

国外计算机科学教材系列

分布式操作系统

Distributed Operating Systems

[美] Andrew S. Tanenbaum 著

韩燕翔 伍卫强 刘建刚 等译校



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

WULIYU TIANHOU WULIYU WULIYU



PRENTICE HALL 出版公司

465430

国外计算机科学教材系列

分布式操作系统

Distributed Operating Systems

Andrew S. Tanenbaum 著

陆丽娜 伍卫国 刘隆国 等译校



00465430

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 提 要

分布式操作系统是为分布式计算机系统配置的一种操作系统。本书涵盖了分布式操作系统的所有内容,并重点介绍设计和构造分布式操作系统的基本原理、算法和实现技术。全书共 11 章,前 6 章较全面地介绍了分布式系统环境中的通信、同步、进程、文件系统和存储器管理等方面的概念与算法;后 4 章详细地给出了 4 个分布式系统的实例:Amoeba、Mach、Chorus 和 DCE,介绍其设计思想和实现技术。本书既注重基础知识的系统性,同时兼顾选材的先进性,内容全面经典、层次清楚。

本书可作为学习、使用和讲授分布式操作系统的大学生、研究生和教师的教学用书,也可供从事分布式计算机系统的科技工作者阅读和参考。

©1995 by Prentice-Hall, Inc.

本书中文简体版由电子工业出版社和美国 Prentice-Hall 出版公司合作出版。未经许可,不得以任何手段和形式复制或抄袭本书内容。版权所有,侵权必究。

图书在版编目(CIP)数据

分布式操作系统/(美)坦恩鲍姆(Tanenbaum, A.S.)著;

陆丽娜等译 - 北京:电子工业出版社, 1999.12

(国外计算机科学教材系列)ISBN 7-5053-5487-6

I.分... II.①坦... ②陆... III.分布式操作系统-教材 IV.TP316

中国版本图书馆 CIP 数据核字(1999)第 63451 号

丛 书 名: 国外计算机科学教材系列

书 名: 分布式操作系统

原 书 名: **Distributed Operating Systems**

著 者: [美] Andrew S. Tanenbaum

译 校 者: 陆丽娜 伍卫国 刘隆国 等

责任编辑: 陆伯雄

特约编辑: 杨宝珍

印 刷 者: 北京天竺颖华印刷厂

出版发行: 电子工业出版社 URL: <http://www.phei.com.cn>

北京市海淀区万寿路 173 信箱 邮编 100036

经 销: 各地新华书店

开 本: 787×1092 1/16 印张: 28.5 字数: 672 千字

版 次: 1999 年 12 月第 1 版 1999 年 12 月第 1 次印刷

印 数: 8000 册

定 价: 45.00 元

书 号: ISBN 7-5053-5487-6/TP·2770

著作权合同登记号 图字: 01-1999-2973

凡购买电子工业出版社的图书,如有缺页、倒页、脱页者,请向购买书店调换。

若书店售缺,请与本社发行部联系调换。电话 68279077

出版说明

计算机科学的迅速发展是 20 世纪科学发展史上最伟大的事件之一。从 1946 年第一台笨重而体积庞大的计算机的发明至今, 仅仅半个多世纪, 计算机已经变得小巧无比却又能力非凡。它的应用已经渗透到了社会的各个方面, 成为当今所谓的信息社会的最显著的特征。

处于世纪之交科技进步的大潮中, 我国正在加强计算机科学的高等教育, 着眼于为下一世纪培养高素质的计算机人才, 以适应信息社会加速度发展的需要。当前, 全国各类高等院校已经或计划在各专业基础课程规划中增加计算机科学的课程内容, 而作为与计算机科学密切相关的计算机、通信、信息等专业, 更是在酝酿着教学的全面革新, 以期规划出一整套面向 21 世纪的、具有中国高校计算机教育特色的课程计划和教材体系。值此, 我们不妨借鉴并引进国外具有先进性、实用性和权威性的大学计算机教材, 洋为中用, 以更好地服务于国内的高校教育。

美国 Prentice Hall 出版公司是享誉世界的高校教材出版商, 自 1913 年公司成立以来, 即致力于教育图书的出版。它所出版的计算机教材在美国为众多大学所采用, 其中有不少是专业领域中的经典名著。许多蜚声世界的教授学者成为该公司的资深作者, 如: 道格拉斯·科默(Douglas Comer), 安德鲁·坦尼伯姆(Andrew Tanenbaum), 威廉·斯大林(William Stallings)……几十年来, 他们的著作教育了一批批不同肤色的莘莘学子, 使这些教材同时也成为全人类的共同财富。

为了保证本系列教材翻译出版的质量, 电子工业出版社和 Prentice Hall 出版公司共同约请北京地区的清华大学、北京大学、北京航空航天大学, 上海地区的上海交通大学、复旦大学, 南京地区的南京大学、解放军通信工程学院等全国著名的高等院校的教学第一线的几十位教师参加翻译工作。这中间有正在讲授同类教材的年轻教师和博士, 有积累了几十年教学经验的教授和博士生导师, 还有我国著名的计算机科学家。他们的辛勤劳动保证了本系列丛书得以高质量地出版面世。

如此大规模地引进计算机科学系列教材, 在我们还是第一次。除缺乏经验之外, 还由于我们对计算机科学的发展, 对中国高校计算机教育特点认识的不足, 致使在选题确定、翻译、出版等工作中, 肯定存在许多遗憾和不足之处, 恳请广大师生和其他读者提出批评、建议。

电子工业出版社

URL:<http://www.phei.com.cn>

Prentice Hall 出版公司

URL:<http://www.prenhall.com>

译者序

近年来, 分布式处理系统获得了突飞猛进的发展, 并呈现出前所未有的广阔前景。为此我们必须学习和系统了解当前国内外分布式系统最新的理论与技术。坦尼鲍姆教授是国际知名的计算机科学和教育家。他在计算机操作系统、分布式操作系统、计算机网络领域都有很深的造诣。对操作系统的介绍, 他先后完成了面向大学生和研究生的三卷专著: 《操作系统: 设计和实现》、《分布式操作系统》和《现代操作系统》, 这些教材被国内外的许多大学广为采用。为此我们翻译了《分布式操作系统》一书。

《分布式操作系统》一书涵盖了分布式操作系统的所有的内容。较全面地介绍了分布式系统环境中的通信、同步、进程、文件系统和存储器管理等方面的概念与算法, 并详细地给出了分布式系统设计实例, 它不仅可作为一本很好的教材, 同时对分布式系统的设计将带来很大帮助。

本书的特点是理论与实践紧密结合, 第 1 章到第 6 章讲述原理, 第 7 章到第 10 章分别介绍 4 个分布式操作系统实例。通过学习, 使读者对分布式操作系统的原理和实现有一个完整的认识 and 了解, 并能够了解与跟踪当前的最新技术。

本书的第 1 章由陆丽娜翻译, 第 2、3 章由刘归荣翻译, 第 4、5、6 章由吴昊、艾小平、崔刚翻译, 第 7、8、9、10、11 章由刘隆国、伍卫国翻译。全书由陆丽娜教授审校并统稿。书中所有的图表由张兴军、冯锐与邢飞翻译并绘制。

在本书翻译过程中, 部分译稿在西安交通大学电子信息工程学院计算机系 97 级研究生分布式系统课程中进行了试用, 许多学生提出了宝贵的意见。伊景冰对本书的文字修饰提出了宝贵意见, 在此对他们表示衷心的感谢。

虽然在翻译过程中我们尽量做到尊重原意、翻译准确, 希望它能对国内的分布式系统的教学与科研有所帮助, 但由于水平所限, 不当和疏漏之处在所难免, 敬请读者提出宝贵意见。

译者
西安交通大学电子与信息学院
计算机科学与技术系
1999. 5

序 言

在 Distributed Operating Systems 一书出版后，我已经完成了对操作系统介绍的三步曲。这个三步曲中的三本书是：

- Operating Systems: Design and Implementation (操作系统：设计和实现)
- Distributed Operating Systems (分布式操作系统)
- Modern Operating Systems (现代操作系统)

然而，这三本书并不是完全独立的。对于有两门操作系统课程（或一门本科操作系统课程和一门研究生操作系统课程）的学校，一种可能的选择可以是使用 Operating Systems: Design and Implementation 作为第一门课程，而用 Distributed Operating Systems 作为第二门课程。

第一本书介绍单处理机系统中操作系统的一些基本原理，包括进程、同步、I/O、死锁、内存管理、文件系统、安全性等等。它还应用 MINIX 系统对这些原理进行十分详细的说明，MINIX 系统是 UNIX 系统的克隆体，它的源文件在书的附录中给出。MINIX 光盘附在书后，可在 IBM PC (8088 和更高档的)，Atari、Amiga、Macintosh 和 SPARC 处理机上运行。

第二本书（即本书），详细介绍了分布式操作系统，包括分布式系统环境中的通信、同步、进程、文件系统和内存管理等方面。并详细地给出了 4 个分布式系统的例子：Amoeba、Mach、Chorus 和 DCE。Amoeba 系统对于大学中用于教学方面是免费的。它可在 Intel 386/486、SPARC 和 Sun3 处理机上运行。要想知道怎样得到 Amoeba 系统的信息，请从 <ftp://ftp.cs.vu.nl> 上 FTP 下载文件 `amoeba/Intro.ps.Z` 或者与作者通过电子邮件联系，地址是：ast@cs.vu.nl。首先要提醒一下各个用户，Amoeba 要比 MINIX 复杂得多：光文档（可通过 FTP 获得）就有 1000 多页，并且系统要想运行得较好，至少需要 5 个大型机和一个以太网环境。

通过先后学习这两本书并使用 MINIX 和 Amoeba 系统，学生能够对单处理机操作系统和分布式操作系统的原理和实现有一个全面的认识 and 了解。现在这三本书都已完成，我计划要修改 MINIX 操作系统及介绍它的书籍。

对于时间不宽松的大学和计算机专家来说，Modern Operating Systems 一书是其它两本书的一个浓缩。那两本书介绍了单处理机操作系统和分布式操作系统的基本原理，但是没有 MINIX 这样详细的例子。它还省略了这本书中介绍的许多高级的主题，包括 ATM 的介绍，容错分布式系统、实时分布式系统、分布式共享存储器，Chorus 和 DCE 等等。总的来说，Modern Operating Systems 一书中大约省略了关于分布式操作系统的 230 页的内容。

许多人帮我完成了此书的创作。我特别感谢以下朋友，他们都阅读了原稿的一些部分并给我提出了许多宝贵的改进意见，他们是：Irina Athanasiu、Henri Bal、Saniya Ben Hassen、David Black、John Carter、Randall Dean、Wiebren de Jonge、John Dugas、Dick

Grune、Anoop Gupta、Frans Daashoek、Marcus Koebler、Hermann Kopetz、Ed Lazowska、Dan Lenoski、Kai Li、Marc Maathuis、David Mosberger、Douglas Orr、Craig Partridge、Carlton Pu、Marc Rozier、Rich Salz、Mike Schroeder、Karsten Schwan、Greg Sharp、Dennis Shasha、Sol Shatz、Jennifer Steiner、Chuck Thacker、John Turek、Walt Tuvell、Leendert van Doorn、Robbert van Renesse、Kees Verstoep、Ellen Zegura、Willy Zwaenpoel 和一些无名的审阅者。我的编辑者 Bill Zobrist，承受着为将本书写得尽善尽美所带来的麻烦，而毫无怨言。

虽然有这么多人的协助，但错误总还是存在的。不管多少人阅读了原稿，仍有错误存在是不可避免的。如果您发现了错误，请通过电子邮件和我联系。

最后，我要再一次感谢 Suzanne。在帮我写完 8 本书后，她又帮我写另一本书了，她的耐心和爱是无穷的。我同样要感谢 Barbara 和 Marvin 两人让我单独使用他们的计算机。教我们如何使用 PC 字处理程序，使我更加喜欢 UNIX 中的 troff 排版编辑程序了。最后，我要感谢 Little Bram 在我写书的时候，为我提供一个很安静的环境。

Andrew S. Tanenbaum

(使用本书中文版作为教材授课的教师，请联络 Prentice Hall 公司北京代表处，电子邮件地址为：ssbj@bupt.edu.cn，通信地址为：北京西三环北路 19 号外研社大厦 2205 室，邮编 100081。目前教师指导书仅能免费提供英文版。)

目 录

第 1 章 分布式系统概述.....	1
1.1 什么是分布式系统?	1
1.2 目标.....	2
1.2.1 分布式系统与集中式系统相比较而言的优点	2
1.2.2 分布式系统与独立 PC 机相比较的优点	4
1.2.3 分布式系统的缺点	5
1.3 硬件概念	6
1.3.1 基于总线的多处理机.....	8
1.3.2 交换型多处理机	9
1.3.3 基于总线的多计算机.....	10
1.3.4 交换型多计算机	11
1.4 软件概念	11
1.4.1 网络操作系统	12
1.4.2 真正的分布式系统	14
1.4.3 多处理机分时系统	15
1.5 设计中的问题.....	17
1.5.1 透明性(Transparency)	17
1.5.2 灵活性	19
1.5.3 可靠性	20
1.5.4 性能.....	21
1.5.5 可伸缩性.....	22
1.6 小 结.....	23
习 题.....	24
第 2 章 分布式系统的通信	25
2.1 分层协议.....	25
2.1.1 物理层	27
2.1.2 数据链路层	28
2.1.3 网络层	29
2.1.4 传输层	29
2.1.5 会话层	30
2.1.6 表示层	30
2.1.7 应用层	30
2.2 异步传输模式网(ATM 网).....	30
2.2.1 什么是异步传输模式.....	30
2.2.2 ATM 物理层	32
2.2.3 ATM 层.....	32

2.2.4	ATM 适配层	33
2.2.5	ATM 交换.....	34
2.2.6	ATM 对分布式系统的影响	35
2.3	客户-服务器模式.....	36
2.3.1	客户机和服务器	36
2.3.2	客户和服务器的一个示例.....	37
2.3.3	寻址.....	41
2.3.4	阻塞与非阻塞原语	43
2.3.5	有缓冲和无缓冲原语.....	45
2.3.6	可靠的和非可靠原语.....	46
2.3.7	客户-服务器模式的实现	47
2.4	远程过程调用.....	49
2.4.1	基本 RPC 操作.....	50
2.4.2	参数传递.....	53
2.4.3	动态捆绑.....	56
2.4.4	失败情况下的 RPC 语义.....	58
2.4.5	实现的问题	62
2.4.6	问题领域.....	70
2.5	组通信.....	72
2.5.1	组通信的引入(Introduction to Group Communication).....	72
2.5.2	设计的问题	73
2.5.3	在 ISIS(组合软件调用系统)中的组通信	80
2.6	小结.....	83
习 题	84
第 3 章	分布式系统的同步	86
3.1	时钟同步	86
3.1.1	逻辑时钟.....	87
3.1.2	物理时钟.....	90
3.1.3	时钟同步算法	92
3.1.4	使用同步时钟	96
3.2	互斥.....	97
3.2.1	集中式算法	97
3.2.2	分布式算法	98
3.2.3	令牌环算法	100
3.2.4	三种算法的比较	101
3.3	选举算法.....	102
3.3.1	欺负 (Bully) 算法	103
3.3.2	环算法	104
3.4	原子事务.....	105
3.4.1	原子事务简介	105
3.4.2	事务模型.....	106

3.4.3	实现.....	110
3.4.4	并发控制.....	113
3.5	分布式系统中的死锁.....	116
3.5.1	分布式死锁检测.....	117
3.5.2	分布式死锁预防.....	120
3.6	小结.....	122
习 题	122
第 4 章	分布式系统中的进程和处理机.....	124
4.1	线程.....	124
4.1.1	线程简介.....	124
4.1.2	线程的用途.....	126
4.1.3	线程包的设计问题.....	128
4.1.4	实现一个线程包.....	130
4.1.5	线程和远程过程调用 (RPC).....	134
4.2	系统模型.....	136
4.2.1	工作站模型.....	136
4.2.2	使用空闲工作站.....	138
4.2.3	处理机池模型.....	141
4.2.4	混合模型.....	144
4.3	处理机分配.....	144
4.3.1	分配模型.....	144
4.3.2	处理机分配算法的设计问题.....	146
4.3.3	处理机分配算法的实现问题.....	148
4.3.4	处理机分配算法举例.....	149
4.4	分布式系统的调度.....	154
4.5	容错.....	155
4.5.1	组成部件错误.....	156
4.5.2	系统失效.....	156
4.5.3	同步系统与异步系统.....	157
4.5.4	使用冗余.....	157
4.5.5	使用主动复制方法的容错.....	158
4.5.6	使用主机后备的容错.....	159
4.5.7	容错系统中的协同一致.....	161
4.6	实时分布式系统.....	163
4.6.1	什么是实时系统?.....	163
4.6.2	设计问题.....	165
4.6.3	实时通信.....	168
4.6.4	实时调度.....	171
4.7	小结.....	176
习 题	176
第 5 章	分布式文件系统.....	179

5.1	分布式文件系统设计	179
5.1.1	文件服务接口	179
5.1.2	目录服务器接口	181
5.1.3	文件共享的语义	185
5.2	分布式文件系统的实现	187
5.2.1	文件的使用	188
5.2.2	系统结构	189
5.2.3	高速缓存(caching, 超缓存)	192
5.2.4	复制	196
5.2.5	例子: SUN 公司的网络文件系统	199
5.2.6	学到的教训	204
5.3	分布式文件系统的发展趋势	205
5.3.1	新的硬件	205
5.3.2	规模	207
5.3.3	广域网	207
5.3.4	移动用户	208
5.3.5	容错	209
5.3.6	多媒体	209
5.4	小结	209
习 题	210
第 6 章	分布式共享存储器	212
6.1	简介	213
6.2	什么是共享存储器?	214
6.2.1	芯片存储器	214
6.2.2	基于总线的多处理机	214
6.2.3	基于环的多处理机	218
6.2.4	交换式多处理机	220
6.2.5	NUMA 多处理机	224
6.2.6	分布式共享系统的比较	227
6.3	一致性模型	229
6.3.1	严格一致性 (Strict Consistency)	230
6.3.2	顺序一致性(Sequential Consistency)	231
6.3.3	因果一致性(Causal Consistency)	234
6.3.4	PRAM 一致性(PRAM Consistency)和处理器一致性(Processor Consistency)	235
6.3.5	弱一致性 (Weak Consistency)	236
6.3.6	释放一致性 (Release Consistency)	238
6.3.7	入口一致性 (Entry Consistency)	240
6.3.8	一致性模型总结	241
6.4	基于分页的分布式共享存储器	242
6.4.1	基本设计	242
6.4.2	复制	243

6.4.3	粒度 (Granularity)	244
6.4.4	实现顺序一致性 (Achieving Sequential Consistency)	245
6.4.5	寻找拥有者 (Finding The Owner)	247
6.4.6	寻找拷贝	248
6.4.7	页面置换 (Page Replacement)	249
6.4.8	同步	250
6.5	共享变量的分布式共享存储器	250
6.5.1	Munin	251
6.5.2	Midway	256
6.6	基于对象的分布共享内存	257
6.6.1	对象	258
6.6.2	Linda	259
6.6.3	Orca	264
6.7	比较	268
6.8	小结	269
	习题	270
第 7 章	实例研究 1: Amoeba	272
7.1	Amoeba 介绍	272
7.1.1	Amoeba 的发展史	272
7.1.2	研究目标	272
7.1.3	Amoeba 的系统结构	273
7.1.4	Amoeba 微内核	275
7.1.5	Amoeba 服务器	276
7.2	Amoeba 中的对象 (object) 和权能 (capabilities)	277
7.2.1	权能 (Capability)	277
7.2.2	对象保护	278
7.2.3	标准操作	279
7.3	Amoeba 中的进程管理	280
7.3.1	进程	280
7.3.2	线程	282
7.4	Amoeba 中的内存管理	283
7.4.1	段 (segment)	283
7.4.2	段映射 (mapped segments)	283
7.5	Amoeba 的通信	284
7.5.1	远程过程调用(RPC)	284
7.5.2	Amoeba 的组通信	287
7.5.3	快速本地互联协议 FLIP (The Fast Local Internet Protocol)	293
7.6	Amoeba 服务器	299
7.6.1	子弹服务器 (Bullet Server)	299
7.6.2	目录服务器	302
7.6.3	复制服务器	306

7.6.4	运行服务器	306
7.6.5	引导服务器 (the boot server)	308
7.6.6	TCP/IP 服务器	308
7.6.7	其他服务器	308
7.7	小结	308
习 题	309
第 8 章	实例研究 2: Mach	311
8.1	有关 Mach 的介绍	311
8.1.1	Mach 的发展历史	311
8.1.2	Mach 的设计目标	312
8.1.3	Mach 微内核	312
8.1.4	Mach 的 BSD UNIX 服务器	314
8.2	Mach 中的进程管理	314
8.2.1	进程	314
8.2.2	线程	316
8.2.3	调度	319
8.3	Mach 的存储管理	321
8.3.1	虚拟内存	322
8.3.2	存储共享	324
8.3.3	外部存储管理器	326
8.3.4	Mach 中的分布式共享存储	329
8.4	Mach 中的通信	330
8.4.1	端口	330
8.4.2	消息的发送与接收	334
8.4.3	网络消息服务器	338
8.5	Mach 的 UNIX 仿真	340
8.6	小结	341
习 题	342
第 9 章	实例研究 3: Chorus	343
9.1	Chorus 简介	343
9.1.1	Chorus 的发展史	343
9.1.2	Chorus 的设计目标	344
9.1.3	Chorus 系统结构	344
9.1.4	内核概念	346
9.1.5	内核结构	348
9.1.6	UNIX 子系统	349
9.1.7	面向对象子系统	349
9.2	Chorus 中的进程管理	349
9.2.1	进程	349
9.2.2	线程	350

9.2.3	调度.....	351
9.2.4	陷阱、异常和中断.....	352
9.2.5	进程管理的内核调用.....	353
9.3	Chorus 的内存管理.....	354
9.3.1	区域和段.....	354
9.3.2	映像程序 (Mapper)	355
9.3.3	分布式共享存储器.....	356
9.3.4	内存管理的内核调用.....	356
9.4	Chorus 中的通信.....	358
9.4.1	消息.....	358
9.4.2	端口.....	358
9.4.3	通信操作.....	359
9.4.4	通信的内核调用.....	360
9.5	Chorus 中的 UNIX 仿真.....	361
9.5.1	UNIX 进程的结构.....	362
9.5.2	对 UNIX 的扩展.....	362
9.5.3	Chorus 上 UNIX 的实现.....	363
9.6	COOL: 一个面向对象的子系统.....	367
9.6.1	COOL 的体系结构.....	367
9.6.2	COOL 基层.....	368
9.6.3	COOL 通用运行时系统.....	368
9.6.4	语言运行时系统.....	369
9.6.5	COOL 的实现.....	369
9.7	Amoeba、Mach 和 Chorus 的比较.....	369
9.7.1	指导思想.....	369
9.7.2	对象.....	371
9.7.3	进程.....	371
9.7.4	内存模式.....	372
9.7.5	通信.....	372
9.7.6	服务器.....	373
9.8	小结.....	374
	习题.....	375
第 10 章	实例研究 4: DCE	376
10.1	关于 DCE 的介绍.....	376
10.1.1	DCE 的历史.....	376
10.1.2	DCE 的目标.....	376
10.1.3	DCE 部件.....	377
10.1.4	信元.....	379
10.2	线程.....	381
10.2.1	DCE 线程介绍.....	381
10.2.2	调度.....	382

10.2.3	同步	383
10.2.4	线程调用	384
10.3	远程过程调用	387
10.3.1	DCE RPC 的目标	387
10.3.2	客户与服务器的编写	387
10.3.3	客户到服务器的绑定	389
10.3.4	RPC 的执行	390
10.4	时间服务	390
10.4.1	DTS 时间模型	391
10.4.2	DTS 实现	392
10.5	目录服务	394
10.5.1	名字	394
10.5.2	信元目录服务	395
10.5.3	全局目录服务	398
10.6	安全服务	401
10.6.1	安全模式	402
10.6.2	安全部件	403
10.6.3	许可证与鉴别码	404
10.6.4	认证过的 RPC	405
10.6.5	访问控制表 (ACL)	407
10.7	分布式文件系统	408
10.7.1	DFS 接口	409
10.7.2	服务器核心中的 DFS 部件	411
10.7.3	客户内核中的 DFS 部件	413
10.7.4	用户空间中的 DFS 部件	415
10.8	小结	416
习 题	417
第 11 章	读物列表与参考书目	419
11.1	阅读材料建议	419
11.1.1	介绍性和普通著作	419
11.1.2	分布式系统通信	420
11.1.3	分布式系统同步	420
11.1.4	分布式系统进程和处理机	421
11.1.5	分布式文件系统	422
11.1.6	分布式共享存储器	422
11.1.7	实例研究 1: Amoeba	422
11.1.8	实例研究 2: Mach	423
11.1.9	实例研究 3: Chorus	423
11.1.10	实例研究 4: DCE	423
11.2	文献目录 (按字母顺序排列)	424

第 1 章 分布式系统概述

计算机系统正在经历着一场革命。从 1945 年现代计算机时代开始到 1985 年前后，计算机是庞大而又昂贵的。即使是小型机，通常也每台价值数万美元。因此，大多数机构只有少数的几台计算机，同时，由于缺乏一种将它们连接起来的方法，所以这些计算机只能相互独立地运行。

但是，从 20 世纪 80 年代中期开始，技术上的两大进步开始改变这种状况。首先是功能更强的微处理机的开发，开始出现了 8 位的机型，随后不久 16 位，32 位，甚至 64 位的 CPU 也开始普及。其中许多机器具有较大主机（即大型机）的计算能力，但价格却只是它的几分之一。

在过去的半个世纪里，计算机技术取得了惊人的进步，这在其他工业中是前所未有的。从每台机器价格高达 1000 万美元，每秒执行一条指令，发展到目前售价 1000 美元而每秒执行 1000 万条指令，其性能价格比提高了 10^{11} 倍。如果在同一时期内汽车工业也能以这样的速度发展，那么现在一部劳斯莱斯牌汽车（Rolls Royce）将会只需要花 10 美元就可买到，而每加仑汽油就能行驶 10 亿英里（不幸的是，那时可能会有一本 200 页的手册告诉你该如何打开车门）。

第二个进步是高速计算机网络的出现。局域网 LAN 使得同一建筑内的数十甚至上百台计算机连接起来，使少量的信息能够在大约 1 毫秒左右的时间里在计算机间传送。更大量的数据则以 $10^7 \sim 10^8$ 比特/秒（bps）或更大的速率传送。广域网 WAN 使得全球范围内的数百万台计算机连接起来，传输速率从 64Kbps（每秒千位比特）到用于一些先进的实验型网络中的每秒千兆比特（gigabits）。

应用这些技术的结果，使得将大量 CPU 组成的计算系统通过高速网络连接在一起不仅成为可能，而且变得十分容易。相对于以前包括单个 CPU、存储器、外设和一些终端在内的集中式系统（又称单处理机系统 single processor system），它们通常被称为分布式系统（distributed systems）。

现在仅存在一个比较棘手的问题，那就是软件。分布式系统需要与集中式系统完全不同的软件。特别是系统所需要的操作系统只是刚刚出现。虽然分布式系统已经向前迈出了最初的几步，但仍有很长的一段路要走。对于分布式操作系统，我们对它的一些基本思想的介绍到这里已经足够了。接下来，本书将致力于研究分布式操作系统的概念、实现和几个实例。

1.1 什么是分布式系统？

分布式系统有很多不同的定义，但其中没有一个是令人满意或者能够被所有人接受的。介绍分布式系统，对它的特点的给予下面大致的描述足够了：

“一个分布式系统是一些独立的计算机的集合，但是对这个系统的用户来说，系统

就像一台计算机一样。”

这个定义有两个方面的含义：第一，从硬件角度来讲，每台计算机都是自主的；第二，从软件角度来讲，用户将整个系统看作是一台计算机。这两者都是必需的，缺一不可。在简要介绍有关硬件、软件的一些背景材料之后，我们将再回到这两点上来进行讨论。

由于给出分布式系统的一些实例可能要比进一步地深入研究定义更有帮助，下面就给出一些分布式系统的例子。第一个例子，设想一个大学或公司部门内的工作站网络。除了每个用户的个人工作站外，机房中可能还有一个共享的处理机池(pool of processor)，这些处理机并没有分配给特定的用户，而是在需要的时候进行动态分配。这样的系统可能会有一个单一的文件系统，其中所有的文件可以从所有的计算机上以相同的方式并且使用相同的路径名存取。另外，当一个用户输入一条命令时，系统能够找到一个最好的地方执行该命令。这可能是在用户自己的工作站上，可能是在别人空闲的工作站上，也可能在机房里一个未分配的处理机上。如果这个从系统整体上看以及运行起来看都像一个典型的单处理机分时系统，那么就可以称它为一个分布式系统。

第二个例子，考虑一个到处是机器人的工厂。每个机器人都有一台功能强大的计算机用于处理视觉、进行计划、通信以及其他任务。当装配线上的某个机器人发现一个它要安装的零件有缺陷时，它就要求该零件供应部门的另一个机器人给它送一个替代品。如果所有的机器人都如同连接于同一中心计算机上的外设一样工作，而且系统的程序也是以这种方式进行编制的话，那么它也是一种分布式系统。

最后一个例子是一个在世界各地有数百个分支机构的大银行。每个分支机构有一台主计算机存储当地帐目和处理本地事务。此外，每台计算机还能与其他分支机构的计算机及总部的计算机对话。如果不管顾客和帐目在哪里都能够进行交易，而且用户也不会感到当前这个系统与被替代的原先的集中式主机有何不同，那么这个系统也被认为是一个分布式系统。

1.2 目标

我们已经能够建立一个分布式系统，但这并不意味着建立一个分布式系统一定是一个好主意。就像以目前的技术水平，我们可以在一台计算机中装入四个软盘驱动器，而问题在于这样做并没有什么意义一样。本节中，我们将讨论典型分布式系统的动机和目的，分析它与传统的集中式系统相比较所具有的优点和缺点。

1.2.1 分布式系统与集中式系统相比较而言的优点

系统倾向于分布式发展潮流的真正驱动力是经济。25年前，计算机权威和评论家 Herb Grosch 指出 CPU 的计算能力与它的价格的平方成正比，后来成为 Grosch 定理。也就是说如果你付出两倍的价钱，就能获得四倍的性能。这一论断与当时的大型机技术非常吻合，因而使得许多机构都尽其所能购买最大的单个大型机。

随着微处理机技术的发展，Grosch 定理不再适用了。现在人们只需花几百美元就能买到一个 CPU 芯片，这个芯片每秒钟执行的指令比 80 年代最大的大型机的处理器每秒钟所执行的指令还多。如果你愿意付出两倍的价钱，将得到同样的 CPU，但它却以更高的

时钟速率运行。因此，最节约成本的办法通常是在一个系统中使用集中在一起的人量的廉价 CPU。所以，倾向于分布式系统的主要原因是它可以潜在地得到比单个的大型集中式系统好得多的性能价格比。实际上，分布式系统是通过较低廉的价格来实现相似的性能的。

与这一观点稍有不同的是，我们发现微处理机的集合不仅能产生比单个大型主机更好的性能价格比，而且还能产生单个大型主机无论如何都不能达到的绝对性能。例如，按目前的技术，我们能够用 10,000 个现代 CPU 芯片组成一个系统，每个 CPU 芯片以 50 MIPS（每秒百万指令）的速率运行，那么整个系统的性能就是 500,000 MIPS。而如果单个处理机（即 CPU）要达到这一性能，就必需在 2×10^{-12} 秒（2 微微秒，0.002 纳秒）的时间内执行一条指令，然而没有一个现存的计算机能接近这个速度，从理论上和工程上考虑都认为能达到这一要求的计算机都是不可能存在的。理论上，爱因斯坦的相对论指出光的传播速度最快，它能在 2 微微秒内传播 0.6 毫米。实际上，一个包含于边长为 0.6 毫米大小的立方体内的具有上面所说的计算速度的计算机产生大量的热量就能将它自己立即熔掉。所以，无论是要以低价格获得普通的性能，还是要以较高的价格获得极高的性能，分布式系统都能够满足。

另一方面，一些作者对分布式系统和并行系统进行了区分。他们认为分布式系统在设计用来允许众多用户一起工作的，而并行系统的唯一目标就是以最快的速度完成一个任务，就像我们的速度为 500,000 MIPS 的计算机那样。我们认为，上述的区别是难以成立的，因为实际上这两个设计领域是统一的。我们更愿意在最广泛的意义上使用“分布式系统”一词来表示任何一个有多个互连的 CPU 协同工作的系统。

建立分布式系统的另一原因存在于一些应用本身是分布式的。一个超级市场连锁店可能有许多分店，每个商店都需要采购当地生产的商品（可能来自本地的农场）进行本地销售，或者要对本地的某些蔬菜因时间太长或已经腐烂而必须扔掉作出决定。因此，每个商店的本地计算机掌握存货清单是有意义的，而不是集中于公司总部。毕竟，大多数查询和更新都是在本地进行的。然而，连锁超级市场的高层管理者也经常了解他们目前还有多少甘蓝。实现这一目标的一种途径就是将整个系统建设成对于应用程序来说就像一台计算机一样，但是在实现上它是分布的，像我们前面所描述的一个商店有一台机器。这就是一个商业分布式系统。

另一种固有的分布式系统是通常被称为计算机支持下的协同工作系统（CSCW，Computer Supported Cooperative Work）。在这个系统中，一组相互之间在物理上距离较远的人员可以一起进行工作，例如，写出同一份报告。就计算机工业的长期发展趋势来说，人们可以很容易地想像出一个全新领域——计算机支持的协同游戏（CSCG：Computer Supported Cooperative Games）。在这个游戏中，不在同一地方的游戏者可以实时地玩游戏。你可以想像，在一个多维迷宫中玩电子捉迷藏，甚至是一起玩一场电子空战，每个人操纵自己的本地飞行模拟器去试着击落别的游戏者的飞机，每个游戏者的屏幕上都显示出其飞机外的情况，包括其他飞入它的视野的飞机。

同集中式系统相比较，分布式系统的另一个潜在的优势在于它的高可靠性。通过把工作负载分散到众多的机器上，单个芯片故障最多只会使一台机器停机，而其他机器不会受任何影响。理想条件下，某一时刻如果有 5% 的计算机出现故障，系统将仍能继续工作，只不过损失 5% 的性能。对于关键性的应用，如核反应堆或飞机的控制系统，采用分布式

系统来实现主要是考虑到它可以获得高可靠性。

最后，渐增式的增长方式也是分布式系统优于集中式系统的一个潜在的重要原因。通常，一个公司会买一台大型主机来完成所有的工作。而当公司繁荣扩充时，工作量就会增大，当其增大到某一程度时，这个主机就不能再胜任了。仅有的解决办法是要么用更大型的机器（如果有的话）代替现有大型主机，要么再增加一台大型主机。这两种作法都会引起公司运转混乱。相比较之下，如果采用分布式系统，仅给系统增加一些处理机就可能解决这个问题，而且这也允许系统在需求增长的时候逐渐进行扩充。表 1-1 中总结了以上这些优点。

表 1-1 分布式系统相对于集中式系统的优点

项目	描 述
经济	微处理机提供了比大型主机更好的性能价格比
速度	分布式系统总的计算能力比单个大型主机更强
固有的分布性	一些应用涉及到空间上分散的机器
可靠性	如果一个机器崩溃,整个系统还可以运转
渐增	计算能力可以逐渐有所增加

从长远的角度来看，主要的驱动力将是大量个人计算机的存在和人们共同工作与信息共享的需要，这种信息共享必需是以一种方便的形式进行的，而不受地理或人员、数据以及机器的物理分布的影响。

1.2.2 分布式系统与独立 PC 机相比较的优点

既然使用微处理机是一种节省开支的办法，那么为什么不给每个人一台个人计算机，让他们各自独立地工作呢？一则，许多用户需要共享数据。例如，机票预订处的工作人员需要访问存储航班以及现有座位信息的主数据库。假如给每个工作人员都备份整个数据库，那么实际上这是无法工作的，因为没有人知道其他工作人员已经卖出了哪些座位。共享的数据是上例和许多其他应用的基础，所以计算机间必须互连。而计算机互连就产生了分布式系统。

共享并不只是仅仅涉及数据。昂贵的外设，例如彩色激光打印机，照相排版机以及大型存储设备（如自动光盘点唱机）都是共享资源。

将一组孤立的计算机连成一个分布式系统的第三个原因是它可以增强人与人之间的沟通，电子邮件比信件、电话和传真有更多的诱人之处。它比信件快得多，不像电话需要两人同时都在，也不像传真，它所产生的文件可在计算机中进行编辑、重排和存储，也可以由文本处理程序来处理。

最后，分布式系统比给每个用户一个独立的计算机更灵活。尽管一种可能的模式是给每个人一台个人计算机并将它们通过 LAN 联在一起，但这种方式并不是唯一的。另外还存在一种模式是将个人计算机和共享计算机混合连接在一起（这些机器的型号可能并不完全相同），使工作能够在最合适的计算机上完成，而并不总是在自己的计算机上完成。这种方式可以使工作负荷能更有效地在计算机系统中进行分配。系统中某些计算机的失效也可以通过使其工作在其他计算机上进行而得到补偿。表 1-2 总结了以上所介绍的各点。

表 1-2 分布式系统相对于孤立的(个人)计算机的优点

项目	描 述
数据共享	允许多个用户访问一个公共的数据库
设备共享	允许多个用户共享昂贵的外围设备(如彩色打印机)
通信	使得人们之间的通信更加容易,如通过电子邮件
灵活性	用最有效的方式将工作负荷分配到可用的机器上

1.2.3 分布式系统的缺点

尽管分布式系统有许多优点,但也有缺点。本节就将指出其中的一些缺点。我们前面已经提到了最棘手的问题:软件。就目前的最新技术发展水平,我们在设计、实现及使用分布式系统上都没有太多的经验。什么样的操作系统、程序设计语言和应用适合这一系统呢?用户对分布式系统中分布式处理又应该了解多少呢?系统应当做多少而用户又应当做多少呢?专家们的观点不一(这并不是因为专家们与众不同,而是因为对于分布式系统他们也很少涉及)。随着更多的研究的进行,这些问题将会逐渐减少。但是目前我们不应该低估这个问题。

第二个潜在的问题是通信网络。由于它会损失信息,所以需要专门的软件进行恢复。同时,网络还会产生过载。当网络负载趋于饱和时,必须对它进行改造替换或加入另外一个网络扩容。在这两种情况下,一个或多个建筑中的某些部分必须花费很高的费用进行重新布线,或者更换网络接口板(例如用光纤)。一旦系统依赖于网络,那么网络的信息丢失或饱和将会抵消我们通过建立分布式系统所获得的大部分优势。

最后,上面我们作为优点来描述的数据易于共享性也是具有两面性的。如果人们能够很方便地存取整个系统中的数据,那么他们同样也能很方便地存取与他们无关的数据。换句话说,我们经常要考虑系统的安全性问题。通常,对必须绝对保密的数据,使用一个专用的、不与其他任何机器相连的孤立的个人计算机进行存储的方法更可取。而且这个计算机被保存在一个上锁的十分安全的房间中,与这台计算机配套的所有软盘都存放在这个房间中的一个保险箱中。分布式系统的缺点如表 1-3 所示。

表 1-3 分布式系统的缺点

项目	描 述
软件	目前为分布式系统开发的软件还很少
网络	网络可能饱和和引起其他的问题
安全	容易造成对保密数据的访问

尽管存在这些潜在的问题,许多人还是认为分布式系统的优点多于缺点,并且普遍认为分布式系统在未来几年中会越来越重要。实际上,在几年之内许多机构会将他们的大多数计算机连接到大型分布式系统中,为用户提供更好、更廉价和更方便的服务。而在十年之后,中型或大型商业或其他机构中可能将不再存在一台孤立的计算机了。

1.3 硬件概念

尽管所有的分布式系统都包含多个 CPU，但仍有一些不同的硬件组织方法，特别体现在这些 CPU 的互连方式及通信方式上。本节将简要介绍分布式系统的硬件，特别是要了解一下多台计算机是如何联结在一起的。下一节将探讨与分布式系统有关的软件问题。

过去几年中人们已经提出过许多 CPU 计算机系统的分类方案，但却没有一种方案真正流行或者被广泛采用。在这些方案中最经常被引用的是 Flynn 的 (1972) 分类方案，虽然它还相当不成熟。Flynn 提出两个他认为十分重要的特征：指令流的数量和数据流的数量。

第一类是具有单指令流、单数据流的计算机，它被称为 SISD (Single Instruction stream, Single Data stream)。所有传统的单处理机 (即，那些只有一个 CPU 的) 计算机，从个人计算机到大型主机，都属于这一类。

第二类是 SIMD (Single Instruction stream, Multiple Data stream)，它有一个指令流和多个数据流。这种类型是指有一个能取一条指令的指令单元的处理机阵列结构。在这个结构中，指令单元取出一条指令后，操纵许多数据单元并行地执行这条指令，而且每个数据单元都有它自己的数据。这种类型的计算机在用多组数据重复进行同样的计算时是非常有用的，例如，把有 64 个独立向量的所有元素累加起来。一些超级计算机就属于 SIMD 型。

第三类为 MISD (Multiple Instruction stream, Single Data stream)。此类型计算机有多条指令流，一条数据流。我们已知的计算机中没有属于这一类的。

最后一类是 MIMD (Multiple Instruction stream, Multiple Data stream)，它在本质上是一组独立的计算机，每个计算机有自己的程序计数器、程序和数据。

所有的分布式系统都是 MIMD 型，所以这种分类系统对于我们来说不是非常有用的。

虽然 Flynn 的分类到此为止，但我们还要更深入地进行分类。在图 1-1 中，我们将所有的 MIMD 计算机群分成两类：那些具有共享存储器的通常称为多处理机 (multiprocessor) 或多处理器；而不具有共享存储器的则称为多计算机 (multicomputer)。它们之间的本质区别在于：在多处理机中，所有的 CPU 共享统一的虚拟地址空间。例如，如果任何一个 CPU 将数值 44 写入地址 1000 中，随后任何其他 CPU 将会从它的地址 1000 中读出数值 44，所有的机器共享同一个存储器。

相反，在多计算机中，每个计算机有它自己私有的存储器。如果一个 CPU 将数值 44 写入它的地址 1000 中，而当另一个 CPU 读地址 1000 中的数据时，将会得到该地址中以前写入的值。写入数值 44 根本没有影响到它自己的存储器。由网络连接的个人计算机的集合就是一个多计算机系统的普通实例。

这两种类型又可以分别根据互连网络的体系结构进一步进行细分。在图 1-1 中，我们将这两种分类描述为总线型 (bus) 和交换型 (switched)。所谓总线型是指只通过单个网络、底板、总线、电缆或其他介质将所有计算机联结起来。有线电视采用的就是与此十分类似的方案，即电缆公司在街道下面布线，所有的用户都通过分接头将他们的电视与这条总线联结起来。

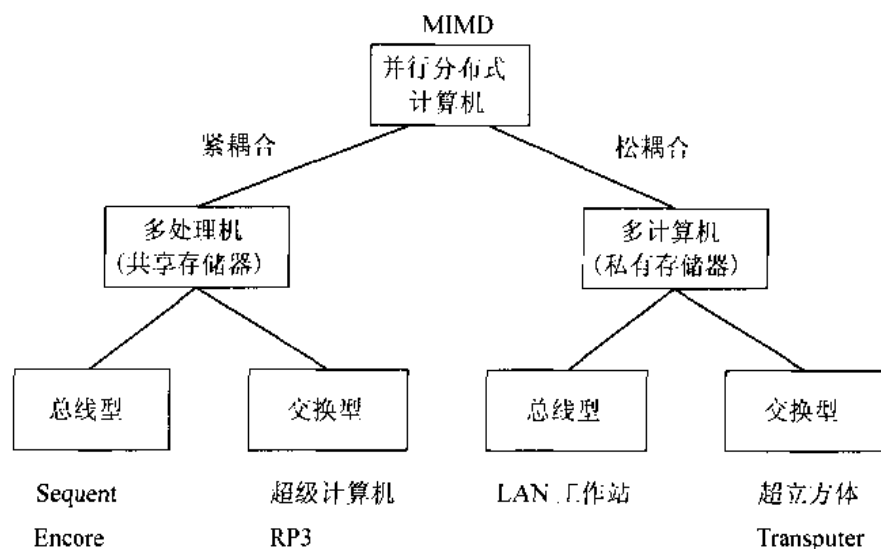


图 1-1 并行及分布式计算机系统分类法

交换型系统并不像有线电视那样有一个网络主干，而是在机器和机器之间有独立的线路，在实际使用中还有许多不同的连线方式。信息沿着线路传送，在每一步都需要进行明确的路由选择以将信息通过某个输出线路发送出去。世界范围的公共电话系统就是采用这种方式组织的。

另一种分类方法是，在一些系统中的机器是紧耦合型(tightly coupled)的，而在另一些系统中它们是松耦合型(loosely coupled)的。在紧耦合的系统中，一台计算机向另一台计算机发送信息的时延很短、数据传输速率高；也就是说它每秒钟所能够传送的比特数大，而在松耦合的系统中则正好相反：机器间信息传送延迟大，数据传输速率也低。例如，同一个印刷电路板上由蚀刻在板上的电路连接的两个 CPU 芯片很可能是紧耦合的。然而，由 2400 比特/秒 (bps) 的调制解调器通过电话系统连接到一起的两台计算机必定是松耦合。

紧耦合的系统多用于并行系统（共同处理一个问题），而松耦合系统多用于分布式系统（处理一些不相关的问题）。然而情况并不总是这样，如一个著名的反例就是：全世界的数百台计算机一起工作，试图对一个巨大的数字（大约 100 位）进行分解因子，每台计算机被安排只计算某个指定范围中的除数。他们都在空闲的时间做这件事，完成后用电子邮件报告结果。

总的来说，多处理机的耦合程度要比多计算机高，因为它们能以存储速率交换数据，但一些基于光纤的多计算机也能以存储速率交换数据。尽管紧耦合和松耦合的概念很模糊，但它们是很有用的概念，就像说“杰克胖，吉尔瘦”表明了有关他们腰身的信息一样，使你对胖和瘦的概念进行大量的讨论。

在下面四节中，我们将更详细介绍图 1-1 中所示的四个类别，即总线型多处理机、交换型多处理机、总线型多计算机、交换型多计算机。尽管这些问题与我们所关心的主题——分布式操作系统没有直接关系，但它们却能够有助于我们更好地阐明主题，正如我们将会看到的，不同类别的计算机系统应采用不同的操作系统。

1.3.1 基于总线的多处理机

基于总线的多处理机是由若干个 CPU 组成的，它们都连接到一个公共的总线上，并且共享一个存储器模块。只要有一个 CPU 和存储器卡插在高速底板或母板上就可以实现一个简单的结构配置。典型的总线有 32 根或 64 根地址线，32 根或 64 根数据线，还有约 32 根或更多的控制线，它们都是并行操作的。为了要读取存储器中的一个字 (word)，CPU 首先将它想要读取的字的地址放到总线的地址线上，然后给控制线发送一个适当的信号表明它想要进行读操作。存储器进行响应将该地址的内容送到数据线上以使 CPU 可以对它进行读取。写操作也以相似的方式进行。

由于存储器只有一个，当 CPU A 给存储器写入一个字，一微秒后 CPU B 在读出该字的内容时会得到 A 刚写入的那个值。具有这种特性的存储器被称为是相关的 (coherent)。相关性在分布式操作系统中以不同的方式发挥重要的作用，这一点我们将在后面进行详细讨论。

这种实现方案的问题在于，只要有仅仅 4 到 5 个 CPU 时，总线就会经常过载，导致性能也会急剧下降。解决的办法是在 CPU 和总线之间增加一个高速缓冲存储器 (cache memory)，如图 1-2 所示。缓冲存储器保留着最近刚存取过的字。所有的内存访问请求都要经过它。如果请求的字在缓冲存储器中，缓冲存储器就会直接响应 CPU，而不产生总线请求。如果缓冲存储器足够大的话，那么成功的可能性 (称为命中率) 将是很高的。而且每个 CPU 的总线通信量也会急剧下降，系统中也就能够容纳更多的 CPU。通常，缓冲存储器的大小从 64K 到 1M，命中率经常可以达到 90% 或更高。

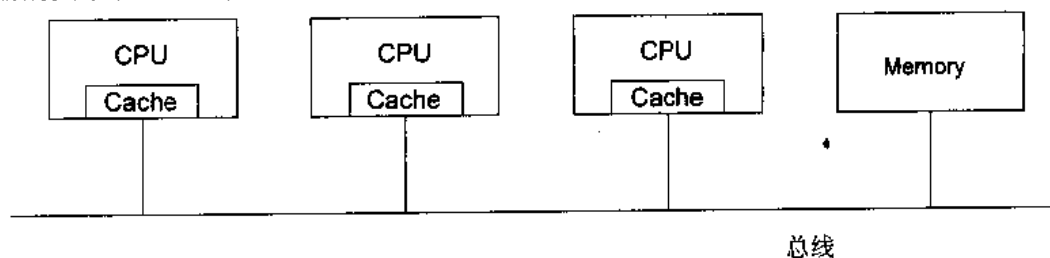


图 1-2 基于总线的多处理机

然而，缓冲存储器的引入也带来了一个严重的问题。假定有两个 CPU：A 和 B，它们将同一个字分别读入自己的缓冲存储器内。在 A 改写这个字后，当 B 再读该字时，从它的缓冲存储器中得到的是原来的旧值，而不是 A 刚写入的值。这时，存储器就是不相关的，使得系统难以进行编程。

许多研究者已经研究了这个问题，也找到了许多解决办法。下面我们就简要地介绍其中的一个。这个方法是将缓冲存储器设计成无论何时在对其写入一个字时，该字就能同时写入存储器中。这种缓冲存储器，并不令人惊讶，称为通写缓冲存储器 (write-through cache)。在此设计中，缓冲存储器读命中不会引起任何总线负载，但是缓冲存储器读失败和所有的写命中和写失败都会引起总线负载。

另外，所有的缓冲存储器都不断监听总线。当一个缓冲存储器发现它所存储的某个地址的字在存储器中正被进行写操作时，它要么从它的存储器中去掉那个字，要么把该存

存储器字更改为新值。这样的缓冲存储器称为监听缓冲存储器 (snoopy cache)，因为它总是在对总线进行“监听”(窃听)。一个包含监听通写缓冲存储器的设计是相关的，并且对于编程人员来说也是不可见的。几乎所有基于总线的多处理机都使用这种或者与之十分类似的体系结构。使用这种结构就能够把 32 个或 64 个 CPU 放在一条总线上。关于基于总线多处理机的更详细的内容请参见 Lilja(1993)。

1.3.2 交换型多处理机

要建立一个拥有 64 个以上处理机的多处理机系统就需要采用一种不同的方法把这些 CPU 和存储器连接起来。一种可能性就是将存储器分成许多存储模块，用十字交叉开关将它们与 CPU 相连。如图 1-3 (a) 所示。在图中我们可以看出，每个 CPU 和每个存储器模块都有一个外部接口同这个交换开关相连。在每个交叉点上都有一个微小的能够用硬件打开或关闭的电子交叉点开关。当 CPU 想要访问一个特定的存储器模块时，连接它们的交叉点就暂时闭合，允许访问。许多 CPU 能够同时访问存储器是十字交叉开关的优点，尽管如果两个 CPU 想要同时访问同一个存储器模块，其中的一个必须等待另外一个访问结束之后才能够进行。

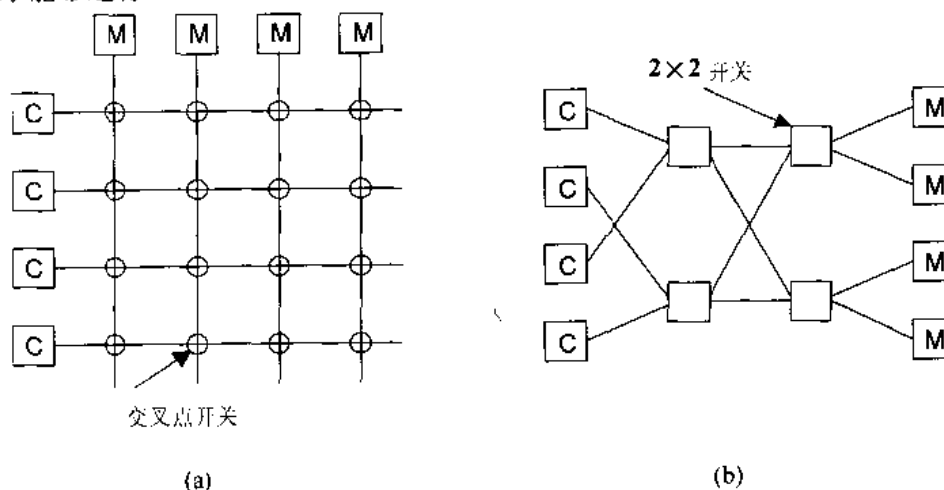


图 1-3 (a) 交叉开关
(b) omega 网络(C: 表示 CPU, M: 表示 Memory)

交叉开关的不足之处在于当有 n 个 CPU 和 n 个存储器模块时，需要 n^2 个交叉开关。如果 n 值很大的话，那么这个数字就会非常大。为了避免这种情况的发生，人们已经找出了替代的交换网络，它只需要较少的开关。图 1-3 (b) 中的 omega 网络就是替代这种网络的一个实例。在这个网络中包括 4 个 (2×2) 交换开关，每个交换开关都有两个输入和两个输出，而且任何一个输入都可以和任何一个输出相连接。如果我们仔细研究此图就会发现，只要正确地设置开关，任何一个 CPU 都可以访问所有的存储器。而这些交换开关的设定工作可以在纳秒级或更少的时间内完成。

在通常的情况下，如果有 n 个 CPU 和 n 个存储器模块，omega 网络就要 $\log_2 n$ 个交换开关级，每级包括 $n/2$ 个交换开关，总共 $(n \log_2 n)/2$ 个交换开关。虽然当 n 值很大时，

它要比 n^2 好(远小于 n^2), 但实际数量仍然相当大。

另外还有一个问题就是时延。例如, 当 $n=1024$ 时, 从 CPU 到存储器有 10 个交换开关级, 而还有另外的 10 个交换开关级用于返回所请求的字。假定 CPU 是现代的 RISC 芯片, 其速度是 100MIPS, 也就是执行一条指令的时间是 10 纳秒。如果一个存储器请求及返回结果必须在 10 纳秒内通过所有的 20 个交换开关级 (10 个用于传送出请求而另外 10 个用于传送返回结果), 那么交换开关的交换时间必须为 500 微微秒 (0.5 纳秒)。根据前面的计算, 整个多处理机系统需要 5120 个 500 微微秒级的交换开关, 这是十分昂贵的。

有人曾经尝试通过采用分层系统减少成本。每个 CPU 都同一些存储器相连接。每个 CPU 可以快速访问它自己的本地存储器, 但是访问其他 CPU 的存储器就会慢许多。这种设计产生了称为 NUMA (非一致存储访问 NonUniform Memory Access) 的机器。尽管 NUMA 机器比基于 omega 网络的机器有更好的平均访问时间, 但是它也有新的复杂的问题, 为了使大部分访问集中在本地的存储器中, 程序和数据的放置位置就十分关键了。

