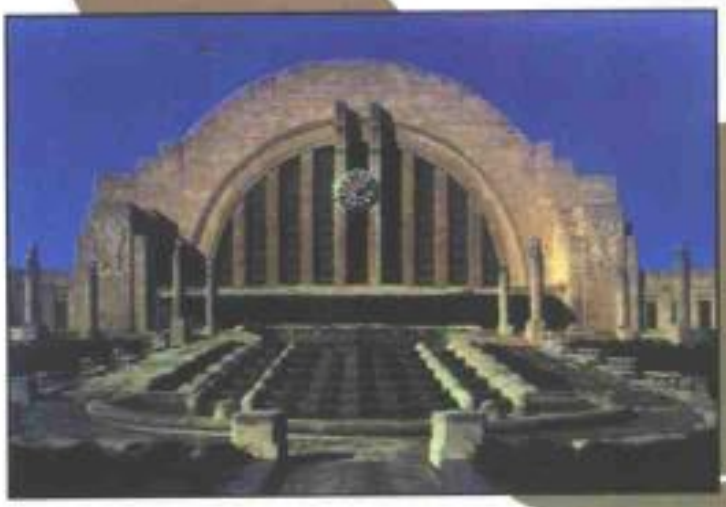
Addison Wesley

现代体系结构上的UNIX系统

——内核程序员的SMP和Caching技术

[美] Curt Schimmel 著
张 辉 译



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

人民邮电出版社
POSTS & TELECOMMUNICATIONS PRESS

Addison Wesley

现代体系结构上的UNIX系统

本书在揭示UNIX内核奥秘的诸多书籍中是具有重要意义的崭新里程碑。在体现当今先进技术水平的系统上，采用对称多处理机技术和高速缓存存储系统来提高系统性能，已是颇为划算的重要技术。

本书是为UNIX内核开发人员编写的，它全面而通俗地阐述了高速缓存和对称多处理机的操作，二者如何协调工作，以及为了在融合两者的机器上运行操作系统所必须解决的问题。

在第一部分中回顾了UNIX内核的内部原理之后，Curt Schimmel开始详细描述高速缓存存储系统，其中包括几种虚拟地址和物理地址高速缓存，另外还用一章的篇幅讲述了高效的高速缓存管理技术。针对每一种高速缓存的类型，本书不仅说明了它们对软件的影响，而且还说明了操作系统为这些系统所进行的必要调整和变化。第二部分详细介绍了紧密耦合、共享存储和对称多处理机的操作。这一部分研究了这些多处理机给操作系统带来的问题，比如竞争条件，死锁以及存储器操作的次序，而且考察了如何对UNIX内核进行调整使之适于在这样的系统上运行。本书最后一部分考察了高速缓存存储系统和多处理机之间的相互作用以及这种相互作用给内核带来的新问题，随后阐述了用于解决这些问题的技术。

本书以大量示例来演示所讲述的概念，其中既有代表CISC处理器的例子，也有代表RISC处理器的例子，比如Intel 80486和Pentium，Motorola 68040和88000，以及MIPS和SPARC处理器。为了增进读者对概念的理解，每一章还包含了一组练习题，在本书末尾有选择地给出了习题的答案。

ISBN 7-115-10876-5

9 787115 108760 >

ISBN7-115-10876-5/TP·3195
定价：39.00元

人民邮电出版社
http://www.ptpress.com.cn

现代体系结构上的 UNIX 系统

——内核程序员的 SMP 和 Caching 技术

[美] Curt Schimmel 著

张 辉 译

人 民 邮 电 出 版 社

图书在版编目（CIP）数据

现代体系结构上的 UNIX 系统：内核程序员的 SMP 和 Caching 技术 /（美）希梅尔（Schimmel,C.）著；张辉译. —北京：人民邮电出版社，2003.4

ISBN 7-115-10876-5

Ⅰ. 现… Ⅱ. ①希… ②张… Ⅲ. UNIX 操作系统 Ⅳ. TP316.81

中国版本图书馆 CIP 数据核字（2003）第 016159 号

版 权 声 明

现代体系结构上的 UNIX 系统
—— 内核程序员的 SMP 和 Caching 技术

◆ 著 [美] Curt Schimmel
译 张 辉
责任编辑 李 际

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
读者热线 010-67132705
北京汉魂图文设计有限公司制作
北京鸿佳印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092 1/16
印张：19.25
字数：460 千字 2003 年 4 月第 1 版
印数：1-4 000 册 2003 年 4 月北京第 1 次印刷

著作权合同登记 图字：01－2002－2768 号

ISBN 7-115-10876-5/TP・3195

定价：39.00 元

内容提要

本书首先回顾了与全书其他内容切实相关的 UNIX 系统内幕。回顾的目的是增进读者对 UNIX 操作系统概念的了解，并且定义随后使用的术语。本书接下来的内容分为 3 个部分。第一部分“高速缓存存储系统”介绍了高速缓存体系结构、术语和概念，详细考察了 4 种常见的高速缓存实现——3 种虚拟高速缓存的变体和物理高速缓存。第二部分“多处理机系统”讨论了调整单处理机内核的实现，使之适合于紧密耦合、共享存储多处理机上运行时所面临的问题和设计事宜，还研究了几种不同的实现。最后一部分介绍多处理机高速缓存一致性，这一部分通过研究高速缓存加入到一个紧密耦合、共享存储器多处理机系统时出现在操作系统和高速缓存体系结构上的问题，从而将前两个部分的内容结合到一起。

本书适合于大学计算机及相关专业高年级本科生或者研究生使用。每一章都包含有一组练习题，问题都需要采用这一章所提供的信息以及一些额外学到的知识来解答，习题大都建立在这一章中所出现的例子的基础之上。在本书的末尾有选择地给出了习题的答案。

序

本书的目标在于，为了让操作系统能够在采用了高速缓存（cache memory）以及（或者）多处理机（multiprocessor）技术的现代计算机系统上运行，就若干必须要解决的问题提供实用的信息。在撰写本书的时候，已经有一些讲述 UNIX 系统实现的书籍，但是却没有一本书详细描述应该如何管理高速缓存和多处理机。许多计算机体系结构方面的书籍都是从硬件方面介绍高速缓存和多处理机的，但是却没有一本书能够成功地解决这些现代体系结构给操作系统所带来的问题。本书的意图就是通过建立计算机体系结构和操作系统之间的桥梁来填补这些缺憾。

本书是为操作系统开发人员编写的，它从系统程序员的角度阐述了高速缓存和多处理机的操作。虽然本书的读者对象是 UNIX 系统程序员，但是本书的编写形式使之能够适用于任何操作系统，其中包括所有的 UNIX 变体。通过概念层次上阐述的问题和解决方案，并且使用 UNIX 系统服务（system service）为例来展示这些问题出现的地方，从而达到了这一效果。所以本书提供的解决方案也可以应用到相应环境下的其他操作系统上。

本书打算采取两种途径向操作系统开发人员提供帮助。首先，读者将学习如何调整现有的操作系统，使之能够在现代体系结构上运行。为了做到这一点，本书从操作系统的角度来详细研究这些体系结构的操作，并且阐述了操作系统要管理这些体系结构必须做什么。其次，读者将学习现代体系结构在不同方法之间进行抉择时所涉及的若干权衡考虑。在投身于采用高速缓存和多处理机的新型计算机系统设计工作时，这会给予操作系统开发人员所需要的背景知识。

本书假定读者熟悉UNIX系统调用接口（system call interface）和 UNIX 内核原理的高级概念。读者还应该熟悉计算机体系结构（computer architecture）和计算机系统组织（computer system organization）在这两方面应该具备计算机科学系本科生水平的课程所教授的知识。

本书扩充了为计算机产业界的 UNIX 系统专业人员所开发的 I 级课程（course I）。在过去的 4 年中，这门课一直在美国的 USENIX 大会和欧洲的 UKUUG 大会上讲授。由于它是一门为期

一天的辅导课，因而在所含材料的数量上就有所限制。本书更为细致地涵盖了有关高速缓存和多处理机课程的所有材料，并且还包含了其他主题。

本书适合于大学高年级本科生或者研究生使用。每一章都包含有一组练习题。选出的问题都需要采用这一章所提供的信息以及一些额外学到的知识来解答，并不是简单地模仿他人的材料来出题。在许多情况下，习题都建立在这一章中所出现的例子的基础之上。答案往往采取一小段话的形式（大多数情况下有 4~5 句话，有时会长一些）。希望读者能够尝试解答所有的问题，以此增进对所学概念的掌握。在本书的末尾有选择地给出了习题的答案。

我们首先回顾了与本书其他内容切实相关的 UNIX 系统内幕。回顾的目的是增进读者对 UNIX 操作系统概念的了解，并且定义随后使用的术语。本书接下来分成 3 个部分：高速缓存存储系统(cache memory system)、多处理机 UNIX 实现以及多处理机高速缓存一致性(cache consistency)。第一部分“高速缓存存储系统”介绍了高速缓存体系结构、术语和概念。接下来详细讲述了 4 种常见的高速缓存实现：3 种虚拟高速缓存（virtual cache）的变体和物理高速缓存（physical cache）。第二部分“多处理机系统”讨论了调整单处理机（uniprocessor）内核的实现，使之适合于在紧密耦合（tightly-coupled）、共享存储多处理机（shared memory multiprocessor）上运行时所面临的问题和设计事宜，还研究了几种不同的实现。最后一部分介绍多处理机高速缓存一致性，这一部分通过研究高速缓存加入到一个紧密耦合、共享存储器多处理机系统时所出现的操作系统和高速缓存体系结构上的问题，从而将前两个部分的内容结合到一起。

本书因地制宜地使用一组经过挑选的现代多处理机体系结构来举例说明相关的概念。其中 Motorola 68040 和 Intel 80X86 系列(80386、80486 和 Pentium)代表了传统的 CISC(complex instruction set computer，复杂指令集计算机）处理器。RISC（reduced instruction set computer，精简指令集计算机）方法则以 MIPS 系列（R2000、R3000 和 R4000）、Motorola 88000 以及来自德州仪器公司（Texas Instruments，TI）的 SPARC v8 兼容处理器（MicroSPARC 和 SuperSPARC）为代表。另外还出现了其他几个例子，包括 Sun 和 Apollo 工作站和 Intel i860。在附录 A 中可以找到这些处理器特性的一份汇总介绍。

我要向在本书付梓之前耗费时间评审书稿的人士表示我的感激之情。特别要感谢下面这些人：Steve Albert、Paul Borman、Steve Buroff、Clement Cole、Peter Collinson、Geoff Collyer、Bruce Curtis、Mukesh Kacker、Brian Kernighan、Steve Rago、Mike Scheer、Brian Silverio、Rich Stevens、Manu Thapar、Chris Walquist 和 Erez Zarok。我也要向 Addison-Wesley 公司的员工们表示感谢，感谢他们在本书出版的过程中所提供的帮助和建议，特别要感谢 Kim Dawley、Kathleen Duff、Tiffany Moore、Simone Payment、Marty Rabinowitz 和 John Wait。他们的帮助使得这本书比我一个人努力的结果要好得多。我还要感谢许多在上辅导课期间花时间填写课程评语，从而提供其深思熟虑后的反馈意见的人士。

符号约定

本部分举例说明在本书的示例中所采用的符号约定。

常数（constant）

十进制、八进制和十六进制常数都以 C 编程语言的标准记法来表示。十进制常数总是以数字 1~9 中的一个开头。八进制常数以 0 开头。十六进制常数以字符序列“0x”开头。这里是一些例子：

12345	十进制常数
07654	八进制常数
0x3af	十六进制常数

字长（word size）

存储器中的一个字（word）为 4 字节（byte）。每字节 8bit，所以每一个字是 32bit。

字节顺序（byte order）

存储器最容易想成是字的数组（array）。要引用一个字中的单个字节，需要有一种给它们编号的约定。所有展示存储器中字节排列的例子都使用了高字节结尾（big-endian）的字节顺序。这意味着每个字的最高有效字节（Most Significant Byte，MSB）拥有最低的地址，而最低有效字节（Least Significant Byte，LSB）拥有最高的地址。在字中的字节是从左到右读取的，首先读取最高有效字节。例如，下面的字节编号方式会运用到字长为 4 字节的机器上：

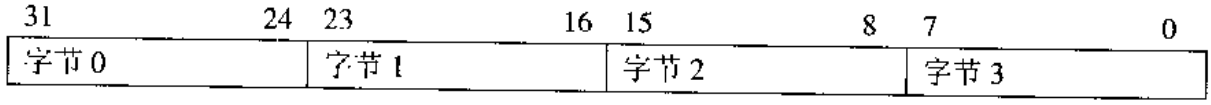
	MSB			LSB
字 0	字节 0	字节 1	字节 2	字节 3
字 1	字节 4	字节 5	字节 6	字节 7
字 2	字节 8	字节 9	字节 10	字节 11

Motorola 的处理器、Sun SPARC 兼容类型的处理器以及 IBM 的处理器（IBM PC 的处理

器除外）都使用高字节结尾的字节顺序。DEC 和 Intel 处理器使用低字节结尾（little-endian）的字节顺序，在这类处理器中，一个字里面的字节是按照逆序编号的。MIPS 处理器可以配置成任何一种字节顺序，在生产基于 MIPS 处理器的系统的所有厂商中，除了 DEC 之外，都选择高字节结尾的字节顺序。

比特位顺序（bit order）

在一个字或者字节内，各个比特位的顺序遵循低字节结尾的顺序，这意味着最低有效比特位（least significant bit，LSb）的编号为 0 位，而 4 字节长的字中，最高有效比特位（most significant bit，MSb）是 31 位。最高有效比特位始终在最左边。在需要用比特位的编号来说清楚一个例子的时候，就在一个字节或者字的上方给出比特位的编号来。例如，下面的字显示出了每个字节内的最高和最低有效比特位，每个字节则按照字内的比特位顺序进行编号。



比特位范围（bit range）

有时需要从一个字或者字节中引用一串连续的比特位，这就称为比特位范围（bit range）。比特位范围可以用尖括号括起范围两端的比特位号来表示。例如，在上面所示的一个字中，构成字节 2 的比特位序列被表示成“比特位<15..8>”。最高有效比特位号始终出现在左边，右边跟着最低有效比特位号。这种记法所表示的范围始终包括这两个比特位的位置本身。

存储器大小的单位（memory size unit）

千字节（kilobyte）被缩写成 K 或者 KB，它包含 1024 字节。兆字节（megabyte）被缩写成 M 或者 MB，它包含 1024KB。吉字节（gigabyte）被缩写成 G 或者 GB，它包含 1024MB。例如，4KB 是 4096 字节，而 8MB 是 8 388 608 字节。存储器大小始终是以字节来衡量的。K、KB、M、MB、G 和 GB 等缩写均会在本书中使用。

前 言

在计算机系统发展历史中的许多时期，构建整体上速度更快的系统的愿望都集中在系统的三大组成部分——CPU、存储子系统和 I/O 子系统——的速度都更快上面。通过提高时钟速度就可以制造出更快的 CPU。通过降低存取时间就可以制造出更快的存储子系统。通过提高数据传输速率就可以制造出更快的 I/O 子系统。但是，随着时钟速度和传输速率的提高，要提高系统的整体速度就变得越来越困难了，因此要设计和构建这样的系统，成本也变得越来越高。随着速度的提高，传输延迟（propagation delay）、信号上升时间（signal rise time）、时钟同步和分发（clock synchronization and distribution）等等都变得越发重要起来。这类高速设计的高成本更难获得有效的性能价格比。

因为受这样和那样的因素影响，系统设计人员扩大了他们的关注范围，以找出提高系统整体性能的其他途径。精简指令集计算机（Reduced Instruction Set Computer，RISC）系统的概念就是其成果之一，在 RISC 系统中对 CPU 指令集进行了简化，从而让一个低成本的快速硬件实现就能完成这些指令。另一项成果是高速缓存存储系统的发展。高速缓存通过把程序中引用最频繁的数据和指令保存在一小块高速存储器中，以此降低主存储系统的负载，从而提高系统的性能。通过增加一小块成本划算、高速度的高速缓存而不是一个成本高、规模大的高速主存储子系统就能加速存储器整体存取速度。采用高速缓存存储器有可能带来更快的存储器整体存取时间，这对于 RISC 系统来说尤为重要，因为要完成相同的任务，使用精简的指令集通常要求 RISC CPU 比传统的 CPU 体系结构取得和执行更多的指令。一般而言，RISC 系统需要更高的带宽来以峰值性能运行。

通过并行运行更多台设备而不是提高任何单台设备的速度就能获得更高的 I/O 传输速率。这就导致了诸如廉价磁盘冗余阵列（Redundant Arrays of Inexpensive Disks，RAID）之类设备的发展，在 RAID 中，多块磁盘并行运行以提供更高的整体传输率。通过增加一个系统中 CPU 的数量来构建多处理机，这样的技术也可以用于提高 CPU 的速度。多处理机把系统负载分散到多个处理器上来增加整个系统的吞吐率。

多处理机和高速缓存是密切相关的。紧密耦合多处理机系统（tightly coupled multiprocessor system）有一个共享的主存储子系统，随着处理器数量的增加，它需要更高的主存储带宽，因为每个处理器在取得和执行一条独立的指令流的同时，都必须在存储器中访问一组独立的数据。将一块高速缓存和每个处理器进行耦合，从高速缓存而不是共享的主存储器来满足处理器大部分的存储器访问请求，就可以降低主存储器的负载。这是一种颇为划算的提高系统性能的途径。

虽然高速缓存能够在多处理机中增加有效的存储器带宽，但是高速缓存结构对于管理它所需要的操作系统开销有很大的影响，这又反过来影响了系统的整体性能。

总而言之，构建速度更快的计算机系统不仅仅是一件诸如提高 CPU 时钟速度这样的事。虽然这样的技术实际上造就了更快的系统，但是它们不一定就是经济上划算的解决方法。通过集中研究如何利用现有的系统部件来提供更高的系统性能，人们已经发现高速缓存和多处理机是划算的解决方案。因此，我们就从这里开始研究高速缓存和多处理机的体系结构，以及它们给操作系统带来的问题。

目　录

第1章 回顾 UNIX 内核原理

本章回顾了 UNIX 内核原理的有关内容，在以后各章中会用到它们。这里没有完整地讨论这个主题，而是作为那些已经熟悉基本概念和术语的人对这些内容进行的一次复习。本章的内容涉及单处理机系统。多处理机的 UNIX 系统实现是本书第二部分的主题。不熟悉 UNIX 操作系统或者 UNIX 内核原理的读者应该首先从本章末尾给出的参考文献中选出一些阅读。

1.1 引 言

UNIX 系统是一种多用户、多任务操作系统，它提供了高度的程序可移植性以及丰富的开发工具集合。UNIX 系统取得成功的一部分原因在于它所提供的可移植的应用程序接口集合（application interface set）。这一接口集合能够轻而易举地处理把应用程序从一家厂商的系统移植到另一家厂商的问题。UNIX 取得成功的另一部分原因在于操作系统、命令和库（library）本身的编写都可以轻松地移植到不同的计算机上，从而促进了市场上 UNIX 硬件平台的多样性。

UNIX 系统在逻辑上具有分层的结构，可以分成两个主要部分：内核（kernel）和用户程序（user program）。图形化的表示如图 1-1 所示。

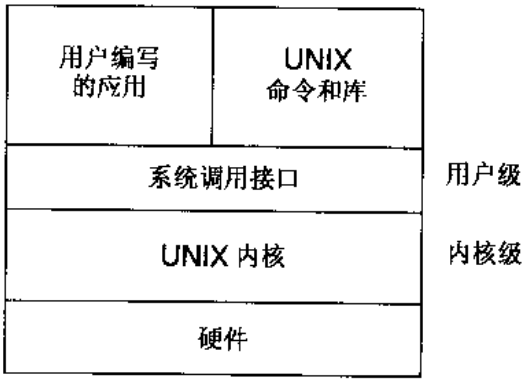


图 1-1 UNIX 系统的逻辑分层

内核的用途是与硬件接口并且控制硬件。内核还向用户程序提供一组抽象的系统服务，称为系统调用，使用可移植的接口就能够访问系统调用。内核在内核级（kernel level）上运行，在这个级别上它能够执行特权操作。这能让内核完全控制硬件和用户级（user-level）程序，并且提供一个让所有的程序协调共享底层硬件的环境。

UNIX 系统调用服务的定义在很大程度上能够让它们在所有的 UNIX 系统上都显得相同，而不管硬件的特殊性如何。这些抽象概念提供了 UNIX 用户级程序高度的可移植性。文件（file）就是一种内核抽象服务的例子。UNIX 文件呈现为一个顺序字节流的形式，其中没有记录（record）或者任何其他类型的边界。用户程序可以从文件的任何部分读取任意数量的字节，而无需考虑对齐任何类型的边界。这就使用户程序在存取一个文件的时候不必关注磁盘的物理扇区（physical sector）、磁道（track）以及柱面（cylinder）边界。文件抽象如何映射到硬件上的细节问题是由内核来负责处理的。

用户的应用程序、UNIX 命令以及库（常用例程的集合）都共存于用户级。用户级包含非特权的硬件执行状态。因此，用户级程序是在一个受限的环境中执行的，它受到了内核的控制，防止同时执行的多个程序彼此互相干扰（无论是恶意的还是无意的）。当用户程序通过执行一次系统调用来请求服务的时候，系统调用会转入内核，在那儿它代表发出请求的用户程序执行一次服务。还可以做权限检查来确保程序有权访问被请求的服务。

图 1-1 描绘出了 UNIX 系统以及其他大多数操作系统传统上是如何实现的：它们都是作为单个庞大程序来实现的。随着时间的推移，这种实现一直在向结构化的方向发展，在结构化的方式中，内核服务被分割成了独立的模块（module）。这就增加了实现的灵活性，更易于添加、改变以及移植系统服务，也有可能将一些服务移到内核之外，在特殊的服务器进程中以用户级来运行它们。这就减少了内核自身所必须提供的服务，从而使其缩小为微内核（micro-kernel）。因为本书所介绍的概念和技术并不依靠内核的内部组织，所以也就无需进一步深入考虑组织的问题。从现在开始，术语“内核”一词将用来指提供 UNIX 系统服务的东西，而不管它是单个程序还是一组模块。

1.2 进程、程序和线程

程序（program）被定义为执行某项任务所需的指令和数据集。进程（process）则是程序加上其执行状态的组合，进程最少要包括所有变量的值、硬件状态（例如，程序计数器、寄存器、条件码等），以及地址空间的内容说明。简而言之，一个进程就是一个执行中的程序。

当一个用户请求运行某个程序的时候，就会创建一个新进程来包含该程序的执行。在进程终止之前，它都存在于系统中，最后它不是自愿终止就是内核使它终止，要么就是用户请求它终止。进程还可以在一定程度上能通过影响诸如进程的调度优先级的系统调用进行控制。

通过进程的抽象概念，内核就让程序有了它自己是在硬件上运行的假象。除非用户程序明确想要和系统中的其他程序以某种方式进行通信（有几种服务可以来完成这个任务），否则它们不需要关心自己与那些程序的交互作用。每个进程都获得了各自的虚拟地址空间

1.3 进程地址空间

内核给每个进程提供了它自己的虚拟地址空间（virtual address space）。在正常情况下，一个进程不能直接访问另一个进程的地址空间；这就提供了一种高度的保护能力，防止来自系统中其他正在执行的进程的干扰。有些实现提供了共享存储器中某些部分的机制。共享存储（shared memory）和映射文件（mapped file）机制都将在本章后面的内容中进行讨论。其他机制（比如vfork系统调用和线程）都能在进程间共享部分或者全部地址空间。对于本书的目的来说，这类机制都具有和共享存储同样的问题，所以就不再深入讨论了。几乎所有的UNIX系统实现都使用请求调页机制（demand paging）来管理物理存储器的分配。

一个进程的地址空间由4个主要部分构成：程序指令、初始化数据、未初始化数据和堆栈。在UNIX的行话中，指令（instruction）也叫做“正文”段，而初始化数据和堆栈可以分别简称为“数据”段和“堆栈”段。未初始化数据则叫做“bss”，它的名字来源于一种叫做“Block Started by Symbol”的古老的汇编程序助记符，这个助记符用于分配未初始化的数据空间。初始化数据和未初始化数据之间的区别在于，初始化数据是在程序编译时已经声明有一个初始值的全局和静态程序变量。未初始化数据是没有明确初始值的全局和静态程序变量。对于这些数据，UNIX系统仅仅依照C程序设计语言（UNIX系统几乎都是用这种语言编写的）的语义在地址空间中分配初始包含0的内存。这种方法的优点是未初始化数据不需要在程序文件中占用空间。

大多数使用32位虚拟地址的系统都将整个4GB的地址空间分给用户程序和内核。虽然每个段的实际起始地址是与实现无关的，但是典型的布局则如图1-2所示。通常低2GB空间供用户使用，常被叫做用户地址空间（user address space）。高2GB空间则为内核保留，不准用户级代码读写，这是内核地址空间（kernel address space）。内核地址空间包括内核的正文和数据结构。当内核正在执行的时候，它就可以访问整个地址空间。这种安排易于让内核在代表用户进程执行一次系统调用的时候可以在用户进程的地址空间中运行。

用户正文段和数据段的大小在程序编译时就固定了，它们只能在执行程序的时候从包含程序的文件中复制到地址空间里。bss段和堆栈段能够在运行时刻动态地增长，在它们之间有一段未用的虚拟地址空间来调节增长的空间。在本书中，堆栈段始终是向较低的存储地址增长的，对于大多数计算机系统来说都是这样。

bss段能够借助系统调用sbrk增长或者缩小。bss段只能向较高的存储地址增长。堆栈随着需要由内核来动态和透明地控制增长。当试图访问当前分配的堆栈段以下的未用区时，就发生了一次缺页错（page fault）。内核检查堆栈指针寄存器的内容，如果它包含的地址比堆栈段顶部当前的地址要小，那么内核就扩大堆栈段，把堆栈指针寄存器中的地址包括进来，并且重新执行导致缺页错的操作。

其他类型的段，比如共享库和共享存储，都可以包含在用户地址空间内。共享库包括附加的正文、数据和bss段，它们用于常用的函数和服务。共享存储则在本章的后面介绍。

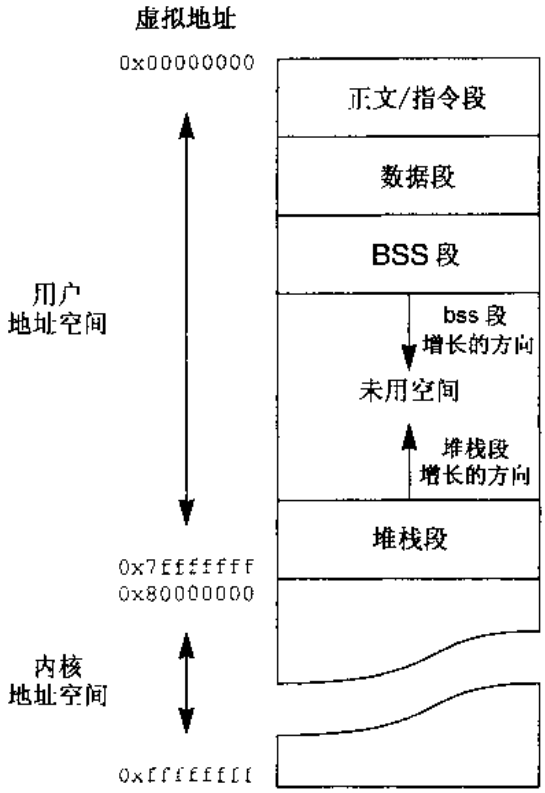


图 1-2　典型的进程虚拟地址空间布局

1.3.1　地址空间映射

内核负责将一个进程的虚拟地址空间映射到计算机的物理地址空间上。大多数计算机允许任何虚拟页面被映射到存储器中的任何物理页面上。例如，一个进程的虚拟地址空间可以按照图 1-3 所示进行映射。

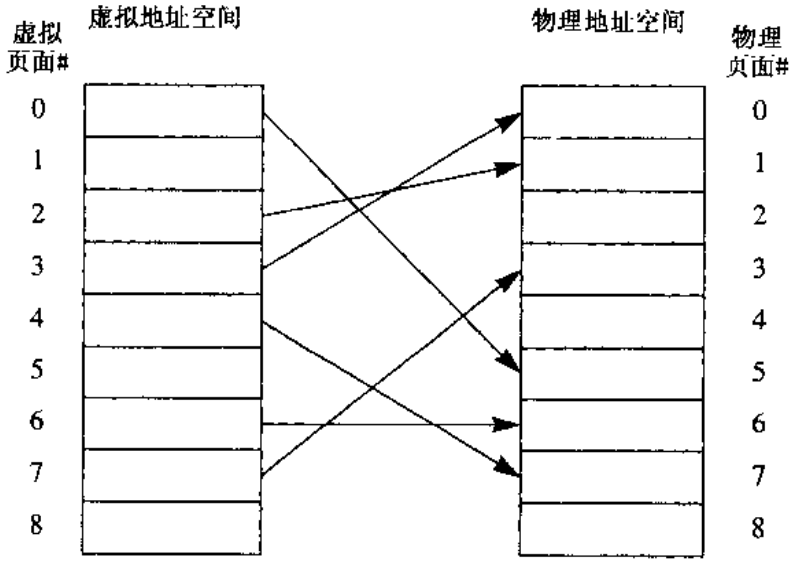


图 1-3　地址空间映射的例子

这幅图中的箭头显示了该进程内的虚拟页面被映射到了哪一个物理页面上。于是，比如说如果进程访问虚拟页面2，那么对该页面的引用将被映射到物理页面1上。从虚拟地址空间到物理地址空间的映射是由存储器管理单元（Memory Management Unit，MMU）来执行的，MMU负责进程所使用的全部地址。

每一个进程都有它自己的映射关系，这种映射关系与它相关，而且作为进程的现场的一部分保存起来。在进程运行的时候，内核将进程映射关系的描述提供给MMU。

注意，不是所有的虚拟页面都需要映射。例如，图1-3中的虚拟页面1、5和8就没有被映射到任何物理页面上。它们表示进程的地址空间中没有使用的页面，也可以是当前没有驻留在内存中的页面。如果一个进程试图访问后一种类型的页面，那么内核就要把相关的物理页面调入到内存中，并且把虚拟页面映射到新分配的物理页面上。

同样，不是存储器中所有的物理页面都由一个进程来使用。在图1-3中的例子里，物理页面2、4和8没有从这个进程到它们那里的映射关系。当前正在执行的进程将无法访问它们。这些物理页面可以属于系统中的其他进程，或者干脆就没有使用。无论是哪一种情况，肯定都不允许当前正在执行的进程访问它们。

通过把各个进程中的虚拟页面映射到同一个物理页面上，内核可以让多个进程共享特定的物理页面（这将在以后更详细地进行讨论）。

大多数MMU都有给每个映射关系关联一个访问权限（access permission）的能力。最常用的两种权限是读（read）和写（write）。这种功能可以让内核将正文页面（text page）映射为只读（read-only）的，同时允许对数据页面（data page）的读写访问。

1.4 现场切换

内核从执行一个进程转为执行另一个进程的操作称为现场切换（context switch）。这项操作包括保存当前进程的状态以便在将来可以恢复、选择一个要执行的新进程，以及把所保存的新进程的状态载入到硬件中。进程在现场切换时刻必须保存和恢复的最少状态是CPU寄存器的内容、PC（程序计数器）、堆栈指针、条件码，以及虚拟地址空间的映射关系。

在同一个进程内从一个线程切换到另一个线程的操作称为线程切换（thread switch）。因为进程不变，所以不需要改变地址空间的映射关系。只有上一段话中列出的寄存器和其他项需要保存和恢复。和进程的现场切换相比，线程切换所减少的开销是使用线程的另一个优点。总体而言，本书所介绍的内容主题都不需要考虑使用线程切换。

如前所述，每个进程都获得了一个独立的虚拟地址空间，这不但给它一个假象，以为它自己独自运行在计算机上，并且把它隔离开来，不受其他进程的干扰。在线程切换期间选择一个要执行的新进程时，必须彻底消除原来进程的地址空间映射，从而让新进程不能访问它。随后，新进程的地址空间被映射进来，于是它可以由这个进程来访问。

根据所用的特定硬件，可能要保存和恢复其他类型的状态。例如，高速缓存可能需要在现场切换时根据它们的实现进行管理（这是接下来几章讨论的主题）。内核必须确保一个进程的现场所要求的所有部分都被保存下来，以便在将来的某个时刻可以恢复它，从而可以继

续执行，就好像从未发生过现场切换一样。这是内核保持每个进程都独自在系统上执行的假象这一任务的一个重要方面。

1.5　存储管理和进程管理的系统调用

UNIX 系统为创建和消除进程以及为改变进程的地址空间提供了几个系统调用。本节简要回顾这些系统调用的内部操作和语义，因为高速缓存和多处理机对 UNIX 操作系统中处理进程地址空间的部分影响最大。

1.5.1　系统调用 fork

系统调用 fork 创建一个新进程。内核通过准确复制调用 fork 的进程的一个副本来创建一个新进程。调用 fork 的进程称为父进程（parent），新创建的进程称为子进程（child）。子进程不但获得了父进程的地址空间以及包括程序变量、寄存器、PC 等值在内的进程状态，而且获得了访问父进程拥有的全部 UNIX 系统服务的权利，比如父进程打开的文件。子进程是用一个控制线程创建的，这个控制线程和父进程中调用 fork 的线程是一样的。子进程一旦创建出来，它就独立于父进程而执行。在 fork 调用完成的时候，两个进程是相同的。两个进程现场的唯一区别在于 fork 系统调用本身的返回值。父进程返回的是子进程的进程 ID（pid），而子进程得到的是 0。这就让每个进程能够判断出它是父进程还是子进程。

因为复制一个较大的虚拟地址空间很花时间，所以要进行几种优化。首先，正文段一般由执行同一个程序（和/或共享相同的库）的所有进程只读共享。这意味着正文不需要在物理上复制给子进程。子进程只要共享父进程正在使用的同一个副本就可以了。因为 UNIX 内核不允许在正文段内擅自修改代码，所以才有可能共享正文（当出于调试的目的需要在正文中插入断点的时候，内核首先要创建一份正文的私用副本，以便执行相同程序的其他进程不会受到影响）。图 1-4 中父进程将要执行 fork 调用，其正文段是只读的，而它的其他页面可以进行读写访问。

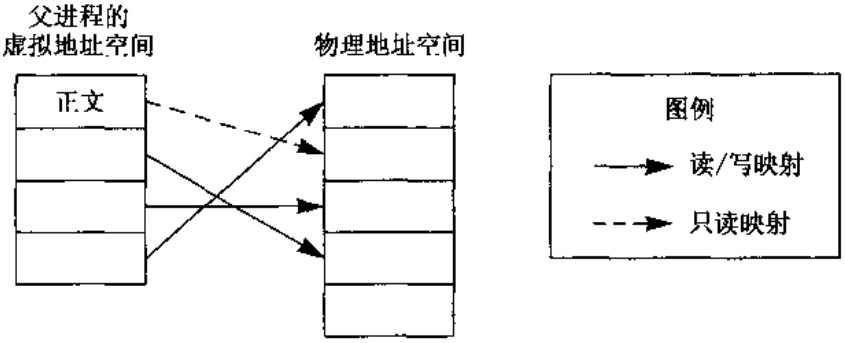


图 1-4　在调用 fork 之前：正文始终是只读的

接下来，几乎所有的实现都使用一种称为写时复制（copy-on-write）的技术来避免复制地址空间中剩余部分的大量内容。大多数 UNIX 进程在执行 fork 调用之后会立即调用 exec（在下一小节介绍）来执行新程序。这一操作丢弃了父进程的地址空间，所以在 fork 期间复制父进程空间而很快又丢弃它的做法会很浪费。相反，数据、bss 和堆栈都不在物理上进行复制，而是临时在父进程和子进程之间只读共享。这可以用图 1-5 来描述。

注意，两个进程在逻辑上仍然对页面拥有写权限。当父进程或者子进程试图写一个页面的时候就复制单独的页面。这样一来，只有要写入的页面才会按照需要来复制，如果子进程只需要在它执行 exec 或者 exit 之前写入其地址空间的一部分的话，那么就有可能节省大量的复制开销。只读、写时复制共享机制（copy-on-write sharing）只是用作一种高效的实现技术，对于涉及到的进程来说是透明的。

只要两个进程都没有企图修改数据，那么就会继续保持共享关系。当两个进程中的一个要写入一个只读页面的时候，就发生了一次保护陷阱（protection trap），内核会截获到这个陷阱。内核复制出进程正在尝试修改的单独页面的一个副本，用它来替换该页面在那个进程的地址空间中被共享的副本。这种做法只用于执行写操作的进程，其他进程的地址空间不受影响。采用这种方式时，可以在父进程和子进程之间共享尽可能多的地址空间，而仅仅根据需要复制进程修改页面的副本。这种处理对于两个进程都是透明的，从而造成复制了整个地址空间的假象。图 1-6 给出了图 1-5 中的子进程修改了它的第 3 个虚拟页面之后地址空间的状态。内核把这个页面的内容复制到了一个新的物理页面中，并且重新映射子进程的地址空间，指向该物理页面上。

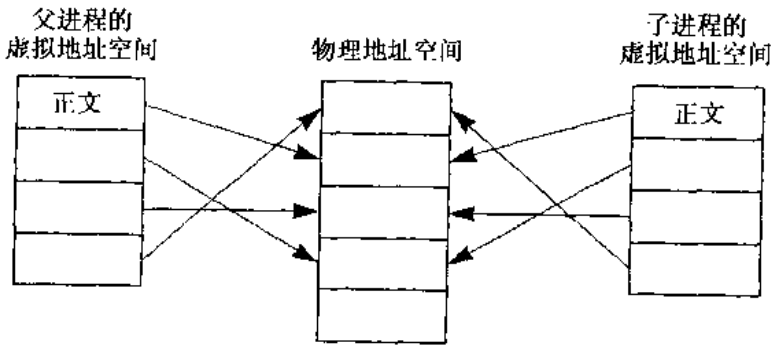


图 1-5 在调用 fork 之后，所有的物理页面都是只读共享的

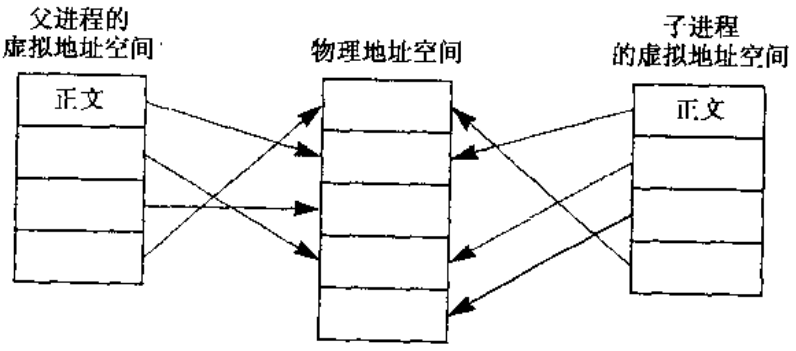


图 1-6 在子进程修改了一页后的地址空间映射关系

为了避免复制那些仅有一个映射关系的页面，内核要计算每个物理页面上的只读、写时复制的映射关系数量。所以，如果父进程现在要写入它的第 3 个虚拟页面，那么内核就会知道这个页面上没有其他的映射关系，只要把映射关系改为读写就行了，不需要复制该页。结果如图 1-7 所示。

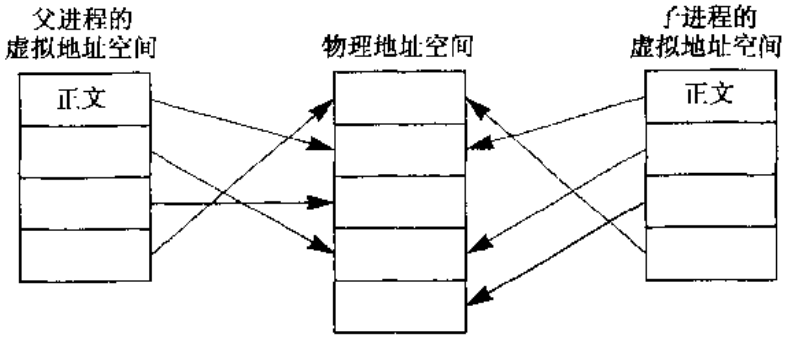


图 1-7　在父进程修改了第 3 个虚拟页面之后的地址空间映射关系

从应用的角度来看，系统调用 fork 是创建新进程的一种很方便的机制，因为它不带任何参数。因为子进程继承了父进程的全部状态，所以不需要像在其他操作系统中创建新进程那样给系统调用传递一组复杂的参数。子进程根据它从父进程那里得到的状态来判断出它应该执行什么任务。在大多数情况下，fork 的目标是创建一个新进程来执行新程序。要做到这一点，子进程通过打开或者关闭文件（例如，可能为了 I/O 重定向）来准备进程状态，然后用系统调用 exec 来执行新程序。

1.5.2　系统调用 exec

系统调用 exec 改变了一个进程正在执行的程序。只有调用 exec 的进程才会受到影响。exec 的参数是一个文件名（该文件包含要执行的新程序）和一组要传递给新程序的参数和环境变量。执行 exec 系统调用的进程保留它与大多数 UNIX 系统服务相关的状态信息，比如，它打开的文件、它的当前目录和主目录，等等。它的状态中和程序本身有关的部分，比如它的寄存器内容、变量、PC（程序计数器）以及地址空间，都要用新程序的来替换。更具体地说，原来程序的正文、数据、bss 和堆栈以及诸如共享存储的其他存储对象都会被丢弃，而要为新程序创建新的虚拟地址空间。新程序的正文和初始化数据则从指定的文件中读入，内核将地址空间内的空间分配给 bss 和堆栈。进程内单个线程的 PC（程序计数器）被设定在新程序的起始地址。当系统调用执行完毕的时候，原来的程序在进程中就不复存在了，新程序开始执行。新程序可以访问 exec 系统调用之前进程所打开的文件，因为这些文件都是和进程相关联的，而不是和程序相关联的。

如前所述，系统调用 exec 往往在 fork 之后执行。最常见的情形就是 UNIX 系统的命令解释器，它可以创建新进程来运行每条命令。命令解释器调用 fork 创建新进程，然后子进程调用 exec 运行该命令。

1.5.3 系统调用exit

系统调用exit会让调用它的进程（以及它的所有线程）终止执行。该系统调用在程序完成它的执行过程之后，并希望终止的时候使用。一个进程还有可能采用系统调用kill来终止另一个进程（假定该进程有适当的权限）。如果发生了无法恢复的错误，系统也可以终止一个进程。在所有的情况下，内核终止一个进程以及在事后进行清理工作的步骤都是相同的。

要终止一个进程，内核必须丢弃进程的地址空间，取消进程正在使用的内核服务，例如，关闭进程留下的任何打开的文件。此刻，进程暂时以一种称为僵进程（zombie）的形式存在，僵进程是一个UNIX的术语。这就提供了一种便捷的手段，让父进程在有机会采用系统调用wait读取子进程的退出状态之前保持进程之间的父子关系（僵进程与本书的讨论无关，不再深入研究）。最后，释放代表进程本身的内部的内核数据结构。此刻，进程就不复存在了，内核执行一次现场切换，选择另一个要执行的进程。

1.5.4 系统调用sbrk和brk

系统调用sbrk和brk都是供一个进程用来分配或者收回它的bss段空间。这两个系统调用都以bss段的BReaK address（断开地址）而得名。这是在bss段内进程能够访问的最大合法地址。断开地址和堆栈顶部地址之间的虚拟存储区不会被映射到任何物理存储器上，进程无法访问到它们（眼下忽略共享存储和映射文件）。系统调用sbrk和brk能够让进程改变它的断开地址，从而增长或者缩小bss段的大小。系统调用sbrk接受一个代表断开地址增量变化的有符号数值作为参数，而系统调用brk接受一个成为新断开地址的虚拟地址作为参数。

如果进程请求增大bss段，那么内核就分配正好在原来的断开地址之上的虚拟地址，从而让进程能够访问到这部分地址空间。新分配的bss内存都被定义用0来填写。bss段只能向更大的地址增长，它的起始地址是固定不变的。支持新分配的虚拟内存的物理存储器则根据需要来分配，因为它是由进程来引用的。如果缩小了bss，那么在新老断开地址之间地址范围内的虚拟和物理存储器都将被释放。访问权限也变了，于是进程再也不能访问它了。

1.5.5 共享存储

有些UNIX系统的实现提供了一种能够让两个或者两个以上的进程共享一个物理存储器区域的系统服务。这通常叫做共享存储段（shared memory segment）。它是将同一个（或者多个）物理页面映射到两个或者更多进程的虚拟地址空间中来实现的。物理存储器的共享区无需在所有的进程中都出现在相同的虚拟地址上，每个共享区都可以按照其选择将它附加到不同的虚拟地址上。共享存储通常被附加到进程内bss和堆栈段之间未用的虚拟存储区中。

共享存储能够充当一种高速的进程间通信（interprocess communication，IPC）机制，因为进程可以通过共享存储传递数据，既不需要执行系统调用，也不需要牵扯到内核。当一个进程把数据写入共享存储中的时候，共享同一共享存储段的其他进程立即就能访问到这些数据，因为它们都共享相同的物理页面。

1.5.6　输入输出操作

在考虑存储器操作的时候，I/O（输入输出）的影响是很重要的。从用户进程请求 I/O 操作的系统调用有两个：read 和 write。这两个系统调用把数据从进程的地址空间传送到一个文件或者设备（write），或者反过来（read）。有些 UNIX 实现提供了额外的 I/O 系统调用，比如 readv、writev、getmsg 和 putmsg（就本书所介绍的主题而言，这些系统调用的作用同 read 和 write 是一样的，所以我们只讨论这两个系统调用）。I/O 的类有两种：有缓冲的（buffered）和未缓冲的（unbuffered）。

在内核中，对于特定文件类型的 I/O 是有缓冲的。内核和用户进程地址空间之间的数据通过一个复制操作来传输。缓冲机制（buffering）的优点是它允许用户程序不必知道物理 I/O 设备的特征属性。程序不需要关心块（block）或者记录（record）的大小，或者任何对齐的限制。例如，磁盘往往以扇区（sector）来存取，这意味着 I/O 必须从一个扇区的边界开始，并且包括多个扇区的大小。当一个用户程序读取有缓冲的文件时，它只要指定它希望 I/O 从这个文件内开始的字节偏移量以及它想读取的字节数就可以了。为了保持文件抽象的概念，内核把用户的字节偏移量转换为包含数据的相应扇区。一个或者更多扇区从磁盘读入到内核缓冲区中，用户想要读取的数据部分就被复制到用户的缓冲区中。

未缓冲的 I/O 则绕过了这种复制操作。用户进程可以使用这种类型的 I/O，它在 UNIX 的行话中称为原始 I/O（raw I/O）。术语“原始 I/O”来源于这一事实，即通过一个缓冲区进行复制的数据被认为是经过处理的，或者说加工过的（cooked）。因此，没有复制的数据则被认为是“原始的”。在采用原始 I/O 的情况下，I/O 设备使用 DMA（direct memory access，直接存储访问）操作直接把数据传送到用户缓冲区。内核在有缓冲的 I/O 期间所缓冲的数据最终也使用 DMA 传送到设备上，或者从设备上传送到缓冲区中。所以，既可以在用户地址空间也可以在内核地址空间执行 DMA 操作。

1.5.7　映射文件

许多 UNIX 系统的实现都提供了将文件映射到一个进程地址空间内的功能。一旦文件被映射到进程地址空间内，这个文件就可以作为地址空间内一段连续的字节区直接访问。这就让进程可以使用内存的上载和保存操作而不是系统调用 read 和 write 来访问文件的内容。映射文件在逻辑上和共享存储器类似：映射到同一文件的多个进程可以选择共享映射，从而让一个进程所做的改动也可以出现在其他进程的地址空间内。

1.6　小　　结

本章回顾了 UNIX 内核的基本原理。UNIX 系统是一种多用户、多任务的操作系统，它通过向进程提供与机器无关的抽象服务，从而在 UNIX 实现之间提供了高度的程序可移植性。程序的执行被限制在保持程序当前状态的进程内，这些状态包括虚拟地址空间、程序的变量值以及硬件状态。内核给每个进程提供了一个环境，让这个环境显得就好像该进程是系统中

正在执行的唯一进程那样。这主要是赋于每个进程自己的虚拟地址空间来实现的。用户程序通过执行系统调用来请求内核的服务。系统调用可以创建新进程（fork），改变进程正在执行的程序（exec），以及终止进程（exit）。还可以使用其他许多系统调用，其中包括动态分配未初始化数据的系统调用（brk/sbrk）、使用共享存储的系统调用，以及执行 I/O 的系统调用（read 和 write）。

有兴趣学习更多有关 UNIX 系统实现知识的读者可以参考本章末尾提供的参考文献。

1.7 习　　题

1.1　如果一个进程试图把数据保存在它的正文段中，将会发生什么情况？

1.2　UNIX 内核会根据需要动态地增加一个进程的堆栈，但是它绝不会尝试缩小它。考虑这样的情况，一个程序调用 C 子程序，在堆栈上分配占用 10Kb 的一个局部数组。内核将会扩大堆栈段来容纳这个数组。当子程序返回的时候，这个空间理论上应由内核释放，但是它不会。解释为什么此刻有可能缩小堆栈，以及为什么 UNIX 内核实际上却没有缩小它。

1.3　如果一个新创建的子进程（进程 C）立即调用 fork 又创建了一个它自己的子进程，那么画出一幅图（采用类似图 1-5 的方式），显示出所有 3 个进程怎样共享物理存储器。假定使用了写时复制技术。如果进程 C 现在退出，那么共享机制会受到什么样的影响？

1.4　假设一个进程有两页正文、一页数据、三页 bss 和二页堆栈，如果内核使用写时复制技术，那么在 fork 期间将会为子进程分配多少物理页面？如果它不使用写时复制技术，那么要分配多少？解释原因。

1.5　如果一个父进程和一个子进程采用写时复制技术共享其堆栈段的所有页面，子进程调用一个子程序让它的堆栈扩大到超出所分配的空间，那么内核将动态地扩大了进程的堆栈。阐述这会给父进程的堆栈和地址空间造成的影响（如果有的话），以及为什么。

1.6　一个子进程使用写时复制技术和父进程共享它的数据段，如果它执行有缓冲的 read 系统调用，而缓冲区位于子进程的数据区内时，将会发生什么情况？如果它代之以执行 write 系统调用，又会发生什么情况？

1.7　代之以使用未缓冲的（原始的）I/O 来重做习题 1.6。

1.8　假定一个进程正在使用一个共享库（也就是说，进程从映射到它的地址空间中的共享库获得正文和数据）。如果进程调用 fork，那么内核应该怎样对待共享库的区域？假定内核支持写时复制技术。

1.9　如果一个使用共享存储的进程执行一次 fork 系统调用，那么内核应该怎样处理共享存储区？假定内核支持写时复制技术。

1.10　如果两个进程采用写时复制技术完全共享它们的地址空间，子进程用系统调用 sbrk 释放了它的部分 bss，然后又有另一次 sbrk 调用将断开地址设定为它原来的值，那么父进程和子进程之间的写时复制关系将会是什么样的？

1.11　考虑一个进程，它执行一次系统调用 exec 调用了一个程序，该程序的 bss 段从地址 0x10000 开始，到地址 0x20000 结束。进程将值 0x1234 保存在位于 0x15000 处的字里。接

下来，它执行一次系统调用 brk，将断开地址设定为 0x12000，然后再执行另一次系统调用 brk，将断开地址设定为 0x18000。如果该进程现在读取地址 0x15000，将会看到什么值？

1.12　如果两个进程采用写时复制技术完全共享它们的地址空间，其中一个进程退出，那么另一个进程的只读映射将会发生什么情况？

1.13　描述带有多个线程的一个进程与使用共享存储的多个进程（其中每个进程都有一个线程）之间的不同之处。

1.8　进一步的读物

[1] Accetta, M.J., Baron, R.V., Bolosky, W., Goulub, D.B., Rashid, R.F., Tevanian, A., and Young, M.W., "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer USENIX Conference*, July 1986, pp.93-112.

[2] Andleigh, P.K., *UNIX System Architecture*, Englewood Cliffs, NJ: Prentice Hall, 1990.

[3] Bach, M.J., *The Design of the UNIX Operating System*, Englewood Cliffs, NJ: Prentice Hall, 1986.

[4] Bic, L., and Shaw, A.C., *The Logical Design of Operating Systems*, Englewood Cliffs, NJ: Prentice Hall, 1982.

[5] Bodenstab, D.E., Houghton, T.F., Kelleman, K.A., Ronkin, G., and Schan, E.P.,"UNIX Operating System Porting Experiences," *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, October 1984, pp. 1769-90.

[6] Bourne, S.R., *The UNIX System V Environment*, Reading, MA: Addison-Wesley, 1987.

[7] Coffman, E.G., and Denning, P.J., *Operating Systems Theory*, Englewood Cliffs, NJ: Prentice Hall, 1973.

[8] Crowley, C., "The Design and Implementation of a New UNIX Kernel," *Proceedings of AFIPS NCC*, Vol. 50, 1981, pp. 1079-86.

[9] Deitel, H.M., *An Introduction to Operating Systems*, Reading, MA: AddisonWesley, 1990.

[10] Denning, P.J., "Virtual Memory," *ACM Computing Surverys*, Vol. 2, No. 3, September 1970, pp. 153-89.

[11] Feng, L., ed., *UNIX SVR4.2 Operating System API Reference*, Englewood Cliffs, NJ: Prentice Hall, 1992.

[12] Gingell, R.A., Moran, J.P., and Shannon, W.A., "Virtual Memory Architecture in SunOS," *Proceedings of the 1987 Summer USENIX Conference*, June 1987, pp. 81-94.

[13] Kay, J., and Kummerfeld, B., *C Programming in a UNIX Environment*, Reading, MA: Addison-Wesley, 1989.

[14] Kernighan, B.W., and Pike, R., *The UNIX Programming Environment*, Englewood Cliffs, NJ: Prentice Hall, 1984.

[15] Kernighan, B.W., and Ritchie, D.M., *The C Programming Language, Second Edition*, Englewood Cliffs, NJ: Prentice Hall, 1988.

[16] Krakowiak, S., *Principles of Operating Systems*, Cambridge, MA: M.I.T. Press, 1988.

[17] Leffler, S.J., McKusick, M.K., Karels, M.J., and Quarterman, J.S., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Reading, MA: Addison-wesley, 1989.

[18] Open Software Foundation, *Design of the OSF/1 Operating System*, Englewood Cliffs, NJ: Prentice Hall, 1993.

[19] Plauger, P.J., *The Standard Clibrary*, Englewood Cliffs, NJ: Prentice Hall, 1992.

[20] Ritchie, D.M., and Thompson, K., "The UNIX Time-sharing System," *The Bell System Technical Journal*, Vol. 57, No. 6, July-August 1978, pp. 1905-30.

[21] Ritchie, D.M., "The Evolution of the UNIX Time-Sharing System," *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, October 1984, pp. 1577-94.

[22] Rochkind, M.J., *Advanced UNIX Programming*, Englewood Cliffs, NJ: Prentice Hall, 1985.

[23] Stevens, W.R., *Advanced Programming in the UNIX Environment*, Reading, MA: Addison-Wesley, 1992.

[24] Tanenbaum, A.S., *Operating Systems: Design and Implementation*, Englewood Cliffs, NJ: Prentice Hall, 1987.

[25] Thompson, K.,"UNIX Implementation," *The Bell System Technical Journal*, Vol. 57, No.6, July-August 1978, pp. 1931-46.

第一部分

高速缓存存储系统

第2章 高速缓存存储系统概述

高速缓存存储系统（cache memory system）是高速存储器，它能够利用局部引用特性来提高系统性能。本章解释了高速缓存的基本术语、规划和操作，阐述了高速缓存是如何配合主存储器运行的，以及高速缓存中的数据是如何定位的。本章还介绍了散列算法（hashing algorithm）、缺失处理（miss processing）、写策略（write policy）、替换策略（replacement policy）以及组相联（set associativity）。本章主要着眼于单处理机系统的高速缓存，多处理机高速缓存将在第三部分中进行介绍。到本章结束的时候，读者会对高速缓存的操作非常熟悉，可以在后面的章节中开始研究操作系统的问题了。

2.1 存储器层次结构

高速缓存是一种高速存储系统，它保存有主存储器很小的一个子集的内容。它的物理位置介于主存储器和 CPU 之间。因为它比主存储器速度快，所以如果频繁使用的指令和数据能够保存在高速缓存中供 CPU 存取，那么就有可能提高系统的性能。图 2-1 给出了包含高速缓存的一个计算机系统的逻辑框图。

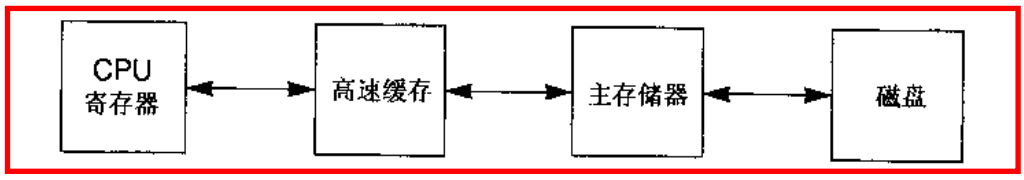


图 2-1 高速缓存相对于其他系统组件的位置

高速缓存利用局部引用特性来改善系统性能。局部引用是大多数程序所表现出来的一种属性，在这些程序中，在一个特定的时间段内，程序指令和数据的一个相当小的子集被频繁地重复引用。如果能够把程序当前的局部引用保存在高速缓存中，那么程序就能执行得更快，因为高速缓存可以比主存储器更快地提供指令和数据。

高速缓存只是存储器层次结构（memory hierarchy）中的一个组成部分。这个存储结构的一端是磁盘存储器，它具有非常高的密度，每比特的位成本很低，而存取速度则（相对）较慢。另一端是 CPU 内的寄存器，在数量上只有几个，寄存器每比特位成本很高，但是存取速

度极快。随着我们从磁盘存储器过渡到寄存器，成本逐渐增加，存取速度逐渐加快，而密度却逐渐降低。

虚拟存储器调页系统（paging sytem）已经显示出只占系统中所有进程所使用的全部虚拟存储空间大小若干分之一的主存储器系统是怎样提供良好的整体性能的。局部引用通过只要求一个进程当前的工作集驻留内存使之成为可能，工作集是那些包含有进程当前局部引用位置的存储页面。进程的另一部分地址空间可以保存在磁盘上，直到需要的时候再加载。

局部引用特性延伸到了比工作集更细的粒度上。在一个进程工作集的页面内，肯定会有一组指令和变量在一段极短的时间内被一个程序频繁地重复引用。例如，考虑图 2-2 中计算矩阵 a 和 b 乘积的 C 代码小片段。假定 a 是一个 m x r 矩阵，而 b 是一个 r x n 矩阵，c 的全部元素都初始化为 0。

```
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < r; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

图 2-2 执行矩阵相乘的代码片段

在这段代码执行的同时，它重复引用了 3 个循环内的指令、矩阵的元素以及循环计数器。这就是当这段代码在执行时程序的局部引用。因为矩阵包括的数据比寄存器能保存的数据多，所以如果没有高速缓存，CPU 就不得不重复引用主存储器，从而将这部分程序的执行速度限制在了主存储器的存取时间上。

图 2-2 中有两种类型的局部性：时间的（temporal）和空间的（spatial）。时间局部性是程序有可能重复使用近期引用过的项的属性。例如，在执行最里面的一层循环期间，数组 c 的同一个元素会被引用两次，循环变量 i、j 和 k 也是如此。类似地，在所有三层循环中的指令也会被重复引用。所有这些引用都体现出了时间局部性。空间局部性是程序有可能重复使用前面附近引用过的项的属性。由于 C 语言中的数组都是按照行的顺序保存的，所以数组 a 和 c 都体现出了空间局部性，因为会在后面的迭代中访问到行中的下一个邻接元素。类似地，串行程序的执行表现出了高度的空间局部性。

主存储器系统速度的提高和成本的降低并不能跟上如今高速 CPU 的发展速度。如果 CPU 的速度和主存储器的存取时间远远不能平衡，这意味着存储器要比 CPU 执行指令以及加载和保存数据的能力慢得多，那么 CPU 就会被限制在存储器的速度上。在这样的情况下，提高 CPU 的速度对于改善系统的整体性能来说几乎没有什么帮助，因为存储系统仍然是限制因素。虽然人们可以制造大规模的高速主存储系统，它们的存取时间能够同 CPU 加载和保存数据的能力相媲美，但是这种存储系统对于高端大型机和超级计算机以外的系统来说往往太贵了。当今很多系统设计人员的另一种选择是使用高速缓存。

通过利用细粒度的局部引用特性，高速缓存能够弥补速度较慢的主存储系统和快速 CPU 之间的缺口。在存储器层次结构中，每一级局部引用属性的应用情况如图 2-1 所示。正如主存储器保存了程序的一个子集（工作集）那样，寄存器保存了当前运算的操作数，高速缓存保存了正在工作的指令和变量集，在那之上形成了细粒度的局部引用。由于有了局部引用，高速缓存只需要拥有主存储器的很小一部分就能够发挥效用。由于它的规模相当小，所以实

际上可以使用比主存储器所能用的速度更快的存储设备，因为并不需要太多的高速缓存。于是，高速缓存的高速度同局部引用特性的使用相结合就能大大提高系统性能，而且在经济上也很划算。

2.2　高速缓存基本原理

因为几乎所有的程序都体现出了局部引用特性，所以在 PC（个人计算机）到超级计算机的各种系统上都能找到高速缓存。甚至在现今大多数微处理器芯片和存储器管理单元（memory management unit，MMU）芯片上也可以找到集成的高速缓存。如果在微处理器或者 MMU 芯片上没有包含高速缓存，那么它一般会在 CPU 板（称为外部高速缓存）上。靠近 CPU 降低了 CPU 和高速缓存之间的存取时间。如果高速缓存位于一块独立的板上，访问它需要总线处理的话，那么就会大大增加时延（latency），从而带来系统性能上的相应损失。有些系统既使用片上高速缓存，也使用外部高速缓存。

高速缓存的大小从几个字节到数百 K 字节都有，在大规模的系统上甚至有 1Mb 以上的高速缓存。例如，如今的片上高速缓存一般介于 4Kb 到 16Kb 之间。外部高速缓存要大一些，介于 128Kb 到 4Mb 之间。随着时间的推移，高速缓存的规模正在逐步扩大。例如，Intel 80386 没有片上高速缓存，80486 有 8Kb 的高速缓存，而 Pentium 则有两块 8Kb 的高速缓存。外部高速缓存已经从 MIPS R2000 的 64Kb 增加到 R4000 的 4Mb。

一般而言，高速缓存越大，性能提升就越多，因为有更大的一个存储了集被高速缓存起来供高速访问。正如随后的各章中将要看到的那样，高速缓存的规模并不能改变操作系统必须正确管理高速缓存的问题，而只能改变要使用的特定算法。高速缓存的性能将在 2.11 节里进一步讨论。

一个系统可以使用分离的指令高速缓存和数据高速缓存。这能够进一步提高系统的性能，因为 CPU 可以同时从指令高速缓存取得指令，而从数据高速缓存加载或者存储数据（参见 2.10 节）。正如第 6 章中将要看到的那样，可以使用多级高速缓存给存储器层次结构增加额外的层次（接下来的几章将集中介绍单级高速缓存）。

2.2.1　如何存取高速缓存

高速缓存的实现使得它们的存在对于用户程序来说基本上甚至完全地被忽略了。大多数实现的目标是隐藏高速缓存管理的全部细节，不论是硬件上的还是操作系统上的高速缓存（在后面的章节中阐述），以便获得用户应用的可移植性。这种方法确保了无需修改程序，应用程序就可以移植到具有不同高速缓存组织、不同高速缓存规模的不同系统上，或者移植到没有高速缓存的系统上，这是如今在市场上的一个重要获益点。如此一来，程序可以像以前那样继续通过地址引用存储器。访问高速缓存不需要特殊的编码技术或者寻址模式。高速缓存可以通过控制高速缓存的硬件自动访问。物理高速缓存（physical cache）这种高速缓存的组织甚至对于操作系统来说都是透明的。这就有可能给原本在设计上没有高速缓存的体系结构

比如（IBM 370 和 IBM PC），增加高速缓存，而且仍然能够和现有的系统软件保持兼容性（物理高速缓存将在第 6 章详细介绍）。

因为高速缓存只保存主存储器的一个子集，所以需要有几种方式来确定当前驻留在高速缓存中的应该是存储器的哪些部分。存储器的那些部分就认为被高速缓存了。这一点是通过在高速缓存中用它的的主存储器地址来标记该数据的方法做到的（对于指令高速缓存中的指令来说也是如此）。接着，硬件能够通过检查高速缓存中的数据标记来判断一个特殊的存储单元是否被高速缓存了。于是，比如说当 CPU 发出一个它想要读取的主存储器地址时，这个地址就被发送给高速缓存，硬件就开始搜索高速缓存来寻找相应的数据（参见图 2-3）。如果在高速缓存中找到了它，那么就称为一次高速缓存命中（cache hit）。如果没有找到相应的数据，那么就称为高速缓存缺失（cache miss）。高速缓存命中与高速缓存缺失的频率之比就称为命中率（hit ratio），它以引用命中的百分比或者命中发生的概率（90%的命中率和 0.9 的命中概率是等价的）来表达。命中率越高，系统性能就越好。

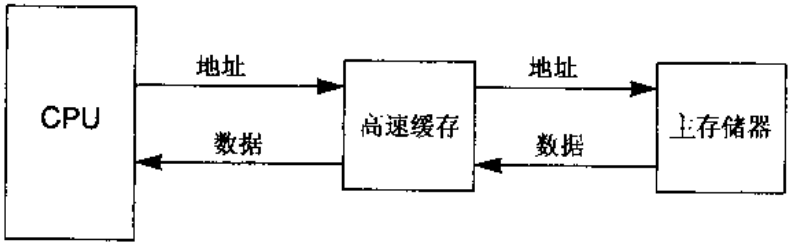


图 2-3 CPU 通过高速缓存读取数据

如果发生了一次命中，那么数据被返回给 CPU，就好像它是从主存储器中读取的一样。如果没有命中，那么就把地址传递给主存储系统，在那里访问寻址单元。在这种情况下，数据既返回给 CPU 也返回给高速缓存。在一次缺失之后数据被保存在高速缓存中，以便利用时间局部性。此时也可以把 CPU 读取的一段数据周围的数据都载入高速缓存，从而也能利用到空间局部性（正如后面将要讨论的那样，在高速缓存缺失期间载入的数据量取决于实现）。因为大多数程序都表现出了局部性，所以 90%或者更高的命中率并非罕见。

在高速缓存的设计中，要考虑的重要一点是用一个标记确定多少数据。例如，独立地标记高速缓存中的每一个字节代价太大。因此，来自主存储器的一个或者更多连续的字被组织到一起形成了一个高速缓存行（line）或者块（block），并且给每一行关联一个标记。所以一个完整的高速缓存行是由一个标记和一段数据组成的，但是高速缓存行的大小是指数据部分的字节数（一般并不包括标记的大小）。片上高速缓存行的典型大小范围是从 16 字节到 32 字节。因为行中数据部分的字节是从连续的存储器单元来的，所以标记只需要包括第一个字节的地址即可；其他字节的地址可以用它们在行内的位置来推算。

除了地址之外，一行的标记部分还包括控制信息（control information）。标记部分始终都要加上一位，作为有效位（valid bit），表明相关的高速缓存行是否投入使用以及是否包含有效的数据。有效位必须为置位，且标记必须吻合，才能出现命中。在系统复位或者启动期间初始化高速缓存的时候，所有的有效位都被清除，于是一开始的时候，要到主存储器内引用全部的指令和数据。

在标记中保存的另一个常见的标志是修改位（modified bit）。正如 2.2.5 节所讨论的那样，在 CPU 把数据保存到使用写回高速缓存机制（write-back caching）的一个高速缓存中的时候就设置这一位。在标记中也可以出现其他依赖于实现的信息，比如键（key），它是第 4 章的主题。

在发生一次高速缓存缺失的时候，就从主存储器中读取填满整个高速缓存行所需要的字节数，并且载入高速缓存。因为反映整行状态的只有一个有效位，所以必须这样做。例如，如果高速缓存行的大小是两个字长，那么在一次高速缓存缺失时不可能只读取 CPU 所引用的那一个字。如果没有读取两个字，那么高速缓存行的一半就会包含无效的数据。一般说来，从主存储器中读取附加的字并不会影响性能，因为大多数现代的主存储系统的设计都能一次读取或者写入多个字。

有些实现选择使用非常长的高速缓存行（128~256 字节或者更多）。这样做的优点是它减少了标记需占的存储器数量，因为一个标记现在能够涵盖行里数据部分中的字节数更多。因此，为保存同等数量的数据，高速缓存所需的标记更少。长高速缓存行的缺点是，将整个行传入和传出主存储器所需要的时间开始变得明显长了起来。为了缓解这一问题，有些实现将一个高速缓存行划分成了多个子行（subline），每个子行都有它自己的有效位。于是每个子行都能独立地进行处理，就好像它是一个单独的高速缓存行一样，差别在于只有一个地址标记涵盖一行中所有的子行。这意味着所有的子行包含来自连续主存储器地址中的数据。每个子行的地址由它在整个高速缓存行中的位置和高速缓存行的标记就可以推算得到。幸运的是，子行的使用对于操作系统来说是透明的，所以它不需要成为随后讨论中的一个考虑因素。

2.2.2　虚拟地址还是物理地址

高速缓存可以设计成通过数据或者指令的虚拟地址或是物理地址来访问。类似地，标记的设计也可以是连同其他信息一起，要么包括虚拟地址，要么包括物理地址。在设计硬件的时候，选择虚拟地址还是物理地址就确定下来了，并且对高速缓存管理技术有很大的影响，而操作系统必须采用这些技术向用户程序隐藏高速缓存的存在。这些复杂的问题将是后续各章的讨论主题。对于本书剩下的内容来说，不需要考虑访问高速缓存所采用的地址类型。采用其中任何一种地址类型对于下面介绍的主题都有相同的影响。接下来的几小节只是简单地称为“地址”。

2.2.3　搜索高速缓存

如果给定 CPU 想要的数据的地址，那么必须快速地搜索高速缓存来查找那个数据，因为高速缓存的全部目的就是要比主存储系统更快地返回数据。搜索技术还必须简单，以便高速硬件的实现既切合实际，又经济划算。比如说，线性搜索技术就不适用于高速缓存，因为除了最小的高速缓存之外，它们对于所有的高速缓存来说都太慢了。它们还难以在硬件上实现，因为它们需要一个状态机（state machine）或者定序器（sequencer）来保存搜索的当前状态。大多数高速缓存代之以使用一种简单的散列表（hash table）技术来进行搜索。

为了搜索一个高速缓存，来自 CPU 的地址经散列处理生成一个索引（index），这个索引指向高速缓存中的一个或者多个位置，如果相应的数据被高速缓存了，那么就会保存在这些位置上。无论采用什么样的散列算法，都会出现不同的地址产生相同索引值的现象。于是必须用这些位置上的标记（它们包含着那些位置上正在高速缓存的数据的地址）和 CPU 所提供的地址进行比较。如果一个标记和来自 CPU 的地址相吻合，那么就发生一次命中；否则，就出现一次缺失。对于高速缓存来说，散列是一种很有用的技术，因为它将搜索限定在了包含一个或者多个位置的小集合中（除了在全相联的高速缓存中之外，这将在以后介绍，在全相联的高速缓存中不使用散列技术）。接下来，硬件会快速地并行搜索这些位置。对于大规模的高速缓存来说，这些技术格外重要，因为在这些高速缓存中只有足够的时间去搜索一小部分高速缓存。上述的所有活动都在硬件中执行，无需软件的介入。

2.2.4 替换策略

因为高速缓存要比主存储器小，所以在高速缓存缺失操作期间载入新数据时有些数据必须丢弃。要丢弃的数据是按照高速缓存的替换策略（replacement policy）来进行选择的（该策略与实现有关）。一旦被选中，那么高速缓存中要丢弃的数据就被新数据所替换。和数据相关联的标记也改成新数据的地址，从而在高速缓存中正确地标识它。

高速缓存所采用的替换策略往往相当简单。虽然在存储管理和虚拟存储调页系统中所看到的页面老化（page aging）和替换（replacement）技术从理论上说也可以应用于高速缓存，但是它们在硬件上实现起来过于复杂。它们经常需要大量的状态或者历史信息，而在高速缓存中使用的昂贵的高速存储器数量有限，没有充足的空间来保存它们。典型的高速缓存替换策略有 LRU（Least Recently Used，最近最少使用）、伪 LRU（pseudo-LRU，一种 LRU 的近似）以及随机替换。这些策略将伴随本章后面讨论的不同高速缓存组织进行介绍。幸运的是，和前面讨论过的那些高速缓存访问的方面一样，替换策略对于软件来说也是透明的。

2.2.5 写入策略

当 CPU 保存数据的时候，大多数实现都把数据直接保存到高速缓存中。将数据保存在高速缓存中有助于改善系统性能的原因有两个。第一，由于时间局部性，被保存的数据在写入之后会被频繁地重新读取，所以就提高了命中率。第二，在高速缓存中使用的速度更快的存储设备保存数据的速度要比主存储器快。这就解放了 CPU，让它能够比其他方式可能的速度更快地开始下一次数据载入或者保存操作。写入高速缓存中的数据也可以同时写入主存储器。高速缓存的写入策略（write policy）也叫做更新策略（update policy），表明了数据是怎样保存到高速缓存和主存储器中的。

要把数据保存到高速缓存中，必须搜索高速缓存，看看高速缓存内是否已经包含有和写入的地址相关的数据。此刻使用了与从高速缓存中读取数据时相同的搜索技术。先考虑在保存期间出现一次命中的情况。在这里，CPU 写入的数据替换了高速缓存行内的老数据，以便利用时间局部性。

虽然高速缓存是用来自 CPU 的新数据更新的，但是该数据可以写入主存储器，也可以不写入主存储器，这取决于所采用的写入策略。两种可能的写入策略是写直通（write-through）和写回（write-back）（也叫做复制回（copy-back））。如果一个高速缓存正在使用写直通策略，那么来自 CPU 的数据既会写入高速缓存也会写入主存储器。写直通策略得名于存储器层次结构的组成（参见图 2-1），它必须“直通”过高速缓存到达主存储器。MIPS R2000/R3000 和 Intel 用于 80486 的 82495DX 外部高速缓存控制器都使用写直通高速缓存机制。写直通策略的效果是存储器始终保持在“最新”状态，这意味着在高速缓存中的数据和存在于主存储器内的数据副本完全相同（也叫做和高速缓存保持一致）。这种策略的缺点是每次 CPU 写入操作都需要有一个主存储器周期，有可能会降低系统的速度。

另一种策略是写回（write-back）。此时来自 CPU 的数据还是按照以前那样被写入高速缓存，但是并不写入主存储器，直到在行替换或者操作系统明确要求写回主存储器期间才写入主存储器。这就消除了采用写直通高速缓存机制时，如果在某个地址的数据被高速缓存，同一个地址被写入若干次时所造成的额外的主存储器周期开销。写回策略的缺点是，主存储器中的内容相对于高速缓存而言变成了“过时的”或者说不一致的。为了重获一致性，往往需要操作系统的介入。

例如，考虑采用写回高速缓存的 CPU 在执行一个程序中 i = i + 1 这条语句时会发生什么情况（假定 i 的值以前没有被高速缓存过）。如果在执行这条语句之前，i 在主存储器中的值为 1，那么当 CPU 试图读取 i 的当前值的时候，它就不会在高速缓存中命中，并且要将值 1 载入到高速缓存中，然后返回给 CPU（图 2-4a）。接着，CPU 给 i 的值加 1，把 i 的值 2 写回。写入操作导致在高速缓存中发生一次命中，并且 i 在高速缓存中的值被更新为 2。此刻写入操作完成时高速缓存中 i 拥有新值，但主存储器内仍然保持着 i 的旧值 1（图 2-4b）。在高速缓存内 i 的新值被认为相对于主存储器内的值被“修改”过了。

被高速缓存的值	主存储器
i: 1	i: 1

(a) 从存储器读取 i 之后的状态

被高速缓存的值	主存储器
i: 2	i: 1

(b) i 加 1 之后的状态

图 2-4　i 在高速缓存内的值和在主存储器内的值

必须确保 i 在主存储器内的过时值不会被程序访问到，否则将会出现无法预测的结果。类似地，也必须确保多处理机系统上的其他进程不会使用过时的存储器内的值（这将在第三部分进行讨论）。只要 i 的新值保存在高速缓存内，那么程序就不会访问到 i 的过时值，因

为程序总是可以在访问主存储器之前在高速缓存中命中。被修改的数据将保持被高速缓存，直到该行被替换为止。

在随后的缺失操作期间内的任何时刻，都可以替换写回高速缓存中的数据。被修改的数据不能被简单地丢弃，因为程序最终会在下一次引用时访问到在主存储器内的原来的过时值。因此，在替换之前，要替换的已被修改过的数据会由高速缓存硬件自动地写回到主存储器中。为了区分高速缓存中需要在替换时写回的已修改数据和不需要写回的未修改数据，在每个标记中加入了一个称为修改位（modified bit）的附加位。CPU 写入数据的每一行的标记中，都设置了修改位。只要在读缺失（read-miss）操作期间载入了数据，修改位就会被清除，因为数据仍然和主存储器保持同步。通过跟踪每一行的修改状态，高速缓存只需要在必要时写回高速缓存行，而无需像写直通高速缓存机制那样每次保存操作都要这样做。所以说，写回高速缓存机制的优点就是主存储器写入操作可能更少，总线操作可能更少，整体性能也就可能更好。缺点是操作系统需要好多次地把被修改的行写回存储器，以此来保持数据的完整性（在后面的章节中解释）。写回高速缓存机制实现起来也要比写直通高速缓存机制的成本更高。一般而言，写回机制的优点大于缺点，所以这项技术得以广泛使用。例如，Intel 486、Pentium 的片上高速缓存和 i860 XR、MIPS R4000、Motorola 88200 和 68040 以及 TI MicroSPARC 和 SuperSPARC（如果没有使用外部高速缓存的话）都使用写回高速缓存机制。

前面的讨论只考虑了保存来自 CPU 的数据期间在高速缓存里命中时情况。如果相反没有命中，那么采取的措施则取决于高速缓存是否支持写分配（write-allocate）。在使用写分配机制的时候，CPU 保存的数据在发生高速缓存缺失的情况下始终会被写入高速缓存（也就是说，为数据分配高速缓存行），以便利用时间局部性和空间局部性的优点。要做到这一点，要执行和读缺失期间相同类型的处理。首先，替换策略选择一行要丢弃的数据，给保存新数据腾出空间。如果使用写回高速缓存机制，而且所选的要被替代的高速缓存行又被修改过，那么就必须把这一行写入到主存储器中。接着，从主存储器中读取与造成缺失的所有地址相关联的全部高速缓存行。之所以必须读取全部高速缓存行（或者子行），是因为正如 2.2.1 小节所阐述的那样，只有一个有效位涵盖高速缓存行或者子行的状态。一旦读取了一行，那么 CPU 写入的数据就被插入到这一行中，并且设置这一行的高速缓存标记，以便反映出新数据的地址。如果使用了写回高速缓存机制，那么还要设置这一行的修改位。如果 CPU 写入数据的大小等于高速缓存行的大小（例如，把一个字保存在每行一个字的高速缓存中），那么就会跳过主存储器的读取操作，因为该行肯定要被 CPU 的数据所替换。MIPS R2000/R3000 就是这种情况，它使用了 4 字节长的高速缓存行。使用写分配的其他处理器有 MIPS R4000、Motorola 68040 和 88200、TI SuperSPARC 以及 Intel 80486（82495DX）的外部高速缓存。

有些高速缓存为了在硬件实现上更简单而放弃了写分配策略的局部好处。在不支持写分配策略的高速缓存中出现保存缺失的时候，数据被独自写入主存储器，而保持高速缓存的内容不变。Intel 80486 和 TI MicroSPARC 就使用了这种技术。

在大多数情况下，写回高速缓存都使用写分配策略，但是写直通高速缓存则不然。这是因为硬件的成本问题：写分配会增加低成本的写直通方法的成本，因为在一次保存缺失期间必须读取一行再把这一行写入主存储器。写回高速缓存机制能够很好地配合写分配工作；另一方面，如果被写入的行尚未被 CPU 读取，那么就不会被高速缓存起来，从而导致所有这样的存储都被送往主存储器。但是，这也有例外。例如，Intel Pentium 的外部高速缓存控制器

（82434LX）能够配置成使用不带写分配的写回策略，MIPS R2000/R3000 使用带有写分配的写直通策略。在 MIPS 芯片的案例中，硬件执行写分配很容易，因为每一个高速缓存行的大小只有 4 字节，从而不需要在发生保存缺失的情况下从存储器中读取一行。

其他的变化也是有可能的。例如，Motorola 88200 上的高速缓存就使用了写回高速缓存机制，但是只要在高速缓存内发生了保存缺失时，就会更新存储器。这称为写一次（write-once），它允许高速缓存行即使刚发生对它的保存也能保持未修改状态，因为主存储器会被更新。幸运的是，对于操作系统来说，诸如这样的变化都是透明的。

现在由下面的各小节探讨几种常见的高速缓存组织，它们将会有助于读者加深对刚介绍过的概念的理解。

2.3　直接映射高速缓存

最简单的高速缓存组织就是直接映射（direct mapped）或者单路组相联（one-way set associative）高速缓存（短语“单路组相联”的意思在下一节介绍“双路组相联”的时候就清楚了）。TI MicroSPARC 的片上数据高速缓存使用的就是这种高速缓存组织。它包含有 2Kb 的数据高速缓存，组织成 128 个高速缓存行，每行 16 字节。在如图 2-5 所示的这种类型的高速缓存中，在保存数据的部分高速缓存里，以散列算法用地址计算出每行拥有且仅有一个索引（也就是说，直接映射关系）。高速缓存可以当作是高速缓存行的线性数组，这些行都以散列算法计算得出的结果作为索引。在搜索期间，索引行中的标记要和地址进行比较以发现是否命中。因此，单有索引还是不够的，因为采用任何类型的散列算法都会有许多不同地址计算出相同的索引结果。在出现一次命中的时候，就从高速缓存行中提取数据，送往 CPU。如果标记和地址不匹配，或者有效位没有打开（要记住有效位是和地址一起保存在标记中的控制信息），那么就出现一次缺失，因为在直接映射的高速缓存中，这是能够保存数据的唯一位置，因此没有必要进一步搜索高速缓存。

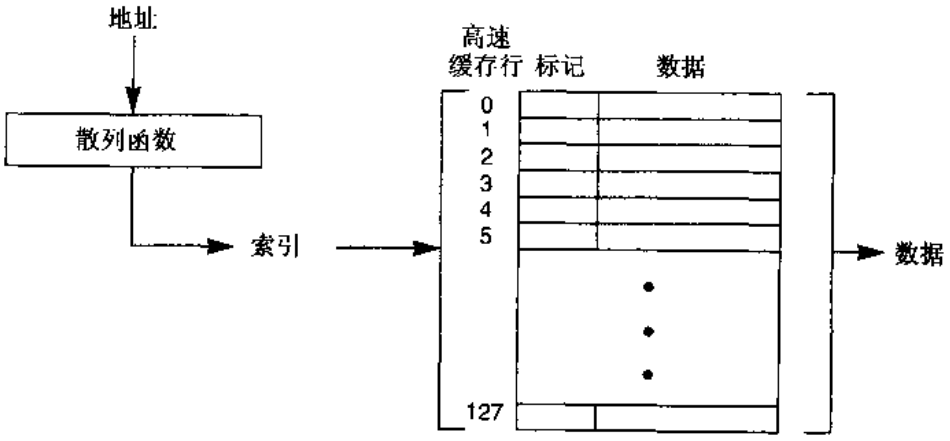


图 2-5　TI MicroSPARC 直接映射的数据高速缓存

使用直接映射高速缓存机制的其他处理器有 Intel Pentium，它用于其外部高速缓存，以及 MIPS R2000/R3000/R4000 所支持的所有高速缓存。

2.3.1 直接映射高速缓存的散列算法

直接映射高速缓存的高速缓存散列算法必须将一个来自 CPU 的给定地址转换为高速缓存中一行的索引值。因为散列算法的计算是在硬件中完成的，所以它必须既简单，速度又快。对于降低成本和获得快速实现来说，简单性是必不可少的。速度非常重要，因为需要索引在适当的高速缓存行上启动读取周期。在比较高速缓存标记，查明是否有一次命中之前，必须完成上面所有的步骤。由于有这些限制因素，所以在高速缓存中使用的散列算法很少会采用算术运算，因为这些操作会花费很长的时间。

最常用的高速缓存散列算法是取模（modulo）函数，这种函数利用了这样的事实，即在直接映射高速缓存中的行数往往是 2 的幂。这就使得散列算法只要从地址中选出若干比特位，选出的比特位数正好等于高速缓存行数以 2 为底的对数，就能直接用它们来作为索引。例如，考虑 TI MicroSPARC 上的数据高速缓存，它有 128（2^7）行，每行 16 字节。这个配置的散列算法应当从地址中选出“位<10..4>”，使用这 7 个比特位从高速缓存的 128 行中选出一行（参见图 2-6）。因为每一行高速缓存有 16 字节，所以该地址的低 4 位将选择 CPU 寻址的高速缓存行中的若干字节。因为高速缓存总共有 2K，所以我们能够看到这种选择索引的方法会让所有模 2K 的地址都索引到高速缓存中相同的行（所有和“位<10..4>”匹配的地址都生成相同的索引）。这叫做取模散列算法（modulo hashing algorithm）。

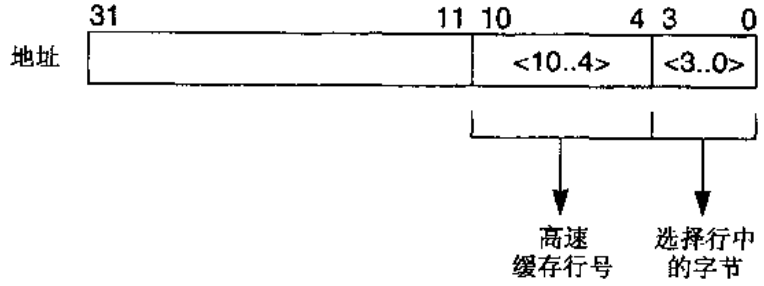


图 2-6 TI MicroSPARC 的高速缓存散列算法

此外，产生相同索引的地址被认为有相同的颜色。如果一个高速缓存的大小是页面大小的倍数，则该高速缓存就被认为其颜色和高速缓存中的页数一样多。如果一组存储页面的地址散列到高速缓存内同一组的高速缓存行上，那么称这组存储页面有相同的颜色。高速缓存颜色是一种用来区分页面如何索引高速缓存的概念。它的用途将在 7.2.3 小节中进一步讨论（参见图 7-8，该图给出了相对于存储器中页面的高速缓存颜色）。

图 2-7 描述了程序的地址空间是怎样映射到高速缓存的存储器上的。因为在“位<10..4>”中拥有相同位模式的所有地址都会索引到高速缓存中相同的行，所以地址 0、2048（2K）、4096（4K）、6144（6K）等都将索引或者映射到高速缓存中的 0 行，而地址 16、2064（2K +

16）、4112（4K + 16）、6160（6K + 16）等都映射到 1 行上。正如 2.2.1 小节所阐述的那样，标记将会解决歧义的问题，因为它们指出了被高速缓存的数据的地址。

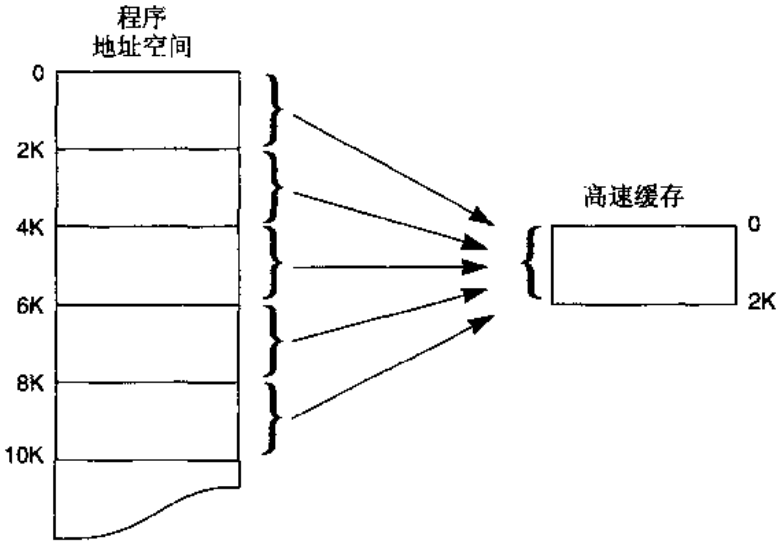


图 2-7　使用图 2-6 中的散列算法将地址空间映射到高速缓存

虽然取模散列算法是最常用的方法，但是用来选择高速缓存行的比特位可以取自地址中的任何部分。要看到取模散列方法的优点，下面的例子将使用如图 2-8 所示的另一种散列算法。这两种散列算法之间的区别在于程序中的地址是如何索引高速缓存的。在第一个例子（图 2-6）中，连续的程序地址映射到高速缓存中的连续位置，而图 2-8 中的算法则导致整个地址范围（那些在“位<11..4>”中有相同值的地址）映射到相同的高速缓存行上。所以从 0x0 到 0xfff 的所有地址都索引到高速缓存中的 0 行，从 0x1000 到 0x1fff 的所有地址都索引到 1 行，依此类推。效果如图 2-9 所示。

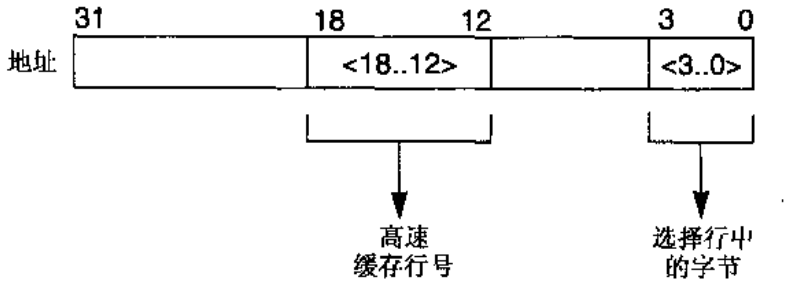


图 2-8　另一种高速缓存散列算法的例子

虽然实现这样的映射关系是可能的，但是我们不使用它，因为它导致了高速缓存利用率的低下（这里只给出举例说明）。例如，完全处于从 0x0 到 0x1fff 地址范围内的一个小程序让其所有的引用都映射到了高速缓存的头两行上。当这个程序正在执行的时候，只使用了 2K 高速缓存中的 32 字节。在这样一种情况下，可能会有很低的命中率，而从高速缓存感觉不出来性能增益。

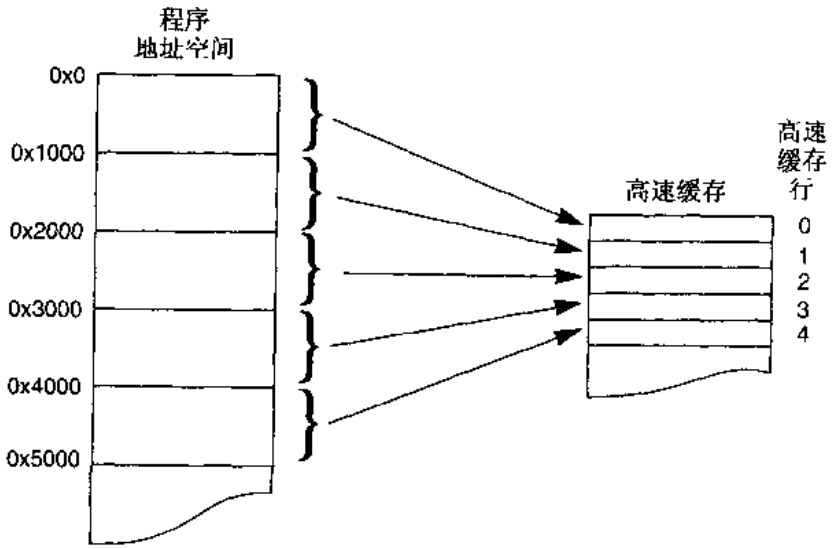


图 2-9 用图 2-8 中的散列算法将地址空间映射到高速缓存

通过对比，读者可以看到图 2-6 所示的散列算法提供了一种地址到高速缓存的良好映射关系，能够把空间局部性扩展到最大。出于这个原因，它就是所有使用散列算法的高速缓存存储系统而选择的算法。

2.3.2 直接映射高速缓存的实例

现在我们将迄今为止所介绍的所有概念联系起来，展示出一个查找（look-up）操作如何发生的完整例子。对于这个例子来说，假定我们有一个字长为 4 字节、有 12 位地址的系统。系统包含的直接映射高速缓存每行 8 字节，共计 8 行（为了简化示例，选择了 12 位的地址和一小块高速缓存。注意，始终都使用八位组的记法）。因为一行中有 8 字节，意味着以“位<2..0>”选择高速缓存行内的字节，而以单独一个比特位 2 就能选择行内的字，因为每行有 8 字节，即 2 个字。根据前面的讨论，最好的散列算法是使用“位<5..3>”作为高速缓存行的索引。索引一个 8 行的高速缓存需要 3 位，直接选择上述的几位选择高速缓存行内的字节，这样做能基于局部引用来充分利用高速缓存。地址中剩下的比特位则用于和高速缓存的标记进行比较（参见图 2-10）。

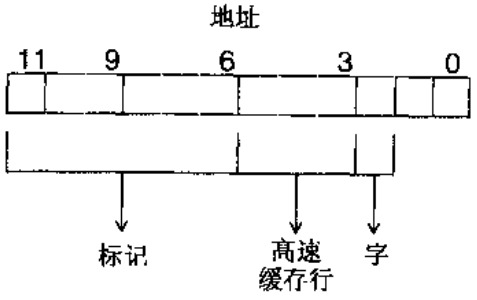


图 2-10 地址比特位用途的例子

对于这个例子来说，我们假定高速缓存的初始内容如图 2-11 所示。在这个配置中，标记部分的符号“---”表示这一行无效。在高速缓存行中数据部分的两个字被一条竖线分隔开了。在高速缓存行中的字采用高字节结尾的顺序，这意味着地址最小的字位于高速缓存行的左端。

高速缓存行	标记	数据	
0	---		
1	---		
2	---		
3	012	052	0777
4	---		
5	---		
6	---		
7	035	067	0

图 2-11　高速缓存的初始内容

举第一个例子，CPU 将发出一次读取地址 01234 的操作。散列算法选择地址中的“位<5..3>”，它们给该地址一个行索引值 3（参见图 2-12）。

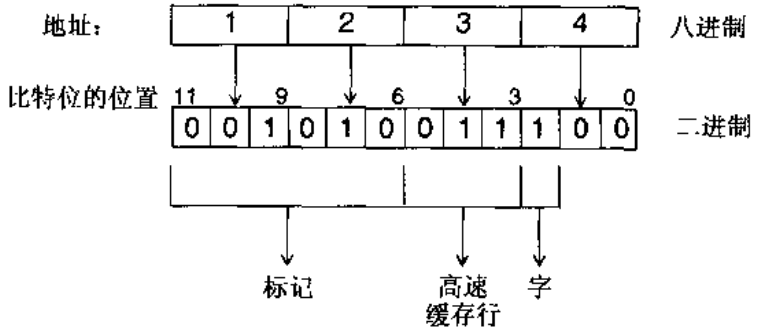


图 2-12　分解地址 01234 供高速缓存使用

因为这是一个直接映射高速缓存，所以这个索引只和一行有关，如果这个地址的数据都在高速缓存中，那么数据就保存在这一行里。高速缓存控制硬件获得索引值，并且读取高速缓存行 3 的内容。它发现有一个有效标记，并且把这个标记同地址中的标记部分（位<11..6>，即 012）相比较。地址中的标记部分和被索引的高速缓存行中的标记相吻合，因此就出现一次命中。现在，高速缓存控制器必须从高速缓存行中的数据部分选择正确的字。因为地址中设置了比特位 2，所以 CPU 想要让高端的字保存在高速缓存行的右边；因而想要的字为 0777。

这里要理解的一件重要的事情是，对于高速缓存的标记来说没有必要包含完整的地址，因为根据地址在高速缓存中的位置就能推算出地址的一部分。这一点之所以重要是因为它减少了高速缓存行中的总比特位数，从而也就降低了高速缓存的成本。一般而言，标记只包括地址中散列算法不使用的比特位。在这个例子中，“位<5..3>”就只选择出了可以保存数据的高速缓存行。于是，用来构成高速缓存行索引的比特位可以从标记中省略。类似地，因为

该高速缓存行包含 8 字节，所以地址中的低 3 位（<2..0>）也不需要保存在标记中。因此，在标记中只需要保存“位<11..6>”。

举第二个例子，考虑当 CPU 试图从地址 0130 读取数据时会发生什么情况。散列算法选择“位<5..3>”用于索引，如图 2-13 所示，这几位又等于 3。

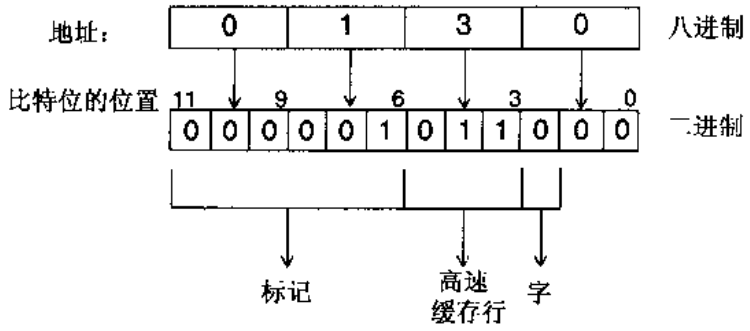


图 2-13 分解地址 0130 供高速缓存使用

读取高速缓存行 3，并且将来自地址的标记 01 同高速缓存中的标记 012 进行比较。它们并不匹配，所以就发生了一次缺失。现在必须把地址发送到主存储系统来检索数据。接着，主存储器内从地址 0130 开始的两个字载入到有索引的高速缓存行中。因为位 2 是 0，所以需要的是双字中低端的那个字，于是高速缓存行左半部分中的字被返回给 CPU。

举最后一个例子，读取地址 06540 会造成一次缺失，因为被索引的行（高速缓存行 4）没有包含有效的标记，这会自动导致一次缺失。

图 2-14 总结了前面的几个例子，还给出了其他几个例子。

地址	被索引的行	返回的数据
01234	3	0777
01230	3	052
00130	3	缺失
03574	7	0
06540	4	缺失

图 2-14 使用图 2-11 查找高速缓存的例子

2.3.3 直接映射高速缓存的缺失处理和替换策略

当出现一次高速缓存缺失的时候，必须从主存储器读取数据并且返回给 CPU。数据还要载入高速缓存，以便在近期内再次引用它，就能立即得到。如果高速缓存行比一个字大，就要从主存储器中多读几个字，以便在高速缓存中载入完整的一行。此时就会造成邻接字被预取（prefetched），从而充分利用了局部引用特性。要读取完整的一行，高速缓存可能需要额

外的存储器周期，也可能不需要，这取决于主存储系统的设计。为了获得最佳性能，主存储系统应该以高速缓存行的大小为单元传送数据，这称为突发模式（burst mode）的传送。这种模式能让完整的高速缓存一行在一次存储器操作中传送。没有突发模式，传送必须以更小的单元来执行，从而增加了开销。

一旦主存储器提供了所需的数据，就必须在高速缓存中找到一个位置保存它。这个位置必须经过挑选，以便散列算法在以后的查找操作期间能正确地定位高速缓存的行。在直接映射高速缓存中，这个位置是通过采用散列算法计算行中首字的地址，从而得到该行将会保存在高速缓存中什么地方的行索引值。最初检查的和发生缺失时找到的结果始终都是同一行。新行必须保存在这个位置，因为把它保存在高速缓存中的其他行，以后引用时就不能通过索引找到它。这是直接映射高速缓存唯一可能采用的替换策略。

如果缺失操作期间被替换的高速缓存行以前并不是有效行，那么就可以将新行的数据载入到这一行中，并且设定标记，指出数据的地址。该行的有效位也设为置位（打开）。如果这一行以前保存着不同地址的有效数据，那么这些数据就会被丢弃，并用新行替换它们。如果采用写回高速缓存机制，而且修改了原来的数据（参见 2.2.5 小节），那么在替换它之前必须将它写回存储器，从而不会丢失修改过的数据。

举一个例子，再次考虑 2.3.2 小节里的例子和图 2-11 描绘的高速缓存。如果 CPU 试图读取位于 00130 的字，那么就会发生一次缺失，因为高速缓存内的行 3 缓存的是从地址 01230 到地址 01237 的数据。如果主存储器在地址 00130 处保存的值为 0222，在地址 00134 处保存的值为 0333，那么在处理缺失之后，高速缓存将被更新为图 2-15 所示的内容。

高速缓存行	标记	数据	
0	---		
1	---		
2	---		
3	001	0222	0333
4	---		
5	---		
6	---		
7	035	067	0

图 2-15　在缺失处理之后高速缓存的内容

新标记和新数据替换了原来的标记和数据，高速缓存内其余的行则保持不变。CPU 最初请求的数据 0222 在新行写入高速缓存的同时返回给 CPU。

2.3.4　直接映射高速缓存的总结

在直接映射高速缓存中，主存储器内数据的地址和高速缓存内可能保存该地址的位置或者行之间有一一对应的关系。数据在高速缓存内是通过散列该地址来定位的，计算得出了高

速缓存中可以保存该数据的唯一一行的索引。直接映射高速缓存既可以使用写直通策略，也可以使用写回策略。

直接映射高速缓存是实现起来最简单的高速缓存，因为在读或者写操作期间只可能有一个搜索命中的位置。这就简化了控制逻辑，从而让实现的成本更低。遗憾的是，这也是直接映射高速缓存的缺点，因为许多不同的地址都被散列到了同一行上。带有病态引用模式的程序，该模式中局部引用的数据或者指令地址都产生了相同的索引或者一小组索引，那么程序就无法从高速缓存获益，因为命中率非常低。在这样的情况下，高速缓存行在被再次命中之前总是被替换掉，所以这种情形称为高速缓存颠簸（cache thrashing）（这个名字来源于虚拟存储调页系统中出现的相同现象）。下面一节介绍对直接映射高速缓存所做的一种改进，以此减少颠簸，提高命中率。

2.4 双路组相联高速缓存

双路组相联高速缓存（two-way set associative cache）类似于直接映射高速缓存，不同之处在于散列函数算出的索引指向高速缓存中可能保存数据的一组两行高速缓存。在这一组内，每一行高速缓存都有它自己的标记，这意味着高速缓存可以同时保存经散列算法算出相同索引的两个不同地址的数据。Intel Pentium 的片上数据高速缓存就是双路组相联高速缓存。它总共保存有 8K 的数据，每行 32 字节。这意味着高速缓存中总共有 256 行（8192 字节÷32 字节/行），组成 128 组（256 行÷2 行/组）。图 2-16 描绘出了这样的一个高速缓存。

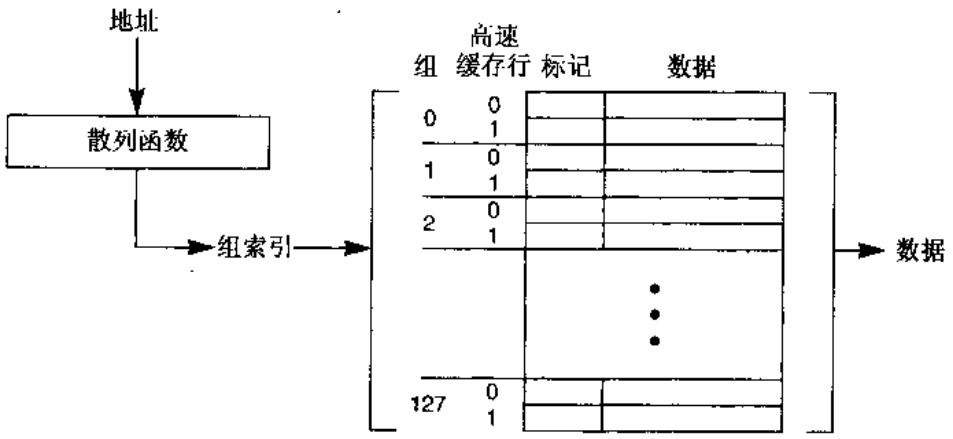


图 2-16 Intel Pentium 的双路组相联数据高速缓存

在查找操作期间，散列函数算出的索引指向一组两行可以保存数据的高速缓存。被索引的一组两行高速缓存中的标记和地址同时进行比较，以查看是否命中了两行中的某一行（组内所有行的标记并行比较，从而不会因采用串行比较而降低高速缓存的访问速度）。双路组相联高速缓存的目的是，减少直接映射高速缓存中两个不同地址经散列计算得出相同的索引值时发生的高速缓存颠簸。在双路组相联高速缓存中，这两个不同的地址都保存在高速缓存中。使用这种类型高速缓存的其他处理器还有 Intel i860 XR 以及 80486 的外部高速缓存。

现在就很清楚为什么直接映射高速缓存也称为单路组相联高速缓存了。“单路”和“双路”的说法是指每一组中高速缓存行的数量（在高速缓存内所有的组都有相同数量的行）。“相联”一词则是指实际上这一组高速缓存就是以内容编址（content-addressable）的或者说相联（associative）的一个存储器，因为它是通过对照组内高速缓存行的位置（或者地址）来检查标记的内容从而判断出一次命中的。直接映射高速缓存是 n 路组相联高速缓存的一种退化形式，因为每一个相联组中只有一行高速缓存。

配合双路组相联高速缓存的散列算法和配合直接映射高速缓存的散列算法相同，区别在于前者所需的比特位数更少，因为对于总量相同的高速缓存存储器来说，双路组相联高速缓存中的组数只有直接映射高速缓存的一半。所以用于图 2-16 中高速缓存的散列算法就只使用“位<11..5>”来选择组。和以前一样，“位<4..0>”选择高速缓存行内的字节（因为一行有 32 字节）。

替换策略稍微复杂一些。采用直接映射高速缓存时，在一次缺失操作期间，载入的高速缓存行必须放入将被索引的位置，从而可以在未来的查找操作期间找到。这一行就在散列算法所索引的高速缓存行组内。但是采用双路组相联高速缓存时，现在组内有两行都可以选择用来替换。两行中的任何一行都可以被替换，因为组内的两行在查找操作期间都可以搜索到。在理想情况下，最好替换在最长时间内不会被再次引用的行，因为这能提高高速缓存的整体命中率。遗憾的是，没有办法知道程序未来的引用模式。时间局部性表明，在组内宜采取 LRU 替换的做法，所以大多数实现（如 Intel 80486 外部高速缓存）都利用了这种方法。这种做法不但易于实现，而且效果相当不错。给每个组加上一个额外的比特位（称为 MRU，代表“最近使用”）就可以实现这种方法。每次在一组内出现一次命中时，MRU 位就被更新，以反映该组内的哪一行产生了命中。当组内的一行必须被替换的时候，高速缓存首先检查其中是否有一行被标记为无效。如果有，那么就替换那一行。如果两行都是有效的，那么 MRU 位就指出上次使用的是那一行，于是就选择替换另一行。然后再更新 MRU 位来指出被替换的行。

2.4.1　双路组相联高速缓存的总结

双路组相联高速缓存通过索引一组两行可能保存数据的高速缓存行，来尝试获得比直接映射高速缓存更好的高速缓存性能。双路组相联高速缓存实现起来要稍微复杂和昂贵一些，因为必须并行比较一组内两行的标记，而且需要一种更复杂的替换策略。

它相对于直接映射高速缓存的优势在于可以减少高速缓存颠簸的现象。如果在一个进程的局部引用中多个地址得出了同一个索引，那么这两个地址会同时被高速缓存，而直接映射高速缓存却一定要替换该行。注意，双路组相联高速缓存的性能绝对不会低于行数相同的直接映射高速缓存。在最差的情况下，如果程序产生地址的顺序是每个地址索引唯一一行，那么双路组相联高速缓存的性能就和直接映射高速缓存一样。另一方面，如果局部引用是由产生冗余索引的多个地址所构成的，那么双路组相联高速缓存的命中率会更高，因为它能同时高速缓存着产生相同索引的两个不同地址的数据。

2.5 n路组相联高速缓存

组内的行数并没有理论上的限制；在当今的计算机中，4路乃至更多路的组相联高速缓存也并非鲜见。例如，Motorola 68040和88200、Intel 80486和i860 XP以及TI SuperSPARC的数据高速缓存都是4路组相联高速缓存。因为在组内并没有索引机制，所以也不要求组的大小一定是2的幂。在这一点上的案例就是TI SuperSPARC的指令高速缓存，它是5路组相联高速缓存。

用于这些高速缓存的散列算法和双路组相联高速缓存所采用的散列算法是相同系列的：使用取模散列函数（modulo hashing function），该函数采用的比特位数等于组数以2为底的对数值。

每组两行以上的高速缓存往往并不使用严格的LRU替换，因为这样做需要的状态信息太多。常常代之以使用历史信息有限的伪LRU算法。除了68040之外，所有提到过的处理器都采用了这种技术。68040的设计人员选择省略了伪LRU算法所需的额外状态信息，而是采用了一种伪随机替换（pseudorandom replacement）策略。

2.6 全相联高速缓存

在高速缓存内，组的大小可以增加到使组内的行数等于高速缓存内的全部行数。此时的高速缓存就称为全相联高速缓存（fully associative cache）。在全相联高速缓存中只有一组，它包含高速缓存中所有的行，不需要散列计算或者索引机制，因为只有一组要检查。采用任何n路组相联高速缓存时，都是并行搜索组内所有的高速缓存行。顾名思义，全相联高速缓存在每次查找的时候都要在全部高速缓存内进行搜索。

之所以需要全相联高速缓存，是因为它能够把高速缓存颠簸（thrashing）的现象减到最少，原因是在高速缓存中的任何位置都可以保存数据的任何部分。于是，从理论上来说，如果一个程序具有局部引用性的地方小于或者等于高速缓存的大小，那么它就会获得100%的命中率，并从高速缓存得到最大可能的性能提升。

全相联高速缓存构建起来要比同等大小、但每组行数较少的高速缓存成本高，因为必须并行搜索高速缓存中的所有行。这就是全相联高速缓存很少用于指令和数据的主要原因。在使用它们的时候，通常是小规模、有特殊用途且有高度时间局部性的高速缓存，比如转换后备缓冲器（translation lookaside buffer，TLB）。TLB高速缓存最近在MMU内使用过虚拟地址到物理地址的转换。小规模、全相联的TLB比较实用，因为大多数程序都体现出了局部引用特性，这意味着工作集的转换会被多次使用。此外，因为每一次转换都映射了完整的一页数据，所以只需几个转换就能提供良好的性能。例如，TI MicroSPARC使用一个有32项的全相联TLB，而SuperSPARC有64项。Motorola 88200和所有MIPS处理器上的TLB也都是全相联高速缓存。

2.7　n 路组相联高速缓存的总结

正如现在所看到的那样，从直接映射到全相联的所有高速缓存组织结构都遵循着相同的组成原则：每一种组织结构都有一种用于选择搜索行的算法，每一种组织结构都有一种替换算法，每一种组织结构都可以使用写直通或者写回策略，而主要的区别则在于每一组内行数的不同。在各种组成结构的一端是直接映射高速缓存，它每组只有一行。对于这种类型的高速缓存来说，以散列算法得到相同索引的所有地址必须在高速缓存中竞争一个可以保存它们的位置。直接映射高速缓存的替换策略相当简单，因为唯一的候选替换行就是散列算法索引的那一行。在各种组成结构的另一端是全相联高速缓存，它只有包括高速缓存内所有行的一个组。这种类型的高速缓存不需要散列计算，因为在每次查找操作期间都必须检查所有的行。在组比较大的高速缓存中使用 LRU 替换并不实用，这让随机替换成为常见的方法。

随着组的大小从单路组相联或直接映射高速缓存到全相联高速缓存逐渐增大，目标是减少多个地址散列到相同组时出现的高速缓存颠簸现象。增加组的大小允许让其地址产生相同索引的更多数据同时保存到高速缓存中。于是，增加组的大小有可能提高命中率和系统性能。组变大的缺点是增加了硬件成本和复杂性，因为必须并行比较被索引组内所有行的标记。实际情况是，除了最小的高速缓存之外，对所有的高速缓存来说，都要避免使用组太大的高速缓存。

2.8　高速缓存冲洗

译者注：

flush / refresh 在很多中文译文中都译为“刷新”，但这两个词在使用对象、描述的技术概念上都不相同，为了予以区分，本书参考一些国内知名学者的意见，将 flush 一词译为“冲洗”。读者可以从下面两段分别援引自操作系统和计算机组成方面两部名作的内容来体会出其中的差别。

flush / purge（冲洗 / 清除）：

The cache usually exports two operations to the kernel—flush and purge—both of which delete or invalidate an entry from the cache. The flush operation also write back any changes to main memory; the purge operation does not. -------<<Unix internals:the new frontiers >>-- Uresh Vahalia pp 506.

refresh（刷新）：

Dynamic RAMs（DRAM），in contrast，do not use flip-flops. Instead，a dynamic RAM is an array of cell，each cell containing one transistor and a tiny capacitor. The capacitors can be charged or discharged，allowing 0s and 1s to be stored. Because the electric charge tends to leak out，each bit in a dynamic RAM must be refreshed（reloaded）every few milliseconds to prevent the data from leaking away.<<structured computer organization>>--Andrew S. Tanenbaum pp. 152

所有的高速缓存实现都向操作系统提供了从高速缓存中删除数据的能力，这种功能一般称为高速缓存冲洗（cache flushing）。根据高速缓存组织结构的不同，这对于防止意外引用过时数据、保持高速缓存一致性来说可能是必不可少的功能（高速缓存一致性在多处理机系统上有更广泛的含义，这将在第三部分讨论）。一个高速缓存需要冲洗的准确时机则取决于它的组织结构，这是接下来的几章要讨论的主题。高速缓存冲洗有两种形式：使主存储器有效（validating main memory）和使高速缓存无效（invalidating cache）。

使主存储器有效的形式是指在采用写回策略的高速缓存中将修改过的数据写回到主存储器内的做法。正如 2.2.5 小节所讨论的那样，这是由高速缓存在替换一个修改过的行期间自动完成的，但是它也可以由操作系统明确地进行控制。一次显式的使有效操作（validation operation）将把行内的数据写入主存储器，然后将高速缓存内数据的修改位关闭（因为相对于主存储系统内的副本而言不再是修改过的）。使主存储器有效的方法还可以让操作系统随时更新。使用这种方法可以防止主存储器内数据的过时副本被系统中不使用高速缓存的其他部分所引用。如果操作系统指定将一个高速缓存行写回主存储器，而这一行要么不在高速缓存中，要么当前并没有被修改过，那么该有效操作对高速缓存什么也不做。这种冲洗类型从不需要采用写直通策略的高速缓存来完成，因为存储器始终和高速缓存中的内容保持同步。

使高速缓存无效的形式是指，如果高速缓存中的数据被修改过，那么无需先将它写回主存储器就把它丢弃的做法。这种功能既可以用于写直通高速缓存，也可以用于写回高速缓存。操作系统使用它来丢弃高速缓存中的过时数据。如果数据在主存储器内的副本更新与高速缓存无关，从而导致被高速缓存的数据副本过时，就会发生这种操作。让该数据在高速缓存中无效会造成下一次引用该数据时出现一次缺失，因此要从主存储器中重新读取该数据的正确副本。

一般而言，采用写回策略的高速缓存允许将独立的使主存储器有效和使高速缓存无效操作组合成单个操作，因为它们是一种常用的序列。在这种情况下，如果高速缓存中出现了该行，而且被修改过，那么就首先完成主存储器有效操作，于是该数据就不会丢失。在此之后往往会跟着进行一次使高速缓存无效操作。

两种高速缓存冲洗形式中的任何一种通常都能够以单行为基础来使用。大多数实现允许操作系统指定要被冲洗的数据的地址。随后在高速缓存中查找这个地址，如果命中则被冲洗掉。如果没有命中，则保持高速缓存不变。MIPS 处理器就是以这种方式工作的。有些实现可以允许一次冲洗一组地址，例如一页甚至整个高速缓存。TI MicroSPARC 和 SuperSPARC 就允许后一种方式，它们称之为“洗净”功能。Motorola 68040 和 88200 支持一次按行、按页面或者整个高速缓存的冲洗。

2.9 无高速缓存操作

大多数实现都允许 CPU 绕过高速缓存直接访问主存储器，这称为无高速缓存（uncached）操作。例如，如果执行一次无高速缓存读取操作，那么即使地址已经在高速缓存中造成一次命中，也还是从主存储器读取数据。在这种情况下，忽略被高速缓存的数据，返回来自主存

储器的值。在本章中提到的所有处理器都支持无高速缓存访问，这项功能往往可以通过页表项（page table entry）中的一个标志位来逐页进行选择。

在访问主存储器中其值的改变与 CPU 写操作无关的单元（用 C 程序设计语言来说就是不稳定的单元）时，采用无高速缓存操作就很有用。例如，映射 I/O 设备上状态寄存器的内存应当包含根据设备的状态而变化的值。一般采用无高速缓存访问来访问这类寄存器，因为一旦设备的状态发生变化，状态寄存器中被高速缓存的任何值都是过时的。

无高速缓存访问通常用于任何内存引用。在频繁要求进行高速缓存冲洗来保持高速缓存一致性的情形中，它们也很有用，这种情形可能会降低系统性能。

2.10　独立的指令高速缓存和数据高速缓存

将指令高速缓存和数据高速缓存分开的做法目前在计算机系统中相当常见。这种做法能够有效地使高速缓存的带宽加倍，因为它能让 CPU 从指令高速缓存预取指令的同时把数据载入或者保存到数据高速缓存中。图 2-17 描绘了这样的一种组织结构。本章提到的所有处理器，除了 Intel 80486 之外，其片上高速缓存都有独立的指令高速缓存和数据高速缓存。

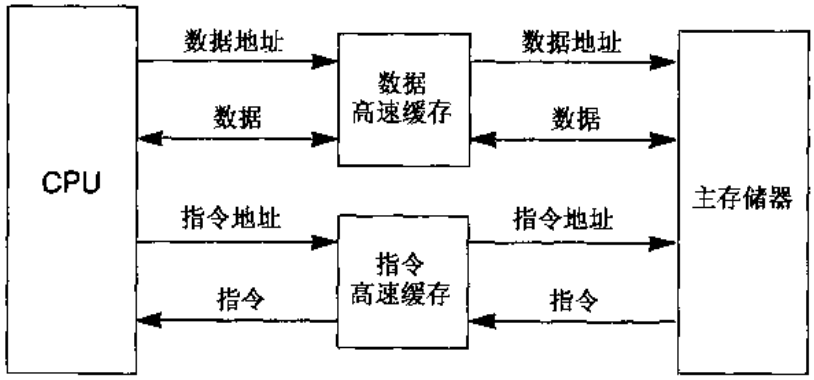


图 2-17　独立的指令和数据高速缓存

因为两块高速缓存都可以同时访问主存储器（缺失处理或者写操作），所以要由硬件来仲裁高速缓存对主存储器的访问，这对于软件来说是透明的。注意，没有办法直接把数据保存到指令高速缓存中。因此，指令高速缓存是只读高速缓存。

这种组织结构最重要的方面就是缺乏数据高速缓存和指令高速缓存之间的直接互连。如果在指令高速缓存中没有命中某个指令，那么指令高速缓存就无法到数据高速缓存中查找它。指令高速缓存始终都是从主存储器读取指令来完成缺失操作。类似地，数据高速缓存中的缺失也要从主存储器读取。把数据保存到数据高速缓存中不会影响指令高速缓存的内容。虽然这是一种最简单的实现，但是它可能会产生高速缓存的不一致性，因为主存储器的内容可能会高速缓存在一个以上的地方。如果使用单一的、将指令和数据结合在一起的高速缓存，就不会出现这类不一致性。

考虑使用自身能够修改代码的程序的情形。这类程序包括诸如LISP解释器这样的程序，因为对于它们来说，部分编译它们正在解释的程序并非鲜见（编译代码通常写入进程的数据区）。如果要执行的指令是在数据区动态生成的，那么有可能出现两种不一致性。第一，如果使用写回高速缓存机制，那么最近写的指令可能尚未写入主存储器。这意味着，如果程序试图执行这些新指令，那么指令高速缓存可能会从内存中取得过时的指令。第二，一旦动态生成的指令高速缓存在指令高速缓存中，那么程序的任何写操作（以此用新指令来替换那些在指令高速缓存中的老指令）都不会对指令高速缓存造成影响。新指令将写入数据高速缓存，并且最终写入主存储器，但是指令高速缓存不知道取得新值。它会继续执行过时的老指令，直到行替换删除这些指令为止。在这种情况下，下一次引用那些指令将会造成一次缺失，并且从主存储器读取新指令。

遗憾的是，操作系统不能把高速缓存藏起来，让这类程序看不到高速缓存的存在，因为操作系统没有办法知道程序什么时候试图执行其数据空间部分。唯一的解决方法是提供特殊的系统调用，以便程序已经产生一组它现在想要执行的指令时就通知操作系统。接着，如果在数据高速缓存中使用了写回高速缓存机制，那么操作系统就使主存储器有效，并且使指令高速缓存的内容无效（在提供特殊指令来冲洗高速缓存的体系结构上，如果应用程序能够直接执行高速缓存冲洗指令，那么就不一定要有特殊的系统调用）。冲洗指令和数据高速缓存通常作为硬件的单独操作来实现。

一般而言，只要操作系统需要使被高速缓存的数据无效来保持一致性，那么它也必须使指令高速缓存无效。各种特定的实例则取决于高速缓存的体系结构，下面的章节将讨论它们。

虽然有可能让构建的系统中的硬件自动保持指令高速缓存和数据高速缓存的同步，但是却很少这样做。这样的系统会要求每次在数据高速缓存上执行写操作的时候检查指令高速缓存，而且还有可能使指令高速缓存无效。这样做需要额外访问指令高速缓存，从而会干扰指令的取操作，并且降低取指令的速度。自身能修改代码的实例很少能值得为其在硬件上增加复杂性。

2.11 高速缓存的性能

虽然全面讨论高速缓存的性能超出了本书的范围，但还是可以做以下一些观察的。首先，高速缓存的性能不仅取决于高速缓存的设计，而且取决于应用的引用模式。因此，必须小心翼翼地尝试采用基准程序来判定高速缓存的性能和总结结果。虽然很容易编写一个获得100%命中率的基准程序，但是把它们运用到实际应用中的时候，这样的结果是毫无意义的。例如，下面的程序会获得100%的高速缓存命中率：

