEINROCK, L.:

Queueing Systems. Vol. 1, New York: John Wiley, 1975.

KNUTH, D.E.:

The Art of Computer Programming, Volume 1: Fundamental Algorithms, 2nd Ed., Reading, MA: Addison - Wesley, 1973.

LAMPSON, B.W.:

"A Scheduling Philosophy for Multiprogramming Systems," Commun. of the ACM, vol. 11, pp. 347-360, May 1968.

LAMPSON, B.W.:

"A Note on the Confinement Problem," Commun. of the ACM, vol. 10, pp. 613-615, Oct. 1973.

LAMPSON, B.W.:

"Hints for Computer System Design," IEEE Software, vol. 1, pp. 11-28, Jan. 1984.

LEVIN, R., COHEN, E.S., CORWIN, W.M., POLLACK, F.J., and WULF, W.A.:

"Policy/Mechanism Separation in Hydra," Proc. of the Fifth Symp. on Operating Systems Principles, ACM, pp. 132-140, 1975.

LEWINE, D.:

POSIX Programmer's Guide, Sebastopol, CA: O'Reilly & Associates, 1991.

LI, K., and HUDAK, P.:

"Memory Coherence in Shared Virtual Memory Systems," ACM Trans. on Computer Systems, vol. 7, pp. 321-359, Nov. 1989.

LINDE, R.R.:

"Operating System Penetration," Proc. AFIPS National Computer Conf., AFIPS, pp. 361-368, 1975.

LIONS, J.:

Lions' Commentary on Unix 6th Edition, with Source Code, San Jose, CA: Peer-to-Peer Communications, 1996.

LIU, C.L., and LAYLAND, J.W.:

"Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," Journal of the ACM, vol. 20, pp. 46-61, Jan. 1973.

MARSH, B.D., SCOTT, M.L., LEBLANC, T.J., and MARKATOS, E.P.:

"First-Class User-Level Threads," Proc. of the Thirteenth Symp. on Operating Systems Principles, ACM, pp. 110-121, 1991.

McKUSICK, M.J., JOY, W.N., LEFFLER, S.J., and FABRY, R.S.:

"A Fast File System for UNIX," ACM Trans. on Computer Systems, vol. 2, pp. 181-197, Aug. 1984.

MORRIS, R., and THOMPSON, K.:

"Password Security: A Case History," Commun. of the ACM, vol. 2, pp. 594-597, Nov. 1979.

MULLENDER, S.J., and TANENBAUM, A.S.:

"Immediate Files," Software - Practice and Experience, vol. 14, pp. 365-368, April 1984.

ORGANICK, E.I.:

The Multics System, Cambridge, MA: M.I.T Press, 1972.

PETERSON, G.L.:

"Myths about the Mutual Exclusion Problem," Information Processing Letters, vol. 12, pp.115-116, June 1981.

ROSENBLUM, M., and OUSTERHOUT, J.K.:

"The Design and Implementation of a Log-Structured File System," Proc. Thirteenth Symp. on Operating System Principles. ACM, pp. 1-15, 1991.

SALTZER, J.H.:

"Protection and Control of Information Sharing in MULTICS," Commun. of the ACM, vol. 17, pp. 388-402, July 1974.

SALTZER, J.H., and SCHROEDER, M.D.:

"The Protection of Information in Computer Systems," Proc. IEEE, vol. 63, pp. 1278-1308, Sept. 1975.

SALUS, P.H.:

"UNIX At 25," Byte, vol. 19, pp. 75-82, Oct. 1994.

SANDHU, R.S.:

"Lattice-Based Access Control Models," Computer, vol. 26, pp. 9-19, Nov. 1993.

SEAWRIGHT, L.H., and MACHINNON, R.A.:

"VM/370 - A Study of Multiplicity and Usefulness," IBM Systems Journal, vol. 18, pp. 4-17, 1979.

SILBERSCHATZ, A., and GALVIN, P.B.:

Operating System Concepts, 4th Ed. Reading, MA: Addison-Wesley, 1994.