总之, 基于总线的多处理机, 即使有监听缓冲存储器, 由于受总线容量的限制, CPU 最多约 64 个。要超过这个数目就需要采用交换网络, 例如十字交叉开关、omega 交换网络或一些其他的类似结构。大型的十字交叉开关价格十分昂贵, 而大型的 omega 网络也是既贵又慢。NUMA 机器需要为好的软件布局提供复杂的算法。我们的结论十分清楚, 建立一个大的、紧耦合的、共享存储器的多处理机 (系统) 是可能的, 但这是困难且价格昂贵的。

1.3.3 基于总线的多计算机

另一方面, 建立一个多计算机系统 (即, 非共享存储器) 是容易的。每个 CPU 都与它自身的存储器直接相连。唯一遗留的问题是 CPU 间应如何通信。很明显, 这里也需要一些互连方案, 但是由于它只用于 CPU 和 CPU 之间的通信, 通信量要比当互连网络用于 CPU 和存储器之间的通信量低几个数量级。

在图 1-4 中, 我们看到一个基于总线的多计算机。它的拓扑结构虽然看起来与基于总线的多处理机很相似, 但是因为其上的通信量很少, 所以并不需要高速的背板总线。事实上, 它可能是更低速的 LAN (相比于 300 Mbps 和需要背板总线支持的速率而言, 它典型的速率只有 10-100Mbps)。因此, 图 1-4 更通常的是一个 LAN 上工作站的集合, 而不是插入高速总线的 CPU 卡的集合 (尽管后一种结构一定是一种可能的设计)。

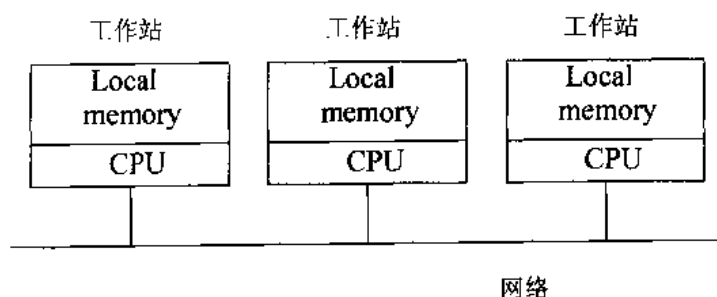


图 1-4 局域网网上由多台工作站组成的多计算机系统

1.3.4 交换型多计算机

最后一个类别是交换型多计算机。虽然人们已经提出并建立了各种互连网络，但是它们都有一个特性，就是每个 CPU 都直接地而且排它地访问自己的私有存储器。图 1-5 给出了两个流行的拓扑结构：网格和超立方体。网格结构很容易理解，它被布置在印刷电路板上。它们最适用于解决固有特性为二维的问题，例如图像理论或视觉（即，机器人的眼睛或分析照片）。

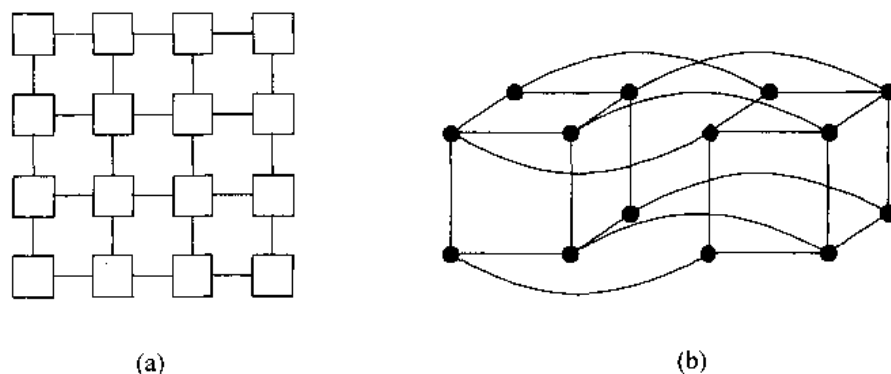


图 1-5 (a) 网格
(b) 超立方体

超立方体(hypercube)是一种 n 维立方体。图 1-5 (b) 中的超立方体是四维的。它可被想像成两个普通的立方体，每个立方体有 8 个顶点和 12 条边。每个顶点都代表一个 CPU，每条边代表两个 CPU 间的连接。两个立方体中相应的顶点是相连的。

要把超立方体扩展为五维，就要在图形中再加另外两个互连的立方体，连接两部分中相应的边，其他的情况也可以此类推。对于一个 n 维的超立方体，每个 CPU 都有 n 个连接与其他 CPU 相连。因此，布线的复杂性按 CPU 数量的对数增加。因为只有最相邻的 CPU 是互连的，许多信息在到达终点前必须进行 n 次跳跃。然而，最长的可能路径也是以 CPU 数量的对数增加的，而不像网格中按 CPU 数量的平方根增加。有 1024 个 CPU 的超立方体在几年前就已经在商业上应用了，而具有 16384 个 CPU 的超立方体也即将可以进行商业应用。

1.4 软件概念

虽然硬件是十分重要的，但是软件更为重要。系统呈现给用户的印象以及用户如何看待系统大都取决于系统的操作系统软件，而不是硬件。本节将介绍上面讨论过的多处理机和多计算机中使用的各类操作系统，同时也将讨论哪种软件与哪种硬件相匹配。

操作系统不像硬件一样可以进行清晰、明确的归类。就其本质而言，软件是模糊的和无定型的。但是，在多 CPU 系统中区分两种不同的操作系统总还或多或少是可能的，它们就是松耦合系统和紧耦合系统。正如我们将要看到，松耦合和紧耦合的软件与相应的

松耦合和紧耦合硬件大致相似。

松耦合的软件允许分布式系统的机器和用户基本上各自独立，但是也在必要的情况下进行一定程度的相互作用。假设有这样一组个人计算机，每个机器都有它自己的 CPU、存储器、硬盘和操作系统，但是它们通过 LAN 共享一些资源，例如激光打印机、数据库等。因为每一台计算机都能够明显地同其他计算机区别开来，每一台也都有自己的任务，所以，这个系统是松耦合的。如果网络因某种原因而崩溃，各个单个的计算机虽然会失去一些功能（例如，打印文件的能力），但是它们在很大程度上仍能继续工作。

为了说明在这个领域内下定义是多么的困难，现在让我们考虑一个与上面相同的系统，只是没有网络。要打印一个文件，用户将该文件写在一张软盘上，将软盘放到有打印机的机器上读入，然后再打印出来。这还是一个分布式系统吗？它仅仅是耦合度更低吗？这很难断定。从基本的观点来看，通过 LAN 进行通信和通过使用软盘进行通信在理论上没有本质的区别。最多人们会说，用软盘通信的延迟更大，数据传输速率更低而已。

在另一个极端的例子中，我们可能会发现一个专门用于并行运行一个下棋程序的多处理机。每个 CPU 都被指定一个棋局进行推演，它分析盘面和所有可能从中产生的盘面。当推演完毕后，CPU 汇报其结果，同时又被指定一个新的盘面进行推演。这个系统的软件，包括它的应用程序以及支持它的操作系统，与前面给出的例子相比很明显是紧耦合的。

现在我们已经讨论过四种分布式硬件和两种分布式软件。从理论上来说，硬件和软件应该有八种结合方式，而事实上只有四种是值得区分出来的，因为对于用户来说，互连技术是不可见的。对大多数应用来说，一个多处理机系统无论是使用有监听缓冲存储器的总线还是使用 omega 网络，都看成是无区别的一个多处理机系统。下面几节将了解一些最常用的软、硬件组合。

1.4.1 网络操作系统

让我们先从松耦合硬件上的松耦合软件开始，因为这可能是许多机构中最常用的组合。最典型的例子是通过 LAN 连接的工作站网络。在这种模式中，每个用户都有自己专用的工作站。虽然这个工作站可以有，也可以没有硬盘，但是它一定有自己的操作系统。所有的命令通常在本地运行，即就在工作站上运行。

然而，有时用户也很可能通过使用如下的一个命令登录到另一个远程工作站上，如：

```
rlogin machine
```

这个命令的作用是把用户自己的工作站变为登录到远程机器上的一个远程终端。键盘输入的命令被送到远程机器上，而那个远程机器上的输出也会显示在用户的屏幕上。要连接不同的远程机器，首先需要退出当前的系统，然后再用 rlogin 命令连接到需要的机器上。在任一时刻只能使用一台远程机器，而且机器的选择也是完全由人工操作完成的。

工作站网络也经常有一个远程拷贝命令将文件从一台机器拷贝到另一台机器上。例如，类似于下面的一条命令：

```
rep machine1 : file1 machine2 : file2
```

就能把文件 file1 从机器 machine1 拷贝到机器 machine2 上，并且给它取名为 file2。在这里，文件的移动是清楚明确的，而且要求用户完全清楚所有文件的位置在哪里和所有的命令在哪里执行。

尽管这种通信方式非常原始，而且已经有系统设计者在寻找更简便的形式进行通信和信息共享，但有这样一种方法总比没有强。一种方法就是提供一种共享的、所有工作站都可存取的全局文件系统。文件系统由一个或多个称为文件服务器（File Server）的机器支持。文件服务器接受运行在称为客户的其他机器（非文件服务器）上的用户程序的读写文件的请求。文件服务器检测、执行每个收到的请求，然后返回应答。如图 1-6 所示。

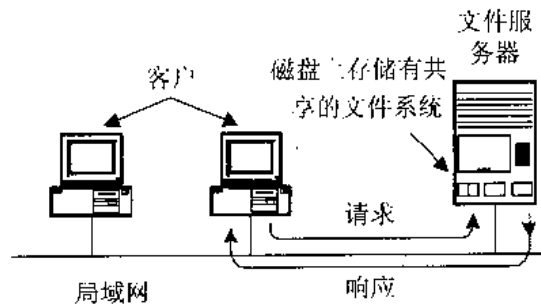


图 1-6 在网络操作系统环境下的两个客户和一个服务器

文件服务器经常使用的是一些分层的文件系统，每个系统都有根目录，它包含子目录和文件。工作站可以引入或装入这些文件系统，这就将存放在服务器上的文件系统增加到它们本地的文件系统中。例如，在图 1-7 中，有两个文件服务器，一个服务器中有一个目录叫 *games*，另一个服务器中有一个目录叫 *work*。这些目录每个都包含一些文件。图中的两个客户都装入了这两个服务器，但是它们却是在它们各自文件系统的不同地方进行装载的。客户 1 将它们装在它的根目录上，这样就能够分别访问 */games* 和 */work* 目录。客户 2 同客户 1 一样，将 *games* 装在它的根目录上，但是由于它将邮件和新闻的读取认作为一种游戏，于是它创建了目录 */games/work* 并将服务器上的 *work* 装在那里。因此，它访问 *news* 就是通过路径 */games/work/news* 而不是 */work/news*。

客户将服务器安装在它目录层的哪一个位置并不重要，重要的是我们应该注意到不同的客户可能会以不同的方式看待文件系统。这样，文件名取决于它的存取位置以及计算机如何建立它的文件系统。因为每个工作站都相对独立于其他工作站工作，所以不能保证它们都能为自己的程序提供相同的目录层次结构。

用于这种环境的操作系统必须管理独立的工作站和文件服务器，同时还要负责它们之间的通信。所有的计算机可能都运行同一种操作系统，但这并不是必需的。如果客户和服务器运行的是不同的操作系统，那么，它们至少需要在它们所可能进行交换的所有信息的格式和含义上取得一致。在类似情况下（即每台机器都高度自主，很少有系统范围的请求），人们通常称之为网络操作系统。

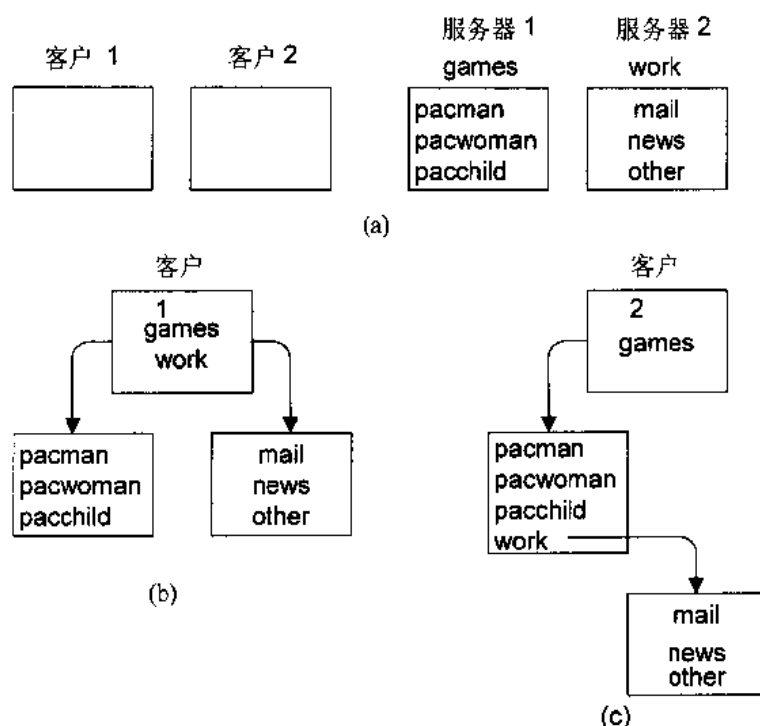


图 1-7 不同的客户可以在不同的地方安装服务器

1.4.2 真正的分布式系统

网络操作系统是运行在松耦合硬件上的松耦合的软件。除了共享文件系统外，对用户显而易见的是这个系统包含有许多计算机。每台计算机都运行它自己的操作系统，而且按它的所有者的要求工作。除了客户/服务器通信必须遵守协议之外，计算机之间基本上没有什么合作。

这种系统的下一步革命是在同样的松耦合硬件（例如多计算机）上的紧耦合软件。这样一个系统的目标是使用户产生一个错觉：整个计算机网络是一个分时系统，而不是一个互不相同的机器的集合。一些人称这种性质为单系统映像（single-system image），而另一些人的看法则稍有不同。他们认为，分布式系统是在网络计算机的集合上运行，但行为却像一个虚拟的单处理机（virtual uniprocessor）。不管人们如何对它进行描述，根本思想是：用户不必意识到系统中有多 CPU 存在。虽然目前还没有一个系统能完全满足这一要求，但是已经有一些系统开始初露端倪。本书的后面将对这些系统进行讨论。

分布式系统的特点是什么？首先必须有一个单一的、全局的进程间的通信机制，从而使任何进程都可以和其他进程进行通信。在不同的计算机上不应有不同的机制，本地通信和远程通信也不应有不同的机制。还必须有一个全局的保护方案。把存取控制表、UNIX 保护位以及通信能力简单混合并不能产生单系统的映像。

进程管理也必须处处相同。进程如何建立、撤消、启动、停止都不能因机器不同而

不同。简而言之，网络操作系统的核心思想：即任何机器只要在进行客户-服务器通信时遵守标准协议，它就可做它想要做的任何事。这是不够的，因此，不仅要有唯一的一整套在所有的计算机上都可以使用的系统调用，而且这些系统调用的设计必须符合分布式环境的要求。

在任何地方，文件系统也必须看起来是相同的。如果在一些地方文件名被限制为 11 个字符，而在另一些地方不作限制是不可行的。同时，每个文件应该是在所有地方都是可见的，当然，这必须遵守保护和安全性约束的限制。

在系统的所有地方都使用相同的系统调用接口，作为它的必然结果，在系统的所有 CPU 上运行相同的内核也就是很正常的事了。这样做使得全局活动的协调更加容易。例如，当一个进程启动时，所有的内核必须协作以寻找最好的位置来执行它。此外，还需要一个全局的文件系统。

然而，每个内核都能对它所拥有的资源有很大控制权。例如，因为没有共享存储器，所以允许每个内核管理它拥有的存储器。又例如，如果使用交换或分页技术，则每个 CPU 上的内核就是决定如何进行交换和分页的合理地点。没有理由去集中这种权限。类似的，如果多个进程在某个 CPU 上运行，那么就在其上进行调度是合理的。

我们已经掌握了有关设计和实现分布式操作系统的相当多的知识。在这里我们先不对这些问题进行讨论，而是先要结束我们对硬、软件不同组合的概略了解，然后在 1.5 节中再回到这些问题上来讨论。

1.4.3 多处理机分时系统

我们要讨论的最后一个组合是运行在紧耦合硬件上的紧耦合软件。虽然在这个分类中存在着各种专用计算机（例如专用数据库计算机），但是最常见的通用型例子是像 UNIX 分时系统一样运行的多处理机系统，只是它有多个 CPU 而不是一个 CPU。对于外部世界而言，一个具有 32 个 CPU，每个 CPU 运算速度为 30 MIPS 的多处理机非常像一个速度为 960 MIPS 的 CPU（这就是上面所描述的单系统映像）。除非使用多处理机可以使系统实现更容易以外，整个设计都是可以集中的。

这类系统的关键特点是存在一个唯一的运行队列：一张系统内所有逻辑上开锁的并就绪的进程列表。运行队列是一个保存在共享存储器中的数据结构。举个例子，参照图 1-8 所示的系统，它有 3 个 CPU 和 5 个就绪的进程。5 个进程都放在共享存储器中，其中的 3 个正在运行：进程 A 在 CPU 1 上，进程 B 在 CPU 2 上，进程 C 在 CPU 3 上。另外两个进程 D 和 E 在存储器中等待运行。

现在假设 B 进程因等待 I/O 或分配给它的时间片到而阻塞。一种方法是 CPU 2 必须挂起 B 再调度另一个进程运行；这时 CPU 2 通常将开始执行操作系统代码（存储于共享存储器中）。当已经保存了 B 进程的所有寄存器后，它将进入一个临界区，运行调度程序调度其他进程，使其投入运行。调度程序作为一个临界区域运行是必要的。这样能够防止两个 CPU 选择同一进程运行。通过使用监听、信号量或其他用于单处理机系统的标准结构就可以获得必要的互斥。

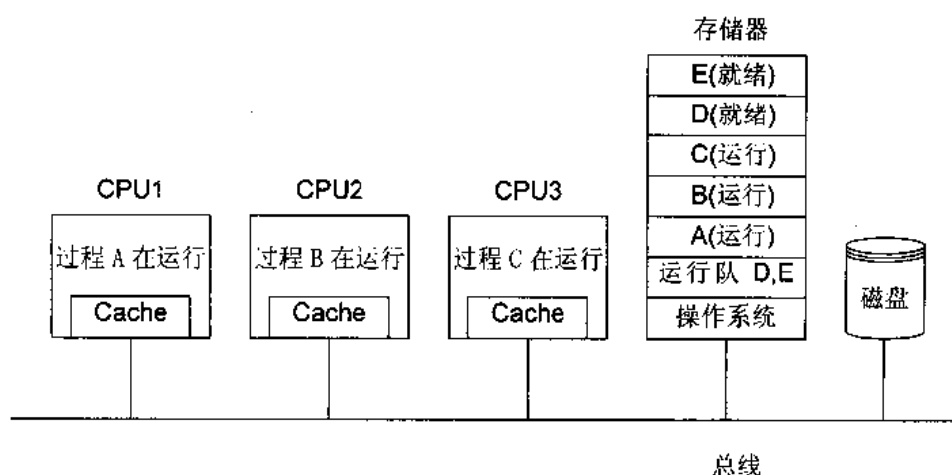


图 1-8 具有一个运行队列的多处理机系统

一旦 CPU 2 获得运行队列的独占访问，它就能移去第一个条目 D，退出临界区，然后开始执行 D。开始时执行操作会十分缓慢，因为 CPU 2 的缓冲存储器中充满了进程 B 在共享存储器中的部分字。但经过一段时间后，这些数据都将被清除，缓冲存储器中将充满 D 的代码和数据，因而执行的速度也将会加快。

因为每个 CPU 都没有私有的存储器，而且所有的程序都存储在全局共享存储器中，所以不论在哪一台 CPU 上执行进程都无关紧要。一般来说，如果一个运行时间较长的进程在完成之前被调度了多次，它在每个 CPU 上运行的时间大致相等。当进程需要在一个 CPU 上运行时，在 CPU 的选择上唯一有影响的因素就是要选择目前缓冲存储器中正保存着这个进程的部分数据的 CPU，因这可以稍微提高一下性能。换句话说，如果所有的 CPU 都是空闲的，正在等待 I/O，一个进程也已经准备好，那么，优先将进程安排在它刚使用过的 CPU 上是较好的，这需要假定从那时起没有别的进程使用过该 CPU (Vaswani 和 Zahorjan, 1991)。

一方面，在多媒体机上的一个进程如果阻塞以进行 I/O 操作，那么操作系统就可以选择挂起它或让它处于忙等待状态。如果 I/O 操作的完成时间小于进程切换时间，那么最好选择忙等待。一些系统让进程继续占有处理机几毫秒，目的是希望 I/O 操作将在这段时间之内完成，但是如果这段时间用完之后它还没有完成，系统就开始进行进程切换 (Karlin et al., 1991)。如果大多数临界区很短的话，这个方法可以避免许多昂贵的进程切换。

多媒体机系统与网络或分布式系统明显不同的一个领域是文件系统的结构。操作系统通常包含了具有一个统一的块缓冲存储器的传统文件系统。当任何一个进程执行系统调用时，在操作系统中产生一个陷阱来执行这个调用，当临界区被执行或存取中央表时，它用信号量、监听或其他一些等效的机制来实现锁住其他 CPU 防止它们干扰。通过这种方法，当执行 WRITE 系统调用时，中心块缓冲存储器被锁定，向其中写入新的数据，然后开锁。正如在一个单处理机系统上一样，任何后来的 READ 调用将看到新的数据。总之，这种文件系统很难与单处理机文件系统进行区分。事实上，在一些多媒体机系统中，一个 CPU 专用于运行操作系统；其他一些 CPU 则运行用户程序。然而，由于运行操作系统计

计算机经常是系统的瓶颈，所以这种情况是并不合适。这一点已由 Boykin 和 Langerman (1990) 作过详细讨论。

应该清楚的是，用多处理机实现虚拟单处理机外部特征的方法不能应用于没有共享存储器的多计算机。只有当所有的 CPU 都可以非常低的延迟访问缓存时，集中的运行队列和程序块缓存才有效。尽管这些数据结构能够在计算机网络上模拟，但是这种方法所造成的通信开销使得这种方法付出的代价异常昂贵。

表 1-4 列出了我们在上面已经讨论过的 3 种系统之间的一些不同之处。

表 1-4 n 个 CPU 的三种不同组织方式的比较

项 目	网络操作系统	分布式操作系统	多处理机操作系统
看起来是否像一个虚拟的单处理机系统?	否	是	是
所有的机器只运行相同的操作系统?	否	是	是
有多少操作系统的拷贝?	n	n	1
怎样通信?	共享文件	消息	共享存储器
需要共同一致的网络协议?	是	是	否
是否只有一个运行队列?	否	否	是
文件共享是否有良好的语义定义?	通常没有	是	是

1.5 设计中的问题

在前面的几节中,我们已经从软件和硬件的观点分析了分布式系统以及一些相关的主题。本节将简单介绍在建立分布式操作系统时所必须处理的几个关键的设计问题。本书后几章将进行详细介绍。

1.5.1 透明性(Transparency)

最重要的一个问题可能是如何实现单系统映像。换句话说，系统设计者如何使每个用户误以为一些机器的集合只是一个老式的分时系统？能达到这个目标的系统通常就被称为是透明的（transparent）。

透明度可分为两个级别：最容易做到的就是对用户隐藏系统的分布性。例如，当一个 UNIX 用户键入 make 命令来重新编辑一个目录中的大量文件时，他并不需要知道所有编辑是在不同计算机上并行处理的，也不需要知道它使用了大量的文件服务器。对用户而言，唯一不寻常的是系统的性能部分损失了。就终端的命令发布和结果显示而言，可以将分布式系统建成像一个单处理机系统一样。

在系统的低层中实现系统对程序也透明是可能的，但是比较困难。换句话说，系统调用接口的设计成使多处理机的存在对编程人员来说是不可见的。然而，要想欺骗编程人员要比欺骗终端用户难得多。

透明度的确切意义是什么呢？它是许多听起来合理而实际上比其表面要微妙得多的含糊概念之一。举一例子，我们可以设想由一些运行标准操作系统的工作站所构成的分布式系统。通常，系统服务（例如，读文件）是通过发出一个嵌入内核中的系统调用来获得的。

在这样的系统中，远程文件应该用同样的方法进行存取。所以一个通过与远程服务器建立明确的网络连接然后向它发消息的方法来访问远程文件的系统是不透明的，因为访问远程服务与访问本地服务是不同的。编程人员能知道其中使用了多台机器，但这是不允许的。

透明的概念适用于分布式系统的多个方面，如表 1-5 所示。位置透明（location transparency）是指在一个真正的分布式系统中，用户不知道硬、软件资源如 CPU、打印机、文件和数据库的位置。资源的名字不应隐含有资源的位置信息。所以像 *machine1: prog.c* 或 */macchine1/prog.c* 之类的名字是不能接受的。

表 1-5 在一个分布式系统中的不同的透明性

种 类	含 义
位置透明	用户不知道资源位于何处
迁移透明	资源可以不改名地随意移动
复制透明	用户不知道有多少个拷贝存在
并发透明	多个用户可以自动的共享资源
并行透明	系统活动可以在用户没有感觉的情况下并行发生

迁移透明（migration transparency）是指资源无须更名就可自由地从一地迁向另一地。在图 1-7 所示的例子中，我们可以看到服务器目录能够在客户目录分层结构中的任何地方装入。因为路径 */work/news* 并未指出服务器位置，所以系统是位置透明的。然而，现在假定正在运行服务器的用户想将阅读网络新闻放在“*games*”目录下，而不是在“*work*”目录下，相应地，将“*news*”从服务器 2 移到服务器 1。接下来，当用户 1 启动并用原来的习惯方法登录服务器，他将会发现 */work/news* 不存在了，取而代之的是另一个路径 */games/news*。因而一个文件或目录从一个服务器迁到另一个服务器，它的名字被强制性的更改了的事实说明远程登录系统不是迁移透明的。

如果分布式系统是复制透明(replication transparency)的，操作系统可以随意地为文件和其他资源进行附加拷贝而无须用户知道。很明显，在前一个例子中，自动拷贝备份是不可能的，因为文件名与位置是紧密联系的。为了了解复制透明是如何实现的，我们可以考虑一个逻辑上由 n 台服务器连成环形的集合。每台服务器都有一个完整的目录树结构，但它只存储全部文件的一个子集。为阅读一个文件，用户发送一个包含全路径名的消息给任一服务器。那个服务器检查它自己是否有这个文件。如果有，它就返回所要求的数据。如果没有，它就将这个要求传给该环形网中的下一个服务器，如此重复。在这个系统中，服务器能够自行决定备份任一服务器或所有服务器中的任意文件，而无须用户知道。由于这种方案允许系统对常用文件进行备份而无须用户知晓，所以该方案是复制透明的。

分布式系统通常有多个独立的用户。当两个或多个的用户试图在同一时间访问相同的资源时，系统应该怎么办呢？例如，如果两个用户试图同时更新相同文件将会发生什么情况呢？如果系统是并发透明(concurrency transparency)的，任一用户不会发现其他用户的存在。获得这种透明类型的机制是：一旦某个用户开始使用一个资源时，系统就自动锁定此资源，直到该用户使用完后再解锁。而在这种方式下，所有资源只能串行使用，而决不是并发使用的。

最后将介绍最难理解的并行透明（parallelism transparency）。从理论上说，一个分布

式系统在用户面前的表现就像一个传统的单处理机分时系统。那么如果一个编程人员知道他的分布式系统有 1000 个 CPU，而他想用其中的相当大一部分 CPU 去运行一个并行地推演多个棋局的下棋程序时，我们又应该怎么办呢？理论上的答案是：通过对编译程序、运行期系统和操作系统的综合，应该能够设计出如何利用这种潜在的并行优势而不为编程人员所察觉的方法。不幸的是，目前的技术发展水平与此相距甚远。如果编程人员真的想用多个 CPU 解决单个问题将必须明确地进行编程，至少是在可以预见的将来仍然会是如此。并行透明被分布式系统设计者视为神圣的追求目标。当并行透明工作完成的时候，整个工作也就完成了，从而也就是开始新领域工作的时候了。

尽管如此，有时用户并不想完全透明。例如，当一个用户要打印一个文档时，他通常喜欢输出到本地的打印机上，而不是 1000 公里以外的某个打印机上，即使远处的那台打印机更快、更方便、是彩色的、带香味的、目前处于空闲状态的。

1.5.2 灵活性

设计中的第二个关键问题是灵活性。由于我们刚开始学习如何建立分布式系统，所以提高对系统的灵活性的重视程度是很重要的。在设计过程中将会产生许多错误，也会引起大量的设计返工。现在看来合理的设计思想也许以后将被证明是错误的。避免问题的最好方法是在设计过程中始终保持有很大的选择余地。

灵活性和透明性，就像父母关系和苹果饼：谁又能反对它们的好处呢？很难想像谁会支持一个不灵活的系统。然而，事情并不是想像的那么简单。关于分布式系统结构的学术观点主要有两种，一种观点认为，每台机器都运行一个传统的内核，内核自身提供了大多数的服务；另一种观点认为是内核应尽可能少地提供服务，大量的操作系统服务可从用户级服务器上获得。这两种模型分别称为单内核（monolithic kernel）和微内核（microkernel），如图 1-9 所示。

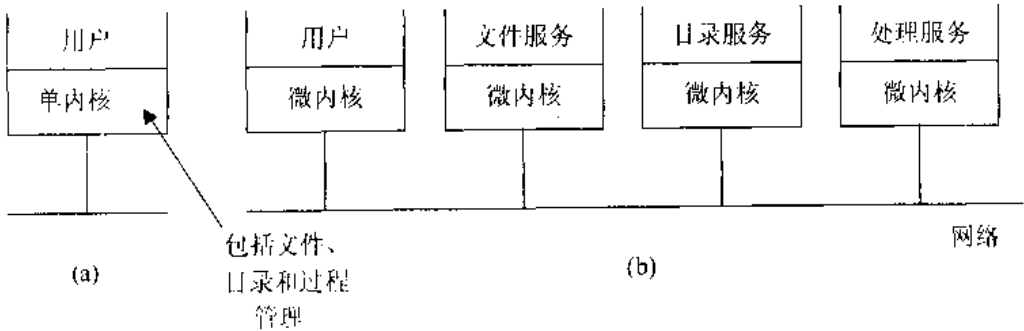


图 1-9 (a) 单内核
(b) 微内核

单内核基本上是目前的集中式操作系统。增加了网络功能和远程服务集合。大部分系统调用都通过陷阱到达内核，在内核中完成所需要的工作，然后内核再将所要求的结果返回给用户进程。使用这种方法，大部分机器都拥有自己的磁盘，也能够管理自己本地文件系统。许多扩展或模仿 UNIX 系统的分布式系统都采用这种方法，因为 UNIX 系统本身有一个巨大的内核。

如果单内核现在是占统治地位的，那么微内核就是很有前途的挑战者。大多数出于应付急需而设计的分布式系统采用了此种方法。微内核具有更好的灵活性，因为它几乎不做什么，只提供四种最小的服务：

- (1) 进程间通信机制；
- (2) 某些内存管理功能；
- (3) 少量的低层进程管理和调度；
- (4) 低层输入/输出服务。

具体说，微内核与单内核不同，它不提供对文件系统、目录系统、整个进程管理，或许多系统调用的处理。微内核中提供的服务包括那些因为在其他地方提供它们很困难，或因为代价昂贵的服务。主要目的是为了保持小。

许多其他的操作系统服务都作为用户级服务器来实现，如找一个名字、读文件或获得其他服务，用户都是发送消息给适当的服务器，由这个服务器完成相应的工作并返回结果。这种方法的优点是高度的模块化，对于每一个服务（服务器能理解的消息的集合）都有一个定义好的接口，每一个服务程序对所有客户来说都是可以访问的，且和位置无关。另外，该系统易于实现、安装和调试新的服务程序，因为增加或改变一个服务程序不需要像有一个单内核那样停止系统和启动一个新的内核。正是因为它所具有的添加、删除和修改服务的功能使得微内核具有很高的灵活性。此外，用户如不满意一个通用的服务程序，也可以编写自己的服务程序来替代它。

这种优势的一个最简单的例子是，一个分布式系统可以有多个文件服务器。一个支持 MS-DOS 文件服务，另一个支持 UNIX 文件服务。用户程序可以自己选择使用两者中的任何一个或两个都使用。而在单内核系统中，文件系统被嵌入内核，用户别无选择，只能使用它。

单内核系统唯一潜在的优点是性能。使用内核陷阱并且在其上执行工作要比给远程服务器发消息更快。然而，这两种分布式系统，一种用单内核（Sprite），另一种用微内核（Amoeba），详细比较的结果说明在实际中这种优势并不存在（Douglis et al., 1991）。由于其他因素占据了主要地位，而且因为发送消息和获得返回结果需要的时间很少（典型的情况下为大约 1 毫秒）而经常被忽略。因此，微内核系统将逐渐主宰分布式系统设计，单内核将最终消失或融入微内核中。也许 Silberschatz 和 Galvin 所著的关于操作系统的书（1994）的未来版本的封面将会以蜂鸟和雨燕为特色而代替剑龙和三角恐龙。

1.5.3 可靠性

建立分布式系统的最初目的之一就是使系统比单处理机系统更可靠。基本思想是，如果一台机器坏了，其他机器能够接替它进行工作。换句话说，理论上，系统可靠性是所有部件可靠性的“布尔或”。例如，有 4 台文件服务器，每台在任一时刻能使用的概率为 0.95，所有 4 台同时坏掉的可能性是 $(0.05)^4 = 0.000006$ ，所以至少有一台能用的可靠性是 0.999994，大大好于任意一个单独的服务器。

这只是理论上的结果，关键是在实践中应有效，现在的分布式系统依靠一些专用服务器的正常工作。因此，这些系统的可靠性更有可能是这些部件的“布尔与”而不是“布尔或”。在一个被大量引用的论断中，Leslie Lamport 曾经将分布式系统定义为“一种由于

我所不知道的某台机器的崩溃而使我不能进行任何工作的系统。”当然这种说法是不能完全信以为真的，因为系统中仍然有可改善的地方。

对可靠性的不同方面的区分是十分重要的。正如我们刚才所讨论的，可用性主要是指系统可用时间的比例。Lamport 的系统显然在这方面做得不好。可以通过不要求大量关键部件同时起作用的设计来提高可用性。提高可用性的另一种方法是冗余，即给关键的硬件和软件提供备份，以便其中之一损坏时可以用备用的部件来进行替换。

一个高度可靠的系统必须有高可用性，但仅仅如此并不够。存储在和进入系统的数据不能丢失或以任何方式混淆。如果文件冗余地存放在多个服务器上，那么所有备份都必须保持一致。一般来说，保存的备份越多，可用性越好，但不一致的可能性却越大，特别是在频繁进行修改的情况下更是如此。所有的分布式系统设计者在任何时候都必须兼顾这两点。

整体可靠性的另一方面是安全性。必须保护文件和其他资源不被非法用户使用。尽管在单处理机系统中存在相同的问题，但在分布式系统中这个问题更为严重。在单处理机系统中，用户通过口令进行登录，同时被确认。此后，系统知道该用户是谁，也能够检查他的每一个访问是否合法。在分布式系统中，当一条消息到达服务器向其发出请求时，服务器没有一个简单的方法来判断该指令是哪个用户发出的。在消息中没有名字或标识域可以被相信，因为发送者可能在撒谎。在这方面至少要进行相当多的考虑。

与可靠性有关的另一个方面是容错。假设一个服务器出了故障，然后很快再启动，那么会发生什么情况呢？服务器崩溃会造成用户系统的死机吗？如果服务器存有正在进行的活动的信息的表格，恢复它将是非常困难的。

总的来说，分布式系统能设计成可以屏蔽错误，也就是对用户隐藏错误。如果一个文件服务或其他服务实际上是由一组紧密协同工作的服务器构成，它应该能够采用这样一种结构，即用户不会意识到损失了一个或两个服务器，而只是感到一些性能上的降低。当然，技巧就在于如何安排这种合作，以便在系统正常运行时，不会给系统增加很大的开销。

1.5.4 性能

总是隐藏在系统内部的一个问题就是性能问题。如果你建立了一个透明、灵活和可靠的分布式系统，但是它的运行速度却像糖蜜的流动一样慢，那么，它是不会给你赢来任何奖赏的。特别是：如果在分布式系统中运行一个具体的应用，它的速度不应该比在一个单处理机中运行同一个应用慢。不幸的是，要做到这一点，说来容易，做起来却困难得多。

我们可以使用不同的性能标准进行衡量。其中之一是响应时间，还有吞吐量（每小时完成的工作量）、系统的利用率和网络容量消耗程度。此外，任何基准程序的测试结果通常取决于该基准程序自身的性质，一个涉及大量独立的高度受限于 CPU 计算的基准程序所给出的结果可能和以某种模式扫描一个大文件的基准程序所给出的结果有很大的不同。

性能问题与分布式系统（单处理机系统所没有的）中所必须的通信通常是很慢的这一事实是分不开的。通过 LAN 发送一个消息并得到应答大概要 1 毫秒，大部分时间用在两端不可避免的协议处理上，而不是用在线路中传送数据位。因此，为了改善性能，通常不得不尽量减少消息的数量。这种策略的困难在于获得性能的最佳方法是让许多活动在不同处理机中并行进行，而这样做就需要传送许多消息。（另一个解决方法是在一台机器上完成全部工作，但这在分布式系统中是很不合适的。）

一种可能的解决办法是密切注意所有计算的粒度大小 (grain size)。启动一个远程的小的计算, 例如两个整数相加, 几乎是不值得的, 因为通信所增加的额外开销抵消了所节省下来的 CPU 周期。而另一方面, 在远程启动一个长时间的受计算量限制的作业而带来的麻烦或许是值得的。通常, 涉及许多小计算的作业, 特别是相互作用程度非常高的作业, 会因分布式系统相对较慢的通信而产生麻烦。上述的这种作业被称为具有细粒度并行性 (fine-grained parallelism), 另一方面, 涉及大计算量、低相互作用和小数据量的作业, 粗粒度并行性 (coarse-grained parallelism) 可能会更合适些。

容错也要付出代价。通常的做法是通过使一些服务器密切合作来处理同一个请求以获得高可靠性。例如, 当一个请求进入一台服务器中时, 它就立即发送该消息的一个拷贝给它的一个同伴, 以便在完成工作前如果系统崩溃, 那个同伴能够接管工作。当然, 当它执行完请求后, 必须发送消息给那个同伴告诉它工作已经完成。因此, 我们至少需要发送两次额外消息, 一般情况下, 这将需要额外的时间和网络容量代价, 而且不会产生真正的效益。

1.5.5 可伸缩性

目前大多数分布式系统设计为有几百个 CPU 同时工作, 将来系统中的 CPU 的数量可能会大几个数量级。很不幸, 对于能使有 200 台机器的系统正常工作的方案对有 200000000 台机器的系统却不一定适用。考虑下面的例子, 法国 PTT (邮政、电话和电报管理机构, Post, Telephone and Telegraph asministration) 正在法国的每个家庭和商业机构中设置终端。这种称作小型电传 (minitel) 的终端允许用户在线访问包括法国所有的电话号码的数据库, 所以省去了打印和销售价格昂贵的电话簿的需要, 同时也极大地减少了对电话号码查询服务操作员的需要。估计该系统在几年内就能收回成本。如果这一系统在法国运行, 其他国家也会不可避免地采用类似的系统。

一旦所有的终端安装完, 将它们用于电子邮件 (特别是和打印机相连) 的可能性也就显而易见了。由于全世界的每个国家都在邮政服务上损失了很大的利润, 而电话服务却有巨大的利润, 所以用电子邮件来代替信件是有巨大诱惑力的。

接下来将讨论对各种数据库和服务的交互式访问, 简单列举几个例子: 从电子银行到飞机、火车、旅馆、剧院和饭馆的预订系统。不久以后, 我们将会有一个数千万用户的分布式系统。问题在于我们现有的方法能够扩大到适用于这样大的规模吗?

尽管对于这个巨大的分布式系统了解得不多, 但有一个指导原则是清楚的, 即避免集中的部件、表和算法 (如表 1-6 所示)。5 千万用户拥有同一台邮件服务器不是一个好主意。即使有足够高的 CPU 速度和足够大的存储容量, 进出邮件服务器的网络通信容量必然会成为一个问题。此外, 系统也不允许出现错误。一次停电能使整个系统瘫痪。最后, 大多数邮件是本地的, 如果在马赛的用户要发送信息给相距两个街区的另一个用户, 却要通过一个在巴黎的机器传送, 这种作法也是不可行的。

表 1-6 在超大型分布式系统中设计者应该尽力避免的潜在瓶颈问题

概 念	例 子
集中式组件	一个邮件服务器对应多个用户
集中式表	一个在线电话号码本
集中式算法	基于完全信息的路由

集中式表和集中式组件一样不好。一个人怎样才能保存 5 千万人的电话号码和地址呢？假定每条数据记录是 50 个字符。一个 2.5G 的磁盘就能够提供足够的存储空间。但在这里，同样单个数据库将无疑会轻易地让所有与它相连的通信线的容量达到饱和。而且它还很容易出故障（一小粒灰尘就能造成磁头失效和整个目录服务的崩溃）。另外，在这里，传送远程处理请求将浪费宝贵的网络容量。

最后，集中算法也不是一个好主意。在一个大型的分布式系统中，大量的消息必须在许多线路上进行路由选择。从理论角度上讲，最优的方法是收集全部所有机器和通信线路上负载的信息，再运行一个图论算法计算出所有的最优路线，然后再将这一消息立即传播到整个系统以便改进路由。

问题在于，由于上面所讨论过的原因，收集和传送所有的输入、输出信息也不是一个好主意。事实上，我们必须避免这种方法：从所有地点收集信息，把这些信息送到单个机器上去处理，然后发布结果的算法。只有分布式算法可以使用。这些算法与集中式算法比较，其特点如下：

- (1) 没有一台机器上存放着关于系统状态的全部信息；
- (2) 机器只是基于本地信息作出决定；
- (3) 一台机器出故障不会破坏算法；
- (4) 不一定存在全局时钟。

到目前为止，前面的三点与我们的讨论是一致的。而最后一点也许不那么显而易见，但它却很重要。任何以“在 12:00:00 点整所有机器都要记录它们输出队列的大小”开始的算法注定会失败，因为要让所有的时钟完全同步是不可能的。算法应考虑到缺少准确的时钟同步的情况。系统越庞大，同步的不确定性也就越大。在单 LAN 中，付出大量的努力可能会使所有的时间同步到几毫秒以内，但是要在全国范围的系统中做到同步却是不太可能的。我们将在第 3 章讨论分布式时钟同步。

1.6 小结

分布式系统由许多独立的 CPU 组成，它们在一起工作使得整个系统看上去像一台计算机。它们有许多潜在的优点，主要包括好的性能/价格比、同分布式应用的匹配能力强、潜在的高可靠性和负载增加时的可扩展性。但是，它们也有一些缺点。例如，软件较复杂、潜在的通信瓶颈和脆弱的安全性。然而，人们普遍地对建立和安装这种系统有极大的兴趣。

现代计算机系统通常有许多 CPU，它们可以被组织成多处理机（有共享内存）或多计算机（没有共享内存）。这两种类型都可以是基于总线或基于交换的。前者趋近于紧耦合，后者则趋近于松耦合。

多 CPU 系统的软件大致分为三类。网络操作系统允许用户在独立的工作站上通过一个共享的文件系统进行通信，否则就让每个用户只能操作自己的工作站。分布式操作系统将整个软、硬件的集合变成一个单一的集成系统，它很像传统的分时系统。共享存储器的多处理机也提供了一个单系统映像，但是因为它是通过把所有事情集中处理而建立的，所以实际上只是一个单系统。共享存储器的多处理机实际上并不是分布式系统。

分布式系统必须仔细设计，因为会有许多让人不注意的陷阱。关键问题是透明性——

一对用户，甚至应用程序隐藏所有的分布特性。另一个问题是灵活性。因为现在该领域的研究还只是处于初期，设计必须考虑到将来易于改变。在这一方面，微内核优于单内核。其他的重要问题还有可靠性、性能和可伸缩性。

习 题

- (1) 自从第一台商业大型机在二十世纪五十年代出现以来，计算机的性能/价格比大约提高了将近 11 个数量级。本书中阐述了若这种情况发生在汽车制造业会产生什么类似的结果。请给出另一例子说明这种巨大进步的意义。
- (2) 请列出分布式系统相对于集中式系统的两个优点和两个缺点。
- (3) 请说明多处理机与多计算机有什么区别？
- (4) 松耦合系统 (loosely-coupled system) 和紧耦合系统 (tightly-coupled system) 两个术语经常用于描述分布式系统，它们之间有什么区别？
- (5) 请说明 MIMD 计算机和 SIMD 计算机有什么区别？
- (6) 一个基于总线的多处理机使用监听缓存来实现相关一致的存储区。在这种计算机上使用信号量可行吗？
- (7) 交叉开关允许一次处理大量的存储请求，并且有良好的性能。但为什么实际中很少使用呢？
- (8) 一个具有 256 个中央处理机的以 16×16 网格组织的多计算机环境。一个消息最差的时延（以 hops 计）是多少？
- (9) 现在考虑一个有 256 个中央处理机的超立方体 (hypercube)。这时，一个消息最差的时延（以 hops 计）是多少？
- (10) 一个多处理机环境有 4096 个 50-MIPS 的中央处理机，它通过 omega 网络连接内存。为了使一个内存请求能在一个指令时间中到达内存并返回结果，转换时间需要多快？
- (11) 单系统映像的含义是什么？
- (12) 分布式操作系统与网络操作系统的主要区别是什么？
- (13) 微内核的主要任务是什么？
- (14) 请说出微内核优于单内核的两个优点？
- (15) 并行透明性是分布式系统想要实现的目标。集中式系统本身具有这种属性吗？
- (16) 请用自己的语言解释并行透明性这个概念。
- (17) 一个试验文件服务器 $3/4$ 的时间正常运行，并且 $1/4$ 的时间由于故障停机。这个文件服务器需要被复制多少次才能使之至少 99% 的时间可用？
- (18) 假设你有一个含有 m 个文件的大源程序需要编译。这个编译工作将在一个有 n 个处理机的系统上进行，这里 $n \gg m$ 。这样，这时最好的结果将是一个处理机运行时速度的 m 倍。哪些因素会使这种速度的提高小于这种最好的结果？

第2章 分布式系统的通信

分布式系统和单处理机系统一个最重要的区别是进程间的通信。在单处理机系统中进程间的通信无疑是利用共享存储器。一个典型的例子是生产者-消费者问题，一个进程向共享存储器写入，而另一个进程从该共享存储器中读出，甚至最基本的同步形式，如信号量(信号量自身可变)都要求必须有一个字的内存是共享的。在分布式系统中没有共享存储器，不管怎样，进程间相互通信的研究必须完全重新开始。本章，我们将讨论许多关于分布式操作系统中进程间通信的问题和例子。

我们将从进程间通信所必须遵循的规则开始讨论，这些规则称作协议。对于广域的分布式系统这些协议通常有许多层，每一层都有自己的目标和规则。将先讨论两个层次集，OSI 和 ATM，然后将比较详细地讨论客户-服务器模式。此后，就可以知道消息是如何进行交换的以及系统设计者的许多可用的选择。

一个特殊的选择就是远程过程调用，它非常重要，所以自成一段。远程过程调用是信息打包传送的较好方法，和常规的编程相似并且容易使用。然而，我们也应该看到它也有自己的特色和问题。

本章还介绍进程组如何通信，而不是两个进程间如何通信，将详细讨论一个进程组通信的实例——ISIS。

2.1 分层协议

由于缺少共享存储器，在分布式系统中所有通信都是基于消息传递的，当进程 *A* 想和进程 *B* 通信时，它首先在它的地址空间建立一条消息。然后执行一个系统调用让操作系统取出消息并且通过网络传送给 *B*。尽管这种基本思想听上去很简单，但是为了避免混乱，*A* 和 *B* 必须对要传送的所有位的意义达成一个协议。如果 *A* 发送一篇用法文写的才华横溢的新文章，用 IBM 的 EBCDIC 字符码编码，而 *B* 用英文写的超级市场的存货清单用 ASCII 码编码，那么这种通信的结果是不会令人满意的。

通信需要许多不同的协议。用多少电压表示“0”？多少电压表示“1”？接收者如何知道哪一位是消息的最后一位？怎样判定消息是否被损坏或丢失，如果发现了又应该做什么？数字、字符串和其他数据项需要有多长，它们如何表示？一句话，不论是低层的位传输还是高层的信息表示，在每一个层次上都需要协议。

为了方便地处理通信中存在的层与层之间的问题，国际标准化组织(ISO)开发了一个参考模型，它清楚地确定了不同层所涉及的内容，给出了各层的标准名称，并指出各层应做的工作。这个模型称作开放系统互连参考模型(OSI)，通常缩写为 ISO OSI，有时也称为 OSI 模型。在这里不打算给出详细介绍，只简单地介绍一下。如果需要更详细的信息，请参考(Tanenbaum, 1988)。

开始，OSI 模型被设计为用于开放系统的通信。一个开放系统就是指使用标准化规则

与其他开放系统通信的系统，而在系统中这些规则管理着发送和接收的消息的格式、内容和意义。这些规则的正式名称是协议。基本上，一个协议就是通信各方就如何进行通信所需要遵守的规则。当一位女士被介绍给一位男士，她可能会伸出她的手，而他是握它还是吻它，要取决于这位女士是在商业会议上的一位美国律师，还是在宫廷舞会上的欧洲皇族中的一员。违反协议将使通信变得非常困难，如果这时通信还是可能的话。

从更偏重技术的角度上来说，许多公司为 IBM PC 制造存储器板。当一个 CPU 想从存储器中读一个字，它将地址和某个控制信号发送到总线上。在一定的时间间隔内，认为存储器板应该能够看到这些信号，并将 CPU 所需要的字放在总线上作为应答。如果存储器板遵守请求总线协议，它就将准确无误地工作，否则它将不能准确地工作。

同样，如果要让网络上的一组计算机进行通信，那么它们就必须遵守网络协议。OSI 模型区分了两种类型的协议。所谓面向连接(connection-oriented)的协议，在交换数据之前，发送者和接收者首先必须明确地建立连接，并且协商可能使用的协议。工作完成后必须断开连接。电话就是一种面向连接的通信系统。在所谓非面向连接(connectionless)的协议中则不需要预先做什么，发送者只要将准备好的报文直接发送出去即可，投信到邮箱就是非面向连接协议的例子，对于计算机的使用，面向连接和非面向连接的通信都是很普通的。

在 OSI 模型中，通信结构分为 7 层，如图 2-1 所示。每一层处理有关通信的一个特定方面，使用这种方法，可以将一个问题分解为几个可管理的块，每一块都解决和其他块不同的问题，每一层都提供一个接口(interface)给它的上一层。接口由同时定义了这一层将提供给它的用户的服务的一系列操作组成。

在 OSI 模型中，当机器 1 上的进程 A 想和机器 2 上的进程 B 通信时，它建立一条消息，并将消息传送到本机的应用层。这一层可以是库过程，也可以用其他方法实现(如，在操作系统中，或在外部协处理器板上，等等)。应用层软件在消息前面加上报头并将结果通过第 6 层和第 7 层的接口传送给表示层，表示层加上自己的报头将结果向下传送给会话层，如此一直下去。一些层中不仅要在消息前面加报头，还要在消息后面加上报尾。当消息到达最低部时，由物理层实际传输报文，如图 2-2 所示。

当消息到达 2 号机器时，它向上传递，每一层都剥掉并检查自己的报头，最后消息到达接收者——进程 B，它以相反的路径响应，第 n 层的报头信息就用 n 层协议。

有一个例子可以说明为什么分层协议很重要。设想有两个公司之间需要通信，Zippy Airlines 和它的配套公司 Mushy Meals 有限公司。每个月 Zippy 公司管理旅客服务的经理让他的秘书与 Mushy 公司销售经理的秘书联系订购 100,000 箱鸡。习惯上，订货是通过邮局传递的，然而邮政服务太慢，两个秘书决定用 FAX 联系，他们不用通报他们的老板就可以这么做，因为他们的协议只处理命令的具体传送而不管它的内容是什么。

同样，旅客服务机构的经理能决定退掉鸡而要 Mushy 公司的新款羊排，而无须影响秘书。请注意这件事有两层：老板和秘书。每一层都有自己的协议，而各自的改变也可以独立于其他层。这种独立性使得分层协议颇具吸引力。随着技术的改进，每一层都可以改变而无须影响其他层。

在 OSI 模型中，不是两层而是七层，如图 2-2 所示。用于一个独立系统中的协议集合称作协议组(protocol suite)或协议栈(protocol stack)。下一节，将从低层开始逐层讨论。在适当的地方，我们将指出在每一层使用的一些协议。

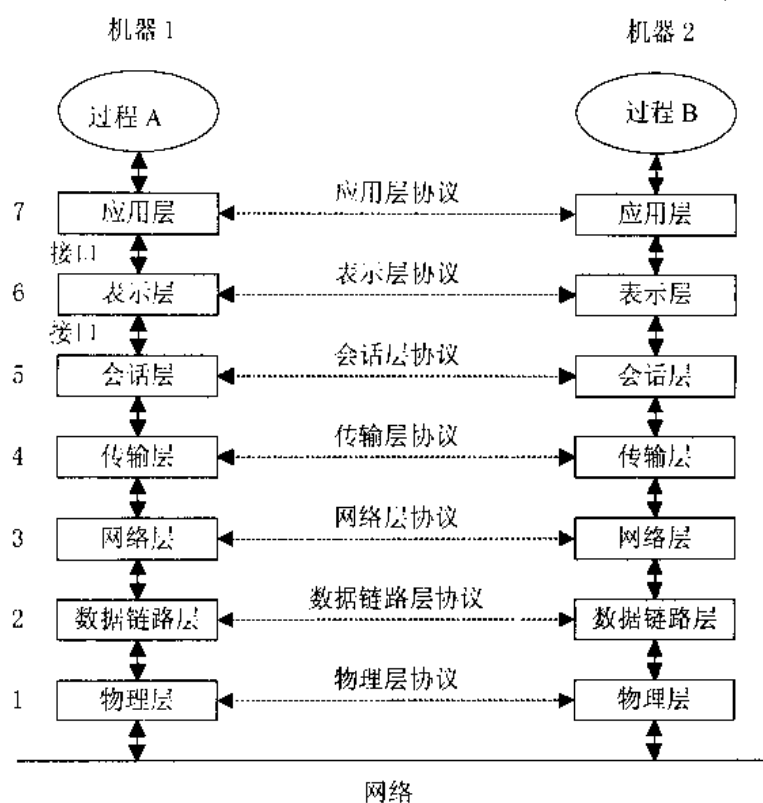


图 2-1 OSI 模型中的层、接口和协议

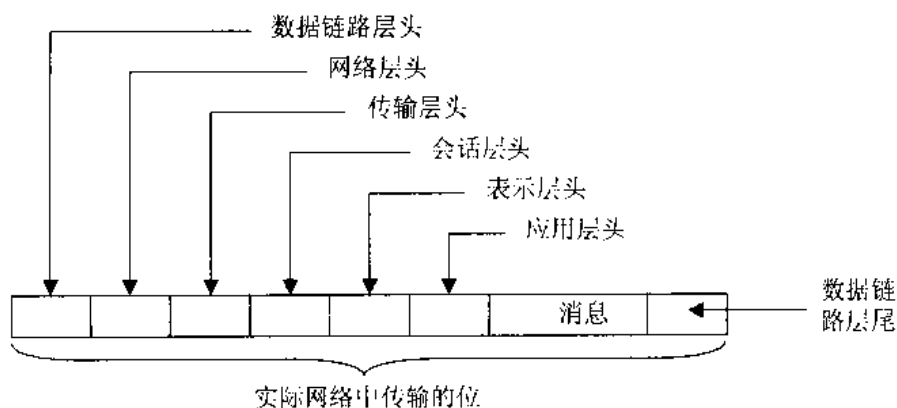


图 2-2 出现在网络上的一个典型消息结构

2.1.1 物理层

物理层是用于传输信号“0”和“1”的。用什么电压表示“0”和“1”、每秒传输多少位,以及是否能够进行同时的双向传输是物理层的关键问题。另外,网络接插件(插头)的大小、形状,有多少引脚,各引脚的意义,都在这里讨论。