```
while (1)
        ;
```

循环执行一次之后，所有的指令引用都会在高速缓存中命中。相反，下面的代码片段给一个数组中的每个元素都乘以常数c，因而会得到相当低的高速缓存命中率（假定数组要比高速缓存大）。

```
for ( j=0; j < YMAX; j++)
        for ( i=0; i < XMAX; i++)
                matrix[i][j] *= c;
```

因为在 C 语言中，数组是按行来保存的，在使用高速缓存的时候，如果数组中一行的大小超过了高速缓存行的长度（假定一开始数组没有被高速缓存），那么每次执行最里面的语句就会出现一次缺失。这样的情形尤其糟糕，因为行很长的高速缓存会读取大量从来都不会用到的数据。互换两层 for 循环会因为空间局部性而提高性能。即使每个元素只读取一次，高速缓存每次都要读取整整一行的事实也意味着引用连续的元素可能会产生一次命中。例外的情况是行很小的高速缓存，像 MIPS R2000/R3000，它们的每个高速缓存行只有 4 字节。如果数组 matrix 的每个元素也是 4 字节，那么就没有空间局部性的好处了。

即使高速缓存的性能是依赖于应用的，在直觉上还是有下面的结论（虽然对于所有应用来说，它们并不一定都对，但是对于包括典型 UNIX 命令在内的许多应用来说，它们都是正确的）。首先，写回策略比写直通策略更可取，因为程序一般会因为时间局部性而多次修改变量。即使它们没有多次修改变量，写回高速缓存机制也往往不会增加任何性能开销，因为写直通一行或者在以后替换行的时候再写回都要花费一个存储器周期。为每一行维护一个修改位以及处理写回增加了复杂性，但这样做是值得的。在最差的情况下，没有时间或者空间局部性可言，有写分配能力的写回策略只会多读一次高速缓存行。完全没有局部性的情形是非常少见的，所以它们不会对性能造成明显的影响。

接下来，增加组的大小一般也会有帮助。对于小规模的高速缓存（1K 或者更小）来说这样的做法特别有用，因为即便多个地址产生了相同的索引，它也能利用更多的高速缓存。对于非常大的高速缓存（1M 或者更多）来说，增大组就没那么重要了，因为随着行数的增加，出现一段数据替换现有的高速缓存数据的可能性也逐渐减小。

由于空间局部性，增加高速缓存行的大小一般也会对高速缓存性能有帮助。高速缓存行太长的缺点是在缺失处理期间读取数据需要开销。在小规模高速缓存的组织结构中找不到很长的高速缓存行，因为它会意味着高速缓存中没有几个单行了。因此出现替换的频率就更高了。

高速缓存的性能也会受到操作系统的影响。不同的高速缓存组织结构需要不同环境下的高速缓存冲洗机制。有若干种技术可以用来减少必须发生的冲洗量。这很重要，因为频繁的冲洗很花时间，并且减少了有用的数据被高速缓存的时间长度。这些技术将在第 7 章里讨论。

2.12　如何区分不同的高速缓存结构

如今能在计算机系统上找到的高速缓存五花八门。最明显的区别则体现在如下几个领域：

- 高速缓存大小
- 行大小
- 组大小
- 写分配的使用
- 替换策略

- 按照虚拟或者物理地址查找
- 如何标记行（通过虚拟地址、物理地址还是其他信息）
- 写直通或者写回策略

前5项影响高速缓存的性能，而且从保持高速缓存一致性的角度来看，除了一项（组大小）之外，其他的项对操作系统都没有直接影响。必须偶尔考虑一下组的大小，这将在3.3.2、4.2.2和4.2.6小节中介绍。清单中的最后3项也影响性能，但是它们还影响操作系统。这几项决定了操作系统为了向系统上运行的程序完全隐藏高速缓存的存在而需要明确执行的高速缓存冲洗量。

下面的几章介绍了4种高速缓存组织结构，而且描述了在什么样的条件下操作系统必须明确执行冲洗。所研究的不同高速缓存组织结构在采用虚拟地址还是物理地址来查找和标记行方面会有些变化。在每种情况下还需要考虑写直通和写回高速缓存机制之间的差异。

2.13 习　题

2.1　除了I/O控制器上的设备寄存器之外，不被高速缓存的数据还能用在其他什么地方？

2.2　解释如果高速缓存中的有效位在系统加电复位期间没有清除将会发生什么情况（在高速缓存中使用的存储设备加电时往往具有随机的内容）。

2.3　为什么高速缓存中的每一行都需要它自己的标记？

2.4　为什么采用以地址的“位<9..2>”来索引一个512行直接映射高速缓存的散列算法是一种不好的选择？如果高速缓存是双路组相联高速缓存又会怎样？

2.5　在使用本章介绍的技术时高速缓存的行数不是2的幂，解释为什么这是可能的或者是不可能的。

2.6　考虑一个直接映射高速缓存，它以地址中的“位<17..8>”来索引1024行高速缓存。每一行包含16字节。如果程序一般仅仅占用从0x1000到0x4fff范围内的地址，那么这是一种好的散列算法吗？解释为什么是或者为什么不是。如果高速缓存是16路组相联高速缓存又会怎样？

2.7　考虑一个写回、双路组相联高速缓存，它有4096行，每行16字节。应该将哪几位用于散列算法？为什么？需要行的标记部分中的多少位来保持地址（假定使用32位地址和随机替换策略）？在标记中为地址使用最少的比特位数同保存全部32位地址相比，高速缓存所需的比特位数节省了百分之多少？

2.8　有一个7路组相联高速缓存，每行256字节，总共512组，应该使用什么样的散列算法？

2.9　考虑下面对直接映射、写直通高速缓存的组织结构的两种选择。两者都保存2048字节（2K）的数据（不包括标记和控制位）。一种组织结构是使用4字节的行，另一种使用32字节的行。每一种选择中，包括数据、标记和控制位在内，高速缓存总共需要多少比特位？讨论在两种方案之间进行选择时，应该注意的有关硬件成本和性能的折衷考虑。假定系统使用32位的地址。

2.10　考虑这样一种环境，其中经常运行文本处理程序。这些程序的特征是它们经常要把字符串从一个地方复制到另一个地方，并且在缓冲区内产生字符串。在这样的环境中最好是使用写回还是写直通高速缓存机制？应该使用写分配吗？阐述理由。

2.11　描述下面一段代码的高速缓存局部性。系统使用独立的指令和数据高速缓存，两者均为 8K 双路组相联高速缓存，每行 16 字节。数据高速缓存使用带有写分配功能的写回策略（假定 int 类型是 32 位的）。如果行的大小增加到 256 字节，会发生什么样的情况？