STALLINGS, W.:

Operating Systems, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1995.

STEVENS, W.R.:

Advanced Programming in the UNIX Environment, Reading, MA: Addison-Wesley, 1992.

STEVENS, W.R.:

"Heuristics for Disk Drive Partitioning in 4.3BSD," Computing Systems, vol. 2, pp. 251-274, Summer 1989.

STOLL, C.:

The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage, New York: Doubleday, 1989.

SWITZER, R.W.:

Operating Systems, A Practical Approach, London: Prentice Hall Int'l, 1993.

TAI, K.C., and CARVER, R.H.:

"VP: A New Operation for Semaphores," Operating Systems Review, vol. 30, pp. 5-11, July 1996.

TALLURI, M., and HILL, M.D.:

"Surpassing the TLB Performance of Superpages with Less Operating System Support," Proc. Sixth Int'l Conf. on Architectural Support for Progr. Lang. and Operating Systems, ACM, pp. 171-182, 1994.

TALLURI, M., and HILL, M.D., and KHALIDI, Y.A.:

"A New Page Table for 64-bit Address Spaces," Proc. of the Fifteenth Symp. on Operating Systmes Principles, ACM. pp. 184-200, 1995.

TANENBAUM, A.S.:

Distributed Operating Systems, Englewood Cliffs, NJ: Prentice Hall. 1995.

TANENBAUM, A.S., VAN RENESSE, R., STAVEREN, H. VAN, SHARP, G.J., MULLENDER, S.J., JANSEN, J., and ROSSUM, G.VAN:

"Experiences with the Amoeba Distributed Operating System," Commun. of the ACM. vol. 33, pp. 46-63, Dec. 1990.

TEORY, T.J.:

"Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems," Proc. AFIPS Fall Joint Computer Conf., AFIPS, pp. 1-11, 1972.

THOMPSON, K.:

"UNIX Implementation," Bell System Technical Journal, vol. 57, pp. 1931-1946, July-Aug. 1978.

UHLIG, R., NAGLE, D., STANLEY, T, MUDGE, T., SECREST, S., and BROWN, R:

"Design Tradeoffs for Software-Managed TLBs," ACM Trans. on Computer Systems, vol. 12, pp. 175-205, Aug. 1994.

VAHALIA, U.:

UNIX Internals - The New Frontiers, Upper Saddle river, NJ: Prentice Hall, 1996.

WALDSPURGER, C.A., and WEIHL, W.E.:

"Lottery Scheduling: Flexible Proportional-Share Resource Management," Proc. First Symp. on Operating System Design and Implementation, USENIX, pp. 1-12, 1994.

WILKES, J., GOLDING, R., STAELIN, C, and SULLIVAN, T.:

"The HP AutoRAID Hierarchical Storage System," ACM Trans. on Computer Systems, vol. 14, pp. 108-136, Feb. 1996.

WULF, W.A., COHEN, E.S., CORWIN, W.M., JONES, A.K., LEVIN, R., PIERSON, C., and POLLACK, F.J.:

"HYDRA: The Kernel of a Multiprocessor Operating System," Commun. of the ACM, vol. 17, pp. 337-345, June 1974.

ZEKAUSKAS, M.J., SAWDON, W.A., and BERSHAD, B.N.:

"Software Write Detection for a Distributed Shared Memory," Proc. First Symp. on Operating System Design and Implementation, USENIX, pp. 87-100, 1994.

1 若某件事可能出錯，則它一定會出錯

作業系統：設計與實現 作者簡介

iv

作業系統：設計與實現 前言

作業系統：設計與實現 譯 序

作業系統：設計與實現 第一章 引 言

1

作業系統：設計與實現 第一章 引 言

2

作業系統：設計與實現 第二章 進 程

作業系統：設計與實現 第三章 輸入/輸出系統

207

作業系統：設計與實現 第四章 記憶體管理

345

作業系統：設計與實現 第五章 檔案系統

作業系統：設計與實現 第六章 閱讀材料和參考文獻