物理层协议讨论电气、机械、信号接口标准化问题，以便一台机器发送 0，实际接收到的也是 0，而不是 1。现在已经有了很多物理层标准，如 RS-232-C 是串行口通信线的标准。

2.1.2 数据链路层

物理层只发送二进制位。只要没有错误产生，一切都好说。然而真正的网络通信是很容易出错的，因此需要一些校验和纠错的机制。这种机制就是数据链路层的主要任务，也就是怎样将一些位组织到单元(有时称作帧)中，并且检查每帧是否被正确地接收。

数据链路层所做的工作是在每帧的开头或结尾处添加特殊的位来标记头和尾，并且以一种确定的方式通过把帧中所有的字节相加来计算其校验和。数据链路层将校验和添加到帧中。当该帧到达时，接收者重新计算校验和并与帧后的校验和相比较。如果一致，则认为该帧是正确的并接受它，否则接收者将请求重发。帧被赋予序列号加以标识(在报头中)，因此很容易进行相互区分。

在图 2-3 中，我们看到一个例子：A 想发送两个信息“0”和“1”给 B。在 0 时刻发送数据消息 0，但是当它在 1 时刻到达时，传输线上的干扰使它遭到破坏。于是校验位是错误的。B 注意到了这一点，在 2 时刻用一个控制消息请求重发。不幸的是，与此同时，A 正在发送数据消息 1。当 A 接收到重发的请求时，它重发 0，然而当 B 接收到消息 1 而不是所请求重发的消息 0，它发送控制消息 1 给 A，解释它想要 0 而不是 1，当 A 看到这个 1，它耸了耸肩，第三次发送了消息 0。

这个例子并不是要看图 2-3 中的协议是否是个不错的协议(或不是)，而是要说明在每一层上都需要发送者和接收者进行对话，典型的消息是“请重发消息 n ”，“我已经重发了”“不，你没有”，“我有的”，“好的，就像你说的，再发送一次”，等等。这种对话发生在报头域中。在报头域中，定义了各种请求和应答，并且提供了各种参数(如帧序号)。

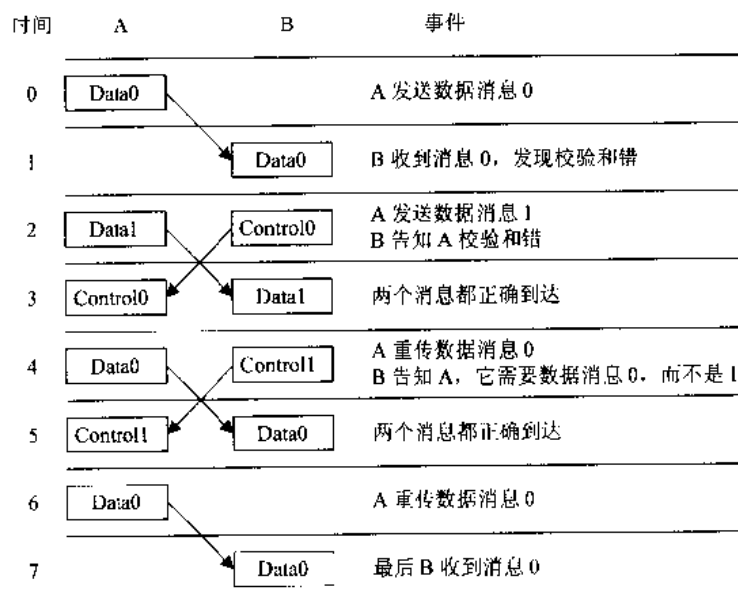


图 2-3 数据链路层上的发送者和接收者之间的对话

2.1.3 网络层

在局域网中,发送者通常并不需要知道接收者在哪里,它只是将消息发送到网络上,然后接收者取走消息。然而,一个广域网是由大量的机器组成的,每台计算机都有一些线和其他机器相连,就像一张展示大城市和与它相连的道路的交通图。为了使一个从发送者获得的消息到达接收者需要建立一些转发,在每一个转发处都要选择一条输出线将消息发出。怎样选择最佳路径的问题就叫路由,它是网络层的主要任务。

问题是非常复杂的。因为,实际上最短的路由并不总是最好的。真正的问题在于给定路由上的延迟量,它又与传输容量以及在不同线路上排队等待发送的消息数目有关。所以延迟能改变传输过程的时间。尽管一些路由算法试图适应不断变化的负载,但从长远来看,其他路由算法作出的决定是令人满意的。

目前广泛使用以下两个网络层协议:面向连接的和面向非连接的。面向连接的协议称作 X.25,它受到公共网操作者的青睐,如电话公司和欧洲的 PTT。使用 X.25 的用户首先发送一个呼叫请求到目的地,目的地可以接收或拒绝接收所提出的连接。如果接收连接,在随后的请求中将给呼叫者一个连接确认。在许多情况下,在建立连接时,网络从发送者到接收者之间选择一条路由,并在以后的传输中使用。

面向非连接的协议称作 IP(Internet 协议),是 DoD(美国国防部)协议包的一部分。一个 IP 信包(网络层中有关消息的一个技术名词)不需要建立任何连接就可以发送。每一个 IP 信包自己寻径到达目的地而独立于其他 IP 信包。它并不像 X.25 那样选择和记忆内部路径。

2.1.4 传输层

信包在由发送者到接收者的路途中有可能丢失。尽管一些应用自己能够进行错误恢复,但一般应用更愿意有可靠的连接。传输层的工作就是提供这种服务。其思想是会话层能将消息传递给传输层,并希望在传输中不会丢失消息。

从对话层接收到消息后,传输层将其分成刚好是一个包大小的片,给每个包分配一个序列号,并且全部发送出去。在传输层的头域知道哪个包已被发送,哪个包已接收,接收者还有多少空间可使用等类似问题。

可靠的传输连接(在面向连接中定义的)可以建立在 X.25 或 IP 上。前一种情况,所有的包将按一个正确的顺序到达(如果它们全部到达),但后一种情况由于采用了不同的路由选择,有可能后发送的包比先它发送的包早到,由传输层软件将所有包重组,目的是维持一个假像,即传输连接像一个大管道——你放进消息,它们被无损传送,并保持进入时的次序。

正式的 ISO 传输有五个变种,即 TP0~TP4。差别在于错误处理和在一条 X.25 连接上建立几个传输的传输能力。选择哪一种取决于低层网络层的特性。

DoD 传输协议称作 TCP(传输控制协议),其细节在(Comer 1991)中描述。它与 TP4 相似,TCP/IP 的结合广泛应用于大学和大多数 Unix 系统中。DoD 协议集也支持非面向连接传输协议,如 UDP(用户数据报协议),这个协议基本上就是具有少量附加条款的 IP。不需要面向连接协议的用户程序常使用 UDP。

2.1.5 会话层

会话层从本质上讲是传输层的增强版本，它提供对话控制、对正在会话的对话者进行跟踪以及同步功能。同步若允许用户在很长的传输中插入检查点是很有用的。会话在被打断时，仅仅需要从最后一个检查点重来，而不是从头开始重发。实际上，很少有应用对会话层感兴趣，它几乎不被支持，甚至在 DoD 协议集中也不存在。

2.1.6 表示层

低层关心的是把“位”可靠有效地由发送者传送到接收者。和底层不同的是，表示层关心的是位的含义。大多数消息不是由随机位串组成的，而更多的是由结构化消息，如人名、地址和钱数等组成。在表示层可以定义一些特殊的记录，它包含着与此相似的域，然后发送者提醒接收者消息以某种格式包含着一个特殊记录。这使得内部表示方法不同的机器间的相互通信变得很容易。

2.1.7 应用层

应用层实际上只是公共事务遵循的一个杂项协议的集合，如电子邮件、文件传输及在网络上连接远程终端等。其中最为人所熟知的是 X.400 电子邮件协议和 X.500 目录服务器。这一层和在它紧下面的两层都不是本书的重点。

2.2 异步传输模式网(ATM 网)

前面讲的 OSI 于 70 年代开发 80 年代实现(在某种程度上)。在 90 年代新的发展取代了 OSI，特别是在某些低层驱动技术方面。本节，我们将简单了解网络上的这些新发展。因为将来的分布式系统很可能建立在它们上面，认识它们对操作系统设计者是很重要的。你如果要察看有关当前网络技术的最新状态，请参考(Kleinrock, 1992; Partridge, 1993, 1994)。

在最近 25 年间，计算机在性能方面已经提高了许多数量级，而网络性能却没有提高。当 ARPANET(Internet 的前身)于 1969 年创始时，其结点间使用 56KB 的通信线路，这在当时是代表了最新水平的通信。70 年代末 80 年代初，这些线路中的大多数由 1.5Mbps 的 T1 线路所取代。最终，主干网发展成为 45Mbps 的 T3 网络，但 Internet 上的大多数传输线仍是 T1 或更慢的线路。

新的发展是通信速率突然以大约 155Mbps 为最低标准，主干网以 1 GMbps 以上的速度运行(Catlett, 1992; Cheung, 1992; 和 Lyles、Swinehart,1992)。这个变化将给分布式系统带来巨大的冲击，使以前你想象不到的应用成为可能，但这也带来了新的挑战。下面将介绍此技术。

2.2.1 什么是异步传输模式

80 年代末以来，世界上的电话公司终于开始意识到电信业务有比在通信通道上发送速率为 4KHz 的模拟语音更多的作为。数字网如 X.25 存在已经一年了，它们虽然像蹒跚

学步的孩子却常常以 56Kbps 或 64Kbps 的速度运行。像 Internet 一样的系统被认为是学术上的奇特事物。这很类似于循环演出杂耍的双头牛，而模拟语音则是正剧。

当电话公司决定为 21 世纪建造网络时，他们进退两难，声音传输是平滑的，需要低但连续的带宽，数字传输是突发性的，常常不需要带宽(当不传输时)，但有时在很短的时间内又要高的带宽，无论是传统的线路交换(在公用交换电话网中)还是包分组交换都不能同时适应这两种传输。

经过长时间的研究后，形成一个混合方案，用固定大小的块通过虚拟线路传输。这个折衷方案对两种类型的传输都有很好的性能。这个方案——ATM(Asynchronous Transfer Mode,异步传输模式)，已成为国际标准，并会在将来的分布式系统中起重要作用——无论是局域网还是广域网。如果你需要得到有关 ATM 的更多的教程，请参考(Le Boudec, 1992; Minzer, 1989 以及 Newman, 1994)。

ATM 模型是发送者首先建立一个连接(也就是一条虚拟线路)到接收者。在建立过程中，从发送者到接收者之间建立一个路由，并将路由信息存放在沿途的交换机上。系统可以利用这个连接来发送包，包先被硬件拆成固定大小的单位，称作信元(cells)。一个已给定虚拟线路的信元，沿着交换机中的路径流动。当不再需要连接时，释放此连接，并从交换机中删除该路由信息。

这种方式与分组交换和线路交换相比有一些优点。最重要的是网络可以随意而高效地混合传送大量的语音、数据、广播、电视、视频录像、无线电和其他信息，而不需要分别用不同的网传送不同的信号(如电话、X.25、有线电视等等)。新的服务如商务可视会议也将使用这种方式。在所有情况下，网络看到的都是信元，而不关心它们内部是什么。这种结合意味着节省大量的开支和简化工作，每个家庭只要接入一条信号线，就可满足所有信息和通信的需要，还可能带来新的应用，如视频点播、视频会议和远程数据库访问。

信元交换很适用于多波传输(一个信元到多个目的地)，一种类似于同时将电视广播信息传送到千家万户的技术。常规的线路交换(用于电话网中)不能这样处理。虽然广播媒介，如有线电视可以做到这点，但是它们不能毫不浪费带宽地处理点-点传输(高效广播每一条消息)。信元交换的优点是既能有效地处理点-点传输，又能处理多波传输(广播)。

固定大小的信元允许快速交换，而这对于采用存储-转发机制的包交换来说是非常困难的，它同时也消除了小包由于被大包阻塞了所需的线路而被延迟的危险。在信元交换中，一个接一个的发送，下一个信元甚至是属于不同的包。

ATM 有自己的协议分层，如图 2-4 所示。物理层和 OSI 模型第 1 层功能相同。ATM 层处理包括路由选择在内的信元和信元传输。它包括 OSI 第 2 层和第 3 层的一部分，但和 OSI 的第 2 层不同，ATM 不修复丢失或损坏的信元。适配层将包拆分成信元并在另一头组装，这些在 OSI 模型中直到第 4 层才明确出现。适配层没有提供可靠的端对端服务，所以传输连接必须在上一层(Upper layer)实现，例如通过使用 ATM 信元来进行 TCP/IP 传输。

下面将从最低层开始依次讨论图 2-4 中的低三层，从而展开我们的工作。

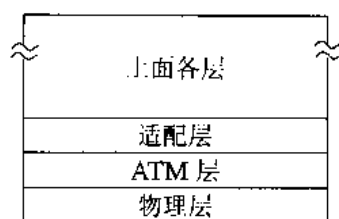


图 2-4 ATM 参考模型

2.2.2 ATM 物理层

ATM 适配板插在计算机中，将信元流发送到线路上或光纤上。传输流必须是连续的。当没有数据可以进行传送时就发送空信元。这意味着在物理层上，ATM 其实是同步的而不是异步的，但在虚拟线路上则是异步的。

作为替换，适配板在物理层上可采用 SONET(Synchronous Optical NETwork,同步光纤网)，将它的信元放入 SONET 帧的负载域上，这种方法的优点是它和 AT&T 的国际传输系统以及其他使用 SONET 的载体相兼容。在欧洲，一个叫 SDH(Synchronous Digital Hierarchy, 同步数字体系)的类似系统已经在一些国家使用。

在 SONET 中，一个基本单位(类似于 T1 帧的 193 位)是一个 9×90 的称作帧的字节数组。在其 810 字节中，36 个字节是头字节，774 字节用来作为负载。每 $125 \mu s$ 传送一帧，和电话系统标准速率 8000 信号/秒相匹配，因此整个数据传送速率(包括开销)是 51.840Mbps，而且网上传输速率(包括开销)是 49.536Mbps。

这些参数是美国、欧洲、日本和其他电话公司为了处理 U.S.T3 数据流(44.736Mbps)以及考虑其他国家已使用的标准，经过 5 年不断的磋商才选择的。在这里计算机厂家并没有起到重要的作用(一个 9×90 的数组有 36 字节头数据是计算机科学家不喜欢看到的)。

基本的 51.840Mbps 通道称作 OC-1。系统可能将发出一组 n 个 OC-1 帧来作为一个组。当它使用 n 个独立的 OC-1 通道和 OC- nc (供连接用)组成一个高速通道时，可用 OC- n 来标识。OC-3, OC-12, OC-48 和 OC-192 所用的标准已经建立，这些标准当中，ATM 最重要的标准是传输速率为 155.520Mbps 的 OC-3C 和传输速率为 622.080Mbps 的 OC-12C。因为在不久的将来，计算机可能以这种速率传输数据，对于电话系统中的长距离传输，目前 OC-12 和 OC-48 应用最广泛。

用于计算机的 OC-3C SONET 适配器允许计算机直接输出 SONET 帧，不久可望看到同样的 OC-12C。由于电话系统甚至连 OC-1 都无法负载，所以许多音频电话不可能直接发送 ATM 或 SONET 帧(ISDN 将代替它)。但对于可视电话，ATM 和 SONET 是理想的。

2.2.3 ATM 层

当 ATM 开始研制时，标准化组织中有两个派别，欧洲人想用 32 个字节的信元，因为它们延迟小并且在大多数欧洲国家不需要回波抑制器；而美国人已经有了回波抑制器，想用 64 个字节的信元从而达到最有效的数据传输。

最后的结果是一个信元 48 个字节，谁也不像。它对于声音太大对于数据太小，更糟的是，加上 5 个字节的头数据，于是一个信元由 53 个字节组成，其中包括 48 个数据域。

注意，53 个字节一信元并不能很好地和 774 字节的 SONET 相匹配，于是 ATM 信元将横跨 SONET 帧。因此，需要两个不同级别的同步信号，一个用于 SONET 帧的起始，一个用于检测在 SONET 的负载中 ATM 所有信元的第一个信元起始。幸运的是，出现了一个组装 ATM 信元到 SONET 帧的标准，而且整个层可以用硬件来实现。

从一台计算机到第一个 ATM 交换机的信元头的设计如图 2-5 所示。不幸的是，在两个 ATM 交换机之间信元头的组成是不一样的，GFC 域将被大于四位的 VPI 域取代。这不太好，因为它带来了计算机到交换机、交换机到交换机以及因此适配硬件之间的不必要的差异。两种有 48 字节信元的负载都直接跟在头数据后。

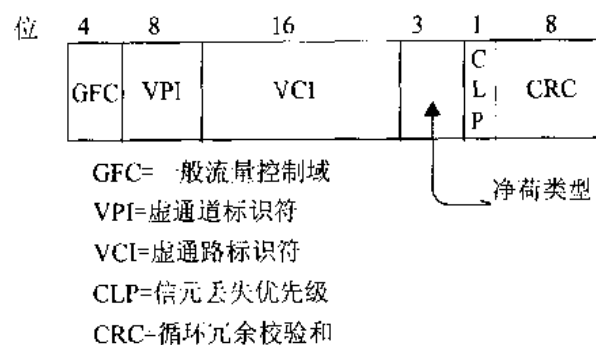


图 2-5 用户-网络信元头设计

一旦对 ATM 如何进行流量控制达成了一致标准，GFC 就会被派上用场。VPI(Virtual Path Identifier)域和 VCI(Virtual Channel Identifier)域一起指出信元虚通路的路径，沿途的路由表使用此信息选择路由。这些域每经过路径的一个节点就要改变它。VPI 域用于将目的地相同的虚拟线路集成一组，并通过载波重新选择路由，而不用检测 VCI 域。

净荷类型域将数据信元和控制信元区分开来，并且更进一步区分不同类型的控制信元。CLP 域用来标记一些不太重要的信元。一旦发生阻塞，将抛弃最不重要的信元。最后，在头数据前是一个字节的检验和 (而不是数据)。

2.2.4 ATM 适配层

在 155Mbps 速率下，每 3 μs 有一个信元到达。现在很少有(如果有的话)CPU 能够处理超过每秒 300000 次的中断。因此需要一个机制以使计算机只发送包，而由 ATM 硬件将包拆分成信元并且传送信元。信元在接收方被重新组装。每个包而不是每个信元产生一次中断。这种拆分和组装信元是适配层的工作。希望大多数的宿主适配器板能在板上运行适配层，并且每进入一个包产生一次中断，而不是每进入一个信元就产生一次中断。

但不幸也正在于此，该标准的作者在设计上犯了一些错误。原始的适配层定义了如下四类传输：

- (1) 恒定位速率的传输(用于音频和视频)；
- (2) 可变位速率的的传输，但具有受限延迟；
- (3) 面向连接的数据传输；

(4) 面向非连接的数据传输。

很快发现 3 类和 4 类在效果上基本是一样的，将它们组合成一个新类 3/4。因此计算机厂家很快从夹缝中苏醒过来，并且注意到没有一个适配层是适合数据传输的，于是他们提出了 AAL5(ATM 适配层 5)用于计算机到计算机的传输(Suzuki, 1994)。它的昵称是 SEAL(Simple and Efficient Adaptation Layer, 简单有效的适配层)，暗示了它的设计者考虑到了其他三个 AAL 层。(公平地说，使两个传统有巨大差异的产业——电信和计算机——完全认同同一个标准是一个非凡的成就。)

让我们看一看 SEAL，由于它的简单性，在 ATM 头中只用了一位，它位于净荷类型域中。这一位常常是 0，但在包的最后一个信元中置 1。最后一个信元在最后 8 个字节中包含一个数据尾。在大多数情况下，一个包的结束和一个数据层的开始将有一些填充(用 0)。采用 SEAL，目的地仅仅组装每个线路中输入的信元，直到它发现一个包结束位置位就取出并处理数据尾。

数据尾有四个域，前两个域每个域有 1 个字节长暂时未用，然后是 2 个字节的域给出包的长度，最后是 4 个字节的包检验和、填充字段和包尾。

2.2.5 ATM 交换

ATM 网络由铜缆或光缆和中继器构成，图 2-6(a)说明了 4 个交换机的网络。在 8 台计算机中的任何一台产生的信元都可以连入系统，并且能够通过一个或更多的交换机连到其他的任何一台计算机上去。每个交换机有四个端口，每个都可以用于输入和输出。

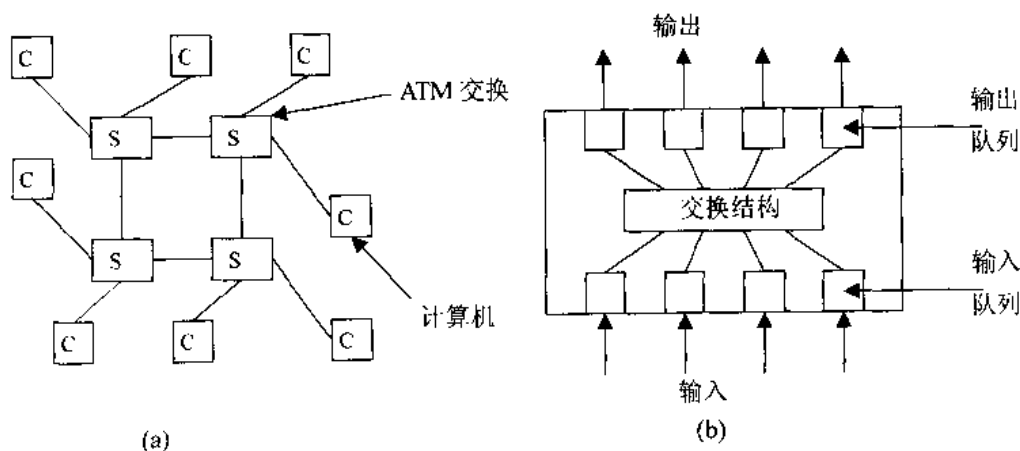


图 2-6 (a)ATM 交换网络
(b)交换机内部构造

一个交换机的基本结构如图 2-6(b)所示。它有输入线、输出线和一个并行的交换结构 (switching fabric) 将它们连接在一起。因为一个信元必须在 $3\mu s$ (在 OC-3) 内完成交换，而且同一时间可以有和输入线数同样多的信元到达，所以并行交换是非常重要的。

当一个信元到达以后，交换机检测其 VPI 和 VCI 域，再根据虚拟线路建立时存放在交换机内部的信息，将信元路由至正确的输出端口。尽管标准允许丢弃信元，但它要求传

送进来的这些信元必须按顺序传送出去。

当两个信元从不同的输入线同时到达并希望从同一个输出口输出时,就产生了问题。当然标准允许将其中的一个丢弃,但如果你的交换机在 10^{12} 个中抛弃多于一个信元,你就不可能卖出多少交换机。一个可选择的方案是随机选取一个并转发它,而一直保留另一个信元。在下一轮中,这个算法又一次被使用。如果两个端口每一个都有对于同一目的地的信元流,就建立了大量的输入队列,阻塞其后欲想输出到空闲输出端口的信元,这个问题叫作线路前端阻塞(head-of-line blocking)。

一个不同的交换机设计是将信元拷贝到和输出缓冲区相联系的队列中,而不是保留在输入缓冲区中。这种方法消除了线路前端阻塞,且性能也较好。交换机也可以有一个缓冲池既用作输入缓冲又用作输出缓冲。还有另一个可能是在输入端有缓冲区,即使不能交换第一个信元,但也允许在线上交换第二个或第三个信元。

许多其他的交换机已进入规划和试制阶段。这其中包括使用共享存储器、总线或环的时分交换和空分交换机,它们在每个输入线和每个输出线之间具有一条或多条的路径。这些交换机中的某些已在(Ahmadi 和 Denzel,1989; Anderson 等,1993; Gopal 等,1992; Pattavina,1993; Rooholamini 等,1994; 以及 Zegura, 1993)中进行了讨论。

2.2.6 ATM 对分布式系统的影响

以 155Mbps, 622Mbps 和可能以 2.5Gbps 的速率运行的 ATM 网络,对分布式系统的设计有重大意义。很大一部分作用在于大量的高频带宽忽然变得可用,而不是因为 ATM 网络的一些特殊特性。这对广域分布式系统来说影响是最为明显的。

现在,设想发送一个 1M 位的文件穿过美国并等待它正确到达的确认,光在光纤中的速度大约是在真空中的 $2/3$, 所以 1 位数据穿过美国大约需要 15ms。在 64Kbps 的线路上传送该文件大约要 15.6s, 于是另外的 30ms 往返延迟无关紧要。在 622Mbps 的线路上,将整个文件传送出去需要 $(1/622)s$ 或大约 1.6ms。在最好的情况下,应答在 31.6ms 后返回,在这期间线路大约空闲 30ms 或 95%的时间。当速度上升后,异步应答的时间接近 30ms,虚拟线路的可用带宽利用率接近于 0。如果消息短于 1Mb(在分布式系统中很常见的),那就更糟了。结论是:对于高速的广域分布式系统,在许多应用中的延时,特别是相互作用的应用中的延时需要新的协议和系统结构处理。

另外一个问题是流量控制,设想我们有一个相当大的 10GB 视频信号文件,发送者以 622Mbps 传送,数据开始涌向接收者,接收者碰巧没有 10GB 缓冲容量,于是它发回一个信元说“停止”。当 30ms 后“停止”信元回到发送者时,大约 20Mb 的数据在线路上,如果由于没有足够的缓冲区而将大多数数据丢失,它们又要被重新发送。使用传统的滑动窗口协议让我们又回到刚才那个位置,如果发送者只允许发送 1Mb 再等待应答,虚拟线路 95%的时间是空闲的。一个选择是在交换机和适配器上设置大量的缓冲区,但这又会导致价格的上升。另一个可能的选择是进行速度控制,接收者和发送者达成协议决定发送者能以每秒多少位传输。关于在 ATM 中的流量控制和阻塞控制,分别在(Eckberg, 1992; Hong 和 Suda, 1991; 以及 Trajkovic 和 Golestani, 1992)中讨论。在(Nikolaidis 和 Onvural, 1992)中给出有关 ATM 网络性能的超过 250 种参考资料的文献。

处理近 30ms 滞后的另一种不同方法是发送一些位,然后停止发送进程,在等待回答

时,运行其他进程,此策略的问题是使那些计算机及许多应用变得这么廉价。每个进程拥有一台计算机,要么什么都不做,要么执行它。浪费 CPU 的时间并没关系,因为它很便宜,但是,即使在通信受限的应用中,速度从 64Kbps 上升到 622Mbps,它的性能也没有带来 10000 倍的改进。

洲际通信延迟的作用以各种形式表现出来,例如,如果在纽约,一些应用程序用 20 条请求命令请求在加利福尼亚的服务器给出应答,对用户来说 600ms 的延迟就要引起注意。人们发现 200ms 延迟都已令人烦恼。

另外,我们将运算本身送到加利福尼亚的计算机上,让每个用户敲键作为一个独立消息,通过全国网发送,并返回显示。对于每次敲键需要加上不被人注意的 60ms,然而这种原因很快会让我们抛弃分布式系统的主意,而将所有远程用户的计算机放在一起。事实上,我们建立了一个大的中心分时系统,而仅仅让用户分散。

另一个发现和 ATM 的特性有关系,就是交换机在阻塞时允许丢弃信元,丢弃一个信元意味着超过等待时间时它将重发整个包,所以像演奏音乐一样,服务器需要一个不变的速率,这可能是个问题(奇怪的是,耳朵要比眼睛对不规则传送要灵敏得多)。

这些问题和其他问题的结论是,一般的高速网络和特殊的 ATM 网络带来了新的机遇,但是利用它们的优点将并不简单。在知道如何有效处理他们之前,研究工作是很有必要的。

2.3 客户-服务器模式

在将来 ATM 网络会很重,但目前对于大多数用户来说太昂贵了。让我们回头考虑普通的网络,乍一看,沿着 OSI 建立分层协议好像是组成分布式系统的最好办法,发送者和接收者建立一个连接(一个位管道),将位送入;这些位能够无错地按顺序到达接收者,这样还有什么问题呢?

事实上还有许多问题,让我们看一下图 2-2,所有这些报文头组成了一个可以想像的数量很大的头,每次要传送一个报文都要经过好几层,每一次都建立并从上到下附加报文头,再从下到上剥掉它,这些工作需要时间。在广域网中,每秒能够发送的二进制位数(bps)是相当少的(通常小于 64Kbps)。对这些报文头的处理不会影响网络性能。限制因素是线路的容量,而且即使报头再增加,CPU 仍然可以保证线路全速运行,因此一个广域分布式系统可以使用 OSI 或 TCP/IP 协议而不损失网络性能,对于 ATM,将有一些严重问题。

然而对于基于局域网的分布式系统,常常需要加大量的协议头,于是许多 CPU 时间是浪费在运行协议上,而有效的传送只占 LAN 传输能力的小部分,因此大多数基于局域网的分布式系统,并不完全使用分层协议,而仅是使用它的一个子集。

另外,OSI 模型只解决了问题的一个很小的方面——把二进制位从发送方传送到接收方(以及在上层协议中如何解释这些数据),而对怎样建立分布式系统却没有说明,解决该问题需要更多的细节。

2.3.1 客户机和服务器

前一节介绍了客户-服务器模型,其思想是构造一种操作系统,它由一组协同进程组成,这组进程称作服务器(server)。为用户提供服务的进程称作客户(client),客户和服务

都运行在相同的微内核中。正像早先看到的一样，让客户和服务端都以用户进程方式运行，一台机器可以运行于单个进程、多个客户、多个服务器或者两者的混合。

为了避免面向连接的协议加报文头，如 OSI 和 TCP/IP 协议，用户服务器模式常常以简单的面向非连接的请求/应答协议为基础。客户向服务器发出一个请求消息请求一些服务(如读文件块)，服务器完成后返回所要的数据或者给出一个错误码，指出工作未完成，如图 2-7(a)所示。

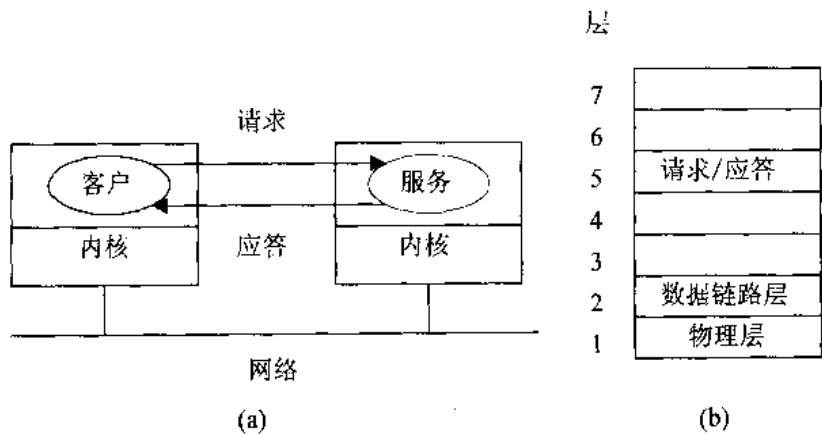


图 2-7 客户机-服务器模式(虽然所有传送的消息实际上都是由内核完成的，但在不发生混淆的情况下使用这张图表。)

图 2-7(a)的主要优点是简单。客户发出一个请求，得到一个应答。在使用之前无须建立连接也不用释放连接。应答消息就是对请求的确认。

从简单性可以得出另一优点:有效性。协议栈比较短因而也就更有效。假定所有的机器都相同，则只需要三层协议，物理层和数据链路层协议处理从客户机发送数据到服务器和接收返回的数据。这些是由硬件处理的，例如以太网和令牌环网。因为无须建立连接，无须路由选择，因此 3、4 层都不需要。5 层是请求/应答协议，它定义了合法的请求集和对这些请求合法的应答集。因为没有会话层，所以没有会话层管理。上层也不需要管理。

根据这种简单的结构，通信服务可以简化为由微内核提供的两个系统调用，一个是负责发送报文，一个是负责接受报文，这些系统调用可通过调用 `send (dest,& mptr)` 和 `receive (addr, &mptr)` 这两个库过程来实现。`send` 调用将发送的消息用指针 `mptr` 传给进程，用 `dest` 标记目的地，然后阻塞调用者直到消息发送为止，`receive` 进程阻塞调用者直到消息接收为止，当调用结束时将消息拷贝到 `mptr` 指向的缓冲区，同时调用者不再被阻塞，`addr` 参数是正在监听的接收者的地址，这两个系统调用及其参数的其他实现方法将在本节之后讨论。

2.3.2 客户和服务器的一个示例

为了更深入探讨客户和服务器的的工作，本节将用 C 语言给出一个客户和一个文件服务器的框架，其中客户-服务器共享一些定义信息，将这些定义收集在头文件 `header.h` 中，如图 2-8 所示。客户-服务器都可用：


```
#include <header.h>
```

语句来使用它，此语句使得整个头文件在源程序编译之前将逐字嵌入源程序中。

```
/*客户与服务器所需要的定义*/
#define MAX_PATH      255          /* 文件名最大长度*/
#define BUF_SIZE      1024        /* 一次传送多少数据*/
#define FILE_SERVER    243        /* 文件服务器网络地址*/
/*所允许操作的定义*/
#define CREATE         1          /* 创建一新文件*/
#define READ           2          /* 读部分文件并返回*/
#define WRITE          3          /* 写部分文件*/
#define DELETE         4          /* 删除当前文件*/
/*错误代码 */
#define OK              0          /* 操作正确完成*/
#define E_BAD_OPCODE   1          /* 请求未知操作*/
#define E_BAD_PARAM    -2         /* 参数错*/
#define E_IO            -3         /* 盘错或其他 I/O 错*/
/*消息格式定义 */
struct message{
    long source           /*发送者标识*/
    long dest;            /*接收者标识*/
    long opcode;          /*哪种操作：CREATE，READ，等*/
    long count;           /*传送字节数*/
    long offset;          /*开始读写文件位置*/
    long extra1;          /*附加字段 1*/
    long extra2;          /*附加字段 2*/
    long result;          /*操作结果报告*/
    long name[MAX_PATH];  /*正操作文件名*/
    long data[BUF_SIZE];  /*读写数据缓冲区*/
};
```

图 2-8 客户和服务端所使用的 header.h 文件

让我们看一下头文件：它首先定义了两个常量 *MAX_PATH* 和 *BUF_SIZE*，这两个常量决定了在报文中两个所需要的数组的大小，前一个定义一个文件名有多少个字符，后一个定义了在一个读写操作中存放数据的缓冲区的大小。下一个常量 *FILE_SERVER* 指出实际可以发报文的文件服务器的网络地址。

第二组常量定义了操作的数目，它用来确认客户和服务端以什么编码表示 *READ*，什么编码表示 *WRITE*，我们这里只写出四个，实际的系统常量要比这多一些。

每个应答都有一个结果编码。如果操作成功，结果编码一般包含有用户信息(如实际

读的字节数)。如果没有数据返回(如建立一个文件),就用“OK”返回。如果因某种原因操作失败,结果编码将用编码 *E_BAD_OPCODE*、*E_BAD_PARM* 等告诉错误的原因。

最后,看头文件中最重要的一部分,关于报文的定义。在我们的例子中,这是一个有 10 个字段的结构。所有从客户到服务器的请求都采用这种格式,所有的应答也一样。在真正的系统中,可能没有固定格式的信息(并不是在一切情况下所有的字段都需要)。这里为了解释方便, *source* 和 *dest* 域指出了发送者和接收者, *opcode* 域是上面已定义过的操作 *CREATE*、*READ*、*WRITE* 或 *DELETE* 中的一个。 *count* 和 *offset* 字段用作参数, *extra1* 和 *extra2* 字段定义为附加参数,以便将来扩展服务器, *result* 不是用于客户对服务器的请求,而是保存服务器对客户的应答消息。最后,有两个数组,第一个 *name* 保留正在访问的文件名,第二个 *data* 保存对 *READ* 的应答数据,或在 *WRITE* 中发送到服务器中的数据。

现在让我们看如图 2-9 所示的程序,在(a)中我们建立了一个服务器,在(b)中我们建立了一个客户。服务器很简单,主循环调用 *receive* 取得一个请求消息。第一个参数用来给出呼叫者的地址;第二个参数指出输入报文的消息缓冲区,库程序 *receive* 嵌入到内核中,挂起服务器直到消息到达。当有一条消息到达时,服务器继续运行并根据操作的类型调度。对于每种操作,调用不同的子程序,为输入与输出服务的缓冲区都要作为参数给出。进程检验输入消息 *m1*,并在 *m2* 建立应答,并返回一个函数值给 *result* 字段。当 *send* 完成时,服务器返回到循环头,执行 *receive*,等待下一个输入消息。

在图 2-9(b)中,给出使用服务器拷贝文件的程序,其程序体包括一个循环(从源程序读出数据并写到目的文件中),这个循环将不断重复,直到源文件拷贝完成,根据返回码中的 0 和负数来判断循环是否结束。

循环第一部分是读操作建立一条消息,并发送到服务器。接收到应答后,进入循环的第二部分,将从服务器取得的数据写到目标文件中。图 2-9 程序只是代码的概括,忽略了许多细节问题,例如, *do_xxx* 进程(实际做的工作),没有给出,而且没有错误检查。还有客户和服务器的互操作也不明确。在下面的章节里,将讨论客户和服务器之间关系的细节问题。

```
#include <header.h>
void main(void)
{
    struct message m1,m2;           /*输入与输出消息*/
    int r;                          /*结果代码*/
    while(1){                       /*服务程序循环执行*/
        receive(FILE_SERVER,&m1);   /*封锁,等待消息*/
        switch(m1.opcode){          /*分派请求类型*/
            case CREATE: r = do_create(&m1,&m2); break;
            case READ:   r = do_read(&m1,&m2);   break;
            case WRITE:  r = do_write(&m1,&m2);  break;
            case DELETE: r = do_delete(&m1,&m2); break;
            default:     r = E_BAD_OPCODE;
```

```

    }
    m2.result = r;                /*返回结果到客户端*/
    send(m1.source,&m2);          /*发送应答*/
}
}

(a)

#include <header.h>
int copy(char *src,char *dst)     /*使用服务程序拷贝文件过程*/
{
    struct message m1;            /*消息缓冲区*/
    long position;                /*当前文件位置*/
    long client = 110;            /*客户地址*/

    initialize();                 /*准备执行*/
    position = 0;
    do {
        /* 从源文件取一数据块 */
        m1.opcode = READ;          /*读操作*/
        m1.offset = position;      /*文件当前位置*/
        m1.count = BUF_SIZE;      /*读多少字节*/
        strcpy(&m1.nac,src);       /*将所读的文件名字拷贝到消息*/
        send(FILE_SERVER,&m1);     /*发送消息到服务器*/
        receive(client,&m1);       /*阻塞，等待应答*/

        /* 将刚接收的消息拷贝到目标文件 */
        m1.opcode = WRITE;         /*写操作*/
        m1.offset = position;      /*文件当前位置*/
        m1.count = m1.result;      /*写多少字节*/
        strcpy(&m1.name,dst);     /*将所写的文件名字拷贝到缓冲区*/
        send(FILE_SERVER,&m1);     /*发送消息到服务器*/
        receive(client,&m1);       /*阻塞，等待应答*/
        position += m1.result;     /* m1.result 为所写的字节数*/
    }while(m1.result>0);           /* 重复执行直到做完*/
    return(m1.result >= 0? OK :m1.result); /*返回 OK 或错误代码*/
}

(b)

```

图 2-9 (a) 一个简单服务器

(b) 一个客户使用服务器拷贝文件的过程

2.3.3 寻址

客户为了发送消息给服务器，它必须知道服务器的地址。前面一节的例子里，服务器的地址是一个常数，在 *header.h* 中给定。这种方法只能用于一个特殊的简单子系统中，我们需要功能更强的寻址机制。这一节，讨论关于寻址的问题。

在例子里，给文件服务器分配了数字地址(243)。但我们没有真正意识到这意味着什么，它是否指一个特定的机器或一个特定的进程。如果这指一个特定的机器，发送内核能从报文结构中取出它，作为硬件地址将包发送给服务器。所有的发送内核用 243 作为数据链路地址的帧，发送到 LAN 上，服务器的接口板将检查帧，辨认出 243 是自己的地址并接收它。

如果只有一个进程在目的机器上运行，内核将知道怎样处理输入的消息——将它交给一个唯一的正在运行的进程。然而，如果在目的机器上有多个进程在运行怎么办？哪一个进程将得到报文？内核是无法知道。结果，如果用网络地址识别进程意味着在每个机器上只能运行一个进程，有时这个限制并不致命，有时则是个重要的约束。

一个可供选择的寻址系统是将消息传送给进程而不是计算机，尽管这种方法减少了关于谁是真正的接收者意义不明确的地方，但它却带来了进程如何识别的问题。一个常见的设计就是用两部分名字：机器号和进程号，如 243.3 和 @243，或和机器 243 上的进程 4 等类似设计。机器号用于使内核将消息正确地发送到适当的机器上，进程号用来使内核决定消息要给哪一个进程。这种解决办法最好是每个机器将它的进程从 0 开始编号，不需要互相协调，因为进程 0 在机器 243 上和进程 0 在机器 199 上不会存在不明确的地方，前一个形式是 243.0，后一个是 199.0，参见图 2-10(a)。

在这种寻址方式上的微小变化是用 *machine.local_id*(机器.本地_id)代替 *machine.process*(机器.进程)。本地_id 字段常常是一个 16 位或 32 位的随机数，一个进程类似一个服务。开始通过系统调用告诉内核，它想听本地_id(*local_id*)。然后当一个以 *machine.local_id*(机器.本地_id)号编址的消息进入时，内核知道是哪一个进程所要的消息。在伯克利的 UNIX 系统上，大部分通信都采用这种方式，即用 32 位 Internet 地址指定机器，给出 16 位数字用于本地_id 字段。然而，*machine.process*(机器.进程)寻址方式并不理想，特别是由于用户要知道服务器在哪里，因此是不透明的，而透明性正是分布式系统设计的目标。为了对这种情况详细进行说明，假设文件服务器通常在机器 243 上运行，但有一天机器坏了，机器 176 可用，但程序编译时使用的头文件中机器名是 243，如果该服务器不能使用，它们也无法工作，这种状况是我们所不希望的。

一个可选择的方法是给每个进程分配唯一的地址，而不确定其机器号。如果要达到这个目标，必须有一个中央控制地址分配器，只包含一个计数器。当接收到一个地址请求时，它返回计数器的当前值并将计数器自动加 1。这个设计的缺点是，像这样的中央控制部件无法用于一个大的系统，因此应该避免。

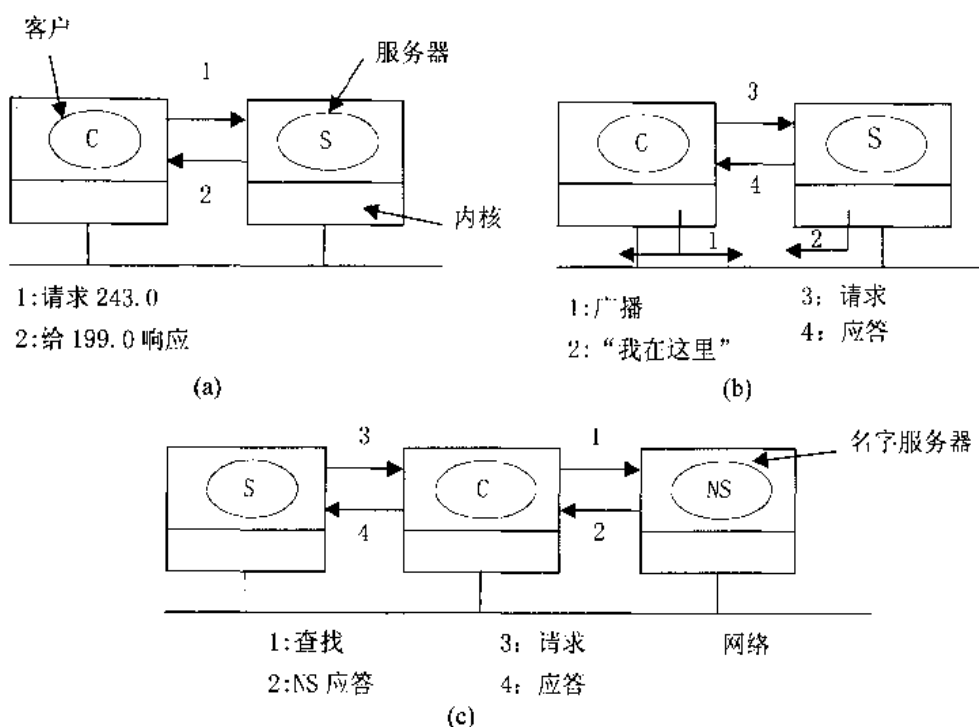


图 2-10 (a) 机器.过程编址方式
(b) 带有广播的进程编址
(c) 通过名字服务器进行地址查询

另一个分配进程标识号的方法是：让每个进程在相当大且专用的地址空间中选择自己的标识号，如 64 位二进制整数的空间。两个进程选择同一数字的可能性很小，系统平衡很好。然而在这里也存在一个问题，发送内核怎么知道发送消息给哪一个机器呢？在支持广播的局域网中，发送者广播一个特殊的定位包，包含目的地进程的地址。因为这是一个广播包，在网络上所有的机器均可收到，所有的内核检查并查看地址是不是它们的，如果是回答“我在这里”消息给出网络地址(机器号)，发送内核使用这个地址并“记住”它，避免下一次再广播，这种方法如图 2-10(b)所示。

尽管这种设计是透明的，甚至用了缓存，但广播给系统增加了额外的负担，其实可以用一台额外的机器提供高层的机器名和机器地址的映射来避免这种额外的负担，如图 2-10(c)所示。当使用这个系统时，进程像服务器一样，可用 ASCII 字符串来指出，并且这些字符串可以嵌入程序中，不用二进制的机器和进程号。每次客户机运行，首先试图使用服务器，客户机发出一请求消息给一个特殊映射服务器，常常称作名字服务器(name server)，问一个目前服务器所在的机器号，有了这个地址后，可以直接发送请求。在上面的情况下，地址能被暂存。

总结一下，有以下几种给进程编址的方法：

- (1) 在客户机代码中指明机器.号(machine.number);
- (2) 让进程选择随机地址，用广播定位进程；
- (3) 在客户机中存放 ASCII 服务器名字，运行时寻找它。

它们中的每一个都有问题，第一个不透明，第二个给系统造成额外负担，第三个需要一个中间部件——名字服务器。当然名字服务器可以复制，但这样做，带来了维护其数据一致性的问题。

一个完全不同的方法是用特殊的硬件，让进程选择随机地址，然而不同于广播定位它们，网络接口电路设计成让处理机存储自己的进程号，帧使用进程地址而不是机器地址。每个帧到来以后，网络接口电路检查帧，看目的进程是否在它的机器上，如果是，帧将被接收，否则，不被接收。

2.3.4 阻塞与非阻塞原语

以上描述的消息传送原语称为阻塞原语(有时称为同步原语)。当一个进程调用 *send* 原语,它指定了目的地以及发送到该目的地的缓冲区数据。消息被传送时，发送的进程被阻塞 (即挂起)。直到消息传送完毕，其后的指令才能继续执行，如图 2-11(a)。同样，调用 *receive* 时，直到一条消息被实际接收并放入由参数指定的缓冲区时才能返回控制权。在一条消息到达前，调用 *receive* 的进程一直挂起，甚至可以长达 1 个小时。在有些系统中，接收者可以指明希望从哪个发送者那里接收消息，在这种情况下，它保持阻塞直到该发送者的消息到达。

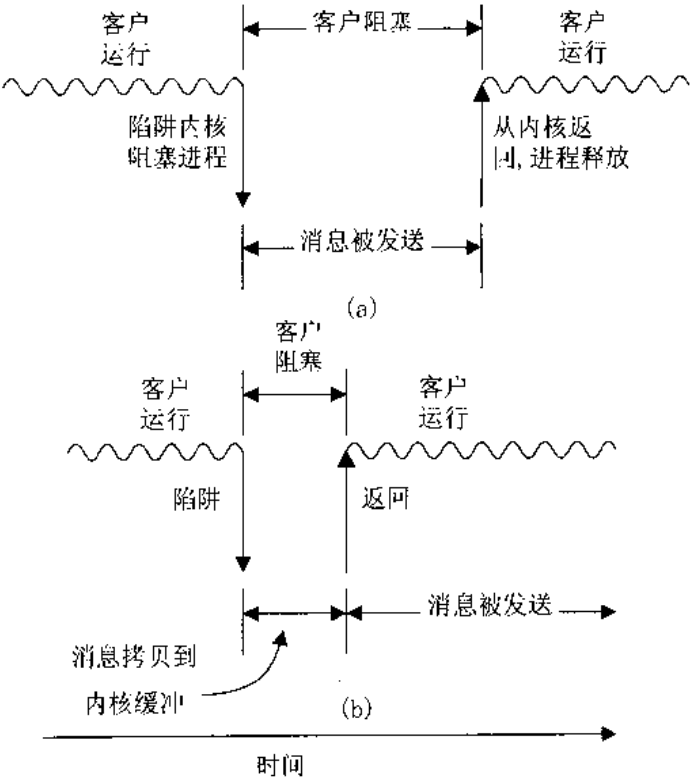


图 2-11 (a) 阻塞发送原语
(b) 非阻塞发送原语

和阻塞原语相对应的是非阻塞原语(有时叫异步原语)。如果 *send* 原语是非阻塞的，

在发送消息前它立即将控制权返回给调用进程。这种方式的优点是：调用发送的进程可以和消息传送并行运行，而不是让 CPU 空闲(设想没有其他进程可执行)。阻塞原语和非阻塞原语的选择一般取决于系统设计者(只有一个原语可用)，但是在一些系统中两种都可以使用，用户可以选择他所喜欢的一种。

然而，非阻塞原语有一个主要的缺点以至于掩盖了性能上的优点，即在消息被发送之前，发送者不能修改消息缓冲区。后继的进程在传输过程中覆盖该消息是很可怕的。更糟的是，发送的进程不知道传送何时进行，所以它无法知道什么时候重新使用缓冲区是安全的。这种情况几乎无法避免。

这里有两种可行的解决方法。第一个解决方法是内核将消息拷贝到内部缓冲区，其后允许进程继续执行，如图 2-11(b)。从发送者的观点看，这个方案和阻塞调用的相同：一旦它获得控制权，就可以重新自由地使用缓冲区。当然，消息还没有被传送，所以发送者事实上并没有被它阻碍。这个方法的缺点是：每一个要传送的消息都要从用户区拷贝到内核区。对于许多网络接口，消息以后将被拷贝到硬件传输缓冲区，第一次拷贝显然被浪费了，额外的拷贝降低了系统的性能。

第二个解决方法是当消息被发送后，中断发送者并通知它缓冲区可用，这里不需要备份，节省了时间。但用户级的中断使编程更具有技巧性更加困难，而且受制于竞争情况，它可使结果不能再现。大多数专家认同，尽管这种方法高效且允许最大程度的并行化，缺点还是大于优点：基于中断的程序不易编写正确，因为它们出错时几乎无法调试。

有时可以在发送者的地址空间启动一个新的控制线程(将在第 4 章介绍)来产生中断。尽管这种中断比原始中断清晰，但还是比同步通信复杂。如果仅有一个控制线程可用，选择如下：

- (1) 阻塞传送(CPU 在数据传输时空闲)；
- (2) 使用拷贝非阻塞传送(CPU 时间浪费在额外的拷贝上)；
- (3) 使用中断的非阻塞传送(使编程困难)。

在正常的情况下，第一种选择是最好的。虽然它不增加并行度，但容易理解和实现。而且它不需要任何内核缓冲区。进一步，从 2.11(a)和 2.11(b)的对比来看，如果不需要拷贝，消息能更快地发送出去。另一方面，如果某些应用中进程和传送需要重叠，使用拷贝的非阻塞发送是最好的选择。

从记录来看，我们需要指出的是有一些作者使用不同的标准来区分同步原语和异步原语(Andrews, 1991)。我们认为，同步原语和异步原语的实质区别在于发送者在取回控制权后能否很快重新使用缓冲区，而不用害怕干扰发送。至于被发送的数据何时到达接收者则是另外一回事。

另一种观点认为，同步原语指的是在接收者接收到消息并将确认消息返回给发送者之前阻塞发送者。从这种观点来看，其他方式都是异步的。已经达成共识的是，如果发送者能在消息被拷贝或发送之前取回控制权，这个原语就是异步的。同样，如果发送者被阻塞到接收者确认这条消息，则这是一个同步原语。

分歧在于如何划分中间状态(消息被拷贝或被拷贝并发送，但没有得到确认)。操作系统设计者趋向于采用我们的观点，因为他们关心的是缓冲区的管理和消息的传送。程序语言设计者趋向于另一种定义，因为它属于语言一级的。

发送可以被阻塞或不被阻塞，接收也是一样。一个不被阻塞的接收原语能告诉内核缓冲区的位置，并几乎立即返回控制。那么，调用者怎样知道操作完成了？一种方法是提供一个明确的等待 (*wait*) 原语，它允许接收者在需要时阻塞。另一种方法(或除了等待原语)是，设计者提供一个检测(*test*)原语以允许接收者进入内核检查其状态。这种想法的一个变种是有条件接收(*conditional_receive*)，无论接收到消息或信号失败，都可立即或在一定的时间间隔内返回。最后，也可以用中断来完成。大多数情况下，阻塞形式的 *receive* 原语更为简单并被广泛接受。

如果在单地址空间有多线程控制，消息到达时可以自发创建一个新的线程。在第 4 章讨论完线程后，再回头讨论这个问题。

和阻塞及非阻塞调用联系很紧密的问题是超时。在使用阻塞的发送原语的系统中，如果没有应答，发送者将一直阻塞下去。为了防止这种情况，在一些系统中调用者采用在一个时间间隔内等待应答，在此期间内如果没有应答，发送调用认为这是个错误状态并终止等待。

2.3.5 有缓冲和无缓冲原语

就像系统设计者可以选择阻塞和非阻塞原语一样，他们也可以选择有缓冲和无缓冲原语。我们上面描述的原语基本上是无缓冲原语。这意味着一个地址指定给一个特定进程，如图 2-9 所示。如调用 *receieve(addr,&m)* 告诉运行的机器内核，调用的进程正在监听地址 *addr*，并且准备接收发送到那个地址的消息。*m* 指出了—个消息缓冲区用于保存传送来的消息。当消息到来，调用接收原语的内核将消息拷贝到缓冲区，并解除该进程的阻塞。在图 2-12(a)说明了将一个地址指定给—特定进程的情况。