```
struct {
        int rec_id;
        char rec_name[16];
        int value;
        int flags;
} array[1000];
...
int i, sum;
sum = 0;
for ( i=0; i<1000; i++)
        sum += array[i].value;
```

2.12　一个系统采用双路组相联高速缓存，每行 8 字节，总共 16 行。高速缓存使用带有写分配功能的写回策略以及 LRU 替换策略。假定高速缓存内的所有行在初始时都是无效的。主存储器包含以下数据：

地址	数据
01230	33
01234	44
01270	7
01274	8
02270	67
02274	42
03270	43
03274	46
03650	100
03654	200
06730	120
06734	210
08670	10
08674	20
08600	64
08640	76
09830	333
09834	355

接着出现了下面的存储器引用（每次引用一个完整的字）：

从 01234 载入
把 5 保存到 03650
从 08670 载入
从 08674 载入
从 01274 载入
从 08670 载入
把 99 保存到 09834
把 12 保存到 02270
从 01230 载入
从 06730 载入
从 03654 载入
把 37 保存到 03654

绘制类似于图 2-11 的图，显示出上面列出的存储器引用完成后的高速缓存内容。包括每行修改位的状态，此外要显示出主存储器最终的内容。

2.13 一个程序一次把一个字保存到存储器连续的地址中，以此来使存储器清零。观察一个系统，它采用带有写分配的写回高速缓存，在一开始被清零的存储器块并没有被高速缓存起来的时候，该系统会发生什么样的情况？假定行的大小比一个字大，第一次把数据保存到每一行中的时候会造成一次缺失，要从主存储器读取该行的内容。随后，程序把零保存到高速缓存行中，替换掉从主存储器读取的老数据。因此，读取高速缓存行是对存储器带宽的一种浪费，因为 CPU 从来不会读取这些数据。假定要被清零的存储器块的起始地址始终和高速缓存行的起始位置相对应，要被清除的存储量是高速缓存行大小的倍数，那么请推荐一种特殊用途的高速缓存操作，比如一条新指令，从而让存储器的填零操作效率更高。

2.14 进一步的读物

[1] Agarwal, A., Hennessy, J., and Horowita, M., "Cache Performance of Operating System and Multiprogramming Workloads," *ACM Transactions on Computer System*, Vol. 6, No. 4, November 1988, pp. 393-431.

[2] Agarwal, A., Horowitz, M., and Hennessy, J., "An Analytical Cache Model," *ACM Transactions on Computer System*, Vol. 7, No. 2, May 1989.

[3] Alexander, C., Keshlear, W., Cooper, F., and Briggs, F., "Cache Memory Performance in a UNIX Environment," *SigArch News*, Vol. 14, No. 3, June 1986, pp.41-70.

[4] Alexandridis, N., *Design of Microprocessor Based Systems*, Englewood Cliffs, NJ: Prentice Hall, 1993.

[5] Alpert, D., and Flynn, M., "Performance Tradeoffs for Microprocessor Cache Memories," *IEEE Micro*, Vol. 8, No. 4, August 1988, pp. 44-55.

[6] Cohen, E.I., King, G.M., and Brady, J.T., "Storage Hierarchies," *IBM Systems Journal*, Vol. 28, No. 1, 1989, pp. 62-76.

[7] Cole, C.B., "Advanced Cache Chips Make the 32-Bit Microprocessors Fly," *Electronics*, Vol. 60, No. 13, June 11, 1988, pp. 78-9.

[8] Deville, Y., "A Low-Cost Usage-Based Replacement Algorithm for Cache Memories," *Computer Architecture News*, Vol. 18, No. 4, December 1990, pp. 52-8.

[9] Duncombe, R.R., "The SPUR Instruction Unit: An On-Chip Instruction Cache Memory for a High Performance VLST Multiprocessor," Technical Report UCB/CSD 87/307, Computer Science Division, University of California, Berkeley, August 1986.

[10] Easton, M., and Fagin, R., "Cold Start vs. Warm Start Miss Ratios," *Communications of the ACM*, Vol. 21, No. 10, October 1978, pp. 866-72.

[11] Gecsei, J. "Determining Hit Ratios for Multilevel Hierarchies," *IBM Journal of Research and Development*, Vol. 18, No. 4, July 1974, pp. 316-27.

[12] Goodman, J.R., "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings of the 10th Annual Symposium on Computer Architecture*, June 1983. pp. 124-31.

[13] Haikala, I.J., and Kutvonen, P.H., "Split Cache Organizations," *Performance '84*, 1984, pp. 459-72.

[14] Handy, J., *The Cache Memory Book*, Boston, MA: Academic Press, 1993.

[15] Higbie, L., "Quick and Easy Cache Performance Analysis," *Computer Architecture News*, Vol. 18, No. 2, June 1990, pp. 33-44.

[16] Hill, M.D., "The Case for Direct-Mapped Caches," *IEEE Computer*, Vol. 21, No. 12, December 1988, pp. 25-41.

[17] Hill, M.D., and Smith, A.J., "Experimental Evaluation of On-Chip Microprocessor Cache Memories," *Proceedings of the 11th Annual International Symposium on Computers Architecture*, June 1984, pp. 158-66.

[18] Hill, M.D., and smith, A.J., "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers*, Vol. 38, No. 12, December 1989, pp. 1612-30.

[19] Jouppi, N.P., "Cache Write Policies and Performance," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 191-201.

[20] Laha, S., Patel, J.H., and Iyer, R.K., "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems," *IEEE Transactions on Computers*, Vol. 37, No. 11, November 1988, pp.1325-36.

[21] Lorin, H., *Introduction to Computer Architecture and Organization, Second Edition*, New York, NY: John Wiley & Sons, 1989.

[22] Mano, M.M., *Computer System Architecture, Third Edition*, Englewood Cliffs, NJ: Prentice Hall, 1993.

[23] Przybylski, S.A., *Cache and Memory Hierarchy Design*: A Performance-Directed Approach, San Mateo, CA: Morgan Kaufmann Publishers, 1990.

[24] Rao, G.S., "Performance Analysis of Cache Memories," *Journal of the ACM*, Vol. 25, No. 3, July 1978, pp. 378-95.

[25] Short, R.T., and Levy, H.M., "A Simulation Study of Two-Level Caches," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988, pp. 81-9.

[26] Smith, A.J., "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory," *IEEE Transactions on Software Engineering*, Vol. 4, No. 2, March 1978, pp. 121-30.

[27] Smith, A.J., "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer*, Vol. 11, No. 12, December 1978, PP. 7-21.

[28] Smith, A.J., "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, September 1982, pp.473-530.

[29] Smith, A.J., "Cache Evaluation and the Impact of Workload Choice," *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 1985, pp. 64-73.

[30] Smith, A.J., "Problems, Directions, and Issues in Memory Hierarchies," *Proceedings of the 18th Annual Hawaii Conference on System Sciences*, 1985, pp. 468-76.

[31] Smith, A.J., "Bibliography and Readings on CPU Cache Memories and Related Topics," *Computer Architecture News*, Vol. 14, No. 1, January 1986, pp. 22-42.

[32] Smith, A.J., "Design of CPU Cache Memories," *Proceedings of the IEEE TENCON*, August 1987, pp. 30.2.1-30.2.10.

[33] Smith, A.J., "Line (Block) Size Choice for CPU Cache Memories," *IEEE Transactions on Computers*, Vol. 36, No. 9, September 1987, pp. 1063-75.

[34] Stone, H.S., *High Performance Computer Architecture, Third Edition, Reading*, MA: Addison-Wesley, 1993.

[35] Strecker, W.D., "Transient Behavior of Cache Memories," *ACM Transactions on Computer Systems*, Vol. 1, No. 4, November 1983, pp. 281-93.

[36] Smith, J.E., and Goodman, J.R., "A Study of Instruction Cache Organizations and Replacement Policies," *Proceedings of the 10th Annual International Symposium on Computer Architecture*, June 1983, pp. 64-73.

[37] Thiebaut, D.F., "On the Fractal Dimension of Computer Programs and Its Application to the Prediction of the Cache Miss Ratio," *IEEE Transactions on Computers*, Vol. 38, No. 7, July 1989, pp. 1012-27.

[38] Thompson, J.G., "Efficient Analysis of Caching Systems," Technical Report UCB/CSD 87/374, Computer Science Division, University of California, Berkeley, October 1987.

[39] Thompson, J.G., and Smith, A.J., "Efficient (Stack) Algorithms for Analysis of Write-Back and Sector Memories," *ACM Transactions on Computer Systems*, Vol. 7, No.1, February 1989, pp. 78-116.

[40] Welch, T.A., "Memory Hierarchy Configuration Analysis," *IEEE Transactions on Computers*, Vol. C-27, No.5, May 1978, pp. 408-13.

虚拟高速缓存

第3章

虚拟高速缓存以被高速缓存的指令或者数据的虚拟地址来做索引和标记。这就给操作系统带来了许多复杂因素，因为不同的进程可以使用相同的虚拟地址。一个进程高速缓存的数据可能被错误地当作属于另一个进程的数据。为了防止出现这种情况，操作系统必须在发生这类歧义之前冲洗高速缓存。本章阐述虚拟高速缓存的操作，歧义（ambiguity）和别名（alias）是如何出现的，以及在一个单处理机环境下的操作系统如何防止它们影响程序的执行。

3.1 虚拟高速缓存的操作

在采用虚拟高速缓存的情况下，程序的虚拟地址被用来索引高速缓存和标记数据。这种方法的主要优点是不需要在每次读取或者写入操作的时候把虚拟地址转换为物理地址，就能够存取高速缓存。如图 3-1 所示，正在 CPU 上执行的一个程序确定了它希望读取或者保存的数据的虚拟地址，或者是它希望取得的指令的虚拟地址。这个虚拟地址被发送到高速缓存，高速缓存执行一次查找操作，以检查数据是否在高速缓存中。如果在一次读取操作期间高速缓存中发生了一次缺失，那么就把存储器管理单元所计算出的物理地址发送给主存储器，主存储器随后返回数据（有些实现和高速缓存查找操作并行地开始地址转换）。接着，数据载入高速缓存，于是将来的引用就可以得到它（局部性），并且被返回给 CPU。

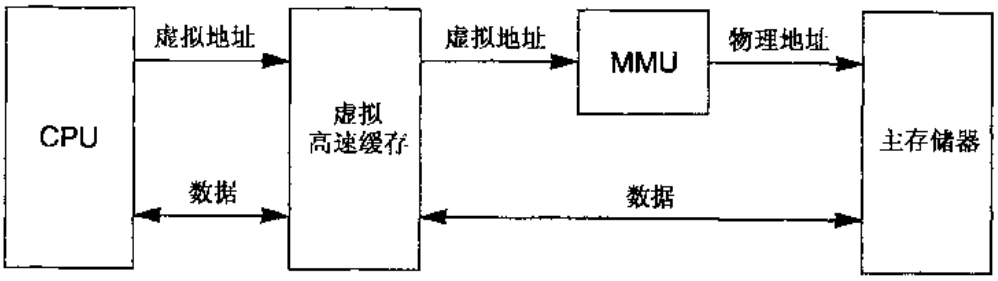


图 3-1 虚拟高速缓存的组织结构

使用虚拟高速缓存的系统可能会在引发高速缓存命中的读取操作期间使用 MMU，也可能不会。例如，Intel i860 系列的处理器就将虚拟高速缓存用于其片上指令高速缓存。这些处理器的 MMU 在取指令期间不会转换在指令高速缓存内引发命中的虚拟地址。这意味着硬件

不会显式地确认存取权限。在 Apollo DN4000 和 Sun 3/200 上的虚拟高速缓存也是如此。这些系统遵循图 3-1 所示的模型，其中的 MMU 只有在高速缓存中出现一次缺失之后才转换虚拟地址。以这种方式使用虚拟高速缓存的系统明确地假定，如果一个程序在一次缺失操作期间成功地读取了数据，那么它仍然有权在后来的高速缓存命中期间读取该数据。正如后面几小节中阐述的那样，操作系统必须确保程序不再有权读取的数据在高速缓存中无效。

保存的步骤如下：如果采用了写直通策略，那么就把要写入的数据及其虚拟地址传送给高速缓存。虚拟地址立即被送往 MMU 进行转换和检查权限，以确保该进程有权向这个地址写入数据。如果进程有写权限，而且在高速缓存中命中，那么就把新数据插入高速缓存行，并且用 MMU 转换后的地址写入主存储器。如果发生缺失，而且使用了写分配机制，那么高速缓存会先使用转换后的地址从主存储器读取高速缓存行（只有当高速缓存行的大小比 CPU 写入的数据大的时候才行，解释见 2.2.5 小节）。接着要保存的新数据被插入高速缓存行，并被写入高速缓存。如果没有采用写分配机制，那么在保存操作造成缺失期间高速缓存的内容不会发生变化。无论是哪一种情况，要保存的数据都会用物理地址写入主存储器。如果这个地址不允许有写权限，那么就向 CPU 触发一次陷阱信号，高速缓存和主存储器都不会有变化。

如果使用写回高速缓存策略，那么确保不违反存取权限要稍微复杂一些。假定采用了写分配机制（写回高速缓存一般会采用），而且在一次写操作时高速缓存内没有虚拟地址，那么 MMU 就需要转换地址，并且像以前那样从主存储器取出完整的高速缓存行。这就有机会确认存取权限。如果允许写权限，新数据就从主存储器插入高速缓存行，并且设置该行的修改位。如果没有采用写分配机制，那么在发生一次缺失的时候，就和没有写分配机制的写直通高速缓存一样：数据只被写入主存储器，高速缓存保持不变。

如果在向写回虚拟高速缓存中保存数据期间发生了一次高速缓存命中，而且该高速缓存行已经修改过了（由标记中的修改位指示出来），那么就能明确地假定进程仍然有权把数据写入这个地址，因为当初它修改这行的时候必须有写权限。在这种场景里，命中了修改过的行时不需要显式的使操作有效。和以前一样，如果进程的执行期间存取权限变化了，那么操作系统必须冲洗高速缓存。

如果要尝试的写操作命中了高速缓存中没有修改过的行，那么就出现困难了。因为不知道是否有写权限，所以就把该地址发送到 MMU 进行确认。这就是 Intel i860 XR 的片上虚拟写回数据高速缓存起作用的方式。它的 MMU 检查所有对数据高速缓存访问的存取权限（命中和缺失都检查）。这种确认是和高速缓存查找操作并行进行的，从而防止高速缓存访问的速度降低。如果一次保存操作允许写访问，那么就把新数据插入到高速缓存行，并且设置该行的修改位。在这种情况下根本不需要访问主存储器，因为这一行已经在高速缓存中了。

像 Apollo DN4000 这样，采用写回高速缓存，只在一次缺失之后转换虚拟地址和检查存取权限的实现，通过在标记的控制部分中加入一个可写位，克服了在这种情况下串行执行高速缓存查找操作/MMU 确认操作所造成的性能损失。只要在一次高速缓存缺失期间读取了新行，而且进程有权写入相应的地址，那么就设置这一位。这一位是页表（page table）内写权限位的一个副本，它能够让高速缓存自身完全处理写入确认，从而节省了多余 MMU 操作的开销。这类实现对于操作系统来说是透明的。

因为写回高速缓存机制把修改过后的数据留在了高速缓存中，所以当一行在缺失处理期间被替换或者显式地被操作系统冲洗掉（指定主存储器应该用修改过的数据进行确认）的时

候，就必须把它写回。如果在这些情况下写回操作是必需的，那么高速缓存就把要写回的数据的虚拟地址（保存在标记中）发送给 MMU。一旦计算出了物理地址，就把被修改过的行写入主存储器。

3.2　虚拟高速缓存的问题

虚拟高速缓存通过省略地址转换步骤能够加快向 CPU 返回数据的速度，它是操作系统最难以管理的高速缓存类型。问题围绕着这一事实而出现，即虚拟地址用来检索和标记高速缓存行。虚拟地址不能唯一地确定一段数据，因为多个进程可以使用相同范围内的虚拟地址。除非操作系统能够采取步骤来防止它们，否则这会在高速缓存中造成歧义和别名现象。下面两小节将讨论这些问题。

3.2.1　歧义

当不同的数据段在高速缓存中有相同的索引和标记时，就出现了歧义现象。这意味着高速缓存将不能区分两段不同的数据，因为索引和标记是高速缓存仅有的确定数据的手段。引用这样的数据则被称为有歧义。对于一个虚拟高速缓存来说，只要使用的虚拟地址在不同的时间点有不同的物理地址映射关系，就会发生歧义。例如（使用图 3-2 给出的物理存储器的内容），如果在第 1 个时刻，虚拟地址 0x1000 被映射到物理地址 0x5000，那么如果引用这个地址，就会把 5678 读入高速缓存，如图 3-3 所示。再次引用它会在高速缓存中命中，从而不访问主存储器（注意，为了清楚起见，在标记中给出了完整的虚拟地址。实际的实现只会保留散列算法不用的地址中前面部分的比特位）。

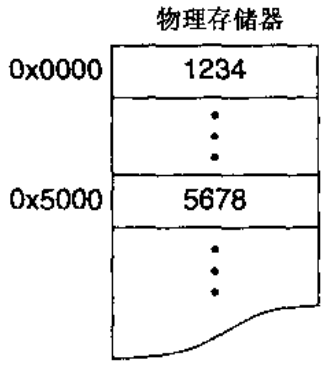


图 3-2

如果在第 2 个时刻，映射关系改到了物理地址 0x0000 上，那么引用虚拟地址 0x1000 还是会返回在物理地址 0x5000 处的数据，因为虚拟地址仍然在高速缓存中产生一次命中（参见图 3-4）。高速缓存查找操作只是基于虚拟地址进行的，所以即使物理地址已经变了，还是给该虚拟地址 0x1000 产生了相同的索引和标记。

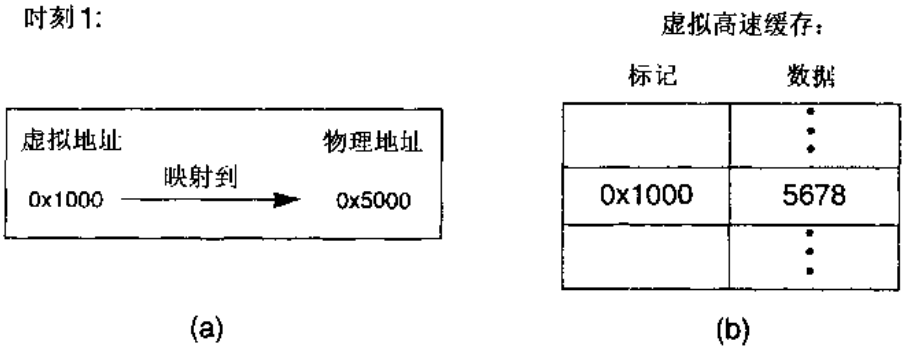


图 3-3 (a)时刻 1 的映射关系

(b)在引用 0x1000 之后的高速缓存内容

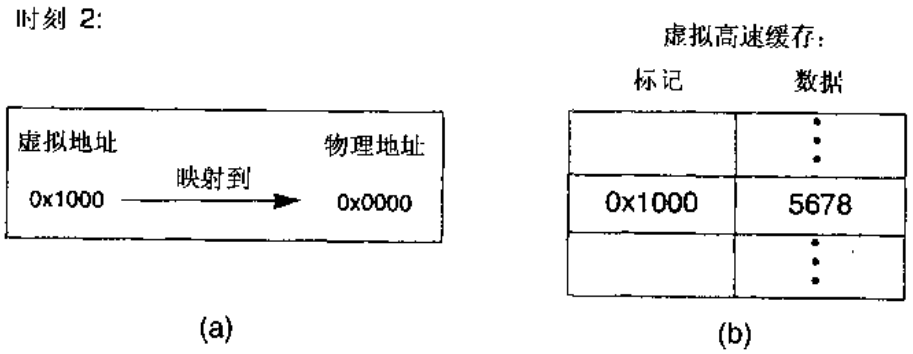


图 3-4 (a)时刻 2 的映射关系

(b)映射关系改变之后高速缓存的内容没有受到影响

结果，程序将继续获得错误的数据（和主存储器内保存的数据不一致），直到包含错误数据的高速缓存行在缺失处理期间被替换，或者显式地从高速缓存中冲洗掉为止。高速缓存不能自动地这样做，因为它只处理虚拟地址，不知道物理地址已经变了。

操作系统负责确保在虚拟地址空间中发生任何变化之前把任何老数据写回到主存储器，从而避免出现歧义。如果让它们发生了，那么程序就会零星地从高速缓存中读取错误的数据，从而导致无法预料和不确定的行为。如果听任操作系统自己的数据出现歧义，那么系统就有可能崩溃，或者至少是运行不正确。

3.2.2 别名

别名也叫做同义（synonym），当使用一个以上虚拟地址来指代相同的物理地址时就发生了别名现象（多个虚拟地址被称为别名）。如果一个进程将同一共享存储区附加在它的地址空间中两个不同的虚拟地址上，或者两个不同的进程在它们各自地址空间的不同地址上使用同一共享存储区，就会发生别名现象。如果每个地址经散列算法计算出不同的行索引，那么相同的数据就会被保存在高速缓存中不同的地方。如果两个版本的数据彼此失去同步，就会发生意想不到的结果。

考虑一个进程，它将物理存储器地址 0x3000 处的存储页面映射到虚拟地址 0x2000 和 0x4000，如图 3-5 所示（假定存储页面有 0x1000（4K）字节大）。有了这样的映射关系，程序既能从 0x2000 也能以 0x4000 读取数据，并且会得到相同的结果，因为这两个地址都引用了存储器内相同的物理页面。假定这个例子中的系统有一个直接映射的虚拟高速缓存，它使用虚拟地址的位<15..4>作为索引。这意味着地址 0x2000 经散列算法得到高速缓存行 0x200，而 0x4000 则得到行 0x400。如果物理地址 0x3000 处的第一个字为 1234，那么如果进程先后引用了两个虚拟地址 0x2000 和 0x4000，则物理地址 0x3000 处的值就会被读入高速缓存内两个不同的地方（如图 3-6 所示），因为这两个地址产生了不同的索引。

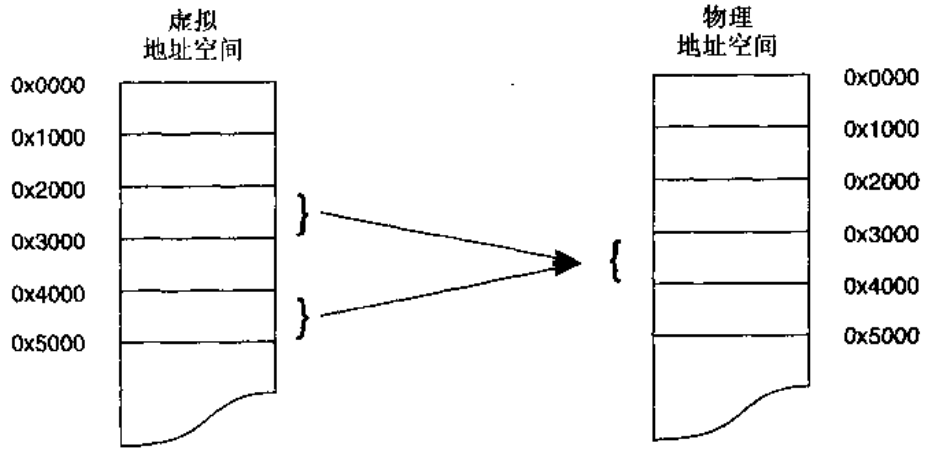


图 3-5　位于 0x3000 物理页面处的两个虚拟地址别名

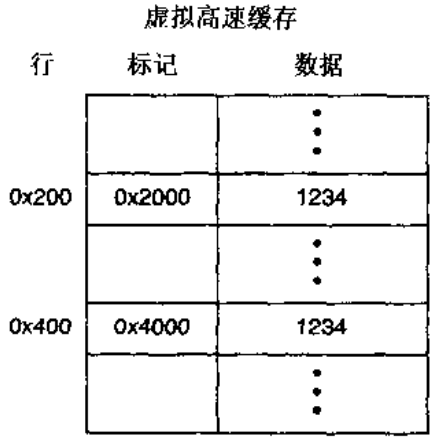


图 3-6　在引用 0x2000 和 0x4000 之后虚拟高速缓存的内容

迄今为止，因为使用任何一个地址都返回了正确的数据，所以进程还没有受到错误的影响。在这个进程尝试向两个地址中的一个写入数据之前，它还能继续在这些位置接收正确的数据。如果它现在把 5678 写入虚拟地址 0x2000，那么行 0x200 就会更新，但是位于行 0x400 的别名不会更新，这将导致两个别名没有包含一致的数据。图 3-7 描绘了这种情况。

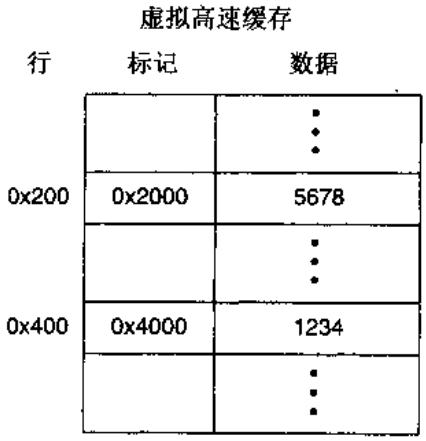


图 3-7 在写入 0x2000 之后虚拟高速缓存的内容

引用虚拟地址 0x4000 将返回物理地址 0x3000 中原来的过时内容。它们会继续这样做，直到这一行在缺失处理期间被替换或者被显式地冲洗掉为止。高速缓存中的过时数据会在程序执行过程中造成奇怪的和无法预测的行为。如果进程把不同的值写入虚拟地址 0x4000，那么会出现奇怪的结果。进一步的结果会根据高速缓存是使用写直通还是写回策略而有所变化（参见习题 3.2）。

如果别名产生了相同的高速缓存行索引，那么别名的影响会完全不同。如果前面例子中所使用的虚拟地址变为 0x2000 和 0x12000，那么两个地址都会索引到高速缓存中的 0x200 行。因为高速缓存是直接映射高速缓存，所以它一次只能保留这两个地址中某一个地址的数据，这样每次都造成高速缓存缺失（因为虚拟地址和标记的当前值不吻合），而要从物理地址 0x3000 重新读取该值。在这种情况下，进程一定能够得到正确的数据，因为每次引用一个别名都会让高速缓存中包含另一个别名的行被替换。

在执行写操作的时候情况就不同了。如果高速缓存使用写分配机制，那么如果进程轮流向两个别名写入数据，它就可以读取到正确的数据。每次都会发生缺失，要从主存储器重新读取该高速缓冲行。如果没有使用写分配机制，那么就会发生不一致的现象。在进程向具有写直通策略的高速缓存中 0x2000 处保存数据时，如果发生了一次命中，那么数据就被保存在高速缓存中，并且写入主存储器。如果下次向 0x12000 保存数据，那么就会发生一次缺失，数据则只写入主存储器。此刻，高速缓存仍然保留着第一次保存的过时数据，如果进程从 0x2000 处读取数据就会造成不一致。

现在考虑如果高速缓存是双路组相联高速缓存而不是直接映射高速缓存的情况。两个地址都算出相同的组索引，但是高速缓存现在能够同时高速缓存两个别名。当别名索引到直接映射高速缓存内不同行时出现的问题同样会在这里出现。

如前所述，不同的地址空间之间也能有别名现象。在我们的例子中，虚拟地址别名 0x2000 和 0x4000 可以在两个不同的进程中，这两个进程也能造成和单个地址空间时一样的不一致现象。

导致访问不同高速缓存行的别名使得返回给进程的数据是错误的。操作系统必须防止发生这些情况。

3.3　管理虚拟高速缓存

在 UNIX 系统中，当使用虚拟高速缓存的时候，如果没有正确地管理好它们，就可能出现一系列造成歧义和别名的情形。当进程的地址空间发生变化的时候，这些情形就出现了。下面几个小节详细介绍这些情形，以及为了保持虚拟高速缓存和主存储器的一致性内核必须采取的措施。在所有的情况下，对于那些使用独立的指令高速缓存和数据高速缓存的系统来说（系统中两者都是虚拟高速缓存），指令高速缓存应该和数据高速缓存同时无效。

正如将在以后的章节中看到的那样，通过改变高速缓存的组织结构，以某些方式使用物理缓存，就有可能避免操作系统显式地执行冲洗高速缓存的操作。例如，Intel i860 XP 上的高速缓存在标记中除了包括虚拟地址之外，包括物理地址。这就让硬件能够检测到一些情况下的别名和歧义（这种组织结构将在 6.2.6 节中介绍）。但是，这里的讨论集中在纯虚拟高速缓存上：仅使用虚拟地址的高速缓存。

3.3.1　现场切换

因为每一个进程都有它自己的虚拟地址空间，所以两个不同的进程就有可能使用相同的一组虚拟地址来引用它们的正文、数据和堆栈。在没有虚拟高速缓存的系统上，这样两个使用相同虚拟地址的进程只能访问它们自己的地址空间，因为虚拟地址空间的映射在现场切换的时候改变了。这两个虚拟地址空间中的每一个都映射到了不同的物理页面上，所以没有哪一个进程能意识到对方的存在。

在向系统加上虚拟高速缓存的时候，两个不同进程的数据之间就发生了歧义。因为进程可能使用相同的虚拟地址空间在现场切换之前和之后引用不同的物理地址，所以会发生歧义。一个虚拟高速缓存不能区分由不同进程使用的相同的虚拟地址，所以可能会返回错误的数据。正如 3.2.1 小节中的例子那样，如果新进程使用的虚拟地址在高速缓存内产生了相同的索引和标记，那么在现场切换之前进程所高速缓存的任何老数据都会在现场切换之后返回给新进程。这就会导致一次命中，在没有虚拟高速缓存的系统中不会发生防止这类歧义的 MMU 操作。

不但新进程在现场切换之后从高速缓存中接收到了错误的数据，如果采用了写回高速缓存机制，有些老进程的数据也可能会丢失。如果在现场切换之前高速缓存中有修改过的数据，那么如果该行在缺失处理期间被替换，该数据就会被写入新进程的地址空间。这就破坏了新进程的地址空间，而且导致老进程失去数据。

为了防止发生这些问题，内核必须在现场切换的时候从虚拟高速缓存中冲洗掉老进程已经高速缓存过的一切内容（正文、数据和堆栈等）。更特别的是，如果采用了写回高速缓存机制，内核就必须用高速缓存内任何修改过的数据让主存储器有效。这些修改过的数据是老进程状态的一部分，因而必须在现场切换之前保存起来，否则数据可能丢失。接下来，必须使高速缓存中的所有行都无效（正如 2.8 节所提到的那样，使主存储器有效而使高速缓存行无效往往可以作为一次操作来执行）。当新进程在现场切换之后继续执行的时候，所有的存

储器引用都在高速缓存中没有命中，因为所有的高速缓存行都是无效的，从而让进程从它自己的地址空间中取得正确的数据。

在每次现场切换的时候冲洗虚拟高速缓存可能是一项耗时的操作，尤其是采用了大规模的写回高速缓存时更是如此。冲洗高速缓存所需的时间不仅和高速缓存的大小成比例，而且也和高速缓存行的数量成比例，因为所有修改过的行都被写回到主存储器中。除了冲洗高速缓存自身的开销之外，还有一项副作用就是新进程会在所有初次引用存储器的时候没有命中，因为高速缓存刚刚完全无效处理过。进程以前拥有的局部引用都不在高速缓存中了，所以在进程重建其高速缓存局部性的同时命中率较低。如果现场切换过于频繁，就好像是交互性很高的 UNIX 应用碰到的情况那样，那么系统中有高速缓存的好处就会降低，因为命中率低而且操作系统冲洗高速缓存所需的开销大。大规模的高速缓存最适合于计算密集型、面向批处理的应用环境，在这样的环境中不会频繁地出现现场切换。

3.3.2 fork

系统调用 fork 的语义要求子进程得到一份父进程地址空间的完整副本。如果在父进程执行 fork 系统调用之前写回高速缓存内缓存了任何修改过的数据，那么这些数据就会出现在子进程的地址空间中。不管采用的高速缓存是什么类型，都必须考虑这种情况。

因为在每次现场切换期间都已经冲洗过高速缓存，所以对于虚拟高速缓存来说，需要有一点儿额外的高速缓存管理操作来正确地处理系统调用 fork。这会消除父进程的数据和子进程的数据之间大多数的不一致，但是必须考虑复制操作期间的高速缓存一致性和写回高速缓存机制的影响。

如果系统没有采用写时复制（copy-on-write）方式，那么在 fork 期间必须把父进程完整的虚拟地址空间复制一份给子进程。要做到这一点，内核一般要把子进程会使用的物理页面也映射到内核地址空间中一块未用的虚拟存储区，如图 3-8 所示。

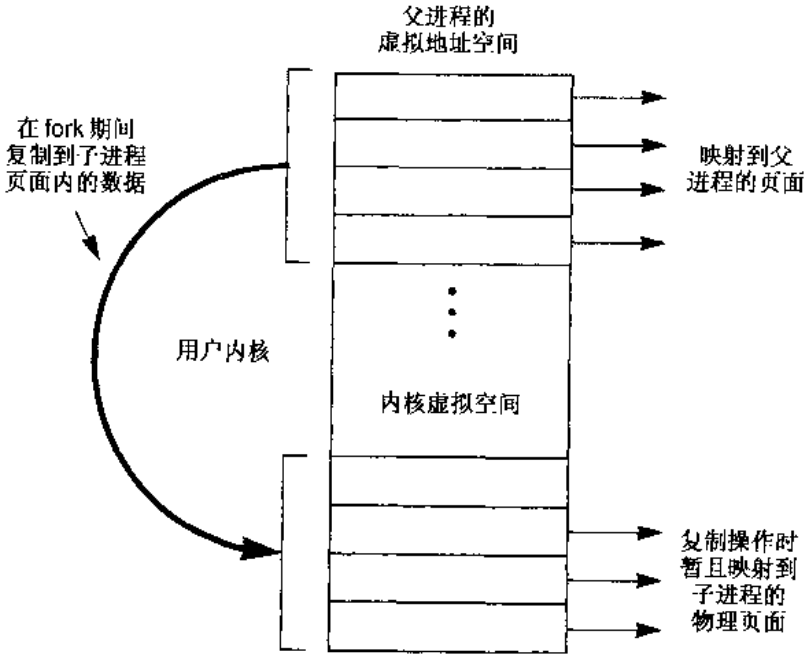


图 3-8 在为 fork 复制地址空间期间映射存储器

这样做能让内核仅仅使用一组不同的虚拟地址就可以从父进程的现场引用子进程的页面。数据沿着图 3-8 中的大箭头方向从父进程的用户虚拟地址复制到临时的映射到子进程页面的内核虚拟地址。一旦复制操作完成，那么就删除到子进程页面的映射。

因为使用父进程自己的虚拟地址作为复制源，所以能保证在 fork 之前高速缓存中存在的任何修改过的数据都被复制到子进程的地址空间中，因为用作复制源的地址会在那些高速缓存行上产生命中。于是这就能在 fork 期间满足高速缓存一致性的要求之一。

当子进程开始执行的时候，必须保证主存储器已经是最新的，而且在高速缓存中没有过时的数据。如果采用写直通高速缓存策略，那么即使在复制期间所使用的内核虚拟地址是子进程地址的别名，在 fork 期间也不需要特殊的冲洗操作，因为至少在子进程执行之前必须有一次现场切换。在现场切换时冲洗高速缓存，保存在高速缓存中以内核虚拟地址标记的任何数据都会消失。但是，如果采用写回高速缓存策略，那么在删除临时映射之前必须让主存储器有效，以便子进程的物理页面也会被更新。之所以必须这样做，是因为高速缓存在使存储器有效期间使用虚拟地址，所以说，为了让 MMU 转换地址，仍然要有映射关系。

如果是写直通高速缓存，而且它支持一种无高速缓存模式（uncached mode），那么可以在复制期间标记所使用的内核虚拟地址为不缓存。使用那些虚拟地址的时候，这些目的地址只被写入一次，而且不会再读取。因此，高速缓存这些数据也就没有什么裨益了。高速缓存这些数据也可能造成有用的数据被替换掉。注意，如果高速缓存没有使用写分配机制，那么无需显式地标记存储页面为不缓存的也能取得相同的效果。如果高速缓存采用了带有写分配机制的写回策略，那么复制时使用不缓存的存储方式可能有好处，也可能没有好处。有没有好处则取决于行的大小以及主存储器和高速缓存的速度之比。例如，如果高速缓存行很短，那么每次存储缺失的时候读取行的开销会相当大。但是，如果行很长，那么连续的保存操作将会产生高速缓存命中，因而就能减少向主存储器写入的次数，那么读取高速缓存行首次缺失的开销可能也是值得的。当然，如果高速缓存没有写分配机制，那么就会直接保存到主存储器里。

如果采用写时复制技术来实现 fork 调用，那么在执行 fork 的时候不需要冲洗写直通高速缓存，因为没有进行复制，主存储器内的数据副本还是最新的。如果采用了写回高速缓存机制，那么在 fork 操作期间必须使主存储器有效，以便删除父进程可能在高速缓存中留下的任何修改过的数据。如果在高速缓存中留下了修改过的数据，那么随后对修改过的行进行的写回操作（在行替换或者现场切换期间）会错误地导致写时复制缺失错（copy-on-write fault）出现。这些写时复制缺失错会让父进程接收到它自己的受影响页面的副本，进而导致子进程丢失了修改过的数据。对于像 Apollo DN4000 上的高速缓存来说就是这样，它们包含有一个可写位（writable bit）。在这种情况下，fork 之前可以设置每一个高速缓存行上的这一位。如果在 fork 返回父进程之前没有出现现场切换，那么父进程可以修改这些行，而不会产生写时复制缺失错（因为发生了一次命中，意味着 MMU 没有用于那次访问）。这两种情况都会违反系统调用 fork 的语义，一定不能允许其发生。此外，有些体系结构（比如 Intel i860 系列）在写回操作期间不检查写权限。这意味着在 fork 之后写回，但在此之前修改过的行不会产生写时复制缺失错。在 fork 期间使主存储器有效能解决这些问题。

当父进程或者子进程随后试图修改通过写时复制共享的页面时，则把一个新物理页面映射到内核的虚拟地址空间而只建立那个页面的一个副本，这类似于图 3-8 所示的例子。主要的区别在于是复制一页而不是整个地址空间。和以前一样，这样做和用户虚拟地址空间一起有了该页面的一个别名。但是，在这种情况下，控制权会交还给用户程序，中间没有现场切换。因此，对应于复制所用临时区的高速缓存项必须写回主存储器（如果是写回高速缓存的话），并且在高速缓存中令其无效。同样，使用无高速缓存的方式访问复制的目的地能够防止出现别名。

注意，如果 fork 使用了写时复制的共享机制，那么在别处认为是父进程和子进程的地址空间之间出现歧义的情况，此时则是可以接受的。因为两个进程一开始在 fork 之后立即共享相同的物理页面，所以即使在父进程和子进程之间发生了现场切换，两个进程之间的虚拟地址也不会有歧义，反之亦然。随后，内核可以消除这样环境下的高速缓存冲洗操作。但是要注意，一旦两个进程中的一个修改了它的数据，那么一定要继续执行高速缓存冲洗操作，因为这会结束该页的写时复制共享。于是引用那一页会有歧义。几乎没有哪个程序能运行这么长时间而不向它们的堆栈或者数据段写入数据。即使子进程立即执行 exec，在 exec 完成之前，任何在堆栈段上传递参数的实现都会导致出现写时复制缺失错。于是，虽然在这种情况下似乎有希望消除现场切换时的冲洗操作，但是实际上却几乎节省不了什么开销。此外，如果需要现场切换到任何其他的进程上去，那么这项技术根本无法使用。

3.3.3 exec

系统调用 exec 丢弃了进程的当前地址空间，而以进程将要运行的新程序的一个新地址空间来替换它。因为新程序有可能驻留在和老程序一样的虚拟地址范围内，所以一旦新程序开始执行，老程序已经高速缓存的任何数据都会造成歧义。如果出现了歧义，新程序就会接收到一些老程序的数据，就好像这些数据是新程序自己的一样，如果老程序的指令被高速缓存下来，那么新程序甚至可能执行一些老程序的指令。为了防止出现这种情况，内核必须在新程序开始执行之前让高速缓存中的所有用户数据都无效。在采用写回高速缓存的情况下，必须以任何修改过的数据使主存储器有效，因为在执行 exec 调用期间，老程序的地址空间已经被丢弃了。

3.3.4 exit

在 exit 系统调用期间会丢弃一个进程的地址空间。内核必须确保也丢弃任何已经高速缓存的数据，从而不会在下一个进程运行的时候出现歧义。系统调用 exit 处理的最后一步就是执行一次现场切换，在运行下一个进程之前冲洗高速缓存。因此，不必在 exit 处理过程中加入一次高速缓存冲洗操作。但是，如果采用了写回高速缓存机制，那么现场切换代码需要知道它是否作为 exit 的一部分被调用。在正常情况下，现场切换代码必须写回高速缓存中修改过的数据。但是，在 exit 的情况下，老的地址空间已经被丢弃了，所以没有地方写回数据。因此，在系统调用 exit 之后的现场切换期间必须省略写回操作。

3.3.5　brk 和 sbrk

系统调用 brk 和 sbrk 用于增大和缩小进程的 bss 段。增大 bss 段不会带来高速缓存的问题，因为分配了新的虚拟存储空间。进程还没有对应于新虚拟存储空间的任何过时的高速缓存数据，如果因为它试图向段外的地方写数据，就会发生一次缺失错。而且，在每次现场切换期间进行高速缓存冲洗也能防止可能在相同虚拟地址范围内的其他进程数据产生任何歧义。

但是，缩小 bss 段就出现问题了，因为必须防止进程访问相应于刚刚释放掉的虚拟存储区的高速缓存数据。要记住，不是所有的实现都将 MMU 用于高速缓存命中的读取操作。所以，除非内核明确地让数据在高速缓存中变得无效，否则进程仍然能够从虚拟存储中被释放掉的部分读取高速缓存数据。注意，对 exit 和 exec 来说，如果采用了写回高速缓存机制，那么用修改过的数据让主存储器有效的做法也不正确，因为那部分地址空间已经丢弃了。

和在其他所有情况下一样，在使用独立的指令和数据高速缓存的系统上缩小 bss 段时，必须使指令高速缓存无效。虽然这样做似乎没必要，因为 bss 是程序的数据区的一部分，但是总是有可能有一个解释器已经把指令写入了这个区域，或者从这个区域执行指令。如果出现了这样的情况，那么那些指令仍然能被高速缓存，并且必须使之无效。如果没有这么做，而 bss 后来重又增大（在下一次现场切换之前），并向其中写入了新指令，那么程序就可能执行自上次缩小操作后遗留下来的过时指令。

3.3.6　共享存储器和映射文件

如果不同的进程使用不同的虚拟地址来共享相同的共享存储段，那么共享存储的设施就可能带来高速缓存的别名现象。但是，并不需要特殊的高速缓存管理措施，因为在每次现场切换的时候都已经冲洗了高速缓存。这将自动消除共享存储内的任何别名，因为在共享存储的其他进程运行之时，已经冲洗过高速缓存了。

考虑如果一个进程将一个共享存储段附加在它自己的地址空间内不同的地址上，将会发生什么情况。这意味着它能引用任何别名而在中间没有一次现场切换和高速缓存冲洗操作，从而导致出现 3.2.2 小节末尾所描述的问题。操作系统必须防止出现这些别名问题。它可以建立共享存储段使之不会被高速缓存，或者它可以简单地禁止任何试图访问附加到进程地址空间上的相同共享存储段一次以上的行为。如果高速缓存是直接映射高速缓存，而且使用了写分配机制，那么也可能采用另一种方法。在这种情况下，如果每次附加的起始地址都索引到相同的高速缓存行的话，操作系统就可以让共享存储区在一个进程的地址范围内附加多次。正如 3.2.2 小节所阐述的那样，每次引用其中的一个别名都会迫使以前高速缓存的任何别名被替换掉，因而会消除别名问题。在这种情况下，限制共享存储段的附加地址是比禁止在同一进程内多次附加相同共享存储段的做法更好的方法。它的性能也比不缓存的访问要好。先前提到的 Sun 和 Apollo 系统都采用了这种方法。和以前一样，如果高速缓存每组有两行或者更多行，或者高速缓存没有使用写分配机制的话，那么这一解决方法也不行。

通过一次仅让其中一个别名的页面有效，可能也是一种解决方法。这样一来，在程序尝试访问另一个别名的时候，就会出现缺页错。此时，对于上次用过的别名来说，内核可以让

主存储器有效，而让高速缓存无效，并且标记来自那个别名的页面都是无效的。标记新近引用的别名的页面有效，从而让进程能够访问它们。这个步骤在进程每次访问不同的别名时重复一次。每次访问不同的别名时出现的缺页错让内核透明地保持高速缓存一致性，代价是增加了内核的开销。

如果进程要从它的地址空间中分离一段共享存储区，那么过时的数据可以保留在高速缓存中。这类似于完成一次缩小 bss 段的系统调用 sbrk 时所发生的情况。对应于被共享存储区所占用的虚拟地址的高速缓存数据必须在高速缓存中变成无效的。如果共享存储段仍由其他进程使用，而且使用了写回高速缓存，那么先要让主存储器有效。如果要销毁共享存储段的话，可以省略这一步。

3.3.7 输入输出

因为内核在系统调用 read 和 write 期间运行在进程的地址空间中，所以在有虚拟高速缓存的系统上针对缓冲文件（buffered file）的 I/O 操作都能正确发挥作用。这是因为内核使用和用户程序一样的虚拟地址空间来引用用户和内核缓冲区之间复制的数据。因此，不会出现别名或者歧义现象。

I/O 设备一般直接连接到主存储器上，而且以物理地址使用 DMA 操作来传输数据。将图 3-1 扩充包含 I/O 设备后如图 3-9 所示，可以看到，DMA 操作不必通过高速缓存就能直接访问主存储器。

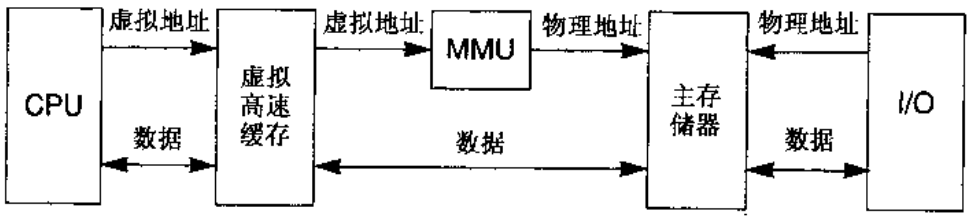


图 3-9 I/O 访问主存储器

因此，没有缓冲的 I/O 所产生的问题类似于别名，因为 I/O 子系统通过 DMA 引用数据所采用的地址（物理地址而不是虚拟地址）和 CPU 通过高速缓存引用相同数据时所采用的地址不同。DMA 操作不检查高速缓存命中，因为 I/O 设备没有直接连接到高速缓存上，DMA 使用物理地址（不能用它来索引一个虚拟高速缓存）。甚至仅有的几个有虚拟 DMA 功能的系统，即使 I/O 子系统得到的虚拟地址而不是物理地址，往往也还是选择绕过高速缓存直接访问主存储器；否则，DMA 操作将会和 CPU 竞争对高速缓存的访问。

如果一个用户进程对一个设备执行没有缓冲的 write 调用，而且还使用了写回高速缓存机制，那么对应于 I/O 缓冲区的修改过的数据可能仍然在高速缓存中，这意味着在主存储器内的数据过时了。当主存储器发生 DMA 操作的时候，它会获得过时的数据。为了防止发生这样的情况，在 DMA 操作开始之前，内核必须用高速缓存内对应于 I/O 缓冲的被修改过的数据让主存储器有效。

类似地，在没有缓冲的 read 调用之前，可以高速缓存对应于 I/O 缓冲的数据。如果在这样的情况下允许 DMA 操作，那么来自设备的新数据就会被放入主存储器，而高速缓存不受影响。没有什么能阻止用户进程引用过时的高速缓存数据。这种状况会持续下去，直到在缺失处理期间从高速缓存中删除了过时的数据为止。为了防止使用过时的高速缓存数据，必须从高速缓存中冲洗掉（使之无效）将要通过 DMA 读入的 I/O 缓冲区所对应的任何数据。此刻没有时机用高速缓存内修改过的数据使主存储器有效，因为 DMA 会覆盖它。

有可能通过把未缓冲的 I/O 操作和现场切换期间完成的高速缓存冲洗操作结合起来，从而消除为未缓冲的 I/O 操作所显式进行的高速缓存冲洗操作。系统调用 read 和 write 阻塞进程直到 I/O 操作完成为止。这会导致一旦发起 I/O 操作，就会发生一次现场切换。注意，在现场切换时所需要的冲洗正确地处理了的未缓冲的 I/O：为写回高速缓存使主存储器有效，并且使高速缓存无效。所以，只要在 DMA 操作真正开始之前执行了现场切换冲洗操作，那么为 I/O 操作保持数据一致性就不需要额外的冲洗。对于大多数系统来说，这仅仅意味着内核在发起 I/O 操作之前执行现场切换冲洗操作，然后在真正发生现场切换的时候跳过冲洗操作。这项技术不能用在支持异步 I/O（asynchronous I/O）的系统中。异步 I/O 允许 I/O 操作和进程的执行并行完成。因为在 DMA 执行的同时并不需要阻塞进程，所以就不能依赖现场切换时刻的冲洗操作，必须进行显式的冲洗操作。

以上讨论假定原始（未缓冲的）I/O 缓冲区的虚拟地址和一个高速缓存行的边界相对应，而且假定缓冲区是高速缓存行大小的倍数。这是简单的情况，因为在缓冲区所占据的高速缓存行内没有包含其他进程的数据。但是，不需要如此。其他数据有可能和缓冲区相邻或者和缓冲区共享高速缓存行。如图 3-10 所示，考虑下面这部分进程的虚拟地址空间（看作是一个线性的数组）。

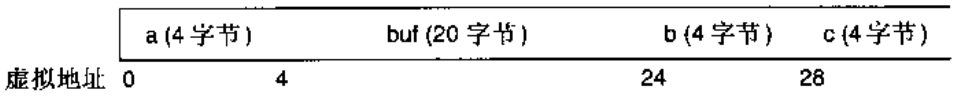


图 3-10　给出 4 个变量的虚拟地址空间

这幅图给出了从虚拟地址 0 开始的 4 个变量：a、buf、b 和 c。变量 a、b 和 c 每个都有 4 字节长，而 buf 则有 20 字节。如果我们现在考察这些变量是怎样在每行 16 字节的高速缓存中出现的（如图 3-11），我们就会看到 buf 跨越了两行高速缓存，而 a、b 以及 c 共享了行的开头和结尾（假定虚拟地址 0 索引到高速缓存行 0）。

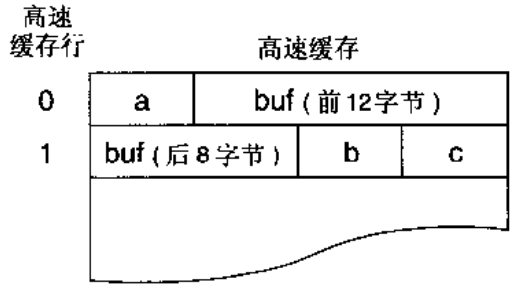


图 3-11　在每行 16 字节的高速缓存内的布局

现在，如果进程执行一次原始read调用，读入buf，那么buf的内容必须是无效的。内核不能简单地让buf占据的所有高速缓存行都无效，因为这样做也会使a、b和c无效。如果采用写回高速缓存，那么这样做能致使这些变量被修改过的版本丢失，从而在将来的引用中使用主存储器内过时的版本。所以，当采用写回高速缓存机制的时候，内核必须首先以I/O缓冲区没有占满的开始和最后的高速缓存行的内容来使主存储器有效，从而确保邻接的变量不受影响。如果采用了前面讨论过的现场切换优化，那么这就不成问题了，虽然如此，理解它还是很重要的。

如果图3-10描绘的I/O缓冲区位于共享存储区内，而且采用写回高速缓存的话，就会出现更复杂的问题（如果使用异步I/O，也会出现这个问题）。在这种情况下，共享相同存储区的另一个进程有可能在DMA访问buf的同时访问缓冲区附近的变量（a、b和c）。如果共享存储的一个进程开始了一次read调用，读入buf，而共享存储的另一个进程开始运行的话，第二个进程就可以引用变量a（举个例子），致使在DMA操作开始之前，包含a和buf前12字节的高速缓存行被带入高速缓存。这意味着buf前12个字节原来的内容被高速缓存下来了。如果现在开始DMA操作，那么它会用来自read的新数据替换存储器内buf的副本。这对高速缓存的内容没有影响，它仍旧保留着缓冲区开头位置原来的数据。现在如果进程修改了a版本在高速缓存内的值，那么当整个高速缓存行被写回存储器的时候，主存储器内缓冲区前12个字节的新数据会被来自高速缓存的原来的数据覆盖掉，从而破坏存储器内buf的副本。观察到这种情形高度依赖于DMA操作的相对执行时刻，以及共享相同存储区域的其他进程的高速缓存存取/写回操作。如果在另一个进程访问共享存储之前就执行了DMA，那么就决不会发生这个问题。

内核一定不能让这类破坏数据的问题出现。对于内核来说，解决这个问题最简单的方法是检测到缓冲区的高速缓存行没有均匀对齐（不是在开头没有对齐，就是在末尾没有对齐，或者开头末尾都没有对齐）。在这些情况下，内核可以在内核存储器中分配另一块缓冲区，在那里可以安全地执行DMA操作。当DMA操作完成的时候，内核就可以把数据复制回共享存储区，而不会干扰其他进程。虽然增加这么一次复制操作似乎减弱了不经缓冲I/O的目的，但是最好不要冒破坏数据的风险。

在内核进行其内部I/O操作的时候必须考虑到本节所描述的问题。这既包括文件I/O，也包括在调页（paging）和交换（swapping）期间所进行的I/O。此外，设备驱动程序也必须意识到系统中存在高速缓存。典型情况下，I/O设备的控制和状态寄存器会映射到存储器中，这意味着它们出现在物理地址空间中，可以通过正常的load和store操作进行引用。这些load和store操作必须通过不缓存的引用方式来完成，从而不会从设备中而是从高速缓存中读取过时的状态信息。类似地，当一条命令被写入控制寄存器的时候，一定不要高速缓存它，并把它保存在写回高速缓存中，否则在该高速缓存行被替换之前，设备都不会收到这条命令。如果高速缓存没有不缓存的模式，那么设备驱动程序就不得不在每次从设备寄存器读取数据的操作之后显式地使高速缓存无效，而在保存操作之后使主存储器有效。

3.3.8　用户-内核数据的歧义

防止内核和用户数据之间出现任何歧义是很重要的。在内核模式（kernel-mode）和用户模式（user-mode）执行期间都会使用高速缓存，所以必须保证用户不能访问任何被高速缓存的内核数据，也必须保证任何被高速缓存的用户数据不会被误认为内核数据。这对于确保系统的完整性和安全性来说是至关重要的。

虚拟高速缓存格外易受这类完整性问题的影响，因为大多数系统都不使用 MMU 来验证高速缓存中命中的访问。这意味着如果被高速缓存的内核数据在系统调用返回用户模式的时候遗留在高速缓存里，那么用户就有可能会通过引用相应的内核地址而读取到这些数据。为了防止用户进程访问到被高速缓存的内核数据，在系统调用或者中断之后，返回用户模式之前，必须冲洗高速缓存（如果采用写回高速缓存机制则使主存储器有效，然后使高速缓存内的数据无效）。如果高速缓存规模大的话，这就会成为一项代价很大的操作，因为 UNIX 程序倾向于频繁地执行系统调用。幸运的是，如果进程的地址空间被分成了图 1-2 所示的用户空间和内核空间，那么被高速缓存的用户数据就不会被误认为是一个系统调用入口的内核数据。这是因为内核会使用不同的虚拟地址来访问它自己的数据，所以即使用户数据碰巧被高速缓存在了索引行中，高速缓存也一定不会返回一次命中。

为了克服每次返回用户模式的时候不得不冲洗高速缓存所造成的性能上的损失，有些实现（比如 Apollo DN-4000）在标记中加入一个比特位来指出数据是在用户模式还是在内核模式读取的。为了使用户模式中能在该行出现一次命中，这个比特位必须指出该行包含用户数据。如果该比特位指出的是内核数据，那么就出现一次正常的缺失。这就防止了用户访问被高速缓存的内核数据，它类似于前面讨论过的可写位的概念。在内核模式高速缓存访问期间，就忽略用户-内核位（user-kernel bit），以便内核能够访问高速缓存中的用户数据（例如，这类数据可能是系统调用参数）。注意，当内核把数据复制到用户空间的时候（比如当返回系统调用的结果时），数据就会被标记为内核数据。这将防止用户进程访问它，于是从高速缓存中冲洗掉要复制的地址范围（使主存储器有效，而使高速缓存无效）。即便如此，这一技术通过消除上述的高速缓存冲洗操作而大大减少了冲洗高速缓存的量，但是除非增加额外的支持硬件，否则它仅在采用写直通高速缓存机制的情况下可行。

写回高速缓存策略不能和这一技术单独配合使用，因为在返回用户模式的时候，留在高速缓存内修改过后的内核数据不能在用户模式内的一次高速缓存缺失所导致的行替换期间被写回到主存储器中。写回这样的高速缓存行要求用户进程能够转换内核虚拟地址，而且能向内核的物理页面写入数据。这类功能会破坏系统的安全性，因为用户进程能随心所欲地向任何内核地址写入数据。为了克服这个问题，Intel i860 在从高速缓存的写回操作期间不检查用户-内核访问权限。但是，它要确认从 CPU 对所有高速缓存访问的访问权限，而不是像 Apollo 系统那样不管。结果，用户进程不能访问被高速缓存的内核数据（也不能修改它），但是它们能在行替换期间写回内核数据。这种方法保持了系统的安全性，而且不需要显式的高速缓存冲洗操作来防止用户-内核的歧义。没有这些功能的写回高速缓存必须显式地冲洗数据。

3.4 小 结

对于在高速缓存中命中的引用来说，虚拟高速缓存不需要 MMU 操作，从而提供了高速的高速缓存访问，但是它们要求操作系统频繁地进行冲洗操作。如果在不同的时刻使用相同的虚拟地址来引用不同的物理地址，那么就会在高速缓存中出现歧义现象。当使用多个虚拟地址引用相同的物理位置时就会出现别名现象。操作系统必须防止发生歧义和别名现象，以便让高速缓存的存在对用户程序是透明的，这对于程序的可移植性来说至关重要。为了做到这一点，必须在现场切换、调用 exec 和 exit、sbrk 缩小操作、原始 I/O 操作以及用户模式和内核模式之间过渡的时候冲洗高速缓存。频繁地冲洗大规模的虚拟高速缓存非常耗时，会导致很差的系统性能。除了冲洗操作的开销之外，还有另外一个缺点，即冲洗高速缓存会导致进程的局部引用特性被丢弃。每次在现场切换之后执行进程的时候，还有可能在每次系统调用之后，进程都会失去所有的存储引用，从而迫使为每次引用进行 MMU 转换和访问主存储器。如果采用了独立的指令和数据高速缓存，那么只要上述任何情形要求一次无效操作，就必须使指令和数据高速缓存都无效。

可以对虚拟高速缓存采取两种常用的修正措施，从而减少必须出现的冲洗次数。这两种方法将在后面两章中探讨。

3.5 习 题

3.1 考虑一个直接映射、写回方式的虚拟高速缓存，它有 16 384 行，每一行包含 16 字节数据。散列算法使用 32 位虚拟地址中的“位<17..4>”来选择高速缓存行。“位<31..18>”则保存在每行的标记中。如果在一次缺失操作期间替换了修改过的一行高速缓存，那么如果仅在标记中保存了高 14 位，高速缓存如何确定被替换的数据的 32 位虚拟地址？

3.2 参考 3.2.2 小节中别名的例子，使用相同的高速缓存组织结构和地址空间映射完成下面的习题。增加一个地址映射关系，将虚拟地址 0x92000 映射到物理地址 0x1000 上。假定应用和操作系统都不执行冲洗操作，也没有现场切换。对于每个小题来说，画出一副类似于图 3-6 的小图，显示出所列操作完成之后高速缓存会出现的内容。此外，指出高速缓存的内容是否和主存储器一致。假定高速缓存在每组操作开始之前都是空的，主存储器地址 0x3000 包含 1234，0x1000 包含 5678。使用写直通和写回高速缓存机制。按照通常那样，写回高速缓存机制使用写分配，而写直通不使用。

a. 使用写直通高速缓存机制。应用读取 0x2000，把 9876 写入 0x2000，并且读取 0x4000。

b. 使用写回高速缓存机制。应用读取 0x2000，把 9876 写入 0x2000，并且读取 0x4000。

c. 使用写回高速缓存机制。应用读取 0x2000，把 9876 写入 0x2000，读取 0x92000，再读取 0x4000。

d. 使用写直通高速缓存机制。应用读取 0x4000，把 9876 写入 0x2000，读取 0x4000，再把结果写入 0x2000。

3.3　在 3.2.2 小节中描述的处于别名状态的高速缓存，其中虚拟地址 0x2000 和 0x12000 映射到了相同的地址上，使用写直通或者写回高速缓存机制有关系吗？它能保证进程一定会获得正确的数据吗？解释原因。

3.4　系统调用 exec 所需的处理可能要求几个 I/O 操作，每一个 I/O 操作都要在内核等待 I/O 完成的同时有一次现场切换。描述为了避免在系统调用 exec 期间显式地冲洗高速缓存可以采取什么样的优化措施。

3.5　在两个或者两个以上共享一个地址空间的线程之间会出现别名或者歧义现象吗？

3.6　一个系统使用 64K 直接映射虚拟高速缓存，每行 16 字节。如果下面每一对虚拟地址代表地址空间内同一共享存储段的别名现象，那么哪些虚拟地址对要求内核采取 3.3.6 小节中所介绍的特殊措施来保持一致性？解释原因。

a. 0x1000 和 0x70000

b. 0x1100 和 0x1a100

c. 0x52a40 和 0x53a40

d. 0x8ffe90 和 0xfafe90

e. 0x123450 和 0x1234560

3.7　代之以使用全相联高速缓存，重做上面一题。

3.8　在 3.3.8 小节中，我们介绍了一种在标记中加入用户-内核位来确定数据是在用户模式还是在内核模式读取的技术。如果在用户和内核之间将 32 位虚拟地址空间分开，所有的内核地址都将高端比特位设为置位（1），而所有的用户地址都将其清零，那么提出一种替代的技术，不需要在标记中增加比特位就能解决歧义问题。讨论它是如何工作的。

3.9　假定内核支持一种消息传递（message-passing）机制，一个进程能够将一段任意长度的消息传递给另一个进程。这是在发送消息时通过内核把数据复制到一个内核缓冲区中来实现的。消息被保留在缓冲区中直到另一个进程要求接收该消息为止，此刻它又被复制到目的进程的地址空间中。如果系统使用虚拟高速缓存，那么必须出现什么样的高速缓存冲洗操作？解释原因。

3.10　正如 1.3 节所说明的那样，内核根据需要动态地分配更多的堆栈。如果系统使用虚拟高速缓存，那么当发生这一情况的时候必须出现什么样的高速缓存冲洗操作？解释原因。

3.11　当内核决定把页面从进程的地址空间内交换出去的时候，必须对高速缓存进行什么操作？解释原因。

3.12　参考图 3-3 和图 3-4，假定一个进程在其地址空间内虚拟地址 0x1000 处的一页开始映射到了物理地址 0x5000 上。接着，内核把这一页交换出去。当进程在这页上发生缺失错的时候内核又把它交换回来，这一页加载到物理地址 0x0000 处，内核把进程的映射关系改为指向这个新的位置。这在一个使用虚拟高速缓存的系统中代表一种歧义吗？一定要冲洗吗？

3.13　本章中描述了一些内核必须冲洗一个虚拟高速缓存来保持一致性的情形，高速缓存的大小会对这些情形有影响吗？它是怎样对性能产生影响的？

3.14　大多数有虚拟高速缓存的系统都允许让某些页面标记为不缓存。这就可以防止在一次缺失之后高速缓存数据，从而迫使以后的引用也不会命中。无高速缓存的访问通常在页

表中指定。如果这里也是如此，那么如果主存储器中被标记为可高速缓存的一页被改成了不缓存的，会发生什么样的情况（假定不会发生现场切换）？此刻应该冲洗高速缓存吗？为什么？

3.6 进一步的读物

[1] Chao, C., Mackey, M., and Sears, B., "MACH on Virtually Addressed Cache Architectures," *Proceedings of the USENIX MACH Workshop*, 1990.

[2] Cheng, R., "Virtual Address Cache in UNIX," *Proceedings of the Summer Usenix Conference*, June 1987, pp. 217-24.

[3] Frink, C.R., and Roy, P.J., "The Cache Architecture of the Apollo DN4000," *Proceedings of the Spring COMPCON*, March 1988, pp. 300-2.

[4] Frink, C.R., and Roy, P.J., "A Virtual Cache-Based Workstation Architecture," *Proceedings of the 2nd International Conference on Computer Workstations*, March 1988.

[5] Goodman, J.R., "Coherency for Multiprocessor Virtual Address Caches," *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987, pp. 72-81.

[6] Inouye, J., Konuru, R., Walpole, J., and Sears, B., "The Effects of Virtually Addressed Caches on Virtual Memory Design and Performance," *Operating Systems Review*, Vol. 26, No. 4, October 1992, pp. 14-29.

[7] Mogul, J.C., and Borg, A., "The Effect of Context Switches on Cache Performance," *Computer Architecture News*, Vol. 19, No. 2, April 1991, pp. 75-84.

第4章 带有键的虚拟高速缓存

本章介绍一种对第 3 章里介绍的虚拟高速缓存实现的修正方式。每一个高速缓存行的标记字段（tag field）都扩充包含一个进程键（process key），这个进程键唯一地确定一行高速缓存属于一个特殊的进程。采用进程键的目标是减少必须出现的冲洗操作的次数，并且在现场切换前后获得一个进程的局部引用特性。这种体系结构除了用于指令和数据高速缓存之外，还用在了 MMU 里。本章探讨这种类型的高速缓存的组织结构，及其对 UNIX 内核的影响。

4.1 带有键的虚拟高速缓存的操作

系统设计人员已经找到了一条降低冲洗虚拟高速缓存开销的途径，就是在高速缓存每行的标记中增加一个进程键。在理想情况下，给每个进程都分配一个唯一的键，于是将进程的虚拟地址和键组合起来就形成了一个唯一的标识符，标识一个特殊的进程。使用键的目的是防止在不同的进程中相同的虚拟地址之间出现歧义。这样一来，就不会和使用纯虚拟高速缓存一样要频繁地冲洗高速缓存了。这样的高速缓存的组织结构和上一章中虚拟高速缓存的组织结构类似，只是增加了一个特殊的硬件寄存器来保存当前执行的进程的键，还在每个标记中增加了几个比特位来保存和高速缓存行相关联的键（参见图 4-1）。

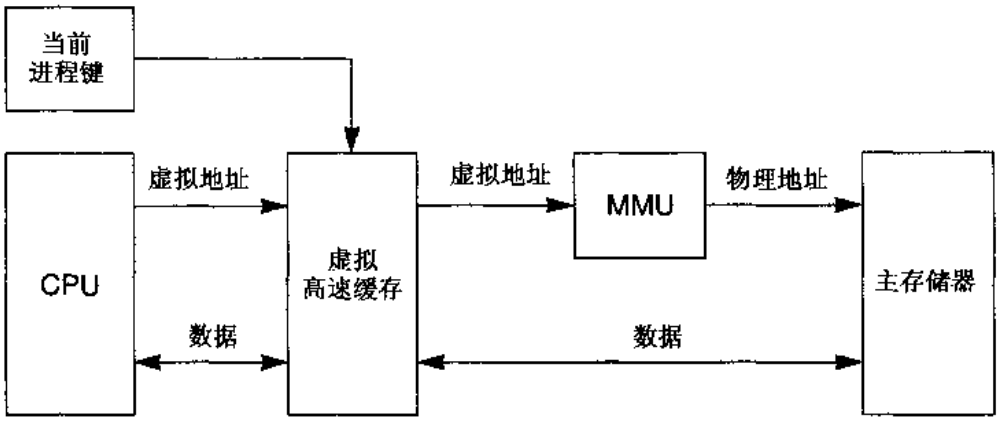


图 4-1 带有键的高速缓存的组织结构

高速缓存仍然以虚拟地址来索引，但是对于出现的命中来说，不但虚拟地址必须吻合，而且当前进程键寄存器（current process key register）也必须和标记中保存的键相吻合才行。当前执行的进程的键由操作系统在现场切换时读入到这个寄存器中（这个寄存器既可以在 CPU 里，也可以在高速缓存控制器上）。这意味着，只要每一个进程有唯一的键，那么使用相同虚拟地址范围的两个不同的进程就不会再有相互引用高速缓存中对方数据的危险。

进程键本身是一个和每个进程相关联的简单整数值。有些实现代之以把键称为任务 id（task id）或者地址空间 id（address space id），但是含义是相同的。不应该把进程键和 UNIX 系统的进程 ID 号（process ID number）或者说 pid 混淆起来，后者是一种独立类型的进程标识符。

进程键本身的实际值并不重要。唯一的需要是它对于每个进程来说是唯一的。这样一来，不同进程所使用的虚拟地址就决不会出现歧义。遗憾的是，可以采用的唯一的键值往往很小，有些实现在标记中只有 3 个比特位来保存键值。如果系统中的进程数比键的数量多，那么多个进程就必须共享键。如果一个以上的进程使用了相同的键，那么高速缓存就不能区分那些进程在高速缓存中的数据，歧义再次出现了。在这些情况下，操作系统不得不冲洗高速缓存来防止出现歧义。这将在 4.2.1 小节中详细讨论。

过去采用这种高速缓存组织结构的系统有 Apollo DN4000 和 Sun 3/200，这两种系统都出现在 20 世纪 80 年代。Apollo 系统使用指令和数据合起来有 8K 的直接映射高速缓存。高速缓存行的大小为 4 字节，它使用采用写分配机制的写直通策略。Sun 系统的高速缓存有 64K，每行 16 字节。它是直接映射高速缓存，指令和数据部分相结合，并且使用写回（带有写分配机制）策略。两种系统的键都使用 3 个比特位，允许 8 个不同的地址空间同时被高速缓存。

4.2 管理带有键的虚拟高速缓存

因为高速缓存仍然采用虚拟地址进行索引，所以仍会出现别名。如果有一个以上的进程使用同一个键，那么也能出现歧义。下面的几小节详细介绍内核必须冲洗高速缓存以防止出现歧义和别名的情形。和采用虚拟高速缓存时一样，只要数据高速缓存需要变得无效，那么使用独立的指令和数据高速缓存的系统也必须使指令高速缓存无效。

4.2.1 现场切换

在正常情况下，如果使用写直通高速缓存机制，那么不需要在现场切换时冲洗高速缓存，因为使用键消除了不同进程中相同虚拟地址之间在高速缓存中的歧义。只要有足够多的键，每一个进程都能分配到一个唯一的键，那么内核只须把硬件中的当前进程键寄存器改为选择执行的新进程的键即可。注意，因为没有冲洗过高速缓存，所以原来进程的局部引用特性就留在了高速缓存中。因此，在下一次选择执行该进程的时候，它的数据有可能还在高速缓存中。这是一种重要的性能收益，因为一个进程可能不会发生和在使用纯虚拟高速缓存的系统上执行时一样多的缺失。如果进程倾向于使用相同的虚拟地址范围，那么就会在一定程度上失去这种好处，因为高速缓存的索引是从虚拟地址中产生的。使用共同虚拟地址空间的进程

会索引到相同的高速缓存行。键能够防止出现歧义，但是前面的进程所读入的高速缓存行仍然会被来自当前执行进程的数据所替换，因而失去了前面进程的局部引用特性。对于小规模的直接映射高速缓存来说尤其如此（试图跨越现场切换而保留局部引用特性的技术将在 7.2.2 小节中介绍）。

当系统中没有足够多的键用于所有的进程时，采用进程键的方法就遇到了困难。这意味着不得不通过把一个键从一个进程重新分配给另一个进程的方法，在多个进程中间共享进程键。在重新分配键的时候，高速缓存中以相关键所标记的所有项都必须被冲洗掉（在采用写回高速缓存的情况下，使主存储器有效，而使高速缓存行无效）。如果没有这样做，那么在以前使用该键的进程的虚拟地址和新进程的虚拟地址之间就会出现歧义。

对于操作系统来说，重要的是在重新分配键的时候做出良好的选择，避免分配的结果落入每次现场切换都要重新分配键的境地。这样的情形会削弱采用进程键的好处，因为冲洗高速缓存的操作会和纯虚拟高速缓存一样频繁。

这种高速缓存组织结构的大多数实现都优先采用写直通策略，就像 Apollo 系统所采用的那样。如果使用了写回高速缓存机制，那么在现场切换到下一个进程的时候，原来进程被修改过的数据还遗留在高速缓存中。如果那些被修改过的高速缓存行中有一行在缺失处理期间被挑选出来进行替换，那么系统就不知道把修改过的行保存到什么地方，因为老进程的地址空间映射关系已经被新进程的替换掉了。高速缓存只包含有老进程数据的虚拟地址，但是在现场切换期间，MMU 已经载入了新进程的地址空间映射关系，所以它不能转换老进程的地址。使用写直通高速缓存机制可以消除这个问题，因为不会在高速缓存中留下修改过的高速缓存行。

为了克服这个问题，Sun 使用的 MMU 能够同时保存和每个进程相关联的映射关系。于是，当有一行需要写回时，高速缓存把虚拟地址和键都传送给 MMU，MMU 使用键来判断如何转换地址。这是一种比较复杂的实现，因为它要求 MMU 一次管理多个地址空间的映射关系，但是它也有优点，即可以使用写回高速缓存机制，从而有助于减少访问主存储器的次数。

如果没有采用这样的硬件，那么其他唯一的选择就是在现场切换的时候用高速缓存中所有修改过的数据使主存储器有效。这就确保了在另一个进程正在运行的时候，不会有前一个进程的数据要写回。

4.2.2　fork

在创建出一个新进程的时候，给它分配一个新的进程键，使其虚拟地址空间不会和父进程或者其他进程的虚拟地址发生歧义，这样的做法比较合适（从理论上说，在某些情况下，父进程和子进程之间有可能会共享相同的进程键，参见习题 4.6 和 4.8）。内核必须确保父进程在 fork 之前所高速缓存的修改过的数据能出现在子进程的地址空间里（如果采用了写回高速缓存机制的话），并且要保证在复制全部或者部分地址空间期间的一致性。

回忆 3.2.2 小节，如果纯虚拟写直通高速缓存没有配合采用写时复制机制，那么就不需要把向子进程复制地址空间期间所使用的内核虚拟地址冲洗掉，因为在子进程运行之前至少会有一次现场切换，从而消除了在复制期间造成的别名现象（参见图 3-8）。因为带有键的虚拟高速缓存不一定要在现场切换时刻进行冲洗，所以除非冲洗了这些项（如果采用写回高速

缓存机制，则使主存储器有效，而使高速缓存无效），否则别名仍然会留在高速缓存中。和以前一样，如果采用了写回高速缓存机制，那么子进程的物理页面会保留过时数据，这些数据对应于那些尚未写回到主存储器的高速缓存行。如果子进程要开始运行了，它会读取这些过时数据，因为它不会在以内核虚拟地址标记的高速缓存行上产生命中。写直通高速缓存机制或者显式地冲洗高速缓存使主存储器有效，都可以消除这个问题。和以前一样，不经过高速缓存地引用复制的目的地就会消除别名现象，也就不需要冲洗高速缓存了。

如果配合写回高速缓存使用了写时复制技术，那么必须按照第 3 章里所描述的那样，在调用 fork 的时刻使主存储器有效。这样做可以防止由于行替换而执行的写回操作触发写时复制缺失错。

看看在父进程和子进程之间出现写时复制共享时高速缓存操作的行为是一件有意思的事情。假定系统中只有两个进程正在运行，它们是父进程和子进程，而且假定它们以写时复制技术至少共享着一页。已经给父进程分配了键 1，子进程使用键 2。高速缓存是直接映射高速缓存，而且使用写直通策略。假定虚拟地址 0x100000 处于一个共享页面之中，高速缓存散列算法将这个地址算到了高速缓存行 0x10 上。假定位于地址 0x100000 的数据是 1011970。如果父进程现在运行并且读取地址 0x100000，那么高速缓存行 0x10 就会载入这个数据，并且用那个地址和键 1 进行标记，如图 4-2 所示（和以前一样，为了清楚起见，标记中给出了完整的地址）。

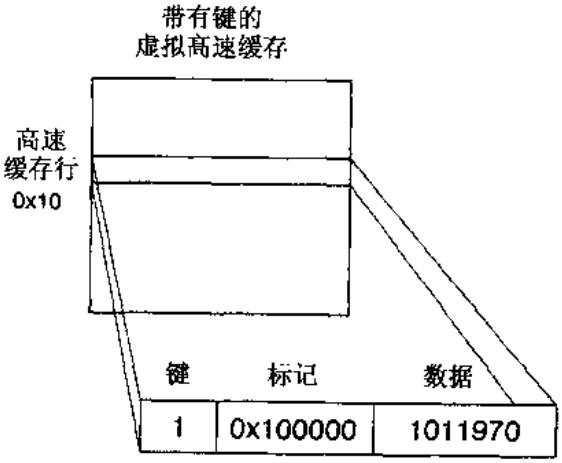


图 4-2 在父进程读取地址 0x100000 后高速缓存的内容

如果现在进行从父进程到子进程的现场切换，那么当前的进程键就会变为 2。如果子进程读取相同的地址，那么即使标记中的地址部分匹配，而且行中的数据部分包含有正确的数据（因为两个进程共享着相同的页面），仍然会发生一次缺失，因为标记高速缓存行的键是错的。这次的缺失致使该高速缓存被替换掉，于是意味着必须再次从地址 0x100000 读取数据，然后用图 4-3 所示的内容替换高速缓存行 0x10。

键	标记	数据
2	0x100000	1011970

图 4-3 在子进程读取地址 0x100000 后高速缓存行 0x10 的内容

注意，除了键值之外（现在是 2 而不是 1），其他一切内容都是一样的。键值不匹配导致高速缓存从主存储器重读数据，而这是不必要的。于是，即便两个进程共享了相同的物理页面，但是由于每个进程以不同的进程键来访问高速缓存，所以两者并没有共享高速缓存中的数据。返回的数据一定是正确的，但是高速缓存却没有办法知道数据应该是被共享的。这就是在高速缓存中使用键的缺点之一。

如果高速缓存中的每一组有两行或者更多行高速缓存，那么这些影响作用就不一样了。在采用这类高速缓存的情况下，父进程和子进程在使用相同虚拟地址的时候一定会索引到同一组上。但是，因为一组里有不止一行高速缓存，所以两个进程可能将相同的数据高速缓存在了不同的行里，而每一行又采用不同的键来进行标记。重新看这个例子，这一次是双路组相联高速缓存，虚拟地址 0x100000 仍然会产生到 0x10 的索引，但是现在一个索引指到了一组两行高速缓存上。在父进程读取 0x100000 之后，该组高速缓存的内容则如图 4-4 所示（假定这一组的第二行现在没有包含数据，在所有列中以“-”指出来）。

	键	标记	数据
组 0x10	1	0x100000	1011970
	-	-	-

图 4-4　在父进程读取地址 0x100000 之后组 0x10 的内容

如果子进程现在要开始运行，并且读取虚拟地址 0x100000，它会像以前那样在高速缓存中发生一次缺失（因为组内第一行的键和子进程的键不吻合）。但是，在采用了双路组相联高速缓存的情况下，替换算法将选择组内的第二行进行替换，因为替换空行总比替换有数据的行好。通过从主存储器读取值并且如图 4-5 所示把值载入高速缓存，缺失处理就完成了。

	键	标记	数据
组 0x10	1	0x100000	1011970
	2	0x100000	1011970

图 4-5　在子进程从地址 0x100000 读取数据后组 0x10 的内容

在采用直接映射高速缓存的情况下，即使共享了物理页面，也不得不重新从主存储器读入数据。虽然这种情形看上去类似于别名，但是它却不会造成任何一致性的问题，因为两个进程都不能访问对方在高速缓存中的数据（因为进程键不匹配）。一旦其中任何一个进程试图修改页面，那么写时复制共享情形就结束了，并且每一个进程都会有自己的一份对该页的副本。

4.2.3　exec

执行系统调用 exec 之后，返回到用户态之前，必须确保高速缓存中任何与老地址空间相对应的高速缓存项都无效。此外必须给新地址空间选择一个唯一的键。最直截了当的实现就

是让 exec 所建立的新地址空间重用相同的进程键。这样一来，进程键就和进程相关而不是与程序相关。因为 exec 丢弃了老进程的地址空间，所以这样做是有意义的。因此，任何以该键标记的高速缓存行都不再需要了。这样做也节省了再找一个没有用过的键所要花的时间。

因为新地址空间使用了相同的键，所以 exec 必须使高速缓存中所有以该键标记的项都无效。这就保证了在新老地址空间之间不会产生歧义。

4.2.4 exit

在采用上一章所介绍的纯虚拟高速缓存时，调用 exit 退出的时候没有必要使高速缓存无效，因为 exit 调用的最后一步会发生现场切换，在此期间一定会冲洗高速缓存。因为在正常情况下，进行现场切换时不会冲洗带有键的虚拟高速缓存，所以在调用 exit 退出的时刻，必须使现有进程地址空间中被高速缓存的数据都无效。为了让未来的新进程可以重用进程键，必须这样做。

另一种方法是推迟冲洗高速缓存，直到给新进程重新分配了进程键为止。这就没有危险会命中已经终止的进程可能留在高速缓存里的任何数据，因为所有其他的进程都使用了不同的进程键。只需要在标记过时高速缓存项的键被另一个进程重用的时候，使这些过时项无效即可。

4.2.5 brk 和 sbrk

在采用带有键的虚拟高速缓存时，处理系统调用 brk 和 sbrk 所需的高速缓存管理技术和采用纯虚拟高速缓存时所需的技术相同（参见 3.3.5 小节）。

4.2.6 共享存储和映射文件

纯虚拟高速缓存依赖于这样的事实，即每次现场切换期间都要冲洗一次高速缓存，从而防止不同进程之间出现别名，这些进程在不同的虚拟地址上使用了相同的共享存储区。对于带有键的虚拟高速缓存来说，情况并非如此。事实上，每个进程都使用了一个不同的进程键，这让情况又进一步复杂了。因此，在带有键的高速缓存内维护共享存储和映射文件的高速缓存一致性要比纯虚拟高速缓存复杂。下面的讨论将首先着眼于直接映射高速缓存，因为它往往有可能避免用显式的冲洗操作来维护采用这种高速缓存组织结构的一致性。随后将考虑每组有两行或者更多行高速缓存的情况（下面的讨论既适用于共享存储，也适用于映射文件）。

如果所有的进程都是把共享存储区附加到各自进程中相同的虚拟地址上，同时采用了直接映射高速缓存的话，那么就不会发生别名现象。这里的效果和 4.2.2 节中第一个例子是一样的（图 4-2 和 4-3），其中的每个进程都把属于其他进程的任何高速缓存数据替换掉了，因为它们都索引到了相同的高速缓存行上。即使使用的是写回高速缓存策略，只要高速缓存采用了写分配机制，也能保持一致性。在这种情况下，如果一个进程留在高速缓存中一个共享页面，而该页面内有一行修改过的高速缓存，那么它将作为另一个进程的缺失处理的一部分而被写回主存储器。要了解这是怎样工作的，可以假定两个进程都将一个共享存储区附加到了

0x100000 上，而且两个进程已经分配得到了进程键 1 和 2。参考图 4-6，假定在高速缓存中有下面一行经过修改的内容（在“修改位”一列中的“M”代表该行的修改位）：

键	修改位	标记	数据
1	M	0x100000	1011970

图 4-6　来自共享存储区的修改数据

如果第二个进程开始运行，而且引用了这个地址，就会发生缺失（因为进程键不匹配）。这一行会被写回到主存储器中，然后再从相同的地址读入来替换原来的一行，从而检索到正确的值，载入高速缓存的数据以当前进程的键来标记（参见图 4-7）。

键	修改位	标记	数据
2	-	0x100000	1011970

图 4-7　在缺失处理之后高速缓存行的内容

如果没有采用写分配机制，那么就有可能出现不一致性（类似于 3.2.2 小节所解释的那些情况）。如果高速缓存行的内容一开始如图 4-6 所示，使用进程键 2 的进程向地址 0x100000 执行了一次保存操作，那么就会发生缺失，并且把数据直接保存到存储器中。以进程键 1 所标记的数据现在是过时的数据，它们不会受到影响，进而使得使用进程键 1 的进程返回时从高速缓存中读取了错误的数据。当使用共享存储的时候，通过在现场切换期间使上存储器有效的操作就能防止出现这种情形。

现在考虑两个进程将同一共享存储区附加到其地址空间内不同虚拟地址时的情况。这样一来，两个进程有可能索引到高速缓存内不同的行，从而导致出现别名问题。这和 3.2.2 小节所介绍的例子类似，不同之处在于现在的虚拟地址来自于两个不同的地址空间，而不仅仅是一个。最终结果则相同，也就是说，每个进程最后会把来自共享存储区的数据缓存到高速缓存内不同的地方，从而导致出现不可预测的结果。

在采用直接映射高速缓存的情况下，有几种替代方案可以防止出现这些别名问题。一种可能是，内核在现场切换期间从高速缓存中冲洗掉共享存储区（使主存储器有效，而使高速缓存无效）。这种方法并不可取，因为即使它的确允许不同的进程将同一共享存储区附加到任意的页面边界（page boundary）上，但是每次现场切换之后，在进程引用共享存储时肯定都会遇到缺失现象。另一种方法是，内核可以选择让共享存储区不被高速缓存。从性能的立足点来看，使用没有高速缓存的访问是最不可取的方法，因为所有的引用都会出现缺失。幸运的是，在许多情况下还有另一种方法表现不错。

如果采用直接映射高速缓存机制，那么内核会选择受限地址技术（这类似于 3.3.6 小节所介绍的技术，当时在一个进程地址空间内有两个别名）。这意味着限制共享存储区所附加的地址，以使让共享同一区域的所有进程在引用相同的数据时都会索引到相同的高速缓存行上。就高速缓存而言，这种效果就好像是所有的进程都附加在相同的地址上一样。Sun 和 Apollo 系统都使用了这种方法。因为前面例子中所说明的理由，这种方法只能在采用了写分配机制的高速缓存上起作用。

为了举例说明它，图 4-8 给出了两个不同进程的逻辑图，它们在不同的虚拟地址上共享一个共同的物理页面，该图还显示出它们怎样映射到高速缓存上。高速缓存是直接映射高速缓存，每行 16 字节，共 1024 行。虚拟地址的“位<13..4>”用于索引高速缓存。进程 A 将共享页面映射到它的地址空间内的地址 0x10300 上，而进程 B 将它映射到地址 0x40300。这两个地址经散列处理之后都指向高速缓存中相同的行索引号，即 0x030。图中的阴影区域代表共享页面开头的 16 字节。即使这两个进程使用了不同的虚拟地址和不同的进程键，它们引用共享页面时也都会使用相同的高速缓存行。

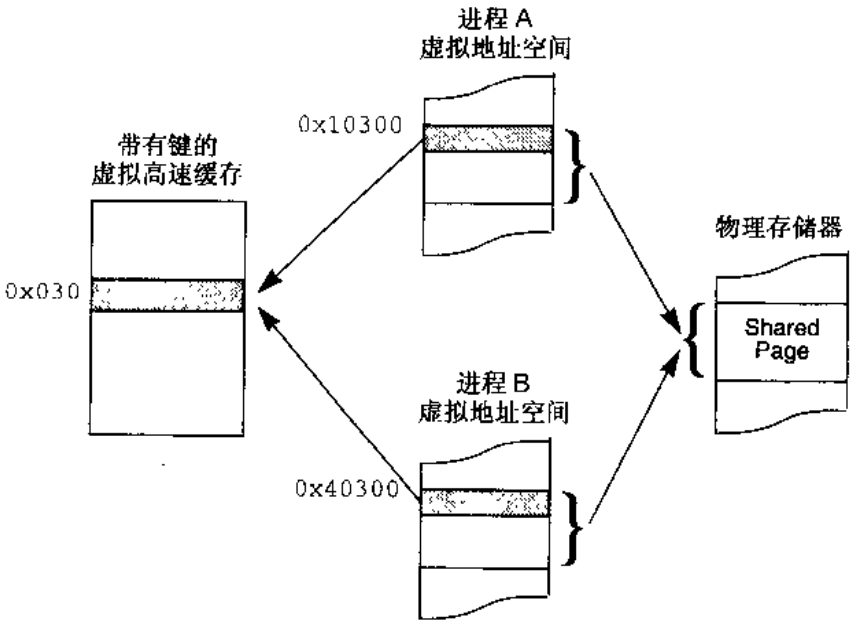


图 4-8

因为每个进程有它自己的键，所以它不会在另一个进程高速缓存的共享数据上产生命中，而是把这一行替换掉，就好像它是被显式地冲洗掉一样。如果前一个进程修改了高速缓存行，那么它就会被当前的进程写回到主存储器，然后再重新读入高速缓存，但此次是以当前进程的键来标记的。这就方便地消除了显式冲洗高速缓存的需要，但是再一次体现出了在高速缓存中使用键的主要缺点之一：共享的数据在高速缓存中不能共享；每次它驻留在高速缓存中，但以不正确的键标记的时候，必须要从主存储器重新读取它。共享页面内所有的高速缓存行都是如此。注意，这仍然比使用没有高速缓存的访问效果好，后者在引用共享存储区时总是要求访问主存储器。如前所述，这项技术只适用于采用写分配机制的高速缓存。

如果高速缓存使用了一种取模散列算法（在 2.3.1 小节中定义），那么内核必须在共享一段存储区的进程上实施唯一的一条规则，即在所有进程中共享存储区起始地址之间的差别必须都是高速缓存大小的整数倍。这就能让每个进程所使用的地址都索引到相同的高速缓存行上，而且能替换掉其他进程所高速缓存的任何数据项。使用受限的地址可以防止来自共享区的数据被载入到高速缓存中一个以上的地方，而且可以保证防止出现别名问题。

这些技术只对直接映射高速缓存有效。如果高速缓存的每一组有两行或者更多行，那么即使两个或者两个以上的进程使用了相同的虚拟地址，它也能同时高速缓存来自这些进程的

共享数据。和图 4-5 所示的例子不同，因为任何一行都可以独立地进行修改，从而有可能导致出现不一致的数据，所以这样做会产生别名。为了防止出现别名，内核必须在现场切换的时候冲洗高速缓存（使主存储器有效，而使高速缓存行无效），或者干脆不高速缓存数据。这就再次暴露出数据在高速缓存中的时候不能共享的缺点。

4.2.7　输入输出

在采用带有键的虚拟高速缓存时，为了确保相关 I/O 操作的一致性，内核所必须采用的技术和采用纯虚拟高速缓存时所需的技术相同（参见 3.3.7 小节）。

4.2.8　用户–内核数据的歧义

正如在 3.3.8 小节中讨论过的那样，Apollo 系统在标记中使用一个模式位（mode bit）来区分被高速缓存的用户数据和内核数据。这能让内核消除在其他情况下为了防止用户访问高速缓存中的内核数据而不得不进行的高速缓存冲洗操作。Sun 采用了一种类似的方法，它在内核态运行的时候干脆就使用唯一的键（例如，给内核保留键 0）。只要除了内核之外没有进程使用这个键，就不会让用户进程访问到内核在高速缓存中的所有数据。为了实现这一点，就要在进入内核态时改变当前进程键寄存器，在退出内核态时恢复到用户的进程键。

这种方法的一个明显的优点是，正在内核中执行的所有进程都可以访问被高速缓存的内核数据。因为在 Sun 系统中，逻辑上只有一个内核和一个内核地址空间，所以内核数据一定是在所有的进程中共享的。通过让所有以内核态执行的进程有相同的进程键，在一个进程的现场所发出的内核引用就能够命中内核在另一个进程中执行时高速缓存的数据。而在 Apollo 系统上，这样的情况会出现缺失。

为内核使用独立的键值有一个缺点：在内核把数据复制到用户地址空间或者从用户地址空间复制到内核的时候，不会命中被高速缓存的用户数据。这和在用户共享存储上遇到的问题一样（参见 4.2.6 小节），可以采用相同的方式来处理：如果使用直接映射高速缓存，那么不需要执行冲洗操作；如果使用双路或者更多路组相联高速缓存，或者高速缓存没有写分配机制的话，那么就需要冲洗。注意，虽然 Apollo 使用一个用户/内核模式位的方法会让复制到用户地址空间的数据标记有正确的进程键，原因是内核是以和用户相同的键运行的，但是，当用户进程访问那些数据的时候仍然会出现缺失。之所以会发生这样的情况，是因为那些数据被标记成了内核数据。因为是写直通高速缓存，而且使用了写分配机制，所以此刻会自动返回正确的数据。

4.3　在 MMU 中使用虚拟高速缓存

高速缓存不仅限于保持程序指令和数据。在几乎所有的 MMU 中都集成了一种特殊用途的高速缓存，它称为转换后援缓冲器（Translation Lookaside Buffer，TLB）或者地址转换高速缓存（Address Translation Cache，ATC）。它通过高速缓存最近使用过的页面映射关系（page

mapping），也就是对应于进程工作集的那些页面，来加速因局部引用特性而带来的“虚拟到物理”的地址转换进程。TLB 本身仅仅是一个虚拟高速缓存而已，通过使用虚拟高速缓存可以找到转换所用的入口地址，它也同样以这个地址来进行索引和标记。TLB 所高速缓存的“数据”是物理页号（physical page number）和页访问权限（page access permission）。TLB 内发生一次命中的时候，MMU 能够从 TLB 的数据立即计算出物理地址和有效的权限。在出现一次缺失的时候，有些实现会读取保存在主存储器内的页表，以获得相关的映射关系，然后将新的映射关系自动高速缓存在 TLB 中供以后使用。传统的处理器体系结构（比如 Motorola 68040 和 Intel 80X86 系列）就采用了这种方法。有些 RISC 体系结构也用到了它，比如 Motorola 88000 和德州仪器公司的 TI SPARC 系列。其他 RISC 实现在 TLB 发生缺失的时候产生了一个操作系统的陷阱。随后，操作系统必须判断正确的映射关系，检查页面的权限，并且将适当的“虚拟到物理”转换信息读入 TLB。MIPS 系列处理器就采用了这种方法。

如果只采用虚拟地址来标记 TLB 的数据，那么在每次现场切换的时候都必须把它冲洗掉。这样做可以防止在新老进程的映射关系之间发生歧义。68040、88000 和 80X86 中的 TLB 是以这样的方式运行的。但是，MIPS 系列和 TI SPARC 系列处理器的 TLB 使用了一块带有键的虚拟高速缓存。就像本章所讨论的数据和指令高速缓存那样，给每个进程分配一个唯一的 TLB 键，也能让 TLB 同时高速缓存来自多个进程的数据项。这样做可以不必在现场切换时冲洗 TLB。例如，MIPS R4000 就带有一个有 48 项的全相联 TLB。其中的每一项包括一个 8 位的地址空间标识符（键）。带有键的 TLB 只需要在地址转换时进行冲洗就可以了，比如在系统调用 sbrk 缩小了 bss 段的时候，或者在另一个进程要重用键的时候。

进程键同样适用于 TLB。例如，TLB 所高速缓存的项和数据高速缓存不同，它们是只读的，不能共享。这就避免了本章前面看到的与共享存储和写时复制页面相关的不一致问题。带有键的 TLB 跨越现场切换，保留了进程的转换关系，这就使得当从操作系统中几乎得不到额外的维护（冲洗操作）时有可能降低 TLB 的缺失率（不命中的比率）。

4.4 小　结

带有键的虚拟高速缓存尝试减少纯虚拟高速缓存上的冲洗开销，与此同时，它仍然保留了无需 MMU 操作就能访问高速缓存的好处。因为在正常情况下，不会在现场切换的时候冲洗高速缓存，所以进程的局部引用特性就有机会一直在高速缓存中保存到下一次选择运行该进程的时候。采用纯虚拟高速缓存是不可能有这样的性能优势的。

虽然需要的冲洗操作少了，特别是在现场切换时的冲洗操作少了，但是它仍然还有明显的缺点，即不能共享高速缓存内的共享数据（无论是 fork 期间的写时复制共享，还是共享存储，抑或是映射文件），因为每一个进程使用的进程键都不同。通过限制共享存储的附接地址，只要每个进程都索引到了相同的高速缓存行，而且又采用了带有写分配机制的直接映射高速缓存，那么就不需要显式地冲洗高速缓存。但是要注意，其效果就仿佛是执行了一次显式的冲洗操作一样：发生一次缺失，然后替换高速缓存行。如果一组有两行以上的高速缓存，那么就需要显式地使主存储器有效，从而防止使用过时的数据。带有键的虚拟高速缓存没有

消除在 exec、exit、brk、sbrk 和 I/O 期间所需要的冲洗操作。和使用纯虚拟高速缓存时一样，这些活动都需要冲洗高速缓存。

如果没有足够多的键提供给系统中所有的进程，那么会给性能造成额外的损失，因为必须在重新分配键的时候冲洗高速缓存。在使用独立的指令和数据高速缓存的系统上，只要是为了维护一致性而使高速缓存无效，就必须同时让指令和数据部分都无效。通过代之以物理地址标记高速缓存行，就可以减少需要进行冲洗的次数，这将在下一章里进行讨论。

4.5 习　题

4.1　一个系统使用 32 位地址。它有一个带有键的虚拟高速缓存，其中有 4 个比特位用于进程键，而且是双路组相联高速缓存。这个高速缓存包含 1024 行，每行 16 字节数据。标记的地址部分需要多少比特位？如果用于键的比特位数增加到 6，那么标记的地址部分需要多少比特位？为什么？

4.2　在 2.8 节中曾经说过，高速缓存的实现可以按行或者按地址来冲洗高速缓存。对于带有键的虚拟高速缓存来说，其他什么类型的冲洗功能会有所帮助？

4.3　改变一个进程的键，如果该进程原来的键没有被系统中的任何其他进程再次使用，则先不冲洗高速缓存，阐述这样做的效果。假定在改变进程键之前，该进程已经高速缓存了数据，而且在高速缓存中没有以新键标记的数据。系统中任何别的程序会得到不正确的数据吗？改变了键的进程会得到不正确的数据吗？考虑写直通和写回这两种高速缓存策略。

4.4　如果面临进程数比键数多的情况，那么在重新分配键上采用 LRU 算法是一种好的选择吗？解释原因。要说明你所做的任何假设。

4.5　在 4.2.1 小节里说过，当使用小规模的直接映射高速缓存的时候，最有可能在跨越现场切换时丢失进程的局部引用特性。为什么对于大规模的全相联高速缓存来说不是这样？

4.6　如果采用写时复制实现了 fork 调用，父进程和子进程有可能共享相同的键吗？这样做会给进程造成什么样的影响（包括性能上的影响）？它们能共享键多长时间？解释原因。

4.7　在图 4-2 到 4-5 所示的例子中，高速缓存是否采用写分配机制有关系吗？考虑直接映射高速缓存和双路组相联高速缓存的情况。

4.8　有些 UNIX 系统的实现支持系统调用 vfork。和 fork 一样，它也创建一个子进程，不同之处在于子进程能够直接共享父进程的地址空间，而且既不需要复制父进程的地址空间也不需要写时复制技术。这意味着父进程能看到子进程对地址空间所做的任何修改。进一步定义的是 vfork 要挂起父进程，直到子进程调用 exit 或者 exec 时为止。在使用这个系统调用的时候，内核应该如何管理带有键的高速缓存？当子进程调用 exit 或者 exec 时需要发生什么事情？

4.9　如果内核在执行的同时使用了一个独立的键，那么若某个用户进程有已被高速缓存的数据，内核一定能从正在执行一次系统调用的这个进程中读取到正确的数据吗？对比考虑写直通和写回策略的效果，以及高速缓存组大小不同时的情况。

4.10　如果两个不同的进程使用了一组相同的虚拟地址，那么带有键的虚拟高速缓存就能解决歧义问题，但是这些地址仍然产生了相同的索引。如果采用直接映射高速缓存机制，

这就会导致一个进程的数据替换掉另一个进程的数据，从而降低另一个进程再次运行时的命中率。削弱这种高速缓存颠簸现象的一种办法是把键用作散列算法的一部分。例如，如果在产生索引的时候将键与虚拟地址进行异或，那么在不同地址空间的同一虚拟地址就会索引到高速缓存中的不同高速缓存行（假定它们使用不同的键）。讨论这一方面的优缺点。将它和使用传统散列算法、但每组有两行或者更多行的高速缓存相比，有什么优缺点？

4.11 如果在一个使用了带有键的 TLB 的系统上，某个进程执行了一次 fork 调用，说明内核应该在 TLB 上有什么操作。

4.12 重做习题 4.8，考虑带有键的 TLB 而非指令和数据高速缓存所带来的效果。

4.13 考虑一个系统，它有大量的键可供高速缓存使用（比如说，键有 10 比特位以上）。如果这对本章所介绍的算法和技术有影响的话，会是什么样的影响？

4.14 共享一个地址空间的线程应该使用相同的键吗？

4.15 考虑一个具有独立的指令和数据高速缓存的系统。指令和数据高速缓存都是带有键的虚拟高速缓存。一种实现是有两个独立的当前进程键寄存器，每个高速缓存一个，讨论其优缺点。操作系统应该利用这种功能吗？

4.6 进一步的读物

[1] Chao, C., Mackey, M., and Sears, B., "MACH on Virtually Addressed Cache Architectures," *Proceedings of the USENIX MACH Workshop*, 1990.

[2] Cheng, R., "Virtual Address Cache in UNIX," *Proceedings of the Summer USENIX Conference*, June 1987, pp. 217-24.

[3] Frink, C.R., and Roy, P.J., "The cache Architecture of the Apollo DN4000," *Proceedings of the Spring COMPCON*, March 1988, pp. 300-2.

[4] Frink, C.R., and Roy, P.J., "A Virtual Cache-Based Workstation Architecture," *Proceedings of the 2nd International Conference on Computer Workstations*, March 1988.

[5] Goodman, J.R., "Coherency for Multiprocessor Virtual Address Caches," *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987, pp. 72-81.

[6] Inouye, J., Konuru, R., Walpole, J., and Sears, B., "The Effects of Virtually Addressed Caches on Virtual Memory Design and Performance," *Operating Systems Review*, Vol. 26, No. 4, October 1992, pp. 14-29.

第5章 带有物理地址标记的虚拟高速缓存

本章介绍对虚拟高速缓存的最后一种增强方式，它不但可以消除歧义，而且还有可能共享高速缓存中的数据。通过从标记中去除虚拟地址这一造成问题的根源，而代之以采用数据的物理地址，就能做到这一点。不过，采用了这种方法以后，高速缓存查找操作就和 MMU 的地址转换有关了，因为需要用物理地址来判断是否命中。下面几节将阐述这种高速缓存组织结构的影响。

5.1 带有物理标记的虚拟高速缓存的组成

带有物理地址标记的虚拟高速缓存建立索引的方式和纯虚拟高速缓存相同。但是，高速缓存行是以数据的物理地址来标记的。标记中没有用虚拟地址，只在建立索引期间才用虚拟地址。在采用这种高速缓存组织结构的情况下，每次访问的时候都需要转换虚拟地址，因为需要用物理地址来判断寻址的数据是否在高速缓存中。图 5-1 给出了这样的一个系统的逻辑组织结构。

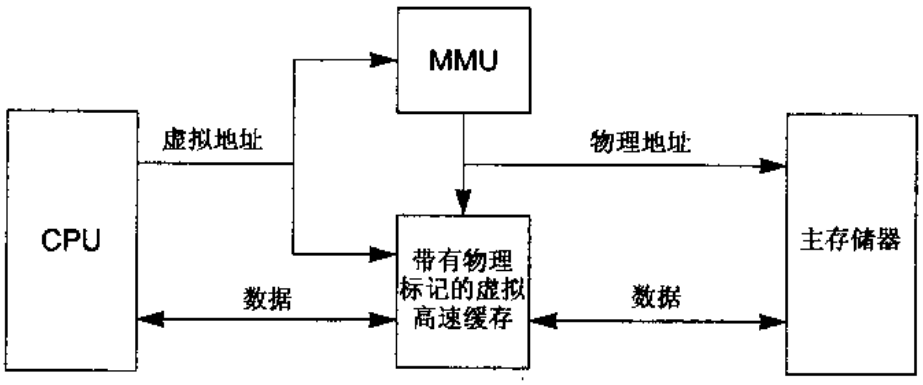


图 5-1 带有物理标记的虚拟高速缓存的组织结构

在这种类型的系统中，从 CPU 来的虚拟地址被同时发送到高速缓存和 MMU。这样一来，地址转换和存取高速缓存在时间上就交叠在一起。在 MMU 转换地址的同时，高速缓存也通

过散列算法从虚拟地址得到索引来开始它的查找操作。接着，高速缓存能够读取被索引到的高速缓存行的内容，或者为比较标记做好准备。MMU一旦转换好了地址，就把物理地址发送给高速缓存。随后，高速缓存把这个地址和选出行中标记里保存的地址进行比较，以判断是否发生了命中或者缺失。

这种组织结构的好处是，在不同进程中未经高速缓存的数据之间一定不会出现歧义问题，因为每个进程的虚拟地址空间都转换成了一个明确有别于其他的物理地址，可以唯一地确定数据。在这个意义上说，物理地址标记起到了上一章中进程键的作用。不再会发生因为键不匹配，导致若干进程共享的数据无法命中，从而迫使从高速缓存中冲洗掉进程数据行的情况。在转换其虚拟地址的时候，共享数据的进程将产生相同的物理地址，所以当它们索引共享数据的时候会发生命中。这将在以后详细说明。

这种高速缓存设计的缺点是高速缓存查找操作的速度受限于MMU所需的转换时间。正如将要看到的那样，这种设计吸引人的地方是它降低了内核显式冲洗高速缓存的要求，所以是在硬件开销和软件开销之间相对比的一种折衷。

因为每次高速缓存操作都要用到物理地址，所以应该说明它和前两章里虚拟标记的高速缓存之间有区别的两个地方。首先，考虑一下采用写回高速缓存的情况。

当替换被修改过的高速缓存时，不再需要转换地址了，因为在高速缓存的标记中保留着物理地址。高速缓存直接把这个物理地址和高速缓存行的内容发送给主存储器，完全绕过了MMU。和以前一样，假定如果进程有权修改高速缓存中的数据，那么它必须有权把数据写回到主存储器，所以不需要MMU再次检查页面访问权限。这就很方便地克服了在带有键的虚拟高速缓存中采用写回高速缓存机制时的问题。

还要说明的是，在转换地址期间，MMU也有一次检查页面访问权限的机会。因此，带有物理标记的虚拟高速缓存不需要像其他类型的高速缓存中那样，保存比如说冗余的写权限位这样的内容。

和所有的高速缓存一样，设计人员也利用了这样的事实：即没有必要在标记中保存完整的地址，而只是保存那些无法通过数据在高速缓存中的位置推算出来的比特位。虚拟地址中低端的一些比特位，包含有在虚拟页面（virtual page）中的页面偏移量（page offset），它们和物理地址中的是一样的。这些低端的比特位不必保存在高速缓存的标记中。

例如，考虑有512行、每行16字节的一块直接映射高速缓存。假定页面的大小为2KB，地址是32位的。这意味着页面偏移量（page offset）有11位，而虚拟页号（virtual page number，VPN）有21位。在高速缓存查找操作期间，这些地址位的解析过程如图5-2所示。

在执行查找操作之后，“位<3..0>”将选择出高速缓存行内的字节。后9位，即“位<12..4>”被发送给高速缓存，用来选择512行高速缓存中要读取的一行。与此同时，虚拟页号“位<31..11>”并行地发送给MMU，以转换为物理页号（physical page number，PPN）。当完成转换时，MMU输出物理地址的“位<31..11>”。如果需要访问主存储器，填补高速缓存的缺失，那么这些位就和页面偏移量“位<10..0>”拼起来形成完整的物理地址。到MMU转换完地址的时候，高速缓存已经读出了被索引的行，并且把标记的内容发送给比较器，以查看是否发生了命中。由MMU转换出的物理页号和保存在标记中的PPN要进行一次比较。如果它们相符就发生一次命中。

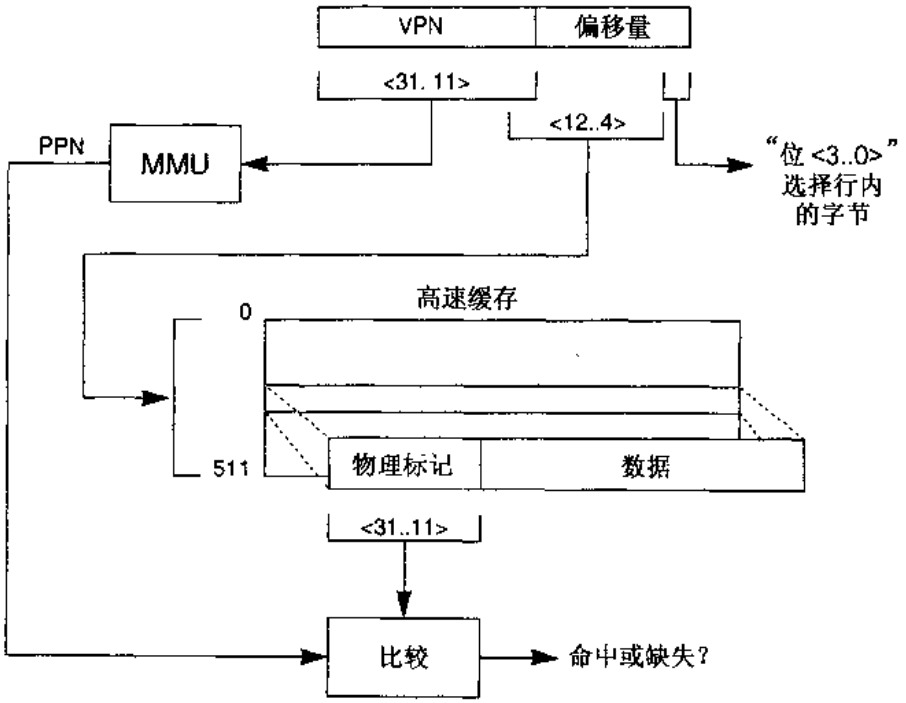


图 5-2　在高速缓存查找操作期间地址的解析过程

注意，必须在标记中保存完整的 PPN。这和前面章节介绍的虚拟标记的高速缓存不一样。虚拟标记的高速缓存只需在标记中保存虚拟地址的“位<31..13>”，因为“位<12..0>”用于索引高速缓存行和选择行内的字节。这些高速缓存都是从同一个实体——虚拟地址——获得标记和索引的。保存标记中 VPN 的低两位（即“位<12..11>”）是一种冗余，因为这些值可以从行号推算出来。但是，在物理标记的高速缓存中，在 VPN 和 PPN 之间没有预先确定的对应关系。PPN 的“位<12..11>”与 VPN 的“位<12..11>”没有关系。因此必须用完整的 PPN 来标记高速缓存行。

如果写回操作需要的话，随后会重组一行内数据的物理地址，方式如下。物理地址的“位<31..11>”保存在标记中。物理地址的“位<10..4>”可以通过取得行索引的低 7 位，从行在高速缓存内的位置推算出来。最后，可以设置“位<3..0>”选择所需的一个特定字节或者字。

MIPS R4000 的片上高速缓存就是这种组织结构的一个例子。它有独立的指令和数据高速缓存，每一个都有 8KB 大，并且是直接映射高速缓存。它的数据高速缓存采用了有写分配机制的写回策略。

对于有些设计来说，使用带有物理标记的虚拟高速缓存和使用物理高速缓存之间有微妙的差别。正如我们将要在下一章看到的那样，物理高速缓存只从物理地址产生索引，从而暗示在高速缓存查找操作能够继续进行之前必须进行 MMU 转换。但是，我们看到的结果却是，如果仅凭页面偏移量内的比特位就能获得高速缓存索引的话，那么就可以立即开始查找操作，因为对于虚拟和物理地址来说，偏移量都是一样的。除了纯物理高速缓存提供的软件性能收益之外，这样的高速缓存也能获得采用物理标记的虚拟高速缓存所获得的性能收益，后者的 MMU 转换和高速缓存查找操作在时间上是交叠进行的。这些高速缓存都应该被认为是纯物

理高速缓存，我们将在下一章介绍它们。MMU 转换和高速缓存查找操作并行执行可以看作是对软件透明的一种硬件优化措施。

注意，MIPS R4000 支持的页面大小可以从 4KB 到 16MB 进行选择（以 4 的倍数）。当使用的页面大小为 4KB 时，片上高速缓存就充当了带有物理标记的虚拟高速缓存的角色，因为需要虚拟页号的一些比特位来索引一个 8KB 直接映射高速缓存。但是，当使用的页面大小为 8KB 或者更大的时候，高速缓存就充当了一个纯物理高速缓存。

最后要说明一点，既有虚拟也有物理标记的高速缓存结构（比如 Intel i860 XP）都不是本章所描述的高速缓存类型。在 i860 上，单独使用虚拟标记，来自 CPU 的访问会进行高速缓存查找。从软件的角度来看，这就让高速缓存充当了纯虚拟高速缓存。为了维护与 DMA 操作有关的高速缓存一致性而采用的物理标记将在下一章里讨论。

5.2 管理带有物理标记的虚拟高速缓存

使用物理标记大大减少了需要内核显式冲洗高速缓存的情况。我们很快将看到，此外还有能共享高速缓存数据的好处。和前面一样，在下面描述的情况下采用独立的指令和数据高速缓存要求使它们分别无效。

5.2.1 现场切换

没有共享存储的无关进程给它们的数据使用不同的物理页面。因此，在正常情况下，没有必要在现场切换期间冲洗高速缓存，因为物理标记能防止不同进程的数据之间发生歧义。这是这类高速缓存体系结构的主要优点之一。第二个优点是没有像在上一章里讨论过的高速缓存体系结构那样需要管理的进程键。

注意，当使用写回高速缓存机制的时候，当前进程运行的同时，在高速缓存中会出现来自其他进程的修改数据。如果当前进程在缺失处理期间使得另一个进程修改过的高速缓存行被替换掉了，那么这一行就必须被写回到那个进程的地址空间中。因为高速缓存不需要使用 MMU 就能产生物理地址，所以这不会造成问题。它可以把数据直接写入到和修改过它的进程的地址空间相对应的物理页面中。这并不代表破坏了安全性，因为当前进程不能改变或者访问数据。它只能在发生命中的时候才可以这样做，但因为标记不会匹配，所以永远不会发生这样的情况（假定没有使用共享存储）。

当使用共享存储的时候，必须在现场切换时冲洗高速缓存。这将在 5.2.6 小节介绍。

5.2.2 fork

不需要为了建立写时复制共享机制而在调用 fork 的时候冲洗高速缓存。和采用带有键的虚拟高速缓存情况不同，即使使用了写回高速缓存机制（就像前面讲述过的一样），也不需要这样做。此外，还有一个优点，那就是可以共享高速缓存本身里的数据，这在其他虚拟高速缓存体系结构中都是不可能的。

虽然父进程和子进程通过写时复制技术来共享页面，但是两者都使用着相同的虚拟地址。这些虚拟地址都会被映射到相同的物理页面上。因为虚拟地址相同，所以当使用一个给定的虚拟地址时，两个进程都会索引相同的高速缓存行或者组。只要数据驻留在高速缓存中，就会继续在两个进程之间共享。这和 4.2.2 小节中带有键的虚拟高速缓存相比，效果相差很大，在后者的情况下，因为数据的进程键不同，所以每个进程访问数据都会发生缺失，被迫从主存储器重新读取数据。结果，同使用键的虚拟高速缓存相比，在使用物理标记的虚拟高速缓存时，预计发生缺失的情形要少。

无需进行冲洗操作，也可以使用写回高速缓存机制和双路或者更多路组相联高速缓存。如果来自父进程的修改数据在调用 fork 之前就出现在了高速缓存中，那么当该数据以写时复制机制进行共享的时候，子进程对该数据的任何引用都会在高速缓存中命中，从而返回正确的数据。这同样也是因为两者都使用了相同的虚拟和物理地址的缘故。此外，将修改过的数据写回到以写时复制机制共享的页面内也不成问题，就像采用了前面几种虚拟高速缓存结构的情况一样。在那些高速缓存结构中，因为页面是只读共享的，所以写回操作会产生一个保护性的陷阱。这将错误地导致发生写时复制处理。在采用物理标记的虚拟高速缓存的情况下，执行写回操作用不着 MMU，因为高速缓存能够产生出物理地址。这就允许它将修改过的数据透明地写回共享页面。在这种情况下，随后出现缺失时，父进程和子进程都能读取到正确的数据。

在父进程或者子进程试图修改某页面而要复制该页面的一个副本时要格外小心。如前所述，在写时复制页面被共享的同时，来自该页面的修改数据可以出现在写回高速缓存中。必须保证在写时复制结束的时候，两个进程都获得数据修改过的版本，而不是来自主存储器的过时数据。应该使用和 3.3.2 小节中所介绍的一样的技术，即为复制操作的目的地建立一个临时的映射。如果复制操作的目的地的内核虚拟地址在引用时会被高速缓存，那么必须使存储器有效，使高速缓存无效，从而消除别名问题。使用不被高速缓存的引用可以避免这次冲洗操作。

一旦进行复制，那么就不需要冲洗操作了，因为此刻进程使用了不同的物理地址。物理标记意味着即便进程使用了相同的虚拟地址，也不会出现别名，更不需要冲洗操作。

5.2.3　exec

在 exec 期间释放老的地址空间时，过时的数据项会遗留在高速缓存中。操作系统可以在系统调用期间使之无效，也可以让它们留在高速缓存中，直到它们所关联的物理地址被释放为止。因为任何进程都没有使用给它们作标记的物理地址，所以一旦 exec 释放了页面，那么就没有另一个进程会命中过时数据的危险。但必须保证在把页面分配给新的进程之前要删除过时数据。这项技术将在 7.4 节中详细介绍。

5.2.4　exit

和 exec 的情况一样，在调用 exit 期间被释放的地址空间既可以在执行系统调用时被冲洗掉，也可以推迟到重用物理页面的时候，这都不会有在高速缓存中的过时数据上出现歧义的危险。这类似于进程键的效果，即直到键被重用之前都不会出现歧义（参见 4.2.4 小节）。

5.2.5 brk 和 sbrk

既然缩小 bss 段的系统调用 brk 和 sbrk 也会释放部分地址空间，所以它们所需的高速缓存管理与 exec 和 exit 的类似。高速缓存不是在执行系统调用时被冲洗掉，就是推迟到重用物理页面的时候。注意，在采用这种高速缓存组织结构的情况下，每次引用都会用到 MMU，所以能防止进程访问任何已释放区域内的数据（参见习题 5.9）。

在调用 brk 或者 sbrk 增大 bss 段的时候不需要冲洗操作（即使如果在以前释放期间推迟了冲洗操作，那么在分配页面期间可能需要进行一些冲洗操作；参见 7.4 节）。

5.2.6 共享存储和映射文件

当使用物理标记的时候，共享存储的高速缓存一致性维护起来要比其他虚拟高速缓存类型容易得多。和调用 fork 的情况一样，如果所有的进程以相同的虚拟地址共享存储器，那么它们就会索引到高速缓存中相同的行或者组。因为共享了存储器，所以物理地址也会吻合，从而在数据上发生命中。在这种情况下，不需要冲洗操作，数据在高速缓存中的时候也能共享。对于写直通和写回高速缓存策略来说，无论组的大小如何都是如此。

当在高速缓存上使用取模散列算法的时候，进程也可以在不同的虚拟地址上共享存储，而无需在虚拟地址同色的时候冲洗高速缓存。只要不同的虚拟地址都能索引到相同的高速缓存行或者组，那么共用的物理地址就会产生命中，并且防止出现别名（参见 4.2.6 小节）。

如果进程试图在不同的虚拟地址上共享存储，而这些地址没有索引到相同的行或者组，那么操作系统就不得不在这类进程现场切换的时候冲洗高速缓存（使主存储器有效，而使高速缓存无效），或者干脆让共享存储不被高速缓存。作为另一种选择，操作系统当然可以禁止将共享存储附加到不能产生相同索引的虚拟地址上。对于一个进程在其地址空间内两次或者多次附加同一个共享存储区来说，情况则与前面章节中描述的相同（参见 3.3.6 小节）。

有一点应该清楚，那就是对于共享存储来说，带有物理标记的虚拟高速缓存所达到的命中率比纯虚拟高速缓存或者带有键的虚拟高速缓存都要高。不仅留在高速缓存中的数据可以跨越现场切换，而且所有共享数据的进程都能共享被高速缓存的数据。

5.2.7 输入输出

在使用物理标记时，为了确保相关 I/O 操作的一致性，内核所必须采用的技术和使用纯虚拟高速缓存时所需的技术相同（参见 3.3.7 小节）。

5.2.8 用户-内核数据的歧义

使用物理标记消除用户和内核数据之间的任何歧义。内核和用户数据始终驻留在独立的物理存储页面内，因此当它驻留在高速缓存中的时候一定具有不同的标记。即使内核要使用虚拟地址来引用它自己的数据，该数据匹配了高速缓存中用户数据的一行，也会因为物理标记仍然是不同的，所以不会造成歧义。

当使用带有物理标记的虚拟高速缓存时，按照前面章节所描述的那样，把地址空间在用户和内核之间分开是有好处的。采用这种方法以后，运行在内核态的任何进程都能访问被高速缓存的内核，但是用户态的进程却不能访问，因为 MMU 会阻止这种访问。内核也能直接引用被高速缓存的用户数据。这种情况要比带有键的虚拟高速缓存好。例如，当给内核分配一个唯一的键时，对用户数据的访问始终不会命中，从而要求在某些场合显式地冲洗高速缓存（参见 4.2.8 小节）。采用将地址空间分开的方法以后，当高速缓存使用物理标记的时候就没有必要进行冲洗了。

5.3 小　结

从操作系统所需的管理量方面来看，带有物理标记的虚拟高速缓存对比前两种类型的高速缓存有了巨大的改进。物理页面消除了几乎所有情况下的歧义问题，这意味着需要进行的冲洗操作少得多了。使用物理标记不但保留了带有键的虚拟高速缓存所有的好处，而且还没有为管理键而增加的开销。此外，这是唯一一种能够高速缓存共享存储，而且在数据驻留高速缓存时可以在进程中间共享的虚拟高速缓存结构。对于采用了大规模的高速缓存，而且共享存储或者映射文件使用率很高的系统来说，这会取得显著的性能收益。

这种结构的主要缺点是，为了判断是否命中，需要在每次访问高速缓存期间进行虚拟到物理的地址转换。如果系统中使用的 MMU 速度慢，那么这就会削弱或者抵消使用物理标记所带来的好处。尽管能够同时高速缓存多个进程的现场，从而有可能在现场切换完继续执行某个进程的时候该进程的数据已经驻留在高速缓存中了，但是除非高速缓存是全相联的，或者每组有大量的高速缓存行，否则进程倾向于使用相同虚拟地址的实际情况会降低这种可能性。这个问题所有类型的虚拟高速缓存都有，而且要归因于如何索引高速缓存（与如何标记高速缓存行相对）。最后有一个缺点，但不大，即：不冲洗高速缓存，或者不采用无高速缓存的操作，就不能在任意边界上共享存储器。

5.4 习　题

5.1　带有物理标记的全相联虚拟高速缓存有意义吗？

5.2　一个 5 路组相联高速缓存，每行 16 字节，共 256 组。如果每上页面 4KB，那么应该将其看成是一个带有物理标记的虚拟高速缓存还是纯虚拟高速缓存？

5.3　一个直接映射高速缓存，每行 32 字节，共 512 行。如果页面的大小是 8KB，那么应该将其看成是一个带有物理标记的虚拟高速缓存还是纯虚拟高速缓存？

5.4　一个系统使用 3 路组相联高速缓存，每行 64 字节。如果页面的大小是 64KB，那么高速缓存可以有多大而且仍然能看作是物理高速缓存？

5.5　一个系统使用 256KB 4 路组相联带有物理标记的虚拟高速缓存。每行包含 16 字节。如果下面的每一对虚拟地址代表了到地址空间内相同共享存储段的一个别名，那么哪些地址对要求内核冲洗高速缓存，以便维护一致性？解释原因。

a. 0x1000 和 0x10000

b. 0x1100 和 0x11100

c. 0x52a40 和 0x53a40

d. 0x8ffe90 和 0xfafe90

e. 0x123450 和 0x1234560

5.6 参考 5.2.2 小节，对于建立写时复制共享机制来说，当组的大小等于 2 或者更大的时候，为什么没有必要像带有键的虚拟高速缓存那样冲洗高速缓存？

5.7 为一个 1MB 大带有物理标记的虚拟高速缓存绘制类似于图 5-2 的一幅图，其中虚拟地址为 48 位，页面的大小为 64KB。该高速缓存是双路组相联高速缓存，每行 256 字节。

5.8 在 5.2.5 小节中说过，因为即使数据可以被高速缓存，MMU 也会防止对页面的访问，所以采用 brk 和 sbrk 释放存储的情况下可以推迟进行冲洗。在取消附加共享存储区或者映射文件的情况下也能推迟冲洗高速缓存吗？

5.9 如果一个进程在带有物理标记的虚拟高速缓存中有修改过的数据，且高速缓存采用了写回策略，该进程使用 sbrk 来释放这些修改过的数据所对应的页面，那么如果按照 5.2.5 小节的描述推迟冲洗高速缓存，当那些行被替换的时候会发生什么情况？

5.10 内核将来自某个进程的一页调出到交换设备上，并且将该物理页面重新分配给一个新进程使用。之后，它又把这一页调入系统，将它放入另一个物理页面中。描述当采用带有物理标记的虚拟高速缓存时，为了保持一致性所必须进行的冲洗操作。

5.11 一个进程执行一次原始 I/O 读取操作，将数据读入其堆栈中的缓冲区。描述当采用带有物理标记的虚拟高速缓存时所必须进行的冲洗操作。如果代之以执行一次原始写操作，那么必须进行的操作有哪些不同？

5.5 进一步的读物

[1] Chao, C., Mackey, M., and Sears, B., "MACH on Virtually Addressed Cache Architectures," *Proceedings of the USENIX MACH Workshop*, 1990.

[2] Frink, C.R., and Roy, P.J., "A Virtual Cache-Based Workstation Architecture," *Proceedings of the 2nd International Conference on Computer Workstations*, March 1988.

[3] Goodman, J.R., "Coherency for Multiprocessor Virtual Address Caches," *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987, pp. 72-81.

[4] Inouye, J., Konuru, R., Walpole, J., and Sears, B., "The Effects of Virtually Addressed Caches on Virtual Memory Design and Performance," *Operating Systems Review*, Vol. 26, No. 4, October 1992, pp. 14-29.

第6章 物理高速缓存

我们研究的最后一种高速缓存体系结构是物理高速缓存。这种类型的高速缓存抛弃了虚拟地址的一切用法，而是使用物理地址来索引和标记高速缓存行。其优点在于彻底消除了别名和歧义问题，但代价是每次访问都需要进行一次地址转换。这种组织结构也能使用一种新技术——总线监视（bus watching）——来保持 I/O 操作的高速缓存一致性。最后以讨论多级高速缓存来结束本章的内容。

6.1 物理高速缓存的组成

物理高速缓存采用数据的物理地址来进行索引和标记。物理高速缓存从不使用虚拟地址。当然，这种类型的组织结构要求每次高速缓存查找操作的时候都要由 MMU 转换虚拟地址。物理高速缓存的组织结构如图 6-1 所示。

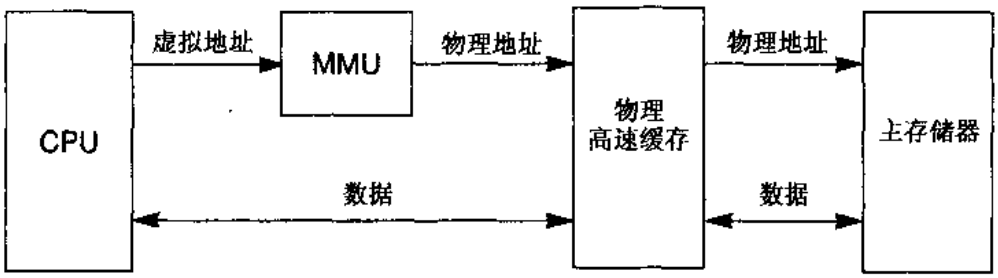


图 6-1 物理高速缓存的组织结构

这类设计的诱人之处在于既不会出现歧义也不会出现别名。没有共享数据的无关进程各自都分配得到不同的存储物理页面。因此，来自不同进程的物理标记决不会吻合。共享数据的进程使用相同的物理页面，使得它们索引到相同的高速缓存行或者组，并且会匹配标记从而发生一次命中。因为访问高速缓存时不牵扯到虚拟地址，所以不可能有别名问题。最终结果一般是不需要冲洗高速缓存。如果采用了总线监视技术（在 6.2.6 小节中说明），那么始终都不必冲洗高速缓存。在这种情况下，高速缓存变成对操作系统和用户程序来说完全透明的。这种情况让物理高速缓存适合于在最初没有在设计中考虑高速缓存的系统上使用。例如，IBM PC 及其兼容机就是这样，它们采用的 Intel 80X86 系列中的早期版本（80386 以及更老的芯

片）没有片上高速缓存。为了保持同给以前的微处理器版本编写的操作系统和应用软件的兼容性，80486 和 Pentium 都有了片上的物理高速缓存。80486 有一个 8K 的写直通 4 路组相联高速缓存。Pentium 包含独立的指令和数据高速缓存，每个都是 8K 的双路组相联高速缓存。Pentium 的数据高速缓存使用了写回策略。

和采用带有物理标记的虚拟高速缓存的情况一样，物理高速缓存中的数据在其驻留于高速缓存内的时候是可以共享的。除此之外，使用物理地址来索引高速缓存还能让来自不同进程（没有共享存储）内相同虚拟地址的数据索引到不同的高速缓存行，因为每个数据都使用了不同的物理地址。这就解决了所有类型的虚拟高速缓存可能有的性能问题之一，即无关的进程竞争相同的高速缓存行，因为这些进程频繁地使用相同的虚拟地址范围。这种竞争降低了被高速缓存的数据在进程下次运行的时候还存在于高速缓存中的可能性。物理高速缓存有可能将这样的数据均匀地分布在高速缓存中，从而获得更好的性能。

一般而言，片外（外部）高速缓存是物理高速缓存。大多数没有片上高速缓存的处理器都是这种情况，比如 Intel 80386 和 MIPS R2000/R3000，这两种芯片的地址行只包含物理地址。片外的硬件不能用虚拟地址，这就使物理高速缓存成为一种自然而然符合要求的选择。尽管现在大多数处理器都有片上高速缓存，但通常是在访问外部高速缓存之前检查它们。到那个时候，虚拟地址往往已经被转换好了，进而能使用外部的物理高速缓存，还不会多花时间。6.3 节将详细介绍多级高速缓存的使用。

根据高速缓存的大小和组成不同，在地址转换完成之前查找操作可能就开始了。正如在 5.1 节中所提到的那样，如果散列算法生成索引所使用的全部比特位都来自页面偏移量，那么通过从虚拟地址获得这些比特位就可以立即开始查找操作。这就提供了一种有好处的硬件优化措施，能够在 MMU 转换地址的同时索引并读取相关的高速缓存行或者组。这还带来了有物理标记的虚拟高速缓存所拥有的性能优点，以及物理高速缓存（如果有的话，规模小，而且供冲洗需要）的软件好处。不过，所有情况下都必须在比较标记之前完成转换。

80486 和 Pentium 的高速缓存就使用了这种技术。例如，Pentium 的高速缓存行有 32 字节，这意味着共有 128 组（8K÷32 字节/行÷2 行/组）。因此，“位<4..0>”会选择高速缓存行内的字节，而“位<11..5>”则选择组。因为使用的是 4K 大小的页面，所以页面偏移包含在“位<11..0>”中，从而可以让高速缓存查找操作在虚拟地址转换之前开始。

如果每一组有足够数量的行，那么这项技术就和比页面大的高速缓存一起使用。80486 和 Pentium 的高速缓存是页面大小的两倍，而德州仪器公司的 SuperSPARC 芯片的高速缓存更大。“位<5..0>”选择行内的字节，而“位<11.6>”选择组。因为使用的是 4K 大小的页面，所以所有的索引位都来自页面偏移。类似地，它的指令高速缓存有 16K 大，且为 4 路组相联高速缓存，每行 32 字节。“位<4..0>”选择行内的字节，而“位<11..5>”选择组，所有的比特位还是都来自于页面偏移。

对于较大的物理高速缓存，需要来自物理页号的比特位来形成索引，那么在开始高速缓存查找操作之前必须完成地址转换。MIPS R4000 就是这种情况，它可以有多达 4M 的外部高速缓存。这个高速缓存是直接映射高速缓存，每行可多至 128 字节。在这种情况下，“位<6..0>”选择行内字节，而“位<21..7>”选择行。因此，在组成高速缓存索引之前，需要来自物理页

号的“位<21..12>”。因为在片上高速缓存的查找操作期间，虚拟地址已经转换好了，所以需要物理页号来索引外部高速缓存，而不会造成性能损失。

6.2　管理物理高速缓存

物理高速缓存不需要或者很少需要冲洗高速缓存，其中的原因将在下面各小节中阐述。

6.2.1　现场切换

因为物理标记能防止歧义问题，而以物理地址索引能防止别名问题，所以在现场切换时不需要冲洗高速缓存。对于有大规模高速缓存的系统来说，这格外有益，因为它能大大减少现场切换的时间。采用大规模高速缓存的第二个好处是，一个进程的数据在别的进程正在运行的同时更有可能会保留在高速缓存中，因为物理索引机制能将来自不同进程的数据在整个高速缓存上均匀地分布。于是，一个进程可以期望下一次执行的时候有更好的命中率。但是，对于小规模的物理高速缓存（小于或者等于典型进程局部引用的大小）来说，情况并非如此，因为每个进程都会用它自己的局部引用替换前一个进程的。

6.2.2　fork

不需要在调用 fork 时为了建立写时复制共享机制而冲洗高速缓存，因为父进程和子进程都使用相同的物理地址来引用共享数据。这意味着它们在寻址相同的数据时都会索引和命中相同的高速缓存行。

物理高速缓存也简化了写时复制共享结束时（或者实现不带写时复制机制的 fork 调用时）复制一页的任务。采用带有键或者物理标记的虚拟高速缓存时，为了避免别名，必须冲洗临时映射的地址范围（参见 4.2.2 小节）。物理高速缓存不必这样做，因为在复制入新页面的操作完成时，已经正确地标记了数据，而且数据处于正确的行或者组里。临时映射所使用的虚拟地址和高速缓存不相干。

6.2.3　exec、exit、brk 和 sbrk

在使用物理高速缓存的时候，这些系统调用执行期间释放内存都不需要任何冲洗高速缓存的操作。在释放内存的时候，没有哪个进程在用相应的物理地址范围；因此，没有哪个进程会命中可能在高速缓存中的任何过时数据。当这块内存最终又重新分配给某个进程的时候，必须确保它无法访问到高速缓存中任何剩下的过时数据。因为 UNIX 内核一定是在 I/O 读操作期间（比如满足一次缺页错的需要）或者通过填零来填页，所以肯定能做到这一点。例如，如果同为了增加堆栈而分配新内存时一样用零来填页，那么内核就会通过高速缓存来清理存储器，从而以零替换掉任何残余的过时数据。I/O 操作的一致性将在 6.2.6 小节中进行讨论。

在其中任何一种情况下，当初始化新分配的页面时，都会清除过时数据，所以不需要显式地高速缓存冲洗操作。

6.2.4 共享存储和映射文件

当使用物理高速缓存的时候，保持共享存储和映射文件的一致性都不需要冲洗高速缓存。和采用带有物理标记的虚拟高速缓存的情况一样，可以在数据驻留在高速缓存中的时候共享它。和物理标记的虚拟高速缓存不同，因为不会出现别名，所以在共享进程访问共享存储所用的虚拟地址上没有限制。

6.2.5 用户-内核数据的歧义

在高速缓存中使用物理地址的时候，不可能在用户和内核之间出现别名和歧义。因此，不需要进行高速缓存冲洗操作来保持一致性。只要内核页面没有被映射到用户进程的虚拟地址空间中，那么就没有哪个用户进程能够访问内核数据，因为每次访问所完成的 MMU 操作都会检查页面权限。

6.2.6 输入输出和总线监视

因为 I/O 的 DMA 操作是和高速缓存无关的，所以前面的虚拟高速缓存结构都需要冲洗高速缓存来保持 I/O 操作的一致性。但是，物理高速缓存具有实现一种称为总线监视的技术的能力，以保持 I/O 操作的高速缓存一致性，而无需由操作系统冲洗高速缓存。除了让高速缓存执行来自 CPU 的查找操作之外，还让它监视（或者说观察）系统总线上来自 I/O 设备的 DMA 操作，就能达到上述目的。以前由操作系统执行的冲洗高速缓存的操作现在则由硬件自动处理。图 6-2 显示了这种系统组成的一个高级视图。

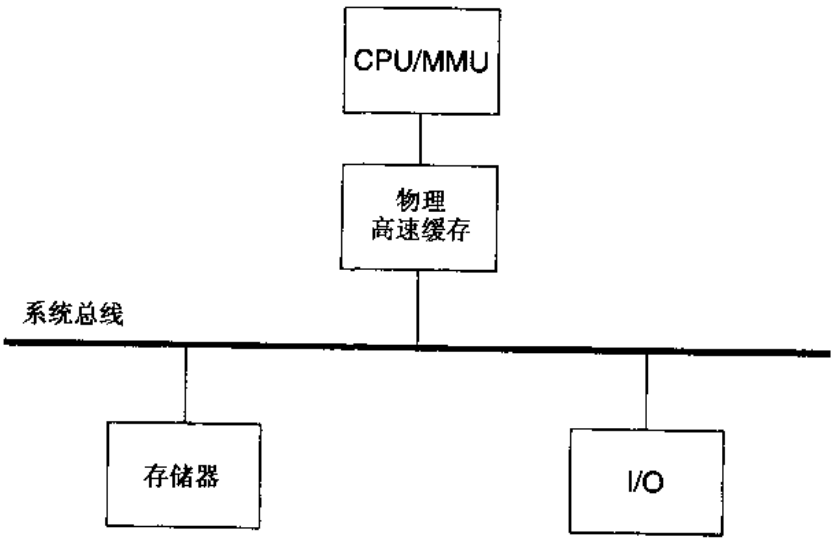
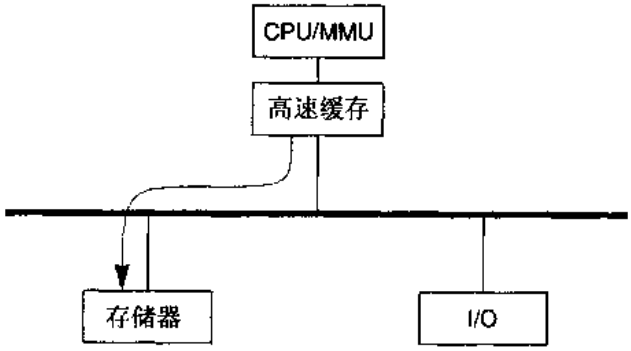
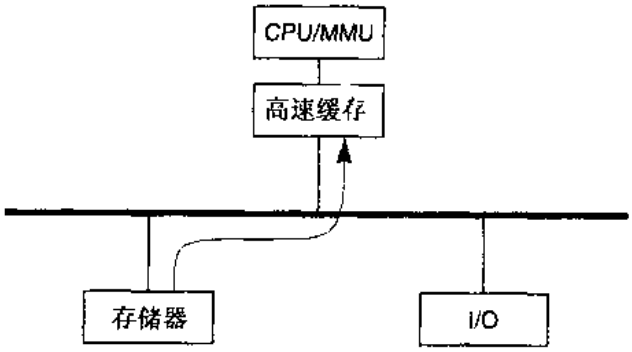


图 6-2 高速缓存、存储器和 I/O 在系统总线上的互连结构

系统总线担当了 CPU/MMU/高速缓存、主存储器和 I/O 设备之间一个共同的互连通道。它是一种基于广播的介质，这意味着任何单元都可以接收到总线上由任何其他连接到总线的单元所发送的信息。CPU/MMU/高速缓存通过把数据的物理地址发到总线上来读取存储器，如图 6-3(a)所示。存储器通过返回总线上被请求的数据来响应地址，高速缓存在总线上接收到数据（参见图 6-3(b)）。



(a) CPU/MMU/高速缓存把地址发送到主存储器单元



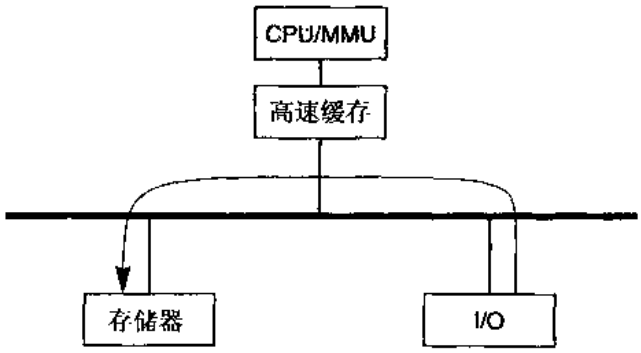
(b) 主存储器把数据返回给高速缓存

图 6-3　CPU/MMU/高速缓存读操作

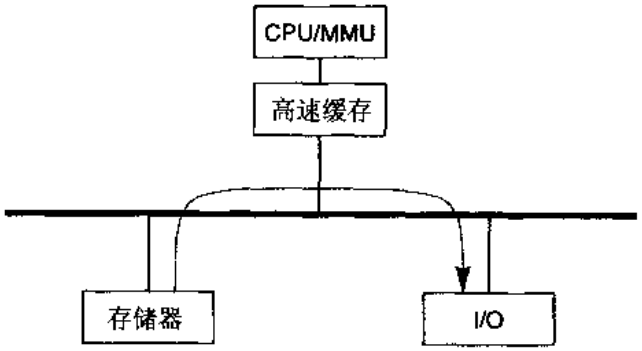
I/O 设备以相同的方式执行 DMA 操作。为了把数据写入设备（比如磁盘）I/O 控制器将数据的物理地址放到总线上（参见图 6-4(a)）。接着，主存储器单元通过把相应的数据发送到总线上来进行响应，设备随后在总线上收到数据（参见图 6-4(b)）。

当数据被 CPU 或者 I/O 写入到存储器的时候，物理目的地址和数据一起被发送到总线上。存储器单元捕获地址和数据，将数据写入被寻址的存储器位置。

数据是在一次或者多次总线交易（bus transaction）中发送到总线上的。每次总线交易只发送一定数量的数据，其范围一般从一次几个字到一次几十个字。因此，如果需要发送大量的数据，比如从磁盘读取一页的时候，传输会被分成多次总线交易，每次都包含有目的地址和数据。



(a) I/O 把地址发送给主存储器单元



(b) 主存储器把数据返回给 I/O 设备

图 6-4 从主存储器到设备的 I/O DMA 操作

因为高速缓存能够看到所有的总线交易，所以总线监视技术实现起来直截了当。当高速缓存没有使用总线来读或者写存储器本身的时候，它会监视所有 I/O 设备执行的总线活动，这常被称为监听（snooping）。对于每个感兴趣（后面介绍）的总线交易，高速缓存检查其内容来查看交易中的物理地址是否驻留在高速缓存中（参见图 6-5）。监听的高速缓存查找操作和 CPU 访问查找操作是一样的：散列计算地址，索引到相应的行或者组，检查标记看是否发生了命中或者缺失。如果发生一次缺失，则高速缓存什么也不需要做。在这种情况下，

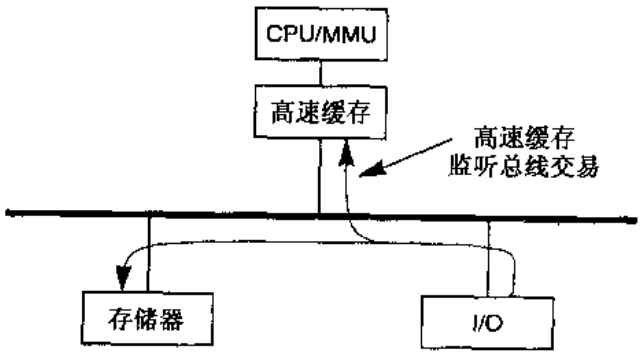


图 6-5 高速缓存监听 I/O 对存储器的访问

总线交易的 DMA 操作就好像在没有高速缓存的系统中一样完成。如果地址在高速缓存中命中，那么硬件要采取的行动取决于总线操作对主存储器是读还是写操作，以及高速缓存是写直通还是写回高速缓存。

首先考虑采用写直通高速缓存机制的情况，因为它比较简单。此时，高速缓存能够忽略从 I/O 设备读主存储器的 DMA 操作。因为是写直通高速缓存，所以始终会更新存储器，所以 I/O 设备一定可以从主存储器读到数据的正确版本。但是，高速缓存必须监听所有从 I/O 设备写主存储器的操作。如果发生一次缺失，那么高速缓存什么也不做，从设备来的数据就照常被写入存储器。如果发生一次命中，那么要使高速缓存中的这行无效，并且不会影响 DMA 写主存储器操作的完成。这就保证了高速缓存决不会保留相对于主存储器来说过时的数据。当 CPU 以后引用相应地址的时候，它将引发高速缓存缺失，并且从主存储器读取数据。当 DMA 写存储器操作期间发生命中的时候，有些实现选择用从设备来的新数据替换高速缓存中的数据。这样做的基本效果等同于保持高速缓存和主存储器同步。这样的实现假定 CPU 很快需要数据，所以值得在高速缓存中保留一个副本。

采用写回高速缓存的时候，情况会更复杂一些，因为这类高速缓存必须监听所有的总线交易。在执行来自存储器的 DMA 读操作的情况下，高速缓存可能包含修改过的数据，这意味着存储器中相应的数据是过时的。I/O 设备一定不能读取这个过时数据，所以高速缓存必须监听 DMA 读操作。在这样一次操作期间，当命中一行修改过的数据时，高速缓存就把它的数据返回给 I/O 设备，防止主存储器以过时的数据进行响应。这个动作对于 I/O 设备来说是透明的。如果命中了一行未经修改的高速缓存行，那么存储器和高速缓存是同步的，两者都可以返回数据。在这种情况下，无论是高速缓存还是存储器返回数据，都是和实现无关的。如前所述，当一次 DMA 读操作期间发生缺失时，高速缓存不需要做任何事情，从而让主存储器返回数据。

和采用写直通高速缓存机制一样，当 DMA 写主存储器的操作没有在高速缓存中命中的时候，写回高速缓存也不需要做任何事情。当 DMA 写操作期间发生命中时，所要采取的操作则取决于高速缓存行的当前状态。如果高速缓存行没有被修改过，那么就可以像写直通高速缓存一样使之无效（有些实现和以前一样把新数据载入高速缓存）。如果高速缓存行被修改过，而且 DMA 写操作替换了整个高速缓存行（也就是说，行的大小等于总线交易），那么就像该行没有修改过一样来处理它，使之无效。在这种情况下，高速缓存中修改过的数据在逻辑上被 DMA 写操作所覆盖，所以可以丢弃过时的高速缓存数据。但是，如果 DMA 写操作没有替换整个高速缓存行，当总线交易的大小比行的大小要小的话就会发生这种情形，那么高速缓存必须捕获来自 DMA 操作的新数据，更新高速缓存行。此时它不能使高速缓存行无效，因为这会失去行中 DMA 没有替换的那部分里被修改过的数据。例如，假定高速缓存行的大小为 32 字节，总线交易为 16 字节。假定高速缓存在第 5 行中保存有修改过的数据。出现一次向存储器中对应于高速缓存第 5 行后半段数据的地址执行的 DMA 写操作。如果高速缓存现在要使整个高速缓存行无效，它就会失去 CPU 写入该行前 16 字节的任何修改过的数据。因此，高速缓存必须把数据载入该行的后 16 字节。该行保持处于修改过的状态，以便最终在替换该行的时候把数据写回主存储器。在存储器中 I/O 缓冲区的起始和结尾处，出现必须更新部分行的情形很常见。这样的缓冲区并不能认为是从对应于高速缓存行边界处的地址开始的，也不会要求它们的长度必须是高速缓存行大小的倍数。这种情形和 3.3.7 小节中所

描述的情形相同（参见图 3-10 和 3-11），后者的内核不得不冲洗高速缓存来保持一致性。带有总线监视机制的物理高速缓存则在硬件中自动执行这项任务。

因为 I/O 操作的高速缓存一致性现在是在硬件中处理的，所以不需要操作系统的干预。既然本章所描述的其他情形都不需要冲洗高速缓存，那么对于软件来说，带有总线监视机制的物理高速缓存是完全透明的。正因为有这样的好处，所以大多数现代处理器都支持总线监视技术。Intel 80X86 系列处理器、MIPS R4000MC、Motorola 68040 和 88000 以及 TI SuperSPARC 都是如此。如果高速缓存没有总线监视功能，比如 MIPS 4000PC 的低档版本和 TI MicroSPARC，那么必须有和前面章节所介绍的相同的冲洗操作来保持 I/O DMA 操作的一致性。

一般而言，只会在物理高速缓存上找到总线监视技术。有几种实现，如 Intel i860 XP，也连同虚拟高速缓存一起支持它。i860 XP 要做到两点来支持这样的一种配置。首先，高速缓存可以用虚拟地址或者物理地址来进行索引。因为 16K 独立的指令和数据高速缓存是 4 路组相联高速缓存，每行 32 字节，所以可以做到这一点。因为总共有 128 组，所以“位<11..5>”选择组，而“位<4..0>”选择行内的字节。因为页面的大小是 4K，所以虚拟地址和物理地址中的“位<11..0>”都是一样的。其次，两块高速缓存都有两个地址标记：一个包含虚拟地址，一个用于物理地址。这就有了两条独立的路径来访问高速缓存，如图 6-6 所示。

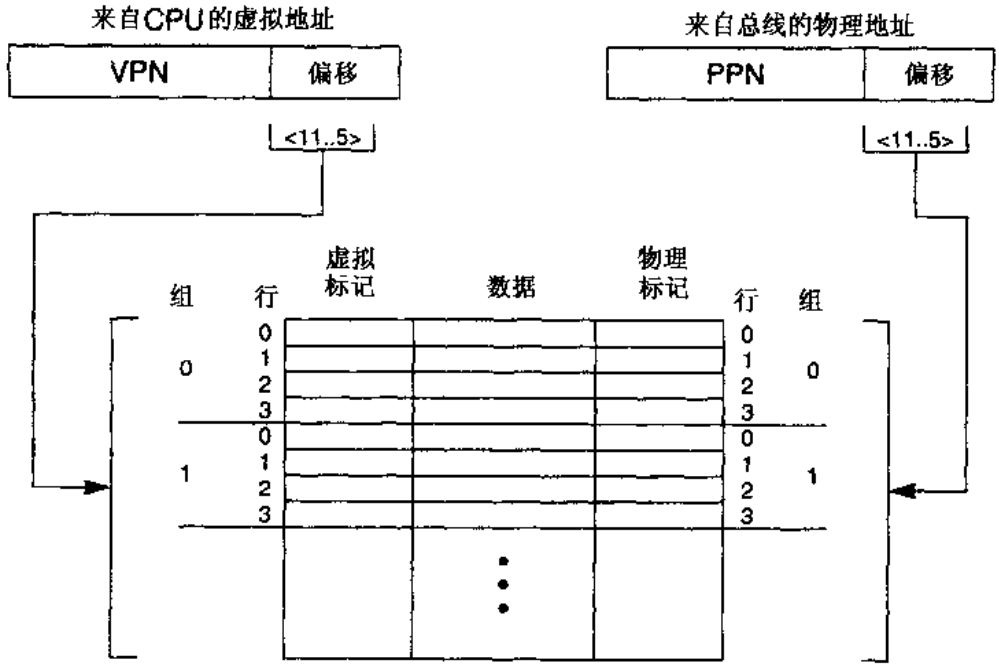


图 6-6 Intel i860 XP 高速缓存的体系结构

和采用虚拟高速缓存的情况一样，CPU 无需等候 MMU 完成地址转换，就可以完成一次高速缓存访问。CPU 以来自页面偏移的比特位来索引高速缓存，并且检查虚拟标记，看是否会命中。当在一次缺失期间载入一行的时候，设置这行的虚拟标记和物理标记都对应于刚刚取得的数据的虚拟地址和物理地址。因此，标记中的物理地址就是同一行内虚拟地址标记的

转换结果。使用来自 DMA 操作物理地址的页面偏移中的比特位来索引高速缓存，并且使用物理标记来检查是否命中，就可以支持总线监视机制。因为在虚拟地址和物理地址中，页面偏移里的比特位都是相同的，所以使用其中任何一种地址都会索引到相同的组。这就保证了即使 CPU 使用一个虚拟地址，而总线监视使用一个物理地址，总线监视硬件也能够定位任何由 CPU 所高速缓存的行。采用这种技术，在 Intel i860 XP 上运行的总线监视机制就和 Intel 80X86 以及前面介绍的其他处理器的物理高速缓存一样。

i860 XP 高速缓存结构的另一个额外的好处是硬件能够自动处理别名问题。在缺失处理期间，通过检查物理标记，查看是否有别名状态就可以达到上述目的。例如，假定位于 0xa000 的物理页面在一个进程的地址空间内 0x15000 和 0x952000 两处虚拟地址上有别名。如果高速缓存一开始为空（所有的行都标记为无效），那么当进程引用 0x15ffo 的时候，就会把物理地址 0xaffo 的数据填入 127 组内的一行。这一行的虚拟标记设为 0x15，物理标记设为 0xa（通常，用于索引高速缓存的比特位没有保存在标记中，因为它们可以根据行的位置进行重构）。现在，假定进程引用位于 0x952ff0 的另一个别名。这就再次索引到了组 127，但是会发生一次缺失，因为没有哪个虚拟标记和这个地址吻合。在 MMU 转换了这个地址之后，处理器开始从主存储器执行一次读操作，取得数据，同时检查被索引的组内的物理标记，看是否命中。在这个例子中，发生了一次命中，表明遇到了别名问题。为了防止将两个别名都载入高速缓存，而且可能让彼此失去同步，i860 XP 忽略了从主存储器返回的数据，只是改变了包含别名的行上的虚拟标记。物理标记则保持不变。因而，过去有虚拟标记 0x15 的行现在有虚拟标记 0x952。这一活动对于软件来说是透明的，从而避免了让操作系统显式地冲洗高速缓存来防止别名问题。在所有其他方面，i860 XP 上的高速缓存都表现为虚拟高速缓存，需要第 3 章中所描述的其他类型的冲洗操作。

6.3　多级高速缓存

有些实现可以扩展图 2-1 中的存储层次结构，包含一级以上的高速缓存机制。这样做的目的是通过为高速缓存提供高速缓存来进一步提高性能。图 6-7 展示了一种有两级高速缓存机制的组织结构。

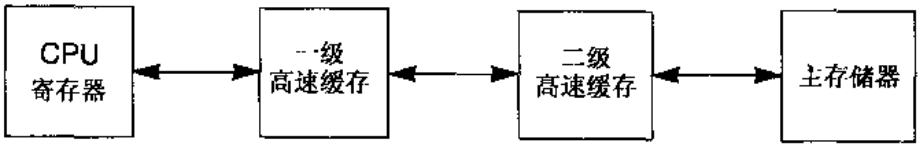


图 6-7　两级高速缓存

和以前一样，存储层次结构的级别越靠近 CPU，它的速度就越快，规模一般也越小。之所以使用层次结构是出于经济因素：高速存储器很贵，它的密度也低。因此，一级高速缓存（也称为主高速缓存，primary cache）比二级高速缓存（也称为次级高速缓存，secondary cache）速度快，通常规模也小一些。通过使用稍微有一点儿慢的二级高速缓存，就可以平衡系统的

成本和性能。对这个高速缓存的速度要求较低，从而允许使用更便宜的部件，这又反过来让二级高速缓存的规模更大。一级高速缓存较高的速度弥补了二级高速缓存较低的速度。

这两种高速缓存的组成没必要相同。一种典型的安排是这样的，大规模的物理二级高速缓存配合某种类型的小规模高速虚拟高速缓存一起使用。这意味着能够快速地访问一级高速缓存，甚至不需要进行一次 MMU 转换。如果发生一次缺失，那么接下来检查二级高速缓存。这样的系统会并行地启动 MMU 转换和一级高速缓存查找操作。这意味着，如果需要检查二级高速缓存，则物理地址已经就绪了。于是，这就将两类高速缓存结构的优点结合了起来，同时又削弱了缺点。

使用多级高速缓存不能改变必须冲洗高速缓存的情形。这些操作是由各自的高速缓存体系结构所决定的，必须在本章和前面章节所介绍的情况下执行。

6.3.1 带有次级物理高速缓存的主虚拟高速缓存

举第一个例子，其中使用了带有外部物理高速缓存的 Intel i860 XR。i860 XR 具有独立的指令和数据高速缓存，两者都是虚拟高速缓存。数据高速缓存使用写回策略。注意，XR 没有前面介绍过的 XP 所使用的物理标记，因此不支持总线监视和别名检测。假定在外部给 i860 增加一个带有总线监视机制的物理高速缓存，这块高速缓存既包括指令高速缓存也包括数据高速缓存，故称之为统一高速缓存（unified cache）。图 6-8 展示了这一组织结构。

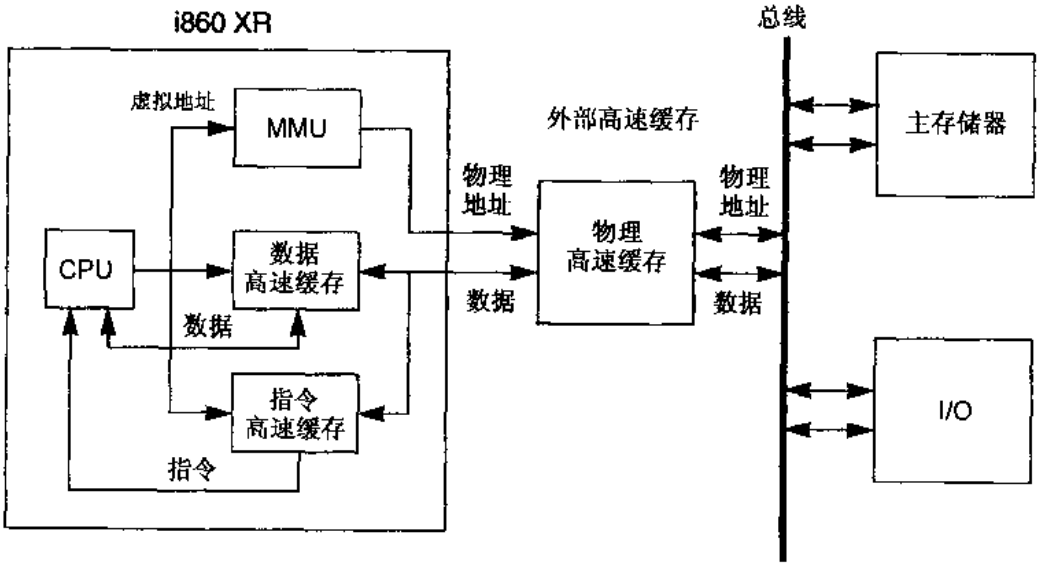


图 6-8 带有外部物理高速缓存的 Intel i860 XR

在这幅图中，i860 芯片内的 CPU 向所需的高速缓存和 MMU 发出虚拟地址。高速缓存按照请求把它们的数据和指令返回给 CPU。只有当内部高速缓存中发生一次缺失的时候，才把物理地址发送给外部高速缓存，以执行查找操作。外部高速缓存中的一次缺失会使得物理地址通过总线转发给主存储器。在这种组织结构中，访问主存储器之前先检查高速缓存，所以称之为直通高速缓存（look-through cache），它得名于访问的串行特性。在检查高速缓存的

同时，把物理地址发送给主存储器的组织结构则称为后援高速缓存（look-aside cache）。后援方式的优点是，如果出现缺失，则可以更快地从主存储器得到数据。在高速缓存大、命中率高的情况下，直通方式是一种更好的方法。使用直通和后援方式对于软件来说是透明的。

因为 i860 上的高速缓存是虚拟高速缓存，可能出现别名和歧义问题，所以在第 3 章所描述的所有情况下，内核都必须冲洗高速缓存。例如，在现场切换时，必须将数据高速缓存写回，并使之无效。考虑到次级高速缓存是物理高速缓存，所以为了保持一致性，只需将主数据高速缓存写回到次级高速缓存即可，不需要把数据写回到主存储器。不出所料，根本不需要冲洗次级高速缓存，因为它是纯物理高速缓存。哪怕是在 I/O 期间，内核也只需写回 i860 的高速缓存，并（或）使之无效就行了。一旦软件顾及到了片上高速缓存的一致性，硬件就会通过总线监视技术来保持次级高速缓存内的一致性。简而言之，内核必须让自己只考虑主高速缓存，它可以忽略外部高速缓存的存在。

主指令高速缓存和主数据高速缓存之间没有直接连接起来。所以，使用可以自我修改的代码有可能在指令高速缓存中造成过时的指令。当 CPU 以前取过指令，让它们被缓存在指令高速缓存中，而随后又修改了这些指令的时候，就可能出现上述情况。当 CPU 修改它们的时候，它会把新指令保存在数据高速缓存中。这样做并不会影响到指令高速缓存，它仍旧继续缓存着过时的数据。在这些情况下，必须使过时的数据无效，以便下次读取它们的时候出现一次高速缓存缺失。既然使用了统一的次级高速缓存，那么就没有必要把新指令从主数据高速缓存写回主存储器。相反，只需要把它们写入外部高速缓存。当主指令高速缓存中发生了一次缺失的时候，访问主存储器之前先要检查次级高速缓存。

6.3.2　带有物理标记的主虚拟高速缓存和次级物理高速缓存

这种情况以 MIPS R4000 为例进行说明。它包含独立的片上指令和数据高速缓存，而且能够控制一块外部的合成指令和数据的高速缓存（R4000 也支持独立的外部高速缓存，但是本例采用了一块统一的高速缓存）。片上高速缓存以物理标记来虚拟索引，而外部高速缓存则是物理高速缓存。所有的高速缓存都是直接映射高速缓存，数据高速缓存和外部高速缓存都支持带有写分配机制的写回策略。此外，外部高速缓存还支持总线监视技术。在 R4000 上，内核能够通过在页表的项上设置的一个代码来逐页地启动总线监视功能。这一组成的高层框图如图 6-9 所示（注意，这是一个简化的概念模型，它从软件的角度展示了结构和数据路径。例如，外部高速缓存不是直接连接到实际系统的总线上的。相反，总线连接到了 R4000 上，R4000 有一条到外部高速缓存的单独连接。这些硬件细节对于下面的讨论没有影响）。

在这种布局下，当 CPU 要引用一次内存的时候（既可以是读入或者保存指令，也可以是取指令），它就把虚拟地址发送给 MMU 和适当的内部高速缓存。在主高速缓存执行索引操作并且访问所需的高速缓存行的同时，MMU 转换虚拟地址并且把物理地址发送给高速缓存，把它和被索引的高速缓存行的标记进行比较。如果发生一次命中，那么就把数据或者指令返回给 CPU，完成内存引用。如果发生一次缺失，那么就使用已经转换好的物理地址访问外部高速缓存。如果在外部高速缓存中找到了数据，那么就把它载入适当的主高速缓存，并返回给 CPU。在外部高速缓存中出现的一次缺失会使得物理地址被发送到主存储器，以检索期望的数据。在这种情况下，数据既读入主高速缓存，也读入次级高速缓存，而且还被发送到 CPU。

因为高速缓存使用写分配机制，所以在发生缺失期间，不管是一次读入还是一次保存操作发生的时候，主数据高速缓存和次级高速缓存都一定会载入数据。这就保证了主高速缓存始终是次级高速缓存的一个子集，很快我们就可以看到这种做法的重要性。

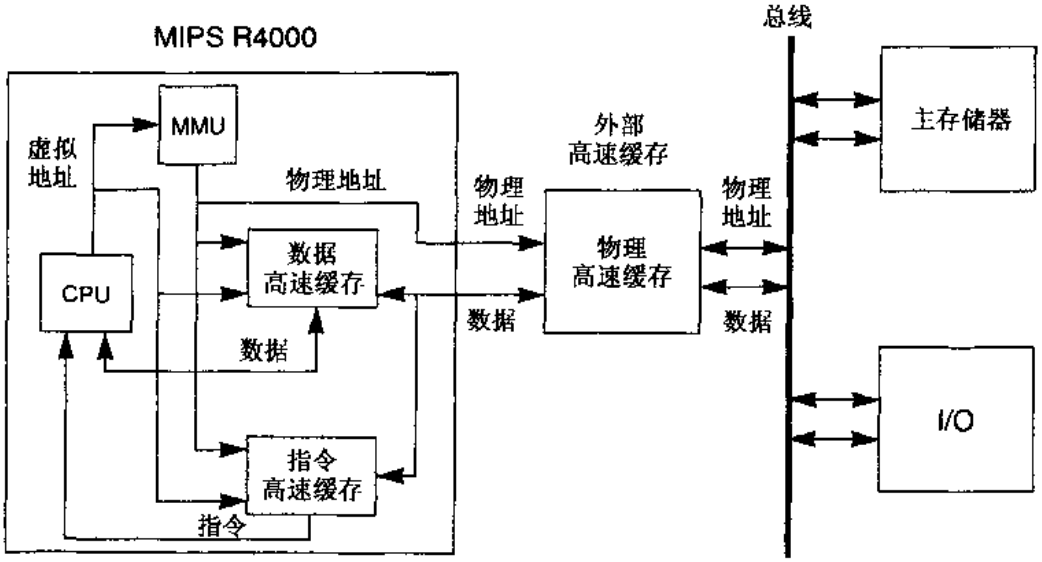


图 6-9 带有外部物理高速缓存的 MIPS R4000

和所有采用有物理标记的高速缓存体系结构一样，操作系统几乎不需要通过冲洗高速缓存来保持一致性。不会出现歧义问题，因为标记中的物理地址解决了这个问题。因为系统中的指令和数据高速缓存被分开了，所以必须按照上一节的介绍来处理具有自我修改能力的代码。剩下要予以解决的两种情况是 I/O 的主高速缓存一致性，以及主高速缓存的别名问题。

通过把主高速缓存的行索引保存在次级高速缓存里每行的标记中，R4000 提供了解决这两个问题的辅助手段。只要在一次缺失期间向主高速缓存读入了数据，那么被读入行的索引就被保存在次级高速缓存中那些保存来自相同地址的数据的行里。如前所述，主高速缓存始终是次级高速缓存的一个子集，所以不可能让数据在主高速缓存中但却没有被次级高速缓存保存。对于硬件来说，这就有可能自动检测主高速缓存的别名问题。当主高速缓存里的一次缺失在次级高速缓存中命中的时候，就把保存在次级高速缓存标记中的主高速缓存索引和缺失期间检索的高速缓存行相比较。如果它们不相同，那么就表明在主高速缓存中可能出现了别名，因为这个次级高速缓存行上次满足缺失时把数据读入的主高速缓存行和当前这次主高速缓存发生缺失期间引用的主高速缓存行是不一样的。因为 R4000 是 RISC 处理器，所以它并没有包含解决别名问题的硬件，而是代之以在操作系统中产生一次虚拟一致性异常（virtual coherency exception），操作系统必须执行高速缓存冲洗操作来消除任何别名。接着，可以设置次级高速缓存中的索引，指向数据在主高速缓存中的新位置。在硬件中实现别名检测功能的情况下，内核只需要响应异常并冲洗高速缓存来处理出现虚拟别名的所有情形。对于这种设计来说，带有物理标记的虚拟高速缓存往往会有的各种限制和技术都没有必要了。

在总线监视操作期间，也会使用保存在次级高速缓存中的索引。如果一个 I/O 设备正在执行 DMA 读主存储器的操作，那么因为它们使用了写回策略，所以必须监听高速缓存。因

为次级高速缓存是物理高速缓存，所以可以使用来自总线的物理地址来索引和检查它。由于主高速缓存也采用了写回策略，所以在监听操作期间也必须检查它。和 i860 XP 不同，索引 R4000 的片上高速缓存不可以使用物理地址，因为它为了形成高速缓存索引，需要从虚拟页面框架号（page frame number，PFN）得到若干比特位，所以它会代之以使用次级高速缓存行的索引。从前面的讨论可以知道，这个索引指向了上次引用过次级高速缓存行的主高速缓存行。于是接着读取指定的主高速缓存行，并且对照物理地址检查它的标记。之所以必须这样做，是因为主高速缓存比次级高速缓存小得多，从而有可能用新数据替换被索引的主高速缓存行。以这样的方式替换主高速缓存行时，并不更新与主高速缓存行的过去内容相对应的次级高速缓存标记。正因为如此，在次级高速缓存中命中的监听操作必须检查被索引到的主高速缓存行里的标记，以确保它仍然包含有与次级高速缓存行相关的数据。如果发生一次命中，那么就会修改主高速缓存的数据，然后把这个数据返回给 I/O 设备。如果在主高速缓存中发生一次缺失，或者发现这个数据没有修改过，而在次级高速缓存中发现了修改过的数据，那么就把这个数据返回给 I/O 设备。如果在次级高速缓存中发生一次缺失，那么不需要检查主高速缓存（因为主高速缓存始终是次级高速缓存的一个子集），I/O 设备将从主存储器读取数据。DMA 写主存储器的操作也会执行类似的检查操作。总体而言，R4000 上的高速缓存监听机制对于软件来说是透明的。任何 I/O 操作都不需要显式地冲洗高速缓存来保持一致性。

6.4 小　结

物理高速缓存大大地减少了由操作系统来进行维护的要求。如果使用总线监视机制，那么由于所有的 I/O 操作一致性都在硬件中进行处理，所以不需要冲洗高速缓存。对于这些系统来说，就好像不存在高速缓存一样（从软件的角度来看）。我们所看到的一切效果就是因为高速缓存的数据减少了访问主存储器的时间，从而获得了更好的性能。物理高速缓存的缺点是 CPU 每次访问时都需要进行 MMU 转换。正因为如此，物理高速缓存把数据返回给 CPU 要比虚拟高速缓存多花时间。通过消除显式地冲洗高速缓存的需要，从而抵消了性能上的损失。物理高速缓存在性能上的缺点可以通过把它与虚拟高速缓存结合起来构成多级高速缓存来予以缓解。这就可以让系统既能利用两种体系结构的好处，又能减弱两者的缺点。

6.5 习　题

6.1　绘制类似于图 5-2 的图，其中系统的页面大小为 2K，采用双路组相联物理高速缓存，该高速缓存共有 512 组，每行 32 字节。

6.2　如果使用了总线监视机制，为什么不把对主存储器的全部更新读入到高速缓存中（例如，对于把数据从 I/O 设备传送到主存储器的每次总线交易来说，即使发生一次缺失，也要让高速缓存把数据保存在高速缓存中）？毕竟有些进程必须要从 CPU 访问数据，否则一定不会马上就能读取到数据。那么为什么不让数据在高速缓存中可用，从而能更快地访问到呢？

6.3 系统使用独立的指令和数据高速缓存是很普通的情况。如果一个系统给这两者都使用了物理高速缓存，区别在于数据高速缓存具有总线监视功能，而指令高速缓存没有，那么操作系统需要有什么变化（也就是说，在什么情况下需要冲洗指令高速缓存）？具体地说明会影响哪些系统调用，需要什么类型的冲洗操作，以及什么时候必须这样做。

6.4 绘制一幅类似于图 5-2 的图，显示出一个 4 路组相联物理高速缓存（128 组，每行 8 字节）对地址比特位的解释过程。假定系统的页面大小为 4K。

6.5 如果高速缓存的大小保持相同，但是使用全相联而不是 4 路组相联高速缓存，那么 Intel i860 XP 使用的总线监视技术还有效用吗？假定页和组的大小都不变。

6.6 考虑使用直接映射高速缓存而不是全相联高速缓存的情况，重做上一题。

6.7 如果在 R4000 上主数据高速缓存中发生了一次缺失，从而使得一个修改过的行要被替换掉，那么应该把修改过的数据写到什么地方？是次级高速缓存还是主存储器？

6.8 R4000 上的两个上高速缓存有可能包含相同数据的一个副本吗？如果有可能，那么主高速缓存的索引必须怎样？

6.9 假定 R4000 的主高速缓存和次级高速缓存都缓存了主存储器内地址 0x1000 里的数据。假定 CPU 引用了一个不同的地址，在两个高速缓存中都造成了一次缺失。如果这次引用索引到了主高速缓存中一个不同的高速缓存行，但在次级高速缓存中却索引到了和地址 0x1000 相同的高速缓存行，那么为了完成缺失处理，又要保持一致性，必须怎样做？

6.10 如果连同虚拟标记一起还保存了一个键（从 CPU 的角度来看，使之成为了一个带有键的虚拟高速缓存），那么 Intel i860 XP 的总线监视技术还能发挥效用吗？别名检测机制还能发挥作用吗？

6.11 如果 MIPS R4000 的片上高速缓存是带有键的虚拟高速缓存，而不是带有物理标记的虚拟高速缓存，那么将上高速缓存索引保存在次级高速缓存中的总线监视和别名检测机制还能够起作用吗？

6.12 如果 MIPS R4000 的组成类似于 Intel i860 XP 的组成，那么将上高速缓存索引保存在次级高速缓存中的总线监视和别名检测机制还能够起作用吗？

6.13 如果 MIPS R4000 的片上高速缓存是物理高速缓存，而不是带有物理标记的虚拟高速缓存，那么有必要把上高速缓存的索引保存在次级高速缓存中吗？

6.14 MIPS R4000 主高速缓存行的大小有没有什么同次级高速缓存行的大小有关的限制？它们是一定要相等吗，还是一个能比另一个短一些？

6.15 如果需要 i860 XP 的一个新版本，其中高速缓存的大小加倍了，为了让总线监视和别名检测功能继续发挥作用，必须怎样做？

6.6 进一步的读物

[1] Azimi, M., Prasad, B., and Bhat, K., "Two Level Cache Architectures," *Proceedings of Compcon* '92, February 1992, pp. 344-9.

[2] Baer, J.L., and Wang, W.H., "Architectural Choices for Multi-Level Cache Hierarchies," Technical Report TR-87-01-04, Department of Computer Science, University of Washington, January 1987.

[3] Baer, J.L., and Wang, W.H., "On the Inclusion Property for Multi-Level Cache Hierarchies," Technical Report TR-87-11-08, Department of Computer Science, University of Washington, November 1987.

[4] Bennett, B.T., Pomerene, J.H., Puzak, T.R., and Rechtschaffen, R.N., "Prefetching in a Multilevel Hierarchy," *IBM Technical Disclosure Bulletin*, Vol. 25, No. 1, June 1982, pp. 88-9.

[5] Chiueh., T., and Katz, R.H., "Eliminating the Address Translation Bottleneck for Physical Address Cache," *SIGPLAN Notices*, Vol. 27, No. 9, September 1992, pp. 137-48.

[6] Gecsei, J., "Determining Hit Ratios for Multilevel Hierarchies," *IBM Journal of Research and Development*, Vol. 18, No. 4, July 1974, pp. 316-27.

[7] Goodman, J.R., "Coherency for Multiprocessor Virtual Address Caches," *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987, pp. 72-81.

[8] Gustavson, D.B., "Cbomputer Buses-A Tutorial," *IEEE Micro*, August 1984, pp.7-22.

[9] Przybylski, S.A., *Cache and Memory Hierarchy Desi8gn: A Performance-Directed Approach*, San Mateo, CA: Morgan Kaufmann Publishers, 1990.

[10] Short, R.T., and Levy, H.M., "A Simulation Study of Two-Level Caches," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988, pp. 81-9.

第 7 章

高效的高速缓存管理技术

前面的章节展示出，为了保持数据一致性，需要如何管理高速缓存。操作系统的设计必须超过这个范围，以一种高效的方式管理高速缓存，以便获得高速缓存的最大性能。本章介绍既能够改善高速缓存性能，又能保持数据一致性的软件技术。

7.1 引 言

一个高速缓存的整体性能是由 3 个因素所决定的：高速缓存的物理设计、在系统上运行的程序的局部引用特性，以及操作系统管理高速缓存的效果。遗憾的是，对于改善局部引用特性很差的程序性能来说，操作系统无能为力。类似地，一旦硬件造好了，那么高速缓存的物理设计在高速缓存行的大小、组的大小、高速缓存的大小等方面都成了固定不变的。但是，在变化的条件下，可以调整操作系统来有效地管理高速缓存。本章将探讨降低高速缓存管理开销、提高高速缓存整体性能的 3 种技术。它们是地址空间布局（address space layout）、延迟高速缓存无效（delayed cache invalidation）和对齐高速缓存的数据结构（cache-aligning data structure）。

7.2 地址空间布局

7.2.1 虚拟索引的高速缓存

每一种 UNIX 系统的实现在一个进程中有 3 个主要区域（正文、数据/bss 和堆栈）的起始地址都是标准的。为了有效地利用高速缓存，均匀地分布数据，在选择这些标准虚拟地址的时候，必须考虑典型的进程地址空间上高速缓存散列算法的影响。对于任何类型的虚拟索引高速缓存来说都是如此。这种影响最好以一个例子来说明。

考虑一个 64K、直接映射、正文和数据高速缓存相结合的高速缓存，每行 16 字节，共 4096 行。这个高速缓存选择使用虚拟地址的“位<15..4>”作为 12 位的行索引。“位<3..0>”选择行内的字节。假定系统上页的大小为 4K。对于本例来说，我们将考虑可能的最小进程，

它的正文、数据和堆栈各有一页。对于第一种情况，假定选择如图 7-1 所示的地址作为标准的区域起始地址。

区域	地址
正文	0x0
数据/bss	0x80000000
堆栈	0xffff0000

图 7-1　第一种情况的虚拟区域起始地址

乍一看，这些地址好像是合理的选择：32 位的地址空间均匀地分配给正文和数据，堆栈从靠近地址空间顶部的地方开始，从而赋予它最大的增长空间（假定在这个系统中，堆栈向着较低的地址增长）。但是，如果我们现在考虑这 3 个虚拟地址将会产生的高速缓存索引，就会出现如图 7-2 所示的不期望的结果。

正文地址	索引
0x0	0
0x10	1
0x20	2
0x30	3
⋮	⋮
0xff0	255

数据地址	索引
0x80000000	0
0x80000010	1
0x80000020	2
0x80000030	3
⋮	⋮
0x80000ff0	255

堆栈地址	索引
0xffff0000	0
0xffff0010	1
0xffff0020	2
0xffff0030	3
⋮	⋮
0xffff0ff0	255

图 7-2　使用图 7-1 中的地址所产生的高速缓存索引

注意，这 3 页中每一页的地址都会索引到同一组高速缓存行：0~255 行。这意味着进程对正文和数据的所有引用都只会索引到高速缓存的前 256 行，而剩余的 3840 行用不上，如图 7-3 所示。

虽然进程正在执行，但是它只能利用全部高速缓存容量的 1/16。来自 3 个区域的所有引用都在竞争同一组高速缓存行，进而因为高速缓存颠簸而导致命中率很低。既然进程只有 12K 大，而且运行在一个有着 64K 高速缓存的系统上，就没有必要出现这种情形。在进程正在执行的同时，如果有必要，高速缓存有着很大的空间，比足够保存整个地址空间所需的容量还要大。

考虑到散列算法的影响，选择不会产生同一组索引的地址，以定义一组新的区域起始地址。例如，可以使用图 7-4 所示的地址。

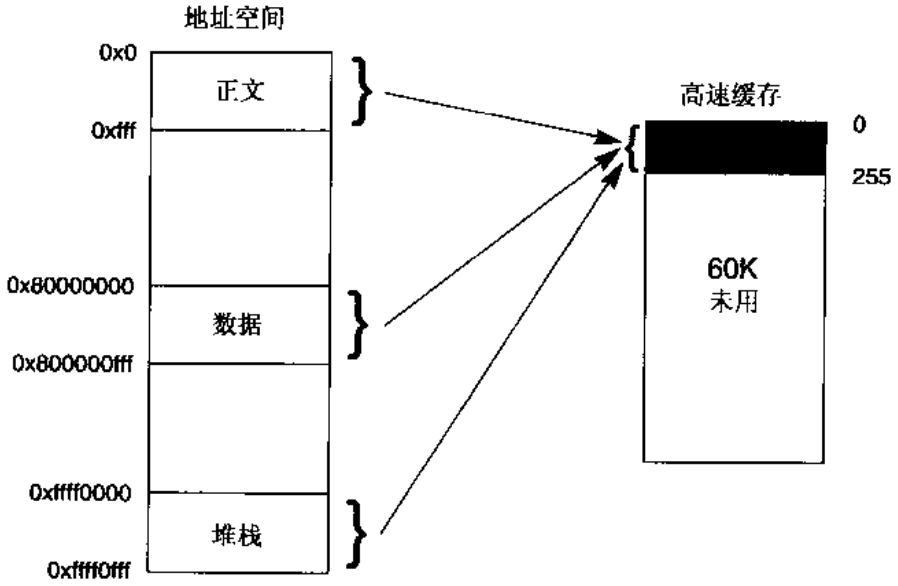


图 7-3 第一种情况的地址空间映射关系

区域	地址
正文	0x0
数据/bss	0x80006000
堆栈	0xfffff000

图 7-4 第二种情况的虚拟区域起始地址

这些地址产生的一组高速缓存索引如图 7-5 所示。选择这些地址以后，来自任何页面的正文或者数据都不会在高速缓存中发生交叠，从而让这个进程最大地利用了高速缓存。观察这样做在图 7-6 中所产生的效果，可以很容易地看出，高速缓存在正文、数据和堆栈中进行了划分。阴影区域标记出一个很小的、仅有 3 个页面的进程的正文、数据和堆栈页在高速缓存中所占据的部分。注意，在正文区域同数据区域所占据的高速缓存行发生交叠之前，它可以有 24K 大。类似地，高速缓存中在 bss 和堆栈区域之间也有 32K 大的空闲部分，供两者增长使用。考虑到这些区域的增长之后，系统就能更好地支持有更大地址空间的进程。因此，选择区域的标准起始地址在很大程度上受到了系统中应用区域预计大小的影响，而且对高速缓存的整体性能有巨大作用。

对于大规模的高速缓存来说，为了充分利用它们的空间，为虚拟索引的高速缓存正确选择区域起始地址这项技术格外重要。其好处随着高速缓存大小的减小而相应降低。对于高速缓存小于最小的进程大小的情况来说，这个优势迅速降低，因为不再完全有可能防止单个进程内的高速缓存争用现象。对于高速缓存小于或者等于系统上页面大小的情况来说，则没有什么作用。此时，所有 3 个区域都会竞争相同的高速缓存行，所以没有什么办法能防止高速缓存颠簸。

正文地址	索引
0x0	0
0x10	1
0x20	2
0x30	3
⋮	⋮
0xff0	255

数据地址	索引
0x80006000	1536
0x80006010	1537
0x80006020	1538
0x80006030	1539
⋮	⋮
0x80006ff0	1791

堆栈地址	索引
0xfffff000	3840
0xfffff010	3841
0xfffff020	3842
0xfffff030	3843
⋮	⋮
0xffffffff0	4095

图 7-5　使用图 7-4 中的地址所产生的高速缓存索引

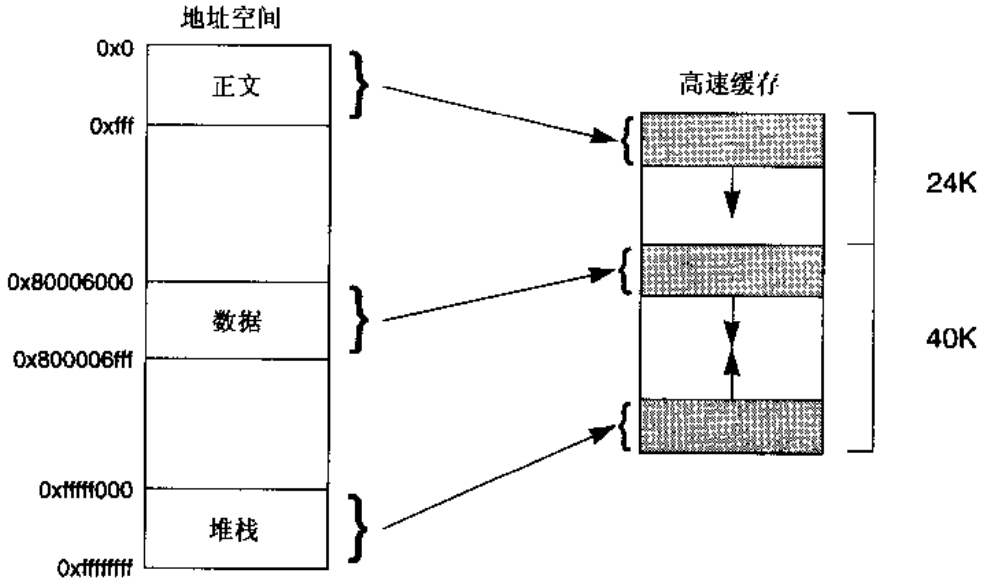


图 7-6　在第二种情况中高速缓存上的地址空间映射关系

7.2.2　动态地址绑定

带有键的虚拟地址和以物理地址标记的虚拟高速缓存都试图同时缓存多个进程的现场。这些类型的高速缓存组成有一个缺点，即不同进程中相同的虚拟地址都索引到同一组高速缓存行上，从而导致在下一个进程运行的时候上一个进程中高速缓存的数据都被替换掉了。在链接时刻，当所有的进程都和同一组标准区域起始地址绑定到一起的时候尤其如此，正如上一小节所介绍的那样。例如，如果运行的两个小进程使用如图 7-7 所示的地址空间布局，那么这两个进程就会竞争高速缓存中相同的 12K 空间。

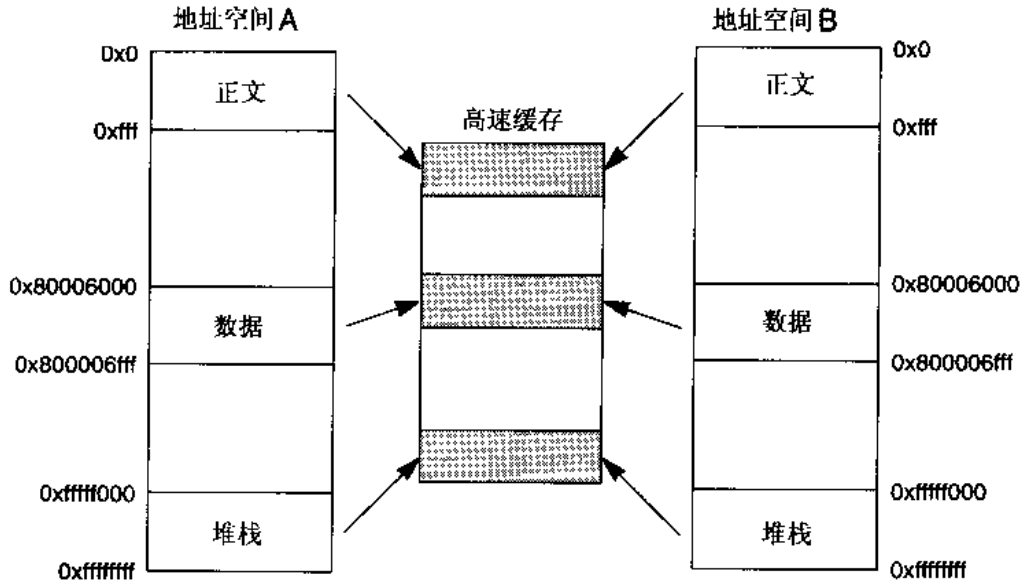


图 7-7 高速缓存上的多重地址空间映射关系

高速缓存再次利用不足，只使用了 64K 高速缓存中的 12K。使用动态绑定机制能够避免这个缺点。

如果进程可以产生的高速缓存索引处于脱节的组内，那么以键或者物理地址标记的大规模虚拟高速缓存就有可能同时缓存来自不同进程的数据。虽然不可能完全防止不同的进程索引到相同的高速缓存行（因为高速缓存要小于进程空间合起来的大小），但是无疑会减少竞争。做到这一点的一种方法是，在链接程序时，把程序随机地绑定到一组区域起始地址之一。类似地，在调用 exec 时也可以随机地选择堆栈区域的地址。这就提高了系统上运行的进程使其数据随机地分布在高速缓存中的概率，从而避免了和其他进程争用高速缓存。不过，这项技术仍然有缺点，即地址绑定关系是在链接时刻静态固定好的，因而阻碍了操作系统动态地进行调整，以适应在一个正在运转的系统上不断变化的程序混合状态。

第二种可能的方法是使用与位置无关的代码。这是指程序在编译或者链接的时候没有绑定到任何特殊的虚拟地址，在这样的程序中没有出现硬编码的地址（没有使用绝对转移或者数据地址）。相反，操作系统可以在任何虚拟地址载入程序，并开始执行它。接下来，程序相对于当前的 PC（程序计数器）或者它被载入的位置来引用其数据和转移目的地。这种方法的优点是，在系统调用 exec 期间，操作系统能够为程序动态地选择虚拟地址。然后，它可以选择一段地址范围，这段地址产生的一组高速缓存行索引和其他正在运行的进程的地址相脱节。这就允许操作系统进行调整，以动态地适应不断变化的进程混合情况。

动态地址绑定的一个缺点是它从程序的执行环境中删除了一个确定性的元素，即固定的区域起始地址。没有了这个确定性的元素，将意味着一个程序必须经受更广泛的测试，以确保它对加载它的虚拟地址没有隐藏的依赖性。第二，这些技术只能在有大规模高速缓存的系统上才能产生可度量的性能增益。较小的高速缓存在任何情况下都不能保存大量进程的局部引用信息，所以计算一个能减少高速缓存行争用的载入地址，所带来的额外开销并不一定划算。最后，不是所有的体系结构都能高效地支持与位置无关的代码。因此，同与位置有关的

代码相比，性能可能还有些降低，所以必须把它和高速缓存性能上可能的提升相对照来进行衡量。

7.2.3　物理索引高速缓存

区域起始地址不会影响物理索引高速缓存的性能，因为在计算高速缓存索引时从不使用虚拟地址。但是，物理索引高速缓存会受到虚拟页映射到的物理页地址的影响。因此，对于操作系统来说，比较合适的做法为尝试分配物理页，从而把物理高速缓存行的竞争降至最低。下面介绍的是一种可能的分配物理页的算法，它能够把对高速缓存的引用分布到高速缓存中，以减少争用的情形。

在采用取模的高速缓存索引机制时，物理存储器的连续页面将映射到高速缓存中连续的位置上。例如，图 7-8 画出了页面大小为 4K 的系统上一块 16K 的高速缓存，它还显示出了物理页面怎样映射到高速缓存上（每个阴影区域代表一页）。

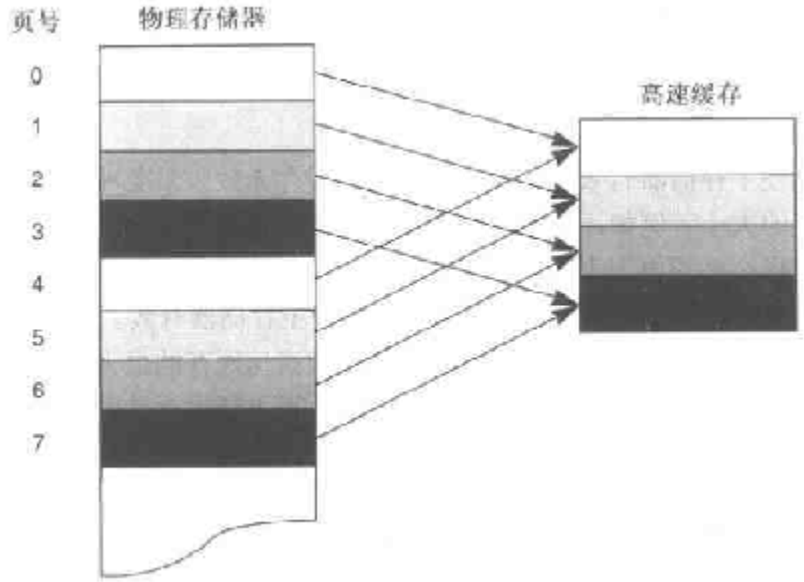


图 7-8　物理页面映射到高速缓存

由此可见，每隔 5 页（例如，图中页号为 4 的物理页面）就“折回”到高速缓存的开头。有相同灰度的物理页面都映射到高速缓存中有对应灰度的部分。不同的灰度经常被称为颜色（color）。于是，我们说每个物理页面都有某种颜色。

将对高速缓存的引用均匀分布的一种直截了当的做法是，将物理页分成 n 组，这里的 n 等于高速缓存大小除以页面大小所得的商（也就是说，高速缓存中的颜色数）。接下来，将索引到高速缓存中相同部分的所有页面（那些有相同颜色的页面）都放入到同一组（也就是说，按照它们的颜色进行保存）。在上面的例子中，n 应该是 4，页 0、4、8、12、16……应该在 0 组，因为它们都索引到高速缓存中第一个 4K 空间。类似地，页 1、5、9、13、17……应该在 1 组，依此类推。不管什么时候分配物理存储器，都要使用循环（round-robin）分配的方法从 4 组中选出一页来。例如，第一页从 0 组分配，第二页从 1

组分配，第三页从2组分配，第四页从3组分配，然后再回到0组、1组等等。这样一来，系统投入使用的页面，其高速缓存索引应该在高速缓存中均匀分布。这种算法几乎不会占用操作系统的开销。

如果操作系统要跟踪已经给一个特殊的进程分配了哪些颜色，那么可以进一步采用这项技术，以便均匀分布每个进程的各个页面，从而获得最好的进程级性能。在理想情况下，一个只有3页（正文、数据和堆栈）的进程会从不同的组分配得到它的每一页，于是，在进程执行的时候，它自己的引用之中不会有任何高速缓存行的争用现象。

本节所讨论的技术可以在大规模的高速缓存上很好地发挥作用，这些高速缓存是页面大小的几倍或者更多。但是，对于小规模的高速缓存来说，它们几乎不会对性能有所改善，因为均匀分配页面的情况只能随机出现。例如，如果高速缓存只有页面大小的两倍，在没有操作系统任何额外操作的情况下，从空闲的页面列表中随机分配出的物理页面正好不和上次分配页面同组的机会只有50%。此外，当高速缓存的大小小于页面大小的话，不会获得任何改善，因为既然只有一种颜色，那么所有的页面都映射到了相同的高速缓存行上。

7.3 受限于高速缓存大小的冲洗操作

不论是为了使主存储器有效，还是为了使高速缓存无效，需要冲洗掉的最大数据量始终受限于高速缓存的大小。例如，如果在一个使用写回高速缓存的系统上，有一个进程请求一次I/O写操作，那么必须先用来自I/O缓冲区且尚在高速缓存中的修改数据（对于没有总线监视机制的所有类型的高速缓存来说，都是如此）使主存储器有效。如果高速缓存有2K大，I/O缓冲区为4K大，那么至多有2K数据（即能够被高速缓存的最大数据量）需要从高速缓存中冲洗掉，以便用来自高速缓存的任何修改过的数据更新主存储器中全部4K大的I/O缓冲区。对于所有类型的高速缓存以及在所有的情况下，高速缓存的冲洗操作都受限于高速缓存的大小，利用这一事实，就可以大大节省冲洗操作的开销。对于小规模的高速缓存来说尤其如此，因为举例来说，大量的I/O操作在逻辑上需要冲洗的数据总量，或者在一个大型进程退出时需要冲洗的数据总量都会轻易地超过高速缓存的大小。下面将会看到，可以进一步增强这项技术。

7.4 滞后的高速缓存无效操作

正如在前面章节中所讨论的那样，在许多可能的情况下，操作系统都必须使高速缓存无效，以保持数据的一致性。虽然有些高速缓存的无效操作必须立即完成（以保持I/O一致性或者防止出现用户-内核歧义），但是如果有特殊的高速缓存实现能保证不引用不一致或者过时的数据，那么就可以推迟其他类型的无效操作。例如，当一个系统上的进程退出的时候，如果这个系统使用带有键的虚拟高速缓存，那么只要其他进程使用不同的键，就不会有哪个进程能命中这个死去进程的过时数据。因此，在调用exit的时候，没有必要使高速缓存无效。

由于频繁的高速缓存无效操作会大大降低系统性能，所以推迟高速缓存的无效操作是一个很重要的考虑方面。性能降低的部分原因是由于无效操作本身的开销。高速缓存的无效操作很花时间，它最少也需要一次写高速缓存标记的操作，以此来使高速缓存行无效。在典型情况下，对于要使之无效的每一行高速缓存来说，都必须重复执行这项操作，因为几乎没有哪种高速缓存体系结构能一次冲洗一行以上的高速缓存。

系统性能降低的另一部分原因是，试图使某些高速缓存行无效，而这些行却没有包含操作系统要尝试删除的数据。典型情况下，在给定的任何时刻，一个进程的地址空间中只有一部分（它的局部引用）在高速缓存中。遗憾的是，操作系统没有办法知道驻留在高速缓存中的是哪些部分。因此，例如，当调用 exit 要使高速缓存无效的时候，操作系统必须假定最糟糕的情况，于是要冲洗掉进程所使用的整个地址范围。于是，如果在页面大小为 4K 的系统上使用 16K 高速缓存，并且一个仅有 3 页（正文、数据和堆栈）的小进程退出，那么操作系统不得不使整个 12K 的地址空间都无效，以确保从高速缓存中删除进程的所有数据。幸运的是，几乎所有的高速缓存体系结构在真正冲洗一行高速缓存之前都会检查是否命中。这就防止了无效操作删除其他进程的数据，但还是需要对 12K 数据逐行进行检查。

一种改进方式为，在那些高速缓存的体系结构能够防止引用过时数据的情况下推迟无效操作，以后在一次冲洗操作中清除过时数据，其好处显而易见。在采用带有物理标记的 12K 虚拟高速缓存且页面大小为 4K 的情况下，即使是最小进程，每次调用 exit 的时候也不得不使 12K 高速缓存无效。如果有 10 个进程依次退出，那么就会致使操作系统不得不从高速缓存中冲洗掉 120K 数据。如果操作系统等着直到第 10 个进程退出，它就只需要一次使 12K 高速缓存无效即可，而且也能保证死去进程的全部数据都被删除掉了。这就节省了 10 次 exit 调用中 90%的无效操作开销。这项技术的确切实现取决于高速缓存的组织结构，但是一般不贵。

滞后的高速缓存无效操作不能用于虚拟标记的高速缓存（参见习题 7.9）。此外，带有总线监视机制的物理高速缓存不需要任何这样的技术，因为它们从来都不需要任何冲洗操作（参考 6.2.6 小节）。

7.4.1　带有键的虚拟高速缓存

对于带有键的虚拟高速缓存来说，系统调用 exit 的无效操作可以推迟到系统用完所有可用的空闲键，需要重用死去的进程所关联的键时再执行。操作系统可以只维护两个键列表：不含与进程关联的高速缓存过时数据的未用键，以及死去进程用过的键。当创建新进程的时候，从第一个列表中取得一个键。当那个列表变空而第二个列表中有键，且使用了写直通高速缓存机制的时候，操作系统只要使整个高速缓存无效，就可以把第二个列表中的内容都移入第一个列表。于是，系统就有了一组新的空闲键供使用。

因为操作系统不一定使正在运行的进程中被修改过的数据无效，所以写回高速缓存机制的过程就变复杂了。这就需要有一种途径，既能使得与过时键相关的数据无效，同时又不会影响其他数据（其中的细节取决于特殊的高速缓存实现）。不能简单地写回数据，然后让高速缓存中的数据都无效，因为与过时项相关联的物理页面可能已经重新分配出去了。一种可

能的方法是改变 MMU 中的映射关系，从而使任何过时键所关联的所有虚拟页面都映射到存储器中未用的物理页面上。此刻，可以将整个高速缓存写回，并使之无效。虽然没必要写回过时数据，但写回操作也无害，因为对应的物理页面并没有投入使用。无论如何，如果两个列表中有一个没有键了，那么为了把一个键从一个正在运行的进程重新分配给另一个进程，仍然必须冲洗高速缓存（如果它采用写回策略，则使主存储器有效）。

7.4.2 没有总线监视机制的物理标记高速缓存

对于带有物理标记的虚拟高速缓存以及没有总线监视机制的纯物理高速缓存来说，操作系统需要维护两个物理存储器的空闲列表（类似于前一小节所提到的键的两个列表），一个包含没有高速缓存过时项的物理页面（清洁列表，clean list），一个用于那些可能有过时项的物理页面（脏列表，dirty list）。在进程调用 exit 退出的时候，它们的物理页面就会被放到脏列表中。当需要为某些新用途分配物理存储器的时候，从干净列表中得到它。如果清洁列表为空，而在脏列表中有页面，那么执行一次高速缓存无效操作（如果使用写直通高速缓存机制的话），就可以把脏列表中所有的页面都移入清洁列表。

对于这两种高速缓存组织结构来说，写回高速缓存机制处理起来要容易一些，因为已知没有投入使用的物理页面。因此，无需进一步的测量就可以写回整个高速缓存，且使之无效。和前面一样，这也会把过时数据写入主存储器，虽不必要但也没有害处，因为这些页面都没有使用。

因为对于带有物理标记的虚拟高速缓存以及纯物理高速缓存来说，每次对缓存的操作 MMU 都要验证存储器的访问，所以滞后的高速缓存无效操作可以不仅限于在 exit 期间使用。在回收存储器的任何时候（通过缩小 bss 的系统调用 sbrk、剥离共享内存等），被释放的物理页面都可以放入到脏列表中。只要没有哪个页表包含到已回收页面的有效映射，那么任何进程都不可能访问过时的内容。这就给这些系统带来了对性能的额外提升作用，而当使用虚拟标记的高速缓存时是不可能有这样的性能提升的。

7.5 按高速缓存对齐数据结构

最后一种使高速缓存的性能最大化的技术是在存储器中按高速缓存对齐数据结构。这里要对一个程序中的数据结构，可能就是程序本身进行修改，以提高高速缓存保存数据的效率。其目标是布置好程序的数据，从而让频繁访问的数据结构很好地与高速缓存行相吻合，从而把不能命中高速缓存的情况减至最少，并且使程序的局部引用更为紧凑。虽然这样的修改不一定能移植，因为对高速缓存行大小的了解必须“嵌入”到程序当中去，所以在目标代码的可移植性没有程序的性能重要的场合里，它们就能够提高性能。操作系统本身就是这类程序的一个良好例证。

为了看到这种做法是怎样有益于提高高速缓存性能的，最好把存储器想成是高速缓存行的一个数组，如图 7-9 所示。

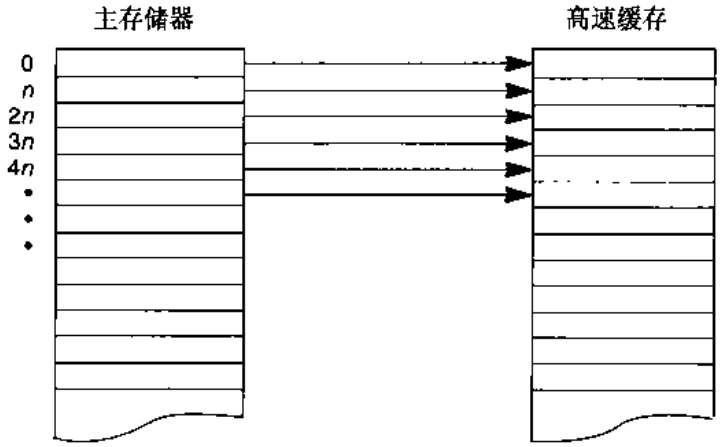


图 7-9　将主存储器看成是高速缓存行的一个数组

如果使用的物理高速缓存每行的大小为 n，那么从物理地址 0 开始的第一组 n 字节就映射到高速缓存中的第一行。接下来的 n 字节映射到下面一行高速缓存，以此类推。于是，从物理地址 0、n、2n、3n 等等起始的 n 字节，每一组都正好装满一行高速缓存（注意，当使用虚拟地址和虚拟索引的高速缓存时情况也是一样）。知道了这一点，那么现在就可以在存储器中放置数据结构，使它们尽可能高效地与高速缓存行相配合。

例如，在 UNIX 操作系统中访问最频繁的数据结构之一是进程表（process table）。进程表是一个结构数组，每个结构对应于系统中的一个进程，其中包含有诸如进程 ID（process ID）、用户 ID（user ID）和组 ID（group ID）这样的信息。如果高速缓存行比一个进程表项大，而又比两个进程表项小，那么就可以按照图 7-10 中的样子把进程表项的数组布置到高速缓存中。

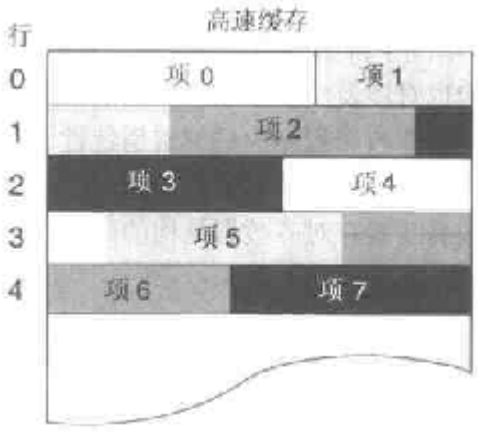


图 7-10　进程表项在高速缓存中的布局

注意进程表项 1、3、4 和 6 如何跨越两行高速缓存，而其余的进程表项则完全在一行之中。在单个进程表项所代表的进程正在执行的同时，该表项内有高度的局部性。因此，我们希望在那些过程中整个进程表项能够驻留在高速缓存里。图 7-10 中的 0、2、5 和 7 项都能够以一次高速缓存缺失而读入高速缓存。1、3、4 和 6 项则会产生两次缺失，而且会占用两行

高速缓存。多出来的缺失会影响系统的性能，因为在进程表项内有高度的局部性。此外，高速缓存其他的进程表项几乎不会获得什么性能上的好处，因为进程在运行的时候，一般只会访问它们自己的进程表项。就像 1、3、4 和 6 项的情况所体现出来的那样，让每一项只占一行而不是两行高速缓存更可取一些。达到上述目的的一种方法是把每个进程表项填充到和高速缓存行一样大（在高速缓存行比一个数据结构大，而又比两个数据结构小的情况下）。这样做形成的布局如图 7-11 所示。

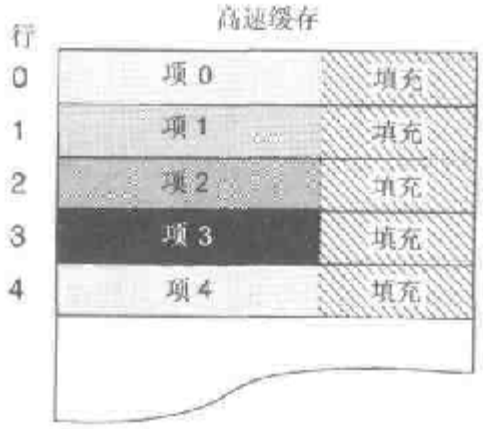


图 7-11 带有填充的进程表项在高速缓存中的布局

现在每一个进程表项都只占一行高速缓存，这意味着读入完整一项仅仅需要一次高速缓存缺失处理。当然，所做的折衷就是每项末尾的填充浪费了存储器。还要说明的一点是，在任何时刻，高速缓存中所容纳的进程表项也更少了。当决定是否使用这项技术的时候，必须要考虑这些因素。如果填充非常小，那么就很值得做。但是，随着填充不断地增加，浪费的高速缓存空间也就越多，而且可能会让性能降低。类似地，在多个进程表项都是局部引用一部分（比如一个指向散列斗（hash bucket）的指针数组）的情况下，在数据结构上使用填充方法也会降低性能。在每一种情况下都必须运行基准测试程序（benchmark）来确定最优方法。

按高速缓存对齐数据结构也对涉及到多个数据结构的情形有所帮助。通过把相关的数据一起放入存储器的做法，人们就能改善程序的局部引用特性，而且令其所需的高速缓存行更少（因此发生的高速缓存缺失也更少），从而高效地运行。正如我们将在第三部分里看到的那样，在多处理机系统中，按高速缓存对齐数据结构的好处更多。

7.6 小 结

本章展示了高效管理高速缓存的几种技术。地址空间的布局之所以重要，是因为它影响到了高速缓存的利用率和高速缓存行的竞争。针对虚拟索引的高速缓存，仔细选择区域起始地址，以及针对物理索引的高速缓存，仔细选择物理页面以均匀分布颜色，就能够提高高速缓存的性能。因为所有的高速缓存冲洗操作都受限于高速缓存的大小，所以就可以采用推迟高速缓存无效操作的技术来优化清除过时高速缓存数据的任务。最后，按照高速缓存对齐数

据结构，通过整合用于特殊局部引用的数据，使其正好和一行高速缓存相吻合，就可以有助于降低出现高速缓存缺失的情形。只要有可能改善高速缓存的性能，所有这些技术都应该使用，尤其是所有这些技术实现起来都很容易，就更要如此了。

7.7 习　　题

7.1　一个系统上独立的指令和数据高速缓存如下：数据高速缓存是 16K 大、带有物理标记的 4 路组相联虚拟高速缓存，高速缓存行大小为 16 字节；指令高速缓存是 8K 大、带有物理标记的直接映射虚拟高速缓存。系统使用 4K 大小的页面，并且以虚拟地址的“位<11..4>”索引数据高速缓存。指令高速缓存则以“位<10..4>”来进行索引。如果对于一般的应用来说，正文段介于 4K 和 16K 之间，数据段介于 4K 和 8K 之间，而堆栈段介于 8K 和 16K 之间（堆栈向下增长），那么为正文、数据和堆栈选择标准的区域起始虚拟地址。解释你做出选择的原因。

7.2　针对带有如下高速缓存的系统重做上一题：一个 16K 直接映射虚拟数据高速缓存，以及一个 20K 直接映射物理指令高速缓存。两者都采用每行 16 字节。

7.3　针对带有如下高速缓存的系统重做 7.1 题：一个 8K 双路组相联统一的主高速缓存和一个 4M 直接映射物理的次级高速缓存。两者都采用每行 32 字节。

7.4　为什么 7.2.2 小节所描述的技术不能运用到纯虚拟和物理高速缓存上？

7.5　如果页面大小为 4K，那么一个每行 8 字节的 16K4 路组相联高速缓存包含多少种颜色？

7.6　MIPS R4000 能够支持一个 4M 直接映射外部物理高速缓存，每行 128 字节。使用 7.2.3 小节里的算法，如果页面大小为 8K，那么内核应该维护多少组物理页面？

7.7　使用 6.3.1 小节中的多级高速缓存组织结构以 7.3 节中介绍的技术，当一个进程剥离一个 1M 大的共享存储区时，必须冲洗多少数据？片上指令高速缓存大小为 4K，数据高速缓存大小为 8K。假定外部高速缓存有 256K。

7.8　解释为什么不能在与没有使用总线监视机制的 DMA 读操作相关的 I/O 缓冲区上使用滞后的高速缓存无效操作？

7.9　解释为什么不能在纯虚拟高速缓存上使用滞后的高速缓存无效操作？

7.10　在 7.4 节的最后提到了可以扩展滞后高速缓存无效操作的范围，把诸如剥离共享内存和缩小 bss 段的 sbrk 调用这样的情况包括进来。解释为什么滞后的高速缓存无效操作不能在采用了带有键的虚拟高速缓存的情况下使用。

7.11　对于这样一个系统，它有带物理标记的主虚拟高速缓存和次级物理高速缓存，那么应该对 7.4.2 小节里所介绍的推迟高速缓存无效操作算法进行怎样的修改？

7.12　如果高速缓存行的大小为 128 字节，现有的数据结构大小为 60 字节，那么应该给数组中的每个数据结构增加多少填充（以字节为单位）？假定和进程表项的情况一样，局部引用一次只包括一个数组元素。解释你给出的答案的理由。

7.13　重做上一题，此刻每行的大小为 32 字节，现有数据结构的大小为 120 字节。

7.8 进一步的读物

[1] Gupta, R., and Chi, C., "Improving Instruction Cache Behavior by Reducing Pollution," *Proceedings of Supercomputer '90*, pp. 82-91.

[2] Lynch, N.L., Bray, B.K., and Flynn, M.J., "The Effect of Page Allocation on Caches," *SIGMICRO Newsletter*, Vol. 23, No. 1-2, December 1992, pp. 222-5.

[3] McFarling, S., "Program Optimiztion for Instruction Caches," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 183-91.

[4] Mogul, J.C., and Borg, A., "The Effect of Context Switches on Cache Performance," *Computer Architecture News*, Vol. 19, No. 2, April 1991, pp. 75-84.

[5] Thiebaut, D.F., and Stone, H.S., "Footprints in the Cache," *ACM Transactions on Computer Systems,* Vol. 5, No. 4, November 1987, pp. 305-29.

[6] Thompson, J.G., "Efficient Analysis of Caching Systems," Technical Report UCB/CSD 87/374, Computer Science Division, University of California, Berkeley, October 1987.

第二部分

多处理机系统

第8章 多处理机系统概述

本章介绍紧密耦合、共享存储的对称多处理机（symmetric multiprocessor），它将成为本书余下部分的研究焦点。这是UNIX系统最常采用的多处理机类型，因为它将标准单处理机UNIX内核实现所假定的执行环境并行化了。接下来的各小节介绍它的组成，从而为后面的各章做好准备，后面的各章将要研究如何调整UNIX操作系统，使之适合于在这类硬件上运行。本章从详细描述存储器模型（memory model）入手，接下来介绍了在这些系统上提供互斥机制（mutual exclusion）的问题，并解释了如何在单处理机内核实现上解决这些问题。本书这一部分所介绍的所有多处理机系统在运行时都不带高速缓存，从而更好地展示出多处理机操作系统必须解决的基本问题。多处理机高速缓存机制将在第三部分中详细介绍。

8.1 引 言

用户一直在追求速度更快、更经济的计算机系统。设计人员能够满足这种需求的一种方法是，通过制造速度更快的单CPU，从而提高单位时间内能够完成的处理量。这种方法的缺点是，一旦突破了CPU性能上的某个阈值，硬件和开发成本的增长速度就会大大超过由此在CPU速度上带来的提高。制造速度非常快的CPU需要在许多权衡因素之间进行平衡。例如，高速硅晶片技术要有较低的元件密度（component density），对信号同步（signal synchronization）和传输延迟（propagation relay）的限制日益重要，它还需要更大的功率，会发出更多的热量（有时甚至需要液体致冷）。正因为有这些困难之处，所以设计人员经常会求助于多处理机，将其作为增加计算机系统整体性能的一种选择。

多处理机（multiprocessor，MP）是由两个或者两个以上的CPU组合成的单个计算机系统所构成的。采用多处理机的方法以后，设计人员代之以采用多个CPU来缓解制造更高速度CPU的需要，于是就可以把工作量分布到所有的CPU上。如果我们把一个单处理机（uniprocessor，UP）系统和一个采用相同CPU的多处理机系统进行比较，就会发现在执行任何单一任务的时候，多处理机一般不会比单处理机速度更快，因为CPU的速度是相同的，但是它可以在单位时间内并行执行更多的任务。这是多处理机的主要诱人之处：在单位时间内执行多项任务，可以使用比尝试制造能够在相同时间内处理相同任务量的单处理机更为经济的CPU技术。此外，可以重新编写一些应用来利用一个MP系统所固有的并行特性。可以将应用划分成一组相互配合的子系统，其中每一个子系统都在不同的处理器上执行。在这种情

况下，可以减少运行这一应用所需的时间。例如，科学应用往往就是由一些执行并行化功能的编译器自动地以这样的方式进行细分的。

从立足于市场的观点来看，多处理机机制也具有优势。多处理机系统可以通过调整 CPU 的数量来进行扩展，从而与应用环境相匹配。例如，对于终端用户和客户来说，从单处理机系统或者双处理机系统起步，然后随着其计算需求的扩大而增加 CPU 数量来升级系统，就是一种很吸引人的方案。此外，还有可能提高系统的可用性。如果一个 CPU 发生故障，那么剩余的 CPU 就可以继续发挥作用（取决于系统的设计），从而保证了系统的可用性，只是性能会有所降低。这就提供了一定程度的容错（fault tolerance）能力，在诸如在线交易处理这样的环境中，系统停机会造成收入上的损失，就需要容错能力

8.1.1 MP 操作系统

多处理机操作系统的设计必须要协调好所有 CPU 上同时发生的活动，这是一项比管理单处理机系统更为复杂的任务。正如将会在后面几章中看到的那样，调整单处理机 UNIX 内核实现，使之可以在一个 MP 系统上运行，所需要的修改程度变化范围很大。但是，每一种实现都必须解决 3 个主要领域的问题：系统完整性（system integrity）、性能（performance）和外部编程模型（external programming model）。

所有的 MP 内核实现都必须保持系统的完整性。这意味着要正确地协调好 CPU 的并行活动，从而避免危害到内核的数据结构。这就保证了系统在所有可能的情况下，不管外部事件以及系统中各 CPU 活动的时间前后如何，都能正常地发挥作用（参见 8.4 节可以了解对这些问题进一步的讨论）。

一旦有了完整性，那么就可以对实现进行修改和调优，以获得最大的性能。虽然有许多种不同的途径能够获得一个满足系统完整性要求的 MP 内核，但是在如何高效地管理 CPU 并因此影响 MP 系统的整体性能方面，可以采用的不同技术五花八门。接下来的几章内容将突出介绍几种不同的 MP 实现，它们在性能上有很大的不同。

第三个因素，外部编程模型决定了有多个 CPU 会对系统调用接口（它是应用程序接口（application program interface）的一部分）产生什么样的影响。MP 系统的操作系统设计人员不得不就是否将 MP 系统“伪装”成一个 UP 系统的样子而做出选择。如果系统调用接口和 UP 的系统调用接口兼容，那么已有的单处理机应用程序无需修改就能在 MP 上运行。另一方面，如果系统调用接口不兼容，那么程序就不得不在明确知道系统中有多个 CPU 的情况下进行编写，而且可能需要使用特殊的系统调用来和运行在其他处理器上的进程进行通信，或者向其传送数据。

例如，考虑 UNIX 的进程 ID 号（process ID number）。在一个单处理机系统上，任何进程都能够以这个进程 ID 号来引用任何其他进程。一个没有保持单处理机系统调用接口的 MP 内核实现，为了引用运行在另一个处理器上的进程，除了需要进程 ID 号外，可能还要给出 CPU 号。这种方法的缺点是它丧失了应用的可移植性。

因为重新编写符合新系统调用接口的程序代价很高，所以大多数实现都选择保持单处理机系统调用接口，于是有多个 CPU 的情况就完全透明了（即看不到了）。这是本书中给予考虑的唯一一种实现类型。这并不是说不允许操作系统提供新接口来让程序员利用 MP 上固有的并行性；这意味着内核必须提供所有业已成为标准的单处理机系统调用接口和设施。

在探讨为了让 UNIX 系统能够在 MP 上运行而需要进行的修改之前，了解多处理机的体系结构以便看到对操作系统的影响是很重要的。

8.2　紧密耦合、共享存储的对称多处理机

对于本书接下来的内容来说，感兴趣的多处理机体系结构是紧密耦合、共享存储、对称多处理机，经常把它简称为 SMP（因为 SMP 是在本书中所介绍的唯一一种多处理机类型，所以将会互换使用 SMP 和 MP 这两个词）。如前所述，这是最常用的一种 MP 的类型，因为它很容易就可以支持能够保持单处理机外部编程模型的操作系统实现。图 8-1 给出的逻辑框图是一个有 4 个 CPU 的系统。

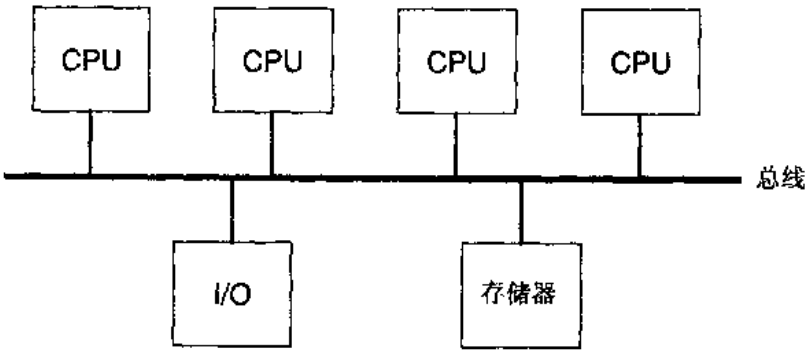


图 8-1　SMP 框图的例子

对于这种类型的多处理机体系结构来说，要了解几个要点。第一，所有的 CPU、存储器和 I/O 都是紧密耦合的。做到紧密耦合有几种方法，但是最简单也是最常用的方法就是用一个公共高速总线把所有的单元都直接互连起来（这也是本书重点讨论的互连类型）。回忆 6.2.6 小节的内容，那里介绍的一条总线是基于广播的通信介质，它能够一次传输一个字或者更多个字的数据，这就可以让连接到总线上的任何单元和任何其他单元进行高速通信。因为总线高速并行传输许多位数据，所以它的长度是非常有限的。紧密耦合的另一个含义是指所有的部件彼此都在很短的距离之内（通常是在同一个物理机柜中）这一事实。

从图 8-1 中可以很容易地看出共享存储的特点：所有 CPU 和 I/O 设备共用单独一个能够全局访问到的存储模块。CPU 本身没有局部存储器（local memory）（高速缓存有可能除外），它们把其全部的程序指令和数据都保存在全局共享存储器（global shared memory）中。这里最重要的一点在于，一个 CPU 保存到主存储器中的数据，所有其他 CPU 都可以立即访问到。在第三部分中我们将会看到，每个 CPU 都有自己的局部高速缓存（local cache），但没有其他的非共享存储器（non-shared memory）。

SMP 体系结构中最后一个重要的方面是存储器访问是对称的，这意味着所有的 CPU 和 I/O 设备都能平等地访问全局共享存储器。存储器中的内容是完全共享的，只要是引用相同的数据，那么所有的 CPU 和设备都使用相同的物理地址。对总线和存储器的访问要进行仲裁，从而保证所有的 CPU 和设备都能有平等的访问权利。此外，它们的访问不会彼此相互干扰。

例如，没有必要让运行在一个 CPU 上的程序去关心另一个 CPU 上正在运行的、读写存储器不同部分的程序。多个 CPU 对存储器相同部分的同时访问要进行仲裁，以便让它们不会彼此干扰（这将在 8.3 节中详细讨论）。

注意，I/O 设备由于其位置的原因，也会以一种对称的方式由所有 CPU 共享。这就让所有的 CPU 可以根据需要发起 I/O 操作。还要注意，出于 DMA 的目的，I/O 设备仍然能够完整地访问全部存储器。这意味着 CPU 写的任何数据都可以借助 DMA 被发送到 I/O 设备，而且所有的 CPU 都能访问从设备到存储器执行 DMA 操作的结果。因为 I/O 设备是通过同样的共享总线来访问存储器的，所以上面有关 CPU 如何对存储器进行访问的所有讨论都可以直接运用到基于 DMA 的 I/O 设备上。

现在很容易看出，SMP 的体系结构是 6.2.6 小节里介绍的 UP 系统结构在逻辑上的一种扩展：两者的总线、存储器和 I/O 结构都是相同的。通过增加更多的 CPU，就把 UP 系统转变成了 SMP 系统，仿佛是给系统增加了额外的 I/O 卡一样。

在 SMP 系统中可以投入实际工作的最大 CPU 数目受到共享总线和存储器的带宽限制。这个带宽在设计系统的时候就固定下来了，它必须足以满足系统中所有 CPU 和 I/O 设备的需要，否则系统的整体性能就会受损。例如，如果总线和主存储器的带宽为 20Mb/s，而 I/O 设备需要在一个给定的应用环境中执行 5Mb/s 的 DMA 传输，那么这就给 CPU 剩下了 15Mb/s 的可用带宽。如果每个 CPU 要连续执行指令而没有延迟，需要 3Mb/s 的带宽，那么系统就最多支持 5 个 CPU。如果增加的 CPU 数量超过了这个限度，对存储的请求总量就超过了总线和主存储器能够提供的数量，从而导致请求被拖延。如果我们考虑 CPU 从主存储器取得指令的情况，就会很清楚地看到，任何延迟都会让 CPU 执行指令的速度更慢，因为它在等候存储器提供下一条指令的时候是空闲的。结果，多加的 CPU 并不会提高系统的性能。

在高性能系统的设计中，设计 SMP 系统的总线和主存储器子系统能够向 CPU 和 I/O 设备提供适当的带宽是一个很重要的方面。然而从操作系统的角度来看，它对保持单处理机外部编程模型时所面临的问题没有影响，也对确保操作系统自身正确发挥作用没有影响。因此，后面几章的内容集中讨论 SMP 系统给操作系统带来的问题。

8.3 MP 存储器模型

MP 系统的存储器模型定义了 CPU 和运行在它们上面的程序如何访问主存储器，以及其他 CPU 的同时访问会给它们造成怎样的影响。既然采用物理地址访问主存储器，那么虚拟地址转换的影响，包括在使用一个没有映射的虚拟地址时可能发生的任何异常在内，都不会当作是存储器模型的一部分来考虑，而是由单个 CPU 处理这些活动。相反，存储器模型关注 CPU 和主存储器之间物理地址和数据的传输。

不同的 MP 硬件系统可能会实现不同的存储器模型。操作系统程序员必须透彻地理解一种特殊机器的模型，以便让编写出的操作系统能够正确地运行，而且能保证系统的完整性。不同机器上的存储器模型之间主要的不同之处集中在硬件如何确定 load（上载）和 store（保存）指令的执行顺序上。改变执行的顺序是为了改善性能。存储器模型也确定了多个处理器同时访问

相同的存储器位置时该如何进行处理。从 CPU 的角度来看，为了降低成本或者提高速度而选择的总线仲裁策略和（或）主存储子系统的不同实现，是改变存储器模型的主要因素。

SMP 的存储器模型包括上一小节提到的那些特性：存储器是全局访问的、紧密耦合的，也是对称的。于是，模型确定了 load-store 的顺序，以及多个 CPU 同时访问存储器的影响（在继续阅读下去之前，读者可能会希望复习一下 6.2.6 小节中就系统总线进行的介绍性讨论）。

8.3.1　顺序存储模型

最简单和最常用的存储器模型是顺序存储模型（sequential memory model，也称为强定序（strong ordering）模型）。在这种模型中，CPU 是按照程序次序（program order）来执行所有的 load 和 store 指令的，即按照它们在程序的顺序指令流中出现的次序来执行的。而且从主存储子系统和其他 CPU 的角度来看，load 和 store 指令也是顺序对主存储器产生影响的。Motorola MC68000 和 MC88000 系列处理器、Intel 80X86 系列处理器以及 MIPS 系列的 RISC 处理器都采用了这种模型。与此相对照，诸如 SPARC v8（version 8）这样的体系结构把 store 指令所关联的数据发送给主存储器的次序和 store 指令的执行次序不一样，它是非顺序的，因为读取同一数据的另一个 CPU 所看到的主存储器内容的变化将不会和保存数据的程序执行指令的次序相同（其他的存储器模型和它们对操作系统的影响将在第 13 章中探讨）。

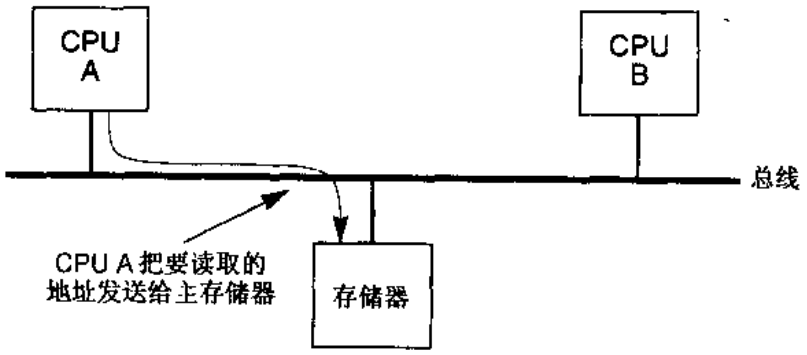
除了定义存储器操作是以程序次序来执行的，顺序存储模型还规定它们是原子的，就像前两节中所描述的那样。为了简单起见，以及因为其常用性，本书在接下来的几章里着重讨论顺序存储模型。所有的例子和讨论都假定使用了这个模型，而且进一步假定编译器（compiler）或者优化器（optimizer）不能改变程序中指令的次序。8.3.2 和 8.3.3 两小节将介绍为了理解这一简化的存储器模型对操作系统的影响而需要知道的核心内容。大多数系统都采用了各种不同的优化措施来改善性能，同时又不会影响到软件。这些都超出了本书的范围。

8.3.2　原子读和原子写

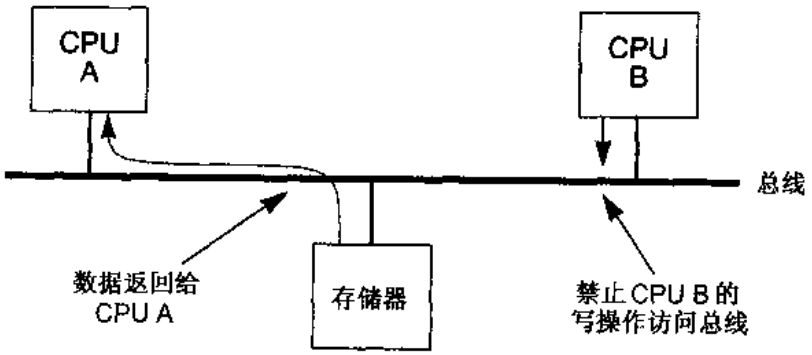
顺序存储模型定义从 CPU 或者 I/O 设备（通过 DMA）到主存储器的单次读或者写操作为原子读或者写操作。这样的一次操作一旦开始，就不能被系统上来自 CPU 或者 I/O 设备的任何其他存储器操作所中断，或者受到它们的干扰（不考虑诸如“split read”这样的能并行执行多个总线操作的特殊硬件实现技术，因为它们对于操作系统来说是透明的）。由于访问主存储器必须使用一条共享总线，所以轻而易举就能保证存储器操作的原子性（atomicity）。一次只能有一个 CPU（或者 I/O 设备）使用总线。如果多个 CPU 同时希望访问存储器，那么在总线上的特殊硬件要在多个请求方（requestor）之间进行仲裁，以确定接下来允许哪一个来使用总线。在选择了一个请求方之后，就将总线授权给它，并且允许那个 CPU 完成一次原子读或者写操作，这个操作涉及到了主存储器中一个或者更多在物理上连续的字。在这一次操作期间，禁止所有其他的 CPU 和 I/O 设备执行任何它们自己的读操作或者写操作。在完成每一次操作之后，就会重复这个周期：对总线进行仲裁，并授权给另一个 CPU。选择接下来该由哪个 CPU 获得总线，可以使用先入先出（first-in-first-out，FIFO）、循环（round-robin）或者在总线硬件中实现的任何其他类型的调度机制。最后，所有希望访问主存储器的 CPU 都

会获得一个周期，不会让任何 CPU 永远访问不到存储器（要记住一点，SMP 体系结构指定了所有的 CPU 都能平等地访问到主存储器）。I/O 设备的总线仲裁机制和 CPU 是一样的。

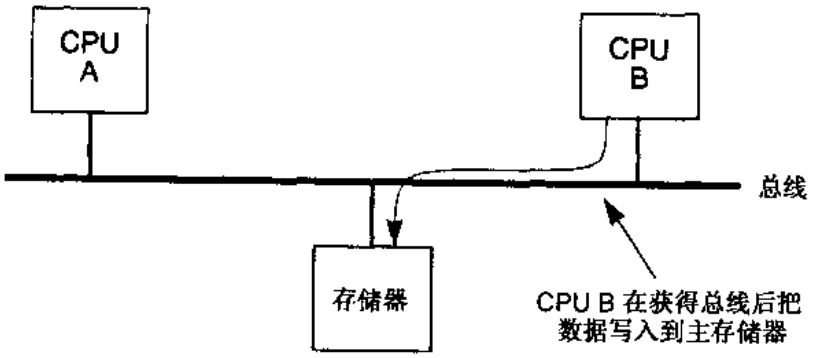
图 8-2 给出了一个带有两个处理器的 SMP 系统（为了简单起见，省略了 I/O）。假定处理器 A 希望从主存储器读数据。如果总线处于空闲状态，那么就会允许它立即开始它的总线周期，把要读的数据的地址发送给主存储器（参见图 8-2 中(a)图）。如果 CPU B 尝试在数据正返回给 CPU A 的同时开始一次写操作，那么就会禁止它的操作，直到 A 完成它的周期为止（图(b)）。一旦 A 的周期完成，总线仲裁器就会把总线授权给 B，于是允许 B 完成它的存储器操作（图(c)）。



(a) CPU A 开始读总线周期



(b) 在 CPU B 等待的时候存储器把数据返回给 CPU A



(c) CPU B 占用总线执行它的存储器操作

图 8-2 总线仲裁的例子

在一次操作中能够传输的数据量被限制在能防止一个处理器阻塞总线的数量之内。虽然实际机器中的典型传输量往往等于高速缓存行或者子行的大小，但是操作系统是直接看不到传输量的大小的。既然现在只考虑没有高速缓存的 MP 系统，那么为了简单起见，假定传输量为一个字。

由此可见，在等待总线被授权给 CPU 的时候会引入延迟。从软件的角度来看，如果一条访问存储器的指令在等候轮到它占用总线的时候被拖延了，那么这条指令似乎仅仅是花了更多一些的时间执行而已。这就是为什么对于总线和存储器来说，其带宽要大于或者等于所有 CPU 和 I/O 设备能够产生的存储器流量之和很重要的原因。如果不是如此，那么 CPU 就不能以峰值性能来执行，因为它们在试图访问存储器的时候会被频繁地延迟。无论如何，总线引发的延迟对于运行在系统上的软件来说都是透明的（除了性能之外）。

对于前面的定义，有一点很清楚，那就是如果每个 CPU 只访问主存储器的一部分，这部分存储器不但是唯一的，而且和其他 CPU 所访问的部分无关，那么从软件的角度来看，每个 CPU 就会像它是系统上唯一一个 CPU 那样来执行。多个 CPU 访问相同的共享主存储器单元的事实是不确切的，因为每个 CPU 只访问了其他处理器从不访问的位置（这又再次显示出 SMP 如何成为 UP 系统模型在逻辑上的一个扩展）。现在，我们必须考虑多个 CPU 同时访问相同位置所带来的影响。

在顺序存储模型中，哪怕是多个 CPU 同时访问主存储器内的同一个字，也要保证是原子操作。例如，如果系统内所有的 CPU 同时发出了写同一个字的指令，那么总线仲裁硬件会禁止除了一个 CPU 之外所有处理器的写操作。这个被选出的 CPU 占用总线，而且允许它完成它的写操作。当它完成以后，总线仲裁器接下来会选择另一个 CPU，允许这个 CPU 执行它的写操作。和前面一样，这会持续下去，直到所有的写请求都得到满足为止。纯粹的结果就是对存储器的同时访问，无论位置是否相同，始终都被总线仲裁器给顺序化了。从主存储器子系统的角度来看，在任何一点时间上，只会出现一个存储器操作。既然所有的操作都由总线进行了顺序化处理，所以在精确的同一时刻，不可能有两个或者以上的 CPU 能实际写入相同的主存储器（虽然有诸如多端口存储器（multiported memory）和多存储器库（multiple memory bank）这样的能够并行执行某种存储器操作的特殊硬件技术，但是任何支持顺序存储器模型的系统仍然要保持这里所介绍的存储器操作的原子性）。

虽然存储器操作是原子的和顺序的，但是来自多个 CPU 同时的存储器操作，其相对的次序却并不是确定的。如果正好在同一时刻有两个 CPU 试图向同一个位置执行写操作，那么软件不能就哪一个操作先执行做出任何假定。类似地，如果一个 CPU 正在从一个存储器位置上读取数据，而同时另一个 CPU 向这个位置写入数据，那么读取数据的 CPU 既可以读到这个位置上老的数据，也可以读到新的数据，因为不能保证先发生读操作还是写操作。这种情况称为竞争条件（race condition），我们将在 8.4 节中对其进行更详细的讨论。

8.3.3　原子读-改-写操作

因为在 SMP 系统上经常需要对共享存储器地址的访问进行同步，所以大多数实现都通过原子的读-改-写（read-modify-write）操作来对此提供硬件支持。这样的操作能够让 CPU 从主存储器读取一个值，修改它，再把修改过的值保存回存储器中相同的位置，整个过程作为一

次原子总线操作。这些操作和前面介绍的正常的存储器和指令取操作（fetch operation）所使用的正规原子读操作和写操作不同。读-改-写操作是作为 CPU 内的特殊指令来实现的，只有在必须要同步的时候才使用。

在读-改-写操作期间对数据所做的修改类型是特定于实现的，但是最常用的是 test-and-set（测试-设置）指令。Motorola MC68040 和 IBM 370 的体系结构就是使用这种操作的处理器的例子。这条指令从主存储器读一个值（通常是一个字节或者一个字），把它和 0 相比较（相应地在处理器中设置条件码），然后无条件地把 1 保存到存储器内的位置上，所有步骤都在一次原子操作中。一旦一条 test-and-set 指令开始了它的总线周期，那么任何其他的 CPU 或者 I/O 设备都不可能访问主存储器。有了这个基本的操作，操作系统就可以建立更高级的同步操作，这将在后面的章节中介绍。

虽然有可能实现更复杂的操作，比如原子递增或者递减操作，但是现代 RISC 系统倾向于提供更简单的操作。例如，可能最简单的一个原子 read-modify-write 指令就是原子交换（swap-atomic）操作。这种类型的操作用在 Sun SPARC 处理器和 Motorola MC88100 RISC 处理器中，这些处理器没有 MC68040 中的 test-and-set 指令。这样的一条指令仅仅是把保存在寄存器中的一个值和存储器中的一个值进行交换。通过把寄存器中的值设为 1，执行原子的交换操作，然后将寄存器中的值（存储器中原来的值）和 0 相比较，就可以构造出一次测试-设置操作来。图 8-3 用 C 描述了这个过程。