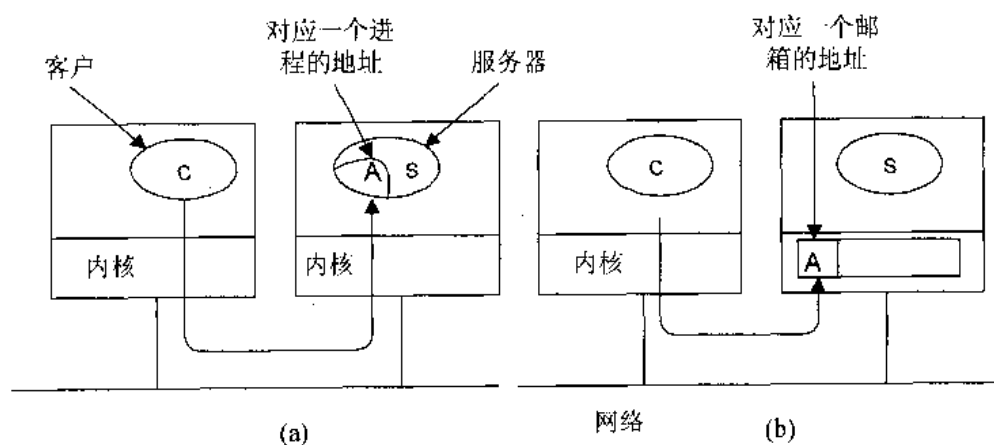


图 2-12 (a) 无缓冲消息传输
(b) 有缓冲消息传输

只要服务器在客户机调用 *send* 原语之前调用 *receive* 原语，这个方案就运行良好。*receive* 调用只是这样一种机制，它告诉服务器内核服务器正在使用的地址以及存放到来消息的位置。当发送比接收先发生，就会产生问题。服务器的内核如何知道它的哪一个进程(如果有)在使用最近到达的消息中的地址，如何知道在哪里拷贝消息？回答很简单，不

知道。

一个实现方案是丢弃消息，使客户机超时，并希望服务器在客户机传送前调用接收原语。这个方法很容易实现。不幸的是，客户机(更可能的是客户机内核)可能在这以前已试了许多次。更糟的是，如果多次失败，客户机内核可能会放弃并错误地认为服务器损坏或者该地址无效。

相似的，如图 2-9(a)，设想有两个或更多客户机使用服务器。服务器从其中一个客户机接受消息后，它不再监听自己的地址，直到完成工作并返回到循环开始时的 *receive* 调用。如果工作只进行了一会，而其他客户机多次试图向它发送消息，其中的一些可能放弃，这取决于它们重发计时器的值和它们的耐心。

处理这个问题的另一种方法是：让接收内核在短时间内保存到来的消息，以防合适的 *receive* 原语可以很快完成。当一个“不想要”的消息到达时启动计数器，如果在合适的 *receive* 原语启动前计数器到时，消息被丢掉。

尽管这些方法减少了消息被丢掉的可能性，但仍然产生了一个问题，即如何存储和管理过早到来的消息。这里需要缓冲区以及缓冲区的分配、释放和一般管理。处理缓冲区管理的一个简单的概念化方案是定义一个称为邮箱的新的数据结构。对接收消息感兴趣的进程可以让内核为之建立一个邮箱，并指定一个地址以便于寻找网络信包。因此，所有具有该地址的输入消息被放入邮箱中，调用 *receive* 时只要从邮箱中取出一条消息，邮箱为空时阻塞(对阻塞原语而言)。这样，内核知道如何处理到来的消息，并有地方存放它们。这个技术常称为有缓冲原语(buffered primitive)，如图 2-12(b)所示。

我们首先看到，使用邮箱表面上消除了因为消息被丢掉或客户机放弃引起的竞争情况。然而邮箱是有限的，可能会满。当一条消息到达一个已满的邮箱时，内核又将面临选择，是寄希望于至少一条消息被及时取出而暂时保存该消息，还是将之丢弃。在无缓冲情况下我们也要作这样的选择。尽管我们可以减少错误的可能性，我们不能消除它，甚至也不能试图改变它的本质。

在一些系统中，还有另外一种方法：如果目的地没有空间存储，阻止进程发送消息。为实现这个方案，发送者必须阻塞到表明该消息已被接收的确认消息返回。如果邮箱满了，可以退回发送者并挂起，就像调度器在它试图发送消息前就决定挂起它一样。当邮箱中有空间可用时，发送者可以重试。

2.3.6 可靠的和非可靠原语

以上我们设想了当客户机发送消息时，服务器会接收它。实际中使用的真正系统通常比抽象模型更为复杂。消息可能会丢失，这影响了消息传送模型的语义。假设使用阻塞原语。当一个客户机发送一条消息，在消息发送完毕之前它被挂起。然而，当它重新开始，并不能保证消息已被成功发送出去，消息可能已经丢失了。

有三种不同方法可以解决这个问题。第一种是重新定义非可靠的 *send* 语义。系统无法保证消息成功发送。完成可靠的通信依赖于用户。邮局就使用这种方式工作，当你将信投入邮箱后，邮局尽最大努力投递，但不承诺什么。

第二种方法是要求接收机器的内核给发送机器的内核发送一个确认消息。只有当收到这个确认消息后，发送内核释放用户(客户)进程。确认消息从一个内核传送到另一个内

核，无论是客户还是服务器都看不到确认消息。正像客户到服务器的请求是由服务器的内核确认一样，从服务器返回客户机的应答由客户机的内核确认。因此一个请求和应答现在需要 4 条消息，如图 2-13(a)所示。

第三种方法利用了客户-服务器通信的优点，事实上，客户-服务器通信由从客户到服务器的请求和从服务器到客户的应答构成。在这种方式中，客户机在发送消息后阻塞，服务器的内核不发送确认消息，而是将应答作为确认消息。因此，客户一直阻塞直到应答消息到来为止，如果时间太长，发送内核会重发请求以防消息丢失，这种方法如图 2-13(b)所示。

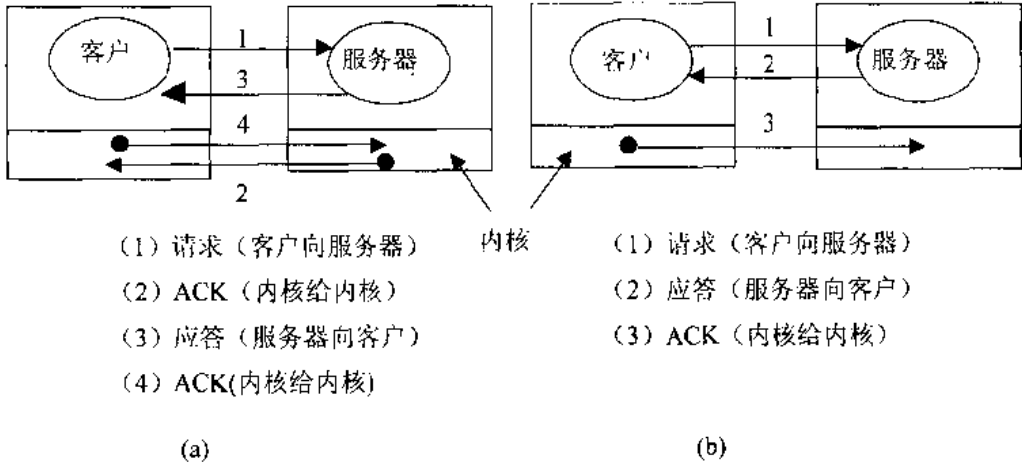


图 2-13 (a) 独立的确认消息
(b) 应答用作请求的确认消息
(需要注意的是应答全部是由系统内核处理的)

尽管应答用于请求的确认，但没有对应答的确认。这个遗漏是否重要由请求的性质决定。例如：如果一个客户机请求服务器读一个文件块，如应答丢失，客户机将重复请求，服务器重发文件块，没有什么损害且不浪费时间。

另一方面，如果请求在服务器上进行大量的运算，在服务器确信客户机已经收到应答的情况下，丢失答案将是非常可惜的。因为这个原因，有时使用客户内核到服务器内核的确认。直到信息包被接受，服务器的 *send* 原语还没有完成，服务器保持阻塞(设想使用了阻塞原语)。在任何情况下，如果应答丢失并重新传送请求，服务器的内核看到这个请求是原来的，不用唤醒服务器而仅仅重新发送应答。这样在某些系统中应答被确认而有些系统中则不是。

对图 2-13(a)和图 2-13(b)的折衷常常是这样工作：当一个请求到达服务器内核，计时器启动，如果服务器能很快给出应答(在计时器到时之前)，这个回答就是确认消息。如果计时器超时，单独发送确认消息。这样在大多数情况下，只需要两条消息，只有当有复杂的请求出现时，才采用第三种方法。

2.3.7 客户-服务器模式的实现

上节我们讨论了客户-服务器模式的四个设计问题，寻址、阻塞、缓冲和可靠性，每

一个问题都有一些选择。表 2-1 归纳了主要的选择。对于每一项我们列出了三种可能，简单的计算表明共有 $3^4=81$ 种组合，但并不是所有的组合都一样好。然而，就在这个领域中(消息传递)系统设计者需要考虑到许多因素以选择一个通信原语集。

表 2-1 通信原语的四个设计问题以及一些可供选择的原理

项目	选择 1	选择 2	选择 3
寻址	机器数目	共享进程地址	ASCII 名字通过服务器查找
阻塞	阻塞原语	具有拷贝到核的非阻塞	具有中断的非阻塞
缓冲	无缓冲，丢弃不期望的消息	无缓冲，临时保持不期望的消息	邮箱
可靠性	非可靠	请求 - 确认 - 应答确认	请求 - 应答 - 确认

因为如何传递消息的实现细节在某种程度上依赖于通信原语集的选择，这些选择还会导致一些有关实现、协议和软件方面的问题。首先，我们假设所有网络有最大的包长度，一般最多有几千字节，大于它的报文必须分成多个包分别传递。有些包可能丢失或损坏，而且它们可能以一个错误的顺序到达。为了讨论这个问题，常常给每一个报文分配一报文号，并将该报文号放入属于这个报文的包中，由一系列的数字给出信包的顺序。

然而，还必须解决确认消息的使用问题。一个策略是对每一个单独的包进行确认，另一个是仅仅对一个报文确认。前者的优点是如果一个包丢失，仅仅这个包需要重新传递，但缺点是在网络上会有更多的包。后一种的优点是信包少，但缺点是一旦包丢失恢复起来更复杂(由于客户机超时要求重传整个报文)，这个选择很大程度上依赖于所用网络的丢失率。

另外一个有趣的问题在客户-服务器通信的低层协议上。表 2-2 给出了 6 个包类型，它们常常用于实现客户-服务器协议。第一个是 REQ 包，用于从客户到服务器发送请求报文(为简单起见，本节我们假定每一个报文刚好是一个包)。第二个是 REP 包，将结果从服务器回到客户机。然后是 ACK 包，在可靠协议中用于确认正确接收前面的包。

表 2-2 客户-服务器协议中的包类型

代码	包类型	来源	至	说明
REQ	请求	客户	服务器	客户要求服务
REP	应答	服务器	客户	服务器对客户机的应答
ACK	确认	服务器、客户	其他	前面的包已到达
AYA	你在这里吗?	客户	服务器	查看服务器是否崩溃
IAA	我在这里	服务器	客户	服务器没有崩溃
TA	再试一次?	服务器	客户	服务器没有空间
AU	地址未知	服务器	客户	没有进程在使用此地址

后面四个包不是基本的，但却经常使用。假设这样一种情况，一个请求已经成功地从客户发送到服务器，并且确认消息已收到。此时，客户的内核知道服务器正处理请求。但是如果在合理的时间内没有收到应答，那么发生了什么事？是这个请求真的很复杂还是服务器崩溃了？为区分这两种情况，有时提供了 AYA 包，这样客户可以询问服务器进展情况。如果回答是 IAA 包，客户机内核知道一切运行良好，继续等待。当然更好的是一个

REP 包。如果 AYA 包没有任何回答，客户机内核等待短时间后再试一次。如果这个过程失败了数次，客户机内核通常放弃它，并向用户报告错误。AYA 和 IAA 包还可用在 REQ 包不被确认的协议中。它们允许客户机检查服务器的状态。

现在，我们看看最后两个包类型(TA、AU)。它们用于 REQ 包不被接受的情况。发生这种情况有两个原因，对客户内核来说区分它们很重要。一个原因是该请求寻址的邮箱满了。通过将这个包发回到客户机内核，服务器内核指出这个地址是有效的，而且稍后可以重复该请求。另一个原因是地址不属于任何进程或邮箱，再重复也没用。

在没有使用缓冲时，或服务器正在 *receive* 调用中被阻塞，这种情况也会发生。如果服务器内核忽略某地址在 *receive* 调用之间存在也会带来问题。在一些系统中，服务器可以执行一个调用，作用是将某一地址登记到内核。这样，内核至少可以区分不被监听的地址和错误的地址。在前一种情况它可以发送 TA 包，而后一种情况发送 AU 包。

许多包顺序都是可能的，一些常用的如图 2-14 所示。在图 2-14(a)中我们有直接的请求/应答包，没有确认消息。在图 2-14(b)中我们使用每个报文分别确认的协议。在图 2-14(c)，我们将应答作为确认消息，将信包减少到 3 个。最后在图 2-14(d)中我们看到客户正忙于检查服务器是否存在。

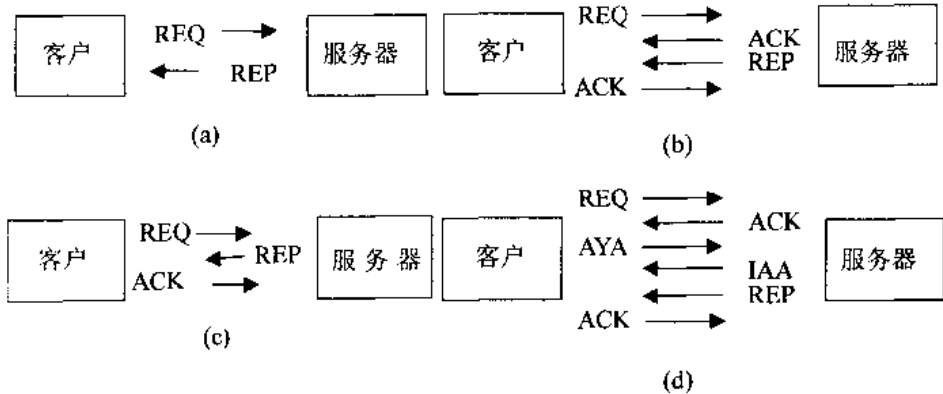


图 2-14 一些客户机-服务器通信使用的包交换示例

2.4 远程过程调用

尽管客户-服务器模式为构造分布式操作系统提供了一种便利的方法,但它也存在着无法克服的缺陷:即所有通信建立的基础都是输入/输出。过程 *send* 与 *receive* 基本上是在做 I/O 操作。由于 I/O 并不是集中式系统的一个主要概念，在分布式计算中将它作为基础会使此领域中的许多工作者产生误解。他们的目标是使分布式的计算看起来像集中式计算一样。用 I/O 为基础实现它并不是一个好的办法。

人们很早就意识到这个问题，但直到 1984 年才由 Birrell 和 Nelson 提出一个全新的解决方法。尽管他们提出的思想极其简单(曾经有人考虑过)，但实现起来却有许多微妙的地方。本节我们将讨论它的概念、实现方法及优缺点。

简而言之，Birrell 与 Nelson 提出的方法就是允许程序去调用位于其他机器上的过程。当位于机器 *A* 的一进程调用机器 *B* 上的某过程时，机器 *A* 上的该进程被挂起，被调用的过程在机器 *B* 上执行。调用者将消息放在参数表中传送给被调用者，结果作为过程的返回值返回给调用者。消息的传送与 I/O 操作对于编程人员是不可见的。这种方法称为远程过程调用(remote procedure call)，或简称为 RPC。

这一想法尽管听起来很简单，但仍存在着许多细微的问题。首先，由于调用的进程与被调用的过程运行在不同的机器上，因而在不同的地址空间执行，这就导致了问题的复杂化。尤其当两台机器不是一种型号时，在机器之间传递参数与调用结果很复杂。最后，调用者和被调用者都有可能会崩溃，任何一种可能的失败都会引起不同的问题。当然，大多数问题还是可以解决的，RPC 是广泛应用于分布式操作系统的一种技术。

2.4.1 基本 RPC 操作

为了解 RPC 是如何工作的，首先要了解清楚传统的(即单机上)过程调用是如何工作的。考虑这样一个过程调用：

```
count=read(fd, buf, nbytes);
```

这里 *fd* 是一个整数，*buf* 是一个字符型数组，*nbytes* 是另一个整数。若主程序调用该过程，调用前堆栈的情况如图 2-15(a)所示。调用开始时，如图 2-15(b)，调用者按序将参数压入堆栈，后进入的先弹出。(C 编译器将 *printf* 的参数按相反顺序压入堆栈的原因是使 *printf* 在执行时总能找到它的第一个参数，即格式串)。在过程 *read* 执行后，它将返回值送入寄存器中，从栈中取出返回地址并将控制权交给调用者。然后，调用者从堆栈中取出参数，返回到调用点，如图 2-15(c)所示。

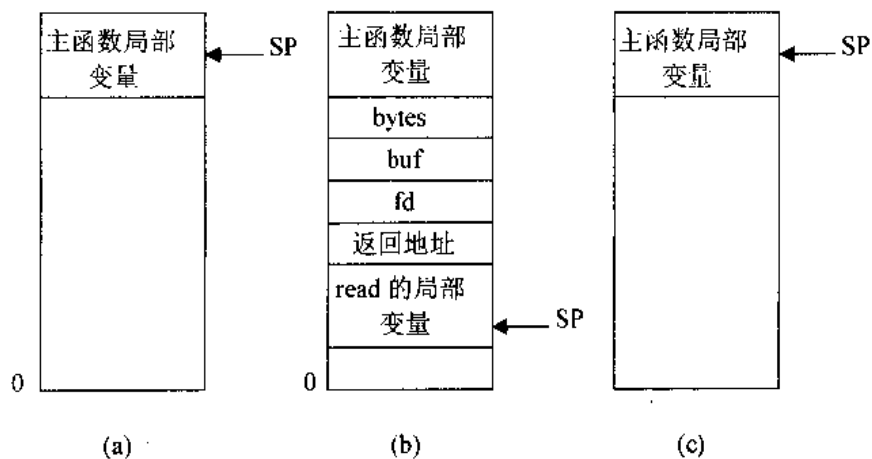


图 2-15 (a) 调用 *read* 之前的栈
(b) 调用过程处于激活状态时的栈
(c) 返回调用者之后的栈

有几个问题是值得注意的。第一，在 C 语言中参数的调用分为值参调用与变参调用。如 *fd* 与 *nbytes* 之类的值参，调用时只需要将它们拷贝到堆栈中，如图 2-15(b)所示。对被调用的过程来说，值参仅仅是一个初始化了的局部变量。该过程可以改变它，但其改变

不影响调用方的初始值。

在 C 语言中, 变参是一个指向变量的指针(即变量的地址), 而不是变量的值。因为数组在 C 语言中常以变参的形式传递, 所以在 *read* 中的第二个参数 *buf* 是一个变参。实际上压入堆栈的是该字符数组的地址。如果被调用的过程使用这个参数向该字符数组存入数据, 它的确修改了调用过程中这个数组的值。值参调用和变参调用的这种区别对 RPC 来说是很重要的。

此外还存在着一种 C 语言中不使用的参数传递机制, 它叫做复制/恢复调用(*call-by-copy/restore*)。在以这种方式执行调用时, 调用者将变量拷入堆栈, 这一点是与值参调用一样的。调用完成后, 将栈中变量的值拷回并覆盖原有的变量值。大多数情况下, 此方法与变参调用的效果一样。但是在有些场合, 例如在参数列表中多次出现同一参数时, 两者的语义是不同的。

到底使用哪一种参数传递机制通常是由语言开发者来决定的, 而且它是语言的固有特性。它有时也与传递的数据类型相关。例如在 C 语言中, 整型与其他数值类型常作为值参传递, 而数组总是以变参的形式传递。不同的是, PASCAL 语言的编程人员可以选择每个参数的传递方式。在缺省状态下是值参调用。编程人员可以在指定参数前加关键字 *VAR* 来强制该参数为变参调用。一些 Ada 编辑器利用复制/恢复来输入和输出参数, 还有一些采用变参形式。在语言定义中同时允许两种参数传递机制, 这使得这种语言的语义有些模糊。

RPC 的内在思想是使远程的过程调用看上去就像在本地的过程调用一样。换句话说, 我们希望实现 RPC 的透明性——调用者不应该意识到此调用的过程是在其他机器上执行的, 反之亦然。设想一个程序需要从一文件中读取一些数据。编程人员在代码中调用 *read* 即可取得数据。在一个传统的(单处理器)系统中, 连接器将 *read* 例程从函数库中取出并插入目标程序中。这是一个小过程, 通常用汇编语言编写, 它将参数放入寄存器, 激活内核的陷阱中断并调用系统调用 *READ*。库函数 *read* 实质上是用户代码与操作系统的接口。

尽管 *read* 激活了内核陷阱, 但它仍然是通过将参数压入堆栈这种常规的方式来调用的, 如图 2-15 所示。因此, 编程人员感觉不到 *read* 调用是如何在进行的。

RPC 使用与本地调用相似的方法获得透明性。当 *read* 为一远程过程调用时(例如, 它将运行在文件服务器上), *read* 的一个不同版本, 称为客户存根(*client stub*), 被放入库中。和前面的本地调用一样, 它采用如图 2-15 所示的调用顺序并同样激活了内核的陷阱。不同的是, RPC 不是将参数放入寄存器中并要求内核返回结果, 而是将参数打成信包, 请求内核将该消息发送到服务器, 如图 2-16 所示。在发送消息后, 客户存根调用 *receive* 原语, 然后阻塞直至收到服务器来的应答。

当消息到达服务器后, 内核将消息传送给与实际服务器进程相捆绑的服务器存根(*stub*)。通常, 服务器存根调用 *receive*, 然后将自己阻塞等待消息的到达。服务器存根拆开信包从消息中取出参数, 然后以一般方式调用服务器进程(即与图 2-15 所示的一样)。从服务器进程的角度来看, 就像由本地的客户进程直接调用一样——所有参数和返回地址都在它们的堆栈中, 没有任何异常。服务器执行它的工作并以一般方式将结果返回调用者。例如, 在 *read* 的例子中, 服务器将在第二个参数所指向的缓冲区内填入数据。这个缓冲区是在服务器存根内的。

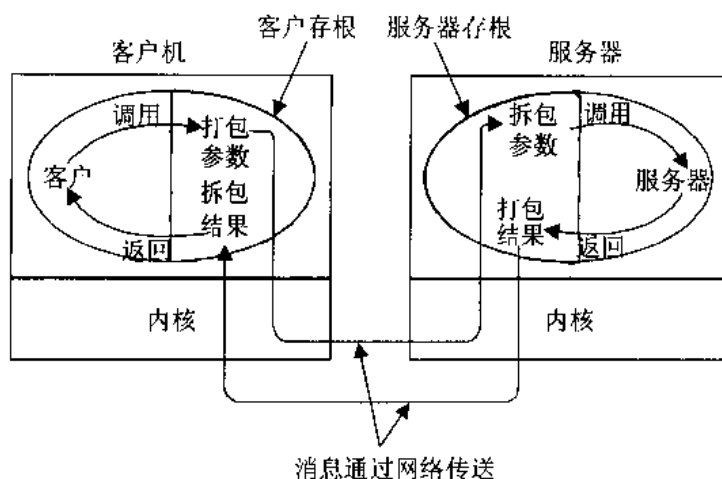


图 2-16 RPC 中的调用与消息。

(其中每一个椭圆都代表了一个进程，而阴影部分则表示存根)

当调用完成后，服务器存根获得控制权，它将结果(缓冲区)打包，然后调用 *send* 原语将消息返回客户。最后，服务器回到 *receive* 状态，等待下一条消息。

消息送回客户机后，内核按地址找到发送请求的客户进程(实际上是该客户进程的存根部分，但内核并不知道)。消息被拷贝到等待缓冲区后，客户进程解除阻塞。客户存根检查并拆开信包，取出结果，并将它拷贝到调用者进程的缓冲区中，然后以一般方式返回。当调用者在调用 *read* 后又得到了控制权，它所知道的只是得到了所需的数据，并不知道该过程的执行是在远程而不是在本地内核。

客户方忽略消息传递的细节是整个方案中最完美的部分。远程服务可以通过一般(即本地)的过程调用来访问，而不用通过调用如图 2-9 所示的 *send* 和 *receive* 原语。所有消息传递的细节都被隐藏于两个库过程中，就像在本地的库函数调用掩盖了系统中断调用的具体细节一样。这就是该机制的最主要的优点。

概括地说，RPC 的主要步骤是：

- (1) 客户过程以普通方式调用相应的客户存根；
- (2) 客户存根建立消息并激活内核陷阱；
- (3) 内核将消息发送到远程内核；
- (4) 远程内核将消息送到服务器存根；
- (5) 服务器存根取出消息中的参数后调用服务器的过程；
- (6) 服务器完成工作后将结果返回至服务器存根；
- (7) 服务器存根将它打包并激活内核陷阱；
- (8) 远程内核将消息发送至客户内核；
- (9) 客户内核将消息交给客户存根；
- (10) 客户存根从消息中取出结果返回给客户。

这些步骤最主要的作用就是将客户过程的本地调用转化为客户存根再转化为服务器

过程的本地调用，对客户与服务器来说它们的中间步骤是不可见的。

2.4.2 参数传递

客户存根的功能是获取调用的参数并将参数打包放入消息中送往服务器存根。虽然这听起来很直截了当，但实际并不像表面上那么简单。本节我们将讨论 RPC 系统中有关参数传递的几个问题。将参数打包形成消息的过程称为参数组装(parameter marshaling)。

举一个最简单的例子，我们考虑远程调用函数 $sum(i,j)$ ，该函数有两个整型参数并返回其代数和。(作为一个实际问题，因为开销问题人们不会将这么简单一个过程作为远程过程。但在这里作为一个例子是可以的。) sum 调用的参数分别为 4 和 7，如图 2-17 中客户进程左半部分中所示。客户存根获取这两个参数并将它们打包入消息中。因为一个服务器可能支持多个调用，所以它也将被调用过程的名字或过程号放入消息中，以确定是哪一个调用。

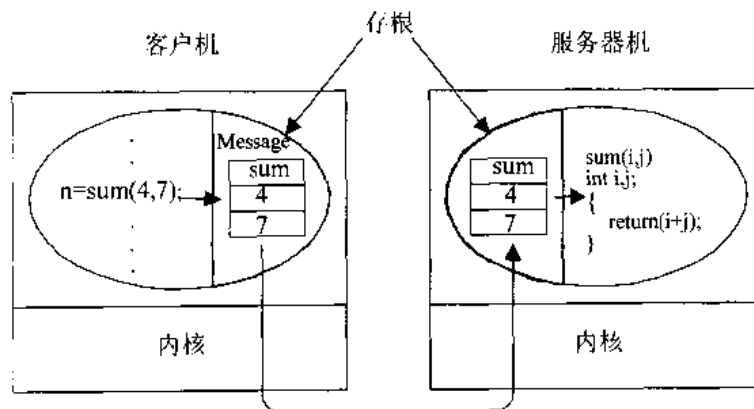


图 2-17 $sum(4, 7)$ 的远程计算

当消息到达服务器后，由存根检查消息以确定需要哪个过程，然后调用相应的进程。服务器可能还支持减、乘、除的远程过程调用，所以服务器存根中可能有一个 switch 语句，它根据消息的第一个字段选择被调用的过程。实际上，从存根到服务器的调用很像原来的客户调用，只不过参数是由到来的消息对之进行初始化的变量，而不是常量。

服务器进程一结束，服务器存根再次取得控制权，它获取服务器提供的运行结果并将其打包形成消息。这条消息被发送回客户存根，客户存根从消息中取出结果，最终将结果返回给客户进程(在图中没有显示)。

只要客户机与服务器机是同样的机器，并且参数与结果都是像整型、字符型、布尔型这样的标量类型，那么上述模型将工作良好。但在一个大型的分布式系统中通常有多种机型。各种机型又常常有自己的表示数字、字符和其他数据项的方式。例如 IBM 主机中使用的是 EBCDIC 码，而 IBM PC 中使用的是 ASCII 码。因此，如果使用图 2-17 所示的简单机制，要从 IBM PC 客户向 IBM 主机服务器传送一个字符参数是不可能的，因为服务器将会错误地解释所传送的字符。

整型数的表示(用反码还是补码表示)，尤其是浮点数的表示也会出现相似的问题。此

外, 还存在着更令人讨厌的问题。如在 Intel 486 中字节是从右向左编号, 而在其他一些系统如 SPARC 中正相反。Intel 的格式称为最低有效字节优先, 而 SPARC 的格式称为最高有效字节优先。例如, 如果一个服务器有两个参数, 一个整数和一个四个字符的字符串。每个参数占一个 32 位长的字。图 2-18(a)说明了在 Intel 486 机上一个客户存根所建消息的参数部分。第一个字包含了在这种表示方式下的整型参数 5, 第二个字包含了字符串“JILL”。

由于消息在网络上是按字节(实际上是按位)传送的, 所以先传送的字节先到达。图 2-18(b)说明了图 2-18(a)所示的消息被 SPARC 机接收后的情况。SPARC 表示数字的格式是字节 0 在最左面(高序字节), 而所有的 Intel 主板的字节 0 却在最右面(低序字节)。当服务器存根分别从地址 0 和 4 中读出参数时, 它将得到一个等于 $83886080(5 \times 2^4)$ 的整数和一个为“JILL”的字符串。

一个简单却不正确的方法是, 当参数到达后, 将每个字倒置。这就导致了图 2-18(c)所示的情况。这时整数虽然仍是 5, 但字符串却成了“LLIJ”。问题在于整型数是由于不同的字节顺序而必须颠倒, 但字符串并不是这样。因此, 如果没有额外信息来指明什么是字符串, 什么是整数的话, 这个问题是没有办法解决的。

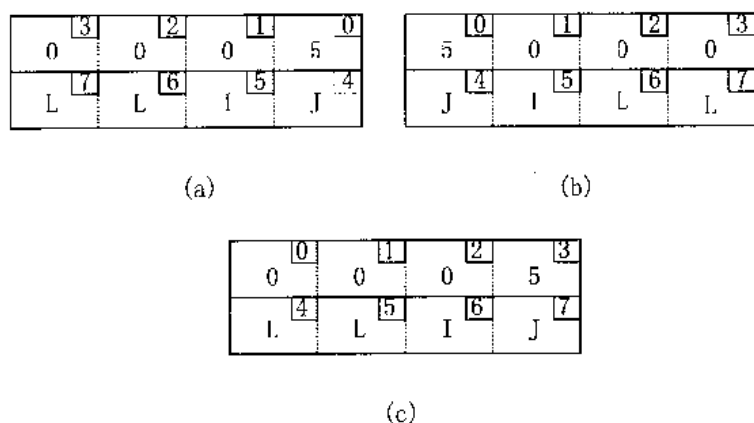


图 2-18 (a) 486 中的原始消息

(b) 在 SPARC 上接收到的消息

(c) 经过翻译之后的消息

(框中的小数字表明了每一个字节的地址)

幸运的是, 这个消息可以准确获得。记住消息中哪些项和过程标识与参数相关的。客户和服务器都知道这些参数的类型。因此, 对应于一个具有 n 个参数的远程过程调用的消息将会有 $n+1$ 个字段, 一个字段标识过程, 另外 n 个字段存放 n 个参数。一旦客户和服务器共同建立一个表示基本数据类型标准, 给定了一个参数列表和消息后, 它们就可以推断出哪些字节属于哪个参数, 这样, 问题就解决了。

作为一个简单的例子, 考虑如图 2-19(a)所示的过程。它有三个参数, 一个字符, 一个浮点数和一个有 5 个整型的数组。如图 2-19(b)所示, 我们或许决定将字符放在一个字的最右边字节中(让剩下的 3 个字节为空), 把浮点数作为整个一个字, 让该数组占用与它长度相等的字数并在数组之前加上一个字长以说明数组长度。根据上述放置参数的规则,

foobar 的客户存根知道必须使用图 2-19(b)的格式,而服务器存根也知道即将到来的消息具有图 2-19(b)的格式。有了这些参数类型的消息后,不同机器间就能够进行任何的必要转换。

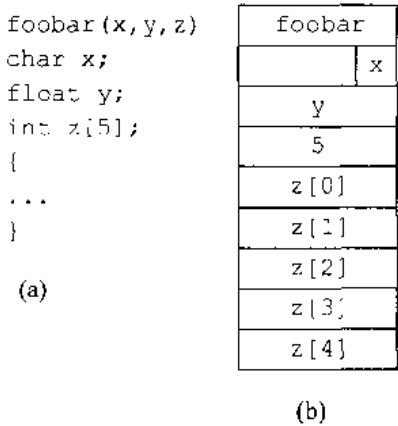


图 2-19 (a) 一个过程
(b) 相应的消息

即使有了这些附加信息后,还是存在不少问题。特别是信息在消息中是如何表示的?一种方法是针对整数、字符、布尔数以及浮点数等设计一个网络标准或规范化形式,并且要求所有的发送者在参数编组时将其所有数据的内部表示转换为符合该标准的表示。例如,可以规定补码表示整数,ASCII 码表示字符,0、1 表示布尔数,IEEE 格式表示浮点数,并且任何数都以最低有效字节优先来存储。对任何整数、字符、布尔数和浮点数的列表,即使最后一位也有确定的模式。这样,服务器存根就不用再担心客户使用的字节顺序,因为此时消息中每一位的顺序都是固定的,独立于客户硬件的。

使用这种方法有时效率太低。设想一个最高有效字节优先的客户正在与一个最高有效字节优先的服务器通信情况。按照网络标准,客户必须将消息从最高有效字节优先转换为最低有效字节优先,服务器接收到消息后又从最低有效字节优先转换为最高有效字节优先。尽管这样做是清楚的,但它需要做两次转换,而实际上是连一次转换都没有必要。这时就出现了第二种方法,客户使用自己本来的格式,并在消息的第一个字节中说明它所使用的格式。这样,最低有效字节优先的客户建立最低有效字节优先的消息,最高有效字节优先的客户也同样。消息到达服务器存根时,服务器存根检查消息的头一个字节来确认客户表示信息的格式。如果与服务器的格式相同则不需转换,若不同,则转换为与服务器的相同的格式。在涉及到反码与补码的不同表示、EBCDIC 码与 ASCII 码的不同表示时,它们之间的转换都可以采用这种方法。这个方法已经知道消息中参数是怎样布局的以及客户是采用什么格式的,剩下的转换工作就容易了(只要一方能够转换为另一方的格式)。

下面我们将讨论存根库中的过程从何而来。在许多基于 RPC 的系统中,这些过程是自动生成的。只要给定了服务器过程的形式说明和编码规则,那么也就是唯一的确定了的消息格式。这样编译器就有可能依据服务器进程的说明生成一个客户存根,这个客户存根将参数放入已定义好的消息格式中。同样,服务器的编译器也可以参照形式说明生成服务器存根,用来将消息中的参数取出,然后调用服务器进程。从服务器的一个形式说明产生

两个存根过程，使编程人员易于使用 RPC 操作，减少了错误，并且能将内部数码的不同表示隐藏起来，提高了系统的透明性。

最后要讨论最困难的问题是**指针如何传递**？指针只有在该进程使用的地址空间中才有意义。在上述例子的 *read* 操作中，假设第二个参数(缓冲区首址)的值是 1000。客户进程不能直接把 1000 传送给服务器进程，同时希望服务器进程正常工作。服务器上的地址 1000 存放的可能是程序文本的一部分。

在通常情况下一一种解决方法是禁止指针与形参的使用。然而，这两种参数非常重要，所以这样做是不可取的。事实上，也没有必要这样做。在上例中，客户存根知道第二个参数是指向一个字符型数组，假设其同时也知道该数组的大小。于是有这样一种方法：将数组拷入消息中，发送到服务器。服务器存根可以用指向这个数组的指针调用服务器过程，尽管这个指针指向的地址值可能与第二个参数的值不同。服务器过程利用该指针在服务器存根的空间对该数组进行操作(如存入数据)。完成后，再将数组拷入消息中送回客户存根，最后返回客户进程。事实上，形参调用已被复制/恢复所代替。虽然两者并不完全一样，但在一般情况下复制/恢复已经够用了。

一个优化的算法可使这个机制提高两倍的效率。如果存根程序知道缓冲区是服务器进程的输入参数还是输出参数，就可减少一次拷贝过程。对服务器而言，如果数组是作为输入参数(例如，在过程调用 *write* 中)，那么就无须将数组拷回给客户。若该数组是输出参数则无须将数组发送至服务器。客户与服务器存根程序是通过服务器的形式说明得知参数的输入/输出情况的。每一个远程过程都有一个用某种语言编写的形式说明书与之对应。形式说明书指出了参数是什么、是作为输入参数还是作为输出参数、参数的最大尺寸等。客户和服务器的存根就可以由这份形式说明书通过专门的存根编译器来生成。

最后要指出的是，尽管我们能处理一些指向简单数组和结构的指针，但这是远远不够的。因为我们仍不能处理那些更常用的大型的数据结构，如一个复杂的图形等。一些系统在解决这个问题时试图采用将实际指针传递给服务器存根，并且在服务器进程中生成一些特殊代码来使用指针。

通常，一个客户进程中的指针是放在寄存器中，通过使用寄存器间接寻址来访问的。在采用这项专门技术时，服务器进程给客户存根发回一个带有指向这个寄存器的指针的消息，并且由客户存根通过间接引用该指针去取出并发送该指针指向的数据项(读)或向所指的地址中存入一个值(写)。这种方法尽管可行，但是往往效率不高。设想一文件服务器向缓冲区写入数据时，对每个字节的写入都需要发回一条消息来完成。但是由于没有比这更好的办法，一些系统中仍然采用了这种方法。

2.4.3 动态捆绑

客户如何定位服务器呢？一种方法是将服务器的网络地址固化到客户机中。但是这种方法的适应性很差，当服务器移动、复制或者在改变其接口后，需要找出并重新编译它的许多有关程序。为避免这些麻烦，一些分布式系统采用了动态捆绑的技术，以使客户能够定位服务器。在这一节，我们将描述动态捆绑技术。

首先应该提到的是服务器的形式说明书。作为一个例子，我们来参考一下图 2-9(a)所示的服务器进程，其形式说明书如图 2-20 所示。该形式说明书指出了服务器名字叫 *file_server*，版本号为 3.1，提供的服务器过程有(*read*, *write*, *create*, *delete*)。

```
# include <header.h>

specification of file_server, version 3.1:

    long read(in char name[MAX_PATH], out char buf[BUF_SIZE],
              in long bytes, in long position);

    long write(in char name[MAX_PATH], in char buf[BUF_SIZE],
              in long bytes, in long position);

    int create(in char[MAX_PATH], in int mode);

    int delete(in char[MAX_PATH]);

end;
```

图 2-20 对图 2-9 无状态服务器的形式说明书

每一过程都给出了参数类型。每个参数都被指明为输入参数、输出参数、或者输入/输出参数。一个输入参数，文件名 *name*，是从客户进程传递给服务器进程的，它告诉服务器进程对哪个文件进行读、写、创建、删除等操作。同样，参数 *bytes* 告诉服务器进程有多少个字节要传送。参数 *position* 指明了文件从何处开始读写。输出参数，例如 *read* 中的 *buf* 用作从服务器进程传递消息给客户进程的。*buf* 是文件服务器存放客户所需数据的地方。输入/输出参数(本例中未给出)从客户进程传递给服务器进程，经服务器进程修改后返回给客户进程(复制/恢复)。复制/恢复典型地用在传送指针参数上，通过指针参数，服务器可以读取并修改该指针指向的数据结构。给出参数传递方向是很重要的，这样客户与服务器的存根就可以知道哪些参数需要发送，哪些参数需要返回。

应该指出，这里所指的服务器只是一个无状态的服务器。对于类似 UNIX 的服务器，可能还会有 *open*、*close* 等过程，并且在 *read* 与 *write* 中所用到的参数也与我们所讲的不同。同样，RPC 的概念也只是一个核心，设计者可以利用这些概念去设计所期望的服务器。

如图 2-20 所示的形式说明书的主要用途是作为存根生成器的输入，以此来产生客户和服务器的存根，然后将这两个存根存放到相应的存根库中。当客户程序调用任何由此说明定义的过程时，相应的客户存根过程将连到程序的二进制代码中。同样，服务器程序编译时也将对应的存根过程连到其 二进制代码中。

当服务器程序开始执行时，*main* 主循环外(见图 2-9(a))的初始化调用(*initialize*)输出服务器的接口。这意味着服务器进程向一个称为 *binder* 的程序发送消息，通过 *binder* 使其其他机器知道该服务器的存在。该进程被认为用于服务器的注册。在注册时要登记服务器的名字、版本号、通常有 32 位长的唯一标识符号以及用于定位的句柄，以便客户进程能寻

找到服务器进程。不同的系统句柄不同。它可能是一个以太网地址、IP 地址、X.500 地址、一个稀疏进程标识或者其他地址。只要是能辨别出服务器进程的合法地址均可作为句柄。服务器进程也可通过调用 binder 来注销登记以停止服务。binder 的接口如表 2-3 所示。

表 2-3 绑定接口

调用	输 入	输 出
注册	名字、版本、句柄、唯一 id	
注销	名字、版本、唯一 id	
查找	名字、版本	句柄、唯一 id

上面介绍了一些背景，下面让我们看看客户是如何定位服务器的。当客户进程第一次调用某个远程过程时，假设为 *read*，客户存根发现其尚未与服务器捆绑，则向 binder 发送消息要求输入名为 *file_server*、版本号为 3.1 的接口。binder 检查是否某个或多个服务器已经输出了有这样名字和版本号的接口。如果当前运行的服务器进程不支持这样的接口，则 *read* 调用失败。在匹配过程中使用版本号，binder 能够确保使用过期接口的客户进程不能定位服务器，而不会由于不正确的参数而导致的错误结果。

另一方面，如果存在合适的服务器进程，binder 将它的句柄和唯一标识符交给客户存根。客户存根把句柄作为地址，向它发送请求消息。消息中包含了送往服务器进程的参数和唯一标识符。当一台机器上运行多个服务器时，服务器内核利用唯一标识符把到来的消息发送到正确的服务器。

这种输入、输出接口的方法有很大的灵活性。例如，它可以处理多个服务器支持同一接口的情况。binder 可以根据需要随机地将客户进程分配给多个服务器进程，以使服务器负载均衡。它还可以通过周期性测试，自动注销任何不能响应调用的服务器进程，以提高容错性。此外，它还可以用来确认合法的使用者，例如，一个服务器可以用一个列表指明愿意为哪些客户的请求服务。如果用户不在此表内，binder 就不将服务器进程的句柄及进程标识传给它。binder 同时还用来确认服务器与客户是否在使用同一版本的接口。

当然，这种动态捆绑方式也有不足之处。输入和输出接口需要额外的开销。由于客户进程的生存期短，而且每个进程运行时都要从头开始，极大地影响了系统性能。另外，在一个大型的分布式系统中，binder 会成为瓶颈，因此需要多个 binder 程序。这样，无论注册还是注销一个接口，都需要有大量的消息传递来保持多个 binder 的同步与更新，这就需要更多的开销。

2.4.4 失败情况下的 RPC 语义

RPC 的目标是隐藏通信细节，使远程过程调用看起来和本地调用一样。除了一些例外，如无法处理全局变量，以及使用复制/恢复而不是变参调用传递指针参数而引起的细微差别等，现在，我们和目标已经很接近了。实际上，只要客户与服务器都正常运转，RPC 将工作良好。但是一旦出现了错误，就会产生一些问题。这时，远程过程调用与本地调用的一些差别是不容易掩盖的。本节中我们将讨论以下可能发生的一些错误及其解决方法。

为使我们的讨论结构化，让我们来区分 RPC 系统中可能出现的五类问题：

- (1) 客户无法定位服务器；

(2) 客户发给服务器的请求消息丢失;

(3) 服务器发给客户的应答消息丢失;

(4) 服务器在收到请求后崩溃;

(5) 客户机在发送请求后崩溃。

以上五类问题均各不相同, 因此需要不同的解决方法。

客户无法定位服务器

首先, 客户可能无法定位合适的服务器。例如, 服务器可能已关闭。此外, 假设客户进程已用某一版本的客户存根编译好了, 但其二进制代码已经有很长时间没有使用了。在这期间, 服务器可能已产生一个新版本的接口, 且产生了新的存根程序并投入使用。这样, 当客户进程运行时, `binder` 就无法将客户进程与服务器进程相匹配, 只能报告出错。尽管这种机制可以防止客户与参数不匹配或被认为不匹配的服务器通信, 但仍然存在如何正确处理错误的问题。

在图 2-9(a)所示的服务器中, 每个过程都有一个返回值, 值为-1 表示调用失败。对这样的过程来说, 返回值-1 可以清楚地告诉调用者调用失败。在 UNIX 系统中, 有一个全局变量 `errno`, 它具有不同的返回值说明错误类型。在这样的系统中, 加入一条新的错误类型“无法定位服务器”是很简单的。

但这种解决办法并不总是有效的。考虑图 2-17 所示的 `sum` 过程。如果参数是 7 到-8 的话, 其返回值-1 是一个合法的值。我们需要其他报告出错的机制。

一种可能的解决办法是在出错时产生一个异常。在一些语言中(例如, Ada), 编程人员可以编写在发生特定错误(如除 0 错误)时激活的特殊过程。在 C 语言中, 信号处理程序的目的就在于此。换句话说, 我们可以定义新的信号类型 `SIGNOSERVER`, 让它像其他信号一样处理错误。

这种方法也有缺陷。首先, 并不是所有语言都有异常和信号处理程序。PASCAL 就是这样一个例子。另外, 编写异常和信号处理程序破坏了我们努力想达到的透明性。假设你是一个编程人员, 如果老板让你去编写一个 `sum` 过程, 你可以轻松地完成。但她还让你去编写一个异常处理程序以防止 `sum` 过程不存在的情况。就这点而言, 很难使人相信远程过程与本地过程没有区别了。因为在单处理机系统中为“无法定位服务器”而编写异常处理程序是相当少见的。

客户请求消息丢失

第二项是处理丢失的请求消息。这是最容易解决的一个问题: 客户内核在发送请求时启动计时器。如果在计时器时满之前无应答或无确认消息返回, 内核重发消息。如果请求消息确实丢失了, 服务器是无法区分收到的请求是原来的还是重发的, 而一切运行良好。当然, 如果消息被多次重发而得不到应答, 客户会放弃请求并认为服务器已经关闭。这就是前面所说的“无法定位服务器”的情况。

服务器应答消息丢失

处理应答丢失的情况要困难一些。一个显而易见的办法是根据计时器重传。如果在合理的时间内未收到应答, 那么就重发请求。这种方法存在的问题是客户内核不知道为什么没有收到应答。是请求丢失还是应答丢失, 或是服务器速度太慢? 对不同原因引起的错误, 其处理的方法也不相同。

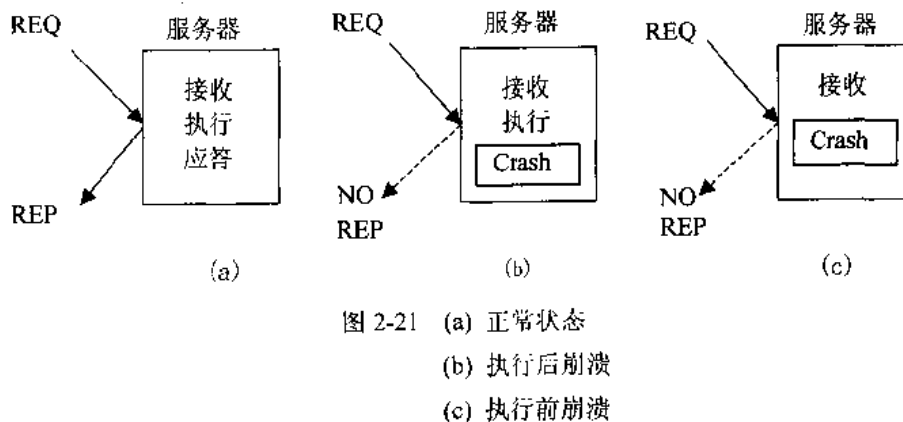
特别是有些操作能多次安全地重复执行而不产生危害。例如，从某文件读出 1024 个字节的请求就不产生负面影响，它可以按需要多次执行而不发生错误。具有这种性质的请求称为幂等的(idempotent)。

现在设想一个发到银行服务器的请求，它要求一个帐户调拨一百万美元到另一个帐户。如果请求到达且被执行，但应答丢失，客户并不知道这种情况发生，将在计时器到时后重发请求。服务器会认为这是一个新的请求，因而再次执行该请求。那么将有二百万美元调拨到那个帐户上。如果应答丢失了十次，那么后果将是多么可怕。这样的请求是非幂等的。

要解决这个问题，一种方法是将每个请求构造成为幂等的。然而，有些请求(例如，汇钱)事实上是非幂等的，因此需要其他的措施。另一种方法是，客户内核给要发送的请求消息分配一个序号。服务器内核保留那些最近来自每个客户的请求序号。这样服务器内核可以区别第一次发送的请求和重发的请求，从而排除了两次执行某个请求的可能性。一个附加的保护措施是在消息头上增加一位以区分是原来的还是重发的消息，能提醒服务器在看到不是原发消息时谨慎处理。

服务器崩溃

服务器崩溃也与可幂等次执行有关，但现在不能使用给消息加序号的方法来解决。图 2-21(a)是一服务器进程正常工作处理一个事件的顺序；一个来自客户的请求到达，执行该请求，服务器进程返回一个应答。如图 2-21(b)所示是一个客户请求到达后，服务器进程执行，在返回应答之前服务器崩溃。最后，如图 2-21(c)所示，一个请求到达后，在执行前服务器崩溃。



图中(b)、(c)的处理方法是不相同的。在(b)中，系统不得不向客户报告失败(例如，引起一个异常中断)，而在(c)中只需重发请求。问题是客户的内核不能区分这两种情况。它只知道计时器到时。

有三种方法可以解决这个问题。

第一种方法是等待服务器重新启动(或与另一个新服务器捆绑)，然后重发请求。这种方法要求不断重试直至应答消息到来并传给客户。这种技术称作至少一次语义(at least once semantics),它保证 RPC 至少要执行一次，但也有可能执行多次。

第二种方法是立即放弃并报告失败。这是最多一次语义(at most once semantics)，它确

保了 RPC 最多执行一次,但可能没有执行。

第三种方法不作任何保证。当服务器崩溃时,客户得不到任何帮助和保证。RPC 可以不被执行或执行相当多次。这种方法最大的优点就是易实现。

这三种方法都不是成熟的方法。人们需要的是精确的执行一次的语义(exactly once semantics),但通常它是不容易实现的。设想一个远程操作包括打印一些文本,它将文件调入打印机的缓冲区,然后在某个控制寄存器中设置一位后准备开始打印。可能在置位的那一微秒的前后服务器崩溃而无法打印。客户由于无法知道崩溃发生在置位前还是置位后,因此也就无法有针对性地进行恢复处理。

简而言之,服务器崩溃在很大程度上改变了 RPC 的性质。单处理机系统和多处理机系统发生服务器崩溃的情况是截然不同的。在单处理机系统中,服务器的崩溃往往意味着客户的崩溃。所以恢复既不可能也没必要。而在分布式系统中则可以采取一些措施来处理这种情况。

客户机崩溃

如果客户已发出请求但在应答到来之前崩溃了,这时会发生什么?此时已经激活了服务器中的相应计算,但没有客户在等待结果。这样的计算称为孤儿(orphan)。

孤儿会导致一系列的问题。起码它浪费了 CPU 周期,同时又锁住了文件或其他宝贵的资源。此外,如果客户重新启动并再次调用了这个 RPC,客户会很快得到那个孤儿的返回值,这将引起调用结果的混淆。

为解决孤儿问题,Nelson(1981)提出了四种解决方法。

方法一,在客户存根发送一个 RPC 前,在日志文件中记下要执行操作的信息。该日志文件保存在不受崩溃影响的磁盘和其他媒介上。当客户重新启动后,系统检查日志文件,并准确地清除孤儿。这种方法称为根绝(extermiation)。

根绝方法的缺陷是对每个 RPC 都要进行磁盘记录,极大增加了系统开销。此外,这种方法也可能不起作用,因为孤儿还可以执行 RPC,这样又生成了一层或更多层的子 RPC,这些子 RPC 很难找到和清除。最后,由于网络或许是分段的,如果网关失败,即便找到这些孤儿也无法清除。总之,这种方法是没前途的。

方法二,称为再生(reincarnation)。这种方法不必做磁盘记录。该方法将时间划分成顺序编号的纪元(epochs)。当一个客户重新启动时,它向所有机器广播一个新纪元的开始。广播后,所有远程计算被终止。当然,如果网络是分段的,有些孤儿还会遗留下来。但是当这些孤儿的应答返回时,返回的消息上带有它们过时的纪元号。这些应答还是容易识别和清除的。

方法三,是对第二种方法的改进,称作温和再生(gentle reincarnation)。当接到某客户开始新纪元的广播后,每台机器检查自己是否有远程计算,若有则试图去找到该远程计算的调用者。若没有找到该计算的调用者,则终止该计算。

方法四,这种方法称为过期(expiration)。每一个 RPC 执行时事先分给一个标准时间段 T 。当 T 到期而调用未完成时就必需再申请一个 T 。这样做确实麻烦。另一方面,如果客户崩溃,服务器在客户重新启动前等候了一个 T 后,所有孤儿都被清除。由于 RPC 有各种不同的请求,如何选择 T 的合适值呢?