```
int
test_and_set(volatile int *addr)
{
        int old_value;
        old_value = swap_atomic (addr, 1);
        if (old_value ==0)
                return 0;
        return 1;
}
```

图 8-3 使用原子交换实现 test-and-set

test_and_set 函数的参数是要进行操作的一个字的地址。声明 volatile 告诉编译器，即使函数自身不会改变整数指针 addr 所指的值，在函数执行的同时这个值也可以发生变化。在这种情况下，另一个处理器可以同时在相同的位置上执行一次测试-设置操作。声明一个变量是 volatile 变量就禁止了编译器原本可以进行的代码优化处理，比如将冗余的 load 或者 store 指令缩入到一次操作中。虽然对于本例来说并非是严格必需的，但是声明所有可能会由多个处理器同时修改的变量为 volatile 的，是一种良好的编程习惯（当访问 I/O 设备的寄存器时，也会用到 volatile 声明）。

继续说明 test-and-set 的例子，假定 swap_atomic 函数只能在其第一个参数所寻址的字上执行 swap-atomic 硬件指令。它在把那个位置上的值和它的第二个参数交换以后，返回那个位置原来的值。这个实现展示出测试-设置操作确实是将两个独立的操作组合到了一条指令中。测试-设置第一个阶段是取得字的当前值，再用 1 替换它，这是原子操作。第二个阶段是

测试在第一个阶段取得的值。RISC 对此的解决方法是使用独立的、更为简单的指令，从而简化了硬件设计（参见本章末尾的习题 8.6，了解使用原子交换的另一个例子）。

有些 RISC 体系结构对此做了进一步的简化，它们提供了一对指令，一起使用这一对指令就能执行一次原子的读-改-写操作，这对指令是 load-linked 和 store-conditional。MIPS R4000 RISC 处理器就采用了这种方法（MIPS 处理器以前的版本没有实现任何原子的读-改-写指令）。load-linked 指令执行原子的读-改-写操作的前半部分，它从存储器读取一个值（通常是一个字），在硬件中设置一个标志，指出那个位置上正在进行一次读-改-写操作（这个标志通常由高速缓存控制器来维护，对于软件来说不可见）。使用 store-conditional 指令把任何期望的值保存回读取该值的存储器位置，但是只有在仍然设置了硬件标志的时候才这么做，从而完成读-改-写操作。自从执行了 load-linked 指令以来，任何 CPU 或者 I/O 设备对该位置所做的任何保存操作都会清除掉这个标志。因此，如果 store-conditional 指令发现标志还在，它就得到保证，自从完成 load-linked 指令以后，那个位置没有改变过，而且对于相关的存储器位置来说，从 load-linked 开始到 store-conditional 结束的整个指令序列是以原子的方式执行的。于是，可以使用这两条基本的指令构造出更多复杂的原子操作（构造细节留下来作为一道习题）。

使用一种称为 Dekker 算法（Dekker's Algorithm）的软件技术，根本不需要任何原子的读-改-写操作，仍然能获得同步。它仅仅使用单独的原子读和原子写操作就能运行。Dekker 算法将在 13.2 节里详细地介绍。

8.4 互　斥

既然顺序存储模型不能保证来自一个以上的 CPU 同时读和写相同的存储器位置时有确定性的次序，那么没有哪个共享的数据结构被一个以上的 CPU 同时更新而不会有破坏数据的风险。次序的不确定性就会导致出现竞争条件。只要 SMP 中一组操作的结果要取决于两个或者两个以上处理器之间操作的相对次序或者时序，那么就会发生竞争条件。对于内核数据结构的完整性来说，这种不确定的行为可能是灾难性的，必须要防止其出现。

为了举例说明这个问题，考虑这样的情况，在一个双 CPU 的系统中有一个全局计数器，任何一个 CPU 都能在各个时间点上使之增加 1。假定这个计数器必须准确地反映出由两个 CPU 执行的所有递增操作之和（也就是说，一次也不能少）。再假定系统使用 load-store 体系结构，这意味着算术指令（以及大多数其他指令）的操作数和结果都必须在寄存器中，并且采用 load 和 store 指令在寄存器和存储器之间移动（这是大多数 RISC 系统上的典型情况）。因此，为了使内存中一个位置的值加 1，需要图 8-4 中所示的 3 条指令的序列。这个序列以伪汇编语言给出，其中的符号%r0 用于表示寄存器 0，counter 表示全局计数器在主存储器中的值。因为 MP 系统中每个 CPU 都有它自己的一组寄存器，所以%r0 表示正在执行指令的 CPU 上的寄存器 0。

```
load    %r0, counter
add         %r0,1
store   %r0,counter
```

图 8-4　使存储器内的计数器加 1 的汇编语言指令

图 8-5 中的时间序列显示了 counter 在主存储器内的内容，以及每个 CPU 的寄存器 0 的内容。假定每个 CPU 在每个时间间隔内执行一条指令。对于本例来说，考虑这样的情况，CPU 1 先执行图 8-4 所示的代码片段中所有 3 条指令，然后是 CPU 2（在一列中出现的短划线表示在那个时间点上，这个地方的值与本例无关）。

	CPU 1			CPU 2	
时间	执行的指令	寄存器 %r0	计数器的值	执行的指令	寄存器 %r0
1	load %r0,counter	0	0		-
2	add %r0,1	1	0		-
3	store %r0,counter	1	1		-
4		-	1	load %r0,counter	1
5		-	1	add %r0,1	2
6		-	2	store %r0,counter	2

图 8-5 如果两个 CPU 顺序执行代码序列所产生的结果

在这种情况下，counter 的结果值是正确的：该值最初为 0，递增两次（每个 CPU 一次），最后得出值为 2。只要两个 CPU 从没有同时执行过 3 条指令的序列，那么 counter 的值就一定是正确的。如果它们同时执行过，那么就会出现图 8-6 中的结果。

在这里我们看到，counter 在主存储器中的值是不正确的。值为 1 而不是 2，这意味着少记了一次递增操作。之所以出现这样的情况，是因为两个 CPU 都从主存储器取得了 counter 的原始值，而没有意识到在另一个 CPU 上也正在同时进行递增操作。在这样的场合下，就称处理器彼此竞争，因为结果取决于首先执行完代码序列的那个处理器。

	CPU 1			CPU 2	
时间	执行的指令	寄存器 %r0	计数器的值	执行的指令	寄存器 %r0
1	load %r0,counter	0	0	load %r0,counter	0
2	add %r0,1	1	0	add %r0,1	1
3	store %r0,counter	1	1	store %r0,counter	1

图 8-6 如果两个 CPU 同时执行代码序列所产生的结果

任何更新在两个或者两个以上处理器之间共享的变量或者数据结构的指令序列都能导致出现一种竞争条件。指令序列本身被称为临界段（critical section），它们操作的数据称为临界资源（critical resource）。一个临界段既可以像图 8-4 中 3 条指令的序列那么短，也可以包含大段代码。为了消除由多个处理器同时执行临界段所造成的竞争条件，一次至多有一个处理器在临界段内执行。这被称为互斥（mutual exclusion），并且能够以多种方式来实现。

在继续展示如何在 MP 系统中实现互斥之前，回顾一下它在单处理机 UNIX 系统上是如何实现的，以及这些技术为什么在 MP 上不能使用，会对我们有所帮助。在后面几章中我们将介绍几种能够在 MP 系统上实现的技术。

8.5　回顾单处理机 Unix 系统上的互斥

即便是在单处理机的操作系统上也可能有竞争条件。任何允许多个控制线程的系统，如多进程，都需要考虑线程之间的互斥。由中断处理程序（interrupt handler）所执行的指令也有可能同它们所中断的代码发生竞争。

下面几个小节回顾了在单处理机 UNIX 系统上互斥实现的重要方面。这些系统按照互斥的类型把它们对这个问题的处理分成了 3 类，这 3 类是短期互斥（short-term mutual exclusion）、和中断的互斥（mutual exclusion with interrupts）以及长期互斥（long-term mutual exclusion）。

8.5.1　短期互斥

短期互斥是指防止短临界段中的竞争条件，比如 8.4 节中介绍的临界段。当内核正处于更新其数据结构之一的期间，就会发生这样的临界段。因为内核的数据结构是由所有正在执行的进程所共享的，如果两个或者两个以上正以内核态执行的进程要同时更新相同的数据结构，就可能出现一个竞争条件。因为单处理机一次只能执行一个进程，所以只有在内核中执行的一个进程能够被另一个抢先时，才有可能出现这样的竞争条件。这就是为什么 UNIX 内核的设计者选择让内核在以内核态执行时为非抢先的原因。回忆 1.2 节的内容，以内核态执行的进程没有分时间片执行，不能被抢先。只有当内核态的进程允许时，才能从当前进程的现场切换到另外一个进程。

以内核态执行的进程非抢先的规则大大地降低了单处理机 UNIX 内核实现的复杂度。因为一次只允许一个进程在内核中运行，而且决不会被抢先，所以在检查和更新内核数据结构的同时不会出现竞争条件。因此，比如像在 8.4 节中所介绍的计数器例子这样的情况，无需做更多的工作就可以保持数据结构的完整性。

8.5.2　和中断处理程序的互斥

如果一个中断处理程序执行的代码访问或者更新了由非中断的代码（通常称为基准代码（base-level code））使用的同一数据结构，那么就会出现竞争条件。例如，以内核态执行的一个进程在执行图 8-4 中代码序列时发生了一次中断。如果中断处理代码也试图让同一个计数器加 1，那么可能会出现和 MP 的情形一样的错误结果。幸运的是，得到允许以内核态执行的进程会临时禁止中断。因此，只要基准代码要更新一个与中断处理程序共享的数据结构，那么它就首先禁止中断，执行临界段，然后再重新允许中断。禁止和允许中断的动作就实现了互斥。在单处理机 UNIX 内核中，spl 函数提供了允许和禁止中断的手段。spl 代表 Set Priority Level（设置优先级），它是指把中断优先级设低，让处理器忽略中断。例如，图 8-7 显示了一段 C 代码，它能正确地保护 counter 的递增操作不受所有可能的中断影响。

```
s = splhi();
counter++;
splx(s);
```

图 8-7 保护临界段不受中断影响

splhi 函数屏蔽了所有中断（把中断优先级设置为最高级）。直到另一次对 spl 函数的显式调用消除对中断的屏蔽之前，都会一直保持对中断的屏蔽。因为来自不同设备的中断可能处于不同的优先级，所以用来屏蔽中断的 spl 函数会返回过去的优先级，以便在临界段完成的时候能恢复它。现在就能安全地递增计数器而不会和中断处理程序中的代码发生竞争。splhi 返回的以前的优先级则用 splx 函数予以恢复。

重要的是理解这种互斥的实现和短期互斥的实现之间有什么不同。在采用短期互斥的情况下，实现内核态进程非抢先的一般性策略就能解决问题，而无需显式地把它编码入内核。在采用中断互斥的情况下，互斥必须通过使用 spl 函数显式地编码入算法中。

8.5.3 长期互斥

从用户程序的角度来看，大多数 UNIX 系统调用提供的服务都保证是原子操作。例如，一旦对一个普通文件（regular file）开始执行系统调用 write，那么操作系统就一定会保持住对同一文件的任何其他 read 或者 write 系统调用，直到当前的系统调用完成为止（有了这种文件操作的互斥，编写针对共享文件的有确定性的用户程序就更加容易了）。为了完成系统调用 write，可能需要一次或者更多的磁盘 I/O 操作。磁盘 I/O 操作与 CPU 能够在同期内完成的工作量相比，是相当长的操作。因此，在这样长的操作上使用非抢先策略是非常不可取的，因为在等候 I/O 操作完成之前，CPU 会处于等候状态。为了避免出现这样的情况，执行系统调用 write 的进程需要让它们自己被抢先，以便能够运行其他进程。不过，一旦允许抢先，那么就需要有一种技术能防止发起对同一文件的其他 read 和 write 调用。单处理机 UNIX 内核以 sleep 和 wakeup 函数实现这种类型的互斥。

函数 sleep 是一个内部的内核例程（只能由以内核态执行的进程使用），它能挂起调用它的进程，直到指定的事件发生为止。这是一个以内核态执行的进程自愿让出控制权，允许自己被抢先的重要手段。函数 wakeup 用于发出一个特殊事件已经出现的信号，它使得所有等候该事件的进程都被唤醒，并放回到运行队列中。事件用一个任意的整数值来表示，它往往是和该事件相关的内核数据结构的地址。我们以下面的例子来说明使用 sleep 和 wakeup 函数实现长期互斥的技术。

内核中每个要求长期互斥的对象都用一个数据结构的实例来表示。要在对象上实现原子操作，就要“锁住”该对象，以便一次仅让一个进程访问它。通过给数据结构增加一个标志，如果对象当前上了锁，那么设置这个标志就可以做到这一点。为了简单起见，假定这个标志保存在对象的数据结构中某个字节里，于是每个数据结构都有一个唯一的标志。图 8-8 展示了一种在任意一个对象上实现互斥的可能途径（在不同的 UNIX 系统版本上，实际的细节也不同，但和这里的讨论不相关）。

```
void
lock_object( char *flag_ptr )
{
        while( *flag_ptr )
                sleep( flag_ptr );
        *flag_ptr = 1;
}
```

图 8-8　锁住一个对象的代码

在这个例子中，标志被设置为 1 表明一个进程现在锁住了对象。在原子操作开始的时候，调用函数 lock_object，通过传递一个指向与之相关联的标志字节的指针来锁住对象。如果对象当前没有被锁住，那么 while 语句中的条件不满足，进程就通过设置标志来锁住对象（之所以需要使用一个 while 循环将在后面解释）。现在，进程可以继续执行操作，自愿地使它自己抢先，比如在原子文件操作的情况下等候磁盘 I/O 完成，而且确保在第一个进程明确地解锁之前，不会有其他进程能够成功地锁定对象。尝试访问同一个对象的任何其他进程都使用同样的数据结构，并且以同样的标志字节地址调用 lock_object 函数来尝试锁住它。但是，这一次 while 语句中的条件为真，那么进程将执行 sleep 调用，这个调用会挂起进程。

重要的是要理解测试标志、发现它被清除以及设置它的操作构成了一个临界段，而且必须采用互斥来处理它。否则就有可能出现竞争条件，从而导致出现两个或者两个以上的进程认为它们都锁住了对象的情况。幸运的是，单处理机的非抢先策略（nonpreemptability policy）防止了竞争条件的出现。

注意，sleep 函数中是标志自身的地址。使用对象所关联的数据结构的地址来标识事件是内核中到处使用的一种普遍约定。之所以采用这样的约定，是因为它能轻而易举地让等候不同对象上的锁的进程在不同的事件上睡眠。那样一来，在某个特殊的锁被打开的时候，只有正在等候该锁的进程才会被唤醒（而不是唤醒等候任何锁的所有进程）。

函数 sleep 本身只需要执行几个简单的操作就能挂起调用它的进程。它首先在一张表中记录下事件，以便随后的 wakeup 操作能确定该唤醒哪些进程。然后它执行一次现场切换，选择执行另一个进程。现在调用 sleep 的进程不再运行了，直到它被唤醒为止（注意，在实际的 UNIX 内核实现中，函数 sleep 有另一个参数，它指定进程被唤醒的时候应该运行的优先级。因为它并不影响互斥，所以为简单起见就省略掉了）。

当保持锁的进程完成了它在对象上的原子操作之后，它就会按照图 8-9 那样调用函数。

```
void
unloc_object( char *flag_ptr )
{
        *flag_ptr = 0;
        wakeup( flag_ptr );
}
```

图 8-9　解开一个对象上的锁的代码

在这里清除了标志，使用函数 wakeup 唤醒所有等候该锁的进程。传递给 wakeup 的事件必须和 sleep 函数使用的事件相吻合，以便可以唤醒正确的进程（即那些正在等候这个特殊锁的进程）。要唤醒一个进程，wakeup 搜索 sleep 记录哪些进程在哪些事件上睡眠的那张表，将所有符合给定事件的进程放入运行队列。函数 wakeup 的一个重要方面是调用它不必知道实际上是否有任何进程在该事件上睡眠。如果 wakeup 发现没有进程等在一个事件上，那么它就什么也不做，直接返回。以后在同一事件上睡眠的进程也一定会被挂起，而不管在上一次 wakeup 操作期间发生了什么事情。

当图 8-8 所示的代码中睡眠的进程被唤醒，随后由调度器（scheduler）选择执行的时候，它就接着从它上次停止的地方开始执行，即在函数 sleep 中发生现场切换的地方。此刻，sleep 就返回了调用它的函数。重新开始检查 while 循环，如果所需的锁仍然是打开的，那么循环终止，进程就获得了自己的锁。如果再次发现对象是被锁定的，那么进程就返回去睡眠，等待被持有锁的新进程唤醒。这可以在两种不同的情况下发生。第一种情况是当已经被唤醒的进程正在运行队列上等候它的时间片的时候，另一个进程出来锁住了对象。第二种情况是有多个进程在等待同样的锁。因为 wakeup 操作没有记忆，所以 wakeup 函数必须唤醒在同一事件上睡眠的所有进程。它们都被放入到运行队列，按照调度器对它们的选择顺序执行。第一个要运行的进程会发现锁被释放了，并且成功地锁定了它。如果为同样的事件而被唤醒的其他进程之一在第一个进程释放锁之前要运行，它会发现已经上锁了，所以就回去睡眠。当它被唤醒的时候，它将再次尝试获得锁。这就是为什么 lock_object 函数必须包含一个 while 循环的原因。

现在可以看到，lock_object 和 unlock_object 函数向它们所括起来的代码段提供了长期互斥功能。和非抢先策略所提供的短期互斥不同，这种类型的互斥必须明确地编码在内核中。

8.6 在 MP 上使用 UP 互斥策略的问题

为了获得一个高性能的 MP 系统，需要让系统调用和其他内核活动出现在所有的处理器上。这样一来，内核的工作量就可以分摊到整个系统上。遗憾的是，前面各小节介绍的技术虽然能够让单处理机内核实现避免竞争条件，但是却不能在 MP 系统上有一个以上的处理器可以同时执行内核代码的时候正确地工作。原因如下，这些问题的解决方法则在以后的章节中介绍。

让单处理机内核不能在 MP 系统上正确运行的主要困难在于，在内核中同时执行的多个处理器违反了支持短期互斥的假定。通过使用内核进程的非抢先策略，UP 系统能够避免许多竞争条件。即使一个 MP 系统上的内核能防止在一个特殊处理器上的某个进程被另一个进程抢先，在另一个处理器上启动系统调用的进程也能够造成和 8.4 节所描述的竞争条件一样的情况。一旦有一个以上的进程以内核态开始执行，那么除非采取额外的措施来防止竞争，否则内核的数据结构就会被破坏。

在 MP 系统上，和中断处理程序的互斥也可能不会发挥正常的功能。函数 spl 只会影响其执行所在处理器的处理器优先级，而不会影响传递给其他处理器的中断。根据硬件设计的不同，中断可以传递给系统中的任何一个 CPU，或者它们也可以始终被定向到一个 CPU。无

论是这两种情况的哪一种，在一个处理器上执行的进程，当它使用 spl 函数在与中断处理程序共享的数据结构上实现互斥时，如果中断处理程序开始在一个不同的处理器上执行，那么它就不能真正保护数据。

最后，在 MP 系统上，以 sleep 和 wakeup 函数实现长期互斥所采用的编码技术也不能正确工作。回忆 8.5.3 小节的内容，lock_object 的实现要依靠短期互斥来防止在测试标志和进程要么开始 sleep 要么设置标志本身之间的时间内出现竞争条件。因为短期互斥策略在 MP 上不再起作用，所以这些代码序列现在可以包含竞争。例如，考虑这样的情况，两个在不同处理器上的进程同时开始执行图 8-8 中的 lock_object 函数。假定当前没有上锁，它们两者都检测标志，然后发现标志被清除了，那么它们将开始设置标志，继续执行。每个进程接下来都会认为它获得了锁，因而违反了互斥的策略。

在 lock_object 和 unlock_object 之间也可以有竞争条件。一个进程执行 lock_object 来检测标志，并且发现它被设置了，正好另一个进程要释放这个锁。假如在第一个进程检测锁和开始执行 sleep 函数之间，第二个进程释放了锁。在这种情况下，释放锁的时候调用的 wakeup 函数什么也不会做，因为刚才要锁住对象的进程还没有睡眠呢。当第一个进程调用 sleep 的时候，即使现在锁已经解开了，它还是会被挂起。注意，现在没有进程会唤醒它（没有哪个进程占有锁，因此也就没有哪个进程会调用 unlock_object）。这个进程继续睡眠，直到另一个进程试图获得同一个锁为止。这个新进程将会发现锁是解开的，于是不必睡眠就能获得锁。当它最终释放锁的时候，wakeup 才会让第一个进程再次运行。因为不能保证第一个进程为此要等候多长时间，所以必须通过消除竞争条件来防止出现这种情况。

在接下来的 3 章里，我们将介绍防止这些问题的 3 种重要技术。为了防止所有可能的竞争条件，它们对内核所必须进行的修改程度也不同。

8.7 小　结

SMP 是最常用的多处理机系统类型，因为它将单处理机系统的执行环境并行化了。这类 MP 系统的重要特点是所有的 CPU 和 I/O 设备都是紧密耦合的，它们共享一个公共的全局主存储器，而且对称和平等地访问存储器。大多数 MP 内核的实现都保留了单处理机的外部编程模型，从而使得应用程序无需修改就可以在 XP 上运行。

MP 的存储器模型描述了 load-store 指令在一个程序中的次序，以及多个 CPU 同时访问主存储器时结果如何。顺序存储模型提供的原子读和写操作要按照程序次序在每个 CPU 上执行，但是它没有为多个 CPU 同时访问相同存储器位置的操作指定相对次序。正因为如此，顺序存储模型通常会提供某种类型的原子读-改-写操作，CPU 可以将其用于同步的目的。

对共享位置的存储器操作缺乏确定的次序会导致出现竞争条件。对共享的存储器位置或者数据结构的任何多指令操作，比如一次递增操作或者给一个链表增加一个元素，都很容易造成竞争，因为多个处理器会同时试图执行这项操作。为了防止出现竞争，内核必须实现互斥机制来使对共享数据的访问顺序化，从而防止数据遭到破坏。

为了简化单处理机 UNIX 内核系统的设计，就要依靠这一事实，即以内核态执行的进程是非抢先的。这个事实消除了大多数竞争条件，因为在当前执行的进程自愿放弃 CPU 之前，

不会有别的进程能访问任何内核数据。使用显式的锁，并且调用 sleep 和 wakeup 函数，就能实现长期互斥（例如，支持原子文件操作所需要的）。和中断处理程序的互斥则通过在基准内核代码中出现临界段期间显式地屏蔽和开通中断来实现。但是，当内核代码可以在一个以上处理器上同时执行的时候，这些策略都不能在 MP 系统上提供互斥。

8.8 习 题

8.1 如果 CPU 和 I/O 设备都同时试图读或者写主存储器内相同的位置，那么会出现什么样的情况？

8.2 如果 3 个 CPU 都同时把不同的值保存到同一个存储位置，那么如果采用强定序的话，你能预测出 3 次保存操作都完成以后那个位置上是什么值吗？（假定 CPU 1 保存 1，CPU 2 保存 2，CPU 3 保存 3。）

8.3 如果存储器的地址 0x100 处一开始保存的值为 10，并且 CPU 1 把值 1 保存到这个位置，几乎与此同时，CPU 2 从这个位置读取数据，那么在两个操作都结束以后，这个位置的值是多少？CPU 2 读取的值是多少？为什么？假定系统使用顺序存储模型。

8.4 在总线上同时能够进行多少个总线交易？假定使用 8.3.1 小节所定义的顺序存储模型。

8.5 考虑一个 SMP 系统，它包含 10 个 CPU 和 5 个能够执行 DMA 操作的 I/O 设备。总线仲裁采用循环技术，这意味着每循环一次每个 CPU 或者 I/O 设备都会获得一次总线交易，直到所有的 CPU 和设备都轮到一次为止。然后重复循环，给每个请求方另一个周期，依此类推。如果一次总线交易会花费单位时间（包括仲裁），那么任意一个 CPU 或者 I/O 设备在它请求总线和获得总线之间的等待时间最长和最短是多少？

8.6 用下面的原型编写一个 C 函数，实现一个原子测试-设置操作：

```
int test_and_set( int *addr );
```

该函数应该无条件地把 1 保存到寻址的位置。如果以前的内容是非零值，那么它应该返回 1；否则，它应该返回 0。对所寻址位置的更新应该是原子的。使用下面的已经编码好的 C 例程来实现这个函数：

```
int load_linked( int *addr );
int store_conditional( int *addr, int value );
```

这里的 load_linked 函数在地址为 addr 的字上执行一次 load_linked 操作，然后返回这个字，store_conditional 有条件地把 value 保存在地址 addr 里，如果保存成功返回 1，否则返回 0。

8.7 用下面的原型编写一个 C 函数，实现一个原子交换操作：

```
int swap_atomic( int *addr, int new_value );
```

地址 addr 原来的值和 new_value 进行的交换是原子操作，并且返回原来的值。使用习题 8.6 中定义的 load_linked 和 store_conditional 函数。

8.8 用下面的原型编写一个 C 函数，实现一个原子递增操作：

```
void inc_atomic ( int *addr );
```

addr 指向存储器内的一个字，它的递增操作是原子的。使用习题 8.6 中定义的 load_linked 和 store_conditional 函数。

8.9　实现前一个问题中的 inc_atomic()函数，但是这次使用 swap-atomic 原语（在 8.3.3 小节中介绍）作为唯一的读-改-写操作（也就是说，test-and-set、load-linked 和 store-conditional 都不能用在这个问题上）。使用习题 8.7 里编写的 atomic-swap 函数。对于这个问题来说，假定要增加的值始终大于或者等于 0，而且它决不会溢出。如果有必要，允许使用一个单独的函数来读取计数器的值，以避免在增加计数器的值时出现竞争条件。如果你的解决方法需要它的话，也同时给出这个函数。

8.10　下面的这个函数可以在包含竞争条件的链表中插入一个新元素吗？假定链表在一个 SMP 系统内所有的处理器之间共享，但是直到那个新元素在链表中才会被共享。解释为什么存在或者不存在竞争条件。

```
struct element_t {
        element_t *next;
        int  data;
};
void insert( element_t *list, element_t *new )
{
        new->next = list;
        list = new;
}
```

8.11　重做上一题，假定是单处理机环境，解释为什么存在或者不存在竞争条件。如果在中断处理程序和基准代码中都使用了 insert 函数，会出现什么样的情况？

8.12　考虑两个 CPU，它们执行如下所示的代码：

CPU 1

```
int
get_wait_count( int *lock_ptr )
{
    return lock_ptr[1];
}
```

CPU 2

```
void
lock_object ( int *lock_ptr )
{
    while ( lock_ptr[0] ) {
        lock_ptr[1]++;
        sleep( lock_ptr );
        lock_ptr[1]--;
    }
    lock_ptr[0] = 1;
{
```

只有 CPU 1 执行 get_wait_count 函数，只有 CPU 2 执行 lock_object 函数。函数 lock_object 是从图 8-8 中所示的版本改过来的，它保持着正在等待锁的进程计数。这是通过传递一个指向双字数组的指针来实现的，双字数组的第一个字保存着锁标志，而第二个字保存着等待锁的进程计数。这里会有竞争条件吗？解释原因。

8.13　为了保证在尝试用图 8-8 中的代码获得锁的时候进入睡眠的进程，即使一直有别的进程在持续竞争相同的锁，最终也能够获得锁，应该做什么？假定是一个单处理机环境。

8.9 进一步的读物

[1] Adve, S., and Hill, M., "Weak Ordering-A New Definition," *Proceedings of the 17th Annual Internation Symposium on Computer Architecture*, May 1990, pp. 2-14.

[2] Adve, S., and Hill, M., "Implementing Sequential Consistency in Cache-Based Systems," *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990, pp. I:47-50.

[3] Dewan, G., and Nair. V.S.S., "A Case for Uniform Memory Access Multiprocessors," *Computer Architecture News*, Vol. 21, No. 4, September 1993, pp. 20-6.

[4] Dubois, M., Scheurich, C., and Briggs, F., "Memory Access Buffering in Multiprocessors," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 1986, pp. 434-42.

[5] Dubois, M., and Scheurich, C., "Memory Access Dependencies in Shared-Memory Multiprocessors," *IEEE Transactions on Software Engineering*, Vol. 16, No. 6, June 1990, pp.660-73.

[6] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990, pp. 15-26.

[7] Gharachorloo, K., Gupta, A., and Hennessy, J., "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," *Proceedings of the Fourth international Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 245-57.

[8] Gillford, P., Fielland, G., and Thakkar, S., "Balance: A Shared Memory *Multiprocessor," Proceedings of the Second International Conference on Supercomputing*, May 1987.

[9] Lamport, L., "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, Vol. C-28, No. 9, September 1979, pp. 241-8.

[10] Lamport, L., "The Mutual Exclusion Problem: Part I-A Theory of Interprocess Communication," *Journal of the ACM*, Vol. 33, No. 2, 1986, pp. 313-26.

[11] Lamport, L., "The Mutual Exclusion Problem: Part II - Statement and Solutions," *Journal of the ACM*, Vol. 33, No. 2, 1986, pp. 327-48.

[12] Miya, E.N., "Multiprocessor/Distributed Processing Bibliography," *SigArch News*, Vol. 13, No. 1, March 1985, pp. 27-9.

[13] Mosberger, D., "Memory Consistency Models," *ACM SIGOPS Operating Systems Review*, Vol. 27, No. 1, January 1993, pp. 18-26.

[14] Peterson, G.L., "Myths About the Mutual Exclusion Problem," *Information Processing Letters*, Vol. 12, No. 3, June 1981, pp. 115-6.

[15] Sawyer, B.B., "Multiprocessor UNIX Utilizing the SPARC Architecture," *UniForum Conference Proceedings*, February 1989, pp. 107-19.

[16] Scheurich, C., "Access Ordering and Coherence in Shared Memory Multiprocessors," Ph.D. thesis, University of Southern California, May 1989.

[17] Stone, H.S., *High-Performance Computer Architecture*, Third Edition, Reading, MA:Addison-Wesley, 1993.

第9章 主从处理机内核

本章介绍修改单处理机内核实现，使之没有竞争条件地运行在SMP系统上所采用的最简单的技术：主从处理机内核（master-slave kernel）。本章还介绍一种称为自旋锁（spin lock）的SMP互斥原语，用它们重建的短期互斥能够防止出现上一章末尾所描述的问题。接下来介绍MP内核中出现死锁（deadlock）的原因，以及用来避免它们的技术。最后介绍实现一个主从系统必须对内核进行的改动，然后讨论对性能带来的影响。

9.1 引 言

8.5.1小节中介绍的短期互斥实现技术是构造单处理机UNIX内核实现的主要基础之一，它不需要把明确的上锁代码编入内核中的许多地方。没有了这一点，当访问和更新内核数据结构的时候就可能出现竞争条件。仅有这一项技术还不足以在SMP系统上防止两个或者两个以上的进程同时在系统中不同的处理器上执行时出现竞争条件。即使允许内核进程非抢先，多个处理器同时执行的内核活动仍然会导致出现竞争。防止这类竞争可以使用几种方法，有些方法需要对内核进行大规模的修改。本章考虑在MP系统上恢复单处理机非抢先要求的最简单的技术，把它作为更复杂实现的入门。

第8章的短期互斥技术依赖于这样的事实，即在内核中决不会有一个以上的进程同时执行。在MP系统上做到这一点的一项简单技术是要求所有的内核活动都在一个物理处理器上执行，这个处理器称为主处理器（master）。系统中所有其他的处理器都称为从处理器（slave），它们只能执行用户代码。以用户态执行的进程可以在系统中的任何处理器上执行。但是，当进程执行一次系统调用的时候，它就切换到主处理器上。一旦系统调用完成，进程就可以再次在任何处理器上运行。运行在从处理器上的用户态进程所产生的任何陷阱（比如缺页错或者算术异常）也会使得进程切换到主处理器上，因此还要维护内核陷阱处理程序（kernel trap handler）的互斥要求。最后，所有的设备驱动程序（device driver）和设备中断处理程序（device interrupt handler）只能在主处理器上运行。

从内核的角度来看，主从处理机的编排保留了单处理机执行环境。这就能让单处理机的内核实现只需要很少的修改就可以在MP系统上运行，而且可以在任意数量的处理器上执行。要修改的重要领域之一就是如何将进程分配给各个处理器。有一项简单技术可以做这件事情，即有两个独立的运行队列，一个包含必须运行在主处理器上的内核态进程，而一个包含运行在从处理器上的用户态进程。在每次现场切换的时候，每个从处理器选择在从运行队列（slave

run queue）中有最高优先级的进程，而主处理器选择在内核进程队列（kernel process queue）中有最高优先级的进程。运行在从处理器上的进程在执行一次系统调用或者产生一次陷阱的时候就被放入主处理器的运行队列。当主处理器执行一次现场切换的时候，它正在执行的老进程如果是以用户态执行的，那么就把它放入从队列（slave queue）；否则，它就返回主队列（master queue）。

因为有多个处理器同时把进程排入队列、取出队列或者在队列中搜索进程，所以需要有一种方法来防止竞争条件。运行队列（run queue）是唯一需要显式 MP 短期互斥技术的数据结构，因为通过让所有的内核代码运行在主处理器，就可以保护所有其他的数据结构。采用自旋锁（spin lock）是提供这样的短期互斥的最简单方法。

9.2 自 旋 锁

自旋锁是一种 MP 短期互斥机制，它可以用来防止在代码的短临界段期间发生竞争条件。和第 8 章中屏蔽中断和实现长期互斥的函数一样，在进入临界段之前要获得自旋锁，而在临界段完成之后要释放它。自旋锁得名于这样的事实，一个进程在等候另一个进程正在使用的锁时就会处于忙等待（busy-wait，在一个循环中自旋）状态。

自旋锁是实现一个主从处理机内核所需的唯一 MP 原语操作（primitive operation）。正如在后面的章节中将要看到的那样，不同的内核实现可能会使用其他类型的原语。保护短临界段的自旋锁的简单性和高效性使得它们在这些实现中也会派上用场。

在存储器中使用一个字，由它反映出锁的当前状态，就可以实现自旋锁。当一个特殊的进程能够把自旋锁的状态以原子方式从开锁（unlocked）状态转变为上锁（locked）状态时，那个进程就需要独占使用它。必须以一次原子操作来做到这一点，以确保一次只有一个进程能够获得锁。对于下面的例子来说，0 值用来表示自旋锁的开锁状态。所有的例程都接受一个指向自旋锁状态字的指针，在它上面执行操作。于是可以使用图 9-1 中的例程来初始化一个自旋锁。

```
typedef int lock_t;
void
initlock( volatile loce_t *lock_status )
{
        *lock_status = 0;
}
```

图 9-1　初始化一个自旋锁

通过使用 8.3.3 小节中介绍的 test-and-set 指令，图 9-2 中的函数就可以用来以原子方式锁定一个自旋锁（test_and_set 函数的定义在习题 8.6 中给出，如果前面的状态不为 0 就返回 1，否则返回 0）。

图 9-2 中的函数通过以原子方式把自旋锁的状态从 0 转变为 1 来锁住它。如果锁的状态已经是 1 了（意味着这个锁正在由另外一个进程使用），那么 test_and_set 函数就返回 1，并

且处理器在循环中自旋，直到该锁被释放为止。只要把锁的状态设置为 0，就可以释放锁，如图 9-3 所示。

```
void
lock( volatile lock_t *lock_status )
{
        while(test_and_set( lock_status ) == 1 )
                ;
}
```

图 9-2 以原子方式锁定一个自旋锁

```
void
unlock(volatile lock_t *lock_status )
{
        *lock_status = 0;
}
```

图 9-3 解开一个自旋锁

自旋锁可以在有任何数量处理器的系统上正常工作。如果多个处理器恰好同时试图获得同一个自旋锁，那么 test_and_set 函数的原子特性就可以一次只让一个处理器把锁的状态从 0 改为 1。其他进程会看到锁已经被设为 1 了，从而进入自旋，直到拥有锁的进程释放它为止。现在，内核可以用 lock 和 unlock 函数调用构成一个临界段。

```
lock( &spin_lock );
perform critical section
unlock( &spin_lock );
```

图 9-4 采用自旋锁实现一个临界段

如果临界段较短（通常不超过几百行机器指令），那么自旋锁就可以工作良好。不应该把它们当作一种长期互斥技术来使用，因为等候锁的处理器在自旋期间什么有用的工作也不做。如果处理器在等待获得锁上花费了太多的时间，那么系统的整体性能就会降低。如果有许多处理器频繁地争用相同的锁，也会发生这样的情况（这几点将在第 10 章里进一步说明）。

减少对锁的争用可以采用两种办法。第一，内核可以针对不同的临界资源使用不同的自旋锁，这就防止了处理器在没有竞争条件威胁的时候被另一个处理器挂起。第二，应该增强 lock 和 unlock 函数，在上锁的时候屏蔽中断。否则，在处理器获得一个自旋锁的同时所出现的中断会进一步拖延其他进程等候那个锁的时间（而且可能导致死锁，下一小节介绍）。

9.3 死　　锁

当一个处理器试图一次获得一种以上资源的独占使用权时，必须小心处理，否则就可能出现死锁。当两个或者两个以上的处理器彼此占有对方所需要的资源，而彼此又等待对方释

放所占有的资源时，就会发生死锁。例如，考虑这样一个数据结构，它的每个元素都在两个独立的链表上。假定使用独立的自旋锁来保护每个列表，以便能够独立地访问它们。再假定某个操作要求遍历一个列表所检索到的元素必须从两个列表中断开链接，而且必须以一次原子操作来完成。这意味着在消除一个元素的链接时必须同时拥有两个列表的锁。如果通过先获得要遍历的列表的锁，再尝试获得另一个列表的锁来实现，那么就可能造成死锁。当两个进程的每一个都从遍历另一个列表开始操作的时候，如图 9-5 所示，就可能出现这种情况。

处理器 1	处理器 2

lock(&lock_a);
find element to unlink on list a
lock(&lock_b);
unlink element from both lists
unlock(&lock_b);
unlock(&lock_a);
	lock(&lock_b);
	find element to unlink on list b
	lock(&lock_a);
	unlink element from both lists
	unlock(&lock_a);
	unlock(&lock_b);

图 9-5　可能出现死锁的情形

如果两个进程恰好同时开始执行它们各自的代码片段，那么处理器 1 就获得 lock_a，处理器 2 获得 lock_b。接着，处理器 1 忙等待 lock_b，而处理器 2 忙等待 lock_a。既然两个处理器都不会放弃它已经获得的锁，那么两个处理器就会在尝试获得另一个锁的循环中永远自旋下去。现在，这两个处理器已经死锁住了。这种特殊的死锁情形称为 AB-BA 死锁，它是指这一事实，即两个处理器试图以相反的顺序获得对方的锁。这种动作一定会造成潜在的死锁状态。

注意，死锁是否真正发生取决于两个处理器之间的相对时间顺序。如果处理器 1 和处理器 2 其中之一能在对方开始执行之前完成其代码序列的话（如图 9-5 所示），就不会造成死锁。

为了防止发生这类死锁，所有的处理器都必须以相同的次序获得嵌套锁（nested lock），即顺序获得且同时占有的锁。更改前面例子中的代码，结果为图 9-6 中的常见代码序列，任何处理器一旦要开始执行一次操作，从两个列表中以原子方式断开一个元素的链接时，就可以执行这段代码。

```
lock ( &lock_a );
lock ( &lock_b );
find element to unlink on list a or b
unlink element from both lists
unlock( &lock_b );
unlock ( &lock_a );
```

图 9-6　修改算法，防止死锁

注意，无论哪一个链表，在不会马上执行一次原子断开链接操作的情况下，只要获得该链表的锁，就仍然可以遍历它。即使另一个进程同时执行一次原子的断开链接操作，也不可能出现死锁。

总而言之，防止 AB-BA 死锁的关键在于，所有的处理器都以完全相同的次序获得和释放嵌套锁。当涉及到 3 个或者更多锁的时候也是如此：只要在获得和释放锁的时候保持相同的次序，那么就不会出现死锁。

在别的情况下使用自旋锁时也可能会出现死锁。例如，如果一个占有自旋锁的进程要执行一次现场切换，那么任何试图获得同一个锁的处理器就会保持自旋，直到占有锁的进程再次恢复运行，并且释放了锁为止。从性能的立足点来看，不希望出现这样的情况，因为其他处理器会自旋任意长的一段时间，但更糟糕的是，它们也能导致死锁。如果系统中所有的处理器都试图获得一个已经被现场切换出去的进程所占有的自旋锁，那么就可能发生这种情况。一旦处理器开始为锁而自旋，那么它们就再也不能执行现场切换了。这意味着占有自旋锁的进程不可能重获执行，因而它再也没有机会释放锁了。所有的处理器将会永远自旋下去，死锁住系统。为了防止发生这种死锁问题，不允许一个进程跨越现场切换还能占有一个自旋锁。

在采用自旋锁时出现死锁的另一种情形是基准内核代码和一个中断处理程序使用了同样的自旋锁。如果一个处理器已经锁住了自旋锁，而且在该处理器上发生了一次中断，那么如果中断处理程序企图获得同一个锁，就会导致死锁。处理器会在中断级上永远自旋下去，因为被中断的进程再也没有机会释放锁了。出于这个原因，当一个中断处理程序要使用的自旋锁被基准内核代码占有的时候，就必须屏蔽中断的发生。

一个试图两次获得同一个锁的进程自身也可以导致一次死锁。和前面的情况一样，进程也会在第二次尝试获得同一个锁的时候永远自旋下去。有些实现修改了自旋锁原语，以检查一个进程是否两次锁定了同一个锁（称为递归上锁（recursive locking）），如果进程已经获得了这个锁，那么就跳过此次操作。进程能够记录已经尝试过的递归上锁的次数，以便在实际释放自旋锁之前必须执行同等数量的解锁操作。这就能让嵌套的过程获得锁，而又无需知道上一层执行了什么样的上锁操作。本书里的实现和例子没有依靠这样的技术，所以就不进一步考虑它们了。

9.4 主从处理机内核的实现

在采用一个主从处理机内核实现的情况下，唯一的临界资源（critical resource）就是两个运行队列。向两个队列加入进程和从两个队列取出进程的操作都必须采用互斥来完成，以防队列被破坏。用一个自旋锁来保护每个队列就能轻而易举地达到这个目的。

9.4.1 运行队列的实现

假定每个运行队列都作为一个无序链表（unsorted linked list）来实现（优先级队列是一种更好的实现，但是简单的无序链表能让这个例子集中在问题的互斥方面）。将队列中进程的进程表项链接起来就构成了链表。在这些信息中，进程表项包含有进程的优先级和一个指向链表中下一个元素的指针。为了防止不必要的争用，为主运行队列和从运行队列使用单独的自旋锁。图9-7显示了这些数据结构的定义。

结构queue的定义展示出MP内核中使用的一种典型的编码技术：将一个临界资源和保护它的锁组合到一个数据结构中。

在系统启动时初始化两个队列所采用的例程如图9-8所示。

```
Typedef struct proc proc_t;
typedef struct queue queue_t;

struct proc {
        proc_t p_next;      /* 运行队列中的下一个进程 */
        int p_pri;          /* 进程优先级 */
        ...
};

struct queue {
        lock_t q_lock;      /* 保护队列的锁 */
        proc_t q_head;      /* 运行队列的开头 */
};

queue_t master_queue;
queue_t slave_queue;
```

图 9-7　主从运行队列的声明

```
Void
init_queue( queue_t *q )
{
        initlock( &q->q_lock );
        q->q_head = NULL;
}

init_queue ( &master_queue );
init_queue ( &slave_queue );
```

图 9-8　初始化一个运行队列

接着可以用图 9-9 中所示的代码把一个进程放入队列之一。系统中的任何处理器都能在任何时刻调用函数 enqueue，因为自旋锁可以防止在处理运行队列的时候可能发生的一切竞争条件。

在占有自旋锁的同时调用 splhi 可以屏蔽所有的中断，这就防止了同使得进程可以运行的中断处理程序发生竞争。例如，当一次 I/O 操作完成，让等待 I/O 的进程被放回到运行队列中的时候，就可能发生这种情况。如前所述，如果运行队列自旋锁被占有时发生了一次中断，而中断处理程序又调用了 enqueue 函数，就会造成死锁（有些实现将 splhi 和 lock 的调用都组合到了一次操作中，参见 12.4.1 小节）。

图 9-10 中所示的 dispatch 函数从一个队列中选择并删除有最高优先级的进程，如果队列为空，则返回 null。假定 p_pri 的最小值代表队列中优先级最高的进程。如果有一个以上的进程有相同的 p_pri 值，而且这个值是最小值，那么就选择最老（最靠近队列末尾）的那个进程。

```
Void
enqueue(queue_t *q, proc_t *p )
{
        int s;

        s = splhi();
        lock ( &q->q_lock );
        p->p_next = q->q_head;
        q->q_head = p;
        unlock ( &q->q_lock );
        splx(s);
}
```

图 9-9 将一个进程排入到运行队列中

```
Proc_t *
dispatch( queue_t *q )
{
   proc_t **p;
   proc_t **highest_ptr;
   proc_t *highest;
   int s;

   highest_ptr = NULL;
   s= splhi();
   lock( &q->q_lock );

   for (p = ( proc_t** )&q->q_head; *p; p = &(*p)->p_next )
      if (highest_ptr == NULL ||
         (*p)->p_pri <= (*highest_ptr)->p_pri)
            highest_ptr = p;

   /* 除非队列为空，否则解除优先级最高的进程的链接 */

   if ( highest_ptr != NULL ) {
        highest = *highest_ptr;
        *highest_ptr = (*highest_ptr)->p_next;
   } else
        highest = NULL;

   unlock ( &q->q_lock );
   splx(s);
   return highest;
}
```

图 9-10 从运行队列中选择进程

在整个搜索操作期间，以及将选出的项从队列中取出时，必须占有队列的自旋锁。在没有获得锁的情况下就搜索列表的做法是不正确的，因为如果另一个处理器加入或者删除一项的话，队列的状态会发生变化。这也确保了同时要分配进程的两个处理器不会选择同一个进程。一旦从队列中删除了选出的进程，那么就可以安全地释放锁。

和以前一样，必须屏蔽中断，防止在执行 dispatch 中的临界段时发生一次中断，而中断处理程序又调用 enqueue 函数的时候发生死锁。

9.4.2　从处理器的进程选择

从处理器只能执行在从运行队列中的进程。当一个从处理器需要选择一个要执行的新进程时，它会执行如图 9-11 所示的代码片段。

```
While((newproc = dispatch(&slave_queue)) == NULL )
        ;
```

图 9-11　选择要运行的新进程的从处理器代码

如果队列中没有进程，那么从处理器只是在循环中忙等待，直到能够得到一个进程为止。在这里的情况下，处于忙等待状态是没有害处的，因为没有什么要做的。但是要注意，在等候一个进程运行的时候，从处理器会重复地调用 dispatch，它每次都要获得和释放队列锁。虽然在从处理器上执行它的开销不明显，但是占有锁的所有从处理器造成的总开销会不必要地延迟试图把一个进程排入队列的处理器。对于主从实现来说，这并不是一个临界问题，因为 dispatch 例程中代码的临界段在队列为空的时候很小。即使如此，看看如何避免不必要的锁争用也是有用处的，因为这项技术可以应用到其他情形。为了减少争用，可以对图 9-10 所示的 dispatch 例程进行修改，在获得锁之前先检测队列的状态。可以把图 9-12 中的代码加入到例程的开头部分，在队列为空的时候提早返回。

即使别的进程同时正在向队列排入进程或者从队列中取出进程，按照上面的方式，无需占有锁也能安全地检查队列的状态。和占有锁时发生的竞争条件相比，这样做所造成的竞争条件是不同的。例如，假定图 9-12 中的代码在发现队列为空和执行返回之前这段时间内，另一个处理器把一个进程排入队列。这种情况和图 9-10 中的代码在发现队列不为空之后释放队列锁的情况不同，而且在执行 return 语句之前，另一个处理器就把一个进程排入了队列。

其次，当队列里只有一个进程的时候，也没有两个不同的处理器会彼此竞争的危险。假定两个从处理器同时开始执行图 9-12 中修改后的 dispatch 代码。如果在队列中有一个进程，那么两个从处理器会发现队列非空，便继续要求获得队列锁。一个处理器将首先获得锁，从而成功地将链表内的一个项取出队列。另一个处理器接着运行，它发现队列为空，就只是返回 null。这里的关键因素在于，在获得队列锁之后要重新检测队列状态。即使函数顶部的测试表明队列不为空，代码也不能假定至少有一个进程在队列中。另一个重要因素是图 9-11 中的代码要继续调用 dispatch，直到它找到一个进程为止。这就消除了在 dispatch 例程正好发现队列为空之后又新加入进程的时候可能出现的任何竞争。

当运行在一个从处理器上的进程执行一次系统调用或者产生一个陷阱的时候，它就通过

使用 enqueue 函数把它放入 master_queue 来切换到主处理器上，接着再使用图 9-11 中的代码来选择一个要在从处理器上执行的新进程。

```
Proc_t *
dispatch( queue_t *q )
{
        ...
        if ( q->q_head == NULL )
                return NULL;
        ...
```

图 9-12 修改后的 dispatch 例程

9.4.3 主处理器的进程选择

因为主处理器既可以运行内核态进程，也可以运行用户态进程，所以它可以从任何一个队列中选择进程，如图 9-13 所示。

```
do {
        if ((newproc = dispatch( &master_queue )) == NULL)
                newproc = dispatch( &slave_queue );
} while( newproc = =NULL );
```

图 9-13 选择要运行的新进程的主处理器代码

这段代码优先从主运行队列选出一个进程，因为那些进程只能在主处理器上运行。用户态进程可以在任何处理器上运行，于是这些用户态进程可以留给从处理器去运行（除非没有要运行的内核态进程）。

9.4.4 时钟中断处理

在主从实现中的每个处理器都要接收和处理它自己的时钟中断（clock interrupt），这就能让每个处理器跟踪当前正在执行的进程的时间配额（time quantum）。为了保持单处理机短期互斥策略，所有和时钟中断相关的普通内核活动，比如跟踪每天的时间、执行 alarm 系统调用、重新计算进程优先级等等，都是由主处理器来处理的。在从处理器上的时钟中断处理程序只能检查应该执行一次现场切换的情形。这会在 3 种情况下发生：在当前进程的时间配额过时的时候，在向从运行队列中加入了一个优先级更高的进程的时候（传统的 UNIX 实现通过设置标志来交流这种情况，在时钟中断处理程序中可以检查这个标志），或者向该进程发送一个信号的时候。在前两种情况下，当前正在执行的进程会被放回到从运行队列中。在第三种情况下，必须将该进程切换到主处理器上，以便无需冒着有竞争条件的风险就能运行处理信号所需的内核代码。

9.5　性 能 考 虑

正如在前面的几小节中所看到的那样，主从 MP 内核的实现很直观，而且在概念上也很简单。它既能满足系统完整性的要求，又能保持 8.1.1 小节中讨论过的单处理机外部编程模型。但是，一个重要的问题是它将如何执行。

在理想情况下，随着系统中加入更多的处理器，SMP 的整体系统吞吐量将等于处理器的数量和单个处理器吞吐量的积。因此，双处理器系统应该能够处理一个 UP 两倍的吞吐量，3 个处理器的系统应该有 3 倍的吞吐量，依此类推。一个 MP 实现能够在多大的程度上接近这个理想值则取决于 3 个主要因素：硬件体系结构、应用作业的混合情况以及内核的实现。

如果硬件设计不适用于一个 MP 系统，那么随着处理器的增加，就无法对软件进行调优，从而让一个实现接近性能线性增长的理想状态。例如，如果存储子系统没有给所有的处理器提供足够的带宽（参见 8.2 节），那么就不能充分利用多出来的处理器。考虑到本书的目的，MP 的性能将集中着眼于软件体系结构，并且假定硬件不是限制因素。

应用作业的混合情况是指在系统上运行的应用的数量和类型。通过在不同的内核实现上采用相同的应用组合，就可以形成一个基准（benchmark），借助这个基准就可以测量系统性能，并且和其他内核实现进行比较。重要的是，为了正确地解释测试结果，要理解任何基准所反映的应用混合情况。例如，由一个只运行一次的程序所构成的基准绝对不能体现出 MP 系统比 UP 系统的任何性能提高，因为它并没有给其他处理器提供要做的任务。当基准贴切地模拟出系统用途的模型时，就能获得最有用的衡量结果。

为了展示出对于同样的内核实现，基准测试的结果可能会截然不同，让我们考虑在一个主从 MP 实现上运行的两种不同的基准。第一个基准由一组完全限于计算（compute bound）的进程所构成。一旦被启动执行，它们不会产生缺页错、也没有系统调用，更不执行 I/O 操作。随着给系统加入更多的处理器，在系统吞吐量上，这样的基准会表现出几乎理想的线性增长。因为基准在用户态花掉了它全部的时间，所以完全利用上了所有的从处理器。另一方面，第二个基准由相同数量的进程所组成，不同之处在于这些进程都是限于系统调用的（system call bound），它显示出了相反的结果。如果每个进程仅仅在一个严密的循环中连续不断地执行一个普通的系统调用，比如 getpid 或者 time，使用户级的处理尽可能得少，这个基准就会显示出，不管 MP 系统中有多少个处理器，在性能上比 UP 系统并没有改善。在这种场合下，基准中的所有进程都需要不断得到内核的服务。既然只有主处理器才能提供这种服务，那么在整个基准中，从处理器都处于空闲状态。类似地，限于 I/O（I/O bound）的基准也会显示出采用主从处理机内核的 MP 系统没有改善性能，因为从处理器增加的 CPU 吞吐量并不会加速 I/O 操作。

由此可以得到结论，对于高度交互性（或者 I/O 密集型）的应用环境来说，因为这些应用有大量系统调用和 I/O 活动，所以采取主从实现是一种糟糕的选择。但主从实现对限于计算的科学应用环境来说则是一个良好的选择。这种类型的 MP 内核实现对其他的应用作业混合情况是否有用可以用下面的例子体现出来。

介于限于计算（compute-bound）和限于系统调用（system call bound）两个极端之间的应用混合情况仍然能够从增加的从处理器上获益。比如，如果在 UP 系统上运行着一种应用混合情况，而且发现它耗费了其 50%的时间在内核中执行，40%的时间在用户级执行，而用 10%的时间等待 I/O 操作完成，那么在用户级的 40%就可以分布到双处理器主从 MP 系统中的从处理器上。这能提高至多 40%的性能，这一结果假定所有的用户级任务都是同内核以及 I/O 活动并行完成的。如果事实并非如此，那么对性能的提高就要低一些。对于部分用户级任务来说，往往要依赖于系统调用或者 I/O 操作的结果。所以，如果结果是只有一半的用户任务可以同内核以及 I/O 任务并行完成，那么就会看到性能提高了 20%。多增加处理器并不能进一步提高性能，因为它们对于减少应用混合在主处理器上运行而花费的 50%时间来说，什么忙也帮不上，它们也不能减少等待 I/O 所花费的时间。这是在一个没有限于计算（non-compute bound）的环境中采用主从处理机内核时碰到的主要问题：当增加了更多的处理器时，主处理器迅速成为了一个瓶颈，而且抑制了吞吐量的增长。仅仅为了 20%的性能提升而增加第二个处理器可能并不划算。对于这样的情形来说，超出双 CPU 主从 MP 系统之外的任何东西都几乎一定不会划算。

9.5.1 主从处理机内核的改进

通过放松所有的系统调用都在主处理器上执行这一要求，就可以改善主从处理机内核实现的性能。任何只能返回一条内核信息的系统调用都可以由从处理器来执行，而且很保险。系统调用 getpid 就是一个例子，因为它只返回在进程的生命期内都不会改变的一个值。这类系统调用还有 getpgrp、getppid、getuid、geteuid、getgid、getegid、getrlimit、time、times 和 uname。

类似地，任何系统调用如果只会修改对于进程来说是私有的数据（也就是说，从来不被任何别的进程所修改），那么它也能在从处理器上运行。如果只有一个进程修改数据，就不会造成竞争。这组系统调用的例子有 alarm、nice、profil、setpgrp、setuid、setgid、setrlimit、ulimit 和 umask（在 10.3 节中将对可以省去上锁机制的场合进行进一步的讨论）。

虽然这些修改很容易做到，但是它们不太可能对一个主从系统的整体性能有显著的影响，因为这些系统调用并不常用，而且它们也不会占用多少内核的 CPU 时间量。

因为将所有的内核活动都栓到一个处理器的做法在主从实现中是一个限制因素，所以改善系统性能的唯一方法就是有可能让多个处理器同时在内核中执行。这需要对内核进行大量的修改工作，以避免发生竞争条件。这些技术将在后面的章节中进行探讨。

9.6 小　结

不经过修改，单处理机内核就不能在 MP 系统上运行。UP 内核所使用的短期互斥技术依赖于这样的事实，即内核从来不会同时执行一个以上的进程。MP 系统保持这个策略有效的一种途径就是将所有的内核活动都限制在系统中的一个处理器上。这个主处理器服务于所有的系统调用、中断和任何其他的内核活动。系统中其他的处理器是从处理器，它们只能在进

程处于用户态时才能执行它们。一旦运行在从处理器上的一个进程启动一次系统调用，或者产生一个陷阱，那么它就必须被切换到主处理器上。

在这样的一种实现中，唯一的临界资源就是运行队列。需要有一项技术把多个处理器对运行队列的访问串行化，从而防止发生竞争条件。自旋锁是一种简单的 MP 互斥原语，它可以用于这个目的。自旋锁可以通过更新一个存储器位置的原子操作来实现，于是，在任何时刻，只有一个处理器能成功地获得锁。一旦某个处理器获得了自旋锁，那么所有其他试图获得该锁的处理器就会处于忙等待状态，直到该锁被释放为止。

如果以一种嵌套的方式来使用锁，而且所有的处理器都没有以相同的次序来上锁，那么就会导致死锁。为了防止出现死锁，所有的处理器都必须以相同的次序来锁定嵌套锁。如果一个占有自旋锁的进程执行了一次现场切换，或者如果一个中断处理程序试图获得已经由被中断的进程占有的自旋锁，也会导致死锁。

在主从处理机内核中，主处理器会成为系统整体性能的限制因素。一旦主处理器饱和了，那么再增加更多的从处理器也不会改善性能，因为它们在一般的情况下无法分担内核的负载。有一些简单的系统调用可以放到从处理器上运行，因为没有别的进程会同时修改这些调用所引用的数据。但是因为它们不是占用主处理器大多数时间的系统调用，所以把它们放到从处理器上运行并不会明显地改善性能。只有让内核活动并行地在多个处理器上执行，才能显著地提高性能。

9.7 习　　题

9.1　使用 8.6 节里介绍的 load_linked 和 store_conditional 函数重写 9.2 节中的自旋锁函数 lock 和 unlock。

9.2　代之以使用原子交换重做上一题。使用习题 8.7 中给出的 C 函数原型 swap_atomic。

9.3　编写下面的一组 C 函数（使用给定的函数原型），实现一个无序的、单链接的、以空结尾的列表。这个列表由下面的结构组成：

```
struct element {
        struct element     *next;
        int                 tag;
        int                 data;
};

typedef struct element elem_t;

void initlist( list_t *list );
int search( list_t *list, int tag );
void add( list_t *list, elem_t *element );
elem_t *remove( list_t *list, int tag );
```

函数 initlist 根据需要初始化结构 list_t 的字段。函数 search 以给定的标记在列表中搜索一个元素，并且返回元素中的数据。如果以给定的标记没有找到元素，则返回 0（可以假定决没有任何重复的标记）。函数 search 必须让多个处理器能同时搜索相同的列表（多个同时的读方）。函数 add 把新元素加入到列表的前头。函数 remove 搜索一个给定的标记，如果找到，则断开它的链接，然后返回指向该元素的指针。如果没有找到，就返回 null。函数 add 和 remove 必须以彼此互斥，而且和任何搜索操作互斥的方式执行（每次有一个写方，就会排斥所有的读方）。使用自旋锁来实现必要的互斥。定义数据结构 list_t 包含实现前面的列表所必需的全部锁和数据。因为读方和写方可能以任何次序访问，所以要确保你的实现在所有环境下都能避免死锁。

9.4 考虑这样一个应用，它由两个进程组成，这两个进程彼此通过共享存储来进行通信。两个进程在 UP 系统上运行时彼此保持同步的方式和在 MP 系统上运行时有不同之处吗？考虑进程必须以原子方式更新一部分共享存储的情况。要记住，两个进程可以在不同的处理器上同时运行。

9.5 下面的例程之间会导致死锁吗？在任何时刻，系统中的一个或者更多的处理器都可以调用这两个例程中的任何一个（但是在一个函数执行的同时决不会出现抢先和中断）。假定在调用函数之前没有其他锁，而且临界段也没有再进一步使用锁。解释你的答案。如果会出现死锁，描述能发生死锁的情况。

```
func1() {
        lock( &lock_a );
        lock( &lock_b );
        /* 执行临界段 */
        unlock( &lock_b );
        unlock( &lock_a );
}

func2() {
        lock( &lock_c );
        lock( &lock_a );
        /* 执行临界段 */
        unlock( &lock_a );
        unlock( &lock_c );
}
```

9.6 包含下面的第三个例程，重做上一题。现在，任何时刻在任何处理器上都可以执行3个例程中的任何一个。

```
func3() {
        lock( &lock_b );
        lock( &lock_c );
        /* do critical section */
        unlock( &lock_c );
```

```
        unlock ( &lock_b );
    }
```

9.7　用系统调用 kill 向一个进程发送信号，要求更新接收信号的进程所关联的数据结构。这个系统调用能够由从处理器来执行吗？如果进程向自己发送一个信号，情况又会怎样？

9.8　假定向内核增加一种新的设施，它能记录一个进程的地址空间内每一个页面上发生的缺页错的次数。再增加一个新的系统调用来检索这一信息，这个系统调用能够在从处理器上运行吗？

9.9　如果运行在主处理器上的一个进程用信号 SIGKILL（该信号导致进程被终止）杀死了一个当前在从处理器上运行的进程，那么从处理器上的进程在它终止之前还会继续执行多长时间？主处理器需要做什么来通知从处理器吗？如果从处理器上的进程正在一个无限循环中执行，情况又会怎样？

9.8　进一步的读物

[1]　Arnold, J.S., Casey, D.P., and McKinstry, R.H., "Design of Tightly Coupled Multiprocessor Programming," *IBM Systems Journal*, Vol. 13, No. 1, 1974, pp. 60-87.

[2]　Clark, B.E.J., and Shirnia, A., "Hardware and Software Aspects of Tightly Coupled Symmetrical UNIX Multiprocessors," *Proceedings of the Autumn 1988 EUUG Conference*, pp. 345-55.

[3]　Coffman, Jr., E.G., Elphick, M.J., and Shoshani, A., "System Deadlocks," *Computing Surveys*, Vol. 3, No. 2, June 1971, pp. 67-78.

[4]　Finger, E.J., Krueger, M.M., and Nugent, A., "A Multiple CPU Version of the UNIX Kernel," *USENIX Conference Proceedings*, January 1985.

[5]　Goble, G.H., and Marsh, M.H., "A Dual Processor VAX 11-780," Purdue University Technical Report, TR-EE 81-31, Septerber 1981.

[6]　Habermann, A.N., "Prevention of System Deadlocks," *Communications of the ACM*, Vol. 12, No. 7, July 1969,pp. 373-385.

[7]　Havender, J.W., "Avoiding Deadlock in Multitasking Systems," *IBM Systems Journal*, Vol. 7, No. 2, 1968, pp. 74-84.

[8]　Holley, L.H., Parmelee, R.P., Salisbury, C.A., and Saul, D.N., "VM/370 Asymmetric Multiprocessing," *IBM Systems Journal*, Vol. 18, No. 1, 1979, pp. 47-70.

[9]　Holt, R.C., "Comments on the Prevention of System Deadlocks," *Communications of the ACM*, Vol. 14, No. 1, January 1971, pp. 36-8.

[10]　Holt, R.C., "Some Deadlock Properties of Commputer Systems," *Proceedings of the Third ACM Symposium on Operating System Principles*, October 1971, pp. 64-71.

[11]　Isloor, S.S., and Marsland, T.A., "The Deadlock Problem: An Overview," *IEEE Computer*, Vol. 13£¬No. 9, September 1980, pp. 58-78.

[12]　Kameda, T., "Testing Deadlock-Freedom of Computer Systems," *Journal of the ACM*,

Vol. 27, No. 2, April 1980, pp. 270-80.

[13] Korth, H.F., "Deadlock Freedom Using Ege Locks," *ACM Transactions on Database Systems*, Vol. 7, No. 4, December 1982, pp. 562-632.

[14] Lomet, D.B., "Subsystmes of Processes with Deadlock Avoidance," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 3, May 1980, pp. 297-304.

[15] Minoura, T., "Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography," *ACM Operating Systems Review*, Vol. 13, No. 2, April 1979, pp. 33-44.

[16] Newton, G., "Deadlock Prevention, Detection, and Resolution: An Annotated Bi9bliography," *ACM Operating Systems Review*, Vol. 13, No. 2, April 1979, pp. 33-44.

[17] Noguchi, K., Isao O., and Hiroshi M., "Design Considerations for a Heterogeneous Tightly Coupled Multiprocessor System," *AFIPS Conference Proceedings, 1975 National Computer Conference*, pp. 561-5.

[18] Paciorek, N., LoVerso, S., and Langerman, A., "Debugging Multiprocessor Operating System Kernels," *Proceedings of the Second Symposium on Experiences with Distributed and Multiprocessor Systems*, March 1991, pp. 185-201.

[19] Parnas, D.L., and Haberman, A.N., "Comment on the Deadlock Prevcention Model," *Communications of the ACM*, Vol. 15, No. 9, September 1972, pp. 840-1.

[20] Probert, D., Berkowitz, J., and Lucovsky, M., "A Straighforward Implementation of 4.2BSD on a High-Performance Multiprocessor," *USENIX Conference Proceedings*, January 1986.

[21] Rypka, D.J., and Luciado, A.P., "Deadlock Detection and Avoidance for Shared Logical Resources," *IEEE Transactions on Sofeware Engineering*, Vol. SE-5, 1979, pp. 465-71.

[22] van de Goor, A.J., Moolenaar, A., and Mulder, J.M., "Multiprocessor UNIX: Sparate Processing of I/O," *EUUG Conference Proceedings*, April 1988, pp. 123-34.

[23] Zobel, D., "The Deadlock Problem: A Classifying Bibliography," *Operating Systems Review*, Vol. 17, No. 4, October 1983, pp. 6-16.

采用自旋锁的内核

第10章

本章介绍调整单处理机内核实现，从而让多处理器可以同时在内核中执行的几种方法之一。通过增加自旋锁来保护内核的数据结构，就可以防止出现竞争条件。首先介绍一种主从处理机内核的变体，然后介绍多线程（multithreading）技术。本章还讨论了设计和性能的诸多折衷因素。

10.1 引 言

对于所要求的内核服务超过了适中量的应用环境来说，主从处理机内核的实现并不够用。交互性的应用、执行大量文件 I/O 操作的应用、频繁产生缺页错的应用等，对大量内核活动的需求都很稳定。为了让 SMP 系统对于这些类型的应用来说在经济上也是划算的，就必须让内核能够支持这样的要求，即运行于不同处理器上的不同进程可以同时发出系统调用。这样的一种内核实现允许一次执行内核活动的多条线索，因而称为多线程内核（multithreaded kernel）。要让操作系统成为多线程的，就必须以某种方式标识和保护所有的临界区（critical region）。自旋锁就是一种能够提供这种保护的机制（其他技术在后面的章节中介绍）。

在使用自旋锁的时候，必须决定锁的粒度（granularity）。锁的粒度是指使用了多少个自旋锁，以及任何一个锁保护多少数据。粗粒度（coarse-grained）的实现仅使用几个锁，每个锁保护内核的很大一部分，或许干脆就是一个完整的子系统。细粒度（fine-graincd）的实现使用很多的锁，其中有一些可能只保护一个数据结构元素。是选择使用粗粒度锁还是细粒度锁，要在时间和空间两方面进行权衡。如果每个锁需要一个字的空间，那么极度细粒度的锁可能会为系统中每种数据结构的每个实例都多用一个字的空间。在另一个极端的情况下，一个非常粗粒度的实现几乎不使用额外的空间，但占有锁的时间很长，从而导致其他处理器在等待锁被释放的时候过分自旋。在为了执行简单的活动而必须要获得多个锁的时候，就会发生这种情况。在这些情况下，操控锁本身的开销变得相当大。

在下面几小节中，我将会研究粗粒度锁和细粒度锁，先从有可能的粒度最粗的实现开始介绍。

10.2 巨型上锁

在内核中仅仅使用少量的自旋锁，则被称为采用了巨型锁（这里的每个锁都保护着“巨

大”数量的数据）。采用巨型锁（giant lock）的最简单的内核实现就是只使用一个锁的内核。在这种极端情况下，锁保护着全部内核数据，防止一个以上的进程同时以内核态执行。虽然使用单个锁并不会造就真正的多线程内核，因为一次仅有一个进程占有锁，但是从这里学到的概念和经验教训却可以运用到更复杂的实现上。在任何实现中都可以出现相同的锁争用（lock contention）的基本类型。

在采用了巨型上锁机制且只有一个锁的内核中，在进入内核的任何地方，比如通过系统调用或者陷阱，都要获得这个锁，而当进程发生现场切换或者返回用户态的时候，则要释放这个锁。在进行进程切换的过程中，必须要占有这个锁，因为它保护着运行队列的状态。一旦某个处理器已经选中了要执行的新进程，而且将其从运行队列中移出，该处理器就必须进行检查，看看该进程是以内核态还是以用户态来执行的。如果进程正在以用户态执行，那么处理器就能释放内核巨型锁，运行该进程。如果它处于内核态，那么处理器要获得锁，以使新进程可以在内核中继续执行。

中断可以在任何时间任何处理器上出现。从最低限度上来说，每个进程会接收时钟中断，但也可能接收I/O中断。在MP系统上，例程spl所提供的保护并不充分，因为它们只能够影响执行它们的处理器上的中断优先级。中断可能会在另一个处理器上出现，如果设备驱动程序正在别处运行，那么就会造成一个竞争条件。因此，中断处理程序也必须获得内核巨型锁，因为它们也代表进入内核的一个入口点。但是，要注意，如果发生中断的处理器已经占有了内核自旋锁（如9.3节所阐明的那样），那么就会出现死锁。防止出现这种情况的一种方法是在占有内核巨型锁的处理器上屏蔽所有的中断。如果在另一个处理器上发生了中断（一个没有占有锁的处理器），那么它只会自旋，就像执行系统调用的进程那样等着得到锁。虽然保持了操作系统的完整性，但是它会增加中断的时延。因为任何中断都可能会被拖延任意长的一段时间，所以I/O子系统的性能会受到不利影响。通过用单独的自旋锁来保护每个设备驱动程序，就可以避免出现这样的情况。当发生中断的时候，中断处理程序为各个设备驱动程序获得自旋锁。拥有独立的锁增加了中断处理程序能够为其驱动程序获得锁，而不必像使用内核巨型锁那样可能会有多余延迟时间的机会。以这样的方式来解决性能问题就向更细粒度的实现迈出了第一步。

这里介绍的巨型上锁技术类似于主从处理机内核。在这两种实现中，所有的内核活动都被限制在一个处理器上，这就保证了内核在类似单处理机的环境中运行。系统中的其他处理器可以自由地在任何时刻执行用户态进程。不同之处在于，对于采用巨型锁的内核来说，任何处理器都能执行内核代码，从而没有必要让现场切换到主处理器上。这乍看起来似乎是个优点，但是采用巨型锁的内核其性能可能比主从处理机内核更差。对于采用巨型锁的内核来说，如果当一个进程执行一次系统调用时，另一个处理器却占据着内核锁，那么产生新系统调用的处理器就会进入空闲，同时自旋等待该锁。在主从处理机内核中，处理器可以选择另一个用户态进程来运行，而不只是空等待。如果占有内核锁的处理器正在执行一个长时间的系统调用，那么其他处理器会自旋很长一段时间才结束。如果已经占有锁的处理器做现场切换又进入另一个内核态进程，继续占据该锁，从而让它有可能锁住其他处理器，无限期地不让它们访问内核，那么会让这种情况更加糟糕。在任何实际的MP内核实现中，必须避免类似这样的情况。

还是有可能通过使用额外的自旋锁来弥补这些缺点的。首先，不希望当处理器能够执行

另一个用户态进程的时候，却让它们空自旋，等待内核锁。要做到这一点，可以使用一种新型的自旋锁操作，该操作能够有条件地获得锁。这可以由图 10-1 中的代码来实现。

```
int
cond_lock(volatile lock_t *lock_status)
{
        if(test_and_set(lock_status) == 1)
                return FALSE;
        else
                return TRUE;
}
```

图 10-1　有条件地锁定一个自旋锁

函数 cond_lock 做一次获得自旋锁的尝试。如果锁是空闲的，那么 cond_lock 就得到了它，并返回真（true）。如果锁已经在使用了，那么它就返回假（false）。现在，一个处理器能够检测到内核锁什么时候在使用，而且如果在用的话，就执行现场切换，切换到另一个用户态进程，而不是等待内核锁。在 SMP 内核实现中，有条件获得锁的概念通常都是很有用的技术。

其次，保护运行队列所用的锁必须和正常的内核巨型锁分开。这对于当一个处理器不能获得内核巨型锁的时候，它仍然能够在很短一段时间内获得运行队列锁，于是它可以把它正在运行的进程排入队列，并且选择一个要执行的新的用户态进程来说是必要的。

注意，这些改进只能让采用巨型锁的内核的性能提高到主从处理机内核的水平。两种方法之间唯一的不同之处在于，现在任何处理器都能称为主处理器，而不是永久性地指定一个处理器作为主处理器。既然巨型锁的方法实现起来更为复杂，所以几乎没有哪个地方会用到它。为了超越主从处理机内核的性能限制，必须有更高的内核并行性。

10.3　不需要上锁的多线程情况

让一个内核多线程化的第一步就是确定出那些没有必要用到上锁机制的实例。这可以避免使用过多的锁，从而既节省了时间又节省了空间。它还消除了在这些情形之下任何可能的锁争用问题，因为没有用到锁。在 9.5.1 小节中提及的所有系统调用不用锁都能够运行。如前所述，只有当使用了对于一个进程来说是私有的数据结构时，才会出现可以使之成为可能的条件。（注意，如果在对进程进行调试，那么即使是一个进程的私有数据结构，可能也会由另一个进程考察或者修改。在允许访问被调试进程的私有数据之前，往往要挂起该进程。这就防止了在调试器和被调试进程之间发生竞争条件，因此不需要再更进一步考虑这种情形。）

每个 UNIX 进程都有一个用户区（user area，缩写为 u-area，u 区），它是包含一个进程大部分私有数据的内核数据结构。它包含诸如当前信号处理程序（signal handler）设置、当前系统调用的参数、寄存器保存区（在系统调用的执行过程中保存用户寄存器的值）以及其他私有的数据这样的内容。只有和 u 区相关的进程才能操控这些数据。除了在调试过程中之外，进程绝对不会共享 u 区，没有哪个进程能访问其他进程的 u 区。因此，当进程正在从它

自己的 u 区读取数据或者向它自己的 u 区写入数据时，不需要上锁。

另一个私有数据结构是进程的内核栈（kernel stack）。每个进程都有它自己的内核栈，在代替用户进程执行系统调用的时候会用到内核栈。因为没有哪个进程要访问另一个进程的内核栈，所以在使用位于内核栈内的任何数据时也不需要上锁。这就正好包括了所有的 C 局部变量（也称为自动变量）和函数参数。

最后，有些系统调用为私有数据动态地分配额外的空间。这样做是为了防止所需的数据量比内核栈能容纳的大，或者比 u 区中永久分配的多的情况。保存作为系统调用参数一部分的路径名的空间，以及传递给系统调用 exec 的参数列表都是这类补充数据区的例子。这类数据可以看作是对进程的私有内核栈或者 u 区的临时扩展，所以，只要没有别的进程会访问到它，就不需要上锁。

每个进程还有一个进程表项。和 u 区不同，部分进程表项可以由系统中的其他进程访问和修改（在除了调试之外的情况下）。大多数这类情形都需要上锁，但即使在全部的进程之间共享数据结构，也有一些情形仍然没必要上锁。例如，进程表项保存有进程 ID、进程的用户 ID 和组 ID。在进程的生命期内，进程 ID 始终不变；因此，任何进程都能够在任何时刻读取这个值，而不需要上锁。类似地，用户 ID 和组 ID 只能由和进程表项相关联的进程来进行修改（通过执行 setuid、setgid 或者其他系统调用）。其他进程可以读取这些值（例如，以此了解它们是否有权限向这个进程发送信号），但都不会修改它们。因此，任何进程都可以无需上锁就能读取这些数据。

注意，如果在其他进程读取一个进程的这几个 ID 号时改变了它们，那么在其他进程之间存在一种本来就有的竞争条件。即使增加了上锁机制来防止在改变这些值的时候读取它们，仍然会导致同样的竞争条件：在这些值被改变之前读取它们的进程看到的是它们以前的值，而在这些值修改之后读取它们的进程看到的则是新的值。上锁对于这种竞争毫无影响。因此，最好完全放弃上锁机制，节省空间，消除不必要的锁争用。总结归纳这种情形，对于任何一个字的存储器中的一份数据来说，在只有一个写方的任何时刻都可以省去上锁。省去上锁的能力取决于这样的事实，即顺序存储模型保证对存储器中任何一个字的存储操作都是原子操作。

类似于此的本来就有的竞争条件在内核的其他地方也会出现。在这些情况下也可以不用上锁。但是，在必须对一个以上的字进行原子更新，或者当需要读-改-写操作的时候，就不能省去上锁。对于这些临界段来说，必须在粗粒度上锁或者细粒度上锁间进行选择。

10.4 粗粒度上锁

可以对只使用了一个锁的巨型上锁技术进行扩展，通过增加更多的锁带来一些额外的并行性。为内核中的每个子系统，比如在不同处理器上的进程，使用独立的锁，如果它们只影响到不同的内核子系统，那么就可以同时执行系统调用。

根据这样的粒度水平，可以分配给进程管理子系统一个锁，从而在系统中对进程表项里保存数据的全部修改操作提供保护。为了了解这是如何起作用的，考虑一个进程正在向另一个进程发送信号的情形。在大多数 UNIX 内核中，进程的挂起信号由保存在进程表项中的一位掩码来表示。要发送一个信号，必须用读-改-写操作来更新掩码，该操作设置正在发送的

信号所对应的比特位。因为多个进程可能会同时试图向同一个进程发送信号，所以这就会需要上锁机制。通过要求正在发送信号的所有进程锁住一个共同的进程管理锁，就可以消除竞争条件。对于进程表项中可以由别的进程修改的其他字段来说也是如此。

依此类推，可以用一个自旋锁保护文件管理子系统。在所有和文件系统有关的系统调用（比如 open、read、write 和 close）期间，都要获得并且占有这个锁。可以用另一个锁保护虚拟存储子系统。当一项操作影响到内核中一个以上的子系统时，就会带来死锁问题。例如，要弥补一个缺页错，可能会要求从文件中读取数据。这样的一次操作首先要加虚拟存储子系统的锁。之后，要从文件系统中读取数据，还必须要获得文件子系统的锁。类似地，如果另一个进程发起了一次文件操作，比如说截断文件，它就会首先加上文件子系统锁，随后，为了使文件被截断部分的任何页都失效（该文件可能已经被映射到某些进程的地址空间中），它又不得不加上虚拟存储子系统的锁。如果这两项操作同时发生，它们之间就会出现死锁，因为它们正好以相反的次序要求获得锁（参见 9.3 节）。

为了防止这类死锁，必须在内核中一致地定义和使用一种上锁的次序。例如，上锁的策略可以是，当同时需要文件系统和虚拟存储子系统锁的时候，总是先获得文件系统锁。因为截断文件的代码以这一次序获得锁，所以不需要对代码进行修改。但是，必须对用于虚拟存储子系统的代码进行修改，使之符合所定义的上锁次序。实现这一点有几种方法。

最简单的方法是，在开始任何虚拟存储操作之前首先加上文件子系统锁。一旦知道不再需要文件系统操作的时候，就释放文件锁。虽然这种方法易于实现，但是因为它倾向于抵消为两个子系统使用单独的锁所要达到的目的，所以并不可取。相对于使用一个锁来保护两个子系统来说，它几乎没有多提供并行度。

第二种解决方案是，当虚拟存储子系统发现需要加上文件子系统锁的时候才这样做。为了避免死锁，代码必须首先释放虚拟存储锁，然后加上文件锁。一旦完成文件操作，就可以再次获得虚拟存储锁。在 MP 内核中，释放一个锁以维护正确的上锁次序的技术并不罕见。但缺点是，在释放锁之前，所有的虚拟存储子系统数据结构都必须处于一致的状态。一旦锁被释放，在不同处理器上的另一个进程就可以自由地发起一次虚拟存储操作。这些影响必须给予考虑，可能要对代码进行其他修改，以防止在不同的进程之间出现竞争条件。例如，一旦文件子系统完成了它的那部分任务，可能有必要重新发起虚拟存储操作。当别的进程可能已经改动过虚拟存储数据结构，从而使以前考察过的状态不再有效的时候，就会有这个必要。

第三种技术是，利用有条件的上锁操作来尝试避免释放虚拟存储锁的要求。通过让虚拟存储代码在发现它必须执行一次文件操作的时候有条件地获得文件系统锁，就可以做到这一点。如果锁是空闲的，那么它就获得这个锁，并且能够继续执行，而不必释放虚拟存储锁。在这种情况下，既没有必要确保虚拟存储数据结构的一致性，也没有必要以后重新发起操作，因为虚拟存储锁从未释放过。但是，在有条件的上锁操作失败的情况下，虚拟存储锁还是必须照前面那样释放。

总而言之，在不按照次序获得锁的时候，使用有条件的上锁机制避免了死锁问题，因为不可能出现为了等待锁而永远自旋下去的情况。如果不能有条件地尝试获得锁，那么进程可以采取另一种措施，这和前面介绍的解决方案是一样的。虽然这个原则一般都有效，但是在粗粒度的系统上，当系统调用速度很快的时候它却不太可能有用。在这类情况下，子系统锁几乎不断被锁住，从而让有条件的上锁操作几乎不太可能成功。

每个内核子系统都可以用上述的方式上锁。剩下来的问题则发生在内核中正在执行的一个进程必须睡眠的时候。正如 9.3 节所述，进程在占有自旋锁的时候不能睡眠（现场切换），因为这会导致死锁。因此，进程所占有的任何子系统锁都必须先行释放。因为这样做可以让其他进程访问到受这些锁保护的数据结构，所以必须确保它们和睡眠之前状态的一致性，或者确保适当长地拥有锁。在单处理机版的内核睡眠的任何情形下，都会出现这两种情形之一。在单处理机上进行现场切换和在 MP 上进行现场切换发生竞争条件的可能性一样大，因为其他进程可以运行和访问相同的数据。因此，调整这段代码使之适用于 MP 环境，只要正确地处理好自旋锁即可。

实现这一点的一种方法是，将某个进程正占有的锁以及获得它们的次序记录在该进程的 u 区中。于是，现场切换代码能够释放正被挂起的进程的锁。当进程在睡眠之后重新执行的时候，它必须获得它以前占有的所有的锁。这也可以由现场切换代码来处理。

虽然一个粗粒度的 MP 内核开始允许在一个以上的处理器上并行地进行内核活动，但对于系统调用密集型的工作量来说，它依然有太多的局限。对于那些要求有大量内核服务的混合应用任务来说，它一般不会比主从处理机内核有太大的性能提升。例如，如果应用需要文件 I/O，那么在粗粒度的内核中，所有这类活动仍然是单线程的，因为即使每个进程使用了不同的文件，文件系统锁却只有一个。

10.5 细粒度上锁

10.5.1 短期互斥

细粒度上锁的目的是为了通过不同的处理器提高并行内核活动的数量。虽然许多进程可能想要使用同一个内核子系统，但是常见的情况却是，它们正在使用有关数据结构的不同部分。通过给每个文件一个独立的锁，如果几个进程正在使用不同的文件，那么就有可能在文件系统中同时执行。这样一来，可以把每个子系统分成独立的临界资源，每个临界资源有自己的锁。这需要分析出单处理机短期互斥技术所保护的全部数据结构，并且增加适当的自旋锁。介绍 UNIX 内核中所有可能的这类独立临界资源超出了本书的范围。下面是一些使用自旋锁的细粒度上锁的例子，源于其中的概念适用于整个内核。

将细粒度上锁机制应用到前面的信号一例中，我们能够在每个进程的进程表项中分配一个自旋锁，保护挂起信号的位掩码。这将让包含不同进程的信号操作在系统内不同的处理器上同时执行。要发送一个信号，发送方进程必须首先锁住接收方进程的信号锁。然后，它可以执行修改该进程的信号掩码以包含将发送的信号的临界段代码。

继续这个例子，每个进程必须周期性地检查，了解它是否已经有了任何挂起信号。在进程做检查的时候不需要让它占有其信号锁，因为这并不会解决在发送信号的进程和检查其掩码看是否有新信号的接收进程之间存在的固有的竞争条件。上锁的话，新信号能够正好在接收方进程检查其掩码之前发送；不上锁的话，新信号正好在接收方进程检查其掩码之后发送（另一种固有的竞争条件）。但是，当进程在更新它自己的信号掩码时，它必须占有其信号锁。当进程向它自己发送信号，或者当它正在处理发给它的信号，并且正在清除掩码中的相应位

时，就会出现这种情况。这两者都是必须按照原子方式完成的读-改-写操作，它们要求上锁。

自旋锁也适用于保护内核的许多全局数据结构和列表。例如，每个已安装的文件是由一个数据结构来表示的，所有的结构都维护在一个链表中。对这个列表搜索、增加或者删除的操作都是临界段。可以分配一个自旋锁来保护这个列表。类似地，内核也维护着各种数据结构的空闲列表（缓冲区、物理页面帧等）。其中的每一种也都可以由一个自旋锁来保护。

作为更进一步的例子，考虑这样的情形，即正在创建一个新进程，必须分配一个新的进程 ID（pid）。必须保证新的 pid 是唯一的，这在典型情况下是通过让一个计数器保存下一个要用到的 pid 来实现的。有些实现使用 30 000 作为 pid 的最大值，于是计数器会周期性地折返。因此，计数器中给出的下一个 pid 的值必须要检查，以确保它尚未被前面创建的进程使用。这可以通过扫描系统中的所有进程表项，确定没有进程有那个值来做到。因为两个处理器可以同时尝试创建一个新的进程，分配唯一 pid 的代码也必须是临界段，以保证两者不会分配相同的 pid。这可以通过用自旋锁保护计数器来做到。每次使用计数器的时候，要获得自旋锁，并且将当前计数器的值作为尝试使用的下一个 pid。于是，在释放锁之前，计数器要累加 1。这样一来，每个处理器从计数器读取到的始终是唯一的值。

从计数器获得一个新的 pid 之后，必须进行搜索，以确保没有哪个进程正在使用那个值。如果进程表是一个长度固定的数组，那么所有的处理器能够同时扫描数组而不必占有任何锁。不会出现任何竞争条件，因为扫描一个固定长度的数组并不依赖于其中的数据。如果进程表项保存在一个链表中，那么在遍历列表的时候，必须使用自旋锁来防止列表被改变（比如，从列表中删除一个已经终止的进程）。这以单线程的方式访问列表，一次仅仅允许一个进程扫描列表。正因为如此，可以去掉用于保护计数器的自旋锁，而转由进程表的列表自旋锁来保护。把计数器的值作为一个独立的临界资源加以保护没有任何好处，因为不管怎样，所有的处理器都被进程表的锁给串行化了。如前所述，应该避免锁的数量过多，以节省空间和上锁的开销。（扫描进程表的操作适合采用多读锁（multireader lock），这将在下一章中进行介绍。）

10.5.2　长期互斥

采用单处理机短期互斥来防止竞争条件的另一个地方是在获得长期互斥锁的代码中，比如 8.5.3 小节中的对象锁。回顾图 8-8 中的代码，注意到 lock_object 函数依赖于这样的事实，即只有一个进程能够立即执行该函数。这样一来，作为一项原子操作，该函数可以检测标志，然后设置它（如果锁是空闲的），或者进入睡眠。通过增加一个如图 10-2 所示的自旋锁，可以在多线程内核上重新构建这个临界段。

自旋锁 object_locking 的使用正确地保护了函数 lock_object 中的临界段。在长期锁空闲的情况下，测试并设置标志的操作可以是原子操作。

要记住，当进程正在睡眠时，不能占有自旋锁。但是，自旋锁不能在调用 sleep 之前释放，因为在函数 unlock_object 中的 sleep 和 wakeup 之间会造成竞争（参见 8.6 节）。决定睡眠以及让进程进入那一状态所必需的工作必须以原子的方式完成，这意味着在整个过程中必须占有自旋锁。可以使用与 10.4 节给出的相同的解决方案，在这个方案里，锁被现场切换代码记录在 u 区中，并且由其释放。和以前一样，当进程被唤醒的时候，要重新获得锁。因为

进程仍然要在函数 lock_object 的临界段中执行，从而需要在测试标志状态以及决定采取何种行动（不是再次睡眠，就是为自己设定锁）的同时占有锁，因此需要这样做。

```
void
lock_object( char *flag_ptr )
{
        lock( &object_locking );

        while( *flag_ptr )
                sleep( flag_ptr );

        *flag_ptr = 1;
        unlock( &object_locking );
}
```

图 10-2 锁住一个对象的 MP 代码

另一种解决方案是，直接把要被释放以及重新获得的自旋锁的地址传递给 sleep 函数（参见 12.4.3 节中这样做的一个 MP 原语的例子）。这是一个结构化更好的方法，因为它不需要使用 u 区中的进程全局数据。然而这里的局限在于，当进程调用 sleep 的时候，它只能占有一个自旋锁。当 lock_object 调用 sleep 的时候可能就是这种情况，因为内核在试图获得一个长期锁的时候不会占有其他自旋锁。

unlock_object 的代码也需要占有自旋锁，因为清除标志并且唤醒等候长期锁的任何进程是一个临界区（参见图 10-3）。

```
void
unlock_object( char *flag_ptr )
{
        lock( &object_locking );
        *flag_ptr = 0;
        wakeup( flag_ptr );
        unlock( &object_locking );
}
```

图 10-3 给一个对象解锁的 MP 代码

object_locking 自旋锁的使用解决了 8.6 节中描述的与长期互斥有关的竞争条件。一旦 object_locking 函数返回，进程就得到保证，能像单处理机实现那样互斥地使用对象。它现在能够修改对象内的数据结构，而不必进一步上锁。它也能睡眠，等待诸如磁盘 I/O 这样的事情，而不必冒有竞争条件的风险。长期互斥的任何实例都能够以同样的方式给予保护。

10.5.3 和中断处理程序的互斥

正如 8.5.2 小节所介绍的那样，当基准驱动程序代码和中断处理程序之间共享数据结构

时，单处理机通过屏蔽中断来防止出现竞争条件的技术，在多线程 MP 内核中还不充分。如果图 8-7 所示的临界段要在一个处理器上执行，那么只会屏蔽在那个处理器上出现的中断。如果在别的处理器上出现中断，那么立即就会有两个处理器同时访问、而且还有可能更新临界资源。主从处理机的内核实现通过在单个处理器上执行所有的内核代码以及所有的中断处理程序来避免这个问题。在采用多线程的内核中必须采取进一步的措施。

既然这些临界段需要短期互斥，那么可以使用自旋锁来防止出现竞争。可以如图 10-4 所示来增强图 8-7 中的 UP 代码。

基准代码

```
s = splhi();
lock(&driver_lock);
counter++;
unlock(&driver_lock);
splx(s);
```

中断处理程序代码

```
lock(&driver_lock);
counter++;
unlock(&driver_lock);
```

图 10-4　中断处理程序互斥

自旋锁和 spl 保护一起运用，可以解决两种可能的竞争条件。如果中断发生在执行基准代码片段的同一处理器上，则 spl 保护可以防止与中断处理程序的竞争，就如同它在 UP 系统上起的作用一样。如果中断处理程序是在和执行基准代码不同的处理器上执行的，那么自旋锁起到保护作用，防止竞争。在这种情况下，在别的处理器完成临界段之前，中断处理程序只做自旋。注意，单独使用自旋锁是不够的，基准代码还必须提高 spl 的级别。除非中断被屏蔽了，否则中断可能出现在占有自旋锁的进程上，造成在中断处理程序试图获得同一个锁的时候发生死锁。

10.5.4　锁的粒度

一旦按照前面的章节所介绍的那样，在整个内核中增加了自旋锁，那么任何处理器都能够执行任何进程，不论它是以用户态还是以内核态来执行该进程的。在这样的多线程内核中，在每次现场切换的时候，每个处理器都会选择运行队列中优先级最高的进程。下一个问题是，细粒度实现应该达到什么样的程度？

如前所述，加锁的粒度能够从粗到细。在决定一个实现应该达到什么样的细粒度时，不仅必须考虑性能，而且必须考虑操作本身的语义。例如，从系统调用 read 和 write 的方面来说，UNIX 外部编程模型的定义保证了对普通文件的 I/O 操作是原子的。传统的 UNIX 内核实现通过使用前面描述的对象上锁技术，即一次仅仅允许一个进程在文件上执行操作，就能够做到这一点。但是，通过允许多个只是读文件的进程同时执行系统调用 read 就可以合法地使锁的粒度更细。这在频繁读取一个文件（比如口令文件）的情况下（因为把用户名转换为 ID 及其逆向转换都是一项频繁的操作），或者在目录中查找文件的情况下，都具有优势。如果进程是对同一个文件中无关的部分执行读写操作，也有可能让它们同时对该文件执行这些操作。实际需要以原子方式完成的操作仅仅是，有两个或者两个以上的处理器同时在一个文件的同一部分上执行操作，而且至少有一个进程要写文件。比如，当多个进程试图向一个日志文件增添信息的时候，就会发生这样的情况。这体现出了锁粒度的界限在哪里：它不能比

操作许可的语义更细。

接下来，在实现中使用的独立的锁的数量同每个锁要保护的数据量之间进行对比是有差异的。例如，上一节介绍过一个用于信号的实现，其中有一个独立的自旋锁，它保护每个进程表项中挂起信号的位掩码。这里的粒度级别比单个进程管理子系统锁要细，它能让信号操作同其他的进程管理操作（比如给新进程分配进程 ID）并行进行。注意，通过使用单个锁，保护所有进程中的信号位掩码，而非给每个进程使用一个独立的锁，也能达到同样的目的。在这两种情况下，诸如发送信号和分配进程 ID 这样的无关操作都可以并行进行。区别在于，相同类型的多项操作（在这里是发送信号）是否能够同时发生。类似地，可以在每个容易被长期上锁的对象上使用一个独立的自旋锁，或者可以不管正在哪个对象上操作而使用单个锁，就像函数 lock_object 和 unlock_object 中所展示的那样。两种实现中的任何一种都既保持了外部编程模型，也保持了操作系统的完整性。但是，它们可能在性能上有差别。

当决定采用什么水平的粒度时，必须既要考虑系统中处理器的数量，也要考虑期望的应用数量。一般而言，处理器数量少的系统（例如，2 个或者 4 个）不会因为给每个进程表项中的信号一个独立的自旋锁来增加粒度，而比出于相同目的但只用一个自旋锁的做法获得太多的好处。因为临界区太小，所以两三个处理器不太可能会同时执行它。即使它们确实同时执行了临界区，短小的临界区也确保了别的处理器只会被短暂地延迟一下，边自旋边等待。随着处理器数量的增加，多个处理器同时发送信号的可能性也增加了。如果应用大量地使用信号（例如，作为一种进程间通信机制）的话尤其如此。在这样的情况下，对单个全局自旋锁的争夺可能会太厉害，于是考虑给每个进程独立的锁才会有意义。注意，在没有或者几乎没有争用信号自旋锁的情况下，性能都是一样的。无论是单个全局锁还是每个进程一个独立的锁，都需要有一套自旋锁的操作。唯一的区别在于，独立的锁需要更多的空间，以及降低锁争用的几率。

决定粒度达到什么程度，与在巨型上锁和多线程之间进行抉择是分开来考虑的。决定采用多线程倾向于选择以独立的锁来保护信号掩码。接下来要决定应该使用多少个独立的锁来实现这一点：一个全局锁，还是每个进程一个锁。无论选择什么样的粒度，无关的内核操作还是有可能并行发生的。正是这一特性让多线程内核有潜力比主从处理机内核以及巨型锁内核实现的性能更好。

10.5.5 性能

为了把多线程内核的性能同第 9 章中主从处理机内核实现的性能进行比较，还要再次混合使用两种极端的应用任务，即受限于计算的进程和受限于系统调用的进程。和以前一样，受限于计算的任务能够在所有的处理器上并行执行，而不会发生争用。多线程内核既不能有助于改善这类应用的性能，也不能损害它们的性能，这不足为奇。差异则体现在当运行系统调用密集型（system-call-intensive）应用时系统的性能上。如果在这些混合应用任务中的进程使用了独立的内核资源，它们各自的数据结构由独立的锁来保护，那么这些进程不管是处于用户态还是内核态，都不会争用相同的锁，并且能够在不同的 CPU 上同时并行运行。这是可能出现的最好情况，因为应用的负载可以随着系统中处理器数量的增长而线性地增加。遗憾的是，应用可以执行而不会发生任何锁争用的情况是很少见的，即使在细粒度的实现中也是

如此。总是会有这样一些共享的数据结构，所有的进程都必须使用它们。运行队列就是个不错的例子，数据结构释放列表也是。

类似地，使用独立文件（所有的文件都由不同的锁加以保护）的进程可能仍会发现，如果这些文件都保存在相同的驱动器上，那么它们仍然要争用共享的资源（磁盘驱动器本身）。这些因素将会限制多线程内核的性能，让它难于达到随着处理器的增加而线性地提高性能的目的。为了获得最大的性能，必须针对特定的应用任务混合情况来对实现进行调优。

10.5.6　内核抢先

最初的单处理机 UNIX 内核在设计上是非抢先的，这简化了短期互斥的实现。我们看到，为了让多线程内核实现能在一个 MP 系统上正确地运转，必须用自旋锁对需要短期互斥的所有情况加以保护。因为短期互斥现在已经做成是显式的了，所以原来的、不抢先内核中正在执行的进程的 UP 方法可以不那么严格。只要一个进程没有占有自旋锁，也没有屏蔽任何中断，那么它就没有在一段临界区内执行，因而可以被抢先。当某个内核进程在这些情况下被抢先的时候，要保证该进程当时只在访问私有数据。例如，这项观察措施在实时系统上有用处，因为它能在可以运行优先级更高的进程时降低执行延迟。注意，被抢先的进程可能会占有一个长期锁，但是这并不表示破坏了相关的临界区，因为在被抢先的进程释放它之前，没有别的进程能够获得长期锁。

10.6　sleep 和 wakeup 对多处理机的影响

在 8.5.3 小节中，我们看到，wakeup 函数能够让所有正在相应事件上睡眠的进程被唤醒和运行。在多个进程等待一个长期互斥锁的情况下，第一个要运行的进程发现锁是空闲的，便可以获得它。在有些情况下，进程能够在现场切换之前执行完临界区并释放锁。例如，在采用文件锁的情况下，当所需的文件数据已经处于缓冲区中的时候，就会出现上述情形。这意味着，在单处理机上，从刚被唤醒的一组进程中，要执行的下一个进程也能够立即获得锁。这是一种人们想要得到的行为，因为如果在第一个进程释放锁之前，被唤醒的其他进程就要运行，那么它们会发现锁又处于忙的状态，于是再度入睡（进程颠簸）。因为单处理机内核确保了一次只能执行一个进程，而且该进程不会被抢先，所以它有可能可以在现场切换之前释放锁，于是避免了颠簸。

但是，在多处理机上，一旦在同一个事件上睡眠的一组进程被唤醒，它们就都有执行的资格，而且可以被不同的处理器从运行队列中挑选出来，并行地运行。幸运的是，如果发生了这种情况，在 lock_object 代码中使用自旋锁就能够防止出现竞争条件。某个进程能够先获得自旋锁，于是它就首先有机会为自己获得对象锁。别的进程则进入自旋，等待第一个进程释放自旋锁。发生这种情况的时候，下一个进程能够获得自旋锁，并且发现对象被锁住了，于是它会再度入睡。这种情况会继续下去，直到被唤醒的一组进程全都再度入睡为止。结果是，在多处理机上使用 sleep/wakeup 的时候，出现颠簸的情况更多了，因为获得锁的第一个进程不太可能在被唤醒的其他进程被调度到其他处理器上运行之前就释放锁。对锁的争夺以

及由此造成的颠簸通常称为 thundering herd（群雄纷争）现象。

这个问题可以通过使用 wakeup 的变体来避免，这个变体称为 wakeup_one，在调用它的时候只唤醒一个进程。如果有多个进程在相同的事件上睡眠，那么将选择优先级最高的进程。如果所有进程的优先级都相等，那么就按照先入先出（FIFO）的顺序唤醒它们。和以前一样，唤醒操作也没有记忆性，所以说，如果调用了 wakup_one，但在事件上没有正在睡眠的进程，那么它什么也不做便返回。在实现长期互斥中使用 wakeup 的所有情况都可以代之以转而使用这个新函数，从而消除了前者可能造成的颠簸现象。

注意，重新实现 wakeup 本身，使之具备 wakeup_one 的功能，这种提法并不正确。这里的问题在于，单处理机内核的一些部分要依赖于唤醒所有睡眠进程的 wakeup 函数的特性。将 sleep/wakeup 用于进程同步而不是实现长期互斥就是这种情形。例如，可能会有许多进程等着向一个已经满了的管道写入数据（管道是一种 UNIX 进程间通信机制，它在进程之间传送一个数据字节流）。当读方从管道读取数据时，它就会唤醒所有正在等候向管道写入数据的进程。这是最简单的实现，因为它不需要读方知道正在等候的写方有多少，以及来自它们的写入操作的全部数据量是否会再次填满管道。相反，读方只是把它们全都唤醒，并且让写方互相争夺管道。在采用这种实现的情况下，只唤醒一个写方会让其他写方被阻塞，不会执行。这里的问题在于，当一个写方完成它的写入操作之后，即使管道中还有空间，该实现也不会让其他写方被唤醒。

当然，还是有可能重新实现内核中依赖于 wakeup 行为的那些部分。既然只要有多个进程正在某个事件上睡眠，wakeup 的语义就能导致颠簸，那么这通常有更好的效果。在管道的例子里，管道自身有一个长期互斥锁，每个写方都必须占有该锁。于是，如果有若干写方被同时唤醒，那么一次只有一个写方能够获得锁。其余的则再度入睡，但是现在却是在不同的事件上睡眠的：是锁自身，而不是等待管道中的空间可以使用。这又再次导致出现颠簸。

如果内核的所有部分都被重写，这次是使用 wakeup_one，那么就可以去掉 wakeup 的原有功能。单处理机 wakeup 函数的缺点致使有些实现用新的、更适合于 MP 环境的原语替换了 sleep/wakeup。

10.7 小　结

通过增加自旋锁，可以修改一个单处理机内核实现，使之在多处理器上运行。增加的这些自旋锁保护着短期互斥技术隐含保护的临界段。有些地方根本没必要用上锁机制。只要用到的数据是一个进程的私有数据，那么就不需要用上锁机制，因为不会有竞争条件。对进程的内核栈的访问、所有的局部变量和函数参数以及进程的 u 区，都是不需要上锁机制的私有数据结构的例子。

可能覆盖所有处理器的、并发的内核活动的数量部分取决于多线程化的程度。粗粒度的实现使用的锁很少，但是在应用频繁需要内核服务的时候，就会遇到和主从处理机内核实现一样的缺点。细粒度实现试图通过用不同的锁保护不同的数据结构来克服这些缺点。如果应用倾向于使用内核中有不相干数据结构的设施，比如使用不同的文件，那么多个处理器就可

以同时以内核态执行。注意，为了获得高性能的多线程内核而给数据结构的每个实例都使用一个独立的自旋锁的做法，并不总是必要的。目的在于，确保无关的进程能够不发生争用地执行它们的内核活动，而不仅仅是在内核中散布许多个锁。

多线程内核的一个优势就是，所有的临界段不是由长期锁或者自旋锁来保护，就是通过提高中断优先级来保护。这使得没有占有任何自旋锁以及没有屏蔽任何中断的内核进程有可能被抢先。对于实时系统来说，这种功能非常重要。

单处理机 wakeup 函数的作用，即唤醒为某个事件而睡眠的所有进程，对于 MP 系统来说一般是不希望有的，因为这些进程有可能在不同的处理器上同时运行。因为这些进程可能在它们运行的时候争夺一个共用的锁，所以如果它们中的大多数在不能获得锁之后都必须再度睡眠的话，就会导致出现颠簸现象。通过引入一种新的唤醒函数，一次只唤醒一个进程，就能解决这个问题。重写内核中依赖于单处理机 wakeup 函数行为的那些部分，也是可以的。

10.8 习　　题

10.1　解释当使用 10.2 节中的巨型上锁内核技术时，如何处理时钟中断。要考虑到两种情况：当时钟中断发生在当前正占有内核自旋锁的处理器上的时候，以及当时钟中断发生在正运行用户态代码的处理器上的时候。

10.2　在图 10-1 所示的条件上锁代码中，在返回失败之前，值得多次尝试获得锁吗？

10.3　考虑 MP 系统不支持顺序存储模型的情况。假设对同一位置的并行读取操作仍然能够正确执行，可是当一次写操作正在进行之中时对同一位置的读操作则可能会导致返回不正确的结果。但是，假设硬件保证了 test-and-set 指令始终能够正确执行，即使同时对内存中同一个字执行测试-设置操作时也是如此。这样的存储模型将对本章介绍的 MP 内核技术有什么样的影响？

10.4　使用本章介绍的细粒度上锁技术重写这里给出的单处理机代码，使之在 MP 上正确执行。这段代码实现了一个简单的邮箱风格的进程间通信设施。进程使用 send_msg 把消息投入到邮箱中来发送消息，其他进程能够使用 recv_msg 检索消息。一次在邮箱中只能有一条消息。一旦邮箱中有了一条消息，那么其他的发送方就必须等着邮箱为空。除了修改代码之外，还要回答下面的问题。在单处理机上，有可能让发送方和接收方同时都在邮箱的地址上睡眠吗？考虑让两个函数都在同一地址上睡眠时的影响。在 send_msg 中的 wakeup 调用会唤醒在 recv_msg、send_msg 中睡眠的进程吗？还是两者中的睡眠进程都会被唤醒？类似地，recv_msg 中的 wakeup 调用会唤醒在 recv_msg、send_msg 中睡眠的进程吗？还是两者中的睡眠进程都会被唤醒？MP 的版本应该使用 wakeup 还是 wakeup_one？解释原因。

```
viod *mailbox = NULL;


void
send_msg( void *msg )
{
        while(mailbox != NULL )
```

```
                sleep( &mailbox );

        mailbox = msg;
        wakeup( &mailbox );
}

void *
recv_msg()
{
        void *msg;

        while(mailbox == NULL )
                sleep( &mailbox );

        msg = mailbox;
        mailbox = NULL;
        wakeup( &mailbox );
        return msg;
}
```

10.5 对于一个多线程的 UNIX 内核实现，给出一种技术建议，使得为单处理机编写的设备驱动程序不必对其本身的源代码做任何改动就能在 MP 环境中使用。

10.6 考虑 10.4 节中介绍的粗粒度内核实现，其中为每个子系统分配了一个锁。假设可能必须同时占有任何两个锁。编写一个 C 函数，在进程已经占有任何别的锁时，还能以一种不会造成死锁的方式获得第二个锁。这个函数有两个参数：已经占有的锁的地址和要获得的锁的地址。该函数必须以占有两个锁来返回。如果有必要，可以允许该函数释放已经占有的锁，然后再获得它，以避免死锁。

10.7 在使用上一题中的函数时，必须有一个特殊的函数来释放锁，以便按照获得锁的顺序来释放它们吗？

10.8 考虑在多线程内核上实现的系统调用 alarm。警告的超时（time-out）值保存在进程表项中，于是进程本身和时钟中断处理程序都可以更新它。时钟中断处理程序每秒将超时值减 1，当它的值为 0 时向进程发送一个信号。在多线程内核中，时钟中断处理程序可能正在一个处理器上运行，并且更新超时值，而在不同处理器上的一个进程正在执行系统调用 alarm 来改变当前值。需要一个锁来保护超时值吗？或者这仅仅是一种固有的竞争条件？

10.9 下面这段代码在一个链表中找到一个元素，然后以原子方式把链表元素中的值同传入的新值进行交换。这个实现在 MP 方面可以有什么改进？假定可以从任何处理器调用函数 find_and_replace，但是只能从 find_and_replace 调用 swap。假定所有的自旋锁都已经正确地进行了初始化。

```
struct element {
        lock_t          e_lock;
```

```
        int             e_tag;
        int             e_value;
        struct element  *e_next;
};

struct element *list;
lock_t list_lock;

int
find_and_replace( int tag, int new_value )
{

        struct element *ip;
        int old_value;

        lock( &list_lock );

        for (ep = list; ep; ep = ep->e_next)
                if( ep->e_tag == tag ) {
                        old_value = swap(ep, new_value);
                        unlock( &list_lock );
                        return old_value;
                }

        unlock( &list_lock );
        return -1;
}

int
swap( struct element *ep, int new_value )
{
        int old_value;

        lock( ep->e_lock );
        old_value = ep->e_value;
        ep->e_value = new_value;
        unlock ( ep->e_lock );
        return old_value;
}
```

10.9 进一步的读物

[1] Anderson, T.E., "The Performance of Spin Lock Alternatives of Shared-Memory Multiprocessors," IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 1, January 1990, pp. 6-16.

[2] Beck, B., Kasten, B., and Thakkar, S., "VLSI Assist for a Multiprocessor," *Second International Conference on Architectural Support for Programming Langusges and Operating Systems*, October 1987, pp. 10-20.

[3] Beck, R.D., and Kasten, R.A., "Multiprocessing with UNIX," *Systems and Software*, October 1984.

[4] Black, D.L., Tevanian Jr., A., Golub, D.B., and Young, M.W., "Locking and Reference Counting in the Mach Kernel," *Proceedings of the 1992 International Conference on Parallel Processing*, August 1992, pp. 167-73.

[5] Boykin, J., and Langerman, A., "The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis," *Proceedings of the USENIX Workshop on Expriences with Distributed and Multiprocessor Systems*, October 1989, pp. 105-26.

[6] Campbell, M., Barton, R., Browning, J., Cervenka, D., Curry, B., Davis, T., Edmonds, ., Holt, R., Slice, J., Smith T., and Wescott, R., "The Parallelization of UNIX System V Release 4,0," *USEUIX Conference Proceedings*, January 1991, pp. 307-23.

[7] Campbell, M., and Holt, R.L., "Lock Granularity Analysis Tools in SVR4/MP," *IEEE Software*, March 1993, pp. 66-70.

[8] CaraDonna, J.P., Poaciorek, N., and Wills, C.E., "Measuring Lock Performance in Multiprocessor Operating System Kernels," *Proceedings of the USENIX Symposium on Experiences with Distrbuted and Multiprocessor Systems*, September 1993, pp 37-56.

[9] Clark, B.E.J., and Shirnia, A., "Hardware and Software Aspects of Tightly Coupled Symmetrical UNIX Multiprocessors," *Proceedings of the Autumn 1988 EUUG Conference*, pp. 345-55.

[10] Conde, D. S., and Hamilton, G., "An Experimental Symmetric Multiprocessor Ultrix Kernel," *USENIX Conference Proceedings*, February 1988, pp. 283-90.

[11] urran, S., and Stumm, M., "Scheduling a UNIX Workload on Small-Scale, Shared Memory Multip-rocessors," *Computer Systems*, Vol. 3, No. 4, October 1990, pp. 551-79.

[12] Keating, C., and White, G., "The Transformation of UNIX to Support Multiple 80386 Processors," *UniForum Conference Proceedings*, February 1989, pp. 121-9.

[13] Langerman, A., Boykin, J., and LoVerso, S., "A Highly Parallelized Mach-based Vnode Filesystem," *USENIX Conferences Proceedings*, Winter 1990.

[14] Paciorek, N., LoVerso, S., and Langerman, A., "Debugging Multiprocessor Operating

System Kernels," *Proceeding of the Second Symposium on Experiences with Distributed and Multiprocessor Systems*, March 1991, pp. 185-201.

[15] Peacock, J.K., "File System Multithreading in System V Release 4 MP," *USENIX Conference Proceedings*, Summer 1992.

[16] Peacock, J.K., Saxena, S., Thomas, D., Yang, F., and Yu, W., "Experiences from Multithreading System V Release 4," *Proceedings of the Third Symposium on Experiences with Distributed and Multiprocessor Systems*, March 1992, pp. 77-91.

[17] Pike, R., Preesotto, D., Thompson, K., and Holzmann, G., "Process Sleep and Wakeup on a Shared-memory Multiprocessor," *EurOpen Conference Proceedings*, Spring 1991, pp. 161-6.

[18] Sinkewicz, U., "A Strategy for SMP ULTRIX," *USENIX Conference Proceedings*, June 1988, pp.203-12.

[19] Stum, M., Unrau, R., and Krieger, O., "Desing a Scalable Operating System for Sharedf Memory Multiprocessors," *Proceedings of the USENIX Worksphop on Micro-Kernels and Other Kernel Architectures*, April 1992, pp. 285-303.

[20] Test, J.A., "Concentrix - A UNIX for the Alliant Multiprocessor," *USENIX Conferenc Proceedings*, January 1986.

采用信号量的内核

第 11 章

本章介绍一种新的 MP 原语，即信号量（semaphore），它代替了单处理机 sleep/wakeup 设施的功能。信号量既能用于互斥，也能用于同步。因为发明它们是为了在不需要外部上锁机制和不做非抢先假定的条件下正确发挥作用，所以它们不同于 sleep/wakeup，使用起来往往要更为简单。正如很快会看到的那样，信号量不是专门为了代替自旋锁来保护短临界段的，所以在采用信号量的内核中仍然会使用自旋锁。这里介绍操作系统中信号量的实现及其使用，下一章将会展示其他各种 MP 原语。

11.1 引 言

上一章讲述了如何向单处理机内核加入自旋锁，使之能够在 MP 系统上正确运行。围绕 sleep 和 wakeup 函数的使用，需要有自旋锁，因为不这样的话，在使用这两个函数的代码中就会出现竞争条件。由于这些竞争的缘故，所以 sleep 和 wakeup 函数本身并不适合于 MP 系统。此外，wakeup 函数的行为是唤醒在某个特殊事件上睡眠的全部进程，对于 MP 系统来说，经常不希望如此。

正是因为有了这些考虑，操作系统的设计人员转向在多处理机 UNIX 系统中用其他原语代替 sleep 和 wakeup。和自旋锁一样，这些原语的设计能在 MP 环境中正确发挥作用，而不管系统中处理器的数量有多少。它们与自旋锁的区别在于它们允许让一个进程在某些判定规则不能立即得以满足（比如，试图获得一个已经投入使用的长期锁）的时候被挂起。虽然发明了各种各样这类的原语供 MP 系统使用（其中的一些将在下一章里介绍），但是最简单的原语之一则是 Dijkstra 信号量。它既可以用于实现互斥，也可以用于实现进程同步。

AT&T Bell 实验室在 20 世纪 80 年代早期完成的 UNIX 操作系统的 MP 实现就使用了信号量，它是在一种称为 3B20 的双处理器 SMP 系统上实现的。这一实现的源代码可以通过许可证获得，于是成为了包括 Silicon Graphics、Sequent 和 Pyramid Technologies 在内的许多公司开发各自 MP 系统的最初基础。从这个基础演变出来的内核已经用到了从微处理器到大型机的各种 MP 系统上，有一些现如今仍在使用。

信号量的状态是用一个有符号的整数值来表示的，根据这个整数值定义了两种原子操作。操作 $P(s)$将信号量 s 所关联的整数值减 1，如果新得到的值小于 0，则阻塞进程。如果新得到的值大于或者等于 0，则允许进程继续执行。逆操作为 $V(s)$，它将信号量的值加 1，如果新得到的值小于或者等于 0，则取消该信号量上 P 操作期间被挂起的进程的阻塞状态。执行 V 操

作的进程决不会被阻塞。除了整数值之外，在信号量上被阻塞的进程队列也被当成是信号量的状态来进行维护。当信号量的值为负数的时候，其绝对值指出了队列中进程的数量。

P 和 *V* 操作同单处理机 sleep 和 wakeup 函数之间的区别因素在于，信号量是更为高级的操作，它让阻塞的决定和阻塞进程的动作都是一项原子操作。这类似于 lock_object 函数，后者测试对象的状态，并且不是阻塞对象就是睡眠，所有的操作都是原子操作。此外，像 wakeup_one 一样，*V* 操作具有一次仅取消一个进程阻塞状态的理想效果。因为信号量掌握着和它们相关联的状态（由每次操作更新的整数值），因此没有必要像 lock_object 函数那样保持外部的标志来记录锁的状态。而且这种状态信息意味着信号量保持着对过去操作的"记忆"，这和 wakeup 不一样（如果没有进程在事件上睡眠，那么它不会记得有操作）。

11.1.1　采用信号量的互斥

将信号量的值初始化为 1，就可以实现长期互斥锁。在进入临界段之前先使用 P 操作，在退出时使用 *V* 操作就能保护它（参见图 11-1）。这样做一次仅允许临界区内有一个进程。一旦第一个进程进入临界区，它就会把信号量的值减为 0。该进程获得许可继续执行，但是随后的任何其他进程在执行 *P* 操作的时候将阻塞。当第一个进程退出临界区的时候，如果有一个等待进程的话，那么 *V* 操作将通过取消该进程的阻塞状态来准许另一个进程进入临界区。

```
P(s)
执行临界段
（进程可以在任何时刻自由切换现场）
V(s)
```

图 11-1　信号量保护的临界区

以这种方式用于互斥的信号量称为二值信号量（binary semaphore）。这种命名指出了这样的事实，即二值信号量在逻辑上只有两种状态：上锁和解锁。二值信号量的值从不会大于 1，因为这会一次让一个以上的进程处于临界段。

和长期互斥的所有用法一样，进程也允许在临界区内阻塞（例如，当等候 I/O 完成的时候）。和使用 lock_object 函数实现长期互斥的单处理机内核一样，这样做不会释放锁。只有在进程执行 *V* 操作的时候才会释放它。这同上一章里讨论过的粗粒度内核中自旋锁的使用不一样，那里的自旋锁（但不是长期锁本身）是围绕现场切换自动释放和获得的（参见 10.4 节）。

11.1.2　采用信号量的同步

将信号量的值初始化为 0，就可以用于进程同步，这样做允许通过使用 *P* 操作让一个进程等待某个事件发生。既然信号量被初始化成了 0，那么进程将会立即阻塞。另一个进程使用 *V* 操作能够发出信号，表明事件已经结束。*V* 操作导致正等待事件的进程被唤醒，并继续执行（参见图 11-2）。因为即使在信号量上没有阻塞进程，*V* 操作也会给信号量加 1，所以在前一个进程能够执行 *P* 操作之前触发事件会导致进程继续执行，不必等待。这是一种受欢迎的情形，因为它不要求有额外的协调工作，就能处理在等候事件的进程同发信号表明该事

件完成的进程之间本来就有的竞争条件。

进程 1	进程 2
P(s) /* 等待事件 */	.
	.
	.
	V(s) /* 触发事件 */

图 11-2 采用信号量的同步

11.1.3 采用信号量分配资源

最后，信号量可以用于控制一个有限资源库内资源的分配。例如，假定系统中有一个特殊用途的缓冲区库，进程必须定期地分配它。如果当前没有可用的缓冲区，那么要求获得一块缓冲区的进程会保持等待，直到有一块缓冲区可用为止。通过把信号量初始化为库中缓冲区的数量，就能执行 *P* 操作来预订一块缓冲区。只要信号量的值保持大于 0，那么每个预留一块缓冲区的进程都会得到允许，不必阻塞就能运行。当释放一块缓冲区的时候，执行一次 *V* 操作。一旦所有的缓冲区都投入使用了，那么要预留一块缓冲区的进程便被阻塞。当释放一块缓冲区的时候，等候一块缓冲区的进程被唤醒，得以继续执行（参见图 11-3）。

注意，*P* 操作并不分配缓冲区，而只是预留。分配和回收必须分别实现。如果进程试图一次从库中预留一块以上的资源，就会造成死锁。这将在下一节里讨论。

```
P(s) /* 预留缓冲区 */
分配和使用缓冲区
...
把缓冲区返回给库
V(s) /* 释放预留 */
```

图 11-3 采用信号量预留资源

11.2 死 锁

在 9.3 节里介绍的一些潜在的死锁情形也可能会在采用信号量时出现。简而言之，和自旋锁一样，信号量也会有上锁次序和 AB-BA 死锁的问题。类似地，如果一个进程试图不止一次获得一个用于互斥的信号量的话，就会向它自己发生死锁。一旦它试图第二次锁住信号量（称为递归上锁），它就会永远阻塞下去。这两个死锁问题和使用自旋锁时是一样的。此外，这些情形对于实现互斥的任何 MP 原语都适用。

在进程占有一个二值信号量时也允许进行现场切换。这不会造成采用自旋锁时潜在的死锁问题，因为试图获得信号量锁的进程将会阻塞，而让别的进程运行。这样一来，等候锁的进程就不会像采用自旋锁时那样，不让占有锁的进程运行和释放锁。

虽然在一个中断处理程序之中获得自旋锁是安全的（假定被中断的代码没有占有同一个锁），但是信号量的 *P* 操作不能用在中断级（interrupt level）上，这和不能使用 sleep 函数是

一样的。中断处理程序没有进程现场，所以如果执行 *P* 操作，而信号量的值小于或者等于 0，就不会阻塞进程。阻塞被中断的进程是不对的，因为它可能就是执行 *V* 操作释放信号量的进程。这就好比这样的情形，中断处理程序永远自旋下去，试图获得由被中断的进程使用的自旋锁。

一个进程可能在用于同步的信号量上执行多次 *P* 操作。这并不会导致死锁，因为事件总是由另一个进程触发的。为了展示这一点，考虑两个进程之间的生产者-消费者（producer-consumer）关系。进程 1 等候一块数据缓冲区，在该缓冲区到达时就处理它。进程 2 准备数据，当它就绪后，使用一个信号量通知进程 1（参见图 11-4）。所以，即使进程 1 在同一个信号量上重复执行 *P* 操作，也不会发生死锁。因为信号量是用于同步目的的，所以这不被当成是递归上锁。

进程1

```
for (;;) {
    P(s); /* 等候数据 */
    处理来自进程2的数据
}
```

进程2

```
for (;;) {
    为进程1准备数据
    V(s); /* 数据就绪 */
}
```

图 11-4　生产者-消费者模型

在把信号量用于资源预留时，如果进程试图一次预留多份资源（通过使用多次 *P* 操作做到这一点），那么就会出现死锁。为了了解这是怎样发生的，设想有一个库，它包含四份资源，有两个进程，每个进程都想立即使用它们中的三份。如果这些进程是在不同的处理器上运行的，而且同时开始它们的连续 3 次 *P* 操作，那么一旦它们中的每一个都获得了两份资源，就会出现死锁。在两个进程中的任何一个满足其需求之前，资源库就被用光了，从而致使两个进程以及任何其他试图从资源库中分配资源的进程都永远地阻塞下去。就和典型的死锁情形一样，两个进程的相对时序决定了实际是否会发生死锁。如果两个进程中的任何一个能够在另一个开始之前完成它的 *P* 操作序列，预留它所需要的资源，那么就不会发生死锁。

有几种算法能够以不会出现死锁的方式分配多项资源，它们都通过让一个进程不会部分地获得它所需要的资源来避免死锁。如果不能获得全部的资源分配，那么就不给进程任何资源，并且阻塞进程，直至它能够获得所需要的全部预留资源为止。这样一来，进程在等候的时候就不会消耗别的进程需要的资源。银行家算法（banker's algorithm）就利用了这种技术。进程并不是执行多次 *P* 操作来获得资源，而是使用一种新的原语，它们在其中指定所需资源的数量，然后就可以按全有全无（all-or-nothing）的方式来分配资源（参见 11.9 节中的参考文献[11]、[14]、[23]和[31]，了解更多的信息）。

最后要注意，就资源分配请求本身而言，只要进程从不企图一次利用资源库中的多份资源，那么就决不会发生死锁。如果资源库为空，那么进程被阻塞，直到无需占有任何一份资源就能获得一份的时候为止，这就防止了出现死锁。

11.3　实现信号量

RISC 处理器的问世意味着，类似信号量那样需要多次自动完成存储器访问的操作很少在

硬件中实现。相反，信号量必须由软件建立在基本硬件原子操作之上。使用自旋锁可以轻易地做到这一点。

每个信号量都需要一个小小的数据结构来维护当前的值和被阻塞进程的队列（参见图 11-5）。队列使用单个链接的列表，还有指向列表首尾元素的指针。在一个信号量上阻塞的进程被加入到列表的末尾，由 *V* 操作解锁的进程则从列表开头删除（另一种方法是，进程可能按照其优先级被唤醒）。在更新数据结构的时候，加入一个自旋锁来提供互斥。

```
struct semaphore {
        lock_t    lock;   /* 保护其他字段 */
        int       count;  /* 当前信号量的值 */
        proc_t    *head;  /* 指向第一个被阻塞的进程的指针 */
        proc_t    *tail;  /* 指向最后一个被阻塞的进程的指针 */
};
typedef struct semaphore sema_t;
```

图 11-5 信号量的数据结构

在使用信号量的数据结构之前必须对其进行初始化，这通常是在系统初始化时，或者在分配一个包含信号量的数据结构时完成的。采用图 11-6 中的函数，可以把信号量初始化为任意值。

```
void
initsema( sema_t *sp, int initial_cnt )
{
        initlock( &sp->lock );
        sp->head = NULL;
        sp->tail = NULL;
        sp->count = initial_cnt;
}
```

图 11-6 初始化一个信号量

信号量的 *P* 操作可以如图 11-7 所示来实现。进程的 u 区包含一个指向其进程表项的指针。这个指针为 u.u_procp，和运行队列的实现一样（参见 9.4.1 小节），proc 结构的元素 p_next 用于把同一信号量上阻塞的进程链接到一起。进程绝不会同时既在运行队列中，又在信号量上被阻塞，所以同一个指针可以用于上述两者。类似地，一个进程一次不能在一个以上的信号量上阻塞。

函数 swtch( )执行一次现场切换。它使得当前执行进程的状态被保存起来，并选择执行一个新的进程（这和 sleep 内部使用的函数是同一个函数）。在以后进程被唤醒的时候（*V* 操作使该进程可以运行之后），它会在函数 swtch()内保存它的状态的地方继续执行。接着，它只是返回到 p( )函数。此刻，进程已经影响了信号量。既然在进程第一次进入 p( )函数的时候就调整了信号量的计数值，所以不需要再进一步操作，它只是返回调用方。

信号量的 *V* 操作可以如图 11-8 所示来实现。如果一个进程已经在前一次 *P* 操作期间被阻塞，那么 *V* 操作就从队列中移走这类进程中最早的一个，并把它加入到运行队列中。如此

一来，这种实现通过以 FIFO 次序唤醒进程，就照顾到了全体的公正性。还可以让它唤醒优先级最高的进程，但副作用是，如果在执行 *P* 操作的时候，优先级较高的进程连续阻塞，那么优先级低的进程可能会被阻塞任意长的一段时间（注意，这正是实时系统所期望的行为）。

```
void
p( sema_t *sp )
{
        lock( &sp->lock );
        sp->count--;

        if ( sp->count <0) {
                if ( sp->head == NULL )
                        sp->head = u.u_procp;
                else
                        sp->tail->p_next = u.u_procp;

                u.u_procp->p_next = NULL;
                sp->tail = u.u_procp;
                unlock( &sp->lock );
                swtch( );
                return;
        }
        unlock( &sp->lock );
}
```

图 11-7　信号量的 *P* 操作

```
void
v( sema_t *sp )
{
        proc_t *p;

        lock( &sp->lock );
        sp->count++;

        if ( sp->count <= 0 ) {
                p = sp->head;
                sp->head = ->p_next;

                if (sp->head == NULL)
                        sp->tail = NULL;
                unlock( &sp->lock );
                enqueue( &runqueue, p );
                return;
        }
        unlock( &sp->lock );
}
```

图 11-8　信号量的 *V* 操作

在阻塞进程的 *P* 操作期间释放信号量内部的自旋锁时，和进程在 swtch 函数内完成现场切换时，这两点时间之间有一个原本就有的竞争条件。另一个处理器有可能正好在锁上自旋，等待执行一次 *V* 操作。只要 *P* 操作释放了自旋锁，那么就可以进行 *V* 操作。如果在信号量队列中的唯一进程是仍在 swtch 内执行的那个进程，那么 *V* 操作就会删除它，并把它加入到运行队列中去。于是，另一个处理器就有可能选择这个进程来执行，并且试图在该进程能在前一个处理器上 *P* 操作内完成它的现场切换之前就要运行它。如果发生了这种情况，那么会出现异常，因为在进程的状态尚未在别的处理器上被完整保存之前，这个处理器就要恢复该进程的状态。如果正好在调用 swtch 之前发生了一次长时间的中断，那么就更有可能出现这种竞争条件。图 11-9 给出了一条时间线索，显示了致使这种竞争条件出现的事件序列。

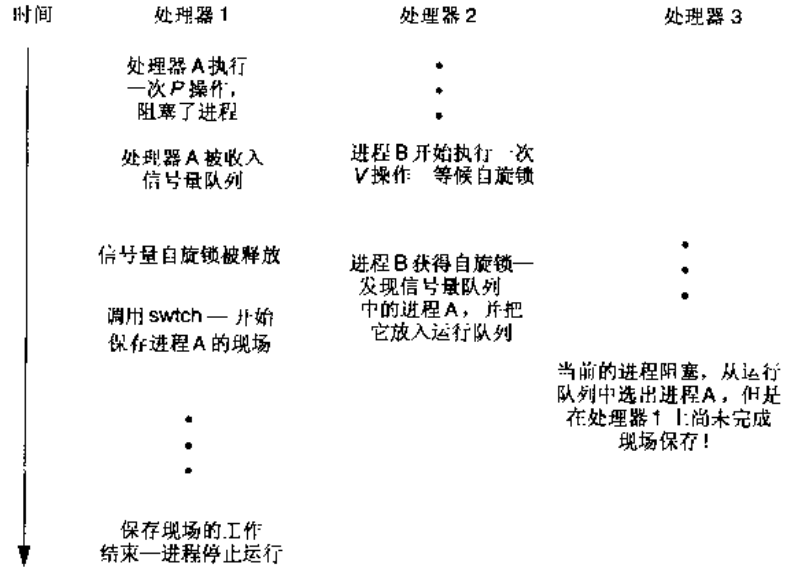


图 11-9 信号量实现的竞争条件

为了防止这种竞争条件，在 swtch 例程已经完全保存了进程的状态之后，它会把进程的状态从 running（正在运行）变为 blocked（被阻塞）。无论处理器何时选择一个要执行的进程，在试图运行它之前，都要检查进程的状态。如果进程的状态被标记为 running，那么则意味着有其他的进程仍然在保存进程的状态。在继续执行之前，新的处理器只是在这个状态上处于忙等待，直到状态变为 blocked 为止。一般情况下，因为这种竞争条件很少见，所以进程已经被标记为 blocked 了。但是，为了保持系统的完整性，内核必须做好处理这种情况的准备。

11.4 粗粒度信号量的实现

有可能使用信号量而不是自旋锁来实现上一章介绍的粗粒度内核（参见 10.2 和 10.4 节）。对于这项应用来说，二值信号量比自旋锁更为方便，因为它们在锁已经被占有时会阻塞进程。在采用自旋锁的情况下，当可以执行用户态进程时，需要一次有条件的锁操作来避免自旋。

在使用信号量的时候不需要这样做。

实现的其他方面都不变：在内核中睡眠之前，仍然必须释放巨型锁，仍然需要独立的运行队列，以便无需获得巨型锁之一就能切换现场。系统的最终性能是一样的，因为信号量只是为实现巨型锁提供了一种比自旋锁更方便的原语，它们不会影响在内核中可能有的并行度。虽然信号量实现的巨型锁更好，但是在诸如保护运行队列这样的情况下，自旋锁仍然不失为一种在粗粒度内核中很有用的原语。

11.5　采用信号量的多线程

当采用信号量实现多线程内核时，同样可以运用上一章介绍的、采用自旋锁的多线程机制的原则。例如，要尽可能避免不必要的上锁（参见 10.3 节），这仍然是我们所非常期望的。类似地，仍然需要考虑锁的粒度（参见 10.5.4 小节）。当采用信号量实现多线程内核时看到的主要不同在于，信号量原语替换了 sleep 和 wakeup。在上一章里，使用了自旋锁来增强原来的机制，以便它们在 MP 系统上能起作用而不会有竞争条件。下面的几小节更为详细地对此予以讨论。

11.5.1　长期互斥

采用信号量很容易就能实现长期互斥。因为信号量的值保留了锁的状态（值为 1 意味着没有上锁，小于或者等于 0 意味着上了锁），不需要有外部标志来传给单处理机 lock_object 函数以表明锁的状态。而且因为信号量已经根据需要执行了阻塞进程和取消阻塞的任务，所以也就不需要 sleep 和 wakeup 了。结果是可以取消 lock_object 和 unlock_object，并以信号量的操作来代替它们。不是在要被锁住的数据结构中保持锁状态标志，而是代之以用一个信号量和它关联起来。于是，在用到 lock_object 的任何地方，都能代之以使用信号量的 *P* 操作，传递给它指向要被锁住的数据结构中信号量的指针。类似地，所有的 unlock_object 调用都可以用信号量的 *V* 操作来替换。

除了不需要单独的状态标志之外，使用信号量和使用传统的单处理机技术实现长期互斥之间还有以下不同应该予以说明。

首先，在使用信号量的时候，不需要 lock_object 函数内的 while 循环。在有多个进程为锁而进入睡眠的情形中，这个循环是必不可少的。因为 wakeup 会唤醒为相同事件而进入睡眠的所有进程，所以单处理机内核不得不确保只有一个这样的进程能获得锁，而其他的进程再度入睡。因而进程会被唤醒，发现锁仍然被锁住了，并且在最终为它们自己获得锁之前还得再睡上任意长的一段时间，于是就需要 while 循环。

甚至在使用上一章介绍的 wakeup_one 函数时，循环仍然是必要的，因为一个首次调用 lock_object 的新进程会同刚刚被唤醒的进程竞争锁。新进程能够比一直为等候锁而睡眠的进程早获得锁，从而致使这些进程要执行循环，再度睡眠。

像本章中那样采用信号量来实现就不会发生这些情况了。如果已经上了锁，那么新接触信号量的进程一定要睡眠。它们是按照 FIFO 的次序被唤醒的，从而使得连续出现的新进程

不能妨碍等待中的进程获得锁。而且一次只唤醒一个进程，这样做对它们来说就是，它们每次醒来的时候不需要为锁而展开竞争。*V* 操作唤醒的进程保证能够获得锁。

11.5.2 短期互斥

虽然可以使用信号量实现短期互斥（在中断处理程序中的除外），但是自旋锁却是更好的选择。正如自旋锁在它们保护的临界段太长时所暴露出的不足一样，信号量在临界段太短时也会暴露出不足。这是因为，如果进程在信号量上阻塞的话，就会有现场切换的开销。如果开销比进程花在自旋锁上的自旋时间还要长，那么临界段对于使用信号量来说就太短了。因此，来自上一章的自旋锁示例，如显示运行队列的操控、信号处理等等，不应该变成使用信号量来实现。对于 MP 内核来说，使用一种以上的原语，每种适用于不同的情形，是很常见的做法。

11.5.3 同步

正如本章开头处所展示的那样，*P* 和 *V* 操作就好像单处理机的 sleep 和 wakeup 函数一样，可以用于进程同步。为了体现这一点，考虑当父进程正在使用系统调用 wait，阻塞父进程，直到子进程退出时，在父进程和子进程之间完成的同步。如果子进程在父进程执行 wait 调用之前退出，那么 wait 立即返回。在两种情况中的任何一种里，wait 都返回子进程的退出状态（它传递给系统调用 exit 的值，或者导致它终止的信号量）。

实现 exit 和 wait 之间同步的一种方法是向进程表项中加入信号量（它将称为 p_waitsema）。在创建进程的时候，将信号量初始化为 0。当进程执行一次系统调用 wait 的时候，它在信号量上执行一次 *P* 操作。图 11-10 给出了这样做的伪代码实现。

```
wait ()
{
        if (no children)
            return ESRCH;
        P( &u.u_procp->p_waitsema );
        find a child that has exited
        return child's exit status
}
```

图 11-10 wait 的算法

当进程退出的时候，它按照图 11-11 所示在其父进程的信号量上执行一次 *V* 操作。

注意，wait 的算法必须首先进行检查，看看是否有子进程。如果没有，那么就出错，并且返回 ESRCH。对于 wait 来说，在做检查之前执行 *P* 操作是不正确的，因为如果没有子进程，那么进程就会和它自己发生死锁。没有子进程，就没有进程在信号量上执行 *V* 操作来让第一个进程再次运行。

还要说明的一点是，对于任意数量的子进程，无论系统调用 wait 和 exit 的相对次序如何，

这个算法都能正确地起作用。例如，如果一个进程有 5 个子进程，其中有 3 个子进程在父进程等待它们中的任何一个之前就退出了，那么此刻信号量的值是 3。这意味着父进程下面要执行的 3 次 wait 系统调用都会返回而不会阻塞。如果剩下来的两个子进程都没有在父进程执行另一次 wait 系统调用之前退出，那么下一次 wait 调用就会阻塞。