实际上,这些方法都不尽人意。终止一个孤儿可能会造成不可预见的后果。例如,

假设一个孤儿正锁住一个或多个文件或数据记录等，如果突然清除该孤儿，那么这些资源可能会一直处于被占用的状态。另外，一个孤儿可能已在某些远程的进程调用队列中等待，期望将来能调用其他进程。这样，即使去除了这个孤儿也不能去除孤儿遗留下的轨迹。Panzieri 和 Shrivastava(1988)详细讨论了孤儿清除问题。

2.4.5 实现的问题

一个分布式系统的成败往往取决于它的性能。系统的性能在很大程度上取决于通信的速度。而通信的速度主要取决于对它的实现方法而不是抽象原理。这一节主要讨论 RPC 系统的实现问题，重点在于系统性能和耗时情况。

RPC 协议族

第一个问题是如何选择 RPC 的协议。从理论上讲，任何已经存在的能够实现从客户内核到服务器内核之间按位传送的协议都可以作为 RPC 的协议。但是我们还需要确定几个重要的问题，这些问题将会对性能有很大影响。首先是用面向连接还是面向非连接的协议。面向连接的协议是，任何时候客户进程都与服务器进程捆绑，它们之间建立了一个连接。两个方向的通信都使用了这个连接。

该协议的优点是通信简单。当内核发送消息时，它不用担心消息是否会丢失，而且也不用处理确认消息。这些都在协议低层通过软件实现了。它的可靠性高，所以对广域网来说是很合适的。

其缺点是性能下降，尤其表现在 LAN 上。额外的软件开销是性能下降的原因。此外，尽管面向连接不会丢失信包，但是 LAN 的可靠性高，所以没有必要。因此，大多数校园网或者在同一建筑物内的分布式系统均采用面向非连接的协议。

第二个问题是选择一个标准的通用协议还是用专门为 RPC 设计的协议。由于 RPC 的协议尚无标准，这就意味着需要自己去设计(或借用他人的)。系统设计人员对这两种选择的支持基本上是一半对一半。

一些分布式系统使用 IP(或 UDP，它建立在 IP 之上)作为基本协议，以下几点促使人们选择了这两个协议：

- (1) 协议已存在，省去了大量的工作；
- (2) 该协议已有许多个现成的工具，这也省去了许多工作；
- (3) 绝大部分 UNIX 系统中都可发送和接收该信包；
- (4) 大多数现有网络支持 IP 与 UDP 信包。

总之，IP 与 UDP 适用于大多数现有的 UNIX 系统和网络系统(如 Internet)。这样，编写在 UNIX 系统上运行的客户或者服务器程序较容易，这样加快了代码运行和测试的速度。

IP 与 UDP 在性能方面也有缺陷。IP 不是为最终使用者设计的协议。它被设计为一个基础，其上可以建立在不同内部网络的可靠的 TCP 连接。例如，它能协助网关将信包拆分，并以网络允许的最大尺寸在网上传送。尽管基于 LAN 的分布式系统并不需要这个特性，但这些信包消息仍然由发送者填入 IP 信包的头字段中，然后由接收者重组形成一个合法的 IP 信包。IP 信包共有 13 个头字段，但是仅有发送地与目的地的地址和信包长度三个字段是有用的。其他 10 个字段只是随着发送。这 10 个字段中有一个字段是头的校验

和，计算校验和将很费时。UDP 协议的情况更糟，它的头字段里有两个字段是校验和。

另一种方法是使用一个专门适用于 RPC 的协议，它不像 IP 协议那样去处理信包，这些信包在网络中传送几分钟后，会突然在某个繁忙的时候到达。当然这样的协议需要设计、实现、测试并嵌入到现有系统中，这需要大量的工作。此外，人们也不会对又一个新协议的诞生而欢呼雀跃。从长远的观点来看开发和广泛地接受一个高性能的 RPC 协议是可取的。但是我们尚未开始此项工作。

最后一个与协议相关的问题是信包和报文的长度。RPC 有一个大而固定的独立于发送数据的头信息。所以，发送一个 64K 长的文件，一次发送 64K 明显要比发送 64 次 1K 要高效得多。因此，协议和网络能否允许较大长度的信包传送是很重要的。有些 RPC 系统只能传送小尺寸的信息(例如：Sun Microsystem 是 8K)。此外，许多网络也不能处理很大的信包(例如：以太网的最大限制为 1536bytes)，所以一个 RPC 不得不拆分成多个信包，从而增加了开销。

确认

如上所述，一个大的 RPC 需要被拆分成多个信包，于是产生了一个新的问题，即是否每个信包都要确认。例如一个客户要向文件服务器写入一个 4K 的数据块，但是系统能处理的信包最大为 1K。一种策略称为停等协议(stope-and-wait protocol)，指的是客户进程先发送 1K 长的信包 0 后，等待服务器的确认，如图 2-22(b)所示。然后，客户再发送下一个 1K 长的信包，等待下一个确认，以此类推。

另一种策略叫爆发协议(blast protocol)。是由客户进程尽快将所有的信包发送完，当所有信包到达服务器后，服务器发回确认消息，而不是收到一个信包就发回一个确认。如图 2-22(c)所示。

这两种协议的性质有很大的不同。在停等协议中，如果一个信包被破坏或丢失，客户不可能按时收到确认，于是重新发送该信包。在爆发协议中，如果信包 1 丢失而信包 2 随后正确到达，服务器会面临选择。服务器可以丢弃收到的信包，等待客户计时器超时后重发整条消息。或者是可将 0、2 这两个正确到达的信包放入缓冲区，等待信包 3 正确到达后，向客户指明重新发送信包 1，这种技术叫做有选择重发。

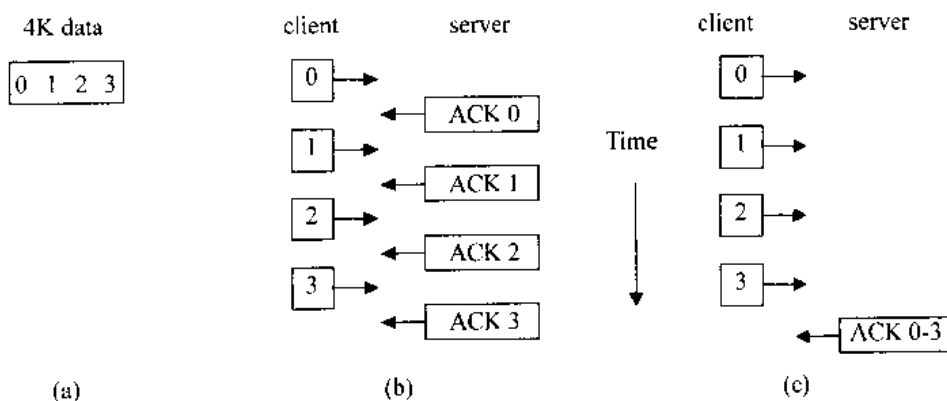


图 2-22 (a) 一个 4K 的消息
(b) 停等协议
(c) 爆发协议

停等协议和丢弃所有信包在发生错误时都易于实现。有选择重发策略需要大量的管理工作，但使用了较少网络带宽。由于一般可靠性高的局域网，几乎不会有信包的丢失，因此使用有选择重发得不偿失。对于广域网而言，使用有选择重发更为合适。

除了对出错的控制外，另一个需要考虑的问题是流的控制。许多网络接口芯片可以将连续的信包几乎无间隙地发送出去，但由于缓冲区有限而不能无限地接收一个又一个的信包。在有些设计中，接口甚至不能连续接收两个信包。因为它接收到一个信包后便产生中断，在中断期间不能接收信包，这样就会丢失第二个信包的开始部分。当接收者不能接收到来的信包时就出现了一个称作超限的错误(overrun error)，该信包随之丢失。实际上，这种错误比线路中由杂音或其他损害而造成的信包丢失更为严重。

对于超限错误，图 2-22 所示的两种方法是不同的。以停等方式发送不会出现超限错误。因为直到接收者明确表示准备好时才发送第二个信包。(当然，如果出现多个发送者，这个协议仍可能出现超限错误。)

在爆发协议中，它明显比停等协议高效得多，但也可能会出现超限错误。当然，这里也有解决超限错误的办法。一方面，如果这个超限错误是由于芯片处理中断而暂时无法接收信包而引起的，那么，发送者可在发送两个信包之间加入一段延迟时间，使得芯片能在这个时间间隔内从接收中断返回到接收状态。如果需要的延迟较短，发送者可以空循环(忙等待)。如果延时过长，可以设定一个计时中断，然后在等待时间里作其他的工作。如果延迟的时间既不长也不短，那么还是等待为好。

另一方面，若超限错误是由于芯片的缓冲区不够而引起的，那么如果缓冲区容量为 n 个信包，则可在发送 n 个信包后留一个间隙，或是在发出 n 个信包后，得到一个确认后再发送后面的信包。

应当阐明的是，减少确认信包并提高性能可能与网络芯片的计时性能有关，所以协议可能不得不求助于所使用的硬件。和通用的协议相比，一个用户自定义的 RPC 协议更容易将类似流控制的问题考虑进去。这就是为什么集中的 RPC 调用比基于 IP 和 UDP 的系统有更高的性能的原因。

在结束确认这个主题之前，还有一个问题值得考虑。在图 2-14(c)中，协议包括请求、应答、确认。确认用来告诉服务器应答已经安全到达客户，可以将之丢弃了。现在假设确认在传送过程中丢失了，那么服务器会保留这个应答，但是对客户来说这个协议已经完成，不再计时和等待信包了。

我们可以修改协议，让确认消息再被确认一次，这种做法增加了复杂性和额外开销，得不偿失。实际上，只需在服务器上设置一个计时器，在应答发送后计时，计时到后，不论客户是否收到应答，服务器都将应答丢弃，或者在收到一确认后就将应答丢弃。当然，若收到客户的新的请求时，也可以认为应答是到达了。

关键路径

由于 RPC 代码对系统性能来说至关重要，下面我们将详细研究一下客户在远程服务器上执行 RPC 的过程。每个 RPC 执行的一系列指令顺序称为关键路径(critical path)，如图 2-23 所示。客户首先调用客户存根，接着激活客户内核陷阱中断，发送消息到服务器内核，同时引起服务器内核中断，最后消息经服务器存根送至服务器进程。它执行操作并返回结果。

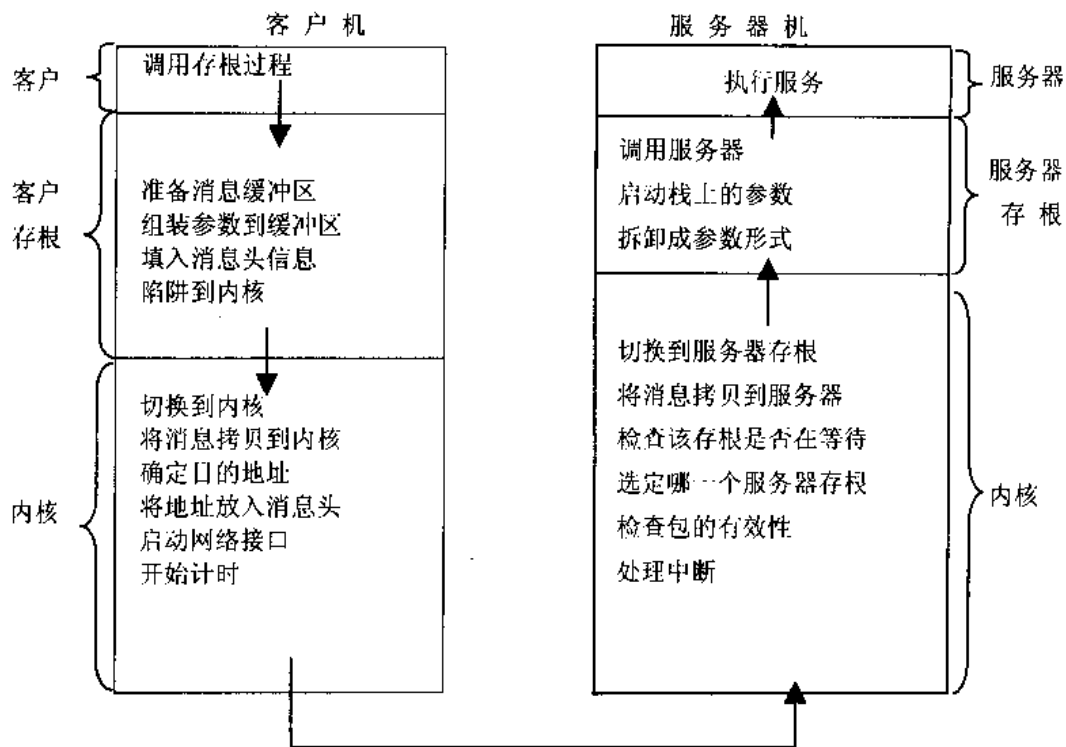


图 2-23 从客户机到服务器的关键路径

让我们来更仔细地看一下这些步骤：调用客户存根后，首要任务是获得缓冲区以便装配要发送的消息。在一些系统中的客户存根仅有一个固定的缓冲区，每次调用时要重新填写它。有些系统中提供缓冲池，其中部分缓冲区已被使用，这部分缓冲区对应于不同的服务器请求。这样在装配消息时可以节省一些工作。这种方法适用于基本信包格式中必需填写大量字段，而且在不同的调用中是一样的情况。

然后，将参数转换为适当的格式并填入消息的缓冲区中，同时填好剩余的头字段。这样要发送的消息就形成了，客户存根向内核发出中断。

当内核获得控制权时，它切换现场，保存 CPU 寄存器的值和内存映射(保护现场)，并且设置一个新的内存映射，以便在核心状态下运行。由于内核与用户区通常不相交。内核必须将消息拷贝到自己的地址空间后才能访问它。然后，客户内核将目的地址(可能还有其他一些头字段)写入消息，将消息拆分成信包拷贝到网络接口。这时，客户的关键路径已经完成，消息将在网络上传输。消息在网上传送的时间不计入 RPC 时间内，内核无法影响消息在网上的传输时间。客户的内核可以设置一个到期重发的时钟，然后开始其他工作，也可以进入阻塞状态等待应答的到来。这里需说明，阻塞等待应答可以加快对对应答消息的处理，但不能同时有效地运行多道并行程序。

在服务器方，网络接口将到来消息的每一位放到接口芯片的缓冲区或内存。等到消息完全到达后，接口产生中断，中断处理程序将检查该信包是否有效，并且决定将它送往哪一个服务器存根。如果等待接收此消息的服务器存根不存在，则此处理程序或将该信包放入缓冲区，或丢弃它。若有服务器存根等待接收该消息，则将消息拷贝到存根的缓冲区。

最后切换现场，将当前寄存器和内存映射的值恢复成存根调用 *receive* 时的值。

此后，该服务器就可以重新启动。它解析所有的参数并设置服务器进程运行时所需的环境，然后调用服务器进程。结果返回的路线和来时的相似，但方法不同。

实际所关心的一个问题是“在关键路径中的什么地方耗时最多？”，如果知道问题出在哪里，就可以想办法减少时耗。Schroeder 与 Burrows(1990)对 RPC 的关键路径进行了详尽的分析，该分析是基于 DEC Firefly 多处理工作站的。分析结果如图 2-24 所示，图中有 14 个立方柱，每个立方柱对应着客户到服务器的关键路径(没有反方向的关键路径，其大致与此相似)的每一步。图 2-24(a)给出了无数据的 RPC 的示意图，图 2-24(b)给出了以 1440 字节的数组作为参数的 RPC 的示意图。尽管这两种情况都有固定的头信息，但在第二种情况中大量的时间花在参数编组和传送消息上了。

在无数据 RPC 中，耗时主要在信包到来时服务器内核的现场转换、服务器的中断例程以及将信包传送到接口。在有数据 RPC 中，示意图变化了很多。在这里，数据在以太网上传送时耗时最多，其次是将信包送入和移出内部接口的耗时。

尽管图 2-24 说明了主要在哪里耗时，但在解释这些数据前需要指出几点。首先，Firefly 是一个有五个 VAX CPU 的多处理机。当对使用单 CPU 的机器进行测量时，RPC 的时间增加了一倍，这说明在 Firefly 具有潜在的并行处理，而大多数其他机器上是不存在并行处理的。

其次，Firefly 使用 UDP 协议，它有一个 UDP 缓冲池，因此，在 Firefly 上运行的客户存根时不必每次填充全部的 UDP 头。

再次，该机器的内核与用户共享同一地址空间，不需要现场转换，也不需要在内核与用户的地址空间之间拷贝有关信息，这样节省了不少时间。页表保护位防止用户读、写内核的某些部分，共享缓冲区和其他部分是可供用户访问的。这个设计非常巧妙地利用了 VAX 体系结构的特性，即便于内核空间 and 用户空间之间进行共享，但是该技术尚未应用在其他机器上。

最后，整个 RPC 系统是用汇编语言编码，而且也是通过手工进行优化的。这点可能就是图 2-24 中各部分的状况比较统一的原因。无疑在第一次测量时，图形曲线会很陡，研究人员将花费大量的时间以使其耗时最多的部分不再那么突出。

Schroeder 与 Burrows 根据他们的经验给未来的设计者提出了一些忠告。首先，他们建议避免使用离奇的硬件(例如，Firefly 的五个处理器中只有一个能够访问以太网，所以，要发送的信包必须先拷贝到那个处理机，接收到达的信包也是如此)。其次，他们对使用了 UDP 协议表示遗憾。他们认为 UDP 头字段中的校验和很耗时，有些得不偿失，因此最好不要使用 UDP 协议。一个简单的用户自定义的 RPC 协议将会工作得更好一些。最后，让服务器存根用忙等待而不是睡眠将大大减少 2-24(a)中耗时最多的那一步(第 13 步)的操作时间。

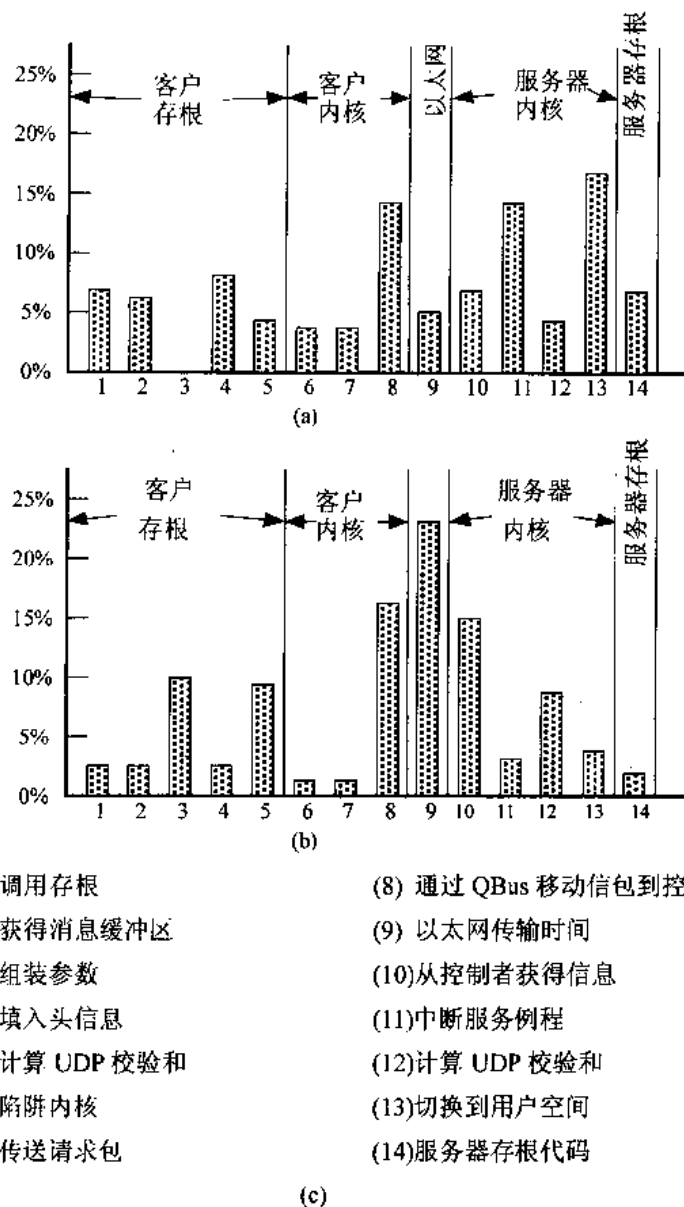


图 2-24 RPC 关键路径的崩溃

(a) 空的 RPC

(b) 共有 1440 字节数组作为参数的 RPC

(c) 在 RPC 中从客户到服务器的 14 步

拷贝(copying)

一个与 RPC 执行时间密切相关的问题是拷贝。在 Firefly 系统中由于缓冲区可以同时映射到内核和用户的地址空间，这个问题并不突出。在大多数其他系统中，内核与用户的地址空间不相交。根据硬件、软件和调用类型的不同，消息可能需要拷贝一次到八次不等。在最好的情况下，有的网络接口芯片支持直接存储器存取(DMA)，它将消息直接从客户存根取到网上(拷贝 1)，并且实时地存放到服务器内核的存储区(即当消息的最后一位以 DMA

方式从客户存根的存储器中取出后,在几个微秒之内就激活信包到达的中断)。内核检查该信包,并且包含它的页映射到服务器进程的地址空间。如果不允许做该映射,那么内核将信包拷贝到服务器存根(拷贝 2)。

在最坏的情况下,或是因为不便于从用户空间直接传送到网络,或是因为网络忙,内核将消息从客户存根拷贝到内核缓冲区(拷贝 1)。内核拷贝消息到接口芯片的硬件缓冲区(拷贝 2)。这时,硬件启动,消息经网络传送到服务器接口芯片的缓冲区(拷贝 3)。有可能内核只有从硬件缓冲区中取出并检查该消息后,才知道要将消息发送到哪个服务器存根。因此,当服务器中信包到达的中断被激活时,服务器内核将消息拷贝到内核缓冲区(拷贝 4)。最后,消息从内核的缓冲区传送到服务器存根(拷贝 5)。此外,如果此调用含有一个大的值参数组时,那么还需另外三步拷贝:将数组拷入客户进程的堆栈以调用客户存根;客户存根组装 (marshaling)时,将数组从堆栈拷贝到消息缓冲区中;服务器存根将数组从消息中取出,放入服务器进程的堆栈中。包括这三次拷贝,总共需要八次拷贝。

假设拷贝一个 32 位的字要 500ns,那么拷贝 8 次需要 4ms,这样不管网络自身的传输速度有多快,最大的传输速率也不过是 1Mbyte/s。在实际使用时,有这样速度的十分之一也已算是不错了。

一种有助于消除不必要拷贝的硬件特性称作分散-集中(scatter-gather)。具有分散-集中功能的网络接口芯片能够通过连结两个或多个内存缓冲区来装配一个信包。这种方法的优点是可以在内核空间建立信包的头字段,用户数据仍放在客户存根中。要发送信包时,该部件将这两个部分连接起来形成信包。在发送方,从多个信包源集成一个信包避免了拷贝。同样,在接收方,可将消息头和消息体分别分散存放到不同的缓冲区中。

通常,在发送方比在接收方减少拷贝更容易些。有了这个特性,在发送方,将内核中的一个可重复使用的信包头和用户空间中的一个数据缓冲区放到网上时不再需要内部的拷贝。然而,在接收方,接口芯片就很难将消息传送给服务器,因为它无法知道应该交给哪个服务器进程来处理。所以只能将消息放入服务器内核的缓冲区,由内核来处理。

使用虚拟存储器的操作系统中,有一种策略可避免将消息拷贝到存根。假设内核的信包缓冲区大小正好为一页并从一页的头开始,而且服务器存根的接收缓冲区也正好占一页整,也正好从一页的头开始。这时,内核可以改变内存映射,将内核中放置信包的缓冲区映射到服务器进程的地址空间。同样,也可以将服务器存根的缓冲区映射到内核的缓冲区。当服务器存根开始运行时,信包就出现在它的缓冲区内,免去了从服务器内核到存根的拷贝。

另外,如果缓冲区能将头字段调整到上一页的最后,而这一页只有数据,这时该页映射的全部是有用的数据。虽然这个方法清楚简便,但对每个缓冲区来说要使用两页,一页放数据;而另一页中只放了几个字节头信息,几乎是空的。

最后,许多信包仅有几百个字节,在这种情况下,就很难说映射是否比拷贝更有效。当然,这个想法值得进一步探讨。

计时管理

所有的协议都包括在通信介质上交换消息的功能。在实际系统中,存在着消息由于线路中的杂音或是超限错误而偶然丢失的情况。因此,在消息已发送而等候回答(应答或是确认)时,许多协议设置了计时器。若计时期满却没有应答,就需多次重发消息,或在

多次重发后放弃发送。

管理计时所需要的时间是不可低估的。设置一个计时器需要建立一个数据结构，指出何时计时到期并且怎样处理。然后，将该数据结构插入到未到时的计时器的链表中。一般此链表是按时间顺序排序的，离到期时间最近的排在链表的首部，离到期时间最远的排在链表尾部。如图 2-25 所示。

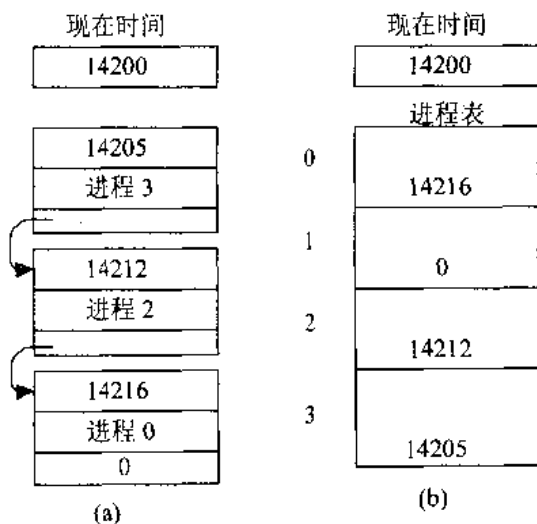


图 2-25 (a) 排序表中的时间片耗尽
(b) 进程表中的时间片耗尽

当一个应答或确认消息到达时，必须找到计时链表的入口并将该数据结构从链表中移走。实际上，许多应答并不超时，因此，查找链表入口并将该数据结构移走的大部分工作是浪费了。此外，计时也无须特别精确，计时初值的设置也只是一个大概猜测的值(“几秒钟看上去差不多”)。使用一个猜测的值不会影响协议的正确性，只是影响其性能而已。值小了会导致经常超时，造成不必要的重发。值大了，会导致在应答消息丢失后长时间的等待。

总结以上两点，可以提出一个新的更有效的计时方法。许多系统都有一个进程表，该表中每一项都包括了系统中每个进程的所有信息。当 RPC 开始执行时，内核中用一个局部指针指向当前进程的表项。我们可以将计时值放入该进程对应的表项的一个字段中而不是排序的链表中，如图 2-25(b)所示。这样设置的 RPC 计时器包括在当前时间上加上计时的长度，并且将该值存入进程表中。如果要停止某个计时的话，则在计时字段中存入 0。这样的设置，在清除计时时可减少许多操作。

欲使这个方法付诸实施，内核可以周期地(比如，每秒一次)检查整个进程表，如果发现一个非 0 的计时值小于或等于当前的时间，则说明超时了。在采取相应的操作后，重置该计时器。例如，有一个系统发送 100 个信包 / 秒，每秒扫描一次进程表的工作仅仅是查找和更新上述链表(每秒 200 次)的 1/200。这种周期按序遍历一个表的算法叫扫描算法(sweep algorithms)。

2.4.6 问题领域

基于客户-服务器模型的 RPC 广泛应用于分布式操作系统。比起纯粹的消息传递, 这个简单的抽象使得对分布式系统内部复杂性的处理更容易些。然而, 仍有一些问题需要解决, 本节将讨论需要解决的几个问题。

理想的 RPC 是透明的, 编程人员进行过程调用时不必知道这个过程是本地的还是远程的。他也应该能够在编写过程时不用考虑该过程将放在本地还是放在远程执行。更严格地说, 若某过程是建立在早先单 CPU 系统上的, 那么, 它不应该采用一套新的规则来禁止原先合法的操作, 也不能对原来可选择的操作加以限制。在这样的定义下, 现行的分布式系统几乎没有一个称得上是透明的。透明性的研究仍需不断地进行下去。

作为例子, 我们来看看全局变量的问题。在单 CPU 系统中, 甚至在库函数中全局变量都是合法的。例如 UNIX 系统中的全局变量 *errno*。在一个系统调用出错后, *errno* 会返回一个值, 报告出错类型。由于 UNIX 标准 POSIX 要求 *errno* 在一个必备的头文件 *errno.h* 中可见, *errno* 成为了一个公共信息。因此, 应用中要求该变量对编程人员是可见的。

现在考虑一个编程人员编制了两个直接访问 *errno* 的过程。一个在本地, 另一个在远程运行。由于编译器不知道哪个变量和过程是本地的, 哪个是远程的, 不论 *errno* 存放在哪里, 总有一个过程对它的访问失败。问题在于, 允许本地过程不受限制地访问远程的全局变量, 而且反之亦然, 是不能够实现的。这种对访问的禁止违反了透明性的原则。

另一个问题是有些语言结构不严谨, 如 C 语言。像 PASCAL 这样结构严谨的语言, 编译器和存根对参数类型、大小等了解得很清楚, 这使存根很容易完成参数的组装工作。而在结构不严谨的 C 语言中, 编写一个过程计算两个向量(数组)的内部值, 而并不指定数组的大小是合法的。数组的大小可能在调用或被调用的程序中给出。这样客户存根根本无法往消息中写入参数, 因为存根不知道作为参数的数组有多大。

通常的解决方法是编程人员在编写服务器的正式说明时给数组设定一个最大值。如果编程人员希望过程能处理任意大小的数组, 结果会怎样? 他有可能设置一个很大的值, 比如说一百万。这就意味着即使数组的实际大小只有 100 个元素, 客户存根也得传送一百万个元素。此外, 如果数组有 1000001 个元素而内存只能容纳 200000 个元素, 该调用就失败了。

将指向复杂图形的指针作为参数也是一个难点。在单 CPU 系统中, 这类指针的传送是常有的事。但是若要在 RPC 时传送此类指针, 客户存根很难找到整个图形。

存在的另一个问题是存根并不总能推断出参数的类型, 甚至在有形式说明书和代码的情况下有时也无从推断。例如, C 语言中的函数 *printf*, *printf* 中有多个参数(至少一个), 它们可以是整型、长整型、短整型、字符型、字符串型和长度不等的浮点型等。如果要远程调用 *printf*, 从实现上讲是不可能的, 因为 C 的语义非常松散。我们可以在 RPC 时不使用 C 语言, 而这又可能破坏了 RPC 的透明性。

以上讨论的问题都和透明性有关, 但这里还有一个更基本的问题。考虑如下 UNIX 命令的执行:

```
sort <f1 >f2
```

由于 *sort* 可以读标准输入、写标准输出, 它可以作为输入和输出的客户, 执行从文

件服务器读取 $f1$ ，向文件服务器写入 $f2$ 的 RPC 操作。

```
grep rat<f3 >f4
```

grep 作为客户从文件 $f3$ 中读出含有 *rat* 内容的行，并写入到文件 $f4$ 中。

现在考虑 UNIX 管道：

```
grep rat<f5 | sort >f6
```

如上述所示，*grep* 和 *sort* 都可以作为标准输入和标准输出的客户。编译到代码中的该管道可以使上两例运行。但是它们会怎样交互工作呢？是 *grep* 作为客户向服务器 *sort* 写入呢，还是 *sort* 作为客户从服务器 *grep* 读出？在这里 *grep* 与 *sort* 其中一个必须作为服务器(被动的)。但是在前两条指令中，*grep* 与 *sort* 均作为客户(主动的)使用，在这里客户/服务器模型显然是不合适的。

在管道中也有这样的问题，例如：

```
p1<f1 | p2 | p3>f2
```

避免以上所示客户-客户接口的一种方法是使管道作为读驱动的，如图 2-26(b) 所示。程序 $p1$ 作为客户(主动)请求从文件服务器中读取 $f1$ ，程序 $p2$ 作为客户请求从文件服务器中读取 $p1$ 和程序 $p3$ 读取请求 $p2$ 。到此为止，一切运行良好。问题是谁作为客户从 $p3$ 中读取最后输出呢？这时读驱动的管道不能正常工作。

在图 2-26(c) 我们将管道作为写驱动的，它存在镜像的问题。 $p1$ 作为客户向 $p2$ 写入， $p2$ 作为客户向 $p3$ 写入， $p3$ 作为客户向文件服务器写入，但是谁作为客户要求 $p1$ 接收输入文件呢？

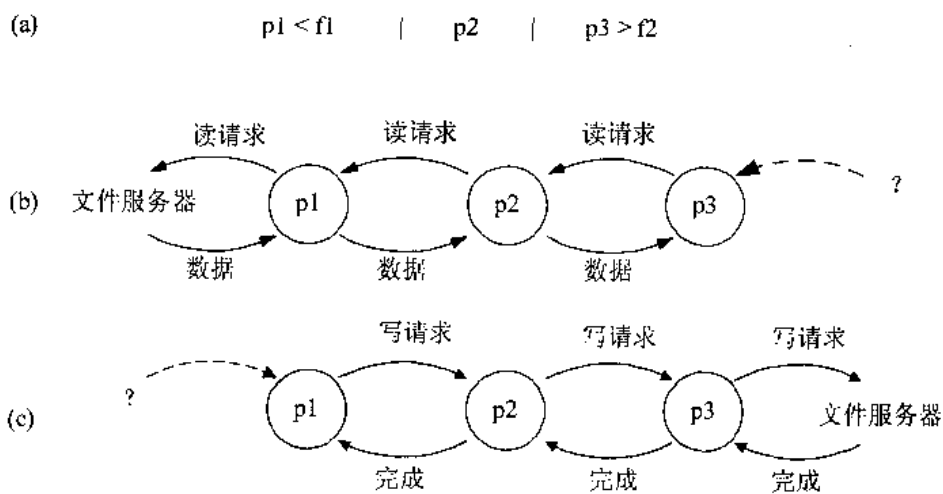


图 2-26 (a) 管道

(b) 读驱动方法

(c) 写驱动方法

由此说明，基于客户/服务器模型的 RPC 并不适合这种通信模式。一个可能解决此问题的方法是将管道作为双向的服务器，既响应左边的写请求，又响应右边的读请求。另外，管道也可作为临时文件来实现，该文件只能从文件服务器里读出或只能写入文件服务器。

但这样又需要不必要的开销。

当 shell 外壳程序要从用户获取输入数据时也出现了相似的问题。通常它发送读请求到终端服务器，该终端服务器收集键入的数据，等候 shell 的调用。但是当用户键入了中断键(DEL、CTRL-C、break 等)时怎么办？如果终端服务器仍是将键入的中断键放入缓冲区并等待 shell 的调用，那么用户就无法从现行的程序中退出。另一方面，如果终端服务器作为客户向 shell 发送 RPC，而 shell 不希望用作服务器，这时情况又如何？显然，这就像在使用管道时出现的客户与服务器概念混淆的情况一样。事实上，只要发送一个意料之外的消息，就会出现潜在的问题。所以，客户/服务器模型只是在一般的情况下运行良好，但是它并不适合所有的情况。

2.5 组通信

RPC 的一个固有的假设通信方式只包括两个成员，即客户与服务器。有时存在这样的情况：通信涉及到多个进程，而不是仅仅两个进程。例如：一组文件服务器合作提供一个单一的、可容错的文件服务。在这样的系统中，客户可能希望向所有服务器发送消息，以确保即使其中一个服务器崩溃仍能执行该请求。除了利用单个服务器执行单独的 RPC 外，RPC 不能处理一个发送者给多个接收者发送消息的通信方式。本节将讨论一种在一次操作中将一条消息发送给多个接收者的通信机制。

2.5.1 组通信的引入(Introduction to Group Communication)

一个组是指在某系统或用户指定方式下协同工作的多个进程的集合。任何组都具有的一个主要的性质是：当一条消息发送到该组后，组内的所有成员都能收到该消息。这就是一对多通信(一个发送者，多个接收者)的形式。它和点对点通信是有区别的。如图 2-27 所示。

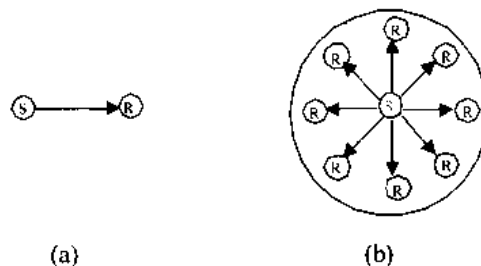


图 2-27 (a) 从一个发送者到一个接收者的点对点通信
(b) 从一个发送者到多个接收者的一对多点通信

组是动态的。可创建新的组，也可注销旧的组。一个进程可以加入或离开一个组。它也可以同时成为多个组的成员。因此，需要一种机制来管理组和组的成员。

组和社会上的组织有些类似。一个人可以是一个读书俱乐部、一个网球俱乐部和—个环境组织的成员。在某个特殊的日子，他可能会收到读书俱乐部的介绍一本新生日蛋糕食谱的信函，网球俱乐部母亲节网球赛的信函和环境组织开始一场保护南部大草原运动的信函。在任何时刻，他可以自由地离开一个或全部组织，也可以加入其他的组织。

尽管在这本书里我们学习的只是操作系统的组(即进程组),但值得一提的是,在计算机系统中经常会碰到其他的组。例如,在 USENET 计算机网上,遍布着成百个新闻组,每个组都有一个专门的主题。当有人给某个特定的新闻组发布消息时,该组的所有成员都能收到它,即使存在成千上万个组成员。通常,这些高层组关于谁是组的成员,消息传送的精确语义等的规则比操作系统组的规则更松散。在大多数情况下,这种松散并不是一个问题。

我们引入组的目的是让进程将多个进程的集合当作一个独立的抽象物处理。这样,进程给服务器组发送消息时无须知道该组有多少成员,位置在哪里,但前后两次调用的情形也可能有变化。

组通信的实现在很大程度上依赖于硬件。在一些网络上,可以创建一个专门的网络地址(例如,可通过设置高序号中某位为 1 来指示),多台机器可通过这个地址来监听。当将一个包发送到这样的一个地址上时,该包就会自动地投递给所有在该地址上监听的机器。这种技术称为多点传送(multicasting)。使用多点传送实现组通信是非常简单的,它只需给每个组分配一个不同多点传送的地址。

无多点发送功能的网络有时仍有广播通信(broadcasting)功能,这意味着包含某一特定地址(比如,0)的包可以发送给网上所有的机器。广播通信同样也用于实现组通信,但效率要低一些。每台机器每收到一个包时,它的软件就检查该包是否是发给自己的。若不是,则丢弃该包。但是,处理这种中断要耗费一些时间。尽管如此,这种方法也做到了仅发送一个包就能使组的所有成员都收到该包。

最后,如果多点传送和广播通信都不可用,也可通过让发送者将包分别送往组的每个成员的方法来实现组通信。对一个有 n 个成员的组来说,就需要发送 n 个包,而不像多点发送和广播通信中只需发送一个信包。尽管这样做效率较低,但其实现方法仍是可行的,特别是在大多数组都比较小的情况下。为和多点传送和广播通信相区别,将从一个发送者到一个接收者的消息发送称为单点传送(unicasting)(点对点传输)。

2.5.2 设计的问题

组通信和通常的消息传递机制在设计上有相似的地方,例如,有缓冲与无缓冲,阻塞与非阻塞等等。然而,在组通信的设计中还有更多必须要作的选择,因为发送消息给一个组与发送消息给一个进程在本质上是不同的。此外,组内部的组织方式也多种多样。他们能够用不同于点对点通信的新颖的方式来编址。本节我们将讨论一些最重要的设计问题,并指出各种选择。

封闭组与开放组

根据消息发送的方式(即谁能发送给谁)将支持组通信的系统划分为两大类。一些系统支持封闭组(closed group),该系统只允许组内成员给该组发送消息。组外成员不能向作为一个整体的组发送消息,尽管它们可以给单个成员发送消息。相反,有一些系统支持开放组(open group),这些系统不具有这一特性。当使用开放组时,系统内的任何进程可以给任何组发送消息。封闭组与开放组的差别如图 2-28 所示。

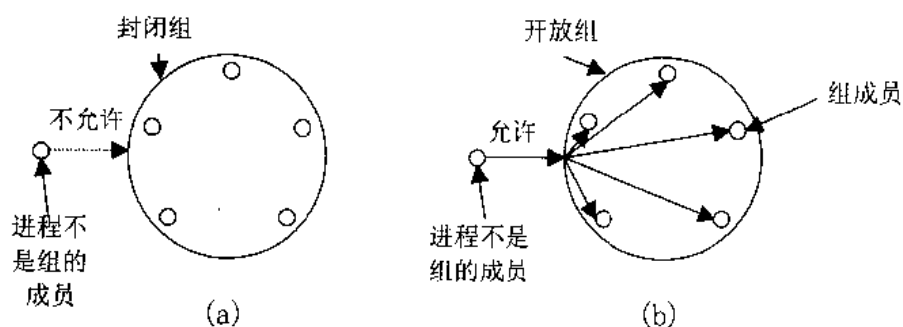


图 2-28 (a) 闭环外的成员不能给封闭组发消息
(b) 闭环外的成员可以给开放组发消息

系统采用封闭组还是开放组首先与组的形成原因有关。封闭组典型地用于并行处理。例如，协作进行象棋游戏的一组进程可形成一个封闭组。组内成员有相同的目的且不与外界交互。

另一方面，当用组的思想来支持复制服务器时，接收来自组外的客户请求是很重要的。此外，组内成员也需要使用组通信。例如，决定哪个成员去执行一个特殊的请求。开放组与封闭组的区别常常作为实现中要考虑的一个问题。

对等(或同位)组与分层组

封闭组和开放组的区别在于谁能与该组通信。两者的另一个重要的区别在于它们的内部结构上。在一些组中，所有的进程的地位相同，没有哪个进程作为领导，决议是由进程集体作出的。在另一些组中，存在某种类型的分层组织。例如，一个进程作为“协调员”，其他所有的进程作为“工作人员”。在这种模型中，当一个外部客户或“工作人员”产生一个工作请求后，就将请求发送给“协调员”。“协调员”然后找出最适合完成该项任务的“工作人员”，并将请求发给它。当然，可能会有复杂的分级。这两个通信模型如图 2-29 所示。

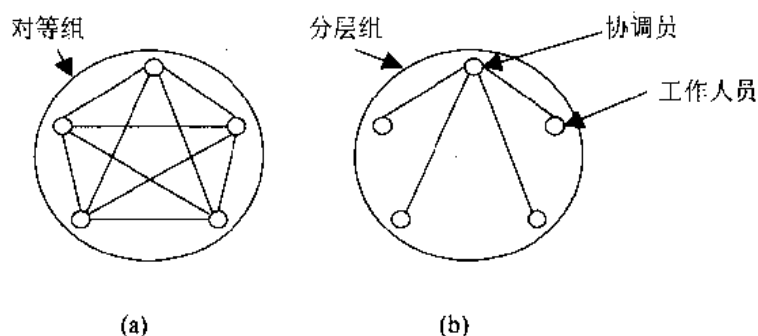


图 2-29 (a) 对等组的通信
(b) 简单分层次组的通信

这两种组织方式各有其优缺点。对等组(peer groups)是对称(symmetrical)的，而且没有单点故障。其中一个进程崩溃只会使组变小，而组中其他的进程仍能继续工作。它的缺点是组内要作出决定更为复杂，每作一个决定都需进行一次组内成员的表决，会造成一定的

时延和系统开销。

分层组(hierarchical groups)却有着相反的性质。如果协调员一崩溃,则整个系统就会瘫痪。但只要协调员在运行,它无须打扰其他任何成员就可作出决定。例如,一个分层组可能适用于一个并行的象棋程序。协调员将棋板上信息取来,生成多条合法的下一步棋,然后分送给组的其他成员,由它们计算。计算期间,生成多个新的棋盘布局,并且送回协调员汇总评估。从而决定下一步棋的走法。当一个组内工作人员空闲的,便请求协调员分给其新的工作。在这种方式下,协调员控制搜索策略和修剪游戏树(例如,使用 alpha-beta 搜索方法),实际的计算留给组内工作人员完成。

组的成员

使用组通信时,需要一些建立或删除组和允许一个进程加入或退出一个组的方法。一个办法是使用一个组服务器,所有这样的请求都发送给组服务器。它应维护一个包括所有组以及它们的确切成员在内的完整的数据库。这种方法简单、有效、易于实现。但不幸的是,它与所有的集中式技术一样,存在一个主要的缺点:单点故障。如果组服务器崩溃,组的管理也就停止了。这时,大多数的组或所有的组将不得不从头开始重新构造,并可能终止所有正在进行的工作。

一种相反的方法是以一种分布的方式来管理组内成员。在一个开放组中,一个要加入的外部进程只需向该组的所有成员宣布它的存在即可加入该组。在封闭组中,也需要做类似的事情(事实上,在一新成员加入时封闭组也只能变为开放)。要离开一个组,成员只需给该组的其他所有成员发送一个再见消息即可。

直到现在,所有这些都很简单。然而,有两个与组成员相关的带欺骗性的问题。

第一,如果一个组成员崩溃,实际上它就离开了这个组。问题是,与自愿离开不一样,没有任何礼貌的对这一事实的宣告。其他成员只有在发往它的消息得不到任何应答后,才发现这个崩溃的成员已离开。一旦确定这个崩溃的成员已经停机,就从这个组中删去该成员。

第二个棘手的问题是,加入与离开一个组它必须与发送的消息同步。换句话说,从进程加入到组的瞬间开始,它必须接收所有发往该组的消息。同样,一旦进程已经离开该组,它就不应收到该组的任何消息。组内其他成员也不应收到这个成员发出的消息。保证将加入或离开加入到消息流中的一种方法是将这个操作转换成一个发往整个组的消息。

最后一个与组成员相关的问题是在许多机器出故障而引起组不能正常工作的情况下应怎么处理。这时需要一些协议来重新建组。某个进程总是要带头开始工作,但是如果有两三个进程同时试图这样做会怎么样呢?这些协议将必须能够经受得起这样的考验。

组的编址(Group Addressing)

一个进程要将消息发送到组,它必须采用一种方法来指明它所需的组。换句话说,也需要像进程一样对组进行编址。

一种编址方法是给每个组一个唯一的地址,这与进程编址非常类似。如果网络支持多点传送,组地址可与一个多点传送地址相关连,这样就可以使用多点传送将消息发往该组。采用这种方法,消息会被送到所有需要该消息的机器上,而不会送到其他机器上。

如果只支持广播通信方式而不支持多点传送,那么消息就通过广播发送。然后每个机器内核得到该消息并从消息中取得组的地址。如果该机器上没有属于该组的进程,则丢

弃该消息，否则，将消息送往该机器上所有该组的成员。

最后，如果不支持多点传送与广播通信，那么每个发送消息机器的内核必须保存一份有进程属于该组的机器的列表。然后，内核给每个机器发送一条点对点的消息。这三种实现方法如图 2-30 所示。在这三种实现方法中需要注意的是，一个进程只需要向一个组地址发送一条消息，该消息就可以被传递到该组的所有成员。具体如何实现是由操作系统来完成的。发送者并不知道组的大小，以及是使用多点传送，广播通信，还是使用单点传送来进行通信的。

第二种编址方法是要求发送者提供一个所有目的地址的显式列表(比如，IP 地址)。使用这种方法时，*send* 调用中指明目的地址的参数是一个指向一个地址列表的指针。这个方法有个严重的缺点，它强迫用户进程(即，组成员)精确了解到哪个进程属于哪个组。换句话说，它是不透明的。此外，一旦组成员改变，用户进程必须更新它们的组成员列表。在图 2-30 中这个更新的管理可由内核来完成，这对用户进程来说是不可见的。

组通信还允许第三种十分新颖的方法来对组进行编址，我们称之为判定编址(predicate addressing)。在这个系统中，使用上面所述的一个方法，每条消息发往到组内的各成员(或整个网上成员)，但该消息不一样。每条消息包含了一个需要计算的判定。这个判定是一个布尔表达式。它可能涉及到接收者的机器号、它的局部变量或其他因素。如果该布尔表达式为 TRUE，则消息被接收。如果为 FALSE，则消息被丢弃。这种方法是可用的，例如，这种方法可用于给至少有 4M 空闲内存并愿意运行一个新进程的那些机器发送一条消息。

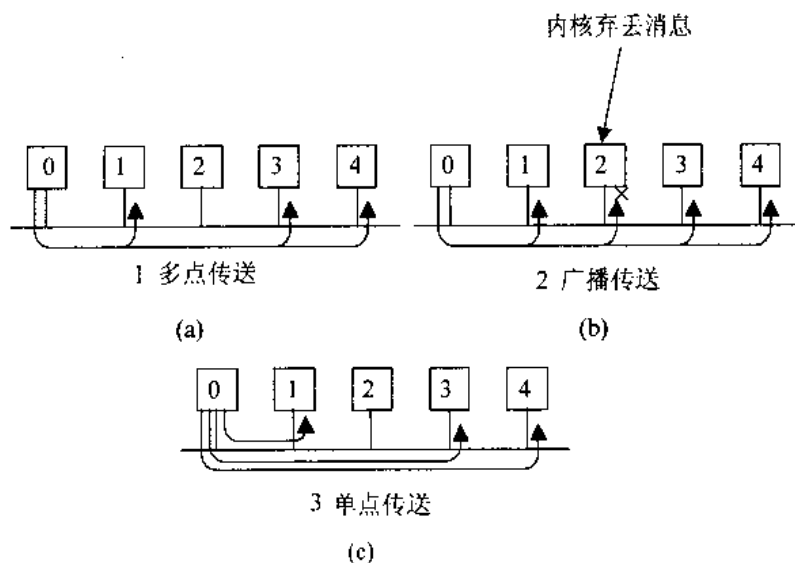


图 2-30 进程 0 向包括进程 1, 3, 4 的进程组发消息

(a) 多点传送实现

(b) 广播传送实现

(c) 单点传送实现

发送和接收原语

理想情况下，应将点对点通信和组通信融于一个单独的原语集中。但是，RPC 是一

个比 *send* 和 *receive* 更常使用的通信机制，它很难与组通信相融合。给一个组发送一条消息不能作为一个过程调用。最大的困难是，在 RPC 中，客户发送一条消息到服务器，之后返回一个结果，但在组通信中，对一条请求可能会有 n 个应答(reply)。过程调用如何能处理这 n 个应答呢？因此，一个常用的方法就是采用单向通信模型 *send* 和 *receive* 的显式调用，而不采用基于 RPC 的双向通信模型请求/应答(request/reply)。

过程调用用来激活一个组通信所用的库函数，可能与点对点通信所用的函数相同，也可能不相同。在基于 RPC 的系统中，用户进程不直接调用 *send* 和 *receive*，因此，将点对点通信与组通信的原语相融合也就更没必要了。如果用户程序自己直接调用 *send* 和 *receive*，可以采用这些已经存在的原语进行组通信而无须另建一套新的原语。

暂时假设我们想融合这两种通信方式。在发送消息时，*send* 的第一个参数指明目的地址。若这个参数是一进程地址，就发送一条消息给那个进程。若这个参数是一个组地址(或是指向一个目的地址表的指针)，就发送一条消息给该组的所有成员。*send* 中的第二个参数指向消息。

点对点通信和组通信，这个调用都可以是有缓存或无缓存的、阻塞或非阻塞的、可靠或不可靠的。通常这些选择都是由系统设计者设计好了并固化在系统中的，而并不是在每条消息的基础上加以选择的。组通信的引入不会改变这一情况。

类似的，*receive* 表示愿意接受一条消息，并且在收到一条消息前可能处于阻塞状态。如果两种通信方式相融合，当一个点对点通信的消息或组通信的消息到来时，就能完成 *receive* 调用。然而，由于两种通信机制经常用于不同的目的，有些系统引入新的库例程，比如说，*group_send* 和 *group_receive*。所以，一个进程就可以指定它需要点对点通信的消息，还是组通信的消息。

在上述设计中，通信是单向的。应答的消息本身是独立的消息，与先前的请求不相关。有时也需要得到与请求相关的应答，以此来接近 RPC 的通信的方式。在这种情况下，在发送一条消息后，需要启动一个进程重复调用 *getreply* 来收集所有应答，每调用一次 *getreply* 可收到一个应答。

原子性

我们多次提到的组通信的一个特性是要么全有要么全无特性。大多数的组通信系统都是这样设计的，当一条消息发送给一个组后，不是该组所有成员都正确收到，就是均未收到。不允许出现组内有些成员收到而有些成员收不到的情况。具有要么全有要么全无的传递叫做原子性(atomicity)或原子广播通信(atomic broadcast)。

原子性可使分布式系统编程容易得多，因此，它是我们想要得到的。当一进程发送消息给组时，进程不必担心如果有些组成员没有收到消息该怎么办。例如，在一个复制的分布式数据库系统中，假设有一个进程给所有的数据库机器发送一条消息创建一个新记录，接着又发送一条消息去修改该记录，如果有些组成员未接收到创建记录的消息，它们就无法执行修改操作，这样数据库也将变得不一致。如果系统能够保证将每条消息发送到该组的所有成员，该过程就比较简单。如果不能保证，该消息就不发给组中的任何一个成员，并将失败报告给发送者以作适当的恢复处理，那么处理上述问题就很简单了。

实现原子广播通信并不像看起来那么简单。图 2-30 的通信模式是不可行的，因为一台或多台机器接收超时是经常可能出现的。要确保每一台机器都能收到每一条消息，唯一

的办法是要求所有机器收到消息时发回一个确认消息。只要机器没有崩溃，这种方法还是可行的。

许多分布式系统以容错为目标，对这些系统来说，在出现机器故障时仍能保持原子性是很重要的。在这一方面，图 2-30 中的三种方法均是不能满足的，因为一些最初的消息可能会由于超时而没有到达接收者，而随后发送者又发生了崩溃。在这种环境下，就会有些组成员收到了消息而有些没有收到。更糟糕的是，没收到消息的组成员甚至并未意识到丢失了数据，因此，它们不会请求重发。最后，即便这些成员知道丢失了消息，而发送者已经崩溃了，无法再提供丢失的消息。这样根本无法确保原子性。

然而，不是毫无办法的。这里有一种简单的算法证明原子广播通信至少是可能的。发送者以发送一条消息给所有组成员开始。设置一时钟并在必要的时候重发消息。当一个进程接收到一条消息时，如果它还没有看到这一特殊的消息，它就将该消息转发给组内其他成员(如果有必要也设置时钟和重发消息)。如果进程已经看到了这个特殊消息，就不必做这一步了，并且将该消息丢弃。这样，不管有多少个机器崩溃或有多少个信包被丢失，最终所有幸存的进程都将收到该消息。以后我们将讨论实现原子性的更有效的算法。

消息的顺序

要使组通信易于理解和使用，有两种性质是不可缺少的：首先是上面讨论的原子广播通信，它确保了一条消息要么被所有的组内成员收到，要么没有一个成员能收到。第二个性质涉及到消息的顺序。在图 2-31 中，有五台机器，每一台机器有一个进程，进程 0、1、3、4 属于同一个进程组。进程 0 与 4 同时想给该组发送一条消息。假设没有多点传送和广播通信，这样每个进程只得发三条分开的消息(用单点传送)。进程 0 发送给进程 1、3、4；进程 4 发送给进程 0、1、3。六条消息在时间上的交错如图 2-31(a)所示。

问题是，当两个进程竞相访问 LAN 时，在网中消息传送的顺序是无法确定的。我们看到在图 2-31(a)中，进程 0 先占用网络向进程 1 发送了消息，随后进程 4 连续三次占用网络，先后向进程 0、1、3 发送了消息。在图 2-31 中的两部分以不同方式表示这六条消息的发送顺序。

现在我们来看看如图 2-31(b)所示的进程 1 和进程 3 的情形。进程 1 首先收到了来自进程 0 的消息，然后收到了来自进程 4 的消息。进程 3 开始没有收到消息，继而先后收到了来自进程 4 和进程 0 的消息。这样 4 与 0 的两条消息以不同的顺序到达了进程 1 和 3。如果进程 0 和 4 都想去改变数据库中的同一记录的值，那么，进程 1 和进程 3 执行的最终结果是不相同的。不用说，这与(真正的硬件多点传送)组内部分成员收到消息而部分成员未收到消息(原子性失败)的情形是一样糟糕的。为使编程更为合理，系统必须很好地定义一种关于消息传递顺序的语义。

最好的保证就是立即发送所有消息并让它们保持发送的顺序。如果进程 0 发送 A 消息，稍后进程 4 发送 B 消息，系统应先将消息 A 发送给组内各成员，然后再将消息 B 发送给组内各成员，这样所有接收者都可以同样的顺序收到消息。这种传递消息的模型编程人员能理解而且它们的软件也可以它为基础。我们将这种方法称为全局时间顺序(global time ordering)，因为它能将消息精确地按发送顺序传递到目的地(为了方便，忽略根据爱因斯坦的狭义相对论所得没有绝对的全局时间)。