```
exit()
{
        save exit status
        release address space
        close files
        etc.
        .
        .
        .
        v( &u.u_procp->p_parent->p_waitsema );
}
```

图 11-11　exit 的算法

这个例子给出了一种情况，其中有一个“消费者”进程（父进程）等候一个事件发生（子进程退出）。还有多个进程可能等待一个事件发生，而当事件发生时应该唤醒所有进程的情况（类似于 wakeup 的语义）。例如，如果多个进程已经在一个共享页面上产生了缺页错，那么当读取该页的 I/O 结束的时候，所有的进程都要被阻塞。这可以轻而易举地通过使用 *V* 操作的一种变形（称为广播 *V* 操作）来实现，这项操作能够解除信号量队列中所有进程的阻塞状态。代码和图 11-8 中所示的一样，不同之处在于它要保持循环，直到信号量的值为 0。因为如果在信号量上没有阻塞进程，那么它什么也不做，所以它和普通的 *V* 操作不同。这正是广播类型（broadcast-type）的操作所期望的行为，因为在事件发生之后才到的进程一般不需要阻塞。为了在前面例子的上下文中显示这一点，如果页面已经在存储器中了，那么在页面上发生缺页错的新进程就不需要等着 I/O 完成，所以它不会第一个执行 *P* 操作。

和在长期互斥的情况一样，信号量能够替换单处理机内核中所有用于同步目的的 sleep 和 wakeup。这些地方太多了，无法在这里列出来，但是它们都遵循本节介绍的指导原则。

11.6　性 能 考 虑

11.6.1　测量锁争用

为了从细粒度、多线程 MP 内核上获得最大的好处，必须确定并修改对锁争用得厉害的区域。过多地争用锁会减少内核并行活动的数量，并且降低系统的整体性能。为了找到性能瓶颈的所在，要在系统正在运行期望的混合应用任务时对它进行测量。测量结果可以用于确定发生锁争用的区域，并且能够修改内核以消除或者减少它们的出现。

因为在进入临界区的时候会执行 *P* 操作，所以对于二值信号量来说，能够轻而易举地确定争用。可以对 *P* 操作的实现进行修改，记录下各种统计数据，这些统计数据能够方便地保存在信号量自身的数据结构中。对于每个二值信号量来说，至少有以下重要的统计信息：

- ✧ 执行过的 *P* 操作的全部数量
- ✧ 在进程不得不阻塞处的 *P* 操作的数量
- ✧ 在该信号量上阻塞的进程队列的最大长度和平均长度

对一个信号量的争用程度可以由导致进程阻塞的 *P* 操作（因为锁已经投入使用了）的数量同操作的总数之比体现出来。比率越高，争用得越厉害，于是可以由队列长度的统计看出受争用影响的进程数量。最糟糕的争用位置是不得不阻塞的操作的百分比和队列长度都大的那些地方。

也可以为自旋锁收集统计信息。因为进程是进入自旋而不是在已经投入使用的锁上阻塞，所以记录的是获得一个锁需要自旋的平均量和最大量，而不是阻塞操作的数量和队列长度。这里产生的数据类似于上一段中的数据。

11.6.2 结对

当有一个持续的进程队列在等候一个二值信号量的时候，就发生了进程的结对(convoy)。在进程不得不被阻塞的地方的 *P* 操作数量几乎等于 *P* 操作的全部数量时，就会发生这一现象。它意味着对该信号量争用得太厉害，以及（或者）锁被占用的时间太久了。在粗粒度实现中，当混合的应用任务频繁执行系统调用的时候，结对是典型的现象。

例如，考虑由一个巨型锁保护文件系统的情形。执行一次文件操作的任何进程都会占有这个锁，甚至在它为等候磁盘 I/O 完成而被阻塞时也是如此。在这段时间内，许多其他的进程有可能试图开始它们自己的文件操作。这些进程都会被阻塞，因为锁已经在用了，从而标志着开始了结对现象。如果是文件 I/O 密集型的应用，那么结对有可能会无限期地持续下去。许多进程试图立即进入一段很长的临界区，而实现却一次只允许一个进程处于该临界区内，于是就造成了瓶颈。图 11-12 描绘了这种情形，图中的进程 A 正在一个由二值信号量所保护的临界段内执行，而同时其他进程则被阻塞，等待进入临界段。

乍一看，把临界段划分成两个或者以上的临界段，每个临界段由一个独立的锁来保护的做法似乎会有用处。在对文件系统采用巨型上锁机制的例子中，让最初的步骤（把用户的文件描述符转换为内部的文件表示、计算文件偏移量，以及执行各种出错检查）在一个锁之下全部完成，而该锁与随后的活动（检查缓冲区获得必要的数据，并且如果需要，执行磁盘 I/O）分离，这种做法是可能的。这会变为图 11-13 所示的情形。

这里的进程 A 和进程 E 都处在临界段内，而进程 B、C、D、F 和 G 正在等待进入临界段。

这种安排不能保证消除结对现象。考虑第一个临界段比第二个短的情况（在本例中是这样的，因为在第二个临界段内完成的磁盘 I/O 可能是最长的操作了）。这意味着进程会快速通过第一个临界段，而在等候进入第二个临界段时形成了结对。如果在两个信号量上都形成了结对，则性能比最初的情况还要糟糕，因为每个进程现在为了完成一次操作必须至少做两次现场切换。

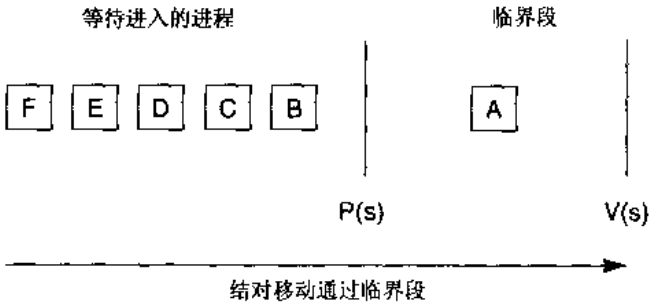


图 11-12　在临界段后形成的结对（convoy）现象

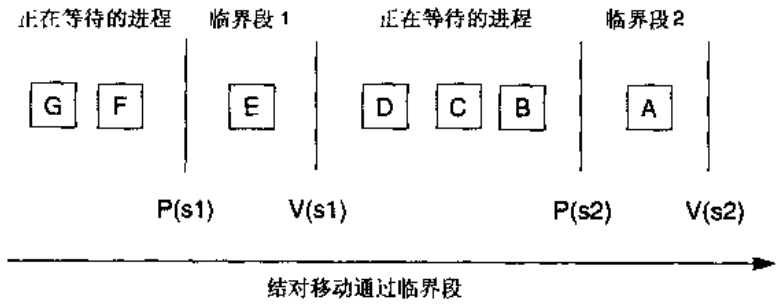


图 11-13　将较长的临界段一分为二

消除结对的途径是将代码并行化，以便多个进程能够在单个临界区内执行。不是像图 11-13 所示的那样，在临界区内一次只能通过一个进程，而是必须允许结对等候的所有进程都并行执行。为了做到这一点，必须对算法和数据结构进行组织，以便这些进程能够完成它们的工作而又不会对同一个锁争用得太厉害。在文件系统的例子中，通过为每个文件使用独立的锁就能做到这一点。这样做允许对不同文件执行 I/O 操作的进程在并行执行的同时，仍然保持单处理机内核的语义（保证原子文件操作）。这可以在图 11-14 中显示出来，图中的进程 A、B 和 C 正在对不同的文件执行 I/O 操作，s1、s2 和 s3 是用于每个文件的独立的信号量。

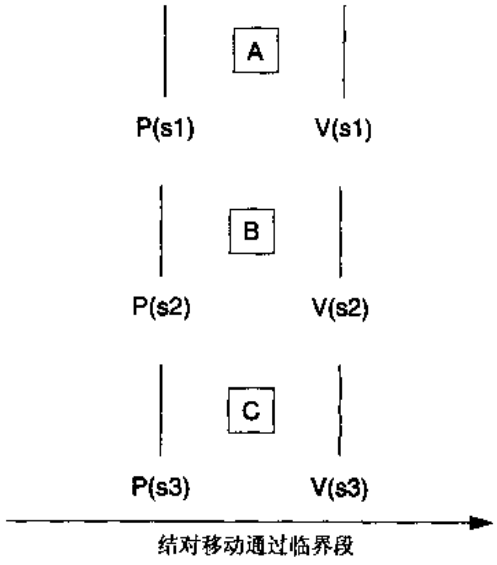


图 11-14　并行化的临界段

只要各个进程使用不同的文件，那么这样的实现就会消除结对现象。如果许多进程都需要对相同的文件立即执行 I/O 操作，那么还会再度出现结对现象。在这个例子中，认识到允许多个进程同时读取相同的文件并不会违反单处理机用于文件操作的系统调用的语义，就能采取进一步的步骤将临界段并行化。这类上锁机制可以采用一个多读锁（multireader lock）来做到。

11.6.3 多读锁

在有些情况下，通过允许多个只需要读取不同数据结构的进程并发执行读操作，就能让临界段并行化。只要没有进程修改任何数据结构，那么就不会出现竞争。当某个进程确实需要进行修改的时候，它可以通过等候读进程（reader process）结束，然后以同所有别的读方和写方互斥的方式进行修改来做到这一点。这样的上锁机制用多读-单写锁（multireader, single-writer lock）这一术语来命名（或者为了简单起见，就叫做多读锁。例如，这对文件操作来说就很有用，因为读文件要比写文件更频繁，对于常用于搜索路径的目录来说尤其如此。哪怕是在有多个写方，而仅有很少（如果有的话）读方的情况下，比如一组进程连续写一个日志文件，和二值信号量相比，这类上锁机制几乎没有多增加开销，所以即使混合的应用任务没有利用它，也不会对性能有不利的影响。按照下面所述采用信号量可以轻而易举地实现多读-单写锁。

图 11-15 给出的数据结构将用于记录多读锁的状态。使用一个自旋锁来保护多读数据结构（multireader data structure）内的计数器字段（counter field）。数据结构既体现了当前在临界段内进程的数量，又体现了等候进入临界段的进程的数量。这些计数值可以在读方和写方之间进行划分：

```
struct multi_reader {
    lock_t    m_lock;  /* 保护下面的 cnt 字段          */
    int       m_rdcnt; /* 在临界段内 rdrs 的数量       */
    int       m_wrcnt; /* 在临界段内 wrtrs 的数量      */
    int       m_rdwcnt;    /* 正处于等待中的读方的数量  */
    int       m_wrwcnt;    /* 正处于等待中的写方的数量  */
    sema_t    m_rdwait;    /* 在读方等候处的同步信号量  */
    sema_t    m_wrwait;    /* 在写方等候处的同步信号量  */
};

typedef struct multi_reader rwrlock_t;
```

图 11-15 多读锁的结构

在使用锁之前，必须通过调用图 11-16 中的例程来对它进行初始化。

由多读锁保护的临界资源的访问控制策略是，只要没有写方正在等待或者当前在临界段内，就允许读方在任何时候进入临界段。一旦写方来了，那么就阻塞后续的读方。

这样做确保了连续到达锁的新的读方不会永远让写方到不了临界段。为了读取目的而希

望获得多读锁的进程使用图 11-17 中的例程。

```
void
init_rdwrlock( rdwrlock_t *mp )
{
        initlock( &mp->m_lock );
        initsema( &mp->m_rdwait,0 );
        initsema( &mp->m_wrwait,0 );
        mp->m_rdcnt =0;
        mp->m_wrcnt = 0;
        mp->m_rdwcnt    0;
        mp->m_wrwcnt - 0;
}
```

图 11-16　初始化一个多读锁

```
void
enter_reader( rdwrlock_t *mp )
{
      lock( &mp->m_lock );
      /*
      * 如果当前有一个写方占有了锁，或者有一个写方正在等候，那么我们必须等待。
      */
     if( mp->m_wrcnt || mp->m_wrwcnt ){
           mp->m_rdwcnt++;
           unlock( &mp->m_lock );
           p( &mp->m_rdwait );
           return;
      }
     mp->m_rdcnt++;
      unlock( &mp->m_lock );
}
```

图 11-17　为了读取目的而获得一个多读锁

一个正在离开由多读锁保护的临界段的读方则调用图 11-18 所示的例程。一旦所有的读方都离开了临界段，那么如果有写方在等候的话，就唤醒一个写方。

当一个写方希望获得锁的时候（参见图 11-19），它必须等到所有使用锁的进程都离开临界区为止。如果当前没有进程在使用临界区，那么写方能够立即获得锁。注意，根据多读-单写锁的定义，m_wrcnt 字段从来都不会大于 1。

释放一个由写方占有的多读锁是最为复杂的操作。为了确保公平性，当读方和写方都在等待锁的时候，首先会唤醒多个读方。这就防止了在锁上出现连续不断的写方，不让读方进入临界区。因为当有一个或者更多写方在等候的时候，后续到达的读方都被阻塞了，所以也

保证了写方不会被无限期地阻塞下去。这样一来，当两种类型的进程都在等候锁的时候，锁就在读方和写方之间轮流。既然读方能够并行地使用临界段，那么只要有一个写方离开了临界段，就会唤醒所有的读方。实现的代码如图 11-20 所示。

```
void
exit_reader( rdwrlock_t *mp )
{
        lock( &mp->m_lock );
        mp->m_rdcnt--;
        /*
        *如果我们是最后一个读方，而且有一个写方正在等待，那么现在就让这个写方执行。
        */
        *if( mp->m_wrwcnt && mp->m_rdcnt == 0){
                mp->m_wrcnt = 1;
                unlock( &mp->m_lock );
                v( &mp->m_wrwait );
                return;
        }
        unlock( &mp->m_lock );
}
```

图 11-18 读方释放一个多读锁

```
 void
 enter_writer( rdwrlock_t *mp )
 {
         lock( &mp->m_lock );
         /*
         *如果已经有进程在使用锁则阻塞。
         */
        if( mp->m_wrcnt || mp->m_rdcnt ){
                mp->m_wrwcnt++;
                unlock( &mp->m_lock );
                p( &mp->m_wrwait );
                return;
        }
        mp->m_wrcnt = 1;
        unlock( &mp->m_lock );
 }
```

图 11-19 为了写入目的而获得一个多读锁

```
Void
exit_writer( rdwrlock_t *mp )
{
        int rdrs;

        lock( &mp->m_lock );

        /*
        *如果有读方在等候，则让其先执行。
        */
       if( mp->m_rdwcnt ){
              mp->m_wrcnt = 0;

              /*
              *唤醒目前正在等待的所有读方。
              */

             rdrs = mp->m_rdwcnt;
             mp->m_rdcnt =rdrs;
             mp->m_rdwcnt = 0;
             unlock( &mp->m_lock );

             while(rdrs-- )
                    v(&mp->m_rdwait);

             return;
        }
        /*
        *没有正在等候的写方，如果有一个写方的话，就让一个写方执行。
        */
       if( mp->m_wrwcnt ){
           mp->m_wrwcnt--;
           unlock( &mp->m_lock );
           v( &mp->m_wrwait );
           return;
        }

        /*
        *没有等候的进程。释放锁。
        */
       mp->m_wrcnt = 0;
       unlock( &mp->m_lock );
}
```

图 11-20　写方释放多读锁

也可以用自旋锁实现多读锁（参见习题 9.3）这对于保护对于信号量来说太短的临界段很有用处。自旋锁、信号量和多读锁一起提供了一组解决多线程内核中资源争用的原语。

11.7 小 结

信号量提供一种有用的 MP 原语，该原语能够替换单处理机 sleep/wakeup 机制的所有用法。信号量要么实现互斥，要么实现进程同步，而且能在有任意个处理器的系统上正确地发挥作用，其中也包括单处理机的情形。比起 sleep/wakeup 机制，信号量的优势在于，它们可以更新信号量的状态，阻塞进程或者取消对进程的阻塞，这些都是原子操作。没有必要像使用 sleep/wakeup 时在 lock_object 函数中要求的那样维护一个单独的标志。其次，信号量一次只唤醒一个进程，从而消除了不必要的颠簸现象。因此，信号量适于实现长期互斥。在粗粒度内核中实现巨型锁时，它们也比自旋锁更好（因为巨型锁是长期互斥锁）。但是，不应该用信号量完全替代自旋锁。在实现短临界段内的互斥方面，仍然首选自旋锁。在这些情况下尝试使用信号量会引入不必要的现场切换开销，从而比自旋所花的时间要多。

和使用任何上锁机制一样，对锁的过度争用限制了内核并行活动的数量，因此限制了系统的整体性能。在极端的情况下，会在临界段的入口处形成进程长时间持续排队的现象（称为结对）。将一个较长的临界段分成两个或者两个以上部分的做法并不会明显改善这种情况，因为它可能带来现场切换过多的结果。必须代之以找到一种将临界段并行化的途径。通过将用于保护临界段的数据结构（比如独立的锁）进行分隔，就能做到这一点。另一种可能是使用多读-单写锁，它允许多个进程立即访问一个共享的数据结构，只要它们都不需要修改该数据结构即可（这在操作系统中是一种常见的操作）。只准许写方互斥访问，以便保持数据结构的完整性。这些技术可以用来提高并行内核活动的数量，从而提高系统的整体性能。

11.8 习 题

11.1 每个信号量都需要它自己的自旋锁吗？无关的信号量能共享同一个自旋锁吗？共享相同的锁能产生什么样的影响？

11.2 UNIX 内核最初能用信号量而不是 sleep/wakeup 设施编写吗？如果能，讨论一下如何进行权衡。

11.3 使用信号量实现两个进程间的双缓冲机制。进程 A 用数据填充缓冲区 1。当完成的时候，它唤醒了进程 B，进程 B 使用数据。与此并行，进程 A 填充缓冲区 2。当完成的时候，它等待进程 B 用完缓冲区 1，然后用新数据填充缓冲区 1，同时进程 B 使用缓冲区 2，如此类推。给出进程 A 和进程 B 的 C 代码片段。

11.4 在 11.6.3 小节给出的多读锁实现中，当一个新的读方恰好在一个已有的写方开始唤醒读方的时候到达，会发生什么样的情况？

11.5 假定多读锁已经为一个写方所使用了，并且有多个读方已经在等待这个锁。现在，一个新的读方调用 enter_reader，正好在写方调用 exit_writer 之前获得了自旋锁。再进一步假

定正好在新的读方释放自旋锁之后，但在它能在 m_rdwait 上执行 *P* 操作之前，出现了一次中断。这会让正在多读锁的自旋锁上自旋的现有写方获得锁，并继续执行。如果现有的写方此刻（在新的读方能够执行 *P* 操作之前）开始唤醒读方，那么会出现什么样的情况？

11.6　例程 exit_writer 能够在唤醒读方时，而不是在 while 循环中执行 *V* 操作时，利用广播 V 操作吗？

11.7　重写多读锁的实现，在既有读方又有写方在等待一个写方退出临界段的情况下增加公平性。此时不是唤醒所有的读方，而是按照到达的次序唤醒读方和写方。如果在一个写方之前到达了一组读方，那么应该唤醒整组的读方，但不唤醒在写方之后到达的读方。例如，如果进程 A 为了执行写操作而占据着锁，其他的进程按照下面的顺序到达：为执行读操作的进程 B 和 C，为执行写操作的进程 D，以及为执行读操作的进程 E，然后当 A 退出临界段的时候，只唤醒 B 和 C（此刻不唤醒 E）。到达的任何新的读方都被阻塞。一旦 B 和 C 都退出临界段，就唤醒进程 D，然后在 D 完成之后再运行进程 E。

11.8　编写 *P* 和 V 函数的修改版本，其中带有一个计数器，记录应该被以原子方式获得的资源的数量。使用银行家算法来防止死锁。

11.9　在图 11-18 中给出的 exit_reader 代码里，为什么在唤醒一个写方时必须将 m_wrcnt 字段设置为 1，而不是让写方自己在 *P* 操作之后的 enter_writer 函数里立即这样做？

11.9　进一步的读物

[1]　Bach, M.J., and Buroff, S.J., "Multiprocessor UNIX Systems," *AT&T Bell Labs Technical Journal*, Vol. 63, No. 8, October 1984, pp. 1733-50.

[2]　Bach, M.J.,*The Design of the UNIX Operating System,* Englewood Cliffs, NJ: Prentice Hall, 1986.

[3]　Beck, B., Kasten, B., and Thakkar, S., "VLSI Assist for a Multiprocessor," *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987, pp. 10-20.

[4]　Black, D.L., Tevanian, Jr., A., Golub, D.B., and Young, M.W., "Locking and Reference Counting in the MACH KERNEL," *Proceedings of the 1992 International Conference on Parallel Processing*, August 1992, pp. II:167-73.

[5]　Blasgen, M.W., Gray, J.N., Mitoma, M., and Price, T.G., "The Convoy Phenomenon," *IBM Research Report,* RJ2516, May 1977, revised January 1979.

[6]　Campbell, M., Barton, R., Browning, J., Cervenka, D.,Curry, B., Davis, T., Edmonds, T., Holt, R., Slice, J., Smith, T., and Wescott, R., "The Parallelization of UNIX System V Release 4.0," *USENIX Conference Proceedings,* January 1991, pp. 307-23.

[7]　Cambell, M., Holt, R., and Slice, J., "Lock Granularity Tuning Mechanisms in SVR4/MP," *Proceedings of the Second Sympoisum on Experiences with Distributed and Multiprocessor Systems*, March 1991, pp. 221-8.

[8]　Campbell, M., and Holt, R.L., "Lock Granularity Analysis Tools in SVR4/MP," *IEEE*

Software, March 1993, pp. 66-70.

[9] CaraDonna, J.P., Paciorek, N., and Wills, C.E., "Measuring Lock Performance in Multiprocessor operating system kernels," *Proceedings of the USENIX Symposiun on Experiences with Distributed and Multiprocessor Systems*, *September* 1993, pp. 37-56.

[10] Courtois, P.J., Heymans, F., and Parnas, D.L., "Concurrent Control with Readers and Writers," *Communications of the ACM*, Vol. 14, No. 10, October 1971,pp. 667-8.

[11] Deitel, H.M., *An Introduction to Operating Systems*, Reading, MA: Addison-Wesley, 1990.

[12] Denning, P.J., Dennis, T.D., and Brumfield, J.A., "Low Contention Semaphores and Ready lists," *Communications of the ACM*, Vol. 24, No. 10, October 1981, PP. 687-99.

[13] Dijstra,E.W., "Solution of a P roblem in Concurrent Programming Control," *Communications of the ACM* ,Vol. 8, No. 5, September 1965, p. 569.

[14] Dijkstra ,E.W.,"Cooperating Sequential Processes,"in *Programming Languages* , F.Genuys, (eb.), Academic Press, New York ,1968, pp.43-112.

[15] Dijkstra, E, W., "The Structure of the T.H.E. Multiprogramming System," *Communications of the ACM*, Vol.11, No.5, may 1968, pp.341-46.

[16] Easton W.B.,"Process Synchronization WITHOUT long-Term Interlock," *ACM Operation Systems Review*, Vol. 6, No. 1, June 1972, pp. 50-95.

[17] Eisenber, M.A., and McGuire, M.R., "Further Comment on Dijkstra's Concurrent Programming Control Problem," *Communications of the ACM*, Vol 15, No. 11, November 1972, p. 999.

[18] Eykholt, J.R., Kleiman, S.R., Barton, S., Faulkner, R., Shivalingiah, A., Smith, M., Stein, D., Voll, J., Wekks, M., and Williams, D., "Beyond Multiprocessing: Multithreading the SunOS Kernel," *USENIX Conference Proceedings*, Summer 1992.

[19] Garg, A., "Parallel STREAMS: A Multiprocessor Implementation," *USENIX Conference Proceedings*, January 1990, pp. 163-76.

[20] Graunke, G., and Thakkar, S., "Synchronization Algorithms for Shared-Memory Multiprocessors," *Computer*, June 1990, pp. 60-9.

[21] Gray, J.N., "Notes on Data Base Operating Systems," *IBM Research Report*, RJ2188(30001), February 19789, p. 83.

[22] Gurwitz, R.F., and Teixeira, T.J., "Stellix: UNIX for a Graphics Supercomputer," *USENIX Conference Proceedings*, June 1988, pp. 321-30.

[23] Habermann, A.N., "Prevention of System Deadlocks," *Communications of the ACM*, Vol. 12, No. 7,, July 1969, pp. 373-7,385.

[24] Hartman, J.H., and Ousterhout, J.K, "Performance Measurement of a Multiprocessor Sprite Kernel," *USENIX Conference Proceedings*, Summer 1990,pp. 279-87.

[25] Kelley, M.H., "Multiprocessor Aspects of the DG/UX Kernel," *USENIX Conference Proceedings, January* 1989, pp. 85-99.

[26] Korty, J.A., "Sema: A Lint-like Tool for Analyzing Semaphore Usage in a

Multithreaded UNIX Kernel," *USENIX Conference Proceedings*, Winter 1989, pp. 113-23.

[27] Lamport, L., "A New Solution to Dijkstra's Concurrent Programming Problem," *Communications of the ACM*, Vol. 17, No. 8, August 1974, pp. 453-55.

[28] Lamport, L., "Concurrent Reading and Writing," *Communications of the ACM*, Vol. 20, No. 11, November 1977, pp. 806-11.

[29] Lauesen, S., "A Large Semaphor-Based Operating System," *Communications of the ACM*, Vol. 18, No,. 11, November 1977, pp. 377-89.

[30] Lee, T.P., and Luppi, M.W., "Solving Performance Problems on a Multiprocessor UNIX System," *USENIX Conference Proceedings*, Summer 1987, pp. 399-405.

[31] Madduri, H., and Finkel, R., "Extension of the Banker's Algorithm for Resource Allocation in a Distributed Operating System," *Information Processing Letters*, Vol. 19, No.1, July 1984, pp. 1-8.

[32] Paciorek, N., LoVerso, S., and Langerman, A., "Debugging Multiprocessor Operating System Kernels," *Proceedings of the Second Symposium on Experiences with Distributed and Multiprocessor Systems*, March 1991, pp. 185-201.

[33] Patil, S.S., "Limitations and Capabilities of Dijkstra's Semaphore Primitive for Coordination among Processes," *M.I.T. Project MAC Computational Structures Group Memo 57*, February 1971.

[34] Sawyer, B.B., "Multiprocessor UNIX Utilizing the SPARC Architecture," *UniForum Conference Proceedings*, February 1989, pp. 107-19.

[35] Saxena, S., Peacock, J.K., Yang, F., Verma, V., and Krishnan, M., "Pitfalls in Multithreading SVR4 STREAMS and Other Weightless Processes," *USENIX Conference Proceedings*, Winter 1993, pp. 85-96.

[36] Torrellas, H., Gupta, A., and Hennessy, J., "Characterizing the Caching and Synchronization Performance of an MP Operating System," *SIGPLAN Notices*, Vol. 27, No. 9, September 1992, pp. 162-74.

第12章 其他MP原语

本章介绍其他一些原语的例子：管程（monitor）、事件计数（eventcount）以及定序器（sequencer）。本章也涵盖了一组原语，源自 UNIX 系统实验室（UNIX System Laboratories）的 UNIX System V 的 MP 版本就使用了它们。我们把这些原语的语义和用法同前面几章介绍的原语和技术进行了比较。

12.1 引 言

数年以来，已经有各种各样的 MP 原语被发明出来。所有的原语不是提供了互斥机制，就是提供了同步机制，或者就是两种功能都提供。正如读者将要看到的那样，它们主要的差别在于，它们处于特殊场合中的方便性或者灵活性如何。为了体现出这些差异，下面几小节介绍了其他几种原语的例子，包括那些在 UNIX SVR 4.2（System V Release 4.2，）MP 软件产品中用到的原语。这些使得操作系统多线程化的原语使用起来和前两章里介绍的用法一样。因此，本章将着重讨论这些原语的语义。

12.2 管 程

管程为临界资源以及访问或者修改该资源的所有临界段提供了互斥机制，它还提供了在使用管程的诸进程间进行同步的手段。一个管程可以想成是一个装有资源的隔间。进程要访问资源，它必须首先进入隔间。通过一次只允许一个进程进入隔间，就做到了互斥。如果在管程已经投入使用的时候，别的进程试图进入它，那么就会被阻塞，直到使用管程的进程退出管程为止，或者在与管程关联的事件上等待。每个管程都可能有一个或者更多的事件，若干进程能够在这些事件上等待。进程被阻塞在这些事件上，直到在管程内执行的其他进程触发事件为止。根据定义，触发操作只能从管程内部完成。这就不需要增加额外的同步机制来解决竞争条件，一个进程可能正要等待，恰好另一个进程要向它发信号时就造成了竞争条件。

当触发一个事件的时候，被阻塞在事件上的一个进程就被唤醒。因为在管程内一次只能唤醒一个进程，所以在发出触发信号的进程退出管程之前，被触发的进程不会继续执行。此刻，被触发的进程在管程内从它等候的位置开始继续执行。管程的语义保证了刚被触发的进程接下来在管程内执行。等待进入管程的其他进程会继续等候，直到所有被触发的进程都结

束为止。如果触发了一个事件，而没有进程等待，那么信号不会造成什么影响，和使用 wakeup 的情况一样。因为管程没有对过去触发操作的记忆，所以一次等候操作一定会让进程被阻塞。

在文献中，管程都是使用特殊的编程语言结构（programming language construct）来表示的，这个结构把临界资源内的数据和构成临界段的例程联系了起来。调用这些例程之一就隐含进入了管程。当它们返回的时候，则显式地退出管程。除了调用这些例程之外，不能访问和管程关联的数据。因为标准 C 语言没有提供直接支持该功能的设施，所以必须和信号量的方式一样，显式地编写管程的代码。可以提供例程来实现进入（enter）、退出（exit）、等候（wait）以及触发（signal）函数。例如，假定有一个链表，需要多个处理器对它进行修改。可以使用图 12-1 中的例程来使用管程保护临界资源，向列表中加入元素，以及从列表中删除元素。假如列表为空，那么删除函数就等着有元素被加入到列表中（因为使用了伪代码，为简洁起见，省略了链表及其元素的声明）。

```
monitor_t list_monitor;

enum list_events { LIST_NOTEMPTY };

void
add( elem_t *new )
{
   mon_enter( &list_monitor );
   向列表中增加新元素
   mon_signal( &list_monitor, LIST_NOTEMPTY );
   mon_exit( &list_monitor );
}

elem_t *
remove(void)
{
    mon_enter( &list_monitor );
    if(列表为空)
         mon_wait( &list_monitor, LIST_NOTEMPTY );

    从列表中删除元素
    mon_exit( &list_monitor );
    return element;
}
```

图 12-1　管程用法的 C 语言示例

在本例中，和采用信号量一样，用一个称为 monitor_t 的小数据结构表示每个管程。例程 add 和 remove 是临界段，所以它们必须按照图示进入和退出管程。和管程相关的事件列为 -

种枚举类型。在这个例子中只有一个事件，但是在一般情况下可以有任意数量的事件。如前所述，在事件上等待的动作会阻塞进程，而让其他的进程进入管程。因为被触发的进程比其他试图进入管程的优先级高，所以 remove 函数在删除一个元素之前不必再度检查列表的状态，因为它得到保证，刚才加入的元素还在。最后要说明的一点是，为了遵循管程的定义，例程 mon_signal 和 mon_wait 必须只能从函数对 mon_enter 和 mon_exit 中来调用。

12.3 事件计数和定序器

事件计数是一个非递减的正整数值，在这个数值上定义了 3 种操作。操作 advance(*E*)将事件计数 *E* 加 1，这叫做触发事件。操作 await(*E*, *V*)致使调用进程被阻塞，直到事件计数 *E* 的值达到值 *V* 为止。如果在调用 await 的时候，事件计数的值大于或者等于 *V*，那么进程继续执行，而不会阻塞，因为事件是在以前触发的。事件计数的当前值可以用 read(*E*)来读取。在创建事件计数的时候，它被初始化为 0，而且在数值上永远不会减小。假定保存事件计数值的存储器位置足够大，于是事件计数在其整个生命期中，一直都不会溢出（通常一个 32 位的无符号整数就够了）。

和事件计数相关的是定序器，它是一个非递减的正整数值，在这个数值上定义了一种操作。操作 ticket(*S*)将定序器加 1，并且返回新值，它是一项原子操作。因此，多个调用者同时调用同一个定序器的 ticket 操作时，保证都会得到唯一的返回值。和事件计数一样，定序器也初始化为 0，而且在它们的生命期内必须永远不溢出。事件计数和定序器至少在一种 UNIX 系统的 MP 版本中使用过：即 Data General 公司的 DG/UX 4.00(参见 12.8 节的参考文献[17])。

事件计数和定序器可以用于通过各种途径实现互斥和同步机制。例如，可以用图 12-2 所示的数据结构像二值信号量那样实现互斥。像这样的简单互斥锁经常简称为 mutex 锁。

```
struct mutex {
    eventcount_t event;     /* 事件计数，在其上等候*/
    sequencer_t seq;        /* 决定次序 */
};

typedef struct mutex mutex_t;

void
init_mutex( mutex_t *mp )
{
     init-eventcount( &mp->event );
     init_sequencer( &mp->seq );
     advance( &mp->event );
}
```

图 12-2 采用事件计数和定序器实现的 mutex 锁

事件计数和定序器分别用数据结构 eventcount_t 和 sequencer_t 来表示。通过调用 init_eventcount 和 init_sequencer 来对它们进行初始化，它们将事件计数和定序器初始化为 0（事件计数和定序器的实现留作习题）。采用图 12-3 中的代码可以获得和释放 mutex 锁。

```
void
lock_mutex( mutex_t *mp );
{
        int my_turn;

        my_turn = ticket( &mp->seq );
        await( &mp->event, my_turn );
}

void
unlock_mutex( mutex_t *mp )
{
        advance( &mp->event );
}
```

图 12-3　对 mutex 锁上锁和开锁

这里的策略是，使用定序器给每个进程一个唯一的标签，就能让多个进程按照它们的标签值依次进入临界段。这好比商店里的服务台，客户在进商店的时候拿一个号，按照进店的顺序接受服务。定序器分发唯一的顺序编号，而事件计数决定下一个轮到谁（类似于商店里的“现在服务第 n 号客户”）。于是每个调用 lock_mutex 的进程都从定序器那里得到下一个标签，并且等着轮到它。因为定序器返回的第一个标签为 1，所以必须用图 12-2 中的初始化代码让事件计数前进一号。这就能让第一个调用 lock_mutex 的进程立即获得锁，因为事件计数等于第一个让 await 调用返回而不阻塞进程的标签。

在释放锁的时候就增加事件计数，让具有下一个标签值的进程获得锁。注意，这里不像使用 sleep/wakeup 实现的 lock_object 代码那样（参见图 10-2）需要一个自旋锁。这是因为，即使没有进程在等下一个值，事件计数也会保留它的状态，即计数器的值。这就消除了如果没有进程在事件上睡眠，wakeup 没有作用所导致的竞争条件。还要说明的一点是，定序器在竞争锁的多个进程之间推行了一种严格的 FIFO 次序。另一方面，可以实现信号量来让进程按照它们的进程优先级获得锁。

事件计数还可以用于只要求进程同步的情形。考虑在生产者进程和消费者进程之间的双缓冲机制。策略是使用两个事件计数：一个用于在缓冲满了的时候执行触发操作，一个用于在缓冲空了的时候执行触发操作。声明和初始化代码如图 12-4 所示。

图 12-5 给出了生产者进程和消费者进程的代码。

```
buf_t       buffer[2];
eventcount_t  full;
eventcount_t  empty;
...
init_eventcount (&full);
init_eventcount (&empty);
advance (&empty);
advance(&empty);
```

图 12-4 双缓冲示例的初始化代码

生产者

```
int cur_buf;
int next;

cur_buf = 0;

for (next = 1; ; next++) {
    await(&empty, next);
    填充 buffer[cur_buf]
    advance(&full);
    cur_buf = !cur_buf;
}
```

消费者

```
int cur_buf;
int next;

cur_buf = 0;

for (next = 1; ; next++) {
    await(&full, next);
    使用 buffer[cur_buf]
    advance(&empty);
    cur_buf = !cur_buf;
}
```

图 12.5 有事件计数的生产者-消费者进程

当一个进程结束其使用缓冲的任务时，它就通过增加相应的事件计数值来通知另一个进程。每个进程都知道，在它可以使用变量 next 中下一个缓冲数据之前，事件计数所必须达到的值。因为两个缓冲一开始都是空的，所以初始化代码两次增加事件计数 empty 的值，从而使得生产者进程先要填写两个缓冲，然后才进入阻塞，等待消费者进程清空第一个缓冲。

12.4 SVR4.2 MP 的 MP 原语

SVR4.2 MP 是一个运行在 SMP 体系结构上的细粒度、多线程的操作系统，它同样也使用前面几章中介绍的上锁策略。SVR 4.2 MP 提供了各种各样的原语，其中一部分已经公开发表了。随后几小节介绍的这些原语的语义和函数原型都取自于 12.8 节中参考文献[10]里列出的 *SVR 4.2 Device Driver Reference Manual*。提供这本手册是为了让开发人员能够编写可以同基础操作系统链接的多线程设备驱动程序。

12.4.1 自旋锁

SVR4.2MP 提供的自旋锁实现类似于本书介绍的自旋锁，它带有附加的参数，用以屏蔽中断，并且在调试期间起到辅助作用。自旋锁被声明为 lock_t 类型，并且通过调用下面给出的 C 函数原型格式来分配空间。LOCK_ALLOC 动态地为自旋锁分配空间，初始化它，并且

返回一个指向它的指针。这样做既方便又灵活，使得独立编译的内核模块不会依赖于 lock_t 结构的大小。

```
lock_t *LOCK_ALLOC( uchar_t hier, pl_t minpl, lkinfo_t
                    *lkinfop, int slpflg );
```

根据约定，所有的 MP 原语接口都用大写字母，以体现出它们是由#define 而不是由函数来定义的。这能让它们的实现易于被重新定义（例如，为了替换为调试版本或者收集统计信息的版本）。

除了 slpflg 之外的参数都只会在调试版本以及收集统计信息的版本上出现。要使用它们必须要特意编译一下内核，因为它们给每项操作都增加了开销；否则，这些参数均被忽略。

参数 hier 是一个指出锁在锁层次结构（lock hierarchy）中的位置的小整数（uchar_t 类型是一个无符号字符）。由于 AB-BA 死锁问题（参见 9.3 节），所以系统的实现者为出现锁的嵌套的所有实例都规定了锁的次序。于是，在每一个嵌套层次上的锁都被赋予了一个在层次结构内唯一的号，最外层（或者说第一层）给的号最小。在后续的每个嵌套层次上，该层次号不断增加。通过在对锁进行初始化的时候把锁的层次号和锁一起保存，每次获得锁以及释放锁的时候都能对它进行检查，以确保没有按照不正确的次序使用锁。于是，如果有潜在的死锁情况要发生，那么该系统就能够向内核开发人员报警。

如果一个中断处理程序试图获得一个已经在同一个处理器上被锁住的锁（在 9.3 节中曾介绍过），那么就会出现另一种死锁情形。为了有助于检测到这类情形，由参数 minpl 给出了一个最小的中断优先级，指出当相应的自旋锁被占据时处理器应该处于的最小中断优先级。等于或者小于这个级别的中断都将被屏蔽，所以在与那些级别相关联的中断处理程序中使用同一个自旋锁是安全的。在 minpl 之上的中断仍然会发生，所以，在那些中断处理程序中，不能使用处理器可能占据的自旋锁中的任何一个。参数 minpl 的值是和实现有关的，但是一般取小整数值，值越大，表示优先级越高。plbase 定义的值意味着允许所有的中断。

参数 lkinfop 是一个指向锁信息（lock information）结构的指针，该结构包含了锁的类型（在这里是自旋锁），以及一个 ASCII 字符串，它是锁的助记名。它用来让调试输出和统计报告的可读性更好。

最后，LOCK_ALLOC 例程需要分配内存来保存自旋锁和调试信息。参数 slpflg 是一个标志，指出为了在不能立即获得内存的情况下分配内存，是否能够阻塞调用进程。初始化自旋锁的中断处理程序应该传递 KM_NOSLEEP 标志，因为它们没有要阻塞的进程现场。

如果只是临时需要一个自旋锁，那么在不再需要它时，可以用下面的函数回收它：

```
void LOCK_DEALLOC( lock_t *lockp );
```

用下面的函数可以锁住自旋锁：

```
pl_t LOCK( lock_t *lockp, pl_t ipl );
```

参数 ipl 指定要屏蔽的中断优先级。等于或者低于这个优先级的所有中断都将被屏蔽，直到自旋锁被释放为止。它必须大于或者等于在初始化锁时指定的 minpl。函数 LOCK 返回在提高到 ipl 之前起作用的中断优先级。正如 9.2 节所介绍的那样，出于性能的原因，占据自旋锁的同时屏蔽中断是很重要的，因为在占有锁的同时发生一次中断会延长其他进程等待该

锁时自旋的时间。还需要防止如果一个中断处理程序试图获得一个已经被处理器占据的锁时可能发生的死锁。出于这些原因，SVR4.2 MP 接口将自旋锁和中断屏蔽组合成了单项操作。

通过调用

```
void UNLOCK( lock_t *lockp, pl_t ipl );
```

来释放自旋锁。

参数 ipl 应该是 LOCK 返回的值，这把中断优先级恢复到获得锁之前的值。

因为偶尔也会希望只有当自旋锁没有投入使用的时候才获得它（以防出现这样的情况，即实现不想自旋任意长的一段时间，或者需要不按顺序地获得锁），所以提供了一个条件锁函数：

```
pl_t TRYLOCK( lock_t *lockp, pl_t ipl );
```

它按照和 LOCK 函数一样的方式，但是只有在锁为空闲的时候，才将中断优先级提高到 ipl，并且获得锁。如果锁已经被锁住了，它就返回符号 invpl（一个在头文件中定义的与实现无关的符号常数）的值，而不是原来的 ipl 级别。

12.4.2 睡眠锁

SVR4.2 MP 没有实现信号量、管程或者事件计数。相反，设计者选择将互斥和同步分开，并且为这两种需求提供不同的原语。睡眠锁（sleep lock）提供了二值信号量的功能，也同样用于长期互斥。它们得名于这样的事实，如果锁已经在使用了，那么试图获得锁的进程就睡眠。

睡眠锁被声明为 sleep_t 类型，并且用下面的函数来分配空间：

```
sleep_t *SLEEP_ALLOC( int arg, lkinfo_t *lkinfop, int slpflg );
```

这个函数给睡眠锁分配空间，并且把它初始化为开锁状态。和采用自旋锁的实现一样，只有在调试期间才使用 lkinfop。这个参数的含义和 12.4.1 小节中的一样。参数 arg 是为将来的实现保留的。

使用下面的函数可以回收一个睡眠锁。

```
void SLEEP_DEALLOC( sleep_t *slp );
```

使用下面的函数可以获得一个睡眠锁：

```
void SLEEP_LOCK( sleep_t *slp, int pri );
```

和采用二值信号量一样，如果有锁可用，进程就获得锁，并且继续执行。如果锁已经在使用了，那么它就睡眠。在这种情况下，参数 pri 指出当进程被唤醒时，在赋予该进程锁之后，进程希望运行的调度优先级。这能让实现来控制在系统内争夺锁的进程的相对优先级。例如，给予被高度争用的锁较高的优先级，以便占据锁的进程一被唤醒就调度运行。这能让它很快运行并完成临界段，于是它就能释放锁供其他进程使用。如果等候临界锁的进程的优先级被设置得太低，那么当占有锁的进程在运行队列中等着轮到它执行的时候，可能就会形成结对现象。

提供的 SLEEP_LOCK 调用的一种变形能和 UNIX 信号进行交互。因为信号可能异步到达，所以接收信号的进程可能在信号到达的时候正在执行一个系统调用。在典型情况下，信号会被忽略，直到系统调用完成，因为这能简化内核的设计。有几个系统调用，如果有信号到达就会让系统调用中止。对于要花任意长的时间才能完成的系统调用来说，比如在管道上等待数据到达，或者在系统调用 read 期间等待数据从终端来，就是这样。在终端 I/O 的情况下，如果终端没有使用，那么输入可能要在几分钟或者几小时才能到达。把所有的信号都推迟那么久是没有实用价值的，如果有人正在试图通过向它发送信号来终止这样的一个进程时尤其如此。因此，所有的 UNIX 内核实现都提供了一种手段来指出信号能够中断长期睡眠。在单处理机实现中，这是通过向 sleep 调用传送低优先级的 pri 参数做到的。在数值上比 PZERO 所定义的阈值大的优先级被认为是可以由信号中断的，别的值则让信号被忽略。

SVR4.2 MP 通过单独的接口提供这种功能。接口 SLEEP_LOCK 忽略为等候锁而睡眠时的全部信号。接口 SLEEP_LOCK_SIG 使得进程在等候锁的时候，如果到达一个不被保持、不被忽略的信号，就放弃睡眠（进程可能会选择忽略或者保持传递的某些信号，因此，有这样的信号到达不会影响进程）。这个函数的原型如下：

```
bool_t SLEEP_LOCK_SIG( sleep_t *slp, int pri );
```

如果有锁可用，或者如果进程必须为锁而睡眠，又没有信号出现，那么这个函数就返回 TRUE（一个非零值），这个调用的效果等同于 SLEEP_LOCK。如果当进程正在睡眠的时候接收到一个信号，那么进程就被唤醒，函数返回 FALSE（零值），表明它没有获得锁。于是内核采取所需的任何恢复措施来放弃系统调用。重要之处在于，当系统调用返回 FALSE 时，没有把锁交给进程。因此，它一定不会进入由锁保护的临界区，可能有其他某个进程正在其中执行。

采用下面的函数可以打开睡眠锁，不管锁是用 SLEEP_LOCK 还是用 SLEEP_LOCK_SIG 获得的。

```
void SLEEP_UNLOCK( sleep_t *slp );
```

用睡眠锁配合有条件的上锁操作偶尔会派上用处。SVR4.2 MP 通过下面的函数提供了这项功能：

```
bool_t SLEEP_TRYLOCK( sleep_t *slp );
```

如果得到了锁，它就返回 TRUE，否则返回 FALSE（说明另一个进程在使用该锁）。和前面一样，返回 FALSE 的时候，进程一定不能进入临界区。因为这个调用保证不会被屏蔽，所以它可能会用在中断处理程序中。

总而言之，睡眠锁在功能上和使用上和二值信号量是等同的。因此，同样也会有死锁现象和限制条件：要求锁有次序，以避免 AB-BA 死锁，在队列中两次锁住同一个锁的进程将会和自身发生死锁现象，中断处理程序不能使用 SLEEP_LOCK 和 SLEEP_LOCK_SIG 接口。

12.4.3　同步变量

SVR4.2 MP 提供了单独的执行进程同步的原语，称为同步变量（synchronization

variable）。因为同步变量不包含状态，所以可以把它们想成是 sleep/wakeup 的一种 MP 变形。相反，所需的任何状态信息都保存在外部标志或者计数器中。当采用 sleep/wakeup 在 MP 系统上测试或者更新状态信息时，出现的竞争条件和采用同步变量出现的竞争条件是一样的。正如 10.5 节所介绍的那样，使用自旋锁能够防止这类竞争。因此，同步变量的设计要同自旋锁配合工作（正如将要在 12.5 节里看到的那样，同步变量的使用倾向于类似管程的使用）。

同步变量被声明为 sv_t 类型，采用下面的函数可以给它分配空间和进行初始化：

```
sv_t *SV_ALLOC( int slpflag );
```

和通常一样，slpflag 指定，如果需要为同步变量分配内存，那么是否能阻塞进程。回收同步变量可以调用

```
void SV_DEALLOC( sv_t *svp );
```

内核希望单独等候的每个事件都用一个不同的同步变量来表示，这就好比配合 sleep 如何使用唯一的事件参数。要等待一个同步变量上发生的事件，可以使用下面的函数：

```
void SV_WAIT( sv_t *svp, int pri, lock_t *lockp );
```

和采用睡眠锁一样，pri 指出，在事件发生且进程被唤醒时，进程将以什么样的优先级运行。参数 lockp 指出在调用 SV_WAIT 之前进程必须要占有的一个自旋锁。接着 SV_WAIT 调用以原子操作打开这个自旋锁，将进程在指定的同步变量上挂起。我们马上会说明这样做的重要性。既然 SV_WAIT 函数打开了自旋锁，那么如果调用方（caller）仍然需要锁，就必须在 SV_WAIT 返回后立即再次获得它。当 SV_WAIT 释放了自旋锁之后，它也消除了对所有中断的屏蔽。

要触发在同步变量上的事件，可以使用下面的函数（参数 flags 目前尚未实现）：

```
void SV_SIGNAL( sv_t *svp, int flags );
```

这个函数唤醒了一个正在同步变量上睡眠的进程。不管它的名字怎么叫，这个调用与上一节讨论的 UNIX 信号机制无关（它并不发出 UNIX 信号）。SV_SIGNAL 与 wakeup 的相似之处在于，如果变量上没有正在睡眠的进程，那么就对过去曾经执行过的操作没有记忆，调用什么也不做。这是同步变量和其他 MP 原语比如信号量和事件计数的主要区别之一，因为不论是否有进程正在等候，V 或者 advance 操作都会增加信号量或者事件计数的值。

sv_wait 和 sv_signal 合起来的表现非常像 sleep/wakeup。例如，可以如图 12-6 所示来实现 lock_object 和 unlock_object 例程（虽然人们不会用同步变量来实现长期互斥，因为睡眠锁更方便，但是这个例子还是给出了同 sleep/wakeup 的一个不错的对比）。

把这段代码同图 10-2 比较一下，后者使用 sleep，可以看到许多相似之处。首先，因为同步变量不保留状态，所以仍然要在对象的数据结构中有一个标志，以指示锁的状态。其次，仍然需要一个自旋锁，以使测试标志同设置标志或者睡眠之间不会有竞争。注意，在调用 SV_WAIT 之后，必须显式地重新获得自旋锁。在图 10-2 的例子中，假定进程睡眠时，内核会自动释放和重新获得自旋锁。同步变量要求上述操作显式地完成。

```
lock_object( char *flag_ptr, sv_t *svp )
{
    lock( &object_locking );

    while( *flag_ptr ) {
        sv_wait( svp, PRT, &object_locking );
        lock( &object_locking );
    }
    *flag_prt = 1;
    unlock( &object_locking );
}
```

图 12-6　使用同步变量锁住一个对象

图 12-7 显示了给对象解锁的代码，它也和 sleep/wakeup 版本的代码很相似。

```
unlock_object( char *flag_ptr, sv_t*svp )
{
    lock( &object_locking );
    *flag_ptr = 0;
    SV_SIGNAL( svp, 0 );
    unlock( &object_locking );
}
```

图 12-7　使用同步变量给对象解锁

和图 10-3 中的代码一样，这里也需要自旋锁，以使触发一个同步变量和在该变量上等待的进程之间不会有竞争。这个版本的代码同以前使用 wakeup 的版本之间主要的区别在于，SV_SIGNAL 只唤醒一个进程（就像 wakeup_one 那样），而 wakeup 唤醒在事件上睡眠的所有进程。后者的效果可以通过使用这个函数，用同步变量做到：

```
void SV_BROADCAST( sv_t *svp, int flags );
```

和采用睡眠锁的情况一样，也存在内核不能无限期地等待事件发生的情形。如果在事件被触发之前出现了一个 UNIX 信号，那么下面的 SV_WAIT 变形会唤醒进程：

```
bool_t SV_WAIT_SIG( sv_t *svp, int pri, lock_t *lkp );
```

返回的代码表明发生了什么样的事件：如果出现了一个 UNIX 信号，那么它返回 FALSE；如果出现了 SV_SIGNAL 或者 SV_BROADCAST，那么它返回 TRUE。

12.4.4　多读锁

SVR4.2 MP 也提供了多读-单写锁。这些锁被声明为 rwlock_t 类型，采用下面的函数给它们分配空间，以及对它们进行初始化：

```
rwlock_t *RW_ALLOC( uchar_t hier, pl_t minpl, lkinfo_t *lkp, int sleep );
void RW_DEALLOC( rwlock_t *lockp );
```

RW_ALLOC 的参数定义和 LOCK_ALLOC 一样。使用下面的函数进行读或者写时需要一个多读锁：

```
pl_t RW_RDLOCK( rwlock_t *lockp, pl_t ipl );
pl_t RW_WRLOCK( rwlock_t *lockp, pl_t ipl );
```

如果不能立即获得锁，那么这些函数是阻塞进程还是自旋就取决于实现。读方的锁和写方的锁都使用下面的函数来释放：

```
void RW_UNLOCK( rwlock_t *lockp, pl_t ipl );
```

这些函数的参数的含义都和前面几小节中介绍的相同。

12.5 比较 MP 同步原语

本节将通过一个例子对信号量和本章介绍的同步变量原语进行比较。

考虑把内核检测到的错误信息记录到一个磁盘文件的情形。出错信息是通过内存中的一个队列来传递给日志进程（logging process）的。当出现一个错误时，就在队列中加入一项，并且通过调用函数 log_error 通知日志进程。出错日志进程接着把队列中的项写到磁盘上（或者采取其他适当的措施）。这就使碰到错误的进程不必等候 I/O 完成或者获得为了向文件执行 I/O 而可能需要的任何锁，并且避免了任何可能的上锁次序问题，如果在文件或者磁盘 I/O 子系统内执行时检测到一个错误，就可能会出现这样的问题。

要采用事件计数来实现通知日志进程的机制，函数 log_error 将会在它每次向队列加入一条出错信息时将事件计数递增。函数和日志进程的伪代码如图 12-8 所示（假定事件计数和通常一样初始化为 0）。

```
log_error(error)
{
    lock(&err_queue);
    把出错信息加入队列
    unlock(&err_queue);
    advance(&err_event);
}
```

日志进程

```
for (next = 1; ; next++) {
    await(&err_event, next);
    lock(&err_queue);
    从队列中删除项
    unlock(&err_queue);
    write error to disk
}
```

图 12-8 采用事件计数的出错日志通知机制

队列本身由一个自旋锁来保护。在本例中，事件计数只用于同步目的，并且不提供互斥。日志进程通过等待事件计数的下一个连续值，来等待队列中要处理的一项到达。然后它处理错误，并等候下一个错误的发生。如果在它已经处理完一个错误之后，队列中又加入了几个新的错误，那么在它处理完挂起的所有错误之前，对 await 的随后几次调用不会阻塞。

图 12-9 所示的代码用同步变量重新实现了前面的算法。

```
log_error(error)
{
    lock(&err_queue);
    把出错信息加入队列
    SV_SIGNAL(&err_syncvar, 0);
    unlock(&err_queue);
}
```

日志进程

```
for (;;) {
    lock(&err_queue);

    if (queue empty) {
        SV_WAIT(&err_syncvar, PRI,
            &err_queue);
        lock(&err_queue);
    }

    从队列中删除项
    unlock(&err_queue);
    把错误写入磁盘
}
```

图 12-9　采用同步变量的出错日志通知机制

注意编程范例的变化。因为同步变量自身没有保留状态，所以当日志进程测试队列的状态并决定是等待一项还是从队列中删除一项的时候，必须占有自旋锁。类似地，log_error 在发送信号的同时必须占有自旋锁。这就确保了，要么是日志进程已经遇到了 SV_WAIT 调用（它自动释放自旋锁并阻塞进程），要么就是它正在执行磁盘 I/O。在前一种情况下，信号将唤醒日志进程，日志进程又发现队列中的新错误。在后一种情况下，它将在 for 循环的下一次循环中发现错误。不论采用哪一种方式，都能避免竞争条件。注意，在唤醒进程的时候，进程在队列中至少会发现一项的要求得到了保证，因为它是唯一要从队列中删除项的进程。在调用 SV_WAIT 之前，还必须检查队列的状态，因为在它写前一个错误的同时，可能又有错误已经加入到了队列中。如果它要在每次循环中都调用 SV_WAIT，那么它必须等到又有一个错误加入了队列，让它暂时忽略已经在队列中的错误。

在使用事件计数的时候，这些步骤并不必要，因为 advance 操作会永久性地改变事件计数的状态。advance 和 await 操作的相对时序没有关系。事件计数的状态消除了使用同步变量时存在的竞争条件。

如图 12-10 所示，这个算法的下一个版本使用了管程。这个算法在根本上和采用同步变量的算法是一样的，不同之处在于，管程的 enter 和 exit 函数替换了需要自旋锁的地方。用对临界段的互斥访问唤醒日志进程的做法稍微简化了代码。和以前一样，采用管程事件时缺乏状态，要求在日志进程的每次循环中都重新检查队列。

```
log_error(error)
{
    mon_enter(&err_mon);
    把出错信息加入队列
    mon_signal(&err_mon,
                NEWENTRY);
    mon_exit(&err_mon);
}
```

日志进程

```
for (;;) {
    mon_enter(&err_mon);

    if (queue empty)
        mon_wait(&err_mon,
                NEWENTRY);

    从队列中删除项
    mon_exit(&err_mon);
    把错误写入磁盘
}
```

图 12-10　采用管程的出错日志通知机制

使用信号量重新实现这个例子，得到的代码如图 12-11 所示。要记住，用于同步的信号量需初始化为 0（在本例中没有体现出来）。

```
log_error(error)
{
    lock(&err_queue);
    把出错信息加入队列
    unlock(&err_queue);
    V(&err_sema);
}
```

日志进程

```
for (;;) {
    P(&err_sema);
    lock(&err_queue);
    从队列中删除项
    unlock(&err_queue);
    把错误写入磁盘
}
```

图 12-11 采用信号量的出错日志通知机制

这个算法类似于使用事件计数的算法。和采用事件计数时一样，要注意，在占有自旋锁的同时不需要触发日志进程。信号量的状态消除了这些竞争。因为在每次向队列中加入一个错误的时候，都会把信号量的值加 1，所以日志进程为每个错误只执行一次 *P* 操作。可以看到，这样会简化代码，因为其至不需要像图 12-8 那样知道事件计数的下一个预计值了。这就体现出了这 4 种同步原语之间的不同：它改变了实现算法所需代码的编程复杂性。不同的原语更适于在不同的环境下简单的使用。在这些例子中，信号量给出了最简单的代码，但是算法的 4 个版本之间在性能上没有明显的差异。提供多种原语的系统之所以这样做，仅仅是为了编码上的方便。因此一般说来，需要同步的任何情形都能用任何一种这样的原语来实现，对于提供互斥的原语来说也是这样。在大多数情况下，关注的焦点在于，适合于一个特殊算法的是一个特殊原语的自旋锁、睡眠锁还是多读锁版本。

12.6 小 结

MP 原语有许多种类型。所有的类型都可以用于实现互斥，实现同步，或者实现这两者。对于一个特殊的场合来说，除了有自旋语义和睡眠语义的对比之外，原语之间的主要区别是在特定环境下使用它们的方便程度。

管程提供了互斥和同步，可以把它想成是一个隔间，在访问临界资源之前必须进入这个隔间。在管程中一次只允许有一个进程，这就实现了互斥。一个进程在管程内的时候，可以等待一个事件发生，或者可以触发一个事件，其他进程可能正在等待这个事件。既然所有的同步都是从管程内出现的，所以不需要附加的锁来消除要等在一个事件上的进程和要触发它的进程之间的竞争条件。

事件计数是一个非递减的正整数值，它初始为 0。使用函数 advance 可以使事件计数递增 1。进程可以使用函数 awaitg 来等待要到达的一个特殊的事件计数值。定序器类似于事件计数，不同之处在于，它只能完成一种操作。函数 ticket 使定序器递增 1，并且返回新的值，这是作为一项原子操作完成的。因此保证了不管同时有多少个调用方，调用 ticket 函数都会接收到一个唯一的值。事件计数和定序器可以用于实现互斥和同步。

SVR4.2 MP 是 UNIX 操作系统的一个 MP 版本，该操作系统源自于 UNIX 系统实验室（UNIX System Laboratories）。它向开发人员提供了多种 MP 原语：自旋锁、睡眠锁、同步变量以及多读锁。睡眠锁提供互斥的方式和二值信号量的方式类似。同步变量的作用类似于单处理机上的 sleep/wakeup 机制，因为它不会记录以前的操作。

12.7 习　　题

12.1　使用自旋锁，给出 12.2 节中描述的 mon_enter、mon_exit、mon_signal 和 mon_wait 函数的 C 语言实现。还要给出内部数据结构和初始化的代码。

12.2　如果管程没有保证被触发的进程在试图进入管程的其他进程之前运行，那么必须怎样编写图 12-1 中的代码？这可取吗？

12.3　有可能实现这样一个版本的管程吗？其中，当进程等待进入管程的时候是自旋而不是阻塞。这有用处吗？在这种情况下的等待操作函数应该是什么样的（也就是说，它应该自旋还是睡眠）？

12.4　使用自旋锁，给出 12.3 节中描述的 advance、await、read 和 ticket 函数的 C 语言实现。还要给出内部数据结构和初始化的代码。

12.5　只用 advance、await 和 read，如何采用事件计数实现一个广播型的操作？假定想要唤醒正在等待事件计数任何值的所有进程。

12.6　用自旋锁和信号量重新编写图 12-1 中所示的代码。讨论每种方法的优缺点。

12.7　用事件计数和定序器重做上题。

12.8　给出睡眠锁的 C 代码实现。定义数据结构 sleep_t 的内容，为 SLEEP_ALLOC、SLEEP_DEALLOC、SLEEP_LOCK、SLEEP_TRYLOCK 和 SLEEP_UNLOCK 编写函数。忽略调试参数和优先级参数。用 SVR4.2 自旋锁原语摹仿图 11-5 到 11-8 所示的信号量来实现。分别使用 malloc 和 free 分配和回收 sleep_t 结构的空间。

12.9　给出同步变量的 C 代码实现。定义数据结构 sv_t 的内容，为 SV_ALLOC、SV_WAIT、SV_SIGNAL 和 SV_BROADCAST 编写函数。忽略调试参数和优先级参数。

12.10　使用同步变量而不是信号量重做习题 11.3。

12.11　使用事件计数重做习题 12.10。

12.12　使用事件计数而不是 sleep/wakeup 重新编写习题 10.4 给出的代码。

12.8 进一步的读物

[1]　Brinch, H.P., "Structured Multiprogramming," *Communications of the ACM*, Vol. 15, No. 7, July 1972, pp. 574-78.

[2]　Brinch, H.p., *Operating System Principles*, Englewood Cliffs, NJ: Prentice Hall, 1973.

[3]　Brinch, H.P., "The Solo Operating System: Processes, Monitors, and Classes," *Software-Practice and Experience*, Vol. 6, 1976, pp. 165-200.

[4] Campbell, M., Barton, R., Browning, J., Cervenka, D., Curry, B., Davis, T., Edmonds, T., Holt, R., Slice, J., Smith, T., and Wescott, R., "The Parallelization of UNIX System V Release 4.0," *USENIX Conferece Proceedings*, January 1991, pp. 307-23.

[5] Campbell, M., Holt, R., and Slice, J., "Look Granularity Tuning Mechanisms in SVR4/MP," *Proceedings of the second Sympoisum on Experiences with Distributed and Multiprocessor Systems*, March 1991, pp. 221-8.

[6] Campbell, M., and Holt, R.L., "Lock Granularity Analysis Tools in SVR4/MP," *IEEE Software*, March 1993, pp. 66-70.

[7] CaraDonna, J.P., Paciorek, N., and Wills, C.E., "Measuring Lock Performance in Multiprocessor Operting System Kernels," *Proceedings of the USENIX Symposium on Experiences with Distrbuted and Multiprocessor Systems*, September 1993, pp. 37-56.

[8] Deitel, H.M., *An Introduction to Operating Systems*, Reading, MA: Addison0Wesley, 1990.

[9] Dijkstra, E.W., "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, Vol. 1, 1971, pp. 115-38.

[10] Hines, R.M., and Wilcox, S., editors, *Device Driver Reference: UNIX SVR4.2*, Englewood Cliffs, NJ: Prentice Hall, 1992.

[11] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, Vol. 17, No 10, October 1974, pp. 549-57.

[12] Hoare, C.A.R., "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978, pp. 666-77.

[13] Hoare, C.A.R., *Communicating Sequential Processes*, Englewood Cliffs, NJ: Prentice Hall, 1985.

[14] Howard, J.H., "Proving Monitors," *Communications of the ACM*, Vol. 19, No. 5, May 1976,pp. 273-9.

[15] Howard, J.H., :"Signaling in Monitors," *Second International Conference on Software Engineering*, San Francisco, October 1976, pp. 47-52.

[16] Keedy, J., "On Structuring Operating Systems with Monitors," *Operating Systems Review*, Vol. 13, No. 1, January 1979, pp. 5-9.

[17] Kelley, M.H., "Multiprocessor Aspects of the DG/UX Kernel," *USENIX Conference Proceeding*, January 1989, pp. 85-99.

[18] Kessels, J.L.W., "An Alternative to Event Queues for Synchronization in Monitors," *Communications of the ACM*, Vol. 20, No. 7, July 1977, pp. 500-3.

[19] Korth, H.F., "Locking Primitives in a Database System," *Journal of the ACM*, Vol. 30, No. 1, January 1983,pp. 55-79.

[20] Lampson, B.W., and Redell, D.D., "Experience with Processes and Monitors in MESA," *Communications of the* ACM, Vol. 23, No. 2, February 1980, pp. 105-117.

[21] Lister, A.M., and Maynard, K.J., "An Implementation of Monitors," *Software-Practice and Experience*, Vol. 6, No. 3, July 1976, pp. 377-86.

[22]　Reed, D.P., and Kanodia, R.K., "Synchronization with Eventcounts and Sequencers," *Proceedings of the 6th ACM Symposium on Operating System Principles*, 1977, p. 91.

[23]　Reed, D.P., and Kanodia, R.K., "Synchronization with Eventcounts and Sequencers," *Communcations of the ACM*, Vol. 22, No.2, February 1979, pp. 115-23.

[24]　Wegner, P., and Smolka, S.A., "Processes, Tasks, and Monitors: A Comparative Study of Concurrent Programming Primitives," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 4, 1983, pp. 446-62.

第13章 其他存储模型

本章讨论非顺序存储模型（nonsequential memory model）。这些模型为了提升存储系统的性能而改变了存储器操作发生的顺序。还将介绍在 SPARC 体系结构中实现的全部 store 定序（total store ordering，TSO）和部分 store 定序（partial store ordering，PSO），包括它们对 UP 和 MP 内核的影响。

13.1 引 言

本书前面的所有例子都假定硬件实现了顺序存储模型（强定序），这是在过去以及将来的计算机系统中使用最广泛的存储模型。顺序存储模型强制存储器操作（比如 load 和 store）都按照程序次序（program order）来执行，即这些指令是按照在随程序执行的指令流中出现的顺序次序（sequential order）来执行的。它也指定了，由不同处理器完成的 load 和 store 操作也要以某种顺序的、但又是非确定性的方式排序。为了说明这一点，可以用图 13-1 中的代码片段（由两个处理器来执行）来测试顺序存储模型的表现（注意，这个例子以及随后的例子都用简化的汇编语言来编写，而不是任何特殊处理器的汇编语言，以避免因为语法细节而使例子的含义模糊）。

处理器 1		处理器 2	
store	%r1,A	store	%r1,B
load	%r2,B	load	%r2,A

图 13-1 测试顺序存储模型的代码片段

两个处理器同时开始执行，但是因为总线将对内存的访问串行化了，所以其中一条 store 指令会首先完成（哪一条先完成则不确定）。处理器只能以 3 种可能的方式执行它们的代码片段且遵循顺序存储模型：某个处理器的一条 store 指令之后可能是两条以某种次序出现的 load 指令；处理器 1 的两条指令在处理器 2 的任何指令之前执行；或者处理器 2 的两条指令在处理器 1 的任何指令之前执行。结果，至少有一条 load 指令会读取由别的处理器保存的新值。两条 load 可能都会返回新值，但是不可能都返回原来的值，因为后者意味着在 load 执行

之前两条 store 指令都没有到达存储器。这将违反代码的顺序特性。需要刚才所述行为的算法的一个例子就是 Dekker 算法。

13.2　Dekker 算法

Dekker 算法是一种用在硬件中没有原子的读-改-写存储器操作的情况下实现临界段的技术。它只用到了 load 和 store，但是要求顺序存储模型起作用，之所以要这样将在以后讨论。下面几段内容介绍该算法如何在采用了顺序定序（sequential ordering）的系统上起作用。

图 13-2 到图 13-4 中给出的代码用 Dekker 算法而不是 atomic-swap 或者 test-and-set 指令实现了双 CPU 系统上的自旋锁。算法通过单独从每个 CPU 的角度记录锁的状态，并且提供一种方法在两个处理器同时试图获得锁的时候打破僵局，从而消除了竞争条件。在使用原子指令的时候不会出现这样的僵局，因为硬件会解决。图 13-2 中定义的数据结构保留着锁的状态。

```
enum state {UNLOCKED, LOCKED } ;

typedef struct {
    char status[2];      /* 每个 CPU 的状态字节 */
    char turn;           /* 在发生僵局的期间下一步哪个 CPU 先执行 */
} lock_t;

void
initlock( lock_t *lock.)
{
    lock->status[0] = UNLOCKED;
    lock->status[1] = UNLOCKED;
    lock->turn = 0;
}
```

图 13-2　自旋锁的结构以及为 Dekker 算法进行的初始化

每个 CPU 在获得一个自旋锁的时候，必须确保另一个 CPU 当前既没有使用锁，也没有为自己争取获得锁。这是用图 13-3 所示的函数来做到的。此处的函数 cpuid( )返回它正在上面运行的处理器的 CPU 号（对于本例来说，不是 0 就是 1），函数 othercpu( )返回另一个处理器的 CPU 号。

如果锁在当前是打开的，而且只有一个 CPU 试图获得它，那么图 13-3 所示算法的执行相当直截了当。在这种情况下，它把自己的状态设置为 LOCKED，并检查另一个 CPU 的状态。如果另一个 CPU 的状态是 UNLOCKED，则没有发生竞争，那么它就已经成功得到了锁。如果另一个 CPU 现在试图获得锁，那么它将发现第一个 CPU 当前占有了该锁，于是就在里层或者外层的 while 循环中自旋，具体在哪一层取决于变量 turn 的值。不管如何设定 turn，在第一个 CPU 释放锁之前，另一个 CPU 都不能为自己获得锁。

当两个 CPU 同时进入 lock 函数来获得同一个锁的时候，它们两个都会把自己的锁状态设置为锁定状态（locked state）。接着，它们将会注意到对方也正在试图获得锁，于是就使用 turn 字段来判断哪一个实际得到了锁。在竞争中失败的一方将放弃自己获得锁的企图，并且在另一方结束之前进行自旋。这个算法的关键要素是，在另一个处理器的锁状态表明它没有使用锁之前，它决不会返回。

```
void
lock( volatile lock_t *lock )
{
    /* 试图为自己获得锁 */
    lock->status[cpuid()] = LOCKED;
    /* 检查看另一个CPU是否正在使用锁 */
    while( lock->status[othercpu()] == LOCKED ){
        /* 如果没有轮到我，则退避一下，直到另一个CPU离开临界区为止 */
        if( lock->turn != cpuid() ){
            lock->status[cpuid()] = UNLOCKED;
            while( lock->turn == othercpu() )
                ;
            lock->status[cpuid()] = LOCKED; /* 再试 */
        }
    }
}
```

图 13-3 使用 Dekker 算法获得一个自旋锁

使用图 13-4 中的函数释放自旋锁。锁的状态和 turn 值都必须相应地做设置，以使另一个 CPU 能够退出 lock 函数中的双重 while 循环。

```
void
unlock( lock_t *lock )
{
    lock->status[cpuid()] = UNLOCKED;
    lock->turn = othercpu();
}
```

图 13-4 使用 Dekker 算法打开一个自旋锁

13.3 其他存储模型

虽然从程序员的角度来看，顺序存储模型是最自然的存储模型，但是通过改变执行期间某些操作的执行次序就能获得更高的性能。对于存储器操作来说，能够获得在存储器层次结

构中不同层面上的数据就意味着 load 和 store 指令的执行要花不同的时间。举个简单的例子，考虑图 13-5 所示的程序片段，它从存储器中读取两个值，把它们相加，然后把结果存回到存储器。

```
load  %r1,A
load  %r2,B
add  %r3, %r1, %r2      /* A 加 B */
store %r3, C            /* 保存结果 */
```

图 13-5　展示重新确定指令次序的程序片段举例

假定 B 的值被高速缓存了，而 A 没有。因此执行第一条 load 指令要多花一点时间，因为它涉及到完成一次高速缓存缺失和一次主存储器操作。有些处理器没有等待第一个处理器执行完 load，而是在第一个处理器正在等待主存储器返回结果的时候，预取并开始执行第二条 load 指令。因为第二条 load 指令造成了高速缓存命中，所以它可能要比第一条 load 指令先执行完。这种机制的优点在于，第二条 load 指令的开销被第一条的延迟完全隐藏掉了。于是第一条 load 指令一结束，硬件就能执行 add 指令。

多指令执行在现今的处理器中很普通。例如，MIPS R4000 能够在每个时钟周期发出（开始执行）多达 2 条指令，在处理器的八段流水线（eight-stage pipeline）上一次可以执行多达 8 条指令。但是，程序员在编写程序的时候不需要直接了解这些。通过提供连锁（interlock），硬件自动地跟踪每条指令的进展以及指令之间的数据依赖关系。虽然 R4000 不能不按次序执行指令，但是在一般情况下，处理器上重新确定指令次序的连锁不允许图 13-5 中的 add 指令在两条 load 指令完成之前执行。类似地，直到 add 完成之后，才会出现最后一条 store 指令。这就在允许一些指令被重新确定执行次序的同时又保持了程序的顺序特性。

通过进一步放松顺序存储模型所强制规定的严格次序，就能获得额外的性能，但是它可能并不是对软件透明的。首先要注意，一个来自主存储器的 load 指令的任意序列，假定它们都使用独立的寄存器的话，就能够以任何次序来执行。只要硬件的连锁确保了在 load 结束之前寄存器不会被后续的指令使用，那么这就不会对程序有影响。

但是，在执行时刻透明地重新确定 load 执行次序的功能并不适用于 store 指令。例如，任意序列的 store 指令可能包含向同一位置的多次保存操作，比如更新一个循环计数器。顺序程序的语义规定，必须按照程序次序完成 store 指令，否则将在错误的时刻保存错误的值，从而导致后续的 load 指令检索到错误的值。

store 指令的相对次序，甚至是对不同位置的相对次序，是 MP 系统上的内核所要关注的重要问题。再回顾一下图 9-3，可以看到，仅仅在锁里保存一个 0，就会释放锁。因为在释放锁的 store 指令之前的指令是由锁保护的临界区的一部分，所以必须确保在这条 store 指令完成之前已经把结果保存到了存储器中。

为了说明这一点，考虑图 13-6 中的例子，它给出了临界段的最后部分。最后的操作是累加计数器，然后释放保护临界段的自旋锁。要简化这个例子，为简洁起见已经省略了获得锁的代码。同样，也没有给出对 unlock 函数的过程调用，这是为了不让这个例子同采用子例程链接（subroutine linkage）的例子混淆。这就把累加计数器以及释放锁的工作留给了连续的指令流。

```
                .
                .
                .
load        %r1, counter          /* 获得原来的值 */
add         %r2, %r1, 1           /* 累加计数器 */
store       %r2, counter          /* 保存新值 */
clear       %r3
store       %r3, lock         /* 释放自旋锁 */
```

图 13-6 临界区指令流

为了让这段代码能在 MP 上正确操作，保存更新后的计数器值的 store 指令必须在释放锁的 store 指令之前发生。如果硬件要重新确定这两条 store 指令的次序，自旋锁将先行释放，从而让另一个处理器获得锁。那个处理器接着就会在第一个处理器完成保存更新值的 store 指令之前读取计数器原来的值。这就破坏了临界段，因此绝对不允许出现这样的情形。任何时候只要使用了自旋锁、信号量或者任何其他 MP 原语，就需要控制 store 指令的次序。

一定要按照发出指令的顺序来执行 load 和 store 的特殊情形是，当寻址目标是 I/O 和处理器控制寄存器而不是物理存储器的时候。因为程序为了执行 I/O，必须顺序地读状态寄存器和写命令寄存器，所以必须确保硬件没有重新确定这 3 种操作的次序。这种情形的特殊之处将在后面的小节中进行讨论。

考虑到这些问题以后，我们将考察两种新的存储模型。这两种模型是从 SPARC v8（SPARC 体系结构的版本 8）开始使用的，它们也用在了其他一些系统上。它们定义了 load 和 store 指令的相对次序，称为全部 store 定序（TSO）和部分 store 定序（PSO）（SPARC 没有使用严格的顺序定序，它默认使用的是 TSO）。这两种模型的目的在于，利用通过 PSO 获得的潜在的最大速度，实现高速存储系统。正如将要看到的那样，不是为顺序存储模型编写的所有程序都可以移植到使用 PSO 的机器上。所以这种存储模型是一种可选的模式，要用处理器控制寄存器中的一个比特位来启动它。

13.4 TSO

为了提高存储系统的性能，SPARC 体系结构在 CPU 中包括了一种 store 缓冲（store buffer，也叫做 write behind buffer）。它的目的是为 store 指令缓冲数据，使得 CPU 不必等待主存储器（或者甚至是高速缓存）的响应。这就让 CPU 在 store 缓冲处理把数据移入存储器的工作的同时继续执行程序。store 缓冲是和 CPU 配合操作的，它独立于任何现有的高速缓存（为了简单起见，先假定提到的系统都没有高速缓存）。MP 系统的每个 CPU 都有它自己的 store 缓冲。缓冲区的大小与实现有关，但是只有几个字大。例如，TI SuperSPARC 上的 store 缓冲包括 8 个双字（double word）。图 13-7 显示了一个双处理器系统的体系结构。

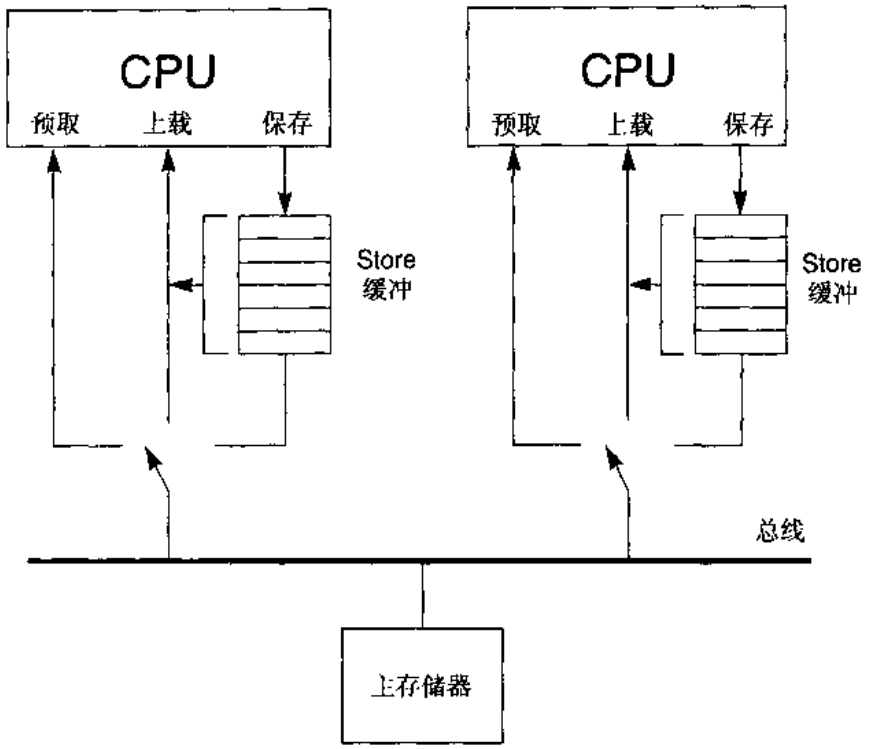


图 13-7　SPARC 的 store 缓冲体系结构

从逻辑上说，每个 CPU 都有用于指令预取、上载和保存的单独连接。尽管如此，每个 CPU 依然只有一条到系统总线的连接。这条连接可以想成是一个开关，切换它就能让一次指令预取、上载或者保存操作访问总线。总线一次只能由一个 CPU 的一次这样的操作使用。

在这种模式下，指令预取总是要去主存储器的。来自 store 指令的地址和数据则被放入到 store 缓冲里，这就保证了以 FIFO 次序把数据发送到主存储器（所有的 store 都要确定次序，所以命名为全部 store 定序，即 TSO）。指令 load 在访问主存储器之前先检查 store 缓冲的内容（就好像它是一小块高速缓存一样）。如果在 store 缓冲中发生一次命中，那么最近向该地址执行的 store 指令所带的数据就被返回给 CPU，注意，在这种情况下没有访问主存储器，它本身就像一块高速缓存那样带来了一些性能上的改善。如果完成上载操作所需要的数据不在 store 缓冲里，那么代之以访问主存储器。一旦要求访问一次主存储器的 load 指令开始执行，CPU 就不再多发出存储器操作。这就在一定程度上保证了顺序性。

SPARC 提供一种 atomic-swap 指令，把对单个位置的上载和保存操作作为一个不可分割的操作。硬件专门处理这些功能。当一条 atomic-swap 指令发出的时候，它就被放入 store 缓冲中，相对于以前发出的 store 指令按 FIFO 次序进行处理，但是它也会屏蔽更多的存储器操作，直到 store 缓冲为空，原子交换操作完成为止。

在对 I/O 或者处理器控制寄存器的 load 和 store 指令时，SPARC 自动恢复到顺序定序。当对特别指定的地址空间（列在体系结构手册中）执行上载和保存操作，或者对通过页表特别标记过的页面执行上载和保存操作时，这些情况就被检测出来。因为这些访问坚持使用熟悉的顺序模式，所以关于它们不需要改动内核。

这种存储模型和顺序存储模型之间的主要区别在于，保存操作不是立即提交给主存储器，而且上载操作也不保证要访问主存储器。注意，对于 UP 系统来说，TSO 的作用并不重要，因为只有一个 store 缓冲要由处理器发出的每次上载操作来检查。这再配以 store 缓冲的 FIFO 性质，就能让 TSO 的行为对这些系统来说是和顺序定序一样的。

但是在一个 MP 系统上，一个处理器保存的数据不会立即由另一个处理器执行的上载操作得到。load 只检查它自己的处理器的 store 缓冲，它不检查其他处理器的 store 缓冲。没有在它自己的 store 缓冲中命中的 load 要访问主存储器，如果另一个处理器的缓冲中有一个挂起的保存操作，那么主存储器保存的可能就是过时的数据。回顾图 13-1 所示的代码，可以看到，它的表现不同于在顺序存储模型上运行时的表现。当两个处理器执行它们各自的保存操作时，两者都被放入到处理器的 store 缓冲里。接下来的上载操作都会访问主存储器，造成两者都获得了 A 和 B 原来的值。这是在采用强定序时绝对不会发生的情况，因此这种新的行为对软件造成的影响必须给予考虑。

总体而言，多个处理器使用 TSO，对存储器中同一个位置同时执行上载和保存操作，结果是不确定的。这对于顺序存储模型也是一样。采用前面章节的上锁结构（locking construct）实现临界段，就可以消除这种不确定性的行为。根据定义，这些临界段一次只允许一个处理器在其中执行，从而防止了对临界资源同时执行上载和保存操作。因此，只要上锁原语仍然能在 TSO 下正确发挥功能，那么临界段将会保持有效，也不需要对内核做进一步修改就能支持这种存储模型。

让我们首先考虑采用 TSO 时自旋锁如何发挥功能。使用 atomic-swap 指令获得一个自旋锁的代码如图 13-8 所示。函数 swap_atomic 把它的第二个参数保存到第一个参数寻址的位置里，并且返回这个位置以前的值。和以前一样，在用作锁的这个字里，值为 1 表示锁被锁住了。因此，在以前的值表明锁尚未被锁住的时候，尝试将用作锁的字设定为 1，就得到了锁。指令的原子性质一次只让一个处理器获得锁。

```
void
lock( volatile lock_t *lock_statue )
{
        while(swap_atomic( lock-status, 1 ) == 1 )
             ;
}
```

图 13-8 使用原子交换操作锁住一个自旋锁

现在假定使用 TSO 的两个处理器同时试图获得当前没有锁住的一个锁。当原子交换操作由 CPU 发出的时候，它就让所有在 store 缓冲里挂起的保存操作完成，并且阻止发出后续的存储器操作。在原子交换操作遇到 store 缓冲 FIFO 的开头之前，处理器以前可能已经对锁字（lock word）做过的任何保存操作都将写入到存储器。此刻，store 缓冲为空，于是处理器的表现就仿佛 store 缓冲不存在一样。和采用顺序定序一样，对锁的争用仍然继续存在。一次只有一个处理器能够获得锁，从而保留了操作的语义。而且可以确保在获得锁之前，不会在指令流中发出访问临界资源的后续指令，因为原子交换操作在它完成之前，屏蔽了更进一步的存储器操作。

接下来，假定一个处理器已经得到了锁，并且已经更新了一个它所保护的数据结构。另一个处理器仍然通过在 lock 函数中的循环里自旋来等待获得锁。当占有锁的处理器释放它的时候，它通过把 0 保存到锁字里来做到这一点。这次保存操作被放入 store 缓冲中，和任何别的保存操作一起按照 FIFO 的次序进行处理。这就保证了对临界资源的所有更新都在更新锁字之前被提交给存储器，因此也满足了前一节所讨论的问题的需要。当释放锁的保存操作遇到 store 缓冲的开头时，它就被写到主存储器，从而让其他处理器能够获得锁。

另一种情况，假定在清除锁字的保存操作尚未通过 store 缓冲提交到存储器之前，刚刚释放锁的处理器就试图再次获得锁。这个处理器执行一次原子交换来重新获得锁，该操作进入 store 缓冲，位于释放锁的保存操作之后。这意味着清除锁的保存操作先进入存储器，原子交换操作跟在后面，它把值从主存储器读回来。这样一来，保证了争夺一个锁的所有处理器都用到了存储器内最新的值，而决不会是它们自己的 store 缓冲内的值。因为锁字在存储器内的值是一致的，所以自旋锁的语义保持有效。让这种做法能起作用的关键因素是，原子交换操作保持了 store 缓冲和主存储器的同步，直到获得了锁以后，后续的上载和保存操作才会发出。

既然自旋锁已经表现出能正确地发挥作用，所以建立在它们之上更高层的原语（如前面章节里给出代码的原语）也将正确地发挥作用。最后，用于内核临界区的算法和数据结构也将正确地发挥作用。

在实现 TSO 的机器上唯一不能正确发挥作用的算法是，其中多个处理器同时执行对相同位置的上载和保存操作，而没有使用某种类型的互斥锁的算法。这可以用 Dekker 算法来说明。

这个算法在 TSO 下不能用。考虑这样的情况，两个处理器同时开始执行图 13-3 所示的 lock 函数，试图获得当前空闲的自旋锁。两个 CPU 都把它们的锁的状态设置为 1，然后彼此检查对方锁的状态。因为前面执行过的、设置锁状态的保存操作还保存在每个处理器的 store 缓冲里，所以两个处理器都从存储器里读到了另一个处理器过时的锁的状态。这一过时信息仍然指示处理器没有占有锁，从而造成两个处理器都认为它们已经成功地得到了锁。注意，这里的 lock 函数以和图 13-1 所示一样的基本代码序列开头：一条 store 指令，向一个将由另一个处理器读取的位置保存数据，后跟一条 load 指令，从由另一个处理器写入的位置读取数据。如前所述，为了让这类算法正确发挥作用，需要顺序存储定序。

需要说明的是，可以对这个算法加以修改，通过以调用 swap_atomic( )替换对锁状态字的赋值语句来让它在 TSO 下运行。这就在允许后续上载操作继续执行之前，迫使 store 缓冲和存储器同步，从而消除了刚才说过的问题。

前面章节里介绍的算法没有一种像 Dekker 算法那样依赖于存储次序。因此，要在 TSO 下支持这些算法，除了使用原子交换操作实现自旋锁之外，不需要对内核做进一步修改。

使用共享存储的用户程序也会受到 TSO 的影响。和内核一样，一个处理器上的一个用户程序对共享存储器所执行的保存操作也不会立即被另一个处理器上运行的用户程序看到。利用共享存储的用户程序必须以某种方式保持同步，以避免竞争条件。这对于顺序存储模型也是如此。只要它们使用的原语在 TSO 下能正确发挥功能，不论它是由内核提供的同步设施，还是共享存储内的 Dekker 算法，那么更高层次上的代码也能发挥功能。

最后要注意指令预取不会检查 store 缓冲是否命中。因此，能自我修改的代码未必能正确地起作用。SPARC 提供一条特殊的 flush 指令，如果需要的话，它能让保存和指令预取保持同步。

13.5 PSO

（部分 store 定序 PSO）使用和图 13-7 一样的体系结构，不同之处在于，并不保证以 FIFO 次序处理 store 缓冲。如果在 store 缓冲里有多条针对相同位置的 store 操作，那么只保证这些保存操作以 FIFO 次序完成，但是针对其他位置的保存和原子交换操作的相对次序则不确定。load 仍然会检查 store 缓冲，看是否有命中，如果有的话，则返回最近向那个位置执行的 store 指令所带的数据。否则，它们就像以前那样去访问存储器。原子交换操作的行为和 TSO 时介绍的一样。

支持单处理机系统的 PSO，不需要改变代码（可能前面介绍的 DMA 操作的情况是个例外）。和 TSO 一样，如果最近向某个位置执行的 store 指令所带的数据还在 store 缓冲里的话，load 操作就返回它；否则，它们就去访问存储器。因此，store 缓冲对于处理器发出的任何 load 操作来说都是透明的。其次，向某个位置执行的 store 指令总是按照 FIFO 的次序完成的，所以结果和采用顺序存储定序的情况一样。

在多处理机系统上，多个处理机几乎肯定会像上一节介绍的那样试图获得同一个锁。原子交换操作阻止了再进一步发出存储器操作，但是在原子交换操作执行的时候，store 缓冲不保证为空。因为 store 缓冲不是按照 FIFO 次序来处理的，所以原子交换操作可能在之前发出的 store 完成之前就结束了。这不会影响到锁所保护的临界段，因为影响它的上载和保存操作都还没有发出呢，而且在得到锁之前，都不会发出。因此，图 13-8 中的代码采用 PSO 也能正确运行。但是，当释放锁的时候，复杂情况就出现了。

非顺序存储模型关注的问题之一是，需要确保临界资源在保护它们的锁被释放之前，在存储器中得到更新。考虑图 13-6 中的代码片段。它执行了两条顺序保存操作：第一条更新计数器的值，第二条释放保护计数器的锁。这两条 store 指令被顺序地放入 store 缓冲，但是在采用 PSO 的情况下，数据到达存储器的次序是不确定的。如前所述，如果和释放自旋锁相关联的保存操作先到存储器，那么另一个处理器就能获得锁，并从存储器中读取到计数器的过时值。

因为绝对不允许这种情况出现，所以 SPARC 体系结构包括了一种称为 store-barrier（汇编程序助记名为 stbar）的指令，强制造成一定程度的顺序性。当 CPU 发出 store-barrier 指令的时候，该指令就被放入 store 缓冲，在此后有任何指令之前，迫使 store-barrier 之前发出的全部 store 指令都先完成。store-barrier 指令将缓冲里的 store 指令分成了若干组，但是不会影响每一组里 store 的次序（因此命名为部分 store 定序）。所以，举个例子，如果在发出 store-barrier 指令之前，在 store 缓冲里有多条 store 指令，那么它们完成的次序仍然是不确定的。对于 store-barrier 指令之后的 store 指令也是如此。如果一个程序有一个 store 指令序列，必须按照发出这些指令的次序来把它们送到存储器，那么就必须在每条 store 指令之后放一条 store-barrier 指令。这显示出了 PSO 带来的利弊：它有获得更好性能的潜力，但是它要改变要求顺序定序的程序才能正确发挥作用。

在向 I/O 和处理器寄存器执行上载和保存操作的特殊情况下，可以像 TSO 那样来使用顺序定序。这样做避免了必须在这类代码中到处显式地写入 store-barrier 指令的要求。

图 13-9 给出了在使用 PSO 时释放自旋锁的正确方式（这是对图 9-3 中代码的一个修改版本）。

```
void
unlock(volatile lock_t *lock_status )
{
        store_barrier();
        *lock_status = 0;
}
```

图 13-9　使用 PSO 打开一个自旋锁

调用 store_barrier 函数（它仅仅执行一条 store-barrier 指令就返回）确保了与该锁所保护的临界资源相关的任何保存操作都会在释放该锁的保存操作之前被写入存储器。

和 TSO 一样，支持 PSO 也不需要改动更高层的 MP 原语，以及它们保护的临界资源。因为其他原语是建立在自旋锁之上的，所以在释放一个自旋锁的时候使用的 store-barrier 也能够使以前所有的 store 都同步。如果内核包括了任何没有使用自旋锁的原语或者算法，比如 Dekker 算法，那么必须显式地采用 store-barrier 指令。例如，通过包括 TSO 所需要的增强特性（用原子交换操作替换针对锁状态域的保存操作），以及在函数 unlock 的开头增加一条 store-barrier 指令，能让图 13-3 和图 13-4 给出的自旋锁实现对 PSO 也起作用。

在现场切换期间，PSO 引入了一个新问题。考虑图 11-9 所示的情况，当释放一个信号量的时候出现了一个竞争条件。通过让从运行队列中选出一个进程的处理器等待，直到进程的状态变为 blocked 为止，就能消除这种竞争。这就确保了在新处理器上尝试恢复进程现场之前，已经在前一个处理器上完整地保存好了该进程的现场。既然 store 缓冲不是 FIFO 的，所以在保存进程现场的 store 指令完成之前，可能出现将进程状态变为 blocked 的 store 指令，这会致使新处理器读入过时的数据。为了防止出现这样的情况，必须在进程状态被变为 blocked 之前，在 store 缓冲内放入 store-barrier 指令。这样一来，在允许另一个处理器开始读取现场之前，保存现场的所有保存操作都已经提交给了存储器。采用 TSO 时，这种情况不是一个问题，因为它总是在最后才执行改变进程状态的保存操作。

在把数据保存到存储器，之后立即有一条针对 I/O 寄存器的 store 指令，开始对那个存储器执行 DMA 操作的时候，还会出现一个类似的问题。虽然 SPARC 保证了对 I/O 位置和物理存储器的上载和保存操作各自都是顺序完成的，但是它没有保证对 I/O 位置和物理存储器的上载和保存操作混合起来还是顺序完成的。所以，就有可能出现这样的情况，当随后发出的开始 DMA 操作的 store 指令被发给 I/O 寄存器时，之前针对存储器的 store 指令相关联的数据却还在 store 缓存中。如果出现这样的情况，那么 DMA 操作将会访问存储器中的过时数据，因为 DMA 操作不检查 store 缓冲。和以前一样，在发起 I/O 的保存操作之前插入一条 store-barrier 指令就能解决这个问题。因为 store 缓冲的规模很小，而设备驱动程序一般为建立 I/O 操作而执行的存储器操作数量相当多，所以在正常情况下不太可能发生这种情况，但

是应该了解和考虑到它。采用 TSO 时也不会出现这个问题，因为对存储器和 I/O 寄存器的保存操作总是顺序完成的。

13.6 作为存储层次结构一部分的 store 缓冲

store 缓冲只是系统存储层次结构中的又一层而已。当系统有高速缓存的时候，store 缓冲就位于 CPU 的寄存器和一级（L1）高速缓存之间。store 缓冲自身的表现就像是一小块高速缓存，因为 load 能够在缓冲中发生命中，更新图 6-7，包括进 store 缓冲，就得到图 13-10 所示的层次结构。

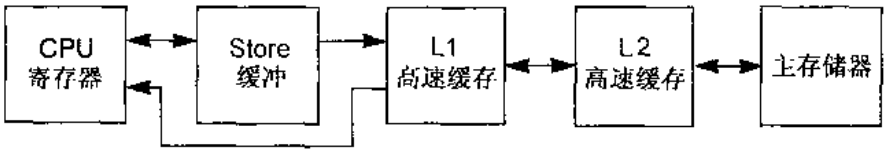


图 13-10 带有 store 缓冲的存储器层次结构

在一个 MP 系统上使用 store 缓冲时出现的问题表现出了 MP 高速缓存技术的普遍问题。因为每个处理机的 store 缓冲是独立运行的，所以当一个处理器的缓冲里尚有挂起的保存操作时，另一个处理机就有可能从存储器或者是它自己的 store 缓冲中读取到过时的数据。就如同 store 缓冲需要改变自旋锁的实现以确保使 store 保持同步那样，要处理 MP 高速缓存技术的普遍情形，挑战在于让所有处理机的高速缓存保持同步，从而使得过失数据不会被访问到。这是本书第三部分的讨论主题。

13.7 小 结

顺序存储模型已经在过去和当前的大多数系统上得到了应用。虽然来自不同处理机的上载和保存操作的次序是不确定的，但是的确能够保证，在一个处理器发出新操作之前，该处理器以前发出的保存操作的执行结果已经提交给了存储器，而且上载操作始终都从存储器读取数据。为了正确地发挥作用，Dekker 算法要依靠这些语义。

为了获得更高的性能，SPARC 体系结构使用了不同的存储模型。一个 store 缓冲被加了进来，使得 CPU 能够在 store 指令被发送给存储器的同时继续执行。store 缓冲是计算机的存储层次结构中的又一个层次。在采用 TSO 的情况下，store 缓冲的内容是以 FIFO 的次序发送给存储器的。在采用 PSO 时，只有针对相同位置的保存操作才以 FIFO 的次序完成，其他保存操作的次序则不确定。无论是这两种模式的哪一种，load 都要在 store 缓冲中检查被请求的地址，并且返回最近向那个地址执行的 store 指令所带的数据。

只要自旋锁能够表现出正确的功能，那么更高层的 MP 原语及其保护的临界区也就能正确地发挥作用。用于锁住一个自旋锁的 atomic-swap 指令，通过让任何以前针对锁字的保存

操作先行完成的做法，迫使 store 缓冲和存储器保持同步。它还阻止了再进一步发出存储器操作。这就确保了所有的 CPU 都将直接访问存储器内的锁字，而不是在它们的 store 缓冲中可能已经过时的副本。它还在获得锁之前，防止来自临界段的上载和保存操作被发出。

采用 PSO 的情况下释放一个自旋锁需要包括一条 store-barrier 指令，这样做迫使以前发出的保存操作在锁本身被释放之前到达存储器。在现场切换期间也需要一条 store-barrier 指令，以确保进程被选出来在另一个处理器上执行之前，它的状态已经完全保存起来了。采用 TSO 时不需要 store-barrier 指令。

13.8 习　　题

13.1　在有任意数量的 CPU 的情况下，实现 Dekker 算法。

13.2　如果改变了 SPARC 的 TSO 存储结构，使得 load 不会检查 store 缓冲是否有命中（也就是说，load 总是直接就到存储器了），那么会对软件产生什么样的影响？软件必须怎样做来进行弥补？

13.3　从 PSO 而不是 TSO 的角度考虑上面的问题，有区别吗？

13.4　如果 SPARC 的 PSO 实现没有保证对相同地址的 store 采用 FIFO 定序，那么将要求软件怎样做？

13.5　如果 SPARC 在 atomic-swap 指令发出之后没有停止再进一步发出存储器操作，那么会发生什么样的情况？

13.6　考虑对 TSO 模型进行修改，store 缓冲像采用物理高速缓存时总线监视所做的那样，监视总线上其他处理器的保存操作。如果被寻址的数据在 store 缓冲里，它就返回最近针对那个位置执行的 store 指令所带的数据。在这种情况下，其他处理器不访问主存储器。从软件的角度描述这会带来的影响。实现它是个好主意吗？

13.7　采用 PSO，重做上面的问题。

13.8　对于 MP 内核来说，当它正在访问一个进程的私有数据时，store 缓冲的存在会是个问题吗？

13.9　在 10.3 节中，有对不需要在 MP 系统上有显式上锁机制情况的讨论。考虑那一节里最后研究的情形，其中有一个一段共享内核数据的写方。根据那里的讨论，不需要锁，因为它对于本来就有的竞争条件无所作为。当硬件使用 TSO 或者 PSO 的时候，依然如此吗？解释原因。

13.9 进一步的读物

[1]　*The SPARC Architecture Manual, Version 8*, SPARC International, 1991.

[2]　Adve, S., and Hill, M., "Implementing Sequential Consistency in Cache-Based Systems," *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990, pp. I:47-50.

[3] Adve, S., and Hill, M., "Weak Ordering-A New Definition," *Proceedings of the 17th Annual Internations Symposium on Computer Architecture*, May 1990, pp.2-14.

[4] Adve, S., Hill, M.D., Miller, B.P., and Netzer, R.H.B., "Detecting Data Races on Weak Memory Systems," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991, pp. 234-43.

[5] Catanzaro, B., *Multiprocessor System Architectures*, Sun Soft Press, 1994.

[6] Dijkstra, E.W., "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM*, Vol. 8, No.5, September 1965, P. 569.

[7] Dijkstra, E.W., "Cooperation Sequential Processes," in *Programming Languages*, F.Genuys (ed.), New York: Academic Press, 1968, PP.43-112.

[8] Dubois, M., Scheurich, C., and Briggs, F., "Memory Access Buffering in Multiprocessors," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 1986, pp. 660-73.

[9] Dubois, M., Scheurich, C., and Briggs, F., "Synchonization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, Vol. 21, No. 2, February 1988, PP.9-21.

[10] Dubois, M., and Scheurich, C., "Memory Access Dependencies in Shared-Memory Multiprocessors," *IEEE Transactions on Software Engineering*, Vol. 16, No.6, June 1990, pp. 660-73.

[11] Eisenber, M.A., and McGuire, M.R., "Further Comment on Dijkstra's Concurrent Programming Control Problem," *Communications of the ACM*, Vol. 15, No. 11, November 1972, P. 999.

[12] Gharachorloo, K., Lenoski, J., Gibbons, P., Gupta, A., and Hennessy, J., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990, pp.15-26.

[13] Gharachorloo, K., Gupta, A., and Hennessy, J., "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp.245-57.

[14] Knuth, D.,"Additional Comments on a Problem in Concurrent Programming Control," Communications of the ACM, Vol. 9, No. 5, May 1966, pp.321-2.

[15] Mosberger, D., "Memory Consistency Models," *ACM SIGOPS Operating Systems Review*, Vol. 27, No. 1,January 1993, pp.18-26.

[16] Peterson, G.L., "Myths about the Mutual Exclusion Problem," *Information Processing Letters*, Vol. 12, No.3, June 1981, pp.115-6.

[17] Scheurich C., "Access Ordering and Coherence in Shared Memory Multiprocessors," Ph.D. thesis, University of Southern California, May 1989.

[18] Torrellas, J., and Hennessy, J.,"Estimating the Performance Advantages of Relaxing Consistency in a Shared Memory Multiprocessor," *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990, pp. I:26-33.

[19]　Zucker, R.N., and Baer, J.-L., "A Performance Study of Memory Consistency Models," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992, pp.2-12.

第三部分

带有高速缓存的多处理机系统

第14章 MP高速缓存一致性概述

本章介绍在多处理机系统中使用高速缓存。先介绍SMP系统常用的组织结构，接着讨论MP高速缓存机制给操作系统带来的问题，然后我们展示如何使用软件技术来解决这些问题。解决MP高速缓存问题的硬件技术则在第15章里介绍。

14.1 引 言

正如我们在第一部分看到的那样，高速缓存通过利用局部引用特性来提供一条减少存储器平均存取时间的途径。当然，多处理机也非常需要有这样的好处，因此成了给这些系统增加高速缓存的动机。带有高速缓存的典型SMP系统如图14-1所示。

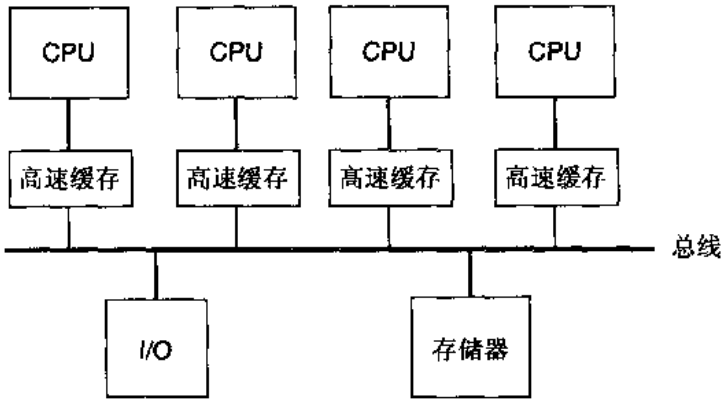


图14-1 带有高速缓存的SMP系统

在这种组织结构中，每个CPU都有一块私有的高速缓存，这是在SMP系统中最常用的方法。虽然CPU还是有可能使用某种形式的共享高速缓存，比如让两个或者两个以上的CPU连接到一块公共的高速缓存上，或者在总线和主存储器之间放一块高速缓存，但是使用私有的高速缓存具有几个优点。首先，将高速缓存直接连接到CPU使得在高速缓存中命中时存取时延最小。这免去了如果高速缓存处于靠近主存储器单元的位置时需要的一次总线操作。既然往往90%或更高的处理器引用都是在高速缓存中命中的，那么这样

做就等同于节省了总线交易量（正如将要看到的那样，维护一致性需要一些额外的总线交易，这稍稍降低了实际的节约量）。减少总线操作反过来又降低了对总线和存储器带宽的需要量，从而有可能比没有高速缓存时，或者比使用和主存储器相关联的共享高速缓存时，使系统支持更多的处理器。减少总线操作也有助于减少总线争用，从而能迅速存取主存储器。最后，任何类型的共享高速缓存都会在访问它的CPU之间造成一定的争用。访问一块私有高速缓存则不会有这样的争用发生。结果，带有高速缓存的MP系统能够比不带高速缓存的MP系统性能更好。

私有高速缓存的组织结构已经为现代微处理器的设计人员所倡导，因为这些系统中的大多数都包含有片上的私有高速缓存。MIPS R4000、Intel 80486和Pentium以及TI SPARC处理器都是例子。因为大多数商业的SMP系统都是使用这些微处理器制造的，所以本书剩下来的内容就着眼于私有高速缓存的设计。在Silicon Graphics、Sequent Computer Systems、DEC、IBM、Pyramid Technology、Sun Microsystems等公司的系统中都已经采用了私有高速缓存的设计。

高速缓存本身可以遵循第一部分介绍的4种主要组织结构中的任何一种，但是，正如将会在下一章里看到的那样，设计人员一般喜欢物理高速缓存，或者是能物理索引的虚拟高速缓存（如Intel i860 XP的片上高速缓存）。每个CPU也可能有独立的指令和数据高速缓存，或者是一种层次结构的高速缓存，但是会认为连接到每个CPU的高速缓存是一样的。虽然可以设计一个系统，其中每个CPU都带有不同组成的高速缓存，但是实际的SMP系统通常都是采用相同的CPU/高速缓存模块来构建的。高速缓存组织结构相同也简化了操作系统的复杂性，因为它的设计只需要处理一种高速缓存的类型。

注意，有了高速缓存并不会改变8.2节里介绍的SMP体系结构的主要性质。这样的系统仍然是紧密耦合的，使用了一个能够全局访问的共享存储器单元，而且仍然表现出了对那个单元存取的对称性。但是，正如行将看到的那样，增加私有高速缓存会影响到系统的存储模型。可以采用硬件或者软件技术（或者二者的结合）来恢复成一种熟悉的存储模型，如顺序存储模型。

在第一部分里已经看到，MP的操作系统必须解决好系统完整性、性能以及外部编程模型等领域内的问题。对于带有高速缓存的MP系统上的内核来说，也是如此。就好比必须对操作系统加以修改以处理多处理机一样，也必须对它进行修改，使之能够在有多块高速缓存存在的情况下正确地执行。根据高速缓存硬件设计的不同，这可能需要比第一部分介绍的高速缓存基本维护任务更多的工作，这些附加的工作也可能会影响到系统的性能。最后还认定，高速缓存的出现不会影响到外部编程模型。考虑到本书的目的，在第二部分中描述的系统上正确运行的任何用户程序无需修改就能在这里描述的系统上正确执行。和第一部分里的一样，高速缓存对于用户程序来说是不可见的。

因此，管理带有高速缓存的一个MP系统，其挑战性涵盖了维护高速缓存（第一部分介绍）一致性的所有方面，以及互斥和同步（第二部分介绍）的所有领域。现在完成这些工作都必须要保证它们在多个高速缓存之上是同步的。

14.2　高速缓存一致性问题

在第一部分中，如果软件不可能访问到过时的数据，那么就称高速缓存同主存储器是一致的。例如，通过适时冲洗高速缓存的软件技术，或者通过利用具有总线监视功能的物理高速缓存的硬件，就能做到这一点。在 MP 系统上，多个 CPU 可能要同时存取共享的数据。对于支持顺序存储模型的 MP 系统来说，如果从任何处理器对任何共享存储位置的读取总是返回那个位置最近写入的值，那么就称该系统的高速缓存是一致的（在 14.3.3 小节中讨论其他存储模型）。如前所述（在第一部分里），术语“高速缓存一致性（cache consistency）”和“高速缓存一致性（cache coherency）”可以互换使用。

用下面的例子可以引出在 MP 系统上维护高速缓存一致性的问题。假定一个双 CPU 系统使用了不带总线监视功能的写回/写分配物理高速缓存，或者能够维护高速缓存一致性的任何特殊硬件。假定两种高速缓存初始为空（也就是说，所有的高速缓存行都标记为无效）。考虑一个这样的进程，它睡眠一秒钟，累加计数器一次，然后再睡眠，周而复始。假定这是系统中唯一正在进行的活动，所以没有别的什么会干扰高速缓存的内容。如果计数器初始为 0，那么在进程第一次运行之前，高速缓存和存储器的状态如图 14-2 所示。

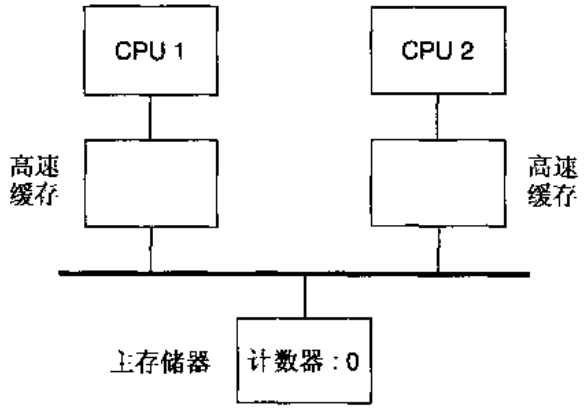


图 14-2　存储器和高速缓存的初始状态

如果进程运行在 CPU1 上，当它试图累加计数器的时候，不会在高速缓存中命中。这将让它从存储器里读取到 0，将这个值加 1 再写回到高速缓存中。因为是物理高速缓存，所以在现场切换的时候不会冲洗它们，于是，当进程再度睡眠的时候，存储器的状态如图 14-3 所示。

如果进程下一次醒来后在 CPU1 上运行，那么它在高速缓存中可以命中计数器变量，并且正确地将其值更新为 2。但是，如果它是在 CPU2 上运行的，那么它在 CPU2 的高速缓存中不会命中计数器变量，并且从主存储器中读取到了过时的值。之所以出现这样的情况，是因为没有采取任何措施来保持两块高速缓存的一致性。绝对不应该允许像这样的不一致性出现。

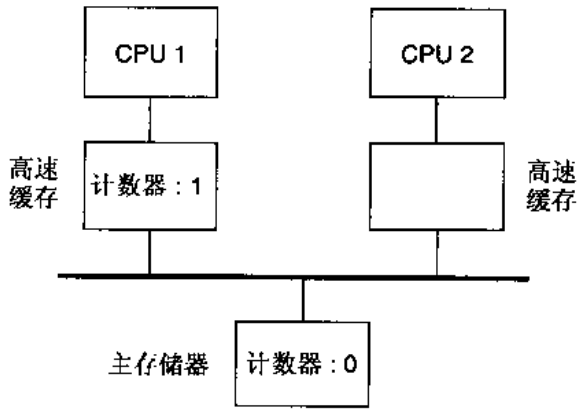


图 14-3 在第一次累加之后存储器的状态

乍一看，似乎改用写直通高速缓存机制可能会解决这个问题，因为那样一来，存储器始终会更新。但是，这不足以完全解决一致性问题。如果在 CPU1 最初把计数器增加到 1 的时候就已经采用了写直通方式，那么主存储器和 CPU1 上的高速缓存都包含有计数器的值 1。如果接下来进程在 CPU2 上运行，那么当它没有在高速缓存中命中的时候，实际上它是从主存储器里读取到正确值的。它也会把更新后的值 2 写回到主存储器，于是存储器里的值也是正确的。但是，如果进程现在又在 CPU1 上运行，那么它会在计数器的过时值上产生一次高速缓存命中。CPU1 仍然高速缓存着值 1，因为从 CPU2 到存储器的写直通操作对 CPU1 的高速缓存没有影响。

举第二个例子，考虑这样的情况，两个处理器同时试图访问并修改共享数据，就像 13.2 节中的 Dekker 算法那样。再假定采用了写直通高速缓存机制，高速缓存初始为空。如果两个处理器同时进入 lock 函数，获得了同一个锁，那么 Dekker 算法就不能正常起作用。在这种情况下，每个 CPU 都把它的锁状态设为 LOCKED；但是，使用写回高速缓存机制则意味着另一个 CPU 不能立即看到这些保存操作的结果。它们都从存储器中取得另一个 CPU 锁状态的过时值，并且认为自旋锁是空闲的。于是，两个处理器都以为它们已经得到了自旋锁。注意，写回高速缓存的出现和上一章里 SPARC store 缓冲的出现非常类似。高速缓存改变了 MP 系统的存储模型，使之类似于 PSO。但是有一点不同，store 缓冲的内容保证会在不确定的一段时间内被写入到主存储器。采用写回高速缓存的情况下，保存在高速缓存中的数据会一直留在那里，直到将来的一次高速缓存缺失期间显式地冲洗或者替换它为止。因此，在一段不确定长度的时间里，主存储器的内容一直是过时的。

和以前一样，使用写直通高速缓存机制不会完全解决采用 Dekker 算法时的一致性问题。如果第二个 CPU 试图获得由第一个 CPU 占有的锁，那么它将从主存储器中读取到另一个处理器正确的锁状态，看到那个锁正在用，于是它要么在另一个 CPU 的锁状态上自旋，要么在锁的 turn 字段上自旋。当这个锁被释放的时候，如果使用了写直通高速缓存机制，那么存储器中的这些字段便会被更新。但是试图获得锁的 CPU 已经在高速缓存中缓存了这些字段的值，它将继续在这些过时的值上自旋下去。因为写入主存储器的操作不会影响到那个 CPU 的高速缓存中的内容，所以在有什么东西让过时的高速缓存行变得无效或者被替换掉了，比如发生了一次中断，中断处理程序引用到了索引到相同高速缓存行的存储器之前，

它一直都在自旋。如果没有什么让过时的高速缓存行变得无效，那么处理器将会永远自旋下去。

接下来，考虑当一个处理器试图重新获得它以前释放的锁时会发生什么情况。假定在处理器释放了锁之后，另一个 CPU 得到了这个锁，并且仍然占据着它。当调用 lock 重新获得锁的时候，它将 CPU 的锁状态字段设为 LOCKED 状态，并且检查另一个 CPU 的状态。因为在前面它获得锁的时候，它已经检查过这个状态，所以到处理器重新获得锁的时候，另一个 CPU 的状态可能仍然缓存在高速缓存中。如果仍然缓存着，那么缓存的状态肯定过时了，因为另一个处理器在获得锁的时候，已经把它的状态设定为 LOCKED 了。重新获得锁的处理器将会看到这个错误的、过时的数据，于是认为锁还是空闲的。这违反了锁的语义，必须予以避免。

打开自旋锁也会有高速缓存一致性问题。如果采用了写回高速缓存机制，释放锁的保存操作没有被立即送到主存储器，从而造成另一个试图获得锁的处理器使用了过时的值。同样，即使采用了写直通高速缓存机制，另一个处理器以前已经在高速缓存中缓存了这些字段的值，对主存储器的写操作现在已经让它们过时了。于是在锁释放以后，那个处理器会以为锁仍然被占用着。

在任何时候，只要有一个以上的 CPU 存取和修改数据，那么这些基本的高速缓存一致性问题就会以各种方式显现出来。注意，这个说法既包括用户数据，也包括内核数据。为了解决这个问题，需要一种技术来保持高速缓存内容的一致性。这能以软件或者硬件的方式做到。软件解决方案包括适时冲洗高速缓存，以防止过时的数据被访问到。例如，通过在 CPU 每次做现场切换的时候将高速缓存的内容写回主存储器（如果使用了写回高速缓存机制的话），并使高速缓存无效，就能修正我们这里研究过的第一个例子。这会让进程每次醒来的时候发生高速缓存缺失，从而让它从主存储器中读取计数器的当前值。遗憾的是，这样做丧失了物理高速缓存的主要优点之一，即免去了现场切换时的冲洗时间。正因为如此，人们已经发明出硬件技术来自动地保持高速缓存和主存储器的一致性。扩展总线监视的概念，使得高速缓存不仅监视总线上的 I/O DMA 交易，而且也监视其他高速缓存的活动，就能达到上述目的。

为了全面理解 MP 高速缓存一致性的复杂性，下面几小节将首先介绍软件上的高速缓存一致性技术。因为纯软件技术的复杂性和糟糕的性能，所以在实际中很少采用，但是它们清楚地演示了必须要解决的问题。随后的讨论适合于所有类型的高速缓存，无论它们是虚拟或者物理索引的，还是虚拟或者物理标记的。硬件高速缓存一致性在下一章里介绍。

14.3　软件高速缓存一致性

当通过软件技术来维护高速缓存一致性的时候，要注意的第一种情况是，对于只读数据来说，不需要特殊对待。因为这类数据决不会改变，被高速缓存的任何副本都将与主存储器中的副本相吻合，因此默认就是一致的。特别地，正常的程序正文（指令）和内核正文都属于这类数据（当然不包括能自我修改的代码）。于是，在上一节中程序累加计数器的例子里，为了保持一致性，在现场切换的时候，只有进程的数据需要从高速缓存中冲洗掉。如果系统

使用独立的指令和数据高速缓存，那么可以不管指令高速缓存。

但是，要注意，在进程退出的时候，不能免去内核执行冲洗指令高速缓存的任务。在采用物理高速缓存的情况下，第 6 章中介绍的所有管理措施都必须执行。但是，现在必须对系统中所有的高速缓存都执行一遍。例如，当上一个例子中的进程退出时，留在两个处理器的高速缓存中的指令必须在重用其对应的物理页面之前冲洗掉。在硬件没有保持高速缓存一致性功能的系统上，只能从高速缓存连接的 CPU 冲洗它。所以，如果进程在 CPU1 上退出，那个 CPU 必须通知 CPU2 冲洗它的高速缓存。根据硬件的不同，这可以有多种方式来实现。例如，有些系统支持 CPU 间的中断（inter-CPU interrupt），这种机制允许一个 CPU 向另一个 CPU 发送中断。使用这种机制时，可以向主存储器写入一则消息，指出要采取什么样的措施。也可以有各种优化措施。例如，如果进程从来不在 CPU2 上运行，那么在进程退出的时候，不需要那个 CPU 冲洗它的高速缓存。因此，内核能够跟踪每个进程运行在哪个处理器上，以减少需要其他处理器完成的高速缓存冲洗量。

另一种可能是使用 7.4 节介绍的滞后的高速缓存无效操作。如果采用了物理高速缓存，那么使高速缓存无效操作能够被推迟到将脏列表（dirty list）的内容移入清洁列表（clean list）时再进行。此刻，所有的 CPU 都得到通知，冲洗它们的高速缓存，旋即清理干净了高速缓存中的全部过时项。

当多个处理器存取和修改共享数据时，会出现 MP 高速缓存一致性的主要问题。接下来的两小节介绍解决这些问题所必需的附加管理措施。处理共享数据有两种主要的方案：使用无高速缓存的操作，以及有选择性地冲洗共享数据。这些技术都可以运用到第一部分介绍的 4 种高速缓存结构上。

14.3.1 共享数据不被高速缓存

在这种方法中，MP 一致性问题首先是通过绝不允许能够为一个以上 CPU 修改的数据被高速缓存来消除的。在 ELXSI SYSTEM 6400（一种在 20 世纪 80 年代中期的系统，见参考文献[20]）上就使用了这项技术。为了实现这个目的，高速缓存必须支持一种无高速缓存的模式（uncached mode）。正如在第一部分里提到的那样，大多数体系结构都允许以页面为单位指定这项要求。因此，必须把处理器之间共享和修改的数据组织到和只读数据分开的页面中。如果使用了层次结构的高速缓存，那么它们都必须把这些共享数据当作无高速缓存的数据来对待。

除了只读数据和共享数据之外，在 MP 系统中还有第三类数据：处理器的私有数据。这是一次只能由一个 CPU 存取和修改的数据。如果像下面这样适时地进行冲洗，那么它也可以被高速缓存。和用户进程相关的正文、数据和堆栈就是处理器私有数据的一个例子。在传统的 UNIX 系统中，进程里只有一条控制线程，因此进程一次只能在一个处理器上执行。除了在调试进程期间之外，系统中的其他处理器都不能访问它的地址空间，所以当它正在运行的时候，可以安全地高速缓存它的地址空间（调试进程的情况和多线程进程的情况一样，以后介绍）。如果执行一次现场切换，并且进程在不同的处理器上运行，那么它以前运行的处理器的高速缓存必须先行冲洗（使主存储器有效，而使高速缓存无效），以此按 14.2 节指出的那样保持一致性。以这种方式将一个进程从一个处理器移到另一个处理器的动作称为进程迁

移（process migration）。

对于使用虚拟高速缓存的系统来说，每次现场切换期间，无论如何都要冲洗高速缓存，所以在进程迁移的情况下就不需要额外的冲洗。但是，对于其他高速缓存组织结构来说，必须按照刚才介绍的那样进行冲洗。注意，这样做抵消了这些高速缓存的主要优点之一：避免在现场切换时刻的冲洗操作。通过把一个进程和一个特殊的处理器绑定起来，意味着在每次现场切换时不允许进程迁移，就能够部分地挽回这一优势。在现场切换期间，只有上次运行这个进程的处理器才能选择它。于是这就解决了一致性问题（以这种方式绑定进程也称为处理器绑定（processor affinity））。如果所有的进程都永久性地被绑定到单个处理器上，那么负载在系统中所有的处理器上极有可能是不均衡的。因此，把进程绑定到处理器上的系统也必须定期地解除绑定，把负载重新均匀分布到各个处理器上。当完成类似这样的负载均衡迁移（load-balancing migration）时，必须冲洗高速缓存，但是这样的冲洗发生的频率要比每次现场切换都冲洗要低。注意，在采用主从处理机内核实现时，除了很短的一段时间之外，很难做到处理器绑定，因为为了执行系统调用，进程必须迁移到主处理器上。因此，处理器绑定更适合于多线程内核。

如果系统允许在一个地址空间中有多个线程，那么就不能允许这些线程同时在多个处理器上执行，因为共享的地址空间不会保持一致。从高速缓存一致性的角度来看，调试进程的情况非常像线程，因为另一个进程获得许可，可以访问其他进程的地址空间（通常是通过特殊的系统调用）。当单独使用软件上的高速缓存一致性技术时，在这些情况下，没有高效的解决方案。一种方法是，使得进程的整个地址空间都不被高速缓存。于是，任何处理器都可以在任何时刻执行任何线程，而不会有一致性问题，代价是为了无高速缓存操作而造成的性能损失。类似地，可以让一个进程的地址空间在调试器开始访问它的地方不被高速缓存。另一种方法是，将一个进程内的所有线程绑定到一个处理器上（类似地，强行让调试器和它正在调试的进程同在一个处理器上）。于是，地址空间可以被高速缓存，但是一次只能执行一个线程。在这种情况下，所有的线程都随着进程一起迁移。

使用共享存储器或者映射文件的两个进程必须像对待线程那样进行处理。如果共享存储器被高速缓存了，那么为了保持一致性，一定不允许这样的进程同时在不同的处理器上运行。和线程一样，将这些进程绑定到一个处理器上，就能解决这个问题。另一种方法是，可以使得共享存储页面不被高速缓存，从而允许共享它们的进程在任何地方执行，并且自由地迁移，但是当访问那些页面的时候，性能会降低。

另一种处理器私有数据是与每个进程相关联的内核数据结构。在典型情况下，这些数据包括内核栈和 u 区。因为其他进程从来都不会访问这些数据，所以它们可以被高速缓存，只需要在迁移进程的时候被冲洗掉。

虽然任何处理器的私有数据都能被高速缓存，但是共享的内核数据结构必须不予缓存。一定不被高速缓存的特殊数据结构取决于内核实现。例如，如果采用了主从处理机 MP 内核，那么处理器之间共享的数据结构只有运行队列和保护它们的锁。所有其他的内核数据结构都只有主处理器才能访问，所以这些数据结构都能被高速缓存。

多线程内核则不同。在这里，任何处理器都能执行任何部分的内核代码，因而可以访问任何共享的内核数据结构。在这种情形下，所有这样的数据结构都必须不予缓存。例如，每个进程的进程表项包含了可能被其他进程访问的数据。因此，这是一种共享的数据结构，不

能被高速缓存。注意，用于实现 MP 原语的所有数据结构，如自旋锁和信号量，也必须不被高速缓存，否则就会出现采用 Dekker 算法时发生的不一致性。

采用不予高速缓存的方法，就能正确地保持一致性，从而保证了操作系统的完整性，并且保持了外部编程模型与单处理机系统的兼容性。这种方法实现起来也相对简单。困难之处在于，在许多情况下，它不能利用处理器上的高速缓存，从而降低了高速缓存给性能带来的好处。因此，我们必须找到一条让更多数据能被高速缓存的途径。

14.3.2 有选择性地冲洗高速缓存

有选择性地冲洗高速缓存是一种允许共享的内核数据在处理器正在使用它的时候被高速缓存的方法。在出现不一致性之前，让内核有选择性地从一个处理器的高速缓存中冲洗掉共享数据，从而保持了一致性。利用所有共享数据都是由某个锁来保护的这一事实，就能使共享数据被高速缓存。一旦某个处理器得到了那个锁，就可以高速缓存相关的内核数据，因为这与其他处理器保证是互斥的。这样一来，该数据结构暂时就变成了处理器的私有数据。但是，当释放了锁以后，另一个处理器就能访问共享数据了；因此，释放锁的处理器必须显式地从它的高速缓存里冲洗掉数据（使主存储器有效，而使高速缓存无效）。这应该正好在释放锁之前完成。当下一个处理器获得锁的时候，主存储器已得到了更新。因为每个处理器在释放锁的时候使它的高速缓存无效，所以如果它随后又再次获得了同一个锁，并且访问共享数据，那么肯定会发生高速缓存缺失。这就确保了处理器能访问到主存储器中更新过的数据副本，而不是使用以前的访问所对应的过时数据。和以前一样，只读数据和程序正文可以被高速缓存，而无需冲洗。如果系统使用了层次结构的高速缓存，那么处理器释放锁之前，必须将临界资源从处理器上所有的高速缓存中冲洗掉。处理器私有数据（包括用户程序正文、数据和堆栈）则按照上一节那样进行处理。

为了保持一致性，这种方法要依靠锁来提供互斥机制。因为处理器在尝试获得一个自旋锁的时候，它们之间没有互斥，所以自旋锁本身的数据结构从来都不能被高速缓存。这就防止了发生 14.2 节里所讲述的不一致性。接下来，用于实现更高层次原语的数据结构，如信号量、睡眠锁、事件计数等等，与共享的内核数据结构本身没有什么区别。这意味着它们可以被高速缓存，但是必须在处理器释放自旋锁之前从高速缓存中冲洗掉它们。另一种方法是，可以让它们不被高速缓存。

有选择性的高速缓存冲洗方法的另一个要求是，每个临界资源必须占有一个独立的高速缓存行。如果两个或者两个以上的临界资源要占据同一个高速缓存行，那么就会有处理器可能访问到过时数据的风险。举个例子，考虑存储器中的一个计数器组，其中每个计数器都由一个独立的自旋锁保护。因此，每个计数器都是一个单独的临界资源。假定计数器为 4 字节，而高速缓存行为 8 字节，这意味着一对计数器将占据一个高速缓存行。如果 CPU1 获得了用于组内第一个计数器的锁，那么将出现一次高速缓存缺失，致使既包含第一个计数器也包含第二个计数器的整个高速缓存行被载入到 CPU 的高速缓存中。如果计数器初始为 0，那么在累加之后，高速缓存的内容以及受影响的高速缓存在主存储器里的内容如图 14-4 所示。注意，高速缓存使用写回机制（第一个计数器出现在高速缓存行的左边）。

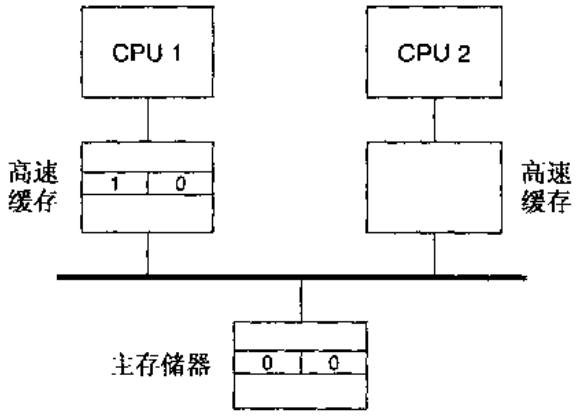


图 14-4　临界资源共享一个高速缓存行

不一致性还没有出现，只要两个 CPU 从不同时访问同在一个高速缓存行内的两个计数器，那么就不会出现不一致性。但是，如果当 CPU1 正在访问第一个计数器的时候，CPU2 获得了第二个计数器的锁，并且累加那个计数器，那么结果如图 14-5 所示。

此刻，每个 CPU 都正好高速缓存着另一个 CPU 正在使用的计数器的过时数据。当 CPU 把它们正在使用的临界资源写回存储器并使这些资源无效时，它们也就把同另一个临界资源相关的过时数据写回到了存储器。要记住，在每个高速缓存行（或者子行）中只有一个修改位，所以一次写回操作把整个高速缓存行（或者子行）写回到主存储器中。根据 CPU 执行写回操作的顺序不同，不是这个计数器就是另一个计数器在主存储器中的值是错误的。观察到这个问题和 3.3.7 小节中描述的问题相类似，在那一小节里，如果其他数据共享着同一个高速缓存行，那么对位于共享存储区内的原始 I/O 缓冲执行 DMA 操作就会造成不一致性。在这两种情况下，如果总线上另一个单元的动作改变了存储器，而又没有更新处理器的私有高速缓存的话，就会造成不一致性。

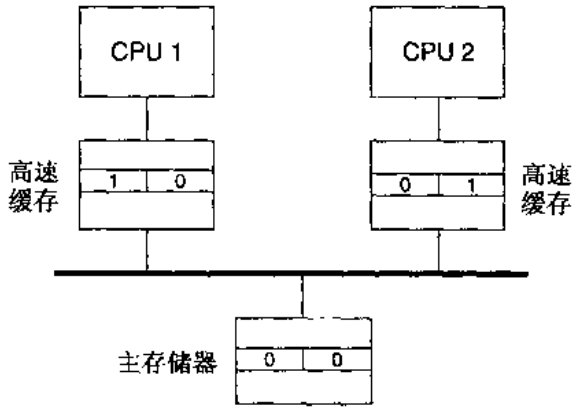


图 14-5　两个 CPU 都访问同一个高速缓存中的临界资源

如果一个处理器试图同时使用一个高速缓存行内的两个临界资源，就会出现另一种类型的不一致性。开始的高速缓存和存储器状态如图 14-5 所示，同时，两个 CPU 都占据着它们各自的锁，假定 CPU1 试图获得计数器组内第二个计数器的自旋锁（CPU2 当前正占据的一

个自旋锁）。当 CPU2 释放锁的时候，它已经用新的计数器值更新过主存储器。CPU1 现在能够获得这个锁，但是因为当它访问第一个计数器时已经缓存了第二个计数器的值，所以它将使用它的高速缓存里的过时值，而不是主存储器里的更新值。正因为有这些不一致的问题，所以必须保证每个临界资源位于独立的高速缓存行。在有这样的高速缓存的情况下，即高速缓存行内每个子行都有单独的有效位和修改位，临界资源就只需要占据单独的子行。

要牢记，占有长期互斥锁的进程可能在占据锁的同时在睡眠。如果进程被迁移到另一个处理器，同时还占据着锁，那么在该进程能在新处理器上执行之前，它所访问的任何用户数据或者共享数据都必须被写回主存储器，并且使之在原处理器的高速缓存中无效。这样做的理由见 14.2 节中的说明。

注意，这项技术不能用来允许用户共享的存储区被高速缓存，以及允许不同处理器上的多个进程同时访问用户共享的存储区。单处理机的外部编程模型不要求在用户程序中显式地编写代码执行任何高速缓存冲洗操作，这项技术会强制这样做。因此，应该继续按照上一节介绍的那样处理用户共享存储区，也就是说不予缓存。

观察到这种方法同为了在 SPARC 体系结构上支持 PSO 所采取的方法（参见 13.5 节）相似。它过于依赖于内核的互斥锁，当一个处理器的 store 缓冲中有一个针对临界资源的保存操作时，防止其他进程访问存储器内的过时数据。在那种情况下，必须确保在释放锁的保存操作之前，将这些保存操作发送到主存储器。通过插入一条 store-barrier 指令，就能做到这一点，这条指令实质上是在锁被释放之前冲洗掉了 store 缓冲。这个做法实现起来很简单，因为不必准确地知道访问的是临界区内的哪一个数据结构。对于有选择性的高速缓存冲洗来说，这并非易事。

实现这项技术的复杂性随内核多线程化的程度不同而不同。对于主从处理机实现来说，很容易增加额外的高速缓存冲洗操作。例如，图 9-9 所示的 enqueue 函数只需要把以下两条数据写回存储器并使之无效：p->p_next 域和 q->q_head 域。这必须正好在调用 unlock 函数之前完成。图 9-10 所示的函数 dispatch 更复杂一些，因为它遍历了整个链表。在这种情况下，它能在释放自旋锁之前把高速缓存的内容写回存储器，并使之无效。这就不必在检查每个队列元素之后还要冲洗它。

细粒度多线程内核处理起来要更复杂，因为必须在释放锁之前修改每个临界区，把相关的临界资源从高速缓存中冲洗掉。还必须把每个临界资源保存到存储器中，这样一来，它就只占唯一一个高速缓存行了。在许多情况下，这需要做一次详细的分析，以判定临界区内的哪些数据访问过了，从而正确地进行冲洗。当然，通过在每次释放锁的时候简单地冲洗掉整个高速缓存就可以避开分析，但是这样做的开销，特别是采用大规模的高速缓存时的开销，会让前面使用的不高速缓存访问的技术更有吸引力。人们也提出过让编译器分析代码所做的数据引用，自动地插入所需高速缓存冲洗操作的技术。这些技术主要用于在缺乏硬件高速缓存一致性的系统上运行的多线程、共享存储的用户程序，一般不在操作系统上运用（见参考文献[6]和[8]）。

虽然这种方法能暂时允许高速缓存内核数据，但是从操作系统性能的方面看，这项技术的好处仍有疑问，因为在释放锁的时候总是要冲洗共享数据的做法降低了高速缓存命中率。用户程序在迁移的时候还是必须要进行冲洗，所以，它比以前的技术没有多提供什么改进。因为它带来的性能增益小，而且实现这项技术又很复杂，所以还没有商业的多线程系统用到

它（在参考文献[2]中可以找到详细的性能分析）。

14.3.3　处理其他存储模型

在采用软件上的高速缓存一致性机制时，包含 store 缓冲的体系结构（如 SPARC）会让问题复杂化。然而，采用第一种技术，在访问共享数据时使用不予高速缓存的操作，除了 14.3.1 小节和第 13 章介绍的之外，不需要再多做什么了。因为不对共享数据和锁进行高速缓存，所以保证了在 store 缓冲中的任何针对共享数据的保存操作都会在释放锁的保存操作之前到达主存储器（要么因为像采用 TSO 那样以 FIFO 次序处理缓冲，要么因为在缓冲里释放锁之前插入一条 store-barrier 指令）。

但是，为了正确地保持一致性，有选择地冲洗高速缓存要多留意。在一个带有高速缓存的系统中使用 store 缓冲时，数据从 store 缓冲送入高速缓存，高速缓存可能会把它送入存储器，也可能不会，这取决于高速缓存的体系结构。在所有情况下，store 缓冲的内容都被送到存储层次结构里的下一级。但是，在此之后如何处理数据则依赖于那一层的体系结构。对于使用写分配的写回高速缓存机制来说，从 store 缓冲来的数据被写入高速缓存，而主存储器不变。随着 store 缓冲清空，写直通高速缓存机制既会更新高速缓存，也会更新主存储器。

所以，譬如说，如果一个系统使用 store 缓冲和一级高速缓存，那么问题则是，在执行有选择地冲洗高速缓存时，针对临界资源的保存操作可能仍然在 store 缓冲中。如果在使用写回高速缓存时出现了这样的情况，那么将会使用来自高速缓存的过时数据，而不是 store 缓冲中的当前数据来更新主存储器。在让高速缓存无效之后，来自 store 缓冲的数据也将到达处理器的高速缓存里。为了使得有选择性的冲洗操作能够起作用，此刻，数据应该只在主存储器中。在锁被释放的时候，其他处理器将访问到主存储器中的过时数据。因此，必须保证在冲洗高速缓存之前先冲洗 store 缓冲。

例如，在 TI SuperSPARC 处理器的情况下，冲洗片上数据高速缓存的操作会导致 store 缓冲的整个内容在继续冲洗之前被送出。它也防止了在完成冲洗操作之前发出更进一步的指令。这就很方便地确保了高速缓存在被冲洗之前是最新的，任何释放锁的保存操作都在高速缓存已被冲洗之后发生。

14.4　小　　结

给 SMP 系统增加高速缓存能够利用局部引用特性的优点提高性能。首选的是私有高速缓存的设计，因为从 CPU 的角度来看，它提供了最小的时延，减少了总线争用和带宽需求。但是，私有高速缓存改变了软件所看到的系统存储模型。除非使用特殊的硬件或者软件技术，否则一个处理器执行的保存操作不会立即被其他处理器看到。这就引发了保持高速缓存一致性的问题。在一个 CPU 发出的上载操作返回的值是系统中任何 CPU 最近对那个位置执行的保存操作所关联的值时，高速缓存就是一致的。

单纯使用软件技术，可以通过两种机制之一保持一致性。第一种机制只是不对有一个以上 CPU 访问和修改的数据进行高速缓存。对任何一个处理器来说是私有的数据，如用户程序

的正文、数据和堆栈，以及进程私有的内核数据结构，像内核栈和u区，可以在进程正运行于一个特殊的CPU上时被高速缓存。如果进程在一次现场切换期间被转移到另外一个CPU上，那么就必须冲洗前面运行该进程的那个CPU的高速缓存，从而保持一致性。这种方法的缺点是它不允许共享的内核数据被高速缓存。

第二种技术即有选择的冲洗，允许共享的内核数据当使用它的处理器在相关的临界区内执行时，被暂时性地高速缓存。在这种情况下，保护临界资源的互斥锁也将防止其他处理器同时访问共享数据，从而防止了不一致的问题。但是，在锁被释放之前，必须从处理器的高速缓存中冲洗掉临界资源。还必须确保每个临界资源占有一个独立的高速缓存行，以防止过时数据被写回到主存储器中。

不管高速缓存的组织结构如何，这两种软件技术都不允许用户的共享存储器被高速缓存，因为这将要求改变外部编程模型。此外，迁移时进程数据也必须冲洗掉，但决不能冲洗自旋锁。这会减少使用带有物理标记的高速缓存组织结构的好处。出于这些原因，纯粹的软件高速缓存一致性方法很少会用到。通过使用额外的硬件来自动保持高速缓存一致性，还是有可能跨越这些界线的。扩展总线监视的概念就能做到这一点，下面一章将会予以介绍。

14.5 习 题

14.1 考虑一个系统，其中所有的处理器都连接到了一个共享物理高速缓存上。如果有冲洗操作的话，那么必须完成什么样的冲洗操作才能在第6章中介绍的高速缓存上保持一致性？

14.2 虚拟高速缓存能被用作共享高速缓存吗？带有物理标记的虚拟高速缓存呢？

14.3 考虑14.2节中介绍的Dekker算法的一致性问题。如果在一个双CPU系统上的高速缓存初始为空，一个CPU在另一个CPU尝试获得锁之前已经获得了锁，那么描述第二个CPU试图获得第一个CPU占有的锁时会出现的一致性问题。说明写直通和写回高速缓存机制各自的问题。

14.4 修改11.3节里给出的信号量函数，在使用有选择地冲洗高速缓存机制的系统上正确工作。每个CPU有一块写回L1高速缓存和一块写直通L2高速缓存。通过调用下面的函数冲洗L1高速缓存：

```
void flush_primary( char *addr, int len, int flag );
```

这个函数获得起始地址和要冲洗数据的字节数。参数flag指出应该执行写回还是、使无效操作，或者两者都执行。它的值为WRITE_BACK和INVALIDATE，可以用OR操作连接两个值表明二者都执行。冲洗第二块高速缓存要调用：

```
void flush_secondary( char *addr, int len );
```

这个函数让高速缓存内给定的地址范围变成无效。在不被高速缓存的存储器内分配一个锁则调用：

```
lock_t *alloc_lock();
```

14.5 给图 12-5 所示的生产者-消费者算法增加有选择的高速缓存冲洗机制。假定事件计数函数已经包含了正确的冲洗机制来管理它们的数据结构。

14.6 图 14-4 和 14-5 展示了在使用写回高速缓存机制时，为什么每个临界资源必须占用一个独立的高速缓存行。解释如果在这个例子中代之以采用写直通高速缓存机制，那么将会发生什么样的情况。

14.7 Motorola MC68040 的高速缓存行为 16 字节，每行有一个有效位，但是有 4 个独立的重写位（dirty bit），对应于该高速缓存行内的每个字。回忆写回高速缓存中每个子行（68040 上每个子行就是一个字）都有独立的重写位时，在该行被替换或者冲洗时只把那些已经修改过的字写回存储器。解释这种高速缓存体系结构对 14.3.2 小节中两个 CPU 访问一个计数器组的例子（其中的临界资源占有独立的高速缓存行）有什么样的影响？

14.8 假定高速缓存行为 16 字节长的高速缓存中，每个字都有单独的有效位和修改位（也就是说，每行有 4 个有效位和 4 个重写位），重做上一题。

14.9 在 4.2.8 小节中，防止用户-内核歧义的一种方法是，在以内核态运行的任何时候都使用独立的键。如果每个处理器都有一块私有高速缓存，那么当以内核态运行时，每个处理器必须使用相同的键吗？

14.10 一个系统上有带有键的虚拟高速缓存，在这个系统上，用户进程在迁移到另一个处理器上时，必须要使用相同的键吗？

14.11 在有多级高速缓存、且所有的高速缓存都是写回高速缓存的系统上，使用有选择的高速缓存冲洗机制，高速缓存被冲洗的次序有关系吗？在为了进程迁移而冲洗高速缓存时，次序有关系吗？考虑写回和使无效两种冲洗操作。

14.12 有选择的高速缓存冲洗技术依赖于互斥锁提供的一次只有一个处理器能缓存临界资源的保证。10.3 节介绍了一种情况，在访问一个进程的 ID 时可以不必上锁。之所以有这种可能是因为访问这样的数据本来就有竞争条件。当使用有选择的高速缓存冲洗机制来保持正确的一致性时，仍然不必上锁吗？

14.13 当采用多读锁时，描述为了保持该锁保护的临界资源的一致性，必须完成的有选择的高速缓存冲洗操作。

14.6 进一步的读物

[1] Adve, S., Adve, V., Hill, M., and Vernon, M., "Comparison of Hardware and Software Cache Coherence Schemes," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp.298-308.

[2] Agarwal, A., and Owicki, S., "Evaluating the Performance of Software Cache Coherence," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 230-42.

[3] Brantley, W.C., McAuliffe, K.P., and Weiss, J., "RP3 Processor-Memory Element," *Proceedings of the 1985 International Conference on Parallel Processing*, 1985, pp.782-9.

[4] Bolosky, W.J., "Software Coherence in Multiprocessor Memory Systems," Ph.D. thesis,

TR 456, Computer Science Department, University of Rochester, May 1993.

[5] Cheong, H., and Veidenbaum, A.V., "The Performance of Software-Managed Multiprocessor Caches on Parallel Numerical Programs," *International Conference on Supercomputing*, June 1987.

[6] Cheong, H., and Veidenbaum, A.V., "A Cache Coherence Scheme with Fast Selective Invalidation," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988, pp. 299-307.

[7] Cheong, H., and Veidenbaum, A.V., "Stale Data Detection and Coherence Enforcement Using Flow Analysis," *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988, pp.138-45.

[8] Cheong, H., "Compiler-Directed Cache Coherence Strategies for Large-Scale Shared-Memory Multiprocessor Systems," Ph.D. thesis, Department of Electrical Engineering, University of Illinios, Urbana-Champaign, 1990.

[9] Cheriton, D.R., Slavenberg, G.A., and Boyle, P.D., "Software-Controlled Caches in the VMP Multiprocessor," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 1986. pp. 367-74.

[10] Cytron, R., Karlovsky, S., and McAuliffe, K.P., "Automatic Management of programmed Caches," *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[11] Dubois, M., and Scheurich, C., "Memory Access Dependencies in Shared-Memory Multiprocessors," *IEEEE Transactions on Software Engineering*, Vol. 16, No. 6, June 1990, pp.660-73.

[12] Dubois, M., Skeppstedt, J., Ricciulli, L., Ramamurthy, K., and Strenstrom, P., "The Detection and Elimination of Useless Misses in MPs," *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993, pp. 88-97.

[13] Edler, J., Gottied, A., Kruskal, C.P., McAuliffe, K., Rudolph, L., Snir, M., Teller, P., and Wilson, J., "Issues Related to MIND Shared-Memory Computers: The NYU Ultracomputer Approach," *Proceedings of the 12th International Symposium on Computer Architecture*, June 1985, pp.126-35.

[14] Edler, J., Gottleib, A., and Lipkins, J., "Considerations for Massively Parallel UNIX Systems on the NYU Supercomputer and IBM RP3," *Winter USENIX Conference Proceedings*, 1986.

[15] Edler, J., Lipkins, J., and Schonber, E., "Memory Management in Symunix II:A Design for Large-Scale Shared Memory Multiprocessors," *Proceedings of the USENIX Supercomputer Workshop*, 1988, pp.151-68.

[16] Gharachorloo, K., Gupta, A., and Hennessy, J., "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp.245-57.

[17] Gupta, A., and Weber, W., "Cache Invalidation Patterns in Shared Memory Multiprocessors," *IEEE Transations on Computers*, Vol. 41, No.7, July 1992, pp.794-810.

[18] Lee, R.L., Yew, P.C., and Lawrie, D.H., "Multiprocessor Cache Design Considerations," *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987, pp. 253-62.

[19] Lee, R.L., "The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors," Ph.D.thesis, Technical Report 670, Center of Supercomputing Research and Development, University of Illinois, Urbana-Champaign, August 1987.

[20] opriore, L., "Software-Controlled Cache Coherence Protocol for Multicache Systems," *Information Processing Letters*, Vol. 33, No. 3, November 1989, pp.125-30.

[21] Louri, A., and Sung, H., "A Compiler Directed Cache Coherence Scheme with Fast and Parallel Explicit Invalidation," *Proceedings of the 1992 International Conference on Parallel Processing*, August 1992, pp. I:2-9.

[22] McGrogan, S., Olson, R., and Toda, N., "Parallelizing Large Existing Programs-Methodology and Experiences," *Proceedings of the Spring COMPCON*, March 1986, pp.458-66.

[23] Min, S.L., and Baer, J.-L., "A Timestamp Based Cache Coherence Scheme," *Proceedings of the 1989 International Conference on Parallel Processing*, 1989, pp.23-22.

[24] Olson, R., Kumar, B., and Shar, L.E., "Messages and Multiprocessors in the ELXSI 6400," *Proceedings of the Spring COMPCON*, March 1983, pp.21-4.

[25] Olson, R., "Parallel Processing in a Message-Based Operating System," *IEEE Software*, June 1985.

[26] Pfister, G.F., et al, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985, pp. 764-71.

[27] Scheurich, C., "Access Ordering and Coherence in Shared Memory Multiprocessors," Ph.D. thesis, University of Southern California, May 1989.

[28] Smith, A.J., "CPU Cache Consistency with Software Support and Using One Time Identifiers," *Proceedings of the Pacific Computer Communications Symposium*, 1985, pp.153-61.

[29] Stenstrom, P., "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer*, June 1990, pp.12-24.

[30] Tartalja, I., and Mulutinovic, V., "An Approach to Dynamic Software Cache Consistency Maintenance Based on Conditional Invalidation," *Proceedings of the 25th HICSS*, January 1992, pp.457-66.

[31] Thacher, C.P., "Cache Strategies for Shared-Memory Multiprocessors," *New Frontiers in Computer Architecture*, March 1986, pp.51-62.

[32] Veidenbaum, A.V., "A Compiler-Assisted Cache Coherence Solution for Multiprocessors," *Proceedings of the International Conference on Parallel Processing*, August 1986, pp.1029-36.

[33] Weber, W.-D., and Gupta, A., "Analysis of Cache Invalidation Patterns in Multiprocessors," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.

硬件高速缓存一致性

第15章

本章介绍在带有私有高速缓存的 MP 系统上，如何通过使用特殊用途的硬件自动地保持高速缓存一致性。首先给出硬件操作的高层描述，然后说明这给软件带来的影响。给出的例子使用了基于总线的 MP 系统中常见的微处理器。本章集中考察基于总线的系统，因为这些系统是最常见的存储器互连（memory interconnect）类型。其他存储器互连类型的一致性只进行简要介绍。

15.1 引 言

硬件高速缓存一致性机制用于在无需软件介入的情况下保持处理器之间共享数据的一致性。这是通过扩展 6.2.6 小节中介绍的总线监视概念，使之也监视其他处理器发生的活动来做到的，总线监视保持了 DMA 操作期间的一致性。监视总线和保持高速缓存内容的一致性对软件来说是透明的。虽然这可以用各种各样的方法来实现，但是所有的技术都要实施一套规则，以此来决定共享数据能够在什么时候被高速缓存。首先，由多个处理器读取的数据可以由各个处理器缓存，因为只要数据是按照只读方式处理的，那么就不会出现不一致性。其次，当一个处理器修改了在各个处理器间共享的数据时，必须用新值使其他处理器高速缓存中的任何已经缓存的副本无效或者被更新。最后，如果一个处理器没有在高速缓存中命中共享数据，由于使用了写回高速缓存机制，该数据在主存储器中的值已过时，那么有该数据最新版本的处理器必须提供它的一个副本给发生缺失的处理器。这些规则合起来就确保了共享数据的任何缓存副本始终都反映为最新的版本。虽然主存储器的值可能是过时的（如果使用了写回高速缓存机制的话），但是决不允许过时的数据进入任何高速缓存。这就防止了上一章中介绍的所有不一致问题。

为了落实这些规则，高速缓存在系统总线上使用一种高速缓存一致性协议（cache consistency protocol）彼此进行通信，以跟踪共享数据的位置，防止缓存过时数据。共享数据是以每个高速缓存行为基础来跟踪的，这要求系统内的所有高速缓存是一样的，因为高速缓存行的大小是保持一致性的单位。此外，使用硬件高速缓存一致性的大多数 MP 系统也使用物理高速缓存。在采用虚拟标记的高速缓存的情况下，硬件没有办法解决使用相同虚拟地址来引用不同数据的处理器之间出现的歧义问题。但是，硬件高速缓存一致性机制可以同也用物理索引的虚拟高速缓存（如 Intel i860 XP 上的高速缓存）一起使用。虽然很少在商业 MP 系统上见到带有物理标记的虚拟索引高速缓存，但也是有可能使用它们的，因为它们在每次

交易的时候要求总线既传送数据的虚拟地址，也传送物理地址。

用在基于总线的系统上的协议被称为监听协议（snooping protocol），因为每个高速缓存都要监视或者监听其他高速缓存的总线活动。监听协议依赖于这样的事实，即所有的总线交易对于系统内的所有其他单元都是可见的（因为总线是一个基于广播通信的介质），因而可以由每个处理器的高速缓存进行监听。有些协议要求在高速缓存标记中保存额外的状态信息，以便能够决定要保持一致性所需的操作（后面的小节里将会给出例子）。用于MP高速缓存一致性的监听机制和用于I/O DMA一致性的监听机制可以组合到一种协议中去，DMA一致性仅仅是MP惯例的一个子集。使用特殊的高速缓存一致性协议，可以准确地指定在什么时候、采取什么样的措施。

这些年来人们已经提出了数十种协议，其中许多都是其他协议的变形或者改进。不同的协议需要不同的通信量。需要太多通信量的协议浪费了总线带宽，使总线争用增多，留下来满足普通高速缓存缺失的带宽就更少。因此，设计人员已经在尝试将保持一致性的协议所需的总线通信量减到最小，或者尝试优化某些频繁执行的操作。

总的来说，这些协议可以分为两大类：写-使无效（write-invalidate）协议和写-更新（write-update）协议。写-使无效协议在一个处理器修改了已经由其他处理器高速缓存的数据时，向系统内的所有其他高速缓存广播一则使无效消息。写-更新协议在一个处理器修改数据的时候广播它的新值，以便系统内的所有其他高速缓存如果正好缓存了受影响的行，就可以更新它们的值。在这两种情况的任何一种里，都是在一个处理器修改了与其他处理器共享的数据的地方消除过时数据，藉此来保持一致性。因为总线使所有的操作串行化了，可以两个试图同时保存同一高速缓存行的处理器会以不确定的次序来串行执行（正如8.3.2小节所阐述的那样）。这就确保了在任何时间点上只有一个高速缓存行的当前版本。

使用硬件高速缓存一致性协议增加了系统的复杂性，因此也就增加了成本。其优点在于，共享数据可以被系统内所有的处理器透明地高速缓存，而没有第14章所讲述的软件高速缓存一致性技术的开销和复杂性。这些优点的好处大大超过了硬件成本，所以几乎在每一种MP系统上都能找到硬件高速缓存一致性技术。

除了数据，硬件技术也可以用于保持指令的一致性。带有统一的指令和数据高速缓存的处理器，如Intel 80486的片上高速缓存，同时保持了指令和数据的一致性。在这种情况下，保持指令的一致性不需要增加复杂度，因为在高速缓存中，指令就如同数据一样。具有独立的指令和数据高速缓存的处理器如果要保持全部的一致性的话，必须在监听操作期间既检查指令也检查数据。像德州仪器公司的SuperSPARC、MIPS R4000和Intel i860 XP这样的处理器就是以这种方式操作的，这在现在是一种常见的做法（早期带有独立的指令和数据高速缓存的MP微处理器经常会不管指令高速缓存的一致性，而把那个任务留给软件，以降低成本和复杂性）。有指令高速缓存的硬件一致性机制的所有处理器并不是都会监视对处理器自己的数据高速缓存的保存操作（例如，处理有自我修改代码的情况）。在这样的系统上，软件仍然必须显式地使指令高速缓存无效，就和它在单处理机系统上必须要做的一样（参见2.10节），从而在这些情况下保持一致性。Intel i860 XP就是以这种方式操作的。它的数据高速缓存使用写回机制，所以被修改的指令必须首先被写回到主存储器中，然后才能使指令高速缓存无效。有自我修改代码的情况很少，不会给软件造成不适的负担。

随后的几节介绍现代多处理机系统使用的几种高速缓存一致性协议，目的是让软件工程

帅熟悉概念，从而让他们在遇到其他协议的时候很容易就能理解。同样，后面对讲解也进行了简化，省略了软件不予关注的硬件细节。在介绍完硬件概念之后的几节里，我们再讨论对软件的影响。

15.2　写-使无效协议

写-使无效协议通过确保在对一个高速缓存行发生保存操作时，主存储器里只有那一行的一个缓存副本存在，来保证一致性。为了说明这一点，考虑一个高速缓存行尚未被系统中任何处理器修改过的情况。此时，多个处理器可能缓存有这一行的多个副本。在对这一行发生一次保存操作时，除了执行保存操作的处理器上的那个副本之外，所有的副本都变成无效的了。如何实现这一点的细节则取决于使用的高速缓存机制是写直通还是写回，以及协议是如何运行的。下面的几个小节介绍 3 种不同的写-使无效协议。

15.2.1　写直通-使无效协议

最简单的写-使无效协议使用的是写直通高速缓存机制，并被 Motorola MC68040 所采用。68040 有独立的指令和数据高速缓存，每一块都是 4K 大小的物理高速缓存，每行 16 字节。两块高速缓存都监听总线，并且适时地更新它们的内容，以保持一致性。它使用的策略如下。

可能会被系统中任何处理器修改的共享数据必须使用写直通策略进行缓存。当一个处理器在共享数据上发生缺失时，它就从主存储器中读取数据。写直通高速缓存机制的使用，保证了主存储器内的数据副本始终都是最新的，而不管上次修改过它的是哪一个处理器。那么，当多个处理器读取相同的数据时，允许每一个处理器缓存一份该数据的副本。当一个处理器写共享数据的时候，它就被写入主存储器。执行此次写操作的总线交易被系统内的其他高速缓存监听，而且如果被写的地址在这些高速缓存中的任何一个里产生一次命中的话，那么它们就会让自己的那份高速缓存行的副本无效。这就迫使与它们自己关联的 CPU 在下次访问该行的时候，从主存储器中重新读取该行的新内容，以确保它们始终接收到数据最近写入的值。注意，这是和 6.2.6 小节 I/O DMA 一例中保持一致性一样的一组动作。68040 用它来保持 MP 和 I/O 的高速缓存一致性。

因为总线一次仅能由一个处理器使用，所以所有的保存操作都要顺序执行，因此即使共享数据被缓存了，系统也会支持顺序存储模型。这意味着 Dekker 算法能够正确工作，自旋锁也可以被缓存。不需要在内核中编入代码来有选择地冲洗高速缓存，因为现在的使无效操作是由硬件完成的。因此，所有的共享内核数据结构和用户共享存储都可以被缓存。15.7 节里将会完整地讲述硬件高速缓存一致性给软件带来的好处。

写直通-使无效协议的缺点是，对于任何共享数据的每一次保存操作都需要一次总线交易。将写回高速缓存机制用于处理器私有数据（68040 支持）能够减少总线通信量，但是不能将写回机制用于共享数据。如果在使用写回机制时，一次保存操作命中了一个没有修改过的高速缓存行，那么高速缓存只会更新它的内容，却不会把修改后的数据发送给存储器。所以，没有交易出现，让其他高速缓存能监听到并使这一行在它们那里的副本无效（如果它们

那里有这一行的副本的话）。因此，支持写回高速缓存机制需要一种不同的协议。

15.2.2 写一次协议

写回高速缓存机制有一种称为写一次（write-once）的简单变形，可以用于构成一种写回-使无效协议。写一次协议是由 Goodman 提出的，它在历史上被认为是第一种写-使无效协议（见参考文献[23]）。Motorola MC88200 支持写一次协议。写一次和正常的写回之间的主要区别在于，首次在保存操作期间命中一个未经修改的高速缓存行时，将更新存储器（也就是说，对一行的首次修改采用写直通）。这就提供了一次总线交易，其他高速缓存能够监听到，并且用于使得它们可能拥有的该行的任何缓存副本都无效。和正常的写回高速缓存机制一样，当一个处理器没有执行过一次初始的上载操作，就对相同的行执行多次保存操作时，使用写分配策略就能减少总线通信量。

总线可以用来把试图同时向相同的行执行保存操作的多个处理器序列化。假定几个处理器有某一缓存行的一个共享副本，它们都同时向这一行执行保存操作。第一个获得总线的处理器用它自己的保存操作更新了存储器，这让该行其他所有的缓存副本变成无效。既然其他处理器不再有该行的一个副本了，那么它们的保存操作就会产生一次高速缓存缺失。接下来获得总线的处理器则会从主存储器读取更新过的副本，从这个处理器的保存操作插入数据（这可能是行内一个不同于第一个处理器修改的字），并且把这行写回到主存储器中（因为写一次规定在处理器首次向一行进行保存操作时实行写直通策略）。所有要修改这行的处理器都以这种方式顺序执行，但次序不确定。因为只有当前修改该行的处理器才能缓存该行的一个副本，因此保持了一致性。

当处理器对一行执行第二次保存操作，而且从第一次写直通到存储器以来，其他处理器都还没有访问过这行时，这一行就进入已修改状态。一旦高速缓存行处于已修改状态，那么同一处理器对相同的行后续执行的保存操作都使用正常的写回语义。除非这行被替换了，否则不会为处于已修改状态的行产生总线交易。既然存储器的内容不再是最新的了，所以高速缓存必须也监视总线，了解其他处理器对存储器的读取情况。如果在一次监听期间，另一个高速缓存正在读取的地址命中了已修改的行，那么带有修改后数据的高速缓存必须把它提供给发生缺失的高速缓存，因为主存储器的内容是过时的。MC88200 通过让试图读取该行的处理器放弃总线交易来实现这一功能。于是，有着已修改过的高速缓存行的处理器把这行写回到主存储器中，从而让它自己的副本成为有效的和未经修改的版本。试图读取该行的处理器接着重新尝试读取，这次就接收到了现在主存储器内的最新副本。有些实现选择让处理器在别的处理器正在把高速缓存行写回到主存储器时读取它。这样做节省了一次总线交易，但是实现起来更复杂。不管实现的细节如何，两块高速缓存都能缓存处于有效的未修改状态的高速缓存行。

如果另一个处理器针对一个地址执行保存操作，没有在它的高速缓存中命中，却命中了另一块高速缓存内的一个已修改的高速缓存行，那么就会出现类似的执行顺序。有了已修改的高速缓存行的处理器把它写回主存储器，但是接着就让它自己的副本无效，因为别的处理器马上就会修改它。这就确保了在修改高速缓存行的任何地方都只有一个它的缓存副本。

当使用正常的写回高速缓存机制时，每个高速缓存行都可以处于 3 种状态之一：无效

（invalid）、有效-未修改（valid-unmodified）和有效-已修改（valid-modified）。为了简洁起见，有效-未修改状态通常也称为有效，而有效-已修改则称为已修改。为了提高写一次协议的性能，加入了一种新状态，每个高速缓存行共有 4 种状态：无效（invalid）、有效（valid）、重写（dirty）和保留（reserved）（“重写”一词和“已修改”一词意思一样）。前 3 种是写回高速缓存的正常状态。保留状态是一种新状态，它表明高速缓存行是有效的，而且正好已经被 CPU 写过一次，主存储器的内容是最新的。它意味着这是该行唯一的缓存副本，因此在 CPU 下一次针对它执行保存操作时，它就会转入已修改状态，而不必向其他高速缓存广播一则使无效消息。它还指出该行只被 CPU 写过一次，当该行被替换的时候不必写回到存储器，因为数据在首次执行保存操作时已经写直通到主存储器里了。保留状态消除了不必要的总线交易，提高了性能。

给高速缓存行加入新状态是硬件高速缓存一致性协议常用的做法。一般用一幅状态图来表示状态以及导致从一种状态向另一种状态转移的事件。图 15-1 给出了写一次协议的状态图，它描绘出高速缓存的一行响应指定的事件所发生的转移情况。系统中每个高速缓存的每一行都执行这个状态图，但它们之间互相独立。

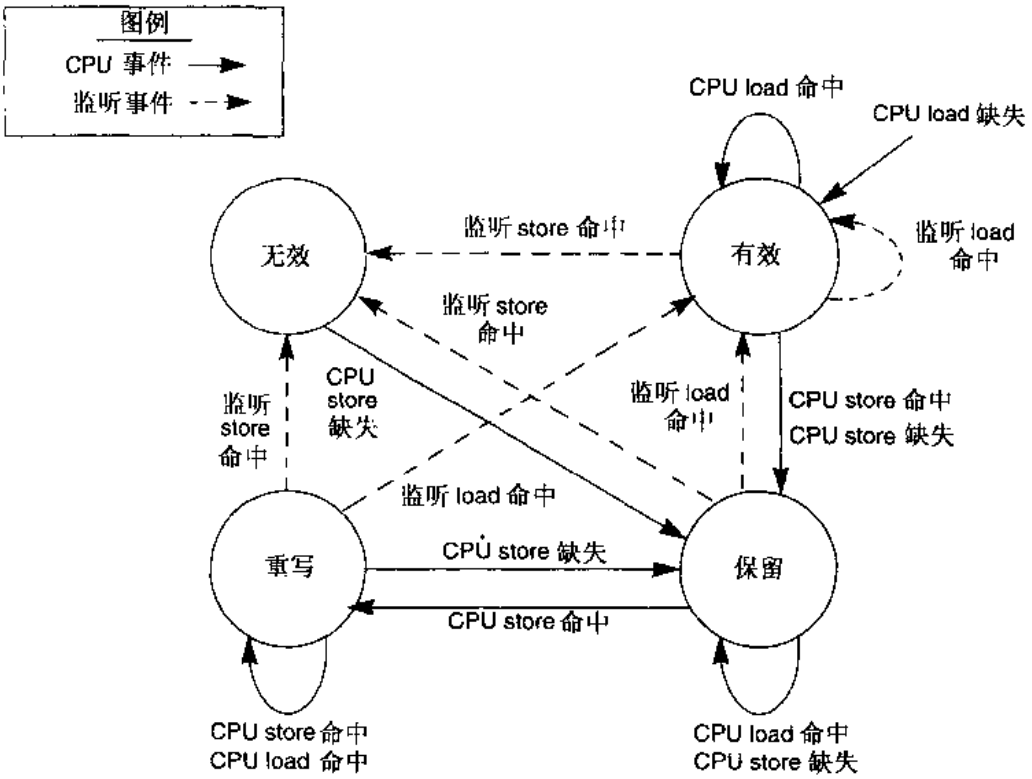


图 15-1　写一次协议的状态图

黑色的弧线表示来自连接到高速缓存的 CPU 的事件所造成的状态转移。例如，如果某一行处于有效状态，CPU 执行了一次上载操作，命中了那一行，那么标有“CPU load 命中”的弧线表明，这一操作最终让高速缓存行保持在有效状态。类似地，如果 CPU 向一个地址执行保存操作，命中了一个有效状态的行，那么标有“CPU store 命中”的弧线表明，作为对该事

件的响应，高速缓存行转为保留状态。注意，这幅状态图只显示了状态的转移，而没有显示状态转移所关联的动作。例如，除了造成转移到保留状态之外，在重写状态的行上，一次“CPU store 缺失”事件会使该行被写回主存储器，然后用缺失地址所关联的数据替换它。因为这是第一次写数据，所以写一次协议要求以写直通方式把它写入主存储器。在执行完这些操作之后，这行才进入保留状态。

在任何状态下，发生一次“CPU load 缺失”事件都会让高速缓存行被替换，然后进入有效状态。为了简化状态图，只画一个进入有效状态的“CPU load 缺失”弧线，因为不管该行最初的状态是什么都一样。

灰色的弧线表示和总线上监听到的事件相关联的转移。所有的高速缓存都能监听到全部总线交易，并且适时地更新它们的内容。例如，“监听 load 命中”意味着另一个处理器在一次上载操作期间，没有在它自己的高速缓存中命中，于是它发出一次总线交易，读取高速缓存行，它正在读的地址命中了这个处理器的高速缓存。监听到没有命中的情况不会改变状态，因此就不画了。

这幅状态图清楚地表明，其他处理器所执行的任何保存操作（导致出现“监听 store 命中”）都会致使该行在其他任何包含该行的高速缓存中无效，因为“监听 store 命中”一定会让该行进入无效状态。这就实现了写-使无效高速缓存一致性协议的规则之一，即只存在一行的一个已修改的副本。类似地，在任何（除了无效状态外）状态中出现的“监听 load 命中”事件一定会让高速缓存行转为有效状态。如果它命中了一个重写行，那么该行会被写回主存储器。之所以要这样做是因为多个高速缓存现在都要缓存该行，所以必须更新存储器。多个处理器都能缓存数据，直到它们中有一个修改了它为止。

最后要注意，即使多个事件会导致相同的状态转移发生，但并不意味着伴随该事件，它们也导致发生了相同的操作。例如，如果原来的状态是有效状态的话，“CPU store 命中”和“CPU store 缺失”都使得状态变为保留状态。在 store 命中的情况下，新数据被插入行中，以写直通方式写入主存储器。对于 store 缺失的情况来说，必须首先从存储器（或者在 snoop 命中的情况下从另一个高速缓存）读取该行，并替换它。然后再把新数据插入行中，以写直通方式写入存储器。

虽然写一次协议允许采用写回高速缓存机制，这免去了每次保存操作都把数据以写直通方式写入主存储器的要求，但是当最初进入的是保留状态时，它仍然要求在纯粹的写回高速缓存机制上多一次写主存储器的操作。这可以用一种更复杂的协议来予以消除，如下一节要介绍的一种。

15.2.3 MESI 协议

MESI 协议给高速缓存行增加了一种所有权（ownership）的概念。一旦高速缓存独自拥有了一个高速缓存行，就可以修改它，而不需要先发送一次写直通总线交易。这就提供了效率更高的写回高速缓存机制。这些协议以一个高速缓存行 4 种状态的首字母而得名：已修改（modified）、独占（exclusive）、共享（shared）和无效（invalid）。Intel Pentium 以及用于 80486 的外部高速缓存控制器都使用了这种类型的协议。它也是 MIPS R4000 上几种可以选择的协议之一。这种协议类型有几种变形，这里介绍其中的一种。

已修改状态等同于写一次协议的重写状态。它意味着相对于主存储器来说，该行已经被修改过了，而且它还暗示其他高速缓存没有该行的副本。因此，在高速缓存中命中的 CPU 保存操作会更新这行，而不必产生任何总线交易。这块高速缓存还必须把数据提供给那些监听操作在这些行上命中的其他高速缓存。

独占状态等同于写一次协议的保留状态。它意味着该行和主存储器保持一致，而且别的高速缓存都没有这行的副本。和写一次的保留状态一样，在独占状态时被 CPU 修改的高速缓存行转入已修改状态，而不会产生一次总线交易。

共享状态意味着该行是有效的，因此 CPU 能够在发生 load 命中时读取它。它也意味着该行可以为其他处理器所缓存，于是，如果不首先发出一次总线交易使其他副本无效的话，就不能对它进行修改。

MESI 协议和写一次协议之间的区别在于，在一次 CPU load 缺失期间如何处理状态之间的转移。当发生一次这样的缺失时，CPU 发出一次总线交易，读取该行。当返回数据时，MESI 协议提供一个特殊的总线信号，指出其他高速缓存中是否有副本（在监听操作期间，找到该行副本的每块高速缓存都确认这个信号）。如果没有缓存别的副本，那么就以独占状态读入该行。如果有别的副本存在，则以共享状态读取它。相比之下，写一次协议一定以有效状态载入数据，并且在发送一次总线交易使可能存在的其他副本无效之后，只能转到保留状态。MESI 协议的状态图和图 15-1 中的状态图类似，不同之处在于，CPU load 缺失后初始进入的要么是独占状态，要么是共享状态。

MESI 协议比写一次协议好的地方是，CPU 能够修改初始以独占状态载入的数据，而不需要一次总线交易使其他高速缓存内的副本无效，因为已经知道不存在这样的副本。和以前一样，有一个已修改状态行的高速缓存必须在监听操作期间把数据提供给其他高速缓存。即使多个高速缓存可能共享一行的一个只读副本，也同样要通过一次只允许一个高速缓存有该行修改过的副本来保持一致性。

15.3　写-更新协议

写-使无效协议通过在一个 CPU 修改一行时使它的其他缓存副本无效来保持一致性。写-更新（write-update）协议则通过在一个 CPU 修改一个被共享的行时更新它的所有缓存副本来保持一致性。下面的例子演示了这类协议。

15.3.1　Firefly 协议

DEC（Digital Equipment Corporation）公司的 Firefly 多处理机工作站使用了一种写-更新协议。这种协议使用的基本状态和 MESI 相同，但在状态转移期间执行的操作上有些不同。和 MESI 一样，在一次 CPU load 缺失之后，高速缓存行依靠来自系统内其他高速缓存的一个总线信号进入独占或者共享状态。类似地，命中独占或者已修改行的 CPU 保存操作只更新局部的高速缓存，因为它得到保证，别的高速缓存都没有这个数据。

这种协议和 MESI 协议的区别在于，针对处于共享状态的行进行的保存操作会产生一次

总线交易，更新系统别处的任何缓存副本，也更新主存储器（在这种情况下，MESI 协议会使其他副本都无效，并且更新存储器，然后转移到独占状态）。注意，在修改处于共享状态的一行时，其他高速缓存可能在缺失期间替换了它们的副本，因此该行可能实际上已经不再是共享的了。替换共享行并不会在总线上得到沟通，所以修改某一共享行的 CPU 必须广播一则更新消息。当它这样做的时候，这行的共享状态被返回给发起更新的高速缓存，就和 load 缺失的情形一样。如果发现别的高速缓存仍然有数据的一份副本，那么该行就留在共享状态。如果没发现，那么该行转移到独占状态。这样一来，如果其他高速缓存不再有该行的副本，就省去了将来发送更新消息的要求。

写-更新协议假定共享数据继续保持共享。写-使无效协议能够造成一种“ping-pong”效应，此时被多个处理器修改的一行会随着每个处理器使其他副本无效的操作而在它们的高速缓存之间来回移动。因为数据在被写入之前频繁读取，所以 MESI 协议要求有一次总线交易在首次缺失时读取该行，还要求有第二次交易在首次写时使其他副本无效。但是，在采用 Firefly 协议的情况下，一旦所有的处理器都获得了它们正在共享的一行的一个副本，那么每次有任何处理器修改该行时，只要有一次总线交易就能更新其他副本。这就有可能将用于大量共享数据的总线通信量减少一半。

但比起 MESI 来说缺点是，对于留在一个 CPU 局部的数据，可能会出现额外的通信量。例如，当一个进程从一个处理器迁移到另一个处理器的时候，它以前运行的处理器上的高速缓存可能仍然缓存着它的部分私有数据，如 u 区和内核堆栈，以及用户程序的正文、数据和堆栈。当新进程引用这个数据的时候，这些行就进入共享状态，并且就呆在那儿，直到它们在以前的处理器上被替换了为止。因为即使以前的处理器没有进一步引用数据，新处理器也要向以前的处理器发送更新消息，从而浪费了总线带宽，这是我们所不希望的。对于这类数据来说，最好是使老的副本无效，而不是更新它们。

15.3.2 MIPS R4000 更新协议

MIPS R4000 允许操作系统逐页地选择 5 种高速缓存一致性策略中的一种（通过在页表项设置适当的比特位）。在这 5 种协议中有一种是更新协议，它的功能类似于 Firefly 协议。R4000 能够配置成使用附加的第 5 种高速缓存状态——已修改-共享（modified-shared）状态，当更新主存储器的时候它就会起作用。只要向共享数据发送了一次更新，或者当处于已修改状态的行被替换时，Firefly 协议就会更新主存储器。在 R4000 上使用已修改-共享状态时，更新被发到处于共享状态的数据的其他高速缓存，但不更新主存储器。修改该行的高速缓存进入已修改-共享状态，以指出当该行被替换的时候需要写回到主存储器。省去更新主存储器可以简化存储子系统的设计，并且提高更新总线交易的性能。这个思想也用在了 Xerox PARC（Palo Alto Research Center，Palo Alto 研究中心）所开发的 Dragon 协议上。

15.4 读-改-写操作的一致性

在 8.3 节里，我们定义了原子的读-改-写操作，它从主存储器中读取一个值、修改它，再

将它写回，全部过程作为一次原子操作。对于使用写直通一致性协议（如 MC68040 上的写直通-使无效协议）的系统来说，正是如此。这些协议在执行写操作时一定要访问主存储器，让其他的高速缓存能够监听到交易，如果它们有该数据的任何副本的话，就会使之无效。如果开始执行原子的读-改-写操作，在读取阶段发生了一次高速缓存命中，那么只要它为了完成操作已经获得了对总线的独占使用权，它就可以使用这个值。此刻，从高速缓存中读到的值保证和主存储器里的一样（否则协议已经让它无效了）。一开始就获得总线，防止了在原子操作正在进行当中，其他处理器向相同的存储位置执行保存操作。在对主存储器的写阶段，其他所有的高速缓存都会监听交易，并且让它们可能有的数据副本无效，在它们下次访问的时候，就会因为不能命中数据而从存储器读取到新的值。和没有高速缓存的系统一样，在原子操作开始时发生的读缺失会自动导致从主存储器中读数据、修改并写回存储器。

在使用写回协议的时候，不一定要访问主存储器，只要确保包含读-改-写操作目标的高速缓存行自动得到更新就够了。高速缓存一致性协议接着就能保证一致性，并且防止两个或者两个以上的 CPU 同时对同一行执行原子操作。

为了说明这一点，考虑 TI SuperSPARC（在 8.3.3 小节中所定义）的原子交换操作。因为 CPU 将要修改的行包含要被交换的字，所以 CPU 的高速缓存必须有该行的唯一一份副本。因为 SuperSPARC 使用 MESI 协议，所以如果这行已经处于已修改或者独占状态，那么由于该 CPU 已经独占了该行，就不需要总线交易。如果发生了缺失，那么高速缓存必须获得该行的一份独占副本。以一次特殊的总线交易指出 CPU 的目的是修改该行，就可以做到。因此，在系统中一个高速缓存向发请求的 CPU 提供该行之后，系统内其他的高速缓存都使它们自己的副本变成无效。如果这行最初处于共享状态，那么就在总线上广播一则使无效消息，于是该行便进入独占状态。此刻，执行原子读-改-写操作的 CPU 独占该行，所以它可能继续执行交换操作，把寄存器内的值同行内所要的字交换一下。在一切情况下，该行最后都进入已修改状态。如果在交换操作正在进行当中的时候，另一个 CPU 试图访问这行，那么在交换结束之后，占有该行的 CPU 就会把它提供给另一个 CPU。和通常一样，当多个 CPU 同时试图访问同一行时，总线将访问顺序化了，而次序却不确定。

MIPS R4000（也是在 8.3.3 小节内介绍的）使用的 load-linked、store-conditional 操作也要依靠硬件高速缓存一致性协议，但是，由于不能保证这对指令的执行是原子的，所以在一对 load-linked/store-conditional 操作之间，可能会有其他处理器访问该行。如果发生了这样的情况，那么就认定 store-conditional 操作失败。实现这两条指令的方法有好几种。一种是在进行保存操作之前观察高速缓存行的状态。当执行完一次 load-linked 操作时，CPU 获得的高速缓存行要么处于独占状态，要么处于已修改状态，就好像执行过原子交换操作，然后 CPU 执行上载操作一样。它还会设置行内的标志，指出一对 load-linked/store-conditional 操作正在执行。任何监听到对该行的访问都会造成标志被清除。当执行 store-conditional 操作的时候，它要检查标志是否还在。如果在，那么说明没有别的处理器访问过这行，保存操作能够成功地完成。否则，说明在 load-linked/store-conditional 指令对之间有别的处理器访问过这行，这会造成 store-conditional 失败，因为该行不是以原子方式进行更新的。

在高速缓存中执行原子操作和要求访问主存储器相比，具有减少总线交易的优点，尤其是在要操作的数据已经驻留在高速缓存中，于是可以有助于提高系统整体性能的情况下，更是如此。需要说明的一点是，当采用写回高速缓存的时候，根本没有必要更新存储器。高速

缓存一致性协议能确保这些操作在所有情况下的正确性。

15.5 多级高速缓存的硬件一致性

当 MP 系统上的每个处理器使用一种层次结构的高速缓存时，也可以将硬件技术用于保持 MP 系统上的高速缓存一致性。MIPS R4000 和 TI SuperSPARC 都支持多级一致性，它们两者的片上高速缓存之间有一点儿不同。两种处理器使用的片上指令和数据高速缓存都配有片外的二级高速缓存（L2 cache）。两者的二级高速缓存都使用写回策略。对于片上数据高速缓存来说，R4000 使用写回策略，而 SuperSPARC 使用的却是写直通策略。两种处理器都使用包含属性（inclusion property）来保持一致性。这意味着一级高速缓存（L1 cache）始终是二级高速缓存的一个子集。为了保持包含属性，只要一级高速缓存中出现了缺失，那么二级高速缓存就会载入数据的一个副本。此外，只要有一个二级高速缓存行被替换或者使之无效，那么相应的一级高速缓存行要被写回存储器，并使之无效。使用包含机制意味着可以通过只询问二级高速缓存，然后对于在二级高速缓存中命中的监听操作只访问一级高速缓存的做法来处理监听机制。没有在二级高速缓存中命中的监听不会对一级高速缓存有影响。这就有助于减少对一级高速缓存的争用，从而让更多的一级高速缓存带宽供 CPU 使用。

为了保持一致性，二级高速缓存要监听其他处理器的总线活动，按照高速缓存一致性协议进行操作。例如，MIPS 和 SPARC 处理器都支持一种写-使无效协议。所以，如果二级高速缓存检测到一次写操作，而它已经缓存了要写的行，那么它就让二级高速缓存内的这行无效。因为一级高速缓存有该行的一份副本，所以只要监听操作在二级高速缓存中命中，那么就要对它进行检查。由于 R4000 上的一级高速缓存是虚拟索引的，所以保存在二级高速缓存行中的一级高速缓存索引，要用来索引可能保存将要无效的数据的一级高速缓存行（MIPS R4000 上的高速缓存操作在 6.3.2 小节中详述）。TI SuperSPARC 使用物理索引的一级高速缓存，所以在二级高速缓存中的监听操作所使用的物理地址要用来索引一个高速缓存。在两种情况中的任何一种情况下，如果在一个高速缓存中发生一次命中，那么就要使高速缓存行无效。

当正在监听高速缓存，以满足另一个处理器上的高速缓存 load 缺失的时候，首先要检查二级高速缓存。因为 SuperSPARC 在它的一级高速缓存中使用了写直通高速缓存机制，所以可以直接提供监听操作所请求的数据，而不必访问一级高速缓存。R4000 上的情况并非如此，因为它的一级高速缓存使用的是写回高速缓存机制。这要求在这些情况下，也要对它进行已修改数据的检查。

来自 CPU 的保存操作的行为和大家期望的一样。在 SuperSPARC 上，在一级高速缓存中命中的保存操作能够立即以写直通方式写入二级高速缓存，因为包含机制保证了二级高速缓存也会有一份该行的副本。如果它命中了二级高速缓存中一个已修改过的高速缓存行，那么写-使无效协议表明，不需要有总线交易，因为这个处理器是唯一缓存该数据的处理器。如果二级高速缓存中的该行没有被修改过，那么必须在总线上发出一次使无效交易，以清除可能存在的其他任何缓存副本。接着，二级高速缓存行进入已修改状态。因为 R4000 在一级高速缓存中使用了写回高速缓存机制，所以当使用 MESI 写-使无效协议时，命中已修改或者独占

行的保存操作不需要访问二级高速缓存，也不会产生任何总线交易。在两种处理器中任何一种的一级高速缓存上发生缺失的保存操作，都会导致在二级高速缓存中给该行分配空间（如果它尚未出现的话），并且从主存储器或者其他处理器的高速缓存中读取数据。

15.6　其他主要的存储体系结构

使用一条总线作为多个 CPU 和主存储器之间的互连媒介，对于有多达数十个处理器的系统来说，是一种简单而且划算的方法。例如，SGI（Silicon Graphics）生产的 Challenge 系统就是这样，它能够在一条主存储器总线上支持多达 36 个处理器。但是，随着总线上处理器数量的增加，争用也会增加，总线成为了瓶颈。此外，电气上的限制因为总线的长度和连到它上面的连接数也抑制了总线的速度。最终的结果是，总线成了大规模 MP 系统性能的限制因素，因此需要一种不同的存储器互连类型。使用的互连类型也影响着硬件高速缓存一致性的实现。

15.6.1　交叉开关互连

为了克服单一总线方法的限制，人们已经提出了其他几种互连方法，并且在一些大规模 MP 系统上得以应用。交叉开关（cross-bar）就是一种这样的方法，它如图 15-2 所示。

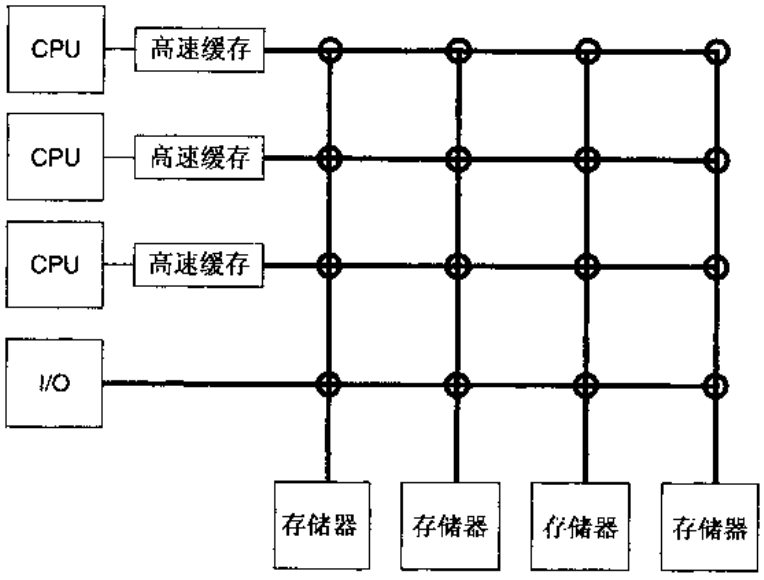


图 15-2　使用交叉开关存储互连的 SMP 系统示例

图 15-2 给出了一个有 3 个 CPU（每个 CPU 有一块私有的高速缓存）、1 个 I/O 子系统以及 4 个主存储器模块的 SMP 系统（当然，设计人员能够随意改变处理器、存储器模块以及 I/O 设备的数量，交叉互连也不需要是方形的）。交叉开关的思想是，提供多条总线，可以同时投入使用，从而减少争用，再就是提供多个存储器模块，可以并行操作，从而提高了存

储器的整体带宽。物理地址空间在诸存储器模块间进行划分，于是与每个地址相关联的数据都保存在一个且仅有一个的位置上。做到这一点的一种方法是，每个模块保存一段连续的物理地址空间。例如，如果每个模块包含 256M，那么第一个模块为从 0x0 到 0xfffffff 的物理地址保存数据，第二个模块保存从 0x10000000 到 0x1fffffff 物理地址内的数据，依此类推。虽然这种方法很简单，但缺点是对邻接地址的访问（因为空间局限性而在短时间内经常发生）会到相同的存储器模块。为了让时钟周期重叠得更好些，许多设计将存储器交错，让物理地址空间在多个模块上横向条带分布。在这种情况下，保存在一个模块中的连续存储量往往等于高速缓存行的大小。例如，如果行的大小是 256 字节，那么物理地址 0x0 到 0xff 的数据将会被保存在第 1 个模块中，地址 0x100 到 0x1ff 在第 2 个模块中，0x200 到 0x2ff 在第 3 个模块中，0x300 到 0x3ff 在第 4 个模块中，依此类推。物理地址在多个模块间交错分布的确切方式对于软件来说是不可见的，不作进一步考虑。

在交叉开关阵列中的每个圆圈都代表一个开关，它可以由硬件打开和关闭，临时连接两条交叉的总线。在正常情况下，所有的开关都是关闭的，直到有个 CPU 或者 I/O 设备需要访问存储器为止。根据要访问的物理地址，交叉开关的硬件切换开关，将产生请求的单元同适当的存储器模块连接起来，形成一条访问路径。例如，如果图 15-2 里最上面的 CPU 需要访问最右边的存储器模块，那么交叉开关就在访问期间把右上角的开关打开。当访问结束以后，开关又被关闭。如果有一个以上的 CPU 需要同时访问相同的存储器模块，那么交叉开关硬件就会对这些请求进行仲裁，方式和一条标准总线差不多，即对任何一个存储器模块，一次只允许一项存储访问。同样，在任何一列，一次只能打开一个开关。

即使没有一条公用的总线，这种存储体系结构仍然能够成为 8.2 节所描述的 SMP 系统。CPU、I/O 和存储器都在彼此很近的距离内紧密地耦合在一起。在交叉网络内的所有存储器都是共享的，能够全局访问。从处理器的角度来看，对存储器的访问是对称的。交叉开关仲裁对模块的同时访问，而且保证公平性。

因为交叉开关一次只允许一个 CPU 或者 I/O 设备被连接到一个给定的存储器模块上，所以它显然能够以和单总线系统相同的方式支持所有类型的原子操作。从每个模块来的垂直总线以不确定的次序把对那个模块的所有存储器访问都顺序化了，就好像一条正常的总线一样。另外，CPU 实现的存储模型也不会由于出现了交叉开关而受到影响。诸如 PSO 这样的模型只会影响 store 被发送到存储器的次序。交叉开关网络如何将数据路由到存储器对存储模型或者软件没有影响。

交叉开关结构的优势在于，它提供了到存储器的多条路径，从而减少了争用。此外，也使并行化有了可能，因为两个或者两个以上的 CPU 可以同时互不干扰地访问不同的存储器。在图 15-2 中，左边的所有 4 个单元可以同时访问存储器，只要它们使用的物理地址解析到了不同的存储器模块上即可。在单总线系统中是不可能有这样的并行性的。交叉开关也减少了每条总线同时的连接数，这缓解了一些电气上的限制，允许更快的总线速度。总而言之，它能减少争用，并允许增加存储器带宽。

交叉开关的缺点是，多条总线比单总线系统更贵。好处是提高了系统性能。第二个问题在于如何维护高速缓存一致性。由于每个 CPU 有它自己到存储器的路径，而且在一个存储器模块被另一个 CPU 使用的时候，别的 CPU 不能连接到该存储器模块，所以无法使用以监听保持高速缓存一致性的机制。监听机制依赖于这样的事实，即所有的总线交易对系统内的所

有高速缓存都是可见的。没有一条公共的总线就丧失了这一能力。相反，通过使用基于目录的高速缓存机制，硬件就可以自动保持高速缓存的一致性。

15.6.2　基于目录的硬件高速缓存一致性

前面介绍的基于监听的一致性机制都要依靠使用高速缓存行的状态来判断监听操作期间采取什么样的措施。在这点上，维护一致性所需的信息分布在了多个高速缓存上。例如，为了判断一行是否是共享的，所有的高速缓存都必须检查是否命中，然后还必须考察高速缓存行的状态。对于交叉开关的结构来说，将维护一致性所需的信息分布到多个高速缓存上并不合适，因为没有广播能力就意味着不能完成监听机制。相反，这种存储结构把维护一致性所需的信息集中保存在主存储器模块中。这些信息称为目录（directory），它的目的是指出哪些高速缓存有存储器每一行的副本，以及高速缓存行的状态。

从概念上说，最好把模块内的存储器分成高速缓存行大小的小块，这样来看待主存储器的结构和目录。和每块相关联的是那行的目录信息，如图 15-3 所示。在这个例子中，假定行的大小为 256 字节，每个存储器模块保存了物理上连续的多行（也就是说，存储器不是在模块间交错分布的）。

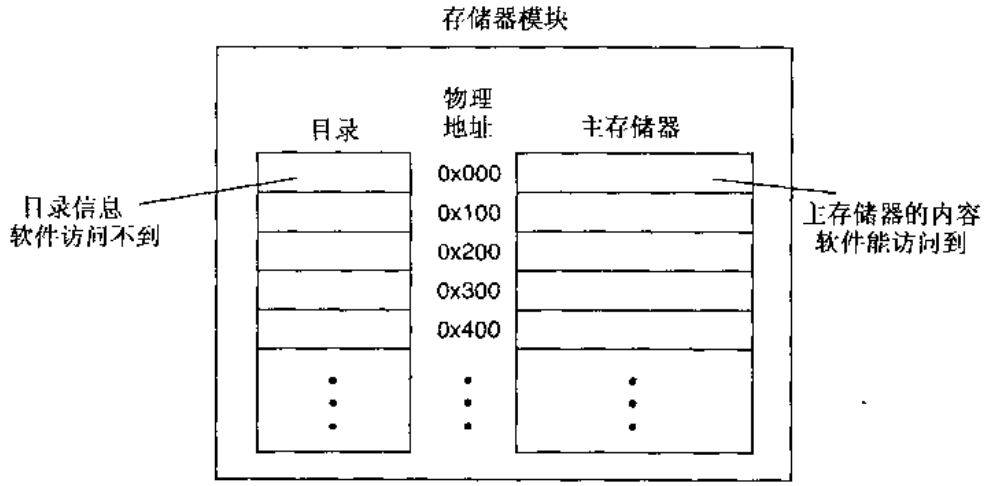


图 15-3　包含一个目录的存储器模块

在最简单的情况下，目录的内容是系统内每个高速缓存的缓存标记中状态位的一个副本，这些高速缓存当前缓存对应的物理存储行。这是 Tang 首先提出的结构（见参考文献[48]）。目录的内容软件不可访问，它们由硬件使用和自动更新，以保持一致性。通过保留高速缓存标记的一个集中副本，硬件不需要向所有的处理器广播存储器访问，就能决定采取什么样的一致性措施。只有对那些有着被引用的存储器行副本的高速缓存才会发生通信。

举个例子，考虑图 15-3 所示的 3 CPU 系统。为了简洁起见，假定高速缓存使用写直通策略。这意味着标记中唯一的状态信息就是有效位（高速缓存标记中的地址不认为是状态信息）。因此，每个存储行的目录信息都由 3 个比特位组成，是 3 个高速缓存的有效位的副本。图 15-4 显示了目录里一行的内容是怎样的。

图 15-4 一个目录行的内容

对于存储器模块内的每一行来说，如果一个特殊的 CPU 正缓存着该存储器行的一个副本，那么就会设置在相应目录行内针对该 CPU 的有效位。只要高速缓存行的状态发生变化，那么就要更新目录内的比特位。所以，当 CPU 执行一次上载操作，导致目录内适当的比特位被设置的时候，就发生了一次高速缓存缺失。这在存储器模块向 CPU 返回被请求的数据时就完成了。在高速缓存中替换一行的时候，要通知存储器模块，这样就可以清除在该行的目录项内的有效位（有些实现放弃了这一步，以减少通信量。在这种情况下，目录内的状态位代表了有该行副本的高速缓存的一个超集）。随后访问要替换高速缓存行的数据（可能在不同的存储器模块里），并且针对 CPU 设置在那行的目录项中的有效位。接着数据被返回给高速缓存。

随后使用标准协议来保持高速缓存一致性。对于本例而言，适合采用写直通-使无效协议，因为采用了写直通高速缓存机制。正如在 15.2.1 小节中所讲述的那样，在多个高速缓存读取相同的存储行的同时可能会缓存它们的副本。当一个 CPU 写该行的时候，必须使其他的副本无效。目录项指出了需要向哪些高速缓存发送使无效命令。考虑这样的情形，CPU 1 和 CPU 2 缓存有物理地址为 0x100 的行的副本。CPU 1 现在对那个位置执行一次保存操作。使用写直通意味着新数据将会被立即发送到存储器。一旦那样的话，就会检查存储行 0x100 的目录项，并且发现 CPU 2 有该行的一个副本。于是硬件会进行仲裁，并且在交叉开关网络上存储器模块和 CPU 2 之间建立一条连接。接着存储器模块向它发送一条用于物理地址 0x100 的使无效命令，然后 CPU 2 让那行从它的高速缓存中无效。同时，存储器模块用来自 CPU 1 的新数据更新那行，并且更新目录，以显示现在只有 CPU 1 才有该行的副本。现在，保存操作就完成了。注意，连接到 CPU 3 的高速缓存决不会被访问到，因为它用于该行的有效位为 off（关闭）。类似地，如果在 CPU 1 执行保存操作的时候，目录指出这个 CPU 有该行的唯一副本，那么就不需要其他的通信来确保一致性。这是基于目录的一致性机制比监听机制的优势所在，因为目录有助于减少所需的通信量。

总而言之，目录能够在不需要广播操作的情况下保持一致性。以前描述的任何协议都可以用目录（包括写回协议）在一个交互连接存储器结构上实现。除了简单地保持高速缓存标记中状态位的一个确切副本之外，目录中的信息还能以若干不同的方法来组织。之所以这么做是为了减少信息量，以及体现人们更乐意要的一致性操作是什么样的。不管目录的结构如何，目录的操作对于软件来说都是透明的。当然，目录的缺点是，它们需要更多的存储器单元，比在基于总线的系统中能找到的分布式高速缓存要多，因为目录信息是高速缓存标记的冗余副本。参考 15.12 节，了解有关这个主题的更多阅读材料。

15.7 对软件的影响

使用前面介绍的任何一种硬件高速缓存一致性协议，无论是基于总线的系统还是别的

存储结构，都能够缓存所有形式的共享数据，并且缓解操作系统冲洗共享数据以维护一致性的负担。首先，考虑用户进程数据的情形。正如 14.2 节描述的那样，当进程从一个处理器向另一个处理器迁移，并且引用了局部高速缓存或者主存储器中的过时数据时，就会出现一致性问题。参考这个例子以及图 14.3 中所示的高速缓存状态，如果进程运行在 CPU 2 上，并且读取计数器的值，然后累加它，那么硬件高速缓存一致性协议将从 CPU 1 的高速缓存中监听到当前的值。当 CPU 2 保存累加后的值时，根据采用的协议不同，缓存在 CPU 1 中原来的值不是变得无效了，就是被更新。在使用硬件高速缓存一致性机制的情况下，能够保证进程引用到数据的正确副本，而不管它当前正运行在哪个处理器上，也不管被缓存的数据在什么地方。

类似地，采用硬件一致性机制能够解决在使用 Dekker 算法时出现的高速缓存问题。问题之一是，在使用写回高速缓存机制时，来自一个处理器的保存操作不能立即被别的处理器看到。第二个问题是，一个处理器会访问到它自己的高速缓存中的过时数据，而且在锁已经投入使用的时候还认为锁是空闲的，反之亦然。硬件高速缓存一致性机制通过给每个处理器提供一个最新的共享数据副本（要么是在高速缓存中，要么是在主存储器中），就可以解决这两个问题，它还能防止高速缓存保留过时数据。出于这个原因，所有的自旋锁不管是由 Dekker 算法实现的，还是由原子的读-改-写操作实现的，都可以安全地缓存。

有选择的高速缓存冲洗这种软件一致性技术的额外限制之一就是，分开的临界资源必须占据分开的高速缓存行，否则不同的 CPU 会同时更新同一高速缓存行的不同部分（见 14.3.2 小节）。在使用硬件高速缓存一致性机制时，对这一限制不作要求，因为硬件逐行维护一致性。如果一个 CPU 引用了一个独立的临界资源，该资源和别的资源共享一个高速缓存行，而且另一个处理器缓存了这一行，该行处于已修改状态（如图 14-4），那么高速缓存一致性协议将会把已修改数据的一个副本提供给引用该行的 CPU。例如，如果图 14-4 中的 CPU 2 在读取由 CPU 1 修改过的数据时，已经使用了写-使无效协议，那么这行将会被写入主存储器，两块高速缓存则以共享状态缓存它。高速缓存和存储器的状态如图 15-5 所示。

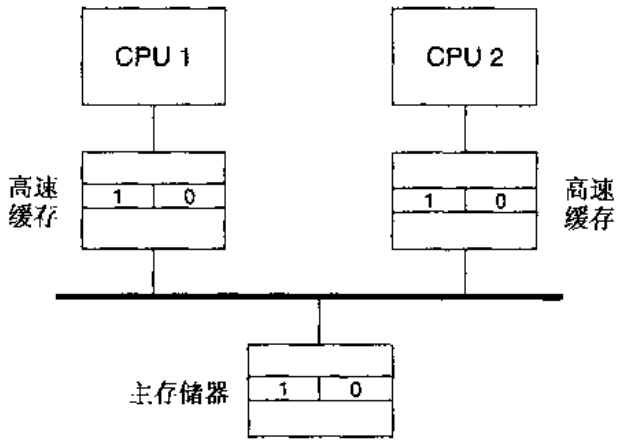


图 15-5　两个 CPU 都访问同一高速缓存行内的临界资源

如果CPU 2现在要累加计数器，并且把计数器更新后的值保存在高速缓存行的右边部分，

那么写-使无效协议会同时使该行的其他任何副本都无效。这意味着绝对不会出现在上一章图 14-5 中所示的情形，因为所有的硬件协议都能够防止两个或者两个以上的高速缓存缓存同一行的不同版本。相反，在 CPU 2 更新了它的临界资源之后，将会出现图 15-6 所示的情况（假定协议在使无效操作期间更新存储器）。

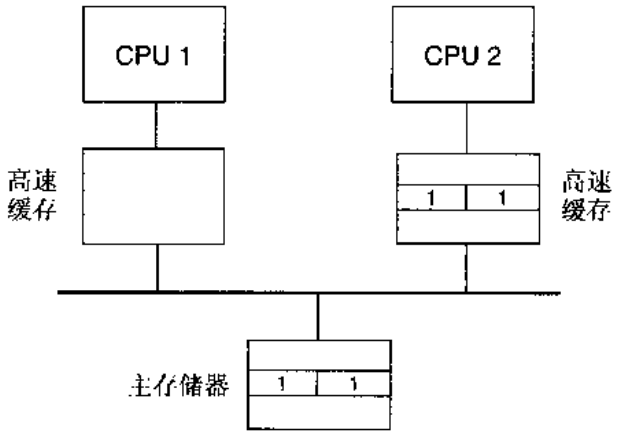


图 15-6 在 CPU 2 更新了它的临界资源之后的状态

如果 CPU 1 要在此刻访问它的临界资源，那么它会发生缺失，根据协议的不同，要么从主存储器，要么从 CPU 2 的高速缓存中读取当前的值。

总而言之，硬件高速缓存一致性协议能够处理所有的情形，因此能够减轻操作系统需要直接管理共享数据所造成的负担。和 DMA 总线监视机制一样，就数据一致性而言，大大缓解了存在高速缓存的影响。不管选择第二部分中的哪一种 MP 内核实现，都是如此。

15.8 非顺序存储模型的硬件一致性

在 14.3.3 小节里介绍过带有高速缓存的 MP 系统，它的 CPU 支持非顺序存储模型（如 TSO 或者 PSO）可以通过向一级高速缓存发送 store 缓冲的内容来运行。此刻如何处理数据则取决于高速缓存的实现和一致性协议。因为硬件高速缓存一致性技术有效地向软件隐藏了高速缓存的存在，所以除了 13.4 和 13.5 节所介绍的变化之外，内核不需要多做什么，就能运行在具有高速缓存和非顺序存储模型的 MP 系统上。在 TSO 下运行时，使用锁以及获得锁的特殊原子操作，对于大多数算法都会得到正确的结果。当释放锁的时候增加的 store-barrier 指令解决了由 PSO 带来的新问题。和没有高速缓存的 MP 系统一样，原本依赖于顺序定序的算法（如 Dekker 算法）不能在采用硬件高速缓存一致性技术的系统上使用。

前面的讨论说明了这样一个重要的事实：store 缓冲和硬件高速缓存一致性技术应该看作是解决系统性能不同方面问题的两种独立的机制。store 缓冲把数据从 CPU 传送到主存储器，从而不会拖延 CPU。硬件高速缓存一致性技术让处理器看到了一致的存储器内容。即使高速缓存要比主存储子系统速度快，但是 store 缓冲仍然很有用处，因为它们能把保存操作期间高速缓存缺失的一些损失屏蔽掉。

15.9　软件的性能考虑

如前所述，人们已经发明了不同的高速缓存一致性协议来减少保持一致性所需要的总线通信量。编程人员理解了协议在特定系统上是如何运行的，就能够通过避免产生不必要监听的操作来进一步减少总线通信量。过多的监听不但会导致总线争用，而且它还能造成对响应监听请求的处理器上高速缓存的争用，因为同时会有来自高速缓存自己的 CPU 和来自总线的请求。因此，减少监听量有助于提高系统的整体性能。下面几小节就介绍几种要予以考虑的例子。

15.9.1　数据结构在高速缓存内对齐

7.5 节中已经介绍过在高速缓存行边界上对齐数据的概念。其目的在于，减少将数据结构载入一个处理器的高速缓存时所需的高速缓存缺失数量（假定数据结构内部有高度的空间局部性）。通过在 MP 系统内对齐数据结构，从 CPU 的角度来看，不仅减少了缺失，而且也减少了监听操作的数量。临界资源对应的数据结构也适用于此项技术，因为大家知道，一次只能有一个处理器使用临界资源。

举个例子，考虑由一个自旋锁保护的临界资源。假定它的数据结构（包括锁）能够放入一个高速缓存行，而且假定是对齐的。在处理器访问数据结构之前，它要先获得锁。如果自旋锁是采用原子交换操作实现的，它按照 15.4 节介绍的那样操作，那么原子交换会获得高速缓存的一份独占副本来执行它的操作。假定锁是空闲的，处理器立即获得了锁，让高速缓存行处于已修改状态。此刻，CPU 能够访问数据结构，而无需任何进一步的总线交易（假定在缺失期间不会替换该行）。在最好的情况下（锁是空闲的），使用 MESI 协议时，只需要一次交易就能获得该行的一份独占副本。因为多个不同的 CPU 使用着临界资源，所以它能用一次交易就从一个高速缓存迁移到另一个高速缓存。

但是，将数据结构和它的锁组合到同一个高速缓存行中有一个缺点。如果对临界资源争用得很厉害，那么别的 CPU 会试图获得它的锁，从而导致高速缓存行在它们之间移来移去。这就使得使用该资源的 CPU 遇到额外的高速缓存缺失。缓解的办法之一是给锁分配的空间在一个独立的高速缓存行内。这就让一个处理器在使用数据结构的时候，其他处理器不会发生争用，因为它们在为锁而自旋。遗憾的是，如果多个处理器都试图获得一个锁，而该锁已经被锁住了，那么随着处理器执行原子操作来获得锁，保存该锁的高速缓存行就会在处理器之间弹来弹去。这就导致了不必要的总线争用，而且浪费了带宽，但使用上一节介绍的技术就可以解决这个问题。

在高速缓存里对齐数据结构和临界资源的时候，一般最好对它们进行填充，让它们每个都占用自己的一行（如图 7-10 所示）。这样做不但减少了初始访问数据结构时导致的缺失，而且防止了错误共享（false sharing）现象。当两个或者两个以上的不同临界资源占有同一高速缓存，而且它们又被多个 CPU 同时使用时，就会出现错误共享。这可以在 14.3.2 小节中看

到（见图 14-4）。虽然硬件维护了正确的一致性，但是当 CPU 之一写这一行的时候，就出现一次使无效或者更新。这会让该行在处理器之间弹来弹去，造成不必要的总线通信。对数据结构进行填充，使它们对齐并占满单个行，就可以消除错误共享和不必要的总线通信。缺点是浪费了存储器和高速缓存空间。

15.9.2 在获得自旋锁时减少对高速缓存行的争用

图 9-2 所示的自旋锁实现在锁字上重复执行原子读-改-写操作，直到得到了锁为止。对于没有高速缓存的系统来说，这是正确的做法。但是，正如在上一小节讲过的那样，它会造成包含锁字的高速缓存在处理器之间弹来弹去。通过改变获得锁的函数实现，就能防止这种现象。注意，在锁被另一个处理器占有的时候，原子读-改-写操作并不能获得锁。因此，在锁空闲之前，没有必要使用这样一种操作。相反，试图获得一个已经投入使用的锁的其他处理器可以只读取锁的当前状态，而在锁被释放的时候只使用一次原子操作。图 15-7 给出了 lock 函数的另一种实现，它使用了这项技术。