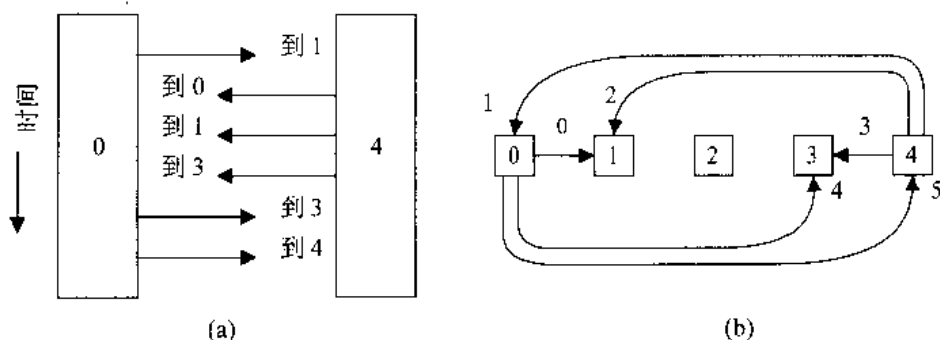


图 2-31 (a) 由进程 0 和 4 发出的消息插入了时间
(b) 图像再现了六条消息，表明了到达顺序

绝对时间顺序不易实现，因此有些系统提供了各种打折扣的方法。其中一个方法是一致时间顺序(consistent time ordering)，使用该方法时，若有两条消息，比如说， A 和 B ，以很小的时间间隔发送，系统先取其中一个作为“第一个”发送给所有组内成员。然后，再取下一个发送给组内各成员。选为“第一”的消息有可能实际不是最早发送的消息，但由于没有谁知道这一点，我们赞成系统的行为可以不用依赖它。实际上，这种方法保证组内成员按同样的顺序收到了各条消息，但这个顺序可能并不是发送消息的顺序。

甚至可以使用更差的时间顺序。在本章后面讲述 ISIS 时，我们将研究一种基于因果关系的更差的时间顺序。

组的重叠

我们曾提到一个进程能够同时成为多个组的成员。然而，这导致了新的不合理性。在图 2-32 中，有组 1 和组 2 两个组。进程 A 、 B 、 C 是组 1 的成员，进程 B 、 C 、 D 是组 2 的成员。

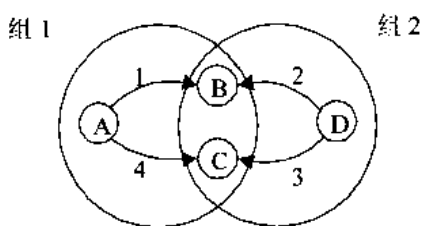


图 2-32 四个进程和四条消息。进程 B 和 C 从 A 和 D 中以不同顺序获得消息

现在假设进程 A 与 D 同时向它们所在的进程组发送消息，组内采用全局时间顺序传送。和上面的例子一样， A 与 D 采用单点传送的方法。如图 2-32，消息发送的顺序由数字 1 到 4 标出。我们可以看出 B 与 C 收到消息的顺序是不一样的。 B 先收到了 A 的消息后收到 D 的消息，而 C 则相反。

这里的问题是：尽管在每个组内都采用了全局时间顺序，但在多个组之间还需要协调。一些系统支持重叠组中的精确定义的时间顺序，有些系统不支持(如果组不相交，不存在这样的问题)。在不同的组间实现一致的时间顺序是很困难的，因此，值不值得去做

还是个问题。

可测量性

最后一个设计上的问题是可测量性。如果进程组内只有几个成员，上述的许多算法将工作良好。但是如果组内有数十个、数百个以至成千上万个成员，或是有成千上万个进程组时会怎样？或者，整个系统不限于一个局域网，而是有网关连接的多个局域网或是洲际网，那么这些算法能实现并有效吗？

网关的出现会影响我们实现中的许多特性，首先，多点广播在这里更为复杂。例如图 2-33 中所示网络，它由四个 LAN 与四个网关构成。

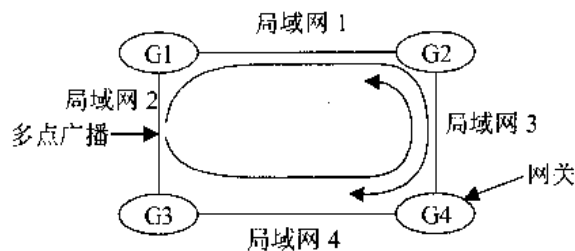


图 2-33 互联网上的多点广播引发问题

现在设想在 LAN2 有一机器发出了一个多点广播的消息。当多点广播的信包到达网关 G1 和 G3 时，它们将怎么做？若 G1 与 G3 丢弃这个信包，则其他三个子网就无法看到该消息。若 G1 与 G3 将消息转发，那么消息到达了 LAN1 与 LAN4，然后分别通过 G2 与 G4 都到达 LAN3，这样 LAN3 会收到两次这样的信包。更糟的是，由于四个子网构成一回路，且又采用了多点广播的方式，经过 G4 的消息发往了 G2 到达 LAN1，经过了 G2 的消息发往了 G4 到达 LAN4，最终都会回到 LAN2。因此，这里需要一个更复杂的算法通过跟踪信包在网上的传送路径来避免信包被成倍地多点广播。

另一个问题是一个内部网上的组通信利用了 LAN 上某一瞬间只能传送一个信包的事实。实际上，信包在网上按绝对全局时间顺序发送是很重要的。但是在带有网关的多个子网构成的网上，同时刻网上传送两个信包是可能的，这就破坏了绝对全局时间顺序要求的某一时刻只有一个信包在网上传递的性质。

最后，需说明的是其他的一些算法或由于计算复杂或使用了集中式部件等，不能用在组通信上。

2.5.3 在 ISIS(组合软件调用系统)中的组通信

作为一个组通信的例子，ISIS 是由 Cornell 发展起来的(Birman,1993;Birman 和 Joseph,1987a,1987b;Birman 和 Van Renesse,1994)。ISIS 是构造分布式应用的工具包。例如，它可以用于协调在华尔街证券商行的所有股票经纪人的股票交易。ISIS 并不是完全意义上的操作系统，而是运行在 UNIX 或其他操作系统上的一组程序。学习它是因为它有很多文字描述并且有几个应用实例。在第 7 章中，我们还将学习 Amoeba 中的组通信，它使用了不同的方法。

在 ISIS 中的关键思想是同步，它的主要通信原语是不同形式的原子广播。在了解 ISIS 如何实现原子广播之前，首先来看几个不同的同步系统。同步系统是指一个每个事件都严格按序发生，并且每个事件(如广播)的完成不需要时间。例如，在图 2-34(a)中，如果进程 A 发送消息给进程 B、C、D。消息从 A 发出立即到达所有目的地。同样，从 D 随后发出了一条消息也立即到达所有的目的地。从系统外观察，系统包括了互不重叠的一些事件，这个性质可使我们更好地理解系统特性。

由于同步系统不太可能实现，一个松散同步系统就应运而生，见图 2-34(b)。松散同步中每个事件占用有限的一段时间，但所有的事件会以同样的顺序在所有成员那里出现。比如，所有的进程都能以同样的顺序收到消息。这个思想与我们前面讨论到的一致时间顺序是相同的。

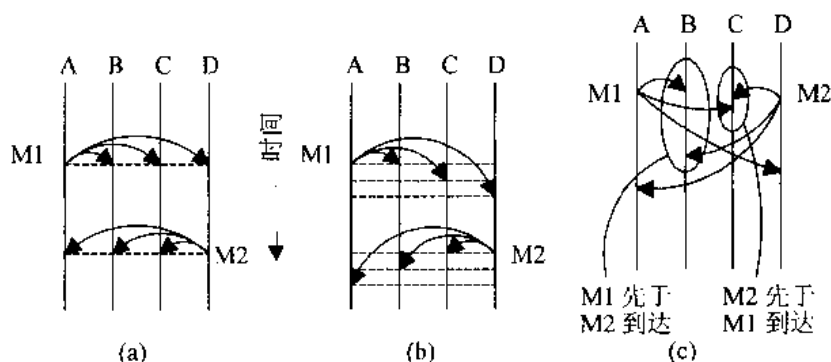


图 2-34 (a) 同步系统
(b) 松散同步系统
(c) 虚拟同步系统

这样的系统是可以构造的，但是在有些应用上，我们还可以采用一些更松散的语义，以提高性能。图 2-34(c)中给出了一个虚拟同步系统。在这个系统中放宽了对顺序的限制，但是在仔细选择的环境下，还是可以实现同步的。

让我们来看一看这些环境。在一个分布式系统中，如果第一个事件可能会影响第二个事件的性质和行为则这两个事件是因果相关的。因此，如果有进程 A 发送一条消息给 B，B 检查该消息后发送一条新的消息给 C。第二条消息与前面的一条消息就是因果相关的。因为发往 C 的消息中可能有先前的消息。这种情况是否会发生并不重要，如果两个事件之间存在着影响，那么就让它存在。

无关的两个事件称为并发。若进程 A 向进程 B 发送消息，同时进程 C 向进程 D 发送消息，这时发往 B 的消息与发往 D 的消息是互不影响的，称为并发。我们所要说的虚拟同步就是，若两条消息是因果相关的，那么所有接收进程必须以同样的顺序接收它们。若两条消息是并发的，那么就不必以同样的顺序接收。系统可以随意地将这些消息以不同的顺序投递给不同的进程。

ISIS 中的通信原语

现在让我们来看看 ISIS 中用到的通信原语。在 ISIS 中已经定义了三种通信原语：ABCAST, CBCAST, GBCAST。ABCAST 提供松散同步通信，用于向组成员传送数据。

CBCAST 提供虚拟同步通信,也用于向组成员传送数据。GBCAST 与 ABCAST 类似,但用于管理组内成员而不传送普通数据。

通常 ABCAST 使用一两阶段提交协议。其工作情况如下:发送者 A 分配一时间戳(确切地说,仅是一个序号),将它放在一消息中,发送给组内各成员(要明确地指出它们)。各成员将自己的时间戳返回给 A,该时间戳应大于该成员以前所收发的任何时间戳。当 A 收到所有时间戳后, A 选择最大的一个,并将提交消息发给各成员。提交消息按时间戳大小提交给各应用程序。这协议保证了所有消息将按同样的顺序投递给各进程。

由于 ABCAST 复杂且昂贵。ISIS 设计者提出了 CBCAST 原语。它仅仅保证了所有因果相关的消息能按序发送到目的地。其工作情况是这样的:如果一个组含有 n 个成员,每个成员进程将保存一个 n 维向量,向量的每一维对应一个成员。第 i 维的值就是从进程 i 发过来的最后一次消息的序号。这向量并非由用户进程自己控制,而是由运行系统来控制的,并且在开始时将它初始化为 0。如图 2-35 的顶部所示。

当一进程有消息要发送时,它在自己保存的向量中属于自己的位置上增加 1。然后将该向量放在消息中。当在图 2-35 中的消息 $M1$ 到达 B 时, B 会检查 $M1$ 中是否有 B 不知道的消息。 $M1$ 中向量的第一位刚好要比 B 向量中的第一位大 1,这正是从 A 发来的消息所要求的。而且,两个向量的其他位均相同,这样,消息被接收并传送给在机器 B 上运行的组成员。如果在 $M1$ 中,向量的其他任何位大于 B 向量中的相应位,则 $M1$ 不能被 B 接收。

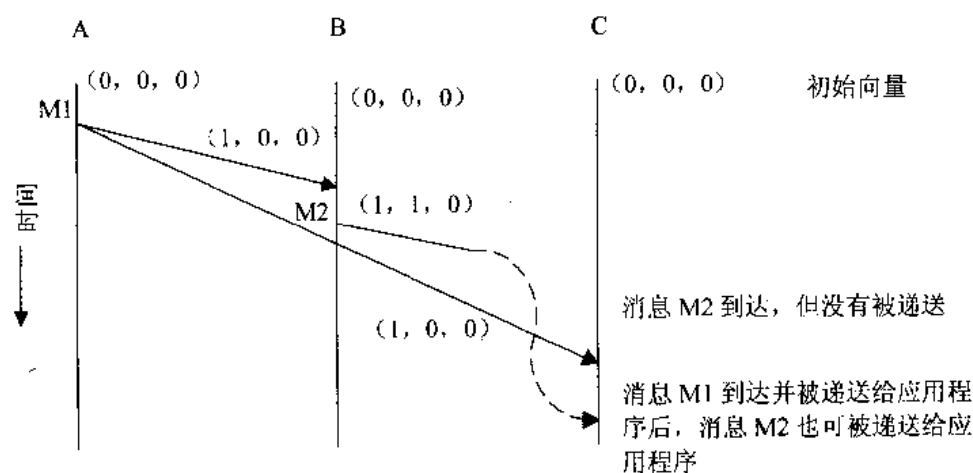


图 2-35 只有当所有先到的消息都被发送出去之后,这些消息才能够被发送

现在 B 发送一个自己的消息 $M2$ 给 C, $M2$ 比 $M1$ 先到达 C。C 从 $M2$ 的向量中看到 B 在发送 $M2$ 之前收到过 A 发送的 $M1$, 由于 C 还没有收到 A 的消息 $M1$, C 将 $M2$ 放入缓冲区,等待 $M1$ 到达。只有 $M1$ 到达后 C 才会接收 $M2$ 。

决定对到来的消息是接收还是推迟的算法描述如下: V_i 是发送方向量中的第 i 位。 L_i 是接受方向量中的第 i 位。假设消息由 j 发送。接收的第一个条件是 $V_j = L_j + 1$ 。表明这是 j 发送来的下一条消息,以前的消息均已收到没有遗漏(从同一个发送者发出的消息总是因

果相关的)。接收的第二个条件是：对所有的 $i \neq j$ 有 $V_i \leq L_i$ 。表明接受方已有所有发送方收到的消息。如果到来的消息通过这两个条件，那么运行系统可将该消息送给本机上的用户进程。否则，该消息必须等待。

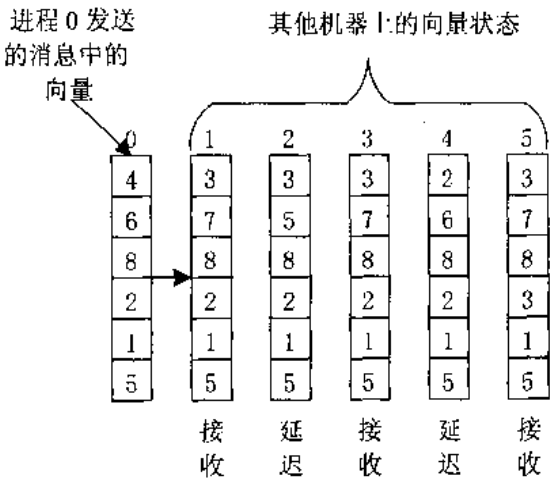


图 2-36 CBCAST 使用的向量的样例

在图 2-36 中我们给出了这个向量机制的一个更详细的例子。这里进程 0 向其他五个成员发送消息，消息中包含向量(4, 6, 8, 2, 1, 5)。进程 1 的除 0 以外的其他位与 0 进程的相同，表明进程 0 收到的那些消息进程 1 也收到了。而进程 1 刚发送一条消息，进程 0 未收到，但这并不影响进程 1 接收消息。此外，进程 1 上的对应 0 的那位刚好比进程 0 的相应位小 1，这样进程 1 接收进程 0 来的消息 4。进程 2 的向量表明尚未收到进程 1 发来的消息 6，因此，进程 2 还不能接收进程 0 发来的消息。进程 3 看上去也收到了发送者收到的其他消息，进程 3 甚至收到了进程 1 的消息 7，只是还没有收到来自进程 0 的消息，因而，进程 3 可以接收进程 0 来的消息 4。由于进程 4 中向量对应的进程 0 的位置上的值是 2，这就说明进程 4 丢失了进程 0 发来的消息 3，因而进程 4 就不能接收进程 0 发来的消息 4。在进程 5 中，该向量在 0 进程的位置上刚好比进程 0 少 1，而且它比进程 0 先收到了来自进程 1 的消息，因而，它可以接收进程 0 来的消息 4。

ISIS 同时也提供了容错性且支持使用 CBCAST 通信的重叠组的消息顺序。这种算法使用起来有些复杂，详情请见(Birman 等., 1991)。

2.6 小结

集中式系统与分布式系统主要区别是后者的通信机制很重要。在分布式系统中提出和实现了通信的不同方法。对于相对传送速度较慢的广域分布式系统来说，有时使用像 OSI 和 TCP/IP 这样的面向连接的分层协议，因为这时我们主要想克服的是物理线路传输数据的不可靠性。

对局域的分布式系统来说，很少使用分层的协议，而是采用较简单的模式。在这种

模式中一个客户向服务器发送请求消息，然后服务器发回应答。由于简化了大多数协议层，这种模式能取得更高的性能。在这种消息传递系统中，关于通信原语，我们讨论的设计问题有：阻塞与非阻塞、有缓冲区与无缓冲区、可靠与不可靠等等。

基本的客户/服务器模型的问题是进程的通信采用输入/输出来处理的。这样广泛应用了一个更先进的进程通信模式 RPC。在 RPC 中，一台机器上运行的客户进程调用在另一台机器上的某个过程。运行系统嵌入在存根的过程库中，用来收集参数、建立消息并与内核一起建立一个实际的传递位的接口。

RPC 尽管比原始的消息传递模式要先进些，但它也有自身的问题：服务器必须被正确定位；指针与复杂的数据结构难以传递；全局变量很难使用；很难有精确的 RPC 语义。因为客户与服务器都有可能崩溃。实现高效的 RPC 是不容易的，还需要更仔细的思考。

RPC 只能用于一个服务器与一个客户通信的方式。当有多个处理器时，例如，复制的文件服务器，这多个服务器需要作为一个组来进行各成员间的通信。这样，就需要一个另外的通信机制。像 ISIS 这样的系统为这个目的提供了一种新的模式：组通信。ISIS 提出了各种不同的原语，其中最重要的是 CBCAST。CBCAST 提供了基于因果相关的松散的通信语义，并在消息中放入表示各进程发送消息情况的向量，以实现这个语义。

习 题

- (1) 在许多层次协议中，每层都有自己的头信息。如果在每一个报文前面用一个头信息包括所有的控制，相信比分开的头信息更有效，为什么不这样做呢？
- (2) 开放系统的含义是什么？为什么有些系统不是开放的？
- (3) 面向连接协议和面向非连接协议有何不同？
- (4) 一个 ATM 系统以 OC-3 的速率传递信元，每个包 48 字节长，刚好放进一个信元。一个中断耗时 $1\mu s$ ，CPU 用于中断处理的时间是多少？如果包长是 1024 字节呢？
- (5) 一个完全混淆的 ATM 头信息被正确接收的可能性是什么？
- (6) 对图 2-9 作一个简单的修改，使网络传输减少。
- (7) 在客户-服务器系统中通信原语是非阻塞的。在报文实际传送之前，对 *Send* 的调用已经完成，为了减少开支，一些系统不拷贝数据到内核，而是直接从用户空间发送。对这样一个系统，设计两种方法，发送者能知道传送已经完成，可以重新使用缓冲区。
- (8) 在许多通信系统中，调用 *Send* 设置一个计数器，以保证当服务器崩溃时客户机不会永远挂起。假设一个容错系统，所有的客户机和服务器都用多处理机实现，因此，客户机或服务器的崩溃可能性为零。你认为在这个系统中能安全地避免超时吗？
- (9) 当使用有缓冲通信时，一个原语通常由用户进程建立一个信箱，在书中，没有指出原语是否必须指出信箱的大小，讨论每一种方法。
- (10) 在本章所有的例子里，一个服务器只能监听一个地址，在实际中，一个服务器在同一时间内监听多个地址可能更方便，例如，同一进程运行一系列密切相关但分配不同地址的服务，写出实现这个目标的设想。
- (11) 过程 *incr* 有两个参数，该过程的作用是给每个参数加 1。假设现在 *incr* 的两个参数均为 *i*，即 *incr(i,i)*。如果 *i* 的初值为 0，求在变参调用后和拷贝/恢复调用后 *incr* 的返回值

分别是什么？

- (12) PASCAL 中有一个构造叫记录的变体部分，该变体部分中可由多个变体组成，每次可以选择一个作为记录的一个域。在运行时，并不能确定使用变体部分中的哪个变体。试解释 PASCAL 的这个特性对 RPC 有何影响？
- (13) 通常 RPC 的执行顺序中包括了向内核请求中断以便将消息由客户发送到服务器。假设存在一个可做网络 I/O 的协处理芯片，且该芯片可从用户空间直接寻址到。请问该芯片有存在的必要吗？如果该芯片存在，那么 RPC 会有哪些步骤？
- (14) SPARC 芯片使用 32 位长的字，以最高有效字节的格式存储。而 486 的存储格式为最低有效字节优先。如果从 SPARC 传送整数 2 到 486，那么在 486 上看到的数值是多少？
- (15) 在 RPC 系统中处理参数转换的一种方法是：客户发送消息时使用自己的格式，由服务器检查客户的格式是否与自己的相同，若不同则转换。对文本而言，有一种建议是：让客户在第一个字的第一个字节放一个编码以说明自己的格式。然而，能否精确地找到第一个字的第一个字节却是个问题，请问该建议是否可行？
- (16) 在表 2-3 中，binder 接口的 *deregister* 过程中有一个参数是进程唯一标识号。由于进程名与版本号已作为参数存在且也能唯一地标识进程，那么请问进程唯一标识号这个参数在 *deregister* 中是否有必要存在？
- (17) 从一个远程服务器读某文件中的第一个块是一个可幂等次执行的操作，对于向文件中第一个块写入的这个操作而言，它是一个可幂等次执行的操作吗？
- (18) 在如下的几个应用中，试讨论应采用至少一次的语义好，还是至多一次的语义好？
 - (a) 在一个文件服务器上读、写文件；
 - (b) 编译一个程序；
 - (c) 远程的银行业务。
- (19) 假设一个空 RPC(0 字节数据)需时间 1.0ms，每增加 1K 数据，时间增加 1.5ms。如果要从文件服务器读 32K 的数据，请计算一次读取 32K 数据的 RPC 所需的时间和 32 次读取 1K 数据的 RPC 所需的时间？
- (20) 原子广播通信是怎么来管理组的成员的？
- (21) 当一个计算要运行较长的时间，一种明智的方法是周期性地设置检查点来把进程的运行状态保存到磁盘上，以备进程崩溃时使用。在这种方式下，该进程崩溃后可从检查点开始重新启动而不用从头开始。试设计一种方法来为一个并行多处理的计算设置检查点。
- (22) 设想在一个所有机器都有冗余多处理器的特殊的分布式系统中，机器崩溃的可能性低到可以忽略不计。请设计一个简单的方法在只能使用单点传送通信的情况下，来实现全局时间顺序和原子广播通信。(提示：将机器置于一个逻辑环中。)

第3章 分布式系统的同步

在第2章，我们已介绍了分布式系统中进程之间是如何进行通信的，使用的方法包括分层协议、请求/应答消息传输机制（包括RPC）以及组通信。尽管通信非常重要，但它并不是分布式系统的全部内容。与此紧密相关的是进程之间如何协作及如何彼此同步。比如说，在分布式系统中临界区如何实现，资源如何分配。本章将研究这些内容及其他与分布式系统中进程合作及同步有关的一些问题。

在单CPU系统中，临界区、互斥和其他同步问题经常使用信号量、管程(monitors)等方法来解决。这些方法在分布式系统中并不十分适用，因为它们必须依赖于共享存储器的存在。比如，有两个进程通过使用信号量而相互作用，它们必须都能访问信号量。如果它们在同一台机器上运行，它们能够共享内核中的信号量，并通过执行系统调用访问它。但是，如果它们运行在不同机器上，这种方法就不适用了，而需要其他技术，甚至看似简单的问题，比如判断事件A在事件B之前还是之后发生的问题也需要认真考虑。

由于时间在一些同步方法中起着重要的作用，我们将先从时间以及它的测量入手，然后介绍互斥和选举算法，之后我们再研究一种称为原子(基本)事务的高层同步技术，最后，介绍分布式系统的死锁问题。

3.1 时钟同步

分布式系统的同步比集中式系统的同步要复杂些，因为前者必须使用分布式算法。如果像集中式系统一样，在某地收集有关系统的所有有关信息，然后让某个进程分析并作出决定，是不切实际的。一般说来，分布式算法有如下性质：

- (1) 相关信息分散在多台机器中；
- (2) 进程决策仅仅依赖于本地信息；
- (3) 系统中单点故障应该避免；
- (4) 没有公用时钟或其他精确的全局时间资源存在。

前三点都说明在一处收集所有信息并对它进行处理是不可接受的。比如，资源分配（以一种无死锁的方式分配），向单一的管理进程发送所有I/O请求，由它来检查它们，根据表中的信息允许或拒绝请求是不实际的。在大系统中，这种解决方法会给某个进程带来太重的负担。

进一步而言，一个单点故障就会造成系统不可靠。最理想的情况是，一个分布式系统应该比单机系统更可靠，一台机器崩溃不影响其他机器的继续运行，我们最不希望的是，一台机器（例如：资源分配器）的故障使大量其他机器（它的用户）停滞不前。不通过集中而获得同步所使用的方法应该与传统操作系统所用的方法不同。

上述的最后一点是十分关键的，在集中式系统中，时间的概念很清楚，当进程想知道时间时，它使用系统调用，由内核提供。如果进程 *A* 询问时间，之后进程 *B* 也询问时间，*B* 得到的时间值就应该大于或等于 *A* 所得到的时间值，但一定不会小于 *A* 得到的时间值。在分布式系统中，获得时间上的一致是并不容易的。

考虑一个简单的例子，在缺少全局时间时的 UNIX 下的 *make* 程序。一般，UNIX 系统中一个大程序通常可被分割成多个源文件，如果某个源文件发生变化，只需将该文件重新编译即可，而不需要对所有的文件进行重编译。如果一个程序由 100 个文件组成，这样修改一个文件不需重新编译其余的文件，将大大提高编程人员的工作效率。

make 程序所使用的这种方法很简单，当编程者修改完所有的源文件，他启动 *make* 程序，看一下源文件和目标文件最后一次修改的时间，如果源文件 *input.c* 时间为 2151，而相应的目标文件 *input.o* 时间为 2150，*make* 就知道 *input.c* 在 *input.o* 创建后被改动过。因此，*input.c* 必须重新编译。相反，若 *input.c* 时间为 2144，而 *input.o* 时间为 2145，就不必再重编译了。*make* 检查所有的源文件并找出哪一个需要重编译，调用编译器重新编译它。

现在设想一下，在没有统一时间的分布式系统中会发生什么情况。假设 *output.o* 的时间如上为 2144，紧随其后 *output.c* 被修改，但是由于它所在机器上的时钟略慢，造成 *output.c* 时间为 2143，如图 3-1 所示。*make* 将不再调用编译器，最终可执行的二进制程序将包括由老的源文件和新的源文件所产生的混合目标文件，这样可能将不能再正常执行。编程者将为此去检查源代码以寻找错误所在，而搞得精疲力尽。

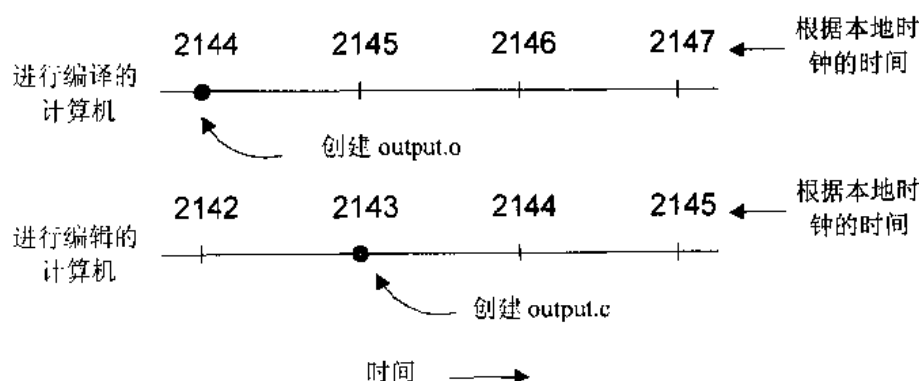


图 3-1 当每台机器有它自己的时钟时，一个发生于另一事件之后的事件可能会被标记一个比另一个事件更早的时间

既然时间是人们考虑问题的基础，然而正如我们看到的，没有同步时钟的后果是如此可怕，我们就先从这一简单的问题开始研究，在分布式系统中所有时钟可能同步吗？

3.1.1 逻辑时钟

几乎所有的计算机都有一个计时电路，尽管广泛使用“时钟”这个词来表示这些设备，但在通常情况下它们并不是通常意义的时钟，可能称它们为计时器更好些。计算机上的计时器通常是由一个精确的石英晶体制成，当在其张力限度内时，石英晶体以一定的频

率振荡，这种频率取决于晶体本身如何切割及其受到的张力大小。与每个晶体相关的是两个寄存器、一个计数器和一个保持寄存器。每次振荡时使计数器减 1，当计数器减为 0 时，产生一次中断，计数器重新从保持寄存器中装入起始值，通过这种方法可以编程使得一个计时器每秒能产生 60 次中断或以其他希望的频率产生中断成为可能，每次中断称作一个时钟点 (clock tick)。

当系统刚启动时，总是要求操作者输入日期和时间，然后将它们转换成某一已知起始时间后的时钟值，并将它存储在存储器中。在每个时钟点时，中断服务程序将此值加 1。用这种方法进行 (软) 时钟计时。

在单机单时钟中，如果这个时钟被瞬间关闭问题不会太大，因为这台机器上的所有进程都使用同一时钟，它们仍将内在地保持一致。比如，文件 *input.c* 时间为 2151，文件 *input.o* 时间为 2150，*make* 将重新编译源文件，假设时钟关闭的时间为 2，真正的时间则分别为 2153 和 2152，所有有用的只是相对时间。

一旦引入多 CPU 系统，每个 CPU 将拥有自己的时钟，情况将发生变化。尽管每个晶体振荡频率总是相当稳定，但保证不同计算机上的晶体以完全相同的频率振荡是不太现实的。实际上，当系统有 n 台计算机时，所有 n 个晶体将以略微不同的速度振荡，导致 (软) 时钟逐渐不同步，当同时读这些时钟时，也将得到不同的值，这种在时间值上的不同称作时钟偏移 (clock skew)。时钟偏移的结果如上例所示，一个程序期望与文件、目标文件、进程或消息相联系的时间应是正确的，且与产生它们的机器 (即使用哪个时钟) 无关。

这一点又把我们带到了最初的问题，即同步所有时钟产生一个单一的、无二义的时间标准是否可能。在以前的一篇经典的论文中，Lamport (1978) 阐明了时钟同步是可能的，并且描述了实现的算法，他还继续对这个问题进行了一些研究 (Lamport 1990)。

Lamport 指出时钟同步不需要绝对同步。如果两个进程无相互作用，它们的时钟无须同步，因为即使缺少同步也觉察不出来，不会产生什么问题。他进一步指出，通常并不必需所有进程在时间上完全一致，而是它们在事件发生的顺序上要完全一致。在上述 *make* 程序的例子中，计时的目的在于说明 *input.c* 比 *input.o* 早还是晚产生，而并不是它们各自产生的绝对时间。

对多数应用来说，所有机器在同一时间上达到一致就足够了，但这里的时间并不一定要与广播中每小时报告一次的真实时间一样。比如对 *make* 程序的运行，只要所有的机器都认为当前是 10:00 就可，而尽管当前的实际时间是 10:02。故对于某一类算法，重要的是内部各时钟的一致，而不是它们是否与真实时间接近，这类算法中通常将时钟称为逻辑时钟 (logical clock)。

如有特殊限制，不仅时钟要完全一致，而且不能与真实时间有一点点的误差，这种时钟称作物理时钟 (physical clocks)，本章将讨论 Lamport 算法，它用逻辑时钟进行同步，接下来，我们将介绍物理时钟的概念，说明物理时钟是如何同步的。

为了同步逻辑时钟，Lamport 定义了一个关系称作“先发生”。表达式 $a \rightarrow b$ 读做“ a 在 b 之前发生”，意思是所有进程认为先发生事件 a ，接着发生事件 b ，这种先发生关系有如下两种情况：

① 如果 a 和 b 是在同一进程中的两个事件，且 a 发生在 b 之前，则 $a \rightarrow b$ 为真。

② 如果 a 是一个进程发送消息的事件， b 为另一个进程接收这一消息的事件，则 $a \rightarrow b$ 为真，消息不能在发送之前接收，也不能在发送的同时接收，因为传送过程还需要一定时间。

先发生是一个传递关系，所以若 $a \rightarrow b$ 且 $b \rightarrow c$ 则 $a \rightarrow c$ 。如果两个事件 x 和 y ，出现在不同进程中，但并不交换消息（甚至不由第三方间接交换），则 $x \rightarrow y$ 为假， $y \rightarrow x$ 也为假。则事件 x 和 y 称为并发事件，即它们之间不存在谁先谁后的问题。

我们需要一种测量时间的方法，使得对每一事件 a ，在所有进程中都认可给它分配一个时间值 $C(a)$ ，这些时间值必须有如下性质：即若 $a \rightarrow b$ 则 $C(a) < C(b)$ 。重述我们前面讨论的情况，若 a 和 b 是同一进程中的两个事件，且 $a \rightarrow b$ ，则 $C(a) < C(b)$ 。同理，若 a 是一个进程发送消息的事件， b 为另一个进程接收消息的事件，那么 $C(a) < C(b)$ 。除此还要求时钟时间值 C 必须向前走（递增），不能倒退（减少）。校正时间的操作是给时间加上一个正值，而不是减一个正数。

现在我们来查看 Lamport 算法怎么给事件分配时间。图 3-2(a) 中描绘了三个进程。各进程运行在不同的机器上，每个机器都有自己的时钟，以各自不同的速率工作，如图所示，在进程 0 的时钟滴答 (tick) 了 6 下时，进程 1 的时钟滴答了 8 下，进程 2 的时钟滴答了 10 下，每个时钟均以其不变的速率工作，但由于晶体之间的差异，其工作速率不同。

在时刻 6，进程 0 将 A 消息发送到进程 1，消息到达的时间取决于你信任哪一个时钟，不管怎样，当它到达时，进程 1 读到的时刻是 16，若消息自身携带的起始时刻是 6，进程 1 会推算到它需滴答 10 下才到达，这个值是可能的。依次类推，消息 B 从进程 1 到进程 2 需要滴答 16 下这也是可能的。

现在出现了有趣的问题，消息 C 从进程 2 到进程 1 是在 60 时刻离开，56 时刻到达。同理，消息 D 从进程 1 到进程 0 是在 64 时刻离开，54 时刻到达，这是绝对不可能出现的，必须防止这种情况的出现。

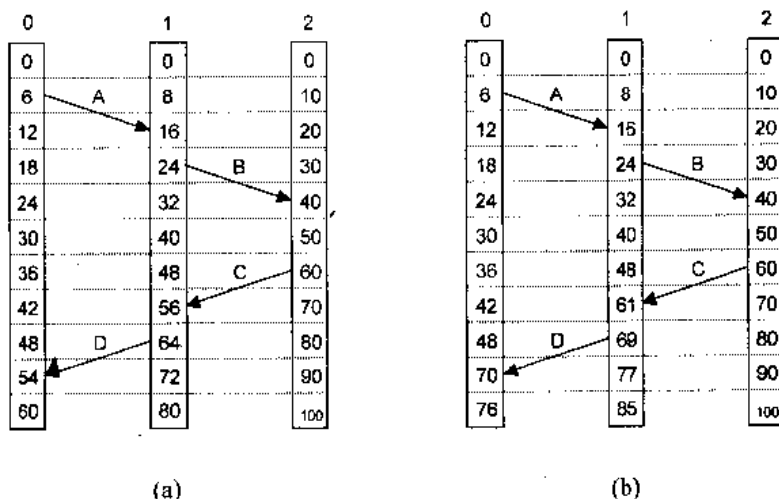


图 3-2 (a) 三个进程，每个均有自己的时钟。每个时钟速率不同
(b) Lamport 校正时钟的算法

Lamport 的解决方案是直接使用先发生关系，因为 C 在 60 时刻离开，它只能在 61 时刻或更晚时刻到达，所以每条消息都携带发送者的时钟以指出其发送的时刻，当消息到达时，接收者时钟若比消息发送时钟小，就立即将自己的时钟调到比发送时间大 1 或更多的值。在图 3-2 (b) 中，我们看到 C 现在到达的时间是 61。同样 D 到达的时间是 70。

为适应全局统一时间的需要，只需对此作一点补充即可，即在每两个事件之间，时钟必须至少滴答一下，如果一个进程以相当快的速度连续发送或接收两条消息，它必须在这中间至少滴答一下。

在某些情况下需要一个附加条件：两个事件不会精确地同时发生，为了实现这个目标，我们可以将事件所在的进程号附加在事件发生时间的低位，并用小数点分开。这样，如果在进程 1 和进程 2 中同时发生了 2 个事件，发生时刻都是 40，则前者记为 40.1，后者记为 40.2。

使用这种方法，现在已有一套在分布式系统中将时间分配给所有事件的方法，它遵照如下规则：

- (1) 若在同一进程中 a 发生在 b 之前，则 $C(a) < C(b)$ ；
- (2) 若 a 和 b 分别代表发送消息和接收消息，则 $C(a) < C(b)$ ；
- (3) 对所有事件 a 和 b ， $C(a) \neq C(b)$ 。

这个算法给我们提供了对系统中所有的事件进行排序的一种方法，许多其他的分布式算法也需要这种排序以避免混乱，所以本著作中广泛引用此算法。

3.1.2 物理时钟

虽然 Lamport 算法给出了一个明了的事件序列，但是分配给每个事件的时间值却不一定要与它们实际发生的时间接近，而在一些系统（实时系统）中，实际时钟时间很重要，对这些系统需要用到外部的物理时钟，为了提高效率和冗余性，一般使用多个物理时钟，但同时又有两个问题：(1) 我们如何用现实世界的时钟将它们同步起来；(2) 我们如何使各时钟之间保持同步。

在回答这些问题之前，先看一下实际中时间是如何测量的，它并不是你想像中的那么简单，当要求精度高时更是如此。自从 17 世纪发明了机械时钟以来，人们一直用天文学方法测量时间，每天太阳都是从东方地平线升起，升到天空中的最高处，再落到西边。太阳到达的最高点称为中天(tranist of the sun)，它在每天正午发生，两次连续的中天之间的时间称为太阳日(solar day)。因为每天 24 小时，每小时 3600 秒，所以精确地定义太阳秒为 $1/86400$ 个太阳日，太阳日的几何计算方法如图 3-3 所示。

在 19 世纪 40 年代，出现了地球自转周期并不是常数的学说，由于潮汐摩擦和空气阻力使地球自转速度减慢，在研究古珊瑚的生长图案后，地质学家认为在 3 亿年以前，每年为 400 天，而一年的时间也就是地球绕太阳一周的时间，它被认为是不变的，那么现在每天的时间就变长了，除了这种长期变化趋势，也存在一天长短的短期变化。这可能是由地球地核层熔岩的剧烈沸腾引起的。这些因素使天文学家通过测量很多天然日后对它们取平均值来计算天的长短，然后除以 86400 结果称作平均太阳秒(mean solar second)。

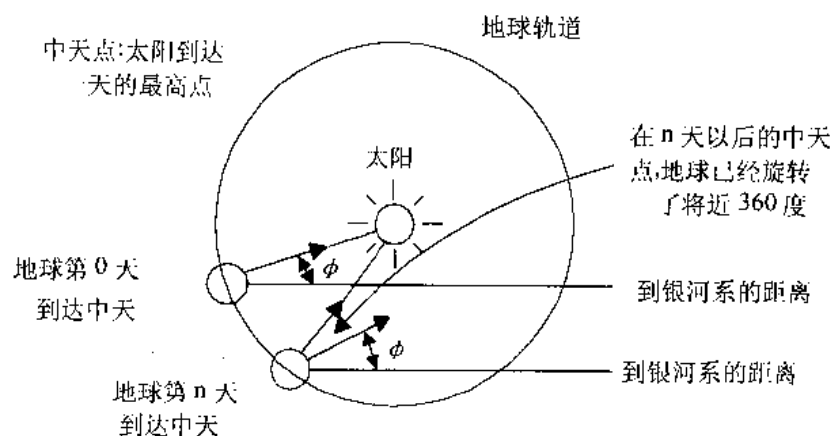


图 3-3 平均太阳日的计算

1948 年出现了原子时钟，它能够更精确地测量时间。原子时钟与地球的摆动和振动无关，而是通过铯 133 原子的跃迁计时。物理学家从天文学家的手中接管了计时工作，定义了秒是铯 133 原子作 9192631770 次跃迁所用的时间。选择 9192631770 是使原子秒与上面介绍的平均太阳秒相等。目前世界上约有 50 个实验室拥有铯 133 时钟，每个实验室都要定期地向巴黎的 BIH 报告它的时钟时间，BIH 将这些值平均起来计算出国际原子时间 (International Atomic Time)，简称为 TAI，这样 TAI 就意味着铯 133 时钟从 1958 年 1 月 1 日（午夜）以来被 9192631770 除后的平均滴答(tick)数。

尽管 TAI 相当稳定，并且任何人都可以去买一只铯时钟，但它仍然存在问题，86400 个 TAI 秒现在比一个平均太阳日少 3 微秒(因为平均太阳日会越来越长)，用 TAI 保持的时间意味着，多年以后，中午会出现得越来越早，直到出现在清晨，人们也许会注意到这一点。我们也可能会像 1582 年出现的情形一样，古罗马 13 世教皇发令将日历中的 10 天删除。这一事件导致了街头暴动，因为地主要收一整月的租金，银行家要付一整月的利息，而雇主拒绝向雇员支付他们没有劳动的那 10 天内的报酬。在一些信教城市，举行了长达 170 年的拒绝接受该法令的示威活动。

BIH 通过引入闰秒(leap second)解决了问题，当用原子秒计时和用太阳秒计时的差距增到 800 微秒时，使用闰秒，如图 3-4 所示。这种纠正使在基于连续的 TAI 秒时间系统加值，但却可以和太阳的运动保持一致，它称作统一协调时间，简称为 UTC。UTC 是所有现代人使用的时间，它基本上取代了老标准——格林尼治天文时间。

大多数电力公司是基于 UTC，用 60Hz 或 50Hz 时钟计时，因此当 BIH 宣布闰秒后，电力公司分别将时钟频率增加到 61Hz 或 51Hz，以便于在他们的分布式地区中时钟超前，因为在计算机中 1 秒是一个相当大的时间间距，为了保持精确的时间，操作系统在一年里必须用专用的软件计算闰秒。到现在为止，UTC 中引入闰秒的所有次数大约有 30 次。

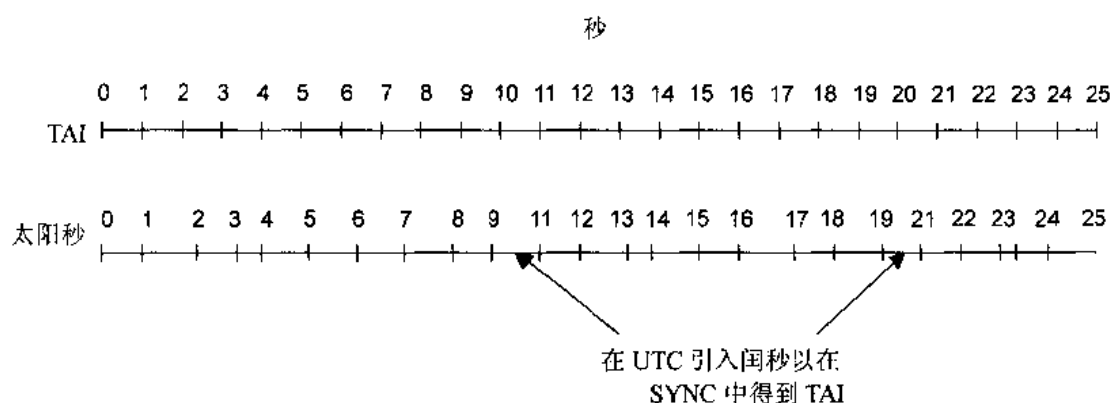


图 3-4 TAI 秒长度恒定，与太阳秒不同。为与太阳保持一致引入闰秒

为需要精确时间的人们提供 UTC，国际时间标准化组织（NIST）使用了一个在 Fort Collins, Colorado 的称为 WWV 自身的短波广播站。WWV 在每 UTC 秒的开始处广播一个很小的脉冲。WWV 精确度为 ± 1 毫秒，但由于空气起伏的随机性会影响到信号传输路径长度，实际精确度在 ± 10 毫秒。英格兰的 MSF 站在 Rugby, Warwickshire 提供了一个相同服务，一些其他的国家也有这样的站点。

几个地球卫星也提供 UTC 服务，地质环境操作卫星能提供精确到 0.5 毫秒的 UTC，还有的卫星甚至更好。

使用短波广播或卫星服务要求对发送者和接受者的相对位置有较精确的了解，这是为了对信号传输延迟进行补偿。在市场上可以买到接受 WWV、GEOS 及其他 UTC 源信号的广播接收器，其价格范围从几千元/套到几万元/套，也可以通过电话从 Fort Collins 的 NIST 得到更便宜的 UTC，但不是很精确。不过这里也需要对信号的传输延迟及 modem 的速度进行校正，这种校正产生了一些不确定性，使得很难获得精度较高的时间。

3.1.3 时钟同步算法

如果某一台机器上有 WWV 接收器，我们的目的是使其他所有机器与它同步。如果没有一台机器上有 WWV 接收器，每台机器按自己的机器时间计时，那么同步的目的就是尽可能地将所有机器放在一起。已经有很多算法是这样同步的（例 Cristian, 1989; Drummond 和 Babaoglu, 1993; Kopetz 和 Ochsenreiter, 1987），在（Ramanathan et al., 1990b）中已给出其述评。

所有算法都有相同的系统基础模型，我们将描述如下。每台机器上假设都有一个每秒产生 H 次中断的计时器。当时间到时，中断处理程序将软时钟加 1，软时钟记录从过去某一约定值开始的中断次数。我们将这个值记为 C 。更特殊的，当 UTC 时间为 t 时，在机器 P 上的时间值是 $C_p(t)$ ，最完美的情况是对所有的 p 和 t ，有 $C_p(t)=t$ ，换言之， dC/dt 理想值为 1。

真正计时器并不是每秒精确的中断 H 次，理论上当 $H=60$ 时，计时器每小时应该产生 216000 个滴答。实际上，用现代计时时钟芯片可以获得相关的延迟是 10^{-5} ，意味着，

一台机器每小时可以获得 215998~216002 范围的滴答，若存在某一常数 ρ ，则有：

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

计时器可以在它规定的范围内工作，制造商标明的常数 ρ 是最大漂移速度。稍慢的、精确的、稍快的时钟如图 3-5 所示。

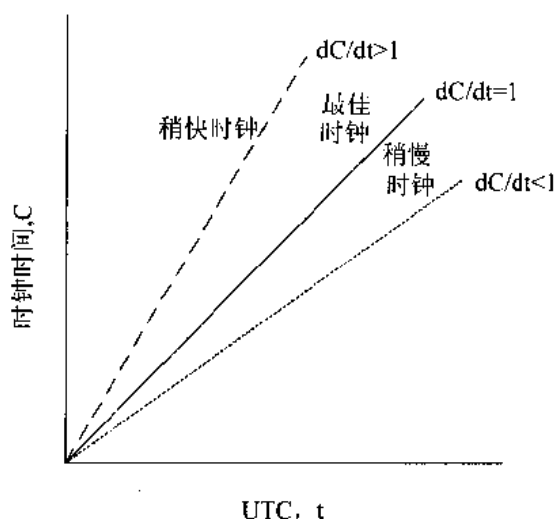


图 3-5 不是所有的时钟都按正确的速率中断（滴答）

若两个时钟相对于 UTC 时间以相反方向漂移，在它们同步后的 Δt 时间内，它们可能的差值为 $2\rho \Delta t$ ，若操作系统的设计者要保证每两个时钟之间的相差不超过 δ ，时钟必须至少在每 $\delta/2\rho$ 秒内再同步一次（用软件方法），不同算法实现再同步的方法不同。

Cristian's 算法

现在开始讨论一种算法，它非常适合于只有一台机器上有 WWV 接收器，其他所有机器与它同步的系统。将拥有 WWV 接收器的那台机器称作时间服务器(time server)，算法是基于 Cristian(1989)和以前的一些工作，每台机器以小于或等于 $\delta/2\rho$ 秒的周期定期地向时间服务器发送消息询问当前的时间，时间服务器接到消息后就尽快回答含有当前时间 C_{UTC} 值的消息，如图 3-6 所示。

第一种近似情况是当发送者得到回答后，就将它的时钟设为 C_{UTC} ，但是这种算法有两个问题，一个很重要，一个次要。第一个重要的问题是时间决不能倒退，若发送者的时钟快， C_{UTC} 将会比发送者的时间值 C 小，若将发送者的时间值直接改成 C_{UTC} 会导致严重的错误，比如在时钟变化后，编译产生的目标文件的时间早于时钟变化前源文件的修改时间。

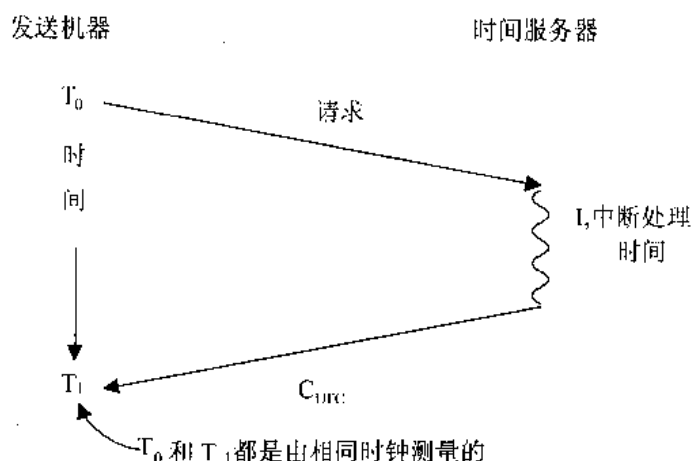


图 3-6 从一个时间服务器得到当前时间

这种变化必须逐步进行，一种方法是假设将计时器设置为每秒产生 100 次中断，通常，每次中断将时间加 10 毫秒，当时钟需要慢下来时，中断服务程序每次仅加 9 毫秒，直到调整好为止。同样时钟要加快时，每次中断服务程序将时钟加 11 毫秒，而不是立即把时间调到所需要的值。

另一个次要问题是从时间服务器端发送的应答到发送者要花费时间，然而不幸的是这种延迟可能较大，而且随着网络负荷的改变而改变。Cristian 的处理方法是试图测量这个延迟值，最简单的方法是发送者精确地记录从向时间服务器发送请求到接收到应答的时间间隔，假设起始时间是 T_0 与结束时间 T_1 ，他们都是通过同一个时钟来测量的，就算发送者的时钟与 UTC 有一定的差值，它所测得的时间间隔还是较精确的。

在没有其他任何信息时，消息传送时间的最佳估计值是 $(T_1 - T_0) / 2$ ，当应答消息到达时，消息中的时间值再加上此值就得到了当前时间服务器的时间估计值，如果理论上知道了最小的传送时间，那么与时间估算相关的其他性质也能推算出来。

如果知道时间服务器中断处理的时间和处理消息的时间，这样的估算还能进一步改进，设中断处理的时间是 I ，那么传输时间间隔为 $T_1 - T_0 - I$ ，所以估算单向传输时间为它的一半，系统中确实存在着从 A 到 B 的消息和从 B 到 A 的消息传输路径不同，因此就有不同的传输时间，但我们暂时不考虑这种情况。

为了提高精确度，Cristian 建议不要只做一次测量，而要做一系列测量，测量中 $T_1 - T_0$ 超出一定范围就认为是网络阻塞，为不可信值。对剩余的测量值取平均值会得到较好的估算值，也就是说，最快返回的消息是最精确的，因为消息遇到阻塞最少，所以它最能代表纯粹的传输时间。

Berkeley 算法

在 Cristian 算法中，时间服务器是被动的，其他机器定期地向它询问时间，它所做的就是回答它们的请求，在 Berkeley UNIX 中采取了完全相反的方法，这里的时间服务器（实际是时间守护进程）是主动的，它定期地询问每台机器的时间。然后基于这些回答，计算

出平均值并告诉所有的机器将它们的时钟拨快或拨慢到一个新的值。这种方法，适合于没有 WWV 接收器的系统，时间守护进程的时间必须由操作者定期地手工设置，这种方法如图 3-7 所示。

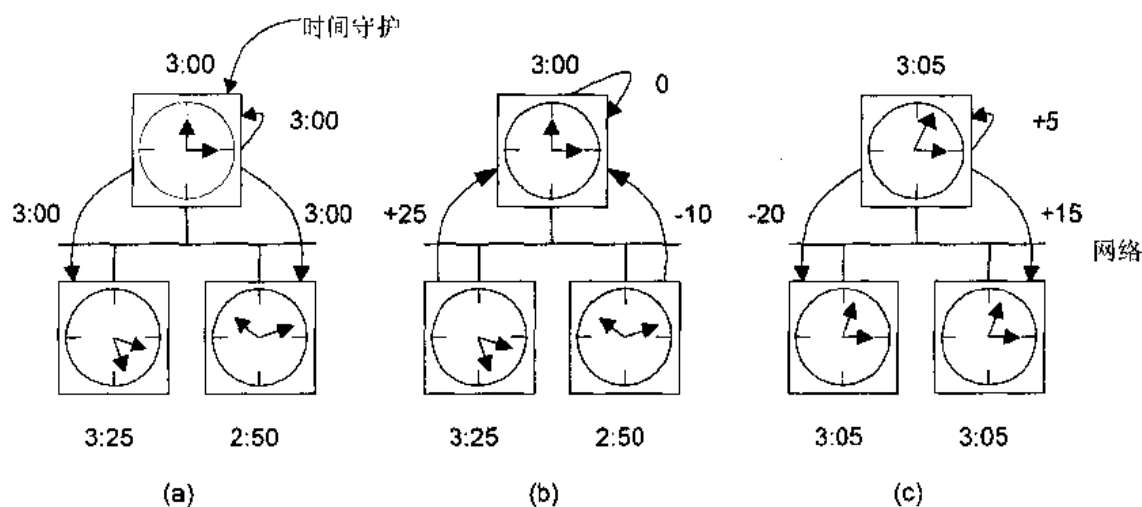


图 3-7 (a) 时间守护进程询问其他机器的时间值
(b) 机器应答
(c) 时间守护进程通知每个机器怎样调整时间

在图 3-7(a)中，时间守护进程在 3:00 把它的时间告诉其他机器，并且询问它们各自的时间，在图 3-7(b)中，各机器将它们各自的时间与时间守护进程时间的差值告诉时间守护进程，然后，守护进程计算出它们的平均值，通知各机器如何调整各自的时间，如图 3-7(c)所示。

平均值算法

上述两种方法都是高度集中的，有它不足之处。存在一些非集中式算法，一种分布式时钟同步算法，它是将时间划分成固定长度的再同步间隔，第 i 次间隔开始于 $T_0 + iR$ ，而结束于 $T_0 + (i+1)R$ ，这里的 T_0 是过去某一约定的时间， R 是一个系统参数。在每次间隔的开始处，每台机器根据自己的时钟广播发送当前的时间，由于在不同机器上的时钟不可能完全同速工作，这种广播就不会完全同时发生。

在机器广播发送时间之后，它启动本地计时器收集在 S 时间间隔中到达的其他广播，当所有广播到达后，执行一个算法，得到新的时间值。最简单的算法是仅将这些值取平均值，稍微不同的另一种算法是先除去 m 个最大值和 m 个最小值，平均其余值。去掉两端值可认为是一种对 m 个错误时钟发出毫无意义的时间值的一种自我保护。

另一种改进是给每条消息值加上一个从源到目的地的传送时间估计值，这种估计值可参考网络的拓扑结构或计算试探消息的响应时间而得出。

其他一些时钟同步算法在以下书中讨论：

(例如：Lundelius-welch 和 Lynch,1988;Ramanathan et al. ,1990a; 和 Srikanth Toueg,1987)。

多个外部时钟源

对使用 UTC 进行同步又要求特别精确的系统来说, 可给系统安装如 WWV, GEOS 或其他 UTC 源的多接收器来实现的。但是, 因为时间源自身固有的不精确性以及信号传送的不定性, 最好的操作系统能做的也只是建立一个 UTC 失效范围。一般来说, 不同的时间源会产生不同的失效范围, 这种范围需要机器和它们达成一致。

为了达到一致, 每个具有 UTC 源的处理机定期地在每一 UTC 精确分的开始处广播它的时间范围, 但这样做没有处理机会同时获得时间包, 而且, 传输和接收延迟依赖于缆线的距离和包必须经过的网关数, 它对于每一对 UTC 源和处理机之间都不相同。还有一些其他因素, 如多台机器要同时在以太网上传输而发生的碰撞。更进一步, 如一台处理机忙于处理以前的包, 它可能在相当长的几微秒内不去理会到来的时间包, 从而导致了时间的不确定性。在第 10 章我们将研究在 OSF 中 DCE 时钟是如何同步的。

3.1.4 使用同步时钟

最近才出现了用于大规模系统(如整个 Internet)上同步时钟所需要的硬件和软件, 使用这种新技术, 使得上百万个时钟能够在几个 UTC 毫秒内同步。同时也开始出现同步时钟的新算法, 下面将总结出由 Liskov 在 1993 年讨论的两个例子。

至多一次消息传递

第一个例子是如何强迫至多一次将消息传递到服务器上, 即使系统面临崩溃也是如此。传统的方法是每条消息都带有唯一的消息号, 每个服务器存储它所见过的所有消息号, 这样它就可以检测出是新消息还是重发消息。此算法的问题是如果服务器故障重启, 它将丢失消息号信息表。同样, 应该每隔多长时间将消息号存储一次?

利用时间, 可以把算法修改如下: 每条消息都携带一个连接标识(由发送者选择)和时间戳, 每次连接时, 服务器记录它所见到的最近一次的时间戳, 如果接收到的要求连接的消息其时间戳小于服务器记录的时间戳, 则认为这条消息是已复制过的而被拒绝。

可以移走旧的时间戳, 但每台服务器上要一直保存一个全局变量:

$G = \text{当前时间} - \text{最大生存时间} - \text{最大时钟偏移}$

这里最大生存时间指一条消息能够存在的最长时间, 最大时钟偏移指允许时钟与 UTC 偏移的最大值, 任何比 G 早的时间戳都能安全地从表中移走, 即所有老的消息都已消失了。如果接收到的消息是一个不可知的连接标识符, 且它的时间戳比表中的新, 则接收该消息, 反之则拒绝。实际上, G 是所有老消息的消息数目的梗概。每隔一定时间 ΔT , 将当前的时间写入磁盘。

当服务器故障重启时, 从磁盘上重新装载 G , 再加上 ΔT 。这样任何时间戳早于 G 的消息由于已复制过而被拒绝接收。结果可能在故障前接收的消息在出现故障后就被拒绝了。而一些新的消息, 也许被不正确地拒收, 但是在所有情况下, 算法都保持了至多一次的思想。

基于时钟的缓冲存储器的一致性

第二个例子是考虑到在分布式文件系统中缓冲存储器的一致性。由于执行的原因,

客户方能够在本地缓冲文件是最理想的，但是，如果两个客户在同一时间修改同一文件会使缓存产生潜在的 inconsistency，通常的解决方案是区分为读缓存文件还是为写缓存文件。此方案的不足之处是如果一个客户有一个为读而缓存的文件，在另一个客户得到为写而缓存的拷贝前，服务器必须先通知读方，使其拷贝无效，即使这个拷贝是几小时前做的。这种额外开销只需使用几个同步时钟即可完成。

基本方法是当客户端需要一个文件时，给它一个标注有拷贝有效期的租约 (Gray 和 Cheriton, 1989)，当租约期满时，客户可以续租。租约到期时，就不能再使用缓存的拷贝。使用这种方法，当客户需要立即读文件时，它就可以申请。当租约使用期满时，仅仅是使用时间到，不需要发送消息告诉服务器，此时拷贝已经从缓冲存储器中删除了。

如果租约已经到期，仍然想用缓冲存储器中的文件，客户可询问服务器，该文件的拷贝 (标有戳) 是否仍然存在，如果存在，则生成新的租约，但不需再传输文件。

如果一个或多个客户有为读而缓冲的文件，而另一个客户要写该文件，服务器将不得不请求读方提前终止其租约。若一个或多个客户方有故障，服务器将等待一定时间，直到服务器租约到期。在传统算法中，要求从客户方到服务器方必须有一个明确的“允许使用缓冲存储器”的答复，如果服务器要求客户返回文件，或客户已返回文件而对方无反应，意味出现了故障。这时服务器不知道是客户方崩溃了还是反映太慢。为了解决这个问题，使用基于时钟的算法，服务器等待到租约到期为止。

除了这两种方法，Liskov 在 1993 年指出了在分布式系统领域，处理原子事务方面如何使用同步时钟计时。随着同步计时器的发展，毫无疑问将会出现新的应用。

3.2 互斥

涉及多个进程的系统使用临界区容易编程。当一个进程必须读或修改某些共享数据结构时，它首先进入临界区获得互斥锁，保证没有其他的进程能同时使用该共享数据结构。在单处理机系统中，使用信号量(semaphores)、管程(monitors)和一些近似的结构来保护临界区。我们看几个例子，在分布式系统中临界区和互斥是如何实现的。作为其他方法的分类和书目请看 (Raynal, 1991)，其他工作在 (Agrawal 和 El Abhadi, Chandy et al., 1983; 和 Sanders, 1987) 中讨论。

3.2.1 集中式算法

在分布式系统中获得互斥的最直接方法是仿照单处理机系统的方法，选一个进程为协调者 (比如在最大网络地址机器上运行的进程)。无论什么时候进程要进入临界区，它将向协调者发送请求消息，说明它想进入哪个临界区并希望获得允许。如果当前该临界区内没有其他任何进程，协调者就发送允许进入消息，如图 3-8(a)所示。当应答到达时，请求者就可以进入临界区。

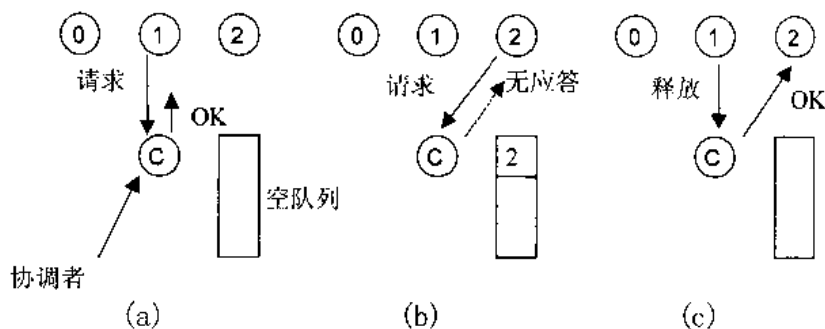


图 3-8 (a) 进程 1 请求协调者允许它进入临界区，请求被应答
(b) 进程 2 请求协调者允许它进入临界区，协调者不应答
(c) 进程 1 退出临界区，它通知协调者，协调者应答进程 2

现在假设有另一个进程，如图 3-8(b)所示，请求进入同一个临界区，协调者知道该临界区已有一个进程，所以不能同意该请求，最好的办法是发出拒绝允许应答。而在图 3-8(b)中，协调者回避应答，这样就阻塞了进程 2，使它等待应答。另一方面，协调者还可以发送“拒绝请求”应答，把进程 2 的请求临时排在队列中。

当进程 1 从临界区退出时，它向协调者发送释放互斥消息访问，如图 3-8(c)所示。协调者从推迟请求队列中取出最前面的进程，向它发送允许进入消息。如果该进程仍然阻塞（即，这是第一条发给它的允许进入消息）它不被阻塞且进入临界区。如果明确发送一消息拒绝它进入临界区，此进程应该查询输入的消息，或者接着将它阻塞，不管怎么样，当它发现允许进入时，它还可以进入临界区。如果进程在阻塞状态，它就会被唤醒进入临界区。

可以看到这个算法保证了互斥的实现，协调者仅能让某一进程在某一时刻进入临界区。它也很公平，因为允许请求的顺序同它们接收的顺序一致，没有进程永远等待（没有饥饿）。这种方案也容易实现，每用一次临界区只需 3 条消息（请求，允许，释放），它不仅管理临界区，也可用于更普遍的资源分配。

集中式方法也存在缺点。即协调者是一个单点故障，如它崩溃，整个系统将瘫痪。如果进程在请求之后被阻塞，以上这两种情况都没有消息返回，请求者不能从“拒绝请求”中辨认出协调者已崩溃。此外，大系统中单协调者会成为执行的瓶颈。

3.2.2 分布式算法

拥有单点故障的情况往往是不可接受的，所以研究者寻找到了分布式互斥算法。第一次出现的是在 1978 年 Lamport 关于时钟同步的论文中，后来 Ricart 和 Agrawale (1981) 对它作了进一步的改进，本章将描述他们的算法。

Ricart 和 Agrawale 算法要求系统中所有事件都是全序的。也就是说，对任何事件组如消息，哪个先发生必须无歧异。在前面的 3.1.1 节中 Lamport 算法是获得这种排序的一种可行方法，还能用来为分布式的互斥提供时间戳。

算法如下。当一个进程想进入临界区时，它要建立一个包括它要进入的临界区的名字、处理机号和当前时间的消息，然后将消息发送给所有其他进程。概念上讲也包括发送给它自身，发送的消息假设是可靠的，也就是，每条消息都应该被确认。如可能应该使用可靠的组通信，避免使用单个的消息通信。

当一个进程接收另一个进程请求消息时，它取决于接收方的状态以及临界区的命名。有三种情况要加以区别：

- (1) 若接收者不在临界区中，也不想进入临界区，它就向发送者发送 OK 消息；
- (2) 若接收者已经在临界区中，它就不必回答，而是负责对请求队列排队；
- (3) 若接收者要进入临界区，但是还没有进入时，它要将发来的消息和它发送给其余进程的时间戳对比，取小的那个。如果来的消息的时间戳小，接收者发送 OK 消息。如果接收者本身的时间戳更小，那么接收者负责排列请求队列而不发送任何消息。

在发送完允许进入临界区的请求后，进程将不做任何事，仅等待所有的允许消息，一旦得到允许，它就进入临界区。它从临界区退出时，向队列中的所有的进程发送 OK 消息，并将它从队列中删除。

让我们来理解一下为什么要这样做。如果没有冲突，则正常工作。但是，假设两个进程要同时试图进入一个临界区，如图 3-9(a)所示

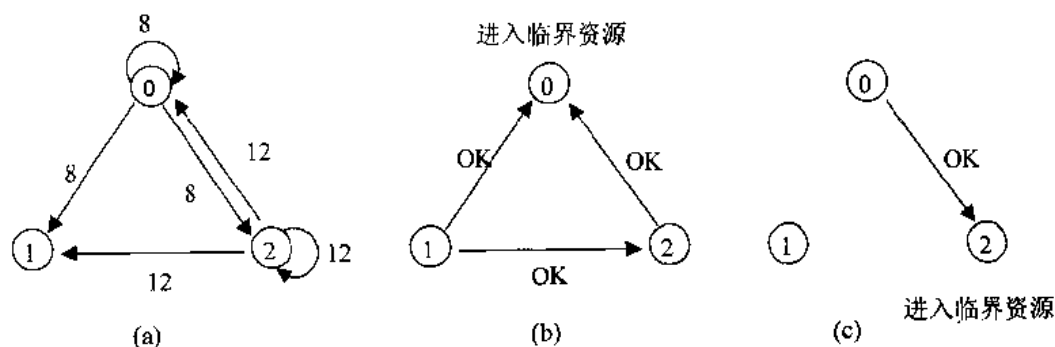


图 3-9 (a) 两个进程在同一时刻进入同一个临界区
(b) 进程 0 的时间戳小，所以它获胜
(c) 当进程 0 完成时，它发送消息 OK，进程 2 现在可进入临界区

进程 0 发出时戳为 8 的请求，而同时进程 2 发出时戳为 12 的请求。进程 1 不想进入临界区，所以它向 2 个发送者发回 OK。进程 0 和 2 同时发现冲突，比较时戳，进程 2 发现自己的大，它只好同意进程 0 进入临界区，向 0 发送 OK，进程 0 就把进程 2 的请求排在队列中，为以后处理用，自己进入临界区，如图 3-9(b)所示。当进程 0 结束退出临界区时，它将进程 2 的请求从队列中取出，并向进程 2 发送 OK，允许进程 2 进入临界区，如图 3-9(c)所示。算法之所以能够使用，因为在产生请求冲突时，遵守按时间戳排序，小时戳优先的规则。

请注意图 3-9 的情况，如果进程 2 比进程 0 早发送消息，进程 0 在它请求进入临界区之前获得了进程 2 的请求消息，它允许进程 2 进入临界区，当进程 2 接收到进程 0 的请求

时，它发现自己已经在临界区了，所以把进程 0 排在等待队列中。

前面讨论的集中式算法，互斥保证了无死锁或无饥饿，每次进入临界区需要 $2(n-1)$ 条消息，这里 n 为系统中的进程数目。最好的是不存在单点故障。不幸的是，这里单点故障被 n 个失效所取代。若有一个进程崩溃，它就不能回答请求。这种不发言将被解释为不允许进入临界区，这样就阻止了任何试图进入临界区的进程，因为 n 个进程之一失效的可能性是单协调者失效的 n 倍，我们必须设法换一种 n 倍复杂和需要更多启动网络通信的算法。

此算法可用我们前面提议的同样技巧修补。当请求到达时，不管它是许可还是拒绝，接收者都要发送应答。一旦请求或应答消息丢失，发送者的等待时间到，它继续发送直到得到应答或者认为目的进程已经崩溃为止。在收到拒绝应答后，发送者应该阻塞等待直到获得 OK 消息。

该算法的另一个问题是要么组通信必须使用原语，要么每个进程必须维持一张组成员表，包括进入组进程、离开组进程和崩溃进程。这种方法最适用于小的从不改变成员的进程组。

最后，回顾集中式算法的一个问题，即处理所有请求会产生瓶颈的问题，在分布式算法中，所有进程都要参与决定是否进入临界区，若一个进程不能这样做，不可能强迫其他进程也这么做。

对该算法有一点小的改进。比如，不需获得每个进程允许后方可进入临界区，只要防止任何两个进程同时进入同一个临界区即可。因此，算法可改成：当一个进程从大多数进程那里获得允许而并不需要所有进程都允许时，它就可以进入临界区。这种算法中，一个进程在批准另一个进程进入临界区之后，在这个进程退出临界区之前，它不能再允许其他进程进入该临界区。

总之，这种算法仍然比原来集中式算法慢、复杂、昂贵，而且不健壮。

为什么要在这些情况下讨论它们呢？原因之一是它说明了我们起初不太清楚的一点，也就是至少分布式算法是可能实现的这一点，而且通过指出它们的不足，我们可以成为未来的理论家来完善它们的实现。其次，用抽象的方法对人的思维是有好处的。

3.2.3 令牌环算法

图 3-10 表示在分布式系统中获得互斥的一种完全不同的方法。如图 3-10 (a) 所示的总线网（以太网），进程没有固定次序。用软件的方法，构造出一个逻辑环，环中每个进程都有一个位置，如图 3-10(b)所示。环的位置可以用网络地址或其他理由方法得到一个序列，总之，每个进程都要知道谁在它的下一个位置。

令牌环被初始化后，进程 0 首先获得令牌。这样，令牌开始绕环运动。它从进程 k 传递到进程 $k+1$ ，当然是以点到点的方式传递的。当进程从它的前一个邻居手中得到令牌时，它要检查它所试图进入的临界区。如果该临界区是空的，则该进程可以进入临界区，做它要做的工作，然后离开临界区。它退出后，将令牌传递到它的下一个邻居，不允许使用同一令牌进入第二个临界区。

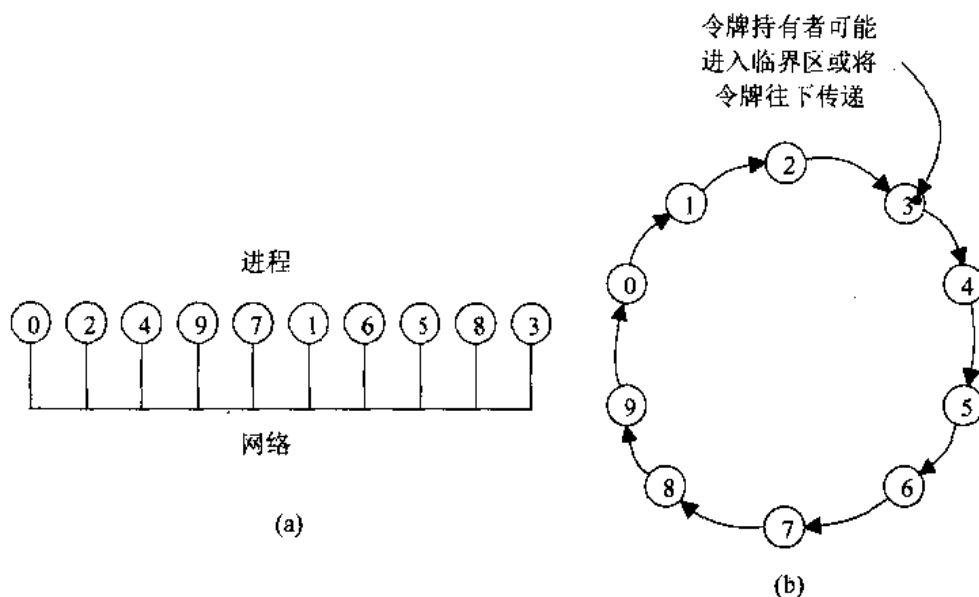


图 3-10 (a) 网络中一组未排序的进程
(b) 用软件构造进程的逻辑环

若一个进程得到了它相邻进程传递来的令牌，但它并不想进入临界区，就将该令牌往下传递。这样，如果没有任何其他的进程想进入临界区时，令牌就会高速地沿环绕行。

这种算法的正确性是显而易见的，在任何时刻只有一个进程拥有令牌，所以只有一个进程可以进入临界区。由于令牌以固定顺序运动，所以不会出现饥饿进程。一旦一个进程想进入临界区，最不理想的情况是等待所有其他进程进入后再退出临界区所用的时间之和。

这种算法也存在一些问题，比如，令牌一旦丢失，它必须重新生成。实际上，检测令牌丢失是很困难的，因为在网络上，令牌两次出现的时间是不定的，一个小时没有发现令牌并不意味着它丢失了，也许某个进程还在使用它。

当进程崩溃时，该算法也会出现麻烦，但是恢复却容易得多。如果我们需要进程在接收到令牌后发回确认消息，当相邻进程试图传递给它令牌但却没有成功时，它就会检测到死进程。这时就将死进程从进程组中移出。它的下个进程就会从令牌持有者的手中接收到令牌。当然，这样做需要每个进程都能维持当前环设置情况。

3.2.4 三种算法的比较

简略地比较一下这三种互斥算法是很有意义的。在表 3-1 中，我们列出了算法和三种主要性质：进程进入或退出临界区需要的消息数目，每次进入前的延迟，一些与算法有关的问题。

表 3-1 三种互斥算法比较

算法	每次进出需要的消息	进入前的延迟 (按消息次数)	问题
集中式	3	2	协调者崩溃
分布式	$2(n-1)$	$2(n-1)$	任何一个进程崩溃
令牌环	1 到无穷大	0 到 $n-1$	丢失令牌,进程崩溃

集中式算法最简单,也最有效的,它在进程进入和退出临界区时,只用了 3 条消息,即请求进入,同意进入和退出时释放。分布式算法需要 $n-1$ 条,(给每个其他进程)请求消息,及另外的 $n-1$ 条允许消息,总共有 $2(n-1)$ 条消息。对于令牌算法该数目是可变的,如果每个进程总是想进入一个临界区,则令牌的每次传递将导致一次进出临界区。平均每一次进入临界区入口都有一条消息。在其他极限情况下,令牌也许将几小时几小时地沿环传递而没有进程想使用它,在这种情况下,每一次进入临界区出入口消息数是无界的。

进程需要进入临界区到它能进入临界区的延迟也随着这三种算法而变化。当临界区很少使用时,造成延迟的主要因素是进入临界区的机制。当临界区经常使用时,延迟的主要因素是等待其他进程使用的过程。在表 3-1 中,我们表示了前一种情况,假设网络在某时刻只能处理一条消息,在集中式算法的情况下,进入临界区只需要处理 2 条消息的时间,在分布式算法的情况下需要 $2(n-1)$ 条消息,对令牌环算法,时间从 0 (刚接收到令牌)到 $n-1$ (释放令牌)变化。

最后,这三种算法在进程崩溃时都损失惨重。为了避免由于某些崩溃造成的系统瘫痪,必须引入专门的测量和附加的复杂方法。在分布式系统对方损坏比在集中式的更敏感。在容错系统中,这些方法都不适用,这一点似乎可笑,但是如果崩溃不经常发生,还是可以接受的。

3.3 选举算法

许多分布式算法需要一个进程充当协调者、发起者、排序者或其他特定的角色。我们已经看到一些例子。例如,在集中式互斥算法中的协调者。通常,选择哪个进程充当协调者并不重要,重要的是要有进程负责。本节,我们将了解选举协调者的算法。

如果所有进程的地位都相同,没有特性上的区别,就无法选择其中一个为特殊进程。现在,我们假设每一个进程有一个特殊的号码,比如它的网络地址(为简单起见,我们假设每台机器只有一个进程),通常选举算法总是找拥有最大号码的进程,将它指定为协调者,各算法在选举时有不同的方法。

下一步,我们假设每个进程都知道所有其他进程的进程号,但不知道目前哪些进程正常,哪些进程不正常。选举算法的目的是确定选举何时开始,它在所有进程都同意的基础上选出协调者。

3.3.1 欺负 (Bully) 算法

选举算法的第一个例子是欺负算法：它是由 Garcia-Molina(1982)提出的，当一个进程发现协调者不再响应请求时，它就发起选举。进程 P 负责选举如下：

- (1) P 向所有号码比它大的进程发送选举 (ELECTION) 消息；
- (2) 若无人响应， P 获胜成为协调者；
- (3) 若有号码比它大的进程响应，响应者接管， P 的工作完成。

在某一时刻，一个进程只能从号码比它小的进程那里得到一个选举 (ELECTION) 消息，当它到达时，接收者就发送回 OK 消息，表明它的存在并接管，然后接收者主持选举（除非它正在主持别的选举）。除了一个进程外的其余进程都得放弃，这一个进程就是新的协调者，它将选举获胜的消息发送给所有进程，告之它是新的协调者。

若一个进程刚刚崩溃过，而又得到恢复，它主持选举，若它刚好是当前运行进程中号码最大的，它就会获得选举的胜利，从而接管协调者的工作，所以最大的进程总能取胜，故而将该算法命名为欺负算法。

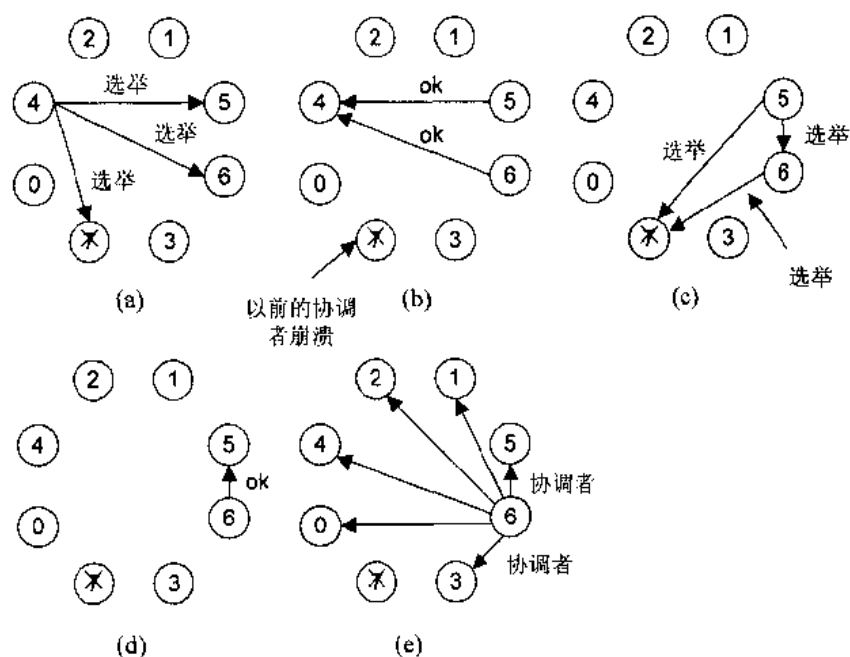


图 3-11 欺负选举算法

- (a) 进程 4 主持选举
- (b) 进程 5 和进程 6 应答，通知进程 4 停止
- (c) 现在进程 5 和 6 分别主持选举
- (d) 进程 6 通知进程 5 停止
- (e) 进程 6 获胜并通知所有进程

在图 3-11 中，我们看到了欺负算法是如何工作的例子。一组由 0~7 号的共 8 个进程

组成，开始 7 号进程是协调者，但是它突然发生了故障，进程 4 第一个注意到这一点，所以它向所有比它进程号大的进程，即进程 5, 6, 7 发送选举消息，如图 3-11(a)所示。进程 5 和 6 接收消息后发送回 OK 消息，如图 3-11(b)所示。进程 4 接收到第一个应答时就知道自己的工作已经结束了，因为已经有比它进程号大的进程即将接管它的工作成为新的协调者，它就等待看谁将在选举中获胜。

在图 3-11(c)中，进程 5 和 6 都主持选举，每个进程仅把消息发送给比自己进程号大的进程，在图 3-11(d)中，进程 6 告诉进程 5 它将成为协调者，而这个时候进程 6 知道进程 7 已经死了，而且它将是获胜者。如果从磁盘或其他地方无法得到一些状态信息表明原来的协调者在哪里失效，进程 6 则要做这部分工作。当它准备接管时，进程 6 向所有的运行进程发送 *COORDINATOR* 协调者消息，进程 4 接收到这条消息后，发现进程 7 已经死了，进程 6 是新的协调者，进程 4 就可以继续它刚才所尝试进行的工作，这样，进程 7 的失效得到了处理，而且工作又得到了继续。

如果进程 7 重新启动，它就会向所有的进程发送协调者消息，使它们受到欺负。

3.3.2 环算法

另外一种选举算法是基于没有令牌的环，假设所有的进程是按物理或逻辑排序的，这样每一个进程都知道谁是它的后继者。当任何一个进程发现协调者不再起作用时，它就构造一个包含它自身进程号的选举消息发送给它的后继者，如果后继者失效，消息将绕过后继者，到达后继者的后继者，或者再下一个，直到找到一个运行进程。每次，发送者都将自己的进程号加入到消息表中。

最后，消息到达了始发者的手中，始发者接收到包括自己进程号的消息后，将消息的类型转化为协调者消息，该消息将再一次绕环运行，向所有的进程通知谁是协调者（在成员表中过程号码最大的那个）和新的环成员。当消息循环一周后，被去掉，每个进程都恢复工作。

在图 3-12 中我们看到如果进程 2、5 同时发现前任协调者进程 7 失效时，它们各自建立一个选举消息沿环发送，最终，两条消息都将沿环运动，进程 2 和进程 5 分别将它们转化成协调者消息，消息中有完全一样的成员，相互顺序也相同，当两条消息再绕环一周时，都被去掉，有多余的消息循环没有坏处，最多是浪费了一点带宽。

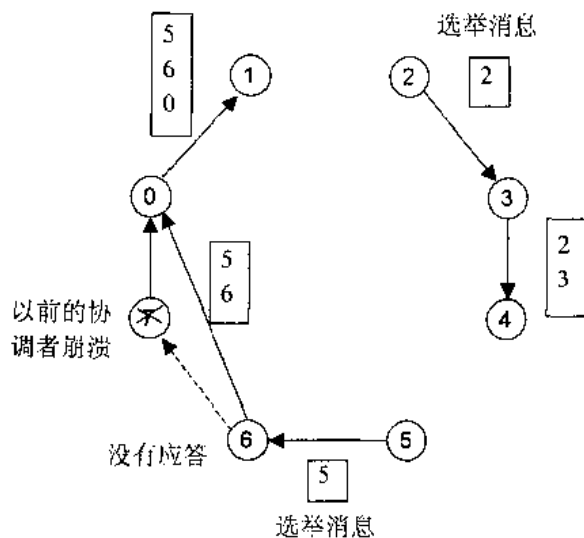


图 3-12 使用环的选举算法

3.4 原子事务

迄今为止我们研究的所有同步技术在其本质上说都是处于底层的,比如信号量。这些技术需要编程人员密切注意互斥、临界区管理、死锁预防、崩溃恢复等问题的细节。而我们真正喜欢的是更高层次的抽象,也就是要隐藏这些技术问题,允许编程人员将精力集中在算法和进程如何并行运行上。这样的抽象是存在的,而且被广泛应用在分布式系统中。我们称其为原子事务,或简称为事务。术语原子操作也被广泛使用。本节将涉及原子事务的使用、设计和实现。

3.4.1 原子事务简介

原子事务的最初模型来源于商业社会。假设 International Dingbat 公司需要一批装饰品,他们与潜在的供应商(因产品质量而远近闻名的美国 Widget 公司)进行了联系,希望 6 月份能交付 10 万件 10 厘米的紫色装饰品。Widget 公司提出 12 月份交付 10 万件淡紫色装饰品。Dingbat 公司同意对方开出的价格,但不喜欢淡紫色,并且希望 6 月份到货,而且因为自己的客户是国际客户,因此坚持要 10 厘米的产品。Widget 公司答复说 10 月份提供 3 15/16 英寸的淡紫色装饰品。经过更进一步的谈判,双方最终同意 8 月 15 日交付 3 959/1024 英寸的紫罗兰色装饰品。

到此为止,双方就可以自由中断本次讨论了,这样就返回到了开始谈判前的状态。然而,一旦公司双方签定了合同,那么不论发生什么事情,他们在法律上都有责任完成该合约。因此在双方还未签字之前,任何一方都可以反悔,就像什么都没有发生一样,但是一旦双方都签了字,他们就不能再反悔,事务就必须被执行。

计算机模型有些类似。一个进程宣布它想同其他的一个或几个进程开始一个事务。

它们可以就不同的选择进行协商、创建、删除对象，执行一段时间的操作。然后发起者宣布它希望其他进程能保证任务完成。如果其他进程都同意，那么就达成了永久的协议。如果有一个或几个进程拒绝（或在同意前崩溃），那么就会返回到事务开始前的状态。这时对象、文件、数据库等方面的副作用都会神奇地消失。这种要么全有要么全无的特性简化了编程人员的工作。

计算机系统中对事务的使用可以回溯到 20 世纪 60 年代。在硬盘和在线数据库出现之前，所有的文件都保存在磁带上。设想有一个有自动盘点系统的超级市场，每天关门后，计算机对两盘作为输入的磁带进行处理，第一盘磁带存有当天早晨开门以前的所有库存，第二盘存有当天的更新列表：已销售给客户的产品和供应商交付的产品。计算机从两盘磁带上读取数据，并生成新的主库存磁带，如图 3-13 所示。

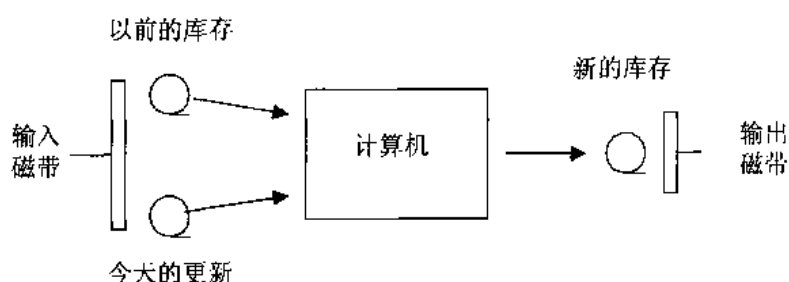


图 3-13 更新主磁带是容错的

这种设计的最大优点（尽管与它生活在一起的人们并没有意识到）在于对任何原因引起的运行错误，所有的磁带都可以倒卷(rewound)，其工作可以毫无损失地重新开始。因此老式的磁带系统具有了原子事务要么全有要么全无的特性。

现在来看一个当前如何在线更新数据库的银行应用的例子。客户通过带有调制解调器的 PC 机连接到银行，想将一个帐户下的钱取出再存入另一个帐户。

操作通过下面两步执行：

- (1) 提取（金额，帐户 1）；
- (2) 存入（金额，帐户 2）。

如果电话连接在第一步之后第二步之前中断，那么第一个帐户已被取出而第二个帐户却没有存入。钱就消失在了未知的空间中。

将这两个操作组成一个原子事务可以解决这个问题。要么两个都执行，要么一个也不执行。关键是事务执行失败后能返回到最初状态。我们真正需要的是像使用磁带时那样的对数据库倒卷的方法。这种能力是原子事务必须提供的。

3.4.2 事务模型

现在将就什么是事务以及它的属性是什么提出一个更精确的模型。假设系统由一些相互独立的进程组成，每个进程都会随机出错。通信一般来说是不太可靠的，消息会丢失。但是底层可以采用超时重发协议恢复丢失的消息。因此我们可以假定通信错误已经被底层

软件透明地处理了。

稳定存储器

存储器有三种分类。第一种是普通的 RAM 存储器，当电源出错或机器崩溃时会丢失信息。第二种是磁盘存储器，它不受 CPU 错的影响，但磁头错会导致信息丢失。

最后一种是稳定存储器(stable storage)。它不受洪水地震之类大灾难之外的任何其他错误的影响。稳定存储器(stable storage)可以通过两个普通磁盘来实现。如图 3-14(a)所示。驱动器 2 上的每一块是驱动器 1 上相应块的精确拷贝。当更新一个块时，首先更新并检查驱动器 1 上的块，然后驱动器 2 上的块。

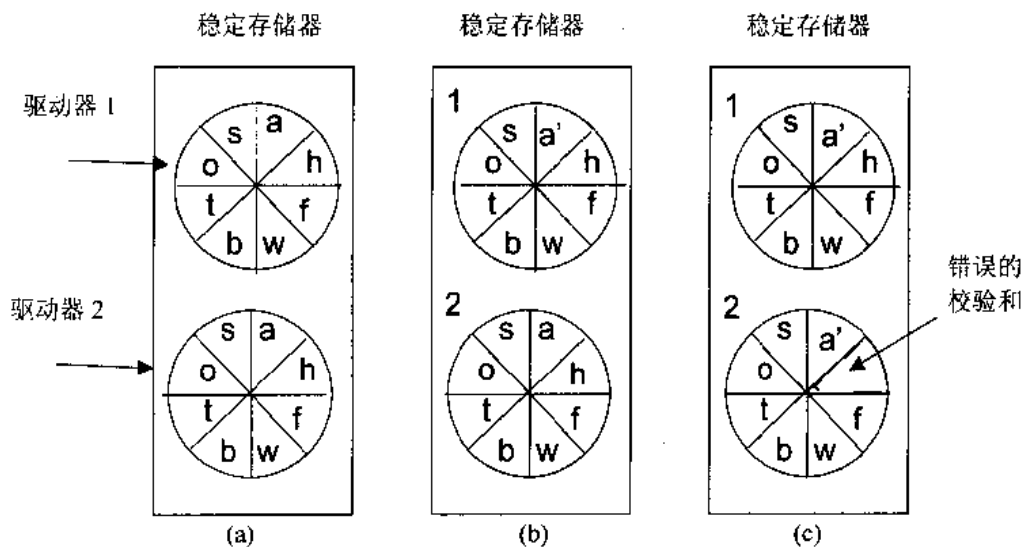


图 3-14 (a) 稳定存储器

(b) 盘 1 更新后崩溃

(c) 错误的地方

假设系统在更新驱动器 1 之后更新驱动器 2 之前崩溃，如图 3-14(b)所示。在恢复时，磁盘可以一块一块地进行比较，当两个相应块不一致时，可以假定驱动器 1 是正确的（因为驱动器 1 总是先于驱动器 2 更新），因此将新块从驱动器 1 拷贝到驱动器 2。当恢复过程完成后，两个驱动器又一致了。

另一个潜在的问题是磁盘块的自然损坏。垃圾微粒或者一般的磨损和破裂都将使一个以前正确的磁盘块会无缘无故或没有警告地出现奇偶校验错，如图 3-14(c)所示。当检测到这样的错误时，坏块可以通过另一台驱动器上的相应块恢复。

实现稳定存储器的价值在于它很适合于需要高度容错的应用，例如原子事务。当数据写入稳定存储器并读出以检验它们是否正确写入时，它们相继丢失的机会非常小。

事务原语

使用事务编程需要由操作系统提供或者由语言运行系统提供特殊原语，例如：

- (1) BEGIN_TRANSACTION: 标记一个事务的开始；
- (2) END_TRANSACTION: 结束事务并设法提交；

- (3) ABORT_TRANSACTION: 取消事务;恢复旧值;
- (4) READ: 从一个文件(或其他对象)读取数据;
- (5) WRITE: 将数据写入一个文件(或其他对象)。

更精确的原语列表取决于事务中正在使用的对象类型。在一个邮件系统中,可能有发送、接收以及转发邮件等原语。而在一个帐目系统中可能会有很大的不同。但是读取和写入却是典型的应用。在一个事务中,也允许使用普通语句、过程调用等。

BEGIN_TRANSACTION 和 END_TRANSACTION 用来限定事务的范围。介于它们之间的操作构成了事务体。这些操作要么全部执行,要么一个也不执行。这些操作可能是系统调用,库过程,或者是某种语言中用括号括起来的语句,这取决于实现的需要。

例如,考虑在航空公司的订票系统中,预定一张从纽约的 White Plains 到肯尼亚的 Malindi 的座票。一条线路是从 White Plains 到 JFK,再从 JFK 到 Nairobi,最后从 Nairobi 到达 Malindi。在图 3-15(a)中看到通过 3 个操作预定了这 3 条独立的航线的座票。现在假设前两条航线已经预定成功,但是第三条已经定满了。那么事务就会因此而终止,前两个预定的结果也将被取消——航空公司的数据库恢复到了事务开始前的状态(见图 3-15(b)),就像什么也没有发生一样。

BEGIN_TRANSACTION	BEGIN_TRANSACTION
reserve WP-JFK	reserve WP-JFK
reserve JFK-Nairobi	reserve JFK-Nairobi
reserve Nairobi-Malindi;	Nairobi-Malindi full→ABORT_TRANSACTION
END_TRANSACTION	
(a)	(b)

图 3-15 (a) 预定三个航班机票的事务

(b) 当第三个航班的机票预定失败后事务中止

事务的特性

事务有四个重要的特性,它们是:

- (1) 原子性(Atomic): 对外部世界来说,事务的发生是不可分割的;
- (2) 一致性(Consistent): 事务不会破坏系统的恒定;
- (3) 独立性(Isolated): 并发的事务不会互相干扰;
- (4) 持久性(Durable): 一旦一个事务提交,改变就是永远存在的。

这几个属性通常按其首字母简称为 ACID。

所有事务具有的第一个特性是原子性。这个特性确保了每个事务要么全部发生,要么全部不发生。如果发生,就是不可分割的瞬间的操作。当一个事务处在处理过程中时,其他进程(无论是否与事务有关)都不能看到任何中间状态。

例如,假设某个事务要在一个 10 字节的文件末尾添加数据。如果在事务处理过程中其他进程读取了这个文件,那么无论该事务已添加了多少内容,那些进程看到的只能是 10

个字节。如果事务成功结束，这时文件在结束的瞬时增长到了新的大小，无论有多少个操作存在，都不会有中间状态。

第二个特性说明事务是一致的。这意味着如果系统拥有某种必须经常保持的不变性，那么一旦在事务开始之前保持有这样的性质，则事务结束后该性质就还应该存在。例如在一个银行系统中，最关键的不变性是资金守恒规则。在任何内部转帐之后，银行的资金帐户应与转帐前保持一致，但是在事务执行的短暂时刻内，这种不变性会受到损害。然而事务结束之后，这种损害就没有了。

第三个特性说明事务是独立的或连续的。这意味着如果两个或两个以上的事务在同时运行，那么对它们自己和其他进程来说，最终结果看起来就像是所有的事务是按某种次序（依赖于系统）顺序运行的。

在图 3-16(a)-(c)中三个事务被三个独立的进程同时执行。如果它们顺序执行，那么 X 的最终结果应该是 1、2 或 3，取决于哪一个事务最后运行（X 可以是一个共享变量，一个文件或某种其他对象）。在图 3-16(d)中我们看到了称之为调度表的不同次序，其中事务可以是交错的。调度 1 是真正串行的，即事务严格按顺序来运行，因此它满足连续性的定义。调度 2 不是串行的，但是它也是合法的，因为它返回的 X 值与严格串行返回的值一致。第三个调度是非法的，因为它将 X 的值置成了 5，这是顺序执行所不可能产生的。为保证独立的操作正确交替执行是系统的责任。通过给予系统按照自己需要选择操作执行顺序的自由——假设它得到的答案是正确的——我们消除了编程人员自己进行互斥处理的需要，因此简化了编程。

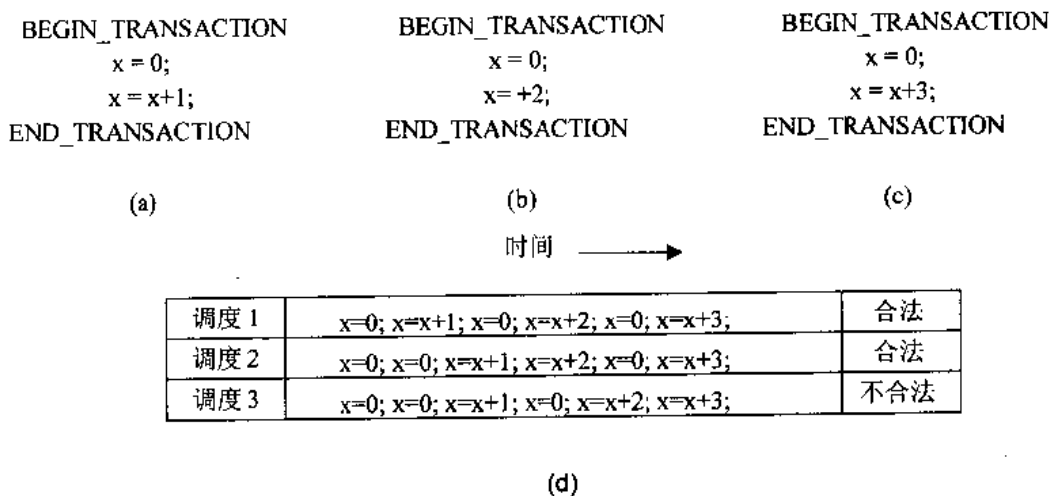


图 3-16 (a)-(c)三个事务

(d) 可能的调度

第四个特性说明事务是持久的。这是指这样一个事实，即一旦一个事务提交，无论发生什么，这个事务都会向前进行，结果不会再变了。提交之后发生的任何错误都不可能使结果取消或丢失。

嵌套事务

事务可以包含子事务，这通常称作嵌套事务。顶层事务可以在不同的处理机上创建并行运行子事务，以提高性能简化编程。这些子事务中的任何一个都可以执行一个或多个子事务，或者创建自己的子事务。

子事务引起了微小但很重要的问题。假设一个事务启动了几个并行的子事务，其中一个已经提交，并使自己的结果对父事务是可见的。经过更进一步的计算，父事务被中止，并将系统恢复到了顶层事务开始前的状态。但是，已经提交了子事务却没有被恢复。于是上面应用中的持久性只是对顶层事务而言。

由于事务可以嵌套任意多层，因此需要管理措施以使其一切正常。但是，语义却是清楚的。当任何事务或子事务开始的时候，在概念上它已给出整个系统全部对象的拷贝，可以按照其意愿进行操作。如果事务被中止，那么它的私有空间就会消失，就像事务从来都没有存在过一样。如果事务提交了，那么它的私有空间就会取代父辈的空间。因此如果一个子事务提交后又开始了一个新的子事务，那么它就可以看到第一个子事务的结果了。

3.4.3 实现

事务听起来是个好主意，但如何实现它呢？本节将解决这个问题。到目前为止，应该清楚如果每个进程执行一个事务仅仅只是更新它自己使用的对象（文件、数据库记录等），那么事务就不会是原子的，而且事务异常中止时它所作的更改也不会神奇地消失。此外，运行多个事务时也不会是串行的。显然，我们需要其他的实现方法。经常使用的有两种方法，下面依次进行讨论。

私有工作空间

从概念上讲，当一个进程开始一个事务时，会给它一个包含了所有需要访问的文件（和其他对象）的私有工作空间。在事务提交或中止前所有的读写操作都是在私有工作空间而不是在我们通常所说的真正的文件系统中进行的。这个观察结果直接导致了第一种实现方法的出现：在进程开始一个事务的时刻给它分配一个私有工作空间。

这种技术的一个问题在于将所有内容都拷贝到一个私有工作空间，其代价是难以承受的，但存在一些优化使得该方法更可行些。第一个优化是基于这样的一种认识，即当一个进程读取文件而不作修改时，就不需要私有拷贝了。该进程可以直接读取真正的文件（除非当事务开始后文件已被改动）。因此，当某个进程开始一个事务的时候，只要创建一个私有工作空间，该私有工作空间只包含一个指向父辈工作区的指针就足够了。当事务处于最顶层的时候，它的工作区就是真正的文件系统。当一个进程打开文件进行读取的时候，指针将向回指，直到可以在父辈（或更老的祖先）工作区中定位文件为止。

当打开一个文件进行写入时，它可用同读取时一样的方法进行定位，除非它是第一次拷入私有工作空间。然而，第二个优化方法大大减少了拷贝工作。该方法并不是将全部文件都拷入私有工作空间，而只将其索引拷入。索引是一个与判断文件所在磁盘块位置有关的数据块。在 UNIX 中，索引是 i 节点。通过私有索引，文件可以按通常的方式读取，因为索引中包含有文件开始几个块的磁盘地址。当一个文件块第一次被修改时，将生成该块

的拷贝，其地址也将被插入索引中，如图 3-17 所示。然后就可以在不影响原始块的情况下更新这个块。添加块也是用这个方法解决的。新块有时也称作影像块(shadow block)。

如图 3-17(b)中所示，运行事务的进程看到了被修改的文件，但是其他进程看到的仍是原来的文件。在一个更复杂的事务中，私有工作空间可能包含了大量的文件而不仅仅是一个。如果事务异常中止，那么私有工作空间就会被删除，它所指向的所有私有块也将被释放回自由列表中。如果事务成功结束，那么私有索引将被自动移到父辈的工作区中，如图 3-17(c)所示。不需要的块将它放回自由列表中。

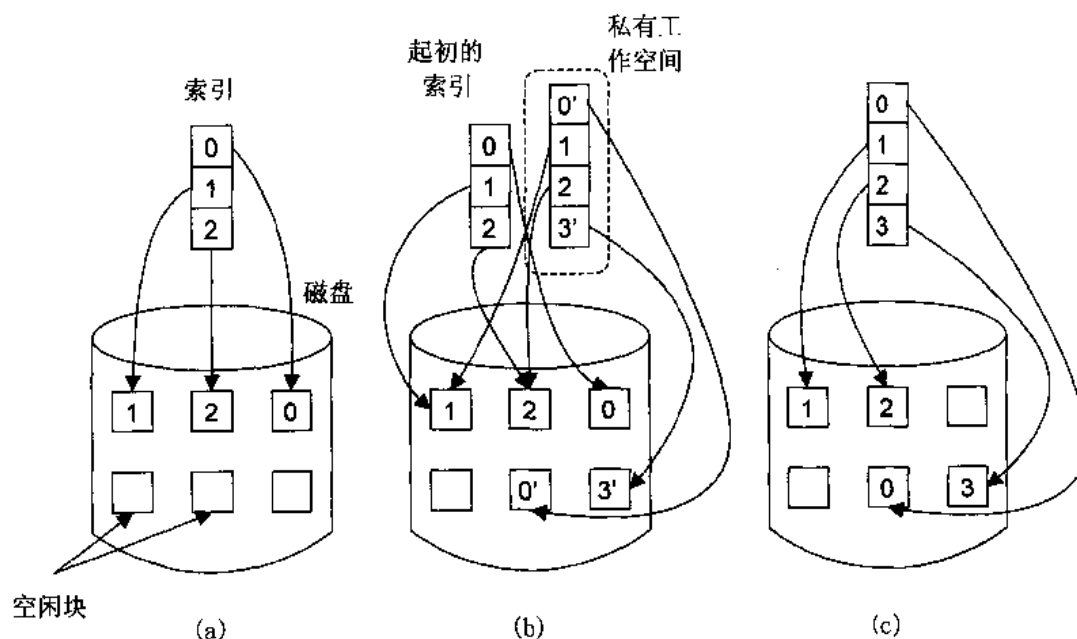


图 3-17 (a) 一个有三个数据块的文件的文件索引和磁盘数据块
(b) 一个事务修改第 0 块和附加数据块 3 后的情况
(c) 提交以后

写前日志

另一个实现事务的常用方法是写前日志(writeahead log)。有时也称为计划列表(intentions list)。使用这种方法时，文件将真正修改，但是在改动任何块之前，将一条记录写入稳定存储器的写前日志中，以判断是哪一个事务在进行修改操作，哪一个文件的哪一块被改动了，旧值与新值分别是什么。只有当日志成功的写入之后才可以进行文件的修改。

图 3-18 给出了一个日志如何工作的例子。图 3-18(a)中有一个使用两个初值均是 0 的共享变量（或其他对象） x , y 的简单事务。对事务中三条语句中的每一条，在执行之前都要写入一条日志，以记录新值和旧值，它们用斜杠来区分。

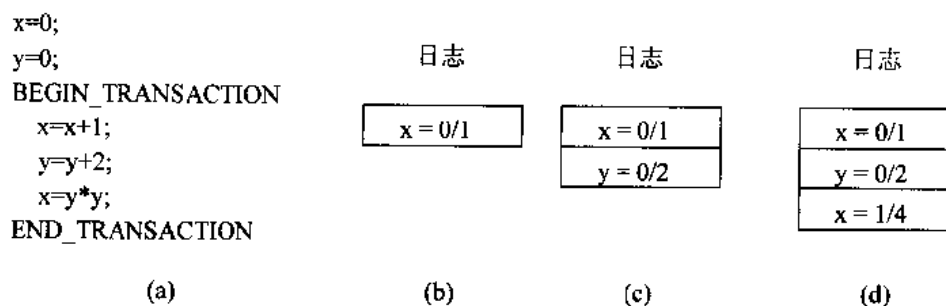


图 3-18 (a) 一个事务

(b)-(d) 每条语句执行前的日志

如果事务执行成功并被提交，那么它的提交记录将被写入日志，但数据结构不需要变动，因为它们已经被更新了。如果事务异常中止，那么可用日志来备份初始状态。从日志的末尾开始向回读取每条记录，再将记录中描述的改动复原。这个操作称之为滚回。

日志也可用来进行系统崩溃后的恢复。假设执行事务的进程在写入图 3-18(d)中的最后一条记录之后及改变 x 之前崩溃。在机器重新启动后，日志可以用来检查在系统崩溃时刻是否有事务正在处理中。当读到最后一条记录时显示 x 的当前值是 1，那么说明崩溃显然是在更新之前发生的，所以 x 的值应被改为 4。相反，如果在系统恢复时 x 的值是 4，那也同样的清楚的说明崩溃是在更新后发生的，因此不需要进行任何变动。使用日志使得事物向前（处理事务）和向后（撤消事务）成为了可能。

两阶段提交协议

就像我们不断指出的那样，事务提交操作必须是原子性的，即瞬时的和不可再分的。在分布式系统中，提交操作可能需要不同机器上的多个进程的协作，这些进程中的每一个都有一些被事务改动过的变量、文件、数据库或者其他对象。本节将研究一个在分布式系统中实现原子性提交的协议。

我们将看到的协议称之为两阶段提交协议（two-phase commit protocol）（Gray,1978）。尽管它不是此类协议中唯一的一个，但它却是使用最广泛的一个，其基本思想如图 3-19 所示。进程中的某一个将起到协调者的作用。一般来说这个进程就是执行事务的进程。提交协议开始时协调者先写入一条日志条目以表明它要开始提交协议，然后它给每个相关进程（下属）发送一条消息通知它们为提交作好准备。

当一个下属收到消息后，它先进行检查以确认是否为提交作好了准备，然后将它的决定发回给协调者。当协调者收到了所有的响应后，它就知道是否可以提交或中止。如果所有的进程都准备提交，那么事务就可以提交了。如果一个或几个进程不能提交（或没有响应），那么事务就得中止。无论是哪一种情况，协调者都要写一条日志记录并给每个下属发送一条消息以便将决定通知它们。正是因为写入的日志才使得事务真正被提交，并且无论发生什么它都会继续下去。

由于使用了稳定存储器上的日志，所以这个协议在（多个）崩溃面前仍是很有弹性的。如果协调者在写入了初始化日志后崩溃，那么在恢复时只需要从停止的地方开始继续工作

就可以了，若需要，还可以重发初始化信息。如果在响应第一条消息之前某个下属崩溃了，那么协调者在放弃之前将会给它不断地发送消息。如果协调者后来崩溃了，那么它就可以从日志中看出自己所处的位置，并能决定该作些什么。