```
void
lock( volatile lock_t *lock_status )
{
    while( test_and_set( lock_status ) == 1 )
        while( *lock_status == 1 )
            ;
}
```

图 15-7 自动锁住一个自旋锁

这里，在进入内层循环之前，要做一次得到锁的尝试，然后就等待锁被释放。函数可以写成，在尝试原子操作之前只读一次锁的状态，但是只有在别的 CPU 想要获得锁的时候，锁往往已经在用的情况下，这样做才有好处。这暗示出锁争用得很厉害，在担心高速缓存争用之前先解决这个问题，能够改善系统的性能。

要看到图 15-7 中的代码是怎样影响高速缓存的，可以假定采用了 MESI 协议，而且假定按照 15.4 节讲过的那样处理原子操作。当一个 CPU 企图获得一个已经在使用的锁的时候，函数 test_and_set 让包含该锁的高速缓存行的一个独占副本被送入处理器的高速缓存。处理器发现锁被设置了，于是执行内层循环，直到锁被释放为止。如果和锁相关联的数据结构占用了一个不同的高速缓存行，那么占据锁的处理器不会受到它的直接影响，因为其他处理器在获得锁之前不会引用这个数据结构。同时，试图获得锁的处理器会在锁状态字上连续执行上载操作。因为它已经在其高速缓存中有了对应的行，所以不会产生总线通信量。这是理想的情形，因为自旋不会浪费总线带宽或者导致争用，而且这是使用硬件高速缓存一致性协议的明显优点。如果与此同时，第二个 CPU 试图获得锁，它就获得该高速缓存行的一份独占副本，然后也进入内层循环。当第一个 CPU 在内层循环上执行的时候，它将发生高速缓存缺失（因为第二个 CPU“偷”了高速缓存行），并且获得该行的一个共享副本，因为两个 CPU 都是只读该行。现在两个 CPU 都使用高速缓存行的共享副本进行自旋，不会再进一步产生总线通

信量。

当最后锁被释放的时候，函数 unlock 更新锁字，这让一致性协议获得了该行的一个独占副本，并且使其他可能存在的副本都变成无效。这就让为锁而自旋的处理器在下一次执行内层循环时发生缺失。第一个获得总线的处理器从释放锁的处理器那里监听到该行已经更新过的副本，而且发现锁是空闲的，它随后就尝试为它自己得到锁。通过在获得高速缓存行的一份独占副本期间屏蔽中断和其他延迟因素，它最有可能在其他处理器之前得到锁。正等候锁的其他处理器在最后能够重新获得高速缓存行的一个副本时，将会发现它仍然是锁住的，于是它们会继续在它们缓存的副本上自旋。这一切的优点是，虽然要试图获得锁，但是往往只有在锁状态发生变化的时候，才会产生总线交易，让处理器在它们其中之一实际获得锁期间，为锁而展开竞争。

15.9.3　一致性协议与数据用途相匹配

有些处理器（如 MIPS R4000）允许为不同的存储页面使用不同的高速缓存一致性协议。将协议与页面内数据的访问方式相匹配，可以减少不必要的总线通信量，并且提高系统性能。R4000 支持 5 种不同的协议：不缓存（uncached）、不一致（noncoherent）、独占（exclusive）、共享（shared）和更新（update）。

不缓存模式迫使针对页面的所有上载和保存操作都要做标记，绕过高速缓存，直达主存储器。处理器的私有高速缓存从不做命中检查，系统内其他的高速缓存不监听这些上载和保存操作产生的总线交易。这种模式主要是在访问 I/O 设备控制和状态寄存器的时候使用。正如在 3.3.2 小节里指出的那样，大块的复制操作也能享受到不缓存的好处。如果已知从复制操作的源和目的地来的数据都不驻留高速缓存，那么将不缓存访问用于复制操作就能防止其他有用的数据被强行送出局部高速缓存。因为在这样的操作期间没有监听，所以它必须得到保证，要么没有数据的缓存副本存在，要么显式地使它们无效，从而保持高速缓存一致性。

不一致模式让处理器检查它的私有高速缓存，看有没有命中，但是当发生缺失的时候，它直接从主存储器读取数据，而不监听系统内的其他高速缓存。在这方面，高速缓存的运行就仿佛它是在一个单处理器系统（在这类系统中安装 R4000 时适于采用的模式）里一样。对于访问特殊处理器所私有的高速缓存来说，这种模式很有用。它减少了总线通信量，因为在首次修改一个有效行时，不需要广播使无效命令。而且监听机制能够干涉 CPU 对高速缓存的访问，所以减少了需要监听的交易数量，也就减少了争用。处理器私有数据的例子包括，诸如在那个 CPU 的异常和中断处理期间所使用的独立堆栈（每个 CPU 都有它自己的堆栈）、每个 CPU 的统计信息、局部运行队列和任何其他私有数据结构这样的东西。它对内核正文（假定内核不使用自我修改的代码）也很有用。它可以用于普通的用户数据（数据、bss 和堆栈，但不包括共享内存），但是如果进程从一个处理器迁移到另一个处理器，那么必须实施显式的冲洗操作。只要用户正文是只读的，就可以使用它。

独占高速缓存模式的操作就好像是一个 MESI 协议，但是没有共享状态。这里，在独占页面内的所有数据都要由硬件来保持高速缓存一致性，但是在任何时刻，只有一个高速缓存可以有一行的一个副本。所以，如果一个 CPU 因上载操作在它的高速缓存里发生缺失，那么其他高速缓存监听总线交易；如果在一个高速缓存中命中，它就把数据返回给发生缺失的高

速缓存，并且使它自己的副本无效。这种模式对于随进程迁移的数据（如进程的 u 区和内核堆栈）有用处。例如，当一个进程迁移时，它可能让 u 区数据缓存在原来的 CPU 上。当它在新的 CPU 上继续执行并且读取这一数据的时候，它能获得的数据处于这样的状态，即能够修改数据而不会再进一步产生总线通信量。另一方面，在这种情况下，使用 MESI 需要两次总线交易：一次是在首次发生缺失的时候读取该行的一个共享副本，一次是在数据被修改的时候使原来的副本无效。因为 u 区和内核堆栈都只会由当前正在运行进程的处理器引用（在调试期间可能有例外），所以独占模式带来的总线交易很少。它也很适合非共享的用户数据（数据、bss 和堆栈），而且能够省去采用不一致模式时所需的显式冲洗操作。

共享模式实现了一种 MESI 协议，它适用于大多数内核数据结构和用户级共享内存。维护一致性采用写-使无效机制。对于可能会由不同处理器上多个进程读取的数据结构来说，它的效果最好，因为行的共享副本可能由每个 CPU 缓存。

最后，更新模式实现了一种写-更新协议。多个高速缓存可能保留有给定数据的共享副本。任何时候只要有一个 CPU 向该行写，那么如果其他 CPU 有该行的副本，它就在总线广播更新。因为它会因每次保存操作而带来一次总线交易，所以这种模式要小心使用。它最适合于始终要由多个 CPU 读取的数据。在这种情况下，同共享模式相比，它减少了总线通信量，因为要更新所有副本只要一次总线交易就够了。除了对数据进行保存操作的那个处理器之外，共享模式会让所有的副本都无效，这接着又要求每个处理器发生缺失，执行一次总线交易读取新的值。适于使用更新模式的数据的例子包括，保存时间的数据以及运行队列的数据结构（如果系统使用一个全局运行队列的话）。此外，把锁保存在使用更新模式的页面内，也能提高性能。如果多个处理器在一个位于共享高速缓存行内的锁上自旋，如图 15-7 中代码的内层循环，那么当发出保存操作来释放锁的时候，其他所有副本都被更新而不是使之无效。如前所述，这样做比 MESI 省总线交易。一旦锁被释放，第一个获得总线的处理器会发出第二个更新，设置锁，这也比写-使无效协议节省总线通信量。

在没有 R4000 那么多选择的系统上，让协议和数据的需要相匹配也是有帮助的。考虑 15.2.1 小节中介绍的 MC68040，它所支持的唯一一种硬件高速缓存一致性协议是写直通-使无效协议。写直通的缺点是每次写操作它都要产生一次总线交易。68040 允许逐页选择是采用写直通还是写回策略，在使用写回策略时不保证高速缓存一致性。但是，在 R4000 上采用不一致模式的情况下，能够使用写回策略。如果用于非共享的用户数据、u 区和内核堆栈，那么和写直通策略相比，它可以减少总线通信量。从长期来看，在进程迁移时显式地冲洗高速缓存，代价可能不会太大，尤其 68040 的片上数据高速缓存仅为 4K 的情况下更是如此。

15.10 小 结

硬件高速缓存一致性不需要由软件显式地进行冲洗，就能保持 MP 系统上高速缓存和主存储器间的数据一致性。在基于总线的系统上，这是通过让每个高速缓存监听总线上的交易，根据其他处理器的活动采取适当的措施来实现的。因为总线上使用物理地址，所以只有物理索引的高速缓存才能使用硬件一致性机制。监听和与之相关的操作则对软件透明。

高速缓存一致性协议指出了高速缓存何时需要进行通信，以及它们应该采取什么样的行

动。有两大类这样的协议：写-使无效和写-更新。这两种协议都允许多个高速缓存占有相同数据的共享只读副本区别在于写操作期间所采取的行动。写-使无效协议在一个 CPU 写数据的时候使其他所有的缓存副本无效。这就迫使其他处理器在它们下一次访问的时候发生缺失，并且从修改数据的高速缓存或者主存储器读取更新过的数据。写-更新协议在总线上广播对共享数据的每次写操作，于是其他所有带有该数据副本的高速缓存都能更新它们的内容。在两种情况的任何一种中，当一个 CPU 修改过时的高速缓存数据时，这些数据就被清除。这些相同的动作也保持了原子存储器操作的一致性。

没有一条公共共享总线的系统（如那些使用交叉开关存储互连的系统）仍然能够通过目录来保持高速缓存的一致性。目录和主存储器相关联，记录着哪个 CPU 正缓存其数据的副本以及高速缓存的状态。这能够让存储器模块决定，为了保持一致性，哪些操作是必要的。接着，它负责和适当的高速缓存进行通信，获得修改过的数据（在采用写回高速缓存机制的情况下），或者使其他高速缓存写过的数据无效，或者更新它们。

为了保持一致性，不同的高速缓存一致性协议产生的总线通信量也会有变化。每种协议最适合于某些特定的情形。如果处理器支持不同的高速缓存一致性协议，那么操作系统就可以使之匹配对数据的期望访问模式。这就减少了总线通信量，因而争用也更少。此外，通过利用协议的工作原理，内核开发人员能够调整内核的算法，以减少总线通信量。这在诸如要获得自旋锁之类的情况下特别有用，因为自旋能够产生快速的存储器操作流。类似地，可以将数据结构对齐高速缓存的边界，并进行填充，以消除额外的缺失和错误共享造成的弹来弹去效应。

最终的结果是，消除了上一章里介绍的保持高速缓存一致性的软件开销。例如，可以缓存用户数据，不必在进程迁移的时候冲洗它。可以缓存自旋锁，在释放保护内核数据结构的锁时，内核不必显式地把它的数据结构写回存储器或者使之无效。同样，效果和单处理机系统上的物理高速缓存是一样的：它们对于软件是透明的。

15.11 习　题

15.1　考虑一个使用写一次协议的 MP 系统，在一次上载操作发生高速缓存缺失的时候，监听操作命中了另一个高速缓存中一个已修改过的行，除了把它发送到发生缺失的高速缓存之外，它一定还要把这行写回主存储器。既然已修改过的数据没有丢失（两块高速缓存现在都有一个副本），为什么必须把修改过的数据写回主存储器？

15.2　如果没有保留状态的话，写一次协议还能发挥作用吗（也就是说，正确地保持高速缓存一致性）？

15.3　在 15.4 节中说过，在使用写回协议的时候，原子的读-改-写指令可能不必访问主存储器。如何将它用于写一次协议？考虑两种情况：当原子操作开始的时候，行处于有效状态；当原子操作开始的时候，行处于保留状态。假定在原子操作正在进行的时候，系统上没有别的处理器访问该行。

15.4　阐述在两个或者两个以上的 CPU 同时对一个共享行执行原子读-改-写操作时（也就是说，所有执行原子操作的处理器都有该行的一个共享副本），Firefly 协议是如何保证一

致性的。

15.5 考虑用于写回高速缓存机制的一种简单协议。除了监听之外，均使用正常的写回语义（也就是说，它不使用写一次）。如果来自一个处理器的上载操作命中了另一个处理器的高速缓存内的重写行，那么那个高速缓存就要提供这行的内容（因为存储器是过时的），同时更新存储器，并且使它自己的副本无效。如果来自一个处理器的保存操作命中了另一个处理器的高速缓存（不管该行的状态如何），那么那个高速缓存就要使它自己的副本无效。这种协议能够正确地保持高速缓存的一致性吗？

15.6 给习题 15.5 中说明的协议绘制一张状态图。

15.7 在 15.9.2 小节中说过，在一个使用 MESI 协议的系统上，当为一个锁而自旋时，会出现 ping-pong（弹来弹去）现象。对于使用写-更新协议的系统来说，在处于忙等待的同时只是读一下锁的状态，而不是使用原子操作的解决方法是一个好主意吗？回忆一下，在一个 CPU 修改数据行的时候，写-更新协议没有使之无效，这样的做法防止了弹来弹去现象的出现。

15.8 考虑 9.4 节介绍的主从处理机 MP 内核实现。如果系统使用 MIPS R4000，你将给每个内核数据结构使用 15.9.3 小节所述的哪一种高速缓存协议？你给用户正文、数据、堆栈、bss 和共享内存使用哪些协议？解释你的选择。

15.9 对于 11.3 节所介绍的信号量，数据结构 semaphore 应该怎样在高速缓存中对齐？应该怎样在高速缓存中填充？应该把自旋锁保存在同一高速缓存行内吗？如果信号量保护的是一个临界资源（和用于同步目的相对照），应该如何？应该把信号量和它所保护的资源放入同一行中吗？

15.10 绘制写直通-使无效协议的状态图。

15.11 设想这样的一个系统，其中一级高速缓存是带有键的虚拟高速缓存，二级高速缓存是物理高速缓存。有可能由硬件来保持两者的一致性吗？如果能，这该怎样做？假定系统使用基于总线的监听机制。

15.12 考虑这样的一个系统，它带有的二级高速缓存支持总线监视，但是其中的一级高速缓存不被监听，也不遵从包含属性。为了保持一致性，软件必须要做什么？

15.12 进一步的读物

[1] Agarwal, O.P., and Pohm, A.V., "Cache Memory Systems for Multiprocessor Architectures," *Proceedings of the AFIPS National Computer Conference*, June 1977.

[2] Agarwal, A., Simoni, R., Hennessy, J., and Horowitz, M., "An Evaluation of Directory Schemes for Cache Coherence," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988, pp.280-9.

[3] Archibald, J., and Baer, J.-L., "An Economical Solution to the Cache Coherence Problem," *Proceedings of the 11th International Symposium on Computer Architecture*, June 1984, pp.355-62.

[4] Archibald, J., and Baer, J.-L., "Cache Coherence Protocols: Evaluation Using a

Microprocessor Simulation Model," *ACM Transactions on Computer Systems*, Vol. 4, No. 4, November 1986, pp.273-98.

[5] Atkinson, R.R., and McCreight, E.M., "The Dragon Processor," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987,pp.65-71.

[6] Baer, J.-L., and Wang, W.H., "Architectural Choices for Multi-Level Cache Hierarchies," Technical Report TR-87-01-04, Department of Computer Science, University of Washington, January 1987.

[7] Baer, J.-L., and Wang, W.H., "On the Inclusion Property for Multi-Level Cache Hierarchies," Technical Report TR-87-11-08, Department of Computer Science, University of Washington, November 1987.

[8] Barroso, L.A., and Dubois, M., "The Performance of Cache-Coherent Ring-based Multiprocessors," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp.268-78.

[9] Bitar, P., and Despain, A.M., "Multiprocessor Cache Synchronization Issues, Innovations, and Evolution," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 1986, pp.424-33.

[10] Bolosky, W.J., and Scott, M.L., "False Sharing and its Effect on Shared Memory Performance," *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, September 1993, pp.57-71.

[11] Censier, L.M., and Feautrier, P., "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, Vol. C-27, No. 12, December 1978, pp.1112-8.

[12] Chaiken, D., Fields, C., Kurihara, K., and Agarwal, A., "Directory-Based Cache Coherence in Large-Scale Multiprocessors," *IEEE Computer*, Vol. 23, No. 6, June 1990.

[13] Colglazier, D.J., "A Performance Analysis of Multiprocessors Using Two-Level Caches," Technical Report CSG-36, Computer Systems Group, University of Illinois, Urbana-Champaign, August 1984.

[14] Cox. A.L., Fowler, R.J., "Adaptive Cache Coherency for Detecting Migratory Shared Data," *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993, pp.98-108.

[15] Dewan, G., and Nair, V.S.S., "A Case for Uniform Memory Access Multiprocessors," *Computer Architecture News*, Vol. 21, No.4, September 1993, pp.20-6.

[16] Dubois, M., and Briggs, F.A., "Effects of Cache Coherence in Multiprocessors," *Proceedings of the 9th International Symposium on Computer Architecture*, May 1982, pp. 299-308.

[17] Dubois, M., and Wang, J.-C., "Shared Data Contention in a Cache Conherence Protocol," *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988, pp. 146-55.

[18] Dubois, M., and Scheurich, C., "Memory Access Dependencies in Shared-Memory

Multiprocessors," *IEEE Transactions on Software Engineering*, Vol. 16, No. 6, June 1990, pp.660-73.

[19] Dubois, M., Skeppstedt, J., Ricciulli, L., Ramamurthy, K., and Strenstrom, P., "The Detection and Elimination of Useless Misses in MPs," *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993, pp.88-97.

[20] Eggers, S., and Katz, R., "Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988, pp.373-83.

[21] Eggers, S., and Katz, R., "Evaluating the Performance of Four Snooping Cache Coherency Protocols," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pp.2-15.

[22] Eggers, S.J., and Jeremiassen, T.E., "Eliminating False Sharing," *Proceedings of the 1991 International Conference on Parallel Processing*, August 1991, pp.A:377-81.

[23] Gharachorloo, K., Gupta, A., and Hennessy, J., "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp.245-57.

[24] Goodman, J.R., "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings of the 10th Annual International Symposium on Computer Architecture*, June 1983,pp. 124-31.

[25] Goodman, J.R., "Coherency for Multiprocessor Virtual Address Caches," *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987, pp.72-81.

[26] Goodman, J.R., Vernon, M.K., and Woest, P.J., "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp.64-75.

[27] Goodman, J.R., and Woest, P., "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp.422-31.

[28] Greenberg, A.G., Mitrani, I., and Rudolph, L., "Analysis of Snooping Caches," *Proceedings of Performance 87, 12th International Symposium on Computer Performance*, December 1987.

[29] Gupta, A., and Weber, W., "Cache Invalidation Patterns in Shared Memory Multiprocessors," *IEEE Transactions on Computers*, Vol. 41, No. 7, July 1992, pp.794-810.

[30] Handy, J., *The Cache Memory Book*, Boston, MA: Academic Press, 1993.

[31] Katz, R.H., Eggers, S.J., Wood, D.A., Perkins, C.L., and Sheldon, R.G., "Implementing a Cache Consistency Protocol," *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 1985, pp.276-83.

[32] Lee., J., and Ramachandran, U., "Synchronization with Multiprocessor Caches," *Proceedings of the 17th International Symposium on Computer Architecture*, 1990,pp.27-37.

[33] Lorin, H., *Introduction to Computer Architecture and Organization*, Second Edition, New York: John Wiley & Sons, 1989.

[34] Mano, M.M., *Computer System Architecture*, Third Edition, Englewood Cliffs, NJ: Prentice Hall, 1993.

[35] Monier, L., and Shindu, P., "The Architecture of the Dragon," *Proceedings of the 13th IEEE International Conference*, February 1985, pp. 118-21.

[36] Norton, P.L., and Abraham, J.L., "Using Write Back Cache to Improve Performance of Multiuser Multiprocessors," *Proceedings of the International Conference on Parallel Processing*, 1982.

[37] Papamarcos, M.S., and Patel, J.H., "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proceedings of the 12th International Symposium on Computer Architecture*, June 1985, pp.348-54.

[38] Prete, C.A., "A New Solution of Cache Coherence Protocol for Tightly Coupled Multiprocessor Systems," *Microprocessing and Microprogramming*, Vol. 30, 1990, pp.207-14.

[39] Ravishankar, C.V., and Goodman, J., "Cache Implementations for Multiple Processors," *IEEE Spring Compcon Conference*, February 1983.

[40] Rudolph, L., and Segall, Z., "Dynamic Decentralized Cache Schemes for MIMD Paraller Architectures," *Proceedings of the 11th International Symposium on Computer Architecture*, 1984, pp.340-7.

[41] Sites, R.L., and Agarwal, A., "Multiprocessor Cache Analysis Using ATUM," *Proceedings of the 15th International Symposium on Computer Architecture*, June 1988.

[42] Stenstrom, P., "Reducing Contention in Shared-Memory Multiprocessors," *IEEE Computer*, Vol. 21, No. 11, November 1988, pp.26-37.

[43] Stenstrom, P., "A Cache Consistency Protocol for Multiprocessors with Multitage *Networks," Proceedings of the 16th Annual International Symposium on Computer Architecture*, May 1989, pp.407-15.

[44] Stenstrom, P., "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer*, June 1990, pp.12-24.

[45] Stenstrom, P., Brorsson, M., and Sandberg, L., "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing," *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993, pp.109-118.

[46] Stone, H.S., *High Performance Computer Architecture*, Thire Edition, Reading, MA: Addison-Wesley, 1993.

[47] Sweazey, P., and Smith, A.J., "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus," *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp.414-23.

[48] Tang, C.K., "Cache System Design in the Tightly Coupled Multiprocessor System,"

Proceedings of the National Computer Conference (AFIPS), June 1976, pp.749-53.

[49] Thacker, C.P., "Cache Strategies for Shared-Memory Multiprocessors," *New Frontiers in Computer Architecture*, March 1986, pp.51-62.

[50] Thacker, C.P., and Stewart, L.C., "Firefly: A Multiprocessor Workstation," *Proceedings of the Second International Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1987, pp.164-72.

[51] Tomasevic, M., and Milutionvic, V., *The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions*, Los Alamitos, CA: IEEE Computer Society Press, 1993.

[52] Torrellas, H., Gupta, A., and Hennessy, J., "Characterizing the Caching and Synchronization Performance of an MP Operating System," *SIGPLAN Notices*, Vol. 27, No. 9, September 1992, pp.162-74.

[53] Vernon, M.K., Jog, R., and Sohi, G.S., "Performance Analysis of Hierarchical Cache-Consistent Multiprocessors," *Performance Evaluation*, August 1989, pp.287-302.

[54] Wilson, Jr., A.W., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987, pp.244-52.

[55] Yen, W.C., Yen, W.L., and Fu, K.-S., "Data Coherence Problem in a Multicache System," *IEEE Transations on Computer*, Vol. C-34, No, 1, January 1985, PP.56-65.

体系结构汇总

附录

本附录提供了处理器和系统的高速缓存组织结构的一份汇总，它们在本书里供举例使用。本附录还包括了有关 MP 高速缓存一致性协议和原子指令的 MP 信息。除了 TI SuperSPARC 之外，所有的处理器都使用顺序存储器模型（强定序）。

Apollo DN 4000

在 20 世纪 80 年代后期生产的一种单处理机系统，它使用 Motorola 68020。该处理器没有片上高速缓存。外部高速缓存的组成为：

高速缓存类型：带有键的虚拟高速缓存，统一的
高速缓存大小：8K
组大小：直接映射
行大小：4 字节
键大小：3 位
写策略：采用写分配机制的写直通
替换策略：直接映射
备注：标记包含修改位和用户-内核位
参考文献：Frink, C. R., and Roy, P.J., "The Cache Architecture of the Apollo DN 4000," Proceedings of the Spring COMPCON, March 1988, pp. 300-2.

Intel 80386

没有片上高速缓存

Intel 80486

高速缓存类型：物理高速缓存，统一的
高速缓存大小：8K
组大小：4 路组相联
行大小：16 字节
写策略：写直通，没有采用写分配机制
替换策略：伪 LRU
MP 一致性协议：写直通-使无效

原子指令：交换-原子（助记名：xchg），也能为一个不能分隔的操作序列锁住总线（助记名：lock、unlock）

参考文献：Intel 486 Microprocessor Family Programmer's Reference Manual, Intel order #240486, 1992.

Intel 82495DX

配合Intel 80486使用的外部高速缓存控制器。

高速缓存类型：物理高速缓存，统一的

高速缓存大小：128K、256K或者512K

组大小：双路组相联

行大小：16、32、64或者128字节

写策略：采用写分配机制的写回

替换策略：LRU

MP一致性协议：MESI

参考文献：Introduction to the 50MHz Intel 486 DX CPU-Cache Subsystem Architectural Overview, Intel order #241085.

Intel Pentium

高速缓存类型：物理高速缓存，独立的指令和数据高速缓存

高速缓存大小：每个都是8K

组大小：双路组相联

行大小：32字节.

写策略：可以选择采用写分配机制的写回或者写直通

替换策略：LRU

MP一致性协议：MESI

原子指令：交换-原子（助记名：xchg），也能为一个不能分隔的操作序列锁住总线（助记名：lock、unlock）

参考文献：Pentium Processor User's Manual Volumn 3: Architecture and Programming Manual, Intel order #241430, 1993.

Intel i860 XR

高速缓存类型：虚拟高速缓存，独立的指令和数据高速缓存

高速缓存大小：8K数据高速缓存，4K指令高速缓存

组大小：双路组相联

行大小：32字节

写策略：采用写分配机制的写回

替换策略：LRU

MP一致性协议：无

原子指令：总线能为一个不能分隔的操作序列而被锁住（助记名：lock、unlock）
参考文献：Intel i860 XR 64-Bit Microprocessor, Intel order #240296, 1992.

Intel i860 XP

高速缓存类型：虚拟高速缓存，独立的指令和数据高速缓存
高速缓存大小：每种高速缓存都是 16K
组大小：4 路组相联
行大小：32 字节
写策略：可以选择采用写分配机制的写回、写直通或者写一次
替换策略：如果所有的行都有效，则随机
MP 一致性协议：MESI
原子指令：总线能为一个不能分隔的操作序列而被锁住（助记名：lock、unlock）
备注：高速缓存包含用于监听和别名检测的物理标记
参考文献：Intel i860 XP Microprocessor Data Book, Intel order #240874, 1991.

MIPS R2000/R3000

没有片上高速缓存，而是控制一个外部高速缓存。
高速缓存类型：物理高速缓存，独立的指令和数据高速缓存
高速缓存大小：每种高速缓存都是 64K
组大小：直接映射
行大小：4 字节
写策略：采用写分配机制的写回
替换策略：直接映射
MP 一致性协议：无
原子指令：无
参考文献：Kane, G., and Heinrich, J., MIPS RISC Architecture, Englewood Cliffs, NJ: Prentice Hall, 1992.

MIPS R4000

片上高速缓存：
高速缓存类型：带有物理标记的虚拟高速缓存，独立的指令和数据高速缓存
高速缓存大小：每种高速缓存都是 8K
组大小：直接映射
行大小：16 或者 32 字节
写策略：采用写分配机制的写回
替换策略：直接映射
外部高速缓存：
高速缓存类型：物理高速缓存，独立的指令和数据高速缓存，或者统一的高速缓存

高速缓存大小：从 128K 到 4M
组大小：直接映射
行大小：16、32、64 或者 128 字节
写策略：采用写分配机制的写回
替换策略：直接映射
备注：标记包含用于监听和别名检测的一级高速缓存索引
MP 一致性协议：可选择 MESI、写-更新、或者写-使无效独占协议（和没有 S 状态的 MESI 一样）
原子指令：load-linked（助记名：ll），store-conditional（助记名：sc）
参考文献：Heinrich, J., MIPS R4000 Users Manual, Englewood Cliffs, NJ: Prentice Hall, 1993.

Motorola 68040

高速缓存类型：物理高速缓存，独立的指令和数据高速缓存
高速缓存大小：4K
组大小：4 路组相联
行大小：16 字节
写策略：可以选择采用写分配机制的写回或者写直通
替换策略：如果所有的行都有效，则随机
MP 一致性协议：写直通-使无效
原子指令：测试和设置（助记名：tas），比较和交换（助记名：cas，cas2）
参考文献：Motorola 68040 32-Bit Microprocessor User's Manual, Phoenix, AZ: Motorola Literature Distribution, 1989.

Motorola 88000

88100 是 CPU 芯片，而 88200 包含 MMU 和片上高速缓存。一个完整的系统由一个 88100 和 2～8 个 88200 芯片所构成。在最少的情况下，需要一个用于指令的 88200 和一个用于数据的 88200，带来独立高速缓存的效果。每个 88200 包含一块片上高速缓存。
高速缓存类型：物理高速缓存
高速缓存大小：16K
组大小：4 路组相联
行大小：16 字节
写策略：可以选择采用写分配机制的写一次、或者写直通
替换策略：伪 LRU
MP 一致性协议：写一次
原子指令：交换-原子（助记名：xmem）
参考文献：Motorola MC88100 RISC Microprocessor User's Manual, Englewood Cliffs, NJ: Prentice Hall, 1990.

Motorola MC88200 Cache/Memory Management Unit User's Manual, Englewood Cliffs, NJ: Prentice Hall, 1990.

Sun 3/200

一个基于 Motorola 680X0 的单处理机系统，产于 20 世纪 80 年代后期，没有片上高速缓存。外部高速缓存的组成为：

高速缓存类型：带有键的虚拟高速缓存，统一的高速缓存

高速缓存大小：64K

组大小：直接映射

行大小：16 字节

键大小：3 位

写策略：采用写分配机制的写回

替换策略：直接映射

参考文献：Cheng, R., "Virtual Address Cache in UNIX," Proceedings of the Summer USENIX Conference, June 1987, pp. 217-24.

Texas Instruments MicroSPARC

数据高速缓存：

高速缓存类型：物理高速缓存

高速缓存大小：2K

组大小：直接映射

行大小：16 字节

写策略：没有采用写分配机制的写直通

替换策略：直接映射

指令高速缓存：

高速缓存类型：物理高速缓存

高速缓存大小：4K

组大小：直接映射

行大小：32 字节

写策略：n/a

替换策略：直接映射

MP 一致性协议：无

原子指令：交换-原子（助记名：xmem）

参考文献：Texas Instruments TMS390S10 MicroSPARC Reference Guide, 1992.

Texas Instruments SuperSPARC

片上数据高速缓存：

高速缓存类型：物理高速缓存

高速缓存大小：16K

组大小：4 路组相联

行大小：32 字节

写策略：当使用 MCC 二级高速缓存控制器的时候，使用没有采用写分配机制的写直通，否则使用带有写分配机制的写回

替换策略：伪 LRU

片上指令高速缓存：

高速缓存类型：物理高速缓存

高速缓存大小：20K

组大小：5 路组相联

行大小：64 字节

写策略：n/a

替换策略：伪 LRU

MCC（多高速缓存控制器）外部高速缓存：

高速缓存类型：物理高速缓存，统一的高速缓存

高速缓存大小：512K、1M 或者 2M

组大小：直接映射

行大小：128 或者 256 字节

写策略：带有写分配机制的写回

替换策略：直接映射

MP 一致性协议：MESI

原子指令：交换-原子（助记名：swap、ldstub）

参考文献：Texas Instruments SuperSPARC User's Guide, 1992.

附录B 部分习题的答案

第1章

1.1 正文是共享只读的。因此，对正文段的保存操作将会导致出现一次保护错。在 fork 期间，决不能将写时复制机制用于正文。唯一允许修改它的时候是在程序调试期间，当调试器在正文中插入了一个断点的时候。此时，因为结束了正文共享，所以断点不会影响到执行相同程序的其他进程。

1.2 有可能通过回收不用的页来收缩一个进程的堆栈。根据约定，超出堆栈当前顶部的存储器的内容不作定义。在几乎所有的体系结构上，堆栈当前顶部的指针都保存在一个定义好的寄存器中。因此，内核可以读取它的内容，并且根据需要回收任何未用的页。如果情况是这样，即用户程序要重复调用的子例程需要给局部变量额外的空间（这是一种很容易发生的情况），那么在调用之间回收堆栈空间，随后又再分配出去，这样做非常费时。如果情况是在程序的生命期内，被调用的子例程只用到一次，而调用的其他子例程都不需要堆栈空间，那么如果内核出于其他目的而需要空间，它最终会把不用的那部分空间通过调页交换出去。在这两种情况的任何一种中，无法保证有能识别出可以收缩堆栈的情形所需要的额外逻辑条件。

1.3 在进程 C 执行 fork 之后，映射关系如图 B-1 所示（所有的映射关系都是只读的）。C 的子进程以写时复制方式和其他两个进程共享相同的物理页面。如果进程 C 要在 fork 出它的子进程之后立即退出，那么进程 A 的映射和进程 C 的子进程的映射不会受到影响。一旦一个进程有了到页面的写时复制映射，那么当进程本身要修改该页面的时候，只要改成读-写方式就可以了。其他进程的活动不会受到影响。

1.4 如果采用写时复制方式，那么不必分配物理页面，因为新的子进程会和父进程共享页面。如果不采用写时复制方式，那么父进程的整个地址空间，除了正文区之外，都必须复制给子进程，因为它总是共享只读的。因此，如果不使用写时复制方式，那么需要给子进程的数据、bss 和堆栈段 7 个物理页面。

1.6 一次被缓冲的读操作的最后一步是把所请求的数据从内核的缓冲区复制到用户的缓冲区。在内核这么做的时候，会发生一次保护错，因为该页面是共享只读的。子进程将会分配得到一份受到影响的页面的新副本，同 read 相关联的复制操作就完成了。主要的一点是，不管是用户还是内核修改了采用写时复制方式共享的页面，结果都是一样的。在两种情况下都会出现相同的处理过程。如果代之以执行一次系统调用 write，那么什么也不需要做，因为

它就能让数据从用户的缓冲区复制到内核的缓冲区。不需要对用户的数据进行修改，所以还是不会影响到写时复制共享机制。

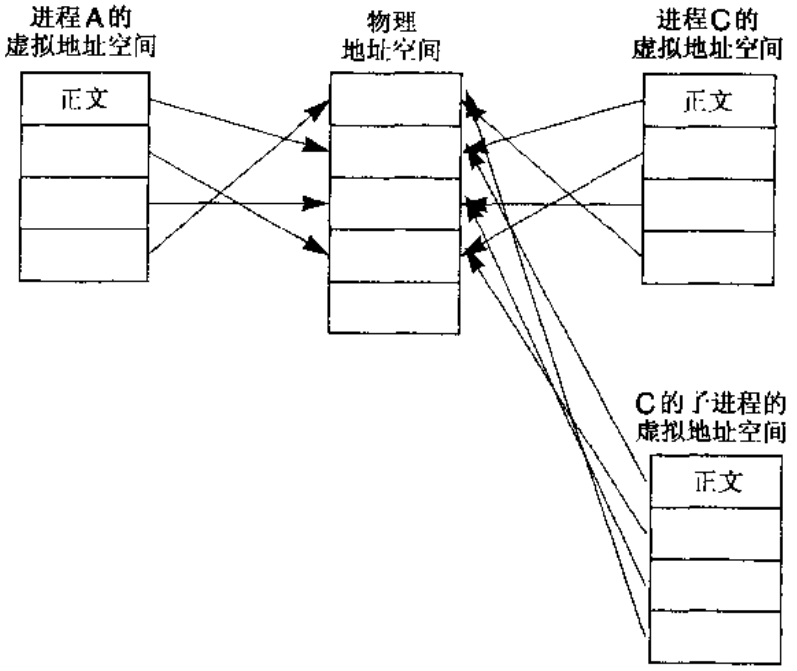


图 B-1 在进程 C 执行 fork 之后的存储器映射关系

1.10 当子进程释放其 bss 段内的页面时，那些页面上的写时复制共享机制就停止了。当子进程将断开地址恢复到它以前的值时，内核会给子进程分配新的虚拟内存，这和父进程的页没有关系。即使虚拟地址是相同的，子进程也不会重新获得父进程的页面上的写时复制共享机制。当子进程访问或者修改这些页面的时候，将会分配新的物理内存。

第 2 章

2.1 应该用不缓存的引用方式访问任何映射到存储器中的硬件设备。这包括 I/O 设备的控制和状态寄存器，以及诸如视频和图形帧缓冲这样的设备。在存储器中的任何不稳定的数据类型（CPU 和系统内其他硬件能够同时修改的数据）也应该用不缓存的引用方式访问。

2.2 用于构建高速缓存的存储设备在加电时其内容是随机的，因此，标记的有效位、地址部分以及行的数据部分都包含着无意义的、随机的数值。如果 CPU 要引用一个地址，而该地址碰巧匹配某行内的标志，这行的有效位又是 on，那么就会发生一次命中，高速缓存将返回行内数据部分的随机数据，而不是进入主存储器。这是一种极其糟糕的情形，因为高速缓存每次加电时都带有不同的内容，从而让其行为不可预测，也不确定。当系统加电启动时，在使用高速缓存之前，硬件或者是软件必须将其中的有效位都清除，从而使行内的随机内容不会影响到系统。

2.3 每行都需要一个标记，标识其内容的主存储器地址。许多不同的地址经散列算法计算出的索引都是一样的。因此，需要由标记来唯一地确定行的内容。同样，在一组内的每一

行也需要它自己的标记。重要的是要知道，高速缓存行内的连续字节始终是从存储器上连续的位置来的，因此不需要单个标记，在连续高速缓存行里的地址之间没有这样的关系。

2.4 在<9..2>内只有8比特位，说明散列算法只能索引2^8或者256个高速缓存行。因为高速缓存有512行，而且是直接映射的，有一半的行从来都没有被索引到，完全是浪费掉了。给512个高速缓存行寻址需要9比特位的索引。如果高速缓存是双路组相联的，那么如果高速缓存里每行4字节，其中用<1..0>选择行内的字节，这就是一种不错的散列算法。

2.6 它不是一种好的散列算法，因为一般的程序只能使用64个高速缓存行。这是因为高速缓存索引是从0x1000到0x4fff的地址的最高两位数字产生的。这些地址索引了从0x10到0x4f的行，总计64行。这意味着高速缓存中只有1024字节（64行*16字节/行）用于对16K区域的全部引用，而让15K高速缓存被浪费了。把组的大小增加到16行，就能消除这个问题，因为这就把为每个高速缓存索引保存的行数增加了16倍。现在整个16K存储区都映射到16K高速缓存内的一行里了。

2.7 有4096行的双路组相联高速缓存将有2048组。因此，散列算法必须产生一个介于0~2047之间的索引，它需要11位。因为行的大小是16字节，所以地址的低4位（位<3..0>）选择行内的字节，为了让数据在高速缓存内获得最好的分布，假定有良好的局部引用特性，应该使用剩下来的高端11位。所以，位<14..4>是用于散列算法的最好选择。行的标记部分必须有尚未使用的全部位。这就是位<31..15>，所以需要17位。

节省的百分比计算如下。首先，使用刚才给出的数据计算全部高速缓存大小。每行包含16字节数据（16字节×8位/字节＝128位）、11位的地址标记、1位有效位以及1位修改位（因为它是一块写回高速缓存），每行总共是141位（这个问题的重点在于，没有忽略有效位和修改位）。有4096行，所以4096×141＝577 536位。如果整个地址都保存在标记中，那么每行需要128位的数据位、32位的地址位、1位有效位以及1位修改位，每行总共是162位，整个高速缓存就是663 552位。因此，保存最少数目的地址位节省了大约13%的空间。

2.9 对于4字节的行：

$$\frac{2048\text{字节}}{4\text{字节/行}} = 512\text{行} = 2^9 \Rightarrow 9\text{位行索引}$$

因为用2个比特位索引行内的字节，那么索引行要用9+2＝11位地址，行标记中剩下还需要的32-11＝21位地址。高速缓存总共的大小为：

$$\begin{array}{r} 21 \text{ 标记中的地址位} \\ 1 \text{ 有效位} \\ +\,32 \text{ 数据位/行} \\ \hline 54 \text{ 位/行} \\ \times 512 \text{ 行} \\ \hline 27\,648 \text{ 位} \end{array}$$

对于32字节的行：

$$\frac{2048\text{字节}}{32\text{字节/行}} = 64\text{行} = 2^6 \Rightarrow 6\text{位行索引}$$

需要5个比特位选择行内的字节，所以索引已经用去6+5＝11位地址，标记中剩下32－11＝21位。这种情况下，高速缓存总共的大小为：

21	标记中的地址位
1	有效位
+256	数据位/行
278	位/行
×64	行
17 792	位

因此，从构建两种不同的高速缓存组织结构所需的比特位数上看，即使两者保存的数据量相同，使用32字节的行也会节省36%的比特位。这就体现出较长的行大小所拥有的好处之一：它减少了高速缓存行数，因此也就减少了用于控制和标记信息的比特位数。行大小较长的另一个好处是，对于有良好的空间局部性的程序来说，命中率更高。这对于指令引用来说是常见的情况，因为它们倾向于顺序执行。较长的行大小在每次缺失时，有效地预取了更多的顺序指令，与采用较小行大小的高速缓存相比，整体上减少了发生缺失的次数。

行大小较小的优点是，对于有良好的时间局部性但空间局部性不好的程序来说，命中率有可能会更高。这是因为高速缓存包含有更多单独索引和标记的行，意味着来自更多不同地址的数据能够同时被缓存。4字节的行有8倍于32字节的行数。

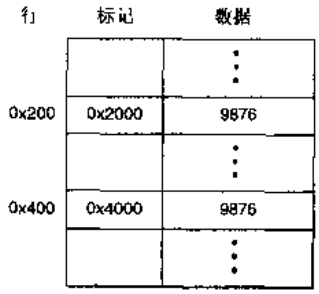
一般而言，32字节的行是首选，因为它减少了硬件成本，而且能够提高有着适度局部引用特性的程序的命中率。

2.13 所需的操作是一条指令，它无需读取主存储器的内容，就能让一个高速缓存行变成有效，并且用0予以填充。这在硬件上实现起来相当容易，因为它类似于正常的缺失处理。指令取得一个存储器位置的地址，而且如果该地址没有在高速缓存中命中的话，那么在行替换和写回方面会出现正常的缺失处理。但是那样一来，不是从主存储器里的那个地址读取数据，而是用0填充高速缓存，并且设置修改位和有效位。让程序的这部分内存清零不需要更进一步的操作。结果是，用一条指令就能让整个高速缓存行的数据都清零，而不需要引用存储器。TI SuperSPARC上的外部高速缓存控制器已经支持这种功能。

第3章

3.1 使用保存在标记中的地址，并且用行在高速缓存内的位置推断剩下的位，就能组成完整的32位地址。从标记来的前14位和被替换行的14位高速缓存索引合了起来。这就肯定产生出地址的前28位，因为只有那些“位<17..4>”索引了相关高速缓存行的地址才能保存在那里。既然这些高速缓存行包含有来自存储器的连续16字节，所以地址的低4位可以根据要存回主存储器的是哪些特定的字节或者字来生成。

3.2 a.高速缓存和主存储器是一致的。



b.高速缓存和主存储器不一致。

行	标记	数据
		⋮
0x200	0x2000	9876
		⋮
0x400	0x4000	1234
		⋮

3.3 与采用的是写直通高速缓存机制还是写回高速缓存机制无关；高速缓存保证一定会返回正确的结果。因为高速缓存是直接映射的，两个别名一定会索引到相同的一组高速缓存行，所以数据是一致的。在一个别名被缓存的同时引用另一个缓存，将会造成一次缺失，并且要从主存储器读取数据。如果采用写直通高速缓存机制，那么存储器始终是会更新的，所以发生缺失的时候会读取到正确的数据。如果采用写回高速缓存机制，那么当发生缺失的时候，任何已修改过的数据都会被写入主存储器，因此会重新读取到正确的数据来满足缺失。

3.8 因为地址空间已经在用户和内核之间进行了划分，所以可以使用高端比特位来标识哪部分地址空间能访问。如果一个运行在用户态的进程所引用的地址中将高端比特位设为置位（1），所以硬件在访问高速缓存之前就能立即产生一个陷阱。这就防止了进程访问被缓存的内核数据。因为保存在高速缓存中的用户数据把高端比特位清零了，所以这些数据也就绝对不会被误认为是内核数据了。

3.10 因为分配了新的虚拟内存，所以什么也不需要。从概念上说，这和 sbrk 分配新内存的时候是一样的。因为新分配的空间进程以前访问不到，所以它没有那个空间相应的缓存数据。

3.11 在大多数 UNIX 内核实现中，在把页面换出的时候，不需要对虚拟高速缓存进行什么操作。在典型情况下，内核使用一个独立的进程来处理换出工作。它根据引用历史选择要换出的页面，然后执行 I/O，把它们换出。因为这些活动是在一个独立的进程中进行的，所以在它们完成之前，必须至少出现一次现场切换。在现场切换时刻进行的高速缓存冲洗操作确保了在换出进程能够运行之前，主存储器的内容是最新的；因此，可以安全地执行从主存储器到交换设备的 DMA 操作。在换出进程允许其他进程再次运行之前，它要消除被换出的页面上的有效位。这就防止了一个其页面被换出的进程在 DMA 正在执行的时候访问它们。

3.14 如果一个可缓存的页面变成了不缓存的页面，那么必须冲洗高速缓存（使主存储器有效，而使高速缓存无效）。因为从页面来的数据以前是可缓存的，所以其中有一些在高速缓存中。把页面改成不可缓存的，可以防止以后发生的缺失所关联的数据被缓存，但是对于已经在高速缓存中的数据却无能为力。当 CPU 执行一次 load 或者 store 时，大多数实现都会在 MMU 能够在页表中检查可缓存性标志之前，先访问高速缓存。如果发生一次命中，那么就立即把数据返回给 CPU。因此，虽然页面被标记为不缓存，但是来自页面的缓存数据将仍然被缓存，直到它被替换为止。

第 4 章

4.1 当计算标记的地址部分所需位数的时候，键的大小没有关系。键的用途是，消除来自不同进程的虚拟地址之间的歧义。高速缓存仍然要识别保存在行内的全部地址。所以“位<3..0>”将选择行内的字节，接下来的 9 位（“位<12..4>”）选择组（1024 行÷2 行/组=512 组=2^9）。那要用掉地址的 4+9=13 位，剩下 19 位没有用（“位<31..13>”）。因此，标记的地址部分需要 19 位。这个位数是恒定的，与键的大小无关。

4.2 对于带有键的虚拟高速缓存来说，一种很有用的冲洗操作是，能够按照键来执行冲洗。有了这样的操作，就可以指定一个键，然后将那个键标记的所有行都写回主存储器（对于一个采用写回策略的高速缓存来说），并且（或者）使它们在高速缓存中变得无效。这在诸如 exec 或者 exit 这样的情况下非常有用，此刻正在回收整个地址空间，所以必须冲洗掉它。本章介绍的 Sun 系统就提供这种功能。

4.4 对于在这些情形下重新分配键的任务来说，LRU 是一种优秀的算法。近期最少使用（LRU）的键所拥有的高速缓存项最少，因为同时运行的所有进程可能已经让它的所有项都移出了高速缓存（除了高速缓存较大可能还没有移完之外）。因此，从缓存数据方面来看，拥有 LRU 键的进程失去的也最少。这要假定高速缓存有一种按地址冲洗（flush-by-address）或者按键冲洗（flush-by-key）的能力，只有在地址或者键命中的时候才进行冲洗。当重新分配键，以保证老进程没有剩下缓存项的时候，仍然必须冲洗高速缓存。如果高速缓存只实现了按行冲洗（flush-by-line）功能，不管键和地址标记是什么，都会无条件地冲洗整整一行的话，那么使用 LRU 键就没有什么省事之处了。

4.6 在父进程和子进程采用写时复制共享地址空间内的所有页面时，它们也可能会共享相同的键。一旦一页首次进行了写时复制，那么必须停止共享键。此刻需要不同的键，因为两个进程的虚拟地址不再全部映射到同一组物理页上了。这样的技术使得系统调用 fork 运行得更快了，因为它在那时候不需要分配一个新键，或者冲洗高速缓存。但是，在第一次发生写时复制缺失的时候，就会带来开销。在子进程 exec 之前，两个进程都不修改其地址空间的情况微乎其微，所以不会看到切实的性能增益。

4.8 让内核来为 vfork 处理高速缓存问题是不太值得的。因为父进程和子进程完全共享了相同的地址空间，所以它们都使用相同的高速缓存键。这就允许子进程直接引用父进程已经缓存的任何数据，并且导致子进程带入高速缓存的任何数据都以父进程的键来标记，于是它将命中相同的数据。共享相同键的做法在写直通和写回方式下都能用，也适合于每组有任意数量高速缓存行的高速缓存。当子进程 exec 的时候，必须给它指派一个新键。如果子进程 exit，那么必须要跳过正常情况下进行的使高速缓存无效的操作，因为父进程仍然在使用该键。

4.11 因为子进程有它自己的地址空间（即使采用写时复制机制也是如此），所以内核必须给它分配一个唯一的 TLB 键。如果采用了写时复制机制，那么子进程一开始会接收到一份其父进程的映射关系的副本。提供给父进程和子进程一个唯一的键，就能够让每个进程独立地修改它自己的地址空间。此外，在启动写时复制共享机制的时候，父进程的映射关系要变为只读。因为在启用了写权限的情况下，TLB 会缓存映射关系，所以必须在父进程返回用户态之前冲洗掉这些缓存。

4.15 最简单的方法是在现场切换的时候把相同的键载入两个键寄存器。于是，高速缓存的行为和本章描述的行为是一样的。另一种可能是使用不同的键。因为指令高速缓存包含只读数据，共享相同正文的所有进程有可能将相同的键用于指令高速缓存。这样一来，每个共享的正文段都会分配得到一个键。每个进程仍然会在数据高速缓存中使用唯一的键。采用了这种方法之后，从共享正文段取得的指令将在使用该正文键的所有进程间共享，从而有可能提高指令高速缓存中的命中率。但是，这种方法有几个缺点。第一，如果高速缓存是直接映射的，而且共享正文段使用相同的虚拟地址空间，那么它们可能会在高速缓存中彼此替换，减少或者干脆是消除了潜在的提高。第二，如果进程一次使用一块以上的共享正文段，譬如，除了程序的正文之外，还使用一个或者多个共享库，那么程序使用的一组共享正文段必须在执行缓存上当作一个单元来对待。这是因为，只有一个键用于一个进程发出的所有指令预取。这样一来的结果是，共享库的正文会在高速缓存中重复，因为使用相同库的每个不同的程序从它那里执行的时候，都要有一个单独的键。最后，如果程序使用自我修改的代码，那么不能使用这种方法。如果它从其数据区读取代码，那么那些指令将会被缓存在指令高速缓存中，而且带有共享正文段的键。如果另一个进程正在执行相同的程序，那么它会命中来自其他进程地址空间的指令。

最后要说明的一点是，给每个进程分配两个唯一的键，每个高速缓存一个，这种做法是没有什么好处的。在两个高速缓存中使用相同的键就能获得相同的结果。

第5章

5.1 没有。带有物理标记的虚拟高速缓存是用虚拟地址索引，用物理地址标记的。全相联高速缓存不做任何索引，所以从虚拟地址得来的索引是没有意义的。在采用全相联高速缓存的情况下，唯一的问题是如何标记行。

5.2 采用这样的高速缓存组织结构，“位<3..0>”选择行内的字节。“位<11..4>”选择256组内的一组。因为所有这些位都是页偏移量（“位<11..0>”）的一部分，所以在虚拟地址和物理地址中，页偏移量是相同的，这块高速缓存应该看作是一块物理高速缓存。没有使用虚拟页号的比特位来索引高速缓存。

5.6 采用带有键的n路组相联虚拟高速缓存出现的问题要归因于别名。即使虚拟地址相同，每个进程也会使用不同的键，在写时复制起作用的情况下，这就让一个进程在引用由其他进程缓存的数据时产生一次命中。这意味着，可以引用组内的另一行。采用带有物理标记的虚拟高速缓存，可以保证以写时复制机制共享数据的两个进程彼此命中高速缓存里对方的数据，因为它们都使用相同的虚拟地址和物理地址。不会出现别名，所以组内的行数没有关系。

5.8 对，冲洗操作可以安全地延期。在把共享存储区或者映射文件从进程的地址空间剥离下来的时候，映射关系也会变化，这些页面再也访问不到了。再引用的话会在高速缓存访问完成之前从MMC引发一次陷阱。

5.9 无论何时在写回高速缓存中替换了一个已修改过的行，都必须把它写回到主存储器。在使用带有物理标记的虚拟高速缓存时，高速缓存能够生成物理地址，把数据写回存储器，而不需要使用MMU。因此，不会出现陷阱，因为高速缓存把数据保存到一个不在进程地址空间内的位置。既然没有使用被写回存储器的物理页面，那么就不会有破坏其他进程数据的危险。正如5.2.5小节中介绍的那样，在将这些页面用作新的用途之前，必须先冲洗高速缓存。

第 6 章

6.1 图 B-2 给出了一种可以接受的图。重要之处如下。VPN 内没有比特位用于索引高速缓存。必须先转换地址，以便能够将来自物理地址的“位<13..5>”用于索引。索引是将来自 PPN 的位和页偏移量（要牢记，在虚拟和物理地址中的页偏移量是相同的）合起来构成的。也没有必要像物理标记的虚拟高速缓存那样，把“位<13..11>”保存到标记中，因为在索引期间已经用了这些位，所以会发生冗余。

6.2 人们可以构建以这种方式起作用的系统，但是它并不可取。原因是系统在为进程传送数据的同时，往往还运行着不同的进程。将所有数据载入高速缓存的做法会牺牲当前执行进程的局部引用特性，从而使高速缓存的性能很差。最好让读数据的进程在它执行的时候在数据内建立局部引用特性。这样一来，当前没有局部引用特性的部分里的数据就不会占据高速缓存中宝贵的空间。

第二，这对于写回高速缓存来说实现起来非常困难。如果来自读操作的数据要替换一个已修改的行，那么必须把修改过的数据写回到主存储器。问题在于，总线已经由 DMA 操作使用了，那么要写回的数据在被写回之前，需要临时保存在别的什么地方。在一次 DMA 读操作期间会有许多替换这一事实又让这个问题更为复杂。

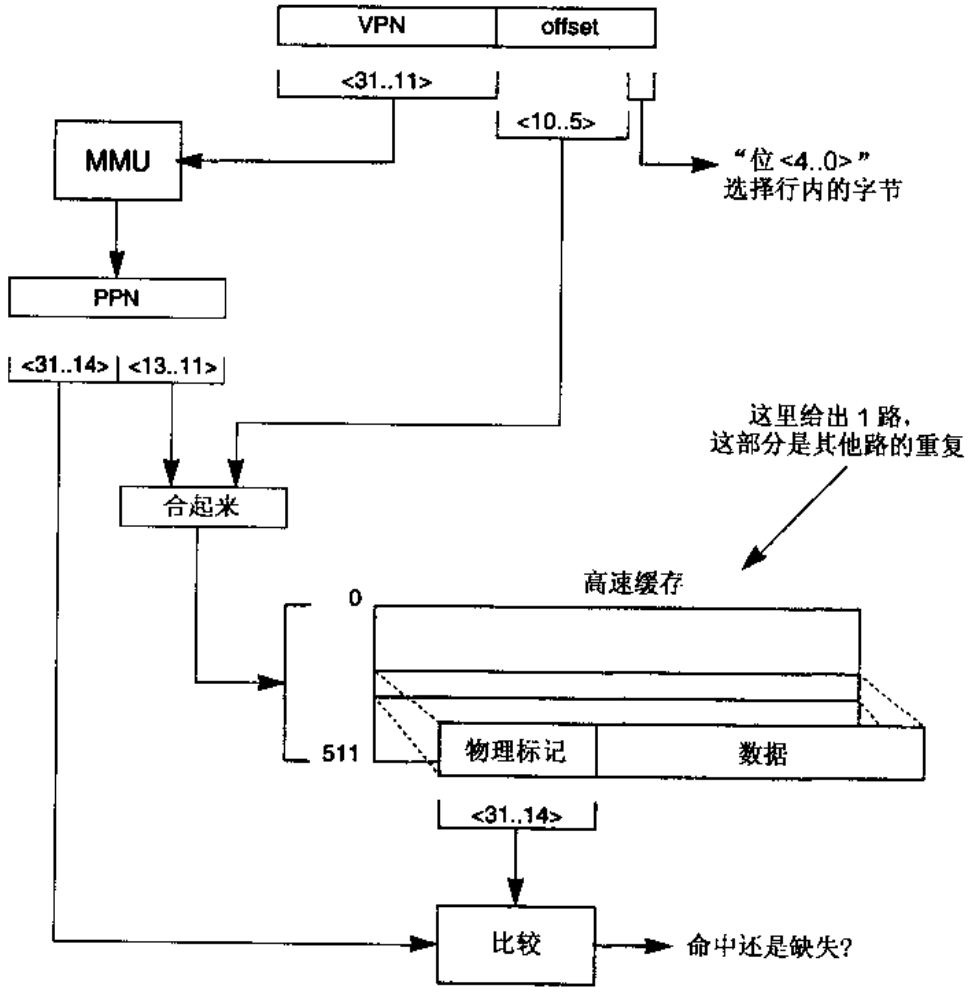


图 B-2 习题 6.1 的答案

6.4 这个问题的一个重要方面是要确保用到了所有的地址位。尤其是“位<11..10>”需要照图正确地处理，因为它们既不是由 MMU 转换的，也不用于索引组。4 路组相联高速缓存一般是通过并行运行 4 路直接映射高速缓存，每组一路来构造的。因此，图 B-3 所示的高速缓存和比较器要重复 4 次（虽然在图中只画出了一次）。

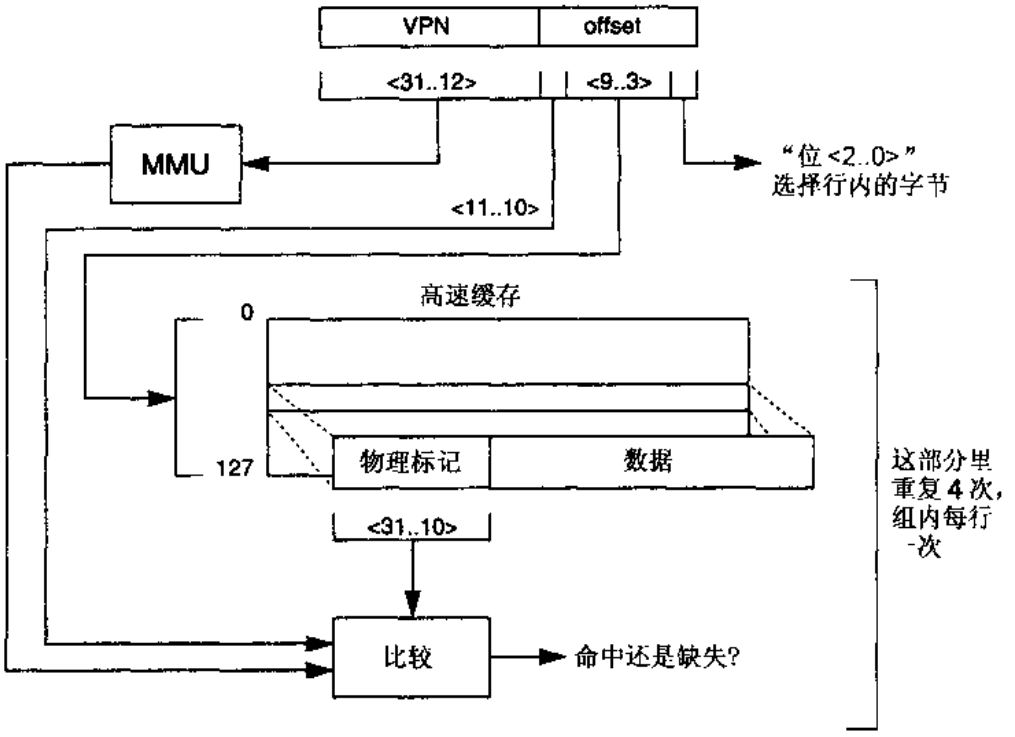


图 B-3 习题 6.4 的答案

6.5 对，它可以起作用。如果高速缓存是全相联的，那么它只是意味着在每次监听操作期间要检查所有的行。对于全相联高速缓存没有作索引，于是不需要从页偏移量获得一个索引。

6.7 它应该写入到二级高速缓存。把数据写入主存储器的做法既不必要，也不可取。首先，因为一级高速缓存始终是二级高速缓存的一个子集，所以就保证了二级高速缓存已经包含有一级高速缓存行的一个副本。所以在写回一级高速缓存行期间，不可能在二级高速缓存内发生缺失。其次，把它写入二级高速缓存要比访问主存储器快。第三，被替换的数据可能仍然在进程的局部引用之中，所以继续把它缓存在二级高速缓存中是有意义的。

6.9 为了满足缺失，必须替换当前包含来自 0x1000 位置的数据的二级高速缓存行。但是，在这样做之前，和这个位置相关联的数据必须写回到存储器（如果有必要），并且要从一级和二级高速缓存中都使之无效。即使在这次缺失期间，一级高速缓存行没有缺失，也要替换它，并使之无效，因为不这样的话，它就会包含二级高速缓存中没有的数据。这在 R4000 高速缓存一致性模型中是不允许的，因为总线监视和别名检测机制不再能可靠地工作了。一级高速缓存必须始终是二级高速缓存的一个子集，所以只要二级高速缓存行被替换了，那么在一级高速缓存中与之关联的任何行都必须删除掉。一旦这些行被写回存储器并变成无效以

后，就可以在一级和二级高速缓存中载入新数据。二级高速缓存行的缓存标记中的索引将会设置为新的一级高速缓存行的索引。

6.11 它不能起作用。在二级高速缓存标记中的索引包含有上次有该数据的一级高速缓存行的索引；但是，一级高速缓存行可能不再包含二级高速缓存行的数据。因为一级高速缓存比较小，所以在一级高速缓存内的行被替换得更频繁一些。R4000 通过检查被索引的一级高速缓存行上的物理标记是否与二级高速缓存行的物理标记相吻合，来辨别这种情况。因为带有键的虚拟高速缓存不包含物理地址，所以无法完成这种测试。

6.13 没必要。在索引一级虚拟一级高速缓存时，只需要二级高速缓存标记中的索引。如果一级高速缓存是物理高速缓存，那么在二级高速缓存中用于索引的物理地址也可以用于索引一级高速缓存。

第 7 章

7.1 如果我们假定在应用中没有自我修改的代码（也就是说，决不会到数据或者堆栈区外执行指令），那么可以选择任何虚拟地址作为标准正文区的起始地址，因为其他区都不会在独立的指令高速缓存内导致争用。

接下来要注意的是，不可能防止数据和堆栈区在数据缓存中交叠，因为它们两者大小的最大值加起来超过了高速缓存的大小。此外，因为用于索引高速缓存的地址位都落在了页偏移量之内，所以除了高速缓存行 0 之外，不可能选择从别处起始的能对齐页面的区域地址。正因为这样，所以可以选择任何地址用于数据和堆栈地址，而不会对高速缓存行的争用产生影响。使用四路组相联高速缓存机制有助于减少发生高速缓存颠簸的可能性。

7.4 纯粹的虚拟高速缓存不能跨越现场切换维持任何进程现场。因此，增加动态地址绑定所带来的开销不会有任何收益（就减少高速缓存行争用而言）。在一次至多可以缓存一个进程现场时，不需要分布不同进程对高速缓存的引用。

这项技术也不能应用到物理高速缓存上，因为在计算一个物理高速缓存的索引时，虚拟地址也没有关系。于是，即使一个物理高速缓存能够缓存多个进程的现场，分布虚拟地址也不会减少高速缓存的争用。

7.5 因为所有的高速缓存索引位都来自于页偏移量，所以高速缓存只有一种颜色。

7.9 滞后的高速缓存无效技术只能应用到能够同时保存多个进程现场的高速缓存上。不可能推迟任何冲洗虚拟高速缓存的操作，因为标记没有包含防止访问过时数据的必要信息。这是导致它们易于出现别名和歧义问题的原因。

7.10 只要高速缓存使用物理标记，那么滞后的高速缓存无效操作就可以用于剥离共享内存和缩小 bss 段的 sbrk 调用。在访问这样的高速缓存时，它总是要转换虚拟地址，所以可以使用 MMU 的页面保护机制来防止对虚拟地址中已经剥离但仍被缓存的部分进行访问。在访问带有键的高速缓存期间不使用 MMU，所以不能防止在已剥离区域的缓存数据上发生命中。因此，滞后的无效操作不能在这些情况下使用。

7.12 应该给数组中的每项增加 4 字节。这样一来，数组里的每两项正好填满一个高速缓存行。目的是防止任何一项占据单独一行，所以将一项填充占满一个高速缓存行的做法对单处理机来说既浪费又不正确（在带有高速缓存的多处理机系统中，有些情况下将一项填充

占满一个高速缓存行的做法是合理的，单个项往往只能由一个处理机访问的情况就是这样的例子）。

第8章

8.1 因为I/O设备与CPU共享总线，所以对其访问的处理无异于对来自CPU访问的处理。总线会把CPU和I/O设备同时发起的访问顺序化处理，确切的次序则不确定。

8.3 在CPU 1完成它的保存操作之后，在0x100处的值为1。CPU 2正在从该处读取数据，并不会影响到保存操作的结果。没有办法预测CPU 2将会读取到什么样的值。和所有的同时访问一样，总线会以不确定的次序将它们顺序化。如果总线选择在CPU 1的写操作之前先向CPU 2的读操作提供服务，那么它就会读到10。否则，它就会读到新值1。

8.4 出于本书讨论的目的，在任何给定的时刻，总线上只有一个交易在进行（高性能的SMP系统使用诸如分割读（split read）这样的技术，它允许同时进行对多个不同位置的读操作）。

8.5 在理想情况下，当总线处于空闲时发生一次请求，则出现最短时间。在这种情况下，交易立即开始，所以最短时间为0（有些系统总线的时钟要求每次交易必须从一个特定的时钟信号开始。所以，即使总线是空闲的，请求方也还要等上一个时间单位才能开始下一个周期。因此，对于这些假设来说，答案为1也对）。最长时间就是给每个CPU和I/O设备都提供一次总线交易服务所需要的时间。因为总线上总共有15个设备，所以任何一个设备都可能在它请求获得服务之前等上14个时间单位。因为使用了循环调度机制，每个设备一次只允许执行一次交易，所以最长时间不会比这更长了。

8.6

```
int test_and_set( int *addr )
{
    int state;
    do

        state = load_linked( addr );
    while( store-conditional (addr, 1) == 0 );
    return( state !=0 );
}
```

8.9 解决这个问题的技术是，将一个标志临时保存在一个字中（在本例中为-1，因为计数值决不会为负），使试图累加同一个字的多个进程保持同步。这要求使用特殊的函数来读这个值。

```
void inc_atomic( int *addr )
{
    int value;
    while( ( value = swap_atomic( addr, -1 ) ) == -1 )
        ;
    *addr = value + 1;
}
int read_counter( int *addr )
```

```
{
    int value;

    while((value = *addr) == -1)
        ;

    return value;
}
```

8.10 对，存在竞争条件。如果两个（或者两个以上）的处理器同时执行这个函数，那么它们两个都会将其各自新链表元素中的next指针设置为list中的当前值。然后它们都会试图把其新元素的地址保存到list中。总线会把这些保存操作顺序化，从而导致第一个被第二个覆盖掉。这会造成保存第一个元素的处理器所插入的新元素丢失。

8.13 大多数多处理机UNIX实现都至少要做两件事中的一件来保证等候锁的进程不会永远等不到。首先，sleep函数实现为一个FIFO链表。当打开对象上锁的代码唤醒所有为该锁而睡眠的进程时，就会按照这些进程睡眠的次序把它们放入到运行队列中，从而让睡眠最长的进程先于其他进程运行。这就保证了先睡眠的进程先有机会获得锁。其次，可以实现调度器，让一个进程的优先级随着它醒来发现锁已经被占据而立即返回去入睡的次数增加而增加。这就会让最老的这类进程有最高的优先级，因此比排在后面的其他进程先有机会获得锁。

第9章

9.1

```
void lock( volatile lock_t *lock_status )
{
   do {
      while( load_linked( lock_status ) == 1 )
   } while( store_conditional(lock_status, 1) == 0 );
}
```

用Unlock的代码和图9-5所示的代码一样。

9.3

```
 *
 *实现一个多读-单写自旋锁的数据结构
 */
struct rw_lock{
    lock_t rd_cnt_lock;   /*保护rd_cnt字段 */
    int rd_cnt;   /*当前读方的数量*/
    lock_t wr_lock;    *写方的锁 */
};

typedef struct rw_lock rwlock_t;

struct list{
    elem_t *head;   /*链表中的第一个元素*/
    rwlock_t lock;    *保护列表的锁 *,
};
```

```
typedef struct list list_t;

void init_rwlock (rwlock_t *lock)
{
    lock->rd_cnt=0;
    initlock(&lock->rd_cnt_lock);
    init_lock(&lock->wr_lock);
}

void initlist (list_t *list)
{
    list->head=NULL;
    init_rwlock(&list->lock);
}

lock_reader (rwlock_t *lock)
{
    /*
     *等待任何写方完成，然后累加读方的计数，屏蔽新的写方，直到我们完成为止。
     */

    lock (lock->wr_lock);
    lock (lock ->rd_cnt_lock);
    lock->rd_cnt++;
    unlock(lock->rd_cnt_lock);
    unlock(lock ->wr_lock);
}

unlock_reader (rwlock_t *lock)
{
    lock(lock->rd_cnt_lock);
    lock ->rd_cnt--;
    unlock(lock->rd_cnt_lock);
}

lock_writer(rwlock_t *lock)
{
    /*
     *等待任何写方完成，然后等待读方完成。占据写方的锁，屏蔽任何新的写方或者读方，直到我们完成
为止。
    */

    lock(lock->wr_lock);

    while (lock->rd-cnt)
        ;
}
unlock_writer(rwlock_t *lock)
{
    unlock(lock->wr_lock);
}
int search (list_t *list, int tag)
{
    elem_t *ep;
    int result;
```

```
    result=0;
    lock_reader(&list->lock);
    for (ep=list->head; ep; ep=ep->next)
        if (ep->tag==tag){
            result=ep->data;
            break;
        }
    unlock_reader (&list->lock);
    return result;
}
void add (list_t *list, elem_t *ep)
{
    lock_writer (&list->lock);
    ep->next=list->head;
    list->head=ep;
    unlock-writer(&list->lock);
}
elem_t *remove(list_t *list, int tag)
{
    elem_t *ep, *prev;
    lock_writer (&list->lock);
    for (ep=list->head; ep; ep = ep->next){
        if (ep->tag==tag){
            if (ep==list->head)
                list->head=ep->next;
            else
                prev->next=ep->next;
            break;
        }
        prev=ep;
}
  unlock_writer(&list->lock);
  return ep;
}
```

9.5 在这些函数中不可能出现死锁。这些例程只有一个公共的锁，所以不会出现 AB-BA 死锁。如果两个函数同时在不同的处理器上开始执行，那么 func2 只会等待 func1 完成。无需 lock_c，func1 就能完成，并且 func1 与 func2 无关。

9.6 这 3 个函数会发生死锁，只有当所有 3 个例程同时在 3 个处理器上开始执行的时候，才会发生死锁。如果发生了这种情况，那么每个处理器都会 3 个锁中的一个。因为所有的锁现在都被占用了，所以它们谁也不能在函数中锁住第二个锁。这是包含有 3 个锁的 AB-BA 死锁问题的变形。func2 和 func3 让 lock_a 和 lock_b 锁住的次序与 func1 中的相反。注意，只要 3 个例程没有同时开始执行，那么就不会出现死锁。

9.9 主处理器不需要做什么特殊的操作来通知从处理器。它只需要像在单处理机内核实现中那样，向进程发送信号就行了。回忆 9.4.4 小节的内容，每个从处理器在每次时钟中断的

时候都要检查是否有针对当前正在执行的进程的挂起信号。因此，在从处理器注意到中断之前，至多过去一个时钟中断周期。即使从处理器正在运行的进程陷入了无限循环，它仍然能够获得时钟中断，因为这在以用户态执行的时候是绝对不会被屏蔽的。当从处理器注意到有信号的时候，它就把进程放入主处理器的运行队列中，让信号能够得到处理。当它运行在主处理器上的时候，就会处理信号，进程将终止。

第 10 章

10.1 当正在运行一个用户态进程的处理器上发生一次时钟中断的时候，要执行的活动只是进行检查，看是否要求进行现场切换（当时间配额用光，或者有更高优先级的程序要运行的时候），或者看是否向进程发出了一个信号。不需要获得内核巨型锁就能检查这两种情况（假定使用了一个独立的运行队列锁，于是如果有必要，可以完成一次现场切换）。这和主从处理机内核实现的情况是一样的。

对于占有内核巨型锁的处理器来说，没有必要在它占有锁的全部时间里屏蔽时钟中断。单处理机 UNIX 内核被设计成在内核中执行时，除了显式地屏蔽中断的代码片段之外，都允许出现时钟中断。

时钟中断处理程序执行的周期性“清理”任务，如跟踪当前时间、系统调用超时报警、调整进程优先级等（所有这些通常每秒钟执行一次），必须在占有内核巨型锁的同时执行完。否则就会造成竞争条件，因为中断处理程序会修改与内核其他部分共享的数据结构。如果时钟中断处理程序决定是到执行这些活动的时候了，那么它必须进行检查，看它是否中断了一个内核态的进程。如果是，那么那个进程必定占据着内核巨型锁（否则，它是不能在内核态中运行的）。因此，时钟中断处理程序能够继续它的任务，而不会有出现竞争条件的危险。如果它中断了一个用户态进程，并且发现是到执行清理任务的时间了，那么它就有条件地尝试获得内核巨型锁。如果它成功了，那么就意味着没有别的处理器正在内核中执行，于是时钟中断处理程序现在是独占访问。如果尝试没有成功，那么说明有其他处理器正在内核中运行。在这种情况下，最容易的做法是把清理工作推迟到处理器当前占据了巨型锁，出现了一次时钟中断的时候去做。这样一来，没有能获得锁的处理器就不需要在它本可以做些有用的用户级工作时却去自旋。

10.3 首先，因为 test-and-set 指令仍然以原子方式发挥作用，所以自旋锁操作不会受到影响。因为一旦获得了自旋锁，它就提供给数据结构互斥机制，保证了没有别的处理器同时还试图要用被锁住的数据结构。这意味着对于由一个自旋锁保护的任何数据结构来说，不能出现对相同位置同时读和写的操作，所以不需要对这段代码作修改。对于由一个长期锁保护的数据结构来说也是如此，访问进程的私有数据结构也不会受到影响。但是，在原本有竞争条件的情况下会出现的问题是，上锁被省略了。在这类 MP 系统上，举例来说，如果不占有保护信号掩码的自旋锁，一个进程就不可能检查它的信号掩码。它必须首先获得锁，以防止别的处理器在它正在读值的同时写掩码。对于诸如读和改一个进程的用户 ID 和组 ID 这样的情形来说也是一样。现在必须在占据锁的同时才能完成这些操作。

10.5 实现它的一种简单方法是，指定一个处理器作为主处理器，运行单处理机的设备驱动程序代码，然后迫使进程在调用设备驱动程序之前切换到那个处理器上。这样的一个系

统体现出了多线程技术和主从处理机技术的一种混合。

10.6 一种实行无死锁上锁次序的简单方法是，使用锁本身的地址来定义次序。通过制定规则，让具有最低地址的锁先被获得，就不会出现死锁了。为了高效起见，当以相反的次序获得锁的时候，下面的代码在花费开销先释放然后再重新获得原来的锁之前，先尝试有条件地锁住新锁。

```
Void
lock_both( lock_t *old_lock, lock_t *new_lock )
{
        if( old_lock<new_lock ){
                lock( new_lock );
                return;
        }
        if( cond_lock (new_lock) == TRUE )
                return;
        unlock( old_lock );
        lock( new_lock );
        lock( old_lock );
        return;
}
```

第11章

11.1 每个信号量不要求有自己的自旋锁。正如在10.5.4小节里指出的那样，不同的数据结构可以由相同的锁来保护。在两种情况的任何一种里，都会有正确的操作；唯一的区别在于对自旋锁的争用程度不同。如果系统内所有的信号量都由一个自旋锁来保护，那么过多的争用将导致要在有大量处理器的系统上采用细粒度的实现。如果代之以给每个信号量一个独立的自旋锁，则不太可能发生争用。

11.2 对，最初的UNIX内核可以采用信号量而不是sleep/wakeup，因为信号量提供了单处理机机制的全部功能。此外，对于其中有任何数量处理器的系统（包括单处理机系统）来说，信号量都能正确地发挥作用。在多处理机系统上使用信号量要权衡的是空间。如果系统里有许多信号量，那么信号量的数据结构会消耗比sleep/wakeup多的空间，因为后者为睡眠的进程使用一组单独的共享队列，而每个信号量要有它自己小小的数据结构。执行*P*操作和*V*操作同sleep和wakeup相比，时间开销差不多相同。但是，当wakeup操作导致许多进程被唤醒并且争夺一个互斥锁的时候，即使是单处理机也会发生颠簸。这在采用信号量时是不会出现的，因为一次只会唤醒一个进程。

11.3 策略是每个缓冲使用两个信号量，一个用于在缓冲满了的时候向进程B发信号，一个用于在缓冲空了的时候向进程A发信号。

一次性的初始化代码：

```
buf_t buffer[2];
sema_t buffer_full[2];
sema_t buffer_empty[2];
...
```

```
initsema( &buffer_full[0],0 );
initsema( &buffer_full[1],0 );
initsema( &buffer_empty[0],1 );
initsema( &buffer_empty[1],1 );
进程A的代码:
current =0;

while(1) {
    p( &buffer_empty[current] );
    用数据填写buffer[current]
    V( &buffer_full[current] );
    current =! current;
}
进程B的代码:
current = 0;

while(1) {
    p( &buffer_full[current] );
    使用buffer[current]中的数据
    V(&buffer_empty[current]);

    current = !current;
}
```

11.5 这是一种无害的竞争条件。因为新的读方已经累加了 m_rdwcnt 字段，现有的写方会执行足够的 *V* 操作，于是一旦新的写方到达执行 *P* 操作的时刻，它不会阻塞。

11.6 使用广播 *V* 操作有两个问题。第一，它会改变实现的语义。实现的设计是，一旦有写方到达，就阻塞新的读方。如果一个写方和几个其他的读方正好在广播 V 操作开始之前到达（一旦释放自旋锁，碰巧能碰上），那么在写方之后到达的读方将会被广播 V 操作所唤醒，而此时它们应该进入睡眠，以便给写方一个机会访问临界段。第二，这会导致 m_rdcnt 字段反映出的读方计数不正确，因为在其他读方到来之前，它被设置为写方的数目。因为计数值可以比读方实际的数目少，所以这会造成写方能够在所有的读方退出锁之前就获得该锁。

第 12 章

12.2 这意味着在被触发的进程有机会运行之前，进入管程的一个新进程可能会得到链表中唯一的一个元素。因此，被触发的进程在它被唤醒的时候不能认定链表中有一个元素。最后的代码同使用自旋锁和 sleep/wakeup 的 lock_object 代码非常相似。必须有一个 while 循环，以便在进程被唤醒的时候能够重新检查链表的状态。如果它发现链表中有一个元素，那么就能把这个元素从链表中删除。如果没发现，于是说明另一个进程已经得到了它，于是它不得不再次等候。这对于管程来说不是一种可取的语义，因为它把使用它们的算法复杂化了。

12.5 对于正在等候一个特殊值的进程来说，事件计数本身可能没有可以访问的信息。因此，为了实现一种广播机制，进程必须在一个外部变量中记录下它们正在等待的值。这个变量仅仅表示任何进程正在等待（而且在占有自旋锁的同时应该给予更新）的最大值。为了唤醒所有这样的进程，只要增加事件计数，直到它等于这个值就行了。

12.6 最简单的方法是，使用信号量的值来计算链表中元素的数量。于是信号量就用作资源分配。信号量 list_count 初始化为 0。

```
Void
add( elem_t *new )
{
    lock( &list_lock );
    给列表增加新元素
    unlock( &list_lock );
    v( &list_count );
}

elem_t *
remove (void)
{
    p( &list_count );
    lock( &list_lock );
    从列表中删除元素
    unlock( &list_lock );
    return element;
}
```

注意，一旦 remove 中的 *P* 操作返回，那么就保证了在链表中至少有一个元素。这与在必须显式地测试链表以了解链表是否为空的地方使用管程的实现稍有不同。还要注意，采用管程的实现依赖于这样的事实，即被触发的进程要在试图进入管程的新进程之前运行。这就保证了当正在等待的进程被唤醒时，链表中有一个元素要删除。采用信号量的实现不保证被唤醒的进程会在首次进入 remove 函数的新进程之前运行。但是，只有当两次调用 add，以确保两个进程都会找到要删除的一个元素时，才能发生这种情况。

12.8

```
typedef struct{
lock_t  *lock; /* 保护这个结构 */
char state; /* 当前睡眠锁的状态 */
proc_t *head; /* 被阻塞的进程列表的开头 */
proc_t *tail; /* 被阻塞的进程列表的结尾 */
} sleep_t
#define LOCKED 1
#define UNLOCKED 0

sleep_t *
SLEEP_ALLOC()
{
    sleep_t *slp;
    slp = malloc( sizeof( sleep_t ) );
    slp->lock = LOCK_ALLOC( 0,0,NULL,0 );
    slp->head = NULL;
    slp->tail = NULL;
    slp->state = UNLOCKED;
    return slp;
}
```

```
void
SLEEP_DEALLOC( sleep_t *slp )
{
    LOCK_DEALLOC( slp->lock );
    free(slp);
}

void
SLEEP_LOCK( sleep_t *slp, int pri )
{
    pl_t old_pl;

    old_pl = LOCK( slp->lock, plhi );

    if( slp->state == LOCKED ) {
            enqueue( slp, curproc );
            UNLOCK( slp->lock, old_pl );
            swtch();
            return;
    }

    slp->stat = LOCKED;
    UNLOCK( slp->lock, old_pl );
}

bool_t
SLEEP_TRYLOCK (sleep_t *slp )
{
    pl_t  old_pl;

    old_pl = LOCK( slp->lovk, plhi );

    if( slp ->state== LOCKED ){
            UNLOCK( slp->lock, old_pl );
            return FALSE;
}
    slp->state = LOCKED;
    UNLOCK( slp->lock, old_pl );
    return TRUE;
}
void
SLEEP_UNLOCK( sleep_t *slp )
{
    proc_t *pp;
    pl_t old_pl;

    old_pl = LOCK( slp->lock, plhi );

    if( slp->head != NULL ){
            pp = dequeue( slp );
            UNLOCK( slp->lock, old_pl );
            enqueue( &runqueue, pp );
            return;
}
slp->state = UNLOCKED;
```

```
UNLOCK( slp->lock, old_pl );
}
```

12.10 因为同步变量不保持状态，所以需要由自旋锁保护的独立变量。

一次性的初始化代码：

```
buf_t buffer[2];
int buffer_state[2];
lock_t *state_lock;
sv_t *buffer_full[2];
sv_t *buffer_empty[2];

#define EMPTY 0
#define FULL 1

state_lock = LOCK_ALLOC(...);
buffer_full[0] = SV_ALLOC(...);
buffer_full[1] = SV_ALLOC(...);
buffer_empty[0] = SV_ALLOC(...);
buffer_empty[1] = SV_ALLOC(...);
buffer_state[0] = EMPTY;
buffer_state[1] = EMPTY;
```

进程A的代码：

```
pl_t old_pl;
current = 0;

while(1) {
    LOCK( state_lock );

    if( buffer_state[current] == FULL )
      SV_WAIT( &buffer_empty[current], pri, state_lock );
else
      UNLOCK( state_lock, old_pl );

    用数据填充 buffer[current]
    old_pl = LOCK(state_lock);
    buffer_state[current] = FULL;
    SV_SIGNAL( &buffer_full[current] );
    UNLOCK( state_lock, old_pl );

    current = !current;
}
```

进程B的代码：

```
pl_t old_pl;
current = 0;

while(1) {
    old_pl = LOCK( state_lock );

    if( buffer_state[current] == EMPTY )
      SV_WAIT( &buffer_full[current],pri, &state_lock );
else
    UNLOCK( state_lock, old_pl );

    使用 buffer[current]中的数据
    old_pl = LOCK( state_lock );
```

```
    buffer_state[current] = EMPTY;
    SV_SIGNAL( &buffer_empty[current] );
    UNLOCK( state_lock, old_p );

    current = !current;
}
```

第 13 章

13.2 影响是动态的。它意味着，如果在 store 缓冲中有一条针对那个位置的保存操作，那么处理器发出的任何上载操作都会返回来自存储器的过时数据。例如，在一次循环期间更新的变量在下一次循环的时候不能可靠地读取到。同样，由一个函数保存在存储器中的返回值也不能保证由调用方可靠地读取到。对于 UP 和 MP 系统都是如此。

为了让上载操作可靠，在一条 store 指令之后发出一条 load 指令之前，必须向存储器内的一个虚拟位置发出一条 atomic-swap 指令。这条 atomic-swap 指令防止了在缓冲排空和交换完成之前再有更多的指令发出，从而确保了存储器对于后续的上载操作都是最新的。注意，只要有子例程返回，调用方希望使用子例程保存在主存储器内的数据时，就必须这样做。这样的做法会带来很大的性能损失，所以不应该做这种体系结构上的改动。

13.4 在对相同位置的连续保存操作之间，软件必须包括一条 store-barrier 指令。注意，在进入和退出子例程时需要附加的 store-barrier 指令。这就保证了在调用子例程之前发出的 store 指令都在子例程向同一个地址发出的 store 指令之前送到存储器了。对于返回的情形也是一样。因为在大多数情况下需要许多附加的 store-barrier 指令，所以整体性能会随着 CPU 预取、解码和发出 store-barrier 指令而降低，它本可以用这些时间来执行有用的指令。

13.6 改动不会影响 UP 系统，因为在这种情况下，TSO 的行为已经和顺序定序一样了。但对于 MP 系统来说，它会让系统的行为仿佛它实现了顺序定序一样。Dekker 算法无需做修改，就能像 13.2 节讨论的那样起作用，因为一个处理器的 store 缓冲对于另一个处理器执行的上载操作是可见的。

虽然这在某种程度上简化了程序，但是它却不值得去实现。因为要支持 TSO 的话，软件要改动的地方非常少，而让 store 缓冲监视总线所需要的硬件复杂性、可能带来的性能损失却是不值得的。

13.8 这根本不成问题。因为进程私有数据一次决不会被一个以上的 CPU 访问，所以在第一个处理器的 store 缓冲中有一条挂起的 store 指令的时候，其他处理器没有机会访问存储器中的过时数据。即使一个进程在它正在访问私有数据的同时进行了迁移，使 store 缓冲和存储器在现场切换期间保持同步的技术也会保证正确的操作。

第 14 章

14.2 虚拟高速缓存不能用作共享高速缓存，因为没有办法防止处理器之间的歧义发生。带有物理标记的虚拟高速缓存可以使用，因为物理标记可以防止每个使用相同虚拟地址来引用不同数据的进程所造成的歧义。

14.4 因为信号量的数据结构很小，所以最简单的方法就是从高速缓存中冲洗掉整个数据结构。这就省去了要了解任何给定的调用到底修改了哪些字段的工作。因为高速缓存是以单个高速缓存行为单位进行冲洗的，所以整个数据结构很可能正好在一两个高速缓存行内。注意，在冲洗二级高速缓存之前，必须将一级高速缓存写回存储器，以避免丢失数据。在引用 u.u_proc 之后没有必要进行冲洗，因为这是进程的私有数据。最后，既然必须将自旋锁放在不被缓存的存储器中，那么在信号量的结构中保存的是指向锁的指针，而不是锁本身。

```
struct semaphore{
        lock_t  *lock; /*保护其他字段*/
        int count; /*信号量的当前值
        proc_t *head; /*指向第一个被阻塞的进程的指针*/
        proc_t *tail; /*指向最后一个被阻塞的进程的指针*/
};

void
initsema( sema_t *sp, int initial_cnt )
{
        sp->lock = alloc_lock();
        initlock( sp->lock );
        sp->head = NULL;
        sp->tail = NULL;
        sp->count = initial_cnt;
        flush_primary( sp, sizeof (*sp), WRITE_BACK | INVALTDATE );
        flush_secondary( sp, sizeof( *sp ) );
}

void
p(sema_t *sp)
{
        lock( sp->lock );
        sp->count--;
        if( sp->count < 0 ){
              if( sp->head == NULL)
                    sp->head = u.u_procp;

              if( sp->tail ){
                    sp->tail->p_next = u.u_procp;
                    flush_primary( &sp->tail->p_next,
                          sizeof( sp->tail->p_next ),
                          WRITE_BACK | INVALIDATE );
                     flush_secondary( &sp->tain->p_next,
                          sizeof( sp->tain->p_next ) );
}

              u.u_procp->p_next = NULL;

              flush_primary( &u.u_procp->p_next,
                    sizeof(u.u_procp->p_next),
                    WRITE_BACK | INVALIDATE);
              flush_secondary( &u.u_procp->p_next,
                    sizeof( u.u_procp->p_next ) );
```

```
                sp->tail = u.u_procp;
                flush_primary( sp, sizeof( *sp ),
                        WRITE_BACK | INVALIDATE );
                flush_secondary( sp, sizeof( *sp ) );
                unlock( sp->lock );
                swtch();
                return;
                }
                flush_primary( sp, sizeof (*sp), WRITE_BACK | INVALIDATE );
                flush_secondary( sp, sizeof( *sp ) );
                unlock( sp->lock );
}
void
v( sema_t *sp )
{
        proc_t *p;

        lock( sp->lock );
        sp->count++;

        if( sp->count <= 0 ){
                p = sp->head;
                sp->head = p->p_next;
                flush_primary( &p->p_next, sizeof( p->p_next ),
                        INVALIDATE );
                flush_secondary( &p->p_next, sizeof( p->p_next ) );

                if( sp->head == NULL )
                        sp->tail = NULL;

                flush_primary( sp, sizeof( *sp ),
                        WRITE_BACK | INVALIDATE );
                flush_secondary( sp, sizeof( *sp ) );
                unlock( sp->lock );
                enqueue( &runqueue, p );
                return;
}
    flush_primary( sp, sizeof( *sp ), WRITE_BACK | INVALIDATE );
    flush_secondary( sp, sizeof( *sp ) );
    unlock( sp->lock );
}
```

14.7 当高速缓存行被替换或者被冲洗时，高速缓存仅仅把那些已经修改过的字写回存储器的事实，解决了保持计数器数组一致性问题的一部分，因为每个CPU只写回它所修改的计数器。这就确保了存储器复制操作仍然保持一致。但是，剩下的问题在于，一个处理器仍然会在同一个高速缓存行内保存相邻计数器可能业已过时的值。所以，一个处理器一次获得两个相邻计数器的锁，这种情况仍然会造成它使用过时的数据。因此，必须进行和14.3.2小节所介绍的相同类型的冲洗

14.9 没有必要每个处理器都使用相同的键。只有在防止特殊处理器上用户数据和内核数据之间的歧义时才需要这些键。一个处理器使用的键不会影响系统中其他任何高速缓存。

为了方便起见，内核可能会使用相同的键，但是保持高速缓存一致性并不对此做要求。

14.11 当使用一个至少包含一块写回高速缓存的多级高速缓存体系结构时，冲洗操作的次序就有关系了，不管内核是正在采用有选择的冲洗操作来保持一致性，还是正在迁移一个进程，都是如此。当进行冲洗以执行一次写回操作时，必须首先将最靠近 CPU 的高速缓存写回存储器。如果存在多级写回高速缓存，那么按照从最靠近 CPU 到最靠近总线的顺序，依次把高速缓存的内容写回存储器。在使高速缓存无效的时候，次序没有关系，因为是在丢弃数据。

14.13 有两种可能的方法。第一种方法只是在每次执行解锁操作的时候冲洗高速缓存。读方只需要使它们的高速缓存无效，而写方必须写回数据，并使之无效。第二种方法注意到了在进程正以只读方式使用锁的时候，不会发生不一致性。对于所有只读数据来说都是这样。只有当为了写操作而获得锁的时候，才必须进行高速缓存冲洗。所以，当为了写操作而获得锁的时候，写方向所有的处理器发送一则广播，使得所有与临界资源有关的数据都变成无效。在这之后，写方就会继续执行，因为它知道所有其他的缓存副本都没有了。当写方释放锁的时候，必须照常把那个处理器上的高速缓存写回存储器，并使之无效。

第 15 章

15.1 必须把数据写回主存储器，因为在一个已修改过的行上发生的命中会导致两个高速缓存都把该行保存为有效状态。数据本身不会立即丢失（因为两个高速缓存都有该行的一个副本），但是该行已经相对于主存储器进行了修改的事实却丢失了。有了现在处于有效状态的高速缓存行，它们可以在任何时刻被替换，两个高速缓存都不会把它们写回到主存储器（假定它们都不对该行执行保存操作），这将造成行内被修改过的数据丢失。此外，在监听命中的高速缓存中留下处于已修改状态的高速缓存行也不正确。这会允许 CPU 无需使得提供给其他高速缓存的副本无效，就可以修改数据。在监听命中时把已经修改过的数据写回主存储器，就可以解决这些问题。

15.3 因为写一次协议是写直通和写回的一种混合，所以在协议指定写直通操作的情况下，必须用原子操作更新主存储器。此刻，原子操作将会更新主存储器。在该行初始为保留状态的情况下，不需要的总线或者存储器操作，因为已知其他高速缓存都没有该行的一个副本。原子操作可以完全在处理器的私有高速缓存中执行。

15.5 它不能正确地保持一致性。一个问题是，对有效的高速缓存行执行的保存操作不会产生总线交易，来让该行在其他高速缓存内的副本无效。于是两个处理器就可以共享这个高速缓存行的一个副本，每个处理器都无需使对方的副本无效就可以修改它。这会导致该行有两个不一致的缓存副本。第二，响应监听到的 store 命中，使一个已修改过的行无效是不正确的。即使其他处理器要修改这行，它也可能会修改这行的不同部分。这会导致现在缓存它的处理器所写的修改数据丢失。

15.7 对，即使在使用写-更新协议时这些行不会产生“ping-pong”效应，使用图 15-7 所示的代码也不失为一个好主意。如果函数 lock 通过执行测试-设置操作处于忙等待状态，那么在有多个 CPU 争用同一个锁的情况下，每次通过循环的时候它都会产生一个总线交易，

更新其他缓存副本。这是对总线带宽的一种浪费，可以通过在锁被释放之前轮询锁的状态来消除这种浪费。

15.9 信号量要实现长期锁和进程同步。因此，不应该期望有很多 CPU 同时访问信号量的数据结构。这就反过来意味着，自旋锁可以和信号量的数据结构剩下来的部分处于同一高速缓存行内，而不必担心“ping-pong”现象。因为在一个信号量上的每次操作都至少会访问锁和数据结构里的计数字段，所以如果数据结构 semaphore 对齐，以便与一个高速缓存行相匹配，并且有必要的话进行填充，从而让同一行内没有其他无关的数据，那么就最好不过了。如果信号量保护了一个临界资源，那么在同一个高速缓存行内一起对齐和填充信号量及其临界资源是有好处的（假定行的大小足够大）。

15.11 有可能，但要求在二级高速缓存的标记中附加空间。摹仿 MIPS R4000 所使用的方法，二级高速缓存必须包含定位一级高速缓存中的行所需要的信息。在这种情况下，可以像 R4000 那样，将一级高速缓存的索引保存在二级高速缓存的标记中。此时，R4000 能够检查在一级高速缓存中的命中，因为行是由物理地址来标记的。因为物理地址是由引发监听的总线交易来提供的，所以很容易就能做到这一点。但是，当检查带有键的虚拟高速缓存时，它毫无用处。相反，需要保存在一级高速缓存标记中的虚拟地址部分和键。和一级高速缓存行的索引一样，在一开始就分配一级高速缓存行（而且随后使用包含属性又分配了二级高速缓存行）的时候，可以把这些信息保存在二级高速缓存的标记中。