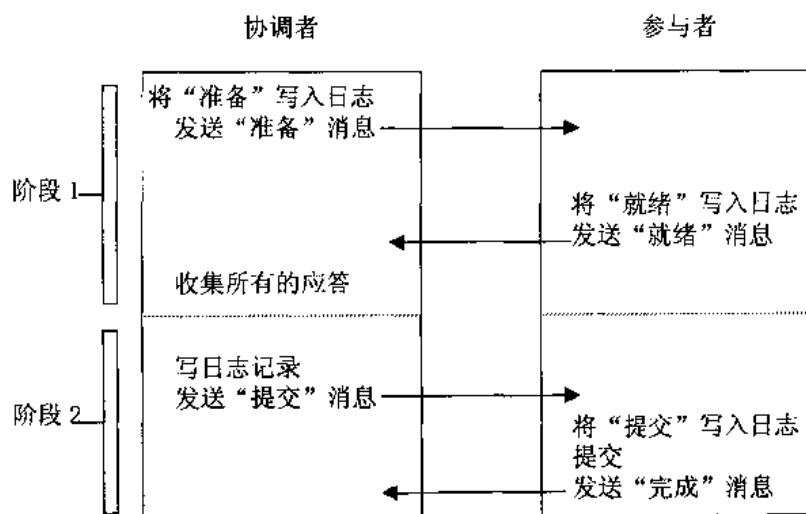


图 3-19 成功的两阶段提交协议

3.4.4 并发控制

当多个事务在不同的进程（在不同的处理机上）中同时执行时，需要一些机制以保证它们互不干扰。这种机制称为并发控制算法。本节将研究三个不同的算法。

加锁法

最古老而且使用最广泛的并发控制算法是加锁法。最简单的形式是：作为一个事务的一部分，当一个进程需要读或写一个文件（或其他对象）时，它首先将这个文件加锁。加锁法可以通过使用一个集中式的加锁管理程序来实现，也可以在每台机器上有一个本地加锁管理程序来管理本地文件。两种情况下加锁管理程序都拥有一个锁定的文件列表，所有欲对已加锁的文件进行的加锁尝试都将被拒绝。由于正常的进程在一个文件被加锁前不会试图去存取它，因此对文件加锁可以防止其他进程对文件的访问，这就保证了在一个事务的生存期内文件不会被改变。锁一般由事务系统请求和释放，不需要编程人员的操作。

这个基本方案的限制过于严格，可以通过区分读锁和写锁来加以改进。如果在一个文件上设置了读锁，那么在它上面设置其他的读锁也是允许的。读锁用来确保文件不会被改变（也就是排斥所有写入者），但是没有理由禁止其他读取文件的事务。与此相反，当一个文件被加锁以进行写入时，其他任何类型的锁都是禁止的。所以说读锁是可以共享的，而写锁必须是互斥的。

为了简单起见，我们曾经假设加锁的单位是整个文件。但在实际中，可能是更小一些的单位，比如记录或页面，也可能是大一些，比如整个数据库。一个加锁单位究竟取多大的问题称为锁的粒度。粒度越合适，加锁就可以越精确，也就能实现更大的并行度（例如，

并不因为某个进程正在使用文件的开头就阻塞另一个试图使用该文件末尾的进程)。另一方面,锁分得越细致,也就越需要更多的锁,这样的开销也就越大,也就更容易导致死锁。

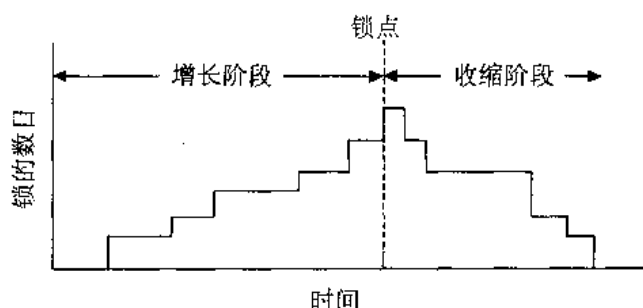


图 3-20 两阶段加锁

恰好在需要或不再需要锁时去请求或释放锁可能会导致不一致和死锁。因此,许多用加锁方法实现的事务都采用所谓的两阶段加锁法。如图 3-20 所示,在两阶段加锁法中,进程在增长阶段先请求它需要的所有锁,然后在收缩阶段释放它们。如果进程直到收缩阶段才试图更新文件,那么对因请求锁而造成的失败的处理仅仅是释放掉所有的锁,等待一段时间后再全部重新开始。此外,可以证明(Eswaran 等,1976)如果所有的事务都使用两阶段加锁法,那么通过交错事务进行的所有调度都是串行的。这也是两阶段加锁法广泛使用的原因。

在许多系统中,收缩阶段是在事务结束运行之后要么提交要么中止时出现。这种称之为精确的两阶段加锁法的策略有两个主要的优点。第一,一个事务总是读由另一已提交的事务写入的值,因此我们不会因为一个事务的计算是基于它本不应该看到的文件而中止它。第二,所有锁的请求与释放都可以由系统来处理而无须事务关心:只要访问一个文件要就可以请求一个锁,当事务结束时就可以释放一个锁。这种策略消除了瀑布型中止:不得不撤消一个已经提交的事务,因为它看到了不该看见的文件。

加锁,即使两阶段锁,也可能导致死锁。如果两个进程都试图以相反的顺序请求同一对锁,那么就会发生死锁。这里可以采用诸如以某种规范顺序请求所有的锁之类的一般方法来防止保持——等待循环的出现。通过对一张精确描述哪个进程拥有哪个锁,它还想请求哪个锁的图进行死锁扫描,以检查是否有环路出现,也可以防止死锁。最后一点,如果事先知道一个锁的拥有时间不会超过 T 秒时,也可以采用一个超时方案:如果某个拥有者连续拥有同一个锁超过了 T 秒,那么一定是出现了死锁。

乐观的并发控制

处理同时运行多个事务的第二种方法是乐观并发控制法(Kung and Robinson,1981)。这种方法的思想惊人的简单:尽管放心去做你想做的,不用在意其他人正在做什么。如果有问题出现,那么以后再考虑吧(许多政治家也采用这个策略)。在实际情况中,冲突相对来说非常少,所以这个策略大部分时间都可以正常工作。

尽管冲突会非常少,但存在的可能性还是有的,因此还需要一些处理冲突的方法。乐观并发控制算法所做的只是记录下有哪些文件曾经被读写过。在提交时刻,它会检测其他

的事务以判断在本事务开始后它的文件是否被其他事务修改过。如果被修改过，那么本事务将被中止。如果没有修改过，那么本事务就可以提交了。

乐观并发控制算法最适合于基于私有工作空间的情况。在这种情况下，每个事务都独立地修改各自的文件，而不会干涉其他的事务。在结束的时候，新的文件要么被提交要么被释放。

乐观并发控制算法的最大优点在于它是避免了死锁，而且允许最大的并行度，因为进程不需要去等待一个锁。它的缺点是有时可能会失效，这时所有的事务都必须退回重新运行一遍。在重负载的情况下，算法失效的可能性将会直线上升，这使得乐观并发控制算法成了一个很糟糕的选择。

时间戳

一个完全不同的并发控制方法是在一个事务开始做 `BEGIN_TRANSACTION` 的时候给它分配一个时间戳 (Reed, 1983)，通过使用 Lamport 的算法，我们可以确保时间戳是唯一的，在这里它很重要。系统中的每个文件都拥有一个相关的读取时间戳和写入时间戳，以判断哪个已提交的进程最近一次读取或写入过该文件。如果事务都很短小而且在时间间隔上比较大，那么一般来说当一个进程试图访问某个文件时，该文件的读写时间戳将低于（即早于）当前事务的时间戳。这种次序意味着事务正在以正确的顺序进行处理，一切正常。

当次序不正确的时候，就表明一个晚于当前事务开始的事务试图插入、访问文件并提交。这种情况意味着当前事务开始得过早了，因此需要中止。在某种意义上，这种方案同 Kung 和 Robinson 的方案一样，也是乐观的，尽管两者的细节完全不同。在 Kung 和 Robinson 的方法中，我们希望并发事务不使用同一个文件。在时间戳方法中，我们不介意并发事务是否使用同一个文件，只要编号小的事务总是先执行就可以了。

用例子来说明时间戳方法将会非常地简单。假设有三个事务 α ， β 和 γ 。 α 很早以前就开始运行了，并且要使用 β 和 γ 需要的所有文件，因此这些文件的读写时间戳都将被设置成 α 的时间戳。 β 和 γ 同时开始，但 β 的时间戳小于 γ （当然会大于 α ）。

首先我们来考虑 β 写文件的情况。第一步它将分别调用自己的时间戳 T 和将要写入的文件的读写时间戳 T_{RD} 和 T_{WR} 。除非 γ 已经结束并提交，否则 T_{RD} 和 T_{WR} 都应该是 α 的时间戳，因此也就小于 T 。在图 3-21(a) 和 (b) 中我们看到 T 比 T_{RD} 和 T_{WR} 都大（ γ 还未提交），因此写入是可接受的，可以尝试进行。 β 提交后，结果就将是持久的了。 β 的时间戳记录在文件中以当作试验写的时间戳。

在图 3-21(c) 和 (d) 中 β 不太走运。 γ 既读取 (c) 又写入 (d) 文件并提交。那么 β 事务就将被中止。但是它可以采用一个新的时间戳全部重新开始。

现在再来看看读取的情况。在图 3-21(e) 中没有冲突，因此读取操作可以立即执行。在图 3-21(f) 中，某个闯入者进入并试图写文件。由于闯入者的时间戳小于 β 的，因此 β 就需要等待直到它提交，随后 β 就可以读取新文件继续执行了。

在图 3-21(g) 中， γ 修改了文件并已提交。这一次 β 仍必须被中止。在图 3-21(h) 中 γ 正在修改文件，尽管它还未被提交，但 β 还是开始的太早仍必须中止。

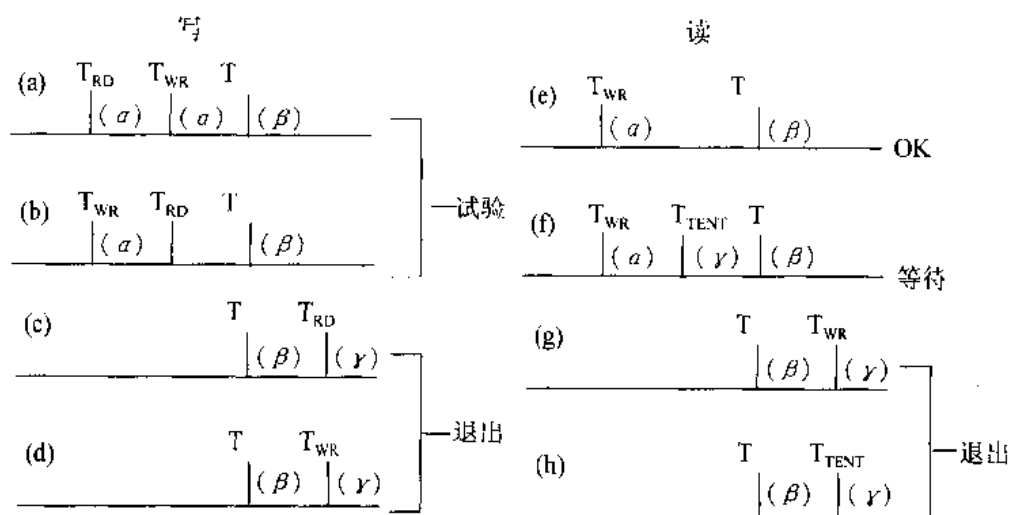


图 3-21 使用时间戳进行并发控制

同加锁法相比，时间戳有着不同的特性。当一个事务碰到了更大（晚）的时间戳时，它就要中止，但是加锁法在相同的情况下要么等待要么立即执行。另一方面，时间戳方法不会出现死锁，这是一个极大的改进。

总而言之，事务具备许多优点，因此对构造可靠的分布式系统而言它就成为了一种很有前途的技术。它的主要问题在于实现的复杂性，这将导致降低性能。好在这些问题正处于研究之中，经过一段时间之后它们可能会得到解决。

3.5 分布式系统中的死锁

分布式系统中的死锁类似于单处理机系统中的死锁，只是情况更糟。它们更难于避免、预防或者检测，即使在检测到以后也很难处理，因为所有的相关信息都分散在多台机器上。在某些系统中，如分布式数据库系统中，死锁的问题可能会相当严重，因此理解分布式系统中的死锁与一般的死锁有何不同，它们应该如何处理就显得非常重要了。

有人将分布式死锁分成了两类：通信死锁和资源死锁。例如，进程 A 试图发送消息给进程 B ，进程 B 试图给进程 C 发送消息，而 C 又试图给 A 发送消息，那么就会发生死锁。在这种情况下导致死锁的原因可能有多种，例如无法得到缓冲区。当多个进程为了互斥访问 I/O 设备、文件、锁或其他资源时就会发生资源死锁。

既然进程可以请求或释放通信信道、缓冲区等资源，它也可以按资源死锁处理，因此这里我不想区分这两种死锁。此外，在大部分系统中如上所述的环路通信类型是非常少见的。例如，在一个客户-服务器系统中某个客户可能要发送一条消息（或执行 RPC）到文件服务器，这可能会导致发送消息到磁盘服务器。然而磁盘服务器不会像一个客户端那样给最初的那个客户发送消息以期望它能像一个服务器那样进行响应。因此环路等待情况不会仅是由于通信而产生的。

处理死锁问题的策略有很多种。下面讨论其中四种最著名的策略。

- (1) 鸵鸟算法（忽略问题）；
- (2) 检测（允许死锁发生，在检测到后想办法恢复）；
- (3) 预防（静态的使死锁在结构上是不可能发生的）；
- (4) 避免（通过仔细的分配资源以避免死锁）。

在分布式系统中这四种方法都是可以使用的。鸵鸟算法在分布式系统中同在单处理机系统中一样好用，一样受欢迎。尽管在诸如分布式数据库等个人应用中，如果需要就可以实现它们自己的死锁机制。但是在编程、办公自动化、过程控制和许多用于其他应用的分布式系统中都没有系统级的死锁机制。

死锁检测与恢复算法也很流行，这主要是因为预防算法和避免算法太难了。后面将讨论几种用于死锁检测的算法。

尽管比在单处理机系统中困难得多，但是死锁预防还是有可能实现的。在原子事务提出之后，有一些新的想法可以采用。后面将讨论两种算法。

最后要说明的是，死锁避免在分布式系统中从来都不采用。既然在单处理机系统中都不采用死锁避免机制，那么在更复杂的分布式系统中我们又何必采用它呢？该方法的问题在于银行家算法和类似算法需要（事先）知道每个进程最终到底需要多少资源。而这样的信息即使有也非常的少。因此我们关于分布式系统中的死锁问题的讨论将只集中于两种技术：死锁检测和死锁预防。

3.5.1 分布式死锁检测

在分布式系统中找出一般的死锁预防和避免的解决方法是相当困难的，因此许多研究人员都只是尝试为更简单的死锁检测问题找出一种解决方法，而不是想办法去禁止死锁的发生。

然而，在一些分布式系统中原子事务的提出使得在概念上有了极大的不同。在普通的操作系统中检测到死锁后，解决方法是中止掉一个或几个进程，但这必然会使一些用户感到不满。在基于原子事务的系统中检测到死锁后，解决方法是中止掉一个或几个事务。但正如上述，将事务设计成允许出现中止。当一个事务因为产生死锁而被中止的时候，首先要作的是将系统恢复到事务开始前的状态，以后事务可以从这一点重新开始。如果运气能好一些的话，那么第二次执行时就应该能成功。因此使用事务与不使用事务的差别在于使用事务时中止一个进程的后果要比不使用事务时的后果小得多的多。

集中式的死锁检测

作为第一个尝试，我们使用集中式的死锁检测算法来尽量模仿非分布式的算法。尽管每台机器都有一幅资源图以描述自己所拥有的进程和资源，但仍旧会有一台中心机器拥有整个系统（所有资源图的集合）的资源图。当协调者检测到了环路时，它就中止一个进程以解决死锁。

与集中式系统中所有信息都放在适当的地方可以自动获得不同，在分布式系统中所有信息都要精确地发送到适当的地方。每台机器的资源图中只包含它自己的进程和资源。但从适当的地方获取所需信息的可能性是存在的。第一种方法，每当资源图中加入或删除一

条弧时，相应的消息就应发送给协调者以提供更新。第二种方法，每个进程可以周期性的把自从上次更新之后新添加和删除的弧的列表发送给协调者。这种方法比第一种方法发送的消息要少。第三种方法是在需要的时候协调者主动去请求信息。

不幸的是这些方法的效果都不太好。我们可以考虑这样一种系统，进程 A 和进程 B 运行在机器 0 上， C 运行在机器 1 上。一共有三种资源 R, S 和 T 。开始的情况如图 3-22(a)(b) 所示， A 拥有 S 并想请求 R ，但它不可能得到，因为 B 正在使用 R ； C 拥有 T 并想请求 S 。协调者看到的情况如图 3-22(c) 所示。这种配置是安全的。一旦 B 结束运行， A 就可以得到 R 然后结束，并释放 C 所等待的 S 。

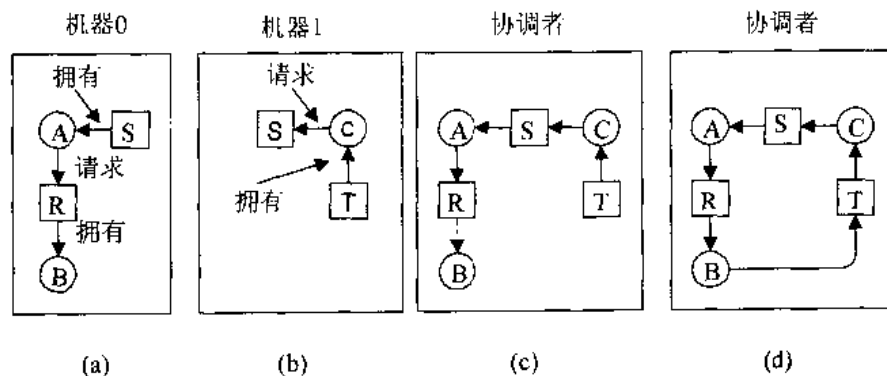


图 3-22 (a) 机器 0 初始资源图
(b) 机器 1 初始资源图
(c) 协调者对系统的观察
(d) 延迟信息后的系统情况

过一会， B 释放 R 并请求 T ，这是一个完全合法的安全操作。机器 0 向协调者发送一条消息声明它释放 R ，机器 1 向协调者发送了一条消息声明进程 B 正在等待它的资源 T 。不幸的是，机器 1 的消息首先到达，这导致协调者生成了一副如图 3-22(d) 所示的资源图。协调者错误地得出死锁存在的结论，并中止某个进程。这种情况称为假死锁。由于信息的不完整和延迟使得分布式系统中的许多死锁算法产生了类似的假死锁问题。

一种可能的解决方法是使用 Lamport 算法以提供全局时间。既然从机器 1 到协调者的消息是由机器 0 的请求发出的，那么从机器 1 到协调者的消息的时间戳就应该晚于从机器 0 到协调者的消息的时间戳。当协调者收到了从机器 1 发来的有导致死锁嫌疑的消息后，它将给系统中的每台机器发送一条消息说：“我刚刚收到一条会导致死锁的带有时间戳 T 的消息，如果任何人有小于该时间戳的消息要发给我，请立即发送。”当每台机器或肯定或否定的响应之后，协调者就会看到从 R 到 B 已经消失了，因此系统仍是安全的。尽管这种方法消除了假死锁，但是它需要全局时间，而且开销很大。另外，其他的一些消除假死锁的方法也很困难。

分布式的死锁检测

已经发表了很多分布式死锁检测算法。关于该主题的统计在 Knapp(1987)和

Singhal(1989)中已给出。我们来看一个典型的算法，Chandy-Misra-Haas 算法(Chandy 等,1983)。该算法允许进程一次请求多个资源（如锁）而不是一次一个。通过允许多个请求同时进行使得事务的增长阶段可观地加速。该模型的这种变化的结果使得一个进程可以同时等待两个或多个进程。

在图 3-23 中，我们给出了一种改进的资源图，图中只给出进程。每条弧穿过一个资源，但为简单起见将资源从图中删除了。可以看到机器 1 上的进程 3 正在等待两个资源，一个由进程 4 占有，一个由进程 5 占有。一些进程在等待本地资源，例如进程 1。但也有—些进程，如进程 2 在等待其他机器上的资源。显然连接机器的弧使得寻找环路更加困难。当某个进程等待资源时，例如进程 0 等待进程 1，将调用 Chandy-Misra-Haas 算法。此时，生成一个特殊的探测消息并发送给占用资源的进程。消息由三个数字构成：阻塞的进程，发送消息的进程，接收消息的进程。由 0 到 1 的初始消息包含三元组 $(0,0,1)$ 。

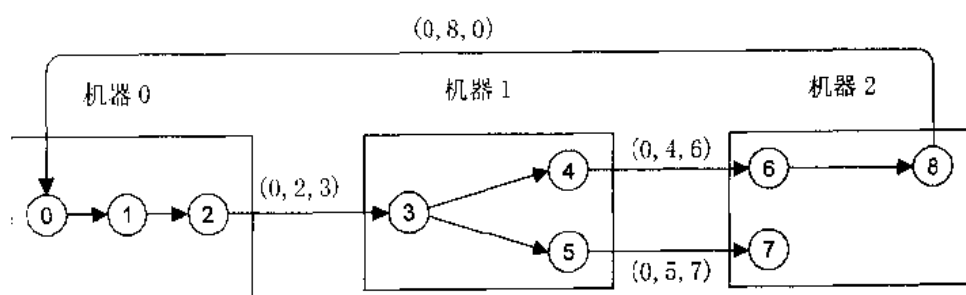


图 3-23 Chandy-Misra-Haas 分布式死锁检测算法

消息到达后，接收者检查以确认它自己是否也在等待其他进程。如果是，那么消息就要被更新，第一个字段保持不变，第二个字段改为当前进程号，第三个字段改为等待的进程号。然后消息接着被发送到等待的进程。如果在等待多个进程，就需要发送多个不同的消息。不论资源在本地还是在远程，该算法都要继续下去。在图 3-23 中我们可以看到标记为 $(0,2,3)$ ， $(0,4,6)$ ， $(0,5,7)$ 和 $(0,8,0)$ 的远程消息。如果消息转了一圈后又回到了最初的发送者，即第一个字段所列的进程，那么就说明存在一个有死锁的环路系统。

可以有不同的方法打破死锁。一种方法是使最初发送探测消息的进程自杀。然而如果有多个进程同时调用了此算法那就会出现—问题。例如在图 3-23 中假设进程 0 到 6 同时阻塞，而且都初始化了探测消息。那么每个进程最终都会发现死锁，并且因此而自杀，然而这是不必要的。中止掉一个进程就足够了。

另一种算法是将每个进程的标识符添加到探测消息的末尾，这样当它返回到最初的发送者时完整的环路就可以列出来了。于是发送者就能看出哪个进程的编号最大，可以将它中止或者发消息给它请求其自杀。无论如何，如果多个进程同时发现了同一个环路，它们就一定会选择同一个牺牲者。

在计算机科学领域中很少有像分布式死锁检测算法这样理论与实际间存在如此大分歧的情况。然而发现—种新的死锁检测算法仍是许多研究人员的目标。不幸的是，这些模型通常都和现实毫无关系。例如，一些算法在进程被阻塞时需要它们发送探测消息，然而当一个进程被阻塞的时候让它发送—条探测消息决不是一件容易的事情。

许多论文都包含有关于新算法性能的细致分析,例如,它们指出当一个新进程需要遍历两遍环路的时候应采用较短的消息,好像这些因素可以弥补性能一样。但是毫无疑问当作者得知在 LAN 上的典型短消息(20 字节)耗时 1 毫秒,而长消息(100 字节)只耗时 1.1 毫秒时他一定会非常吃惊。对这些人来说当他们意识到实验测试表明所有死锁环路的百分之九十都是由两个进程引起的时候(Gray 等,1981),毫无疑问他们一定会非常震惊。

最糟糕的是,该领域已经发表的算法中很大一部分都有明显的错误,甚至包括那些已经证明是正确的。Knapp(1987)和 Singhal(1989)给出了一些例子。下面的事情会经常发生:一个算法刚刚提出,被证明是正确的,然后就发表了,但是人们随后却发现了反例。所以说我们有一个很活跃的研究领域,但是问题却与现实不太一致,找到的解决方案一般都不太现实,给出的性能分析也是毫无意义的,已经被证明的结论却经常是错误的。为了以积极的态度结束本节内容,我们可以说这是一个提供了大量改进机会的研究领域。

3.5.2 分布式死锁预防

死锁预防是由细致的系统设计构成的,因此死锁从结构上来说是不可能的。不同的技术包括在某时刻只允许进程占有一个资源,要求进程在初始阶段请求所有资源,当进程请求新资源时必须先释放所有资源。但在实践中这些方法都不太方便。有时采用的一种方法是必须预定资源,并要求进程以严格增序请求资源。这种方法意味着一个进程不可能既占有了一个高序资源又去请求一个低序资源,这就使得环路不可能出现了。

在拥有全局时间和原子事务的分布式系统中,另外两种实用的算法也是可能的。这两种算法都是基于在一个事务开始时给它分配一个全局时间戳的思想。同许多基于时间戳的算法一样,在这两种算法中保证不会有二个事务分配了完全一致的时间戳,这点是非常重要的。正如我们所看到的,Lamport 的算法有效地保证了时间戳是唯一的(通过使用进程号)。

这种算法的思想是当一个进程因等待一个正被其他进程占用的资源而要阻塞时,进行检查以判断哪个进程的时间戳更大(即更新)。只有当等待进程的时间戳小于(早于)被等待进程的时间戳时我们才允许等待发生。按这种方式,任何沿着等待进程链时间戳总是增大的,因此环路是不可能的。或者,只有当等待进程拥有大于(新于)被等待进程的时间戳时,我们才允许等待发生,在这种情况下沿着时间戳链时间戳总是减小的。

尽管两种方法都能预防死锁,但是给予老的进程以优先权更明智些。它们已经运行了较长时间,因此系统对它们的投入会更大一些,它们占有的资源也就更多一些。另外,一个被中止的新进程在它最终成为系统中最老的进程之后仍能够再生,因此这种选择消除了饿死现象。正如我们前面所指出的,中止一个事务相对来说是无害的,因为按照定义随后它能够安全地再生。

为了使该算法更清楚,我们考虑图 3-24 所示的情况。图(a)中,一个较老的进程想得到一个被新进程占用的资源。图(b)中,一个新进程想得到被较老进程占用的资源。一种情况应该允许进程等待,另一种情况应该中止进程。假设标记(a)为中止(b)为等待。那么我们就应该中止掉老进程,它试图使用被新进程占用的资源,但这样的效率较低。所以我

们应以相反的方式进行标记，如图所示。在这种情况下，箭头总是指向事务编号增长的方向，使得环路不可能出现。这种算法称为等-死算法（wait-die）。

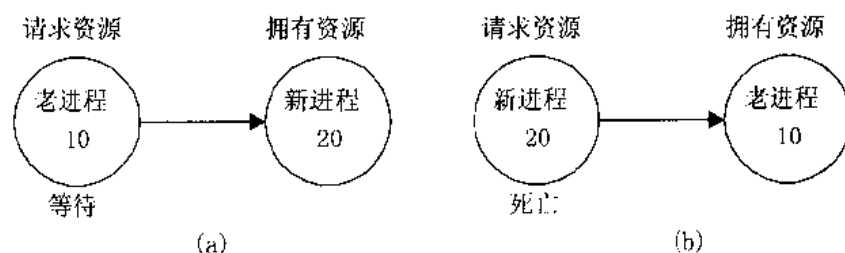


图 3-24 等-死死锁预防算法

一旦我们假设了事务的存在，我们就可以做一些在以前是被禁止的事情：从运行的进程中取走资源。当冲突发生的时候，我们不需要中止提出请求的进程，我们可以中止资源的拥有者。如果没有事务，中止一个进程可能会有严重的后果，例如进程可能已经修改了文件。有了事务后，当事务死亡时这些效果将会神奇地消失。

现在来分析图 3-25 中的情况，在这里我们允许抢先的存在。假设我们的系统尊敬老者，如同我们上面所讨论的，我们不希望年轻人抢在可敬的老人前面，因此是图 3-25(a)而不是图 3-25(b)被标记为抢先。我们现在可以安全地标记图 3-25(b)为等待。这种算法称为伤-等算法（wound-wait），因为一个事务可能会受到伤害（实际是被中止）而其他的事务等待。这种算法还不可能达到 Nomenclature Hall 的知名度。

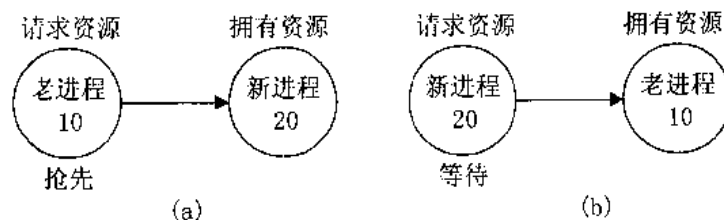


图 3-25 伤-等死锁预防算法

如果一个老的进程希望得到一个被新进程占用的资源，那么老的进程将会抢先，于是新事务将被中止，如图 3-25(a)所示。随后新事务可能会立即重新开始，并试着请求资源，如图 3-25(b)所示，然后被迫等待。将这种算法与等-死算法进行比较可以发现：在等-死算法中，如果一个老的事务想得到正被新事务占用的资源，那么它会很有礼貌地等待。然而如果一个新事务想要得到老的事务占用的资源，它将被中止。毫无疑问它还会重新开始，并且又被中止。在老的事务释放资源之前，这个循环可能要重复多次。伤-等算法没有这么糟糕的特性。

3.6 小结

本章是关于分布式系统同步的内容。开始时我们介绍了 Lamport 的时钟同步算法而没有使用内部时钟源，随后我们又给出了这个算法的用途。另外我们还介绍了当实时非常重要时是如何利用物理时钟来进行同步的。

接着介绍分布式系统中的互斥并研究了三种算法。集中式算法将所有信息保存在一个地方。分布式算法在所有的地方并行进行计算。令牌环算法将控制沿环传递。每种算法都有它的优势和劣势。

许多分布式算法都需要一个协调者，在此列举了两种选举协调者的算法：欺负算法和另一种环算法。

尽管前面的内容都很有趣很重要，但是它们都是底层概念。事务是一个使编程人员处理分布式系统中的互斥、加锁、容错和死锁问题更简单的概念。我们介绍了事务模型，事务是如何实现的，以及三种并发控制方案：加锁，乐观并发控制以及时间戳。

最后，我们回顾了死锁问题并介绍了一些在分布式系统中进行死锁检测和预防的算法。

习 题

- (1) 在图 3-2 (b) 中加入一个与消息 A 并行的新消息，即它既不在 A 之前发生也不在 A 之后发生。
- (2) 说出至少三种可在 WWV 广播发送时刻和分布式系统处理机设置内部时钟之间引入的延迟源。
- (3) 考虑一个分布式系统中的两台机器。这两台机器的时钟假设都每毫秒滴答 1000 次，但实际上只有一个是这样，而另一个每毫秒仅滴答 990 次，如果 UTC 每分钟更新一次那么时钟的最大偏移量将是多少？
- (4) 在使用租约实现缓冲一致性的方法中，时钟同步是必需的吗？如果不是，需要什么要求？
- (5) 在实现互斥的集中式方法中（图 3-8），协调者在收到一个进程释放它互斥访问临界区的消息后，通常给等待队列中的第一个进程授权允许其访问临界区。请给出协调者另一个可能的算法。
- (6) 请再考虑图 3-8。假设协调者崩溃。这将必然导致整个系统瘫痪吗？如果不是，在什么条件下将使系统发生瘫痪？有什么方法避免这个问题使系统能够忍受协调者崩溃？
- (7) Ricart 和 Agrawala 算法中存在这样一个问题：即如果一个进程崩溃并对另一个欲进入临界区的进程的请求未作应答，没有应答将意味着拒绝其请求。我们建议所有的请求应当立即被应答，以便很容易地检测到崩溃的进程。是否存在一些情况，即使

是使用这种方法也不是足够的呢？请讨论。

- (8) 一个分布式系统可能会有多个独立的临界区。假设进程 0 想进入临界区 *A* 和进程 1 想进入临界区 *B*。Ricart 和 Agrawala's 的算法会导致死锁吗？请解释为什么？
- (9) 在图 3-11 中可进行一点优化，试问怎么优化？
- (10) 假设两个进程同时发现协调者死亡，并且它们都决定用 bully(欺负) 算法进行选举。将会发生什么情况？
- (11) 在图 3-12 中，我们有两个 *ELECTION*(选举) 消息同时循环。但对它们都没有什么不利的影响，如果消灭其中的一个会更好。设计一个算法来实现这个想法而不影响基本的选举算法的运行。
- (12) 在图 3-13 中我们看到了一种用磁带实现对存货列表自动更新的方法。因为磁带可以很容易地用磁盘上的一个文件来模拟，你认为为什么这种方法（用磁盘文件代替）不被使用呢？
- (13) 对于一些非常敏感的应用程序，可以想像即使是用两个磁盘实现的稳定存储器也是不够可靠的。这种想法同样适用于三个磁盘吗？如果是，为什么？如果不是，给出理由。
- (14) 在图 3-16 (d) 中有三个调度策略，两个合法的，一个不合法的。对于同一个事务处理，给出最终 *x* 所有的可能值和合法的与不合法的状态的一个完整的列表。
- (15) 当用一个私有工作空间实现事务处理时，可能需要将大量的文件索引拷贝到父辈工作区。怎样实现这种操作而不引入竞争条件？
- (16) 在写前日志中，旧值和新值都被存储于日志的记录中，只存储新值可以吗？存储旧值的好处是什么？
- (17) 在图 3-19 中，在什么时刻达到无返回点？即是在什么时候原子事务提交真正执行？
- (18) 给出一个完整的算法判定一个试图锁住一个文件的操作是否成功。要同时考虑到读锁和写锁，以及文件未锁住、读锁住或写锁住的可能性。
- (19) 用加锁的方法实现并发控制的系统通常区分读锁和写锁。如果一个进程已经得到了一个读锁但现在又想将它换成写锁，将会怎么样呢？将写锁换成读锁又会怎么样呢？
- (20) 乐观并发控制与使用时间戳，哪一个更严格呢？为什么？
- (21) 使用时间标记法进行并发控制保证了串行性吗？试讨论。
- (22) 我们已经反复地提及当一个事务处理被中止时，整个系统将恢复到它原来的状态，就像这个事务处理从没发生过一样。其实并不是这样的，给出一个例子，说明重新设置整个系统使之恢复原来的状态是不可能的。
- (23) 文中描述的集中式死锁检测算法初始时给出一个假死锁，但是以后用全局时间来修补。假设它现在决定不再维护全局时间（因太昂贵）。设计一个替代的方法来修正这个算法中的小错误。
- (24) 一个其事务时间戳为 50 的进程需要一个事务时间戳为 100 的进程的资源。如下各情况：
 - (a) Wait-die(等-死)
 - (b) Wound-Wait(伤-等) 会怎样呢？

第4章 分布式系统中的进程和处理机

前两章，我们已经了解了在分布式系统中通信和同步这两个相关的主题。本章将转到另一个不同的主题：进程。尽管进程在单机系统中也是重要的概念，但本章强调的是进程管理的特殊部分，它不同于经典操作系统中的内容。特别是在多处理机环境下如何对它进行处理。

在许多分布式系统中，一个进程可能有多个线程，这种能力提供了很多重要的好处，但也带来了各种问题。我们首先研究线程概念，接着研究如何管理处理机和进程，并且介绍几种不同的模型；然后介绍分布式系统中处理机的分配和调度；最后研究两种特殊的分布式系统。

4.1 线程

在大多数传统的操作系统中，每个进程有一个地址空间和单线程控制。事实上这几乎已成为进程的定义。然而，很多情况下希望多个线程共享一个地址空间并可并行运行，就好像它们是多个独立的进程。本节将讨论这些内容和它们的含义。

4.1.1 线程简介

例如，假设文件服务器有时不得不因等待磁盘的响应而阻塞，如果这个服务器有多个线程，当第一个线程阻塞时第二个线程就可运行，实际结果可能是更高的吞吐量和更好的性能。不可能创建两个独立的服务进程来达到这个目的，因为它们共享同一缓冲区，要求它们在同一地址空间中。因而需要一种新的机制，但在单一处理机系统的历史上还未找到一种这样的机制。

图 4-1(a)中，可看到一台机器有三个进程，每个进程有它自己的程序计数器、堆栈、寄存器和地址空间，这些进程之间互不相干，它们能够通过系统进程间通信原语，如：信号量、管程、消息进行通信。图 4-1(b)我们看到另一台机器，有一个进程，它包含多个线程控制，经常称为线程或有时称为轻量级进程。在许多方面，线程像微小进程，每个线程按顺序执行，并有自己的程序计数器和堆栈来记录运行到什么地方。线程像进程一样共享处理机：首先是一个线程运行，然后是另一个线程运行（分时）。仅在多处理机时它们才并行运行。线程能创建子线程也能阻塞以等待系统调用的完成，像通常进程一样，当一个线程被阻塞时，运行同一进程中的另一线程，类似于一个进程阻塞，另一个进程运行一样。这种情况类似：线程相对于进程，犹如进程相对于机器。

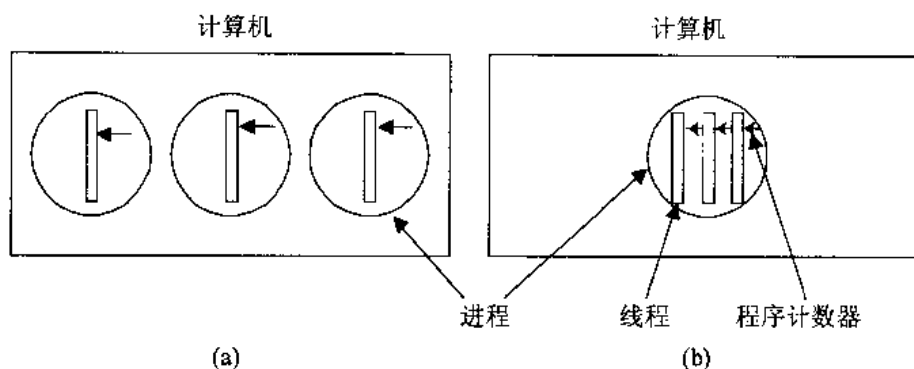


图 4-1 (a) 三个进程，每一个进程有一个线程
(b) 一个进程有三个线程

一个线程被阻塞时，运行同一进程中的另一线程，类似于一个进程阻塞，另一个进程运行一样。这种情况类似：线程相对于进程，犹如进程相对于机器。

然而，同一进程中的不同线程并不像不同进程之间完全是相互独立的，所有线程有同一地址空间，也就是它们共享全局变量。由于每个线程能存取每个虚拟地址，每个线程能读写，甚至清除另一线程的堆栈，线程之间没有保护。因为，第一这不可能保护；其二也没有必要。不像进程间，它们属于不同的用户，相互排斥，一个进程总是属于一个使用者，用户创造了多线程是为了相互合作，而不是冲突。而且共享一个地址空间，所有线程共享同一批打开文件、子进程、时钟和信号量等。如图 4-2 所示。在图 4-1(a)中的结构适合于当三个进程互不相干时。当三个线程是同一任务的一部分且相互紧密合作时，图 4-1(b)将是最适合的。

每个线程的项目	每个进程的项目
程序计数器	地址空间
堆栈	全局变量
寄存器组	打开的文件
子线程	子进程
状态	计时器
	标志
	信号量
	计算信息

图 4-2 每一线程与每一进程的内容

像传统的进程(如：只有一个线程的进程)一样，线程处于以下几种状态之一：运行、阻塞、就绪或者结束。正在运行的线程占用 CPU，并且是激活的。一个阻塞的线程等待另一正在运行的线程唤醒（例如通过信号量），并调度一个就绪线程运行。最后，一个结束的线程将退出，但它还没有被父线程回收（在 UNIX 中，父进程不执行 WAIT）。

4.1.2 线程的用途

线程的引入是为了使并行执行与顺序执行相结合。再考虑文件服务器的例子，一种可能的结构，如图 4-3(a)所示：在这里某一线程是派遣者（dispatcher），它从系统邮箱内读出输入请求。然后检查请求，选择一个空闲的工作者线程去处理它，可能是通过将指向那个请求消息的指针写入到一个与每个线程相关联的一个特殊字中。然后派遣者唤醒睡眠的工作者。

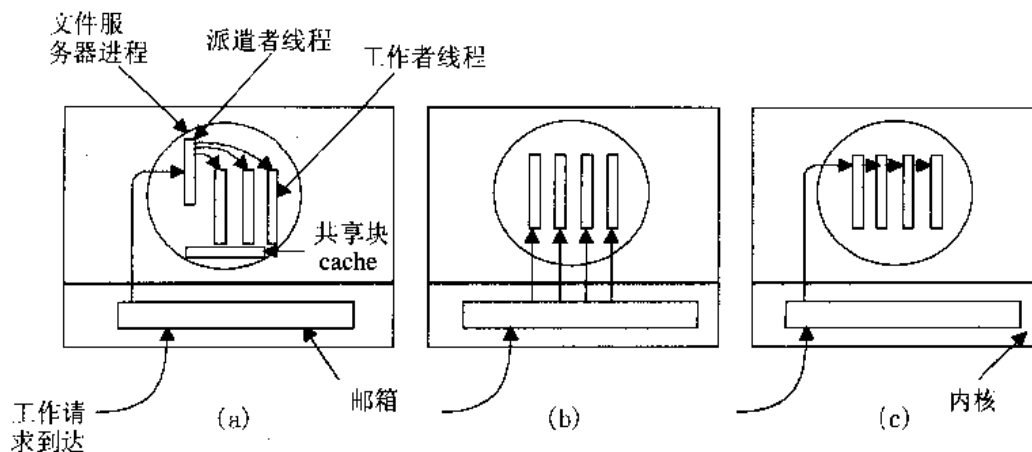


图 4-3 线程与进程的一种组织

(a) 派遣者/工作者模型

(b) 团队模型

(c) 管道线模型

当工作者被唤醒后，它检查任何一个线程可访问的共享块缓冲区是否可以满足这个请求。如不能满足，给磁盘发出消息，要求所需的数据块(假设是 READ)，且进入休眠状态等待磁盘操作的完成。现在调用调度程序，开始另一个线程，为了获得更多的工作，此线程可能是派遣者，或者可能是另一个工作者准备运行。

文件服务器在无多线程的情况下是怎样被写入的呢？一种可能性是让它作为单独线程执行。文件服务器的主循环是接收一个请求、检查它，而且在下一个请求到来前完成它。当文件服务器等待磁盘操作时，它是空闲的且不处理另一请求，如果文件服务器运行于一个专用的机器上（实际中，大多数情况是这样），当文件服务器等待磁盘时，CPU 也是空闲的。实际结果是每秒钟可处理的请求大大减少。因而多线程能得到相当好的性能，但每个线程是以平常方式顺序执行的。

到目前为止我们看到两种可能的的设计，多线程文件服务器和单线程文件服务器。假设多线程不可用，但系统设计者又发现由于单线程而引起的性能降低是不可接受的。那么第三种可能性是将服务器作为大的具有有限状态机运行，当请求到来后有唯一的一个线程检查它，如果缓冲区能满足，进行运行，但是如果不能，就必须向磁盘发送一条消息。

然而，这时文件服务器并不阻塞，而是将当前请求的状态记录在一张表中，然后去获得下一条消息，下条消息可能是请求一个新工作或者是磁盘关于上次操作的应答。如果

是请求一个新工作，就激活它。如果是从磁盘发来的应答，那么从表中取出相关信息并处理这个应答。由于这里不允许发送消息并且阻塞以等待应答，因此不能使用远程过程调用，原语应是非阻塞调用的 *send* 和 *receive*。

在此设计中，前两种情况没有使用“顺序进程”模型，对于表中每条发送和接收的消息运算的状态都必须能清楚明确的保存并恢复。实际上，我们在以一种生硬的方式模拟多线程和这些线程的堆栈。将进程作为一个有限状态机运行，它接收事件后根据事件本身特性响应处理它。

现在应该很清楚多线程能提供什么。他们既保留顺序进程的思想又实现了并行性，阻塞系统调用使编程容易，而并行性提高了性能，单线程服务器保留了阻塞系统的优点，但放弃了性能，有限状态机方法通过并行性取得高性能，但使用非阻塞调用因而编程困难，这些模型在表 4-1 中作一概括。

表 4-1 构造服务器的三种方法

模式	特 性
线程	并行,阻塞系统调用
单线程进程	不并行,阻塞系统调用
有限状态机	并行,非阻塞系统调用

图 4-3(a)的派遣者结构不是组织多线程进程的唯一方法，图 4-3(b)的团队模型也是一种方法。在这种情况下所有的线程都是平等的，每个都获得和处理自己的请求，这里没有派遣者，有时工作来了线程不能处理，尤其是如果每个线程用来处理一种特殊的工作，这种情况下，可以维护一个作业队列，挂起的作业保持在作业队列中。使用这种组织结构，线程在查看系统信箱前应先查看作业队列。

多线程也能用如图 4-3(c)所示的管道线模型来组织。这种模型中的第一个线程产生一些数据传给下一个线程去处理。数据持续从一个线程传到另一个线程，经过的每一个线程都进行处理。尽管这对于文件服务器不适合，但对于其他问题如生产者 - 消费者问题来说可能是一种好的选择。管道线在计算系统的很多方面得到广泛使用，如从 RISC CPU 的内部结构到 UNIX 的命令行。

多线程对客户端来说通常也很有用。例如，如果一个客户端想将某文件复制在多个服务器上，它可用一个线程与每一个服务器通信。客户端多线程的另一用处是处理信号。像来自键盘的中断 (DEL 或 BREAK)，不是让这些信号中断进程，而是让一个线程专用于等待这些信号，通常这个线程是被阻塞的。但当信号到来时，唤醒并处理该信号。因此使用线程能够消除用户层中断的需求。

对多线程的另一讨论是与 RPC 或通信无关的。有些应用使用并行处理很容易编程。例如生产者 - 消费者问题，生产者和消费者是否真正的并行是次要的。这样编程是为了使软件设计更简单。由于它们共享缓冲区，让它们处于不同的进程做不到这点，多线程恰好适合这种情况。

最后，尽管在这里没有明确讨论多处理机系统的情况。但多线程真正可以在同一地址空间的不同 CPU 中并行运行，实际上，这也就是在那些系统中实现共享的一种主要方法。另一方面，一个使用多线程的合理设计的程序，它应能在分时使用线程的单 CPU 的

条件下运行与在一个真正多处理机条件下运行其效果是一样好，所以软件中的问题对两种情况几乎完全相同。

4.1.3 线程包的设计问题

与线程相关的用户可得的原语集（即库调用）叫作线程包，本节将考虑与线程包的结构和功能有关的一些问题，下一节将介绍线程包如何实现。

我们首先讨论线程管理。它可从静态多线程和动态多线程两种中选一种，对于静态设计，当程序编写或被编译时就要决定选择多少线程。每个线程分配一个固定堆栈。这种方法虽简单，却不灵活。

更普遍的方法是允许线程在运行过程中动态的创建和回收，线程创建调用通常指定线程的主程序（使用指向过程的指针）和堆栈的大小，也可能同时指定其他参数，例如：调度优先级。线程创建调用通常返回一个线程标识符，用于这个线程以后的调用。这种模型中进程以一个线程开始运行，但能根据需要创建多个线程，该线程完成后可以退出。

线程有两种方法结束：当一个线程完成它的工作时，可以自己退出；或者被外界中止。这方面，线程像进程。在许多情况下，如图 4-3 所示的文件服务器那样，进程启动后线程立即被创建，并从不被中止。

由于线程共享存储器，多个线程能利用这个特性共享数据，像生产者-消费者问题中的缓冲池。共享数据的存取通常是用临界区方法编程实现，这样做是为了防止多个线程在同一时间存取同一数据。临界区很容易用信号量、管程和类似的结构来实现。一种在线程包中普遍使用的技术是互斥体（mutex），也是一种消耗信号量（watered-down-semaphore）。互斥体总是处于两种状态：打开和锁住。互斥体定义了两种操作，一种是加锁操作，如果互斥体处于打开状态，它将仅仅用一个原子操作锁住互斥体。如果两个线程企图在同一时刻锁住同一互斥体，这仅在多处理机环境中是可能的，在这种环境下不同的线程运行在不同的 CPU 中，它们中一个成功锁住而另一个失败。如一个线程要给一个已经锁住的互斥体加锁则它将被阻塞。

开锁操作是打开互斥体。如果一个或多个线程由于互斥体被锁住而等待，实际上它们之中只能有一个被开锁，其余的继续等待。

有时提供另一种操作试锁（trylock），它尝试锁住互斥体。如果互斥体是打开的，试锁将返回表示成功的状态标识码。反之，如互斥体是锁住的，试锁不会阻塞线程，而是返回失败状态的标识码。

互斥体像二进制信号量（例如：仅有 0 和 1 二个值的信号量），它们不像计数信号量，用这种方法限制它们使它们更容易实现。

有时，线程包中可用的另一同步特征是条件变量，它类似于在管程中用于同步的条件变量，每一条件变量通常在创建时与一个互斥体相关联。互斥体与条件变量的区别是互斥体用于短期加锁，以监视进入临界区。而条件变量是用于长时间等待直到资源可用为止。

下列情况随时可能发生：一个线程锁住互斥体以进入临界区，一旦它进入临界区，就检查系统表并发现它所需的某些资源正处于忙的状态。如果它简单地锁住第二个互斥体（与它所需资源相关联），外部互斥体将保持锁状态，而正占用此资源的那个线程就不能进入临界区来释放该锁（与所需资源相关联），结果产生了死锁。打开外部的互斥体让其

他线程可进入临界区，将产生混乱，因此这种解决方法是不可行的。

一种解决办法是用条件变量获得资源，如图 4-4(a)所示。这里，在条件变量上等待定义为执行等待和打开互斥体。稍后，当占用资源的线程释放资源后，如图 4-4(b)所示，它调用 *wakeup*, *wakeup* 用于唤醒在特定条件变量上等待的一个或所有的线程。在图 4-4(a)中使用 *WHILE* 而不用 *IF* 是用来防止这种情况的：某线程被唤醒了但其他某个线程在它运行之前先占用了那些资源。

<pre>lock mutex; check data structures; while(resource busy) wait(condition variable); mark resource as busy; unlock mutex;</pre>	<pre>lock mutex; mark resource as free; unlock mutex; wakeup(condition variable);</pre>
(a)	(b)

图 4-4 互斥变量与条件变量的使用

有能力唤醒所有的线程而不仅仅是一个线程，这种需要在读者—写者问题中已阐释。当写者完成时，它可以唤醒正在等待的读者或写者，如果选择读者，它应该唤醒所有读者而不是其中的一个。通过使用仅仅唤醒一个线程原语和唤醒所有线程原语提供了所需的灵活性。

线程的代码通常由多个过程构成。像一个进程一样，这些过程有局部变量、全局变量和过程参数。局部变量和参数不会产生任何麻烦。但相对于线程的全局变量而不是相对于整个程序的全局变量会产生麻烦。

例如：考虑 UNIX 系统支持的 *errno* 变量。当进程（或线程）的系统调用失败后，将错误代码放入 *errno* 中。在图 4-5 中，线程 1 执行系统调用 *ACCESS*（访问）看是否允许存取某文件。操作系统用全局变量 *errno* 来应答。在控制权返回线程 1 后并在它读取 *errno* 前，调度程序认为线程 1 已经占用足够的 CPU 时间了，且决定调度线程 2。线程 2 执行 *OPEN*（打开）调用失败，引起重写 *errno*，使线程 1 的存取代码永远丢失，当以后再调度线程 1 时，它将读取错误的值并且执行错误的操作。

对此问题有不同的解决办法。第一种办法是禁止使用全局变量，不管这种思想多么有价值，还是与许多现存的软件相冲突，如 UNIX。另一种办法是给每个线程分配它自己的私有全局变量。如图 4-6 所示。这种方法，每个线程有自己私有的 *errno* 变量的一个拷贝和其他全局变量，因此可以避免冲突。实际上，这种措施创建了新的辖区级（*scoping level*），对于线程中的所有过程该变量都是可见的，这不同于现存的辖区级变量仅对一个过程可见而不是在程序中任何地方皆可见。

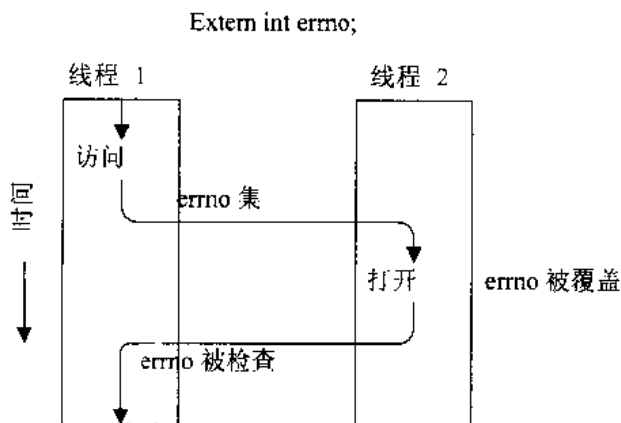


图 4-5 使用全局变量线程之间的冲突

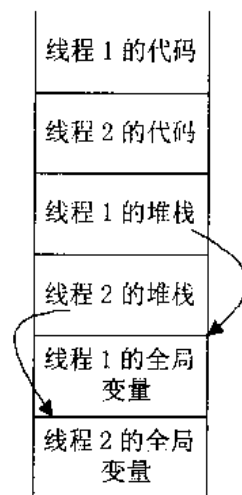


图 4-6 线程私有全局变量

因为大多数的编程语言都能表达全局变量和局部变量，所以存取私有全局变量有些技巧。但它不能表达中间形式，可以分配一大块内存给全局变量并将它作为一个额外的参数传递给线程中的每一个过程，虽然这不是一个很好的解决方法，但它却有效。

另一替代方法是引入新的库过程来创建、设置和读取这些线程全局变量。第一个调用是：

```
create_global( "bufptr" );
```

它为 `bufptr` 指针在堆栈中或在调用线程所保留的特殊的存储空间中分配所需存储空间，无论存储空间在什么地方分配，仅有调用线程能存取该全局变量，如果另一线程创建了同名的全局变量，它将分配不同的存储块，而不会与已存在的冲突。

存取全局变量需要两种调用：读和写。对于写，如：

```
Set_global( "bufptr", &buf );
```

它将指针的值存储在前面用 `create_global` 调用生成的存储区中。为了读全局变量调用如：

```
bufptr=read_global( "bufptr" );
```

这个调用将返回全局变量中存储的地址值，因此能够读取数据值。

最后讨论线程调度，线程可用不同的调度算法进行调度，包括：优先级法、轮转法等等。线程包通常提供系统调用使用户能够指定调度算法和设置调度优先级。

4.1.4 实现一个线程包

有两种方法可以实现线程包：在用户空间中和在内核中。对于这两种方法还存在一些争议，可能会出现一种混合的方法。本节将讨论这些方法以及它们的优缺点。

在用户空间中实现线程

第一种方法是线程包完全放到用户空间中去，内核对此一无所知。目前所涉及的

内核是管理普通的、单线程的进程。首先且最明显的好处是用户级的线程包能够在不支持线程的操作系统中实现。例如：UNIX 并不支持线程，但已经有了为它而写的各种各样的用户空间的线程包。

所有这些的实现都有相同的结构，如图 4-7(a)所示。线程运行在运行期系统（runtime system）的上层，运行期系统是一些管理线程的过程集，当一个线程执行了一个系统调用时就进入休眠，对信号量或互斥体执行一个操作，或其他什么原因而挂起，它都调用一个运行期系统过程，这个过程检查线程是否必须挂起。如果是，它将该线程的寄存器存储到一个表中，然后寻找另一未阻塞的线程运行，将新线程所保存的值重新装入到它自己的机器寄存器中。一旦切换了堆栈指针和程序计数器的内容，新的线程就自动进入运行状态。如果机器有一条指令用来保存所有寄存器且有另一条指令用来装入所有寄存器的值时，整个线程切换可用有限的几条指令来完成。用这种方法实现线程切换至少比使用内核陷阱快一个数量级，这也是用户级线程包优点的有力论据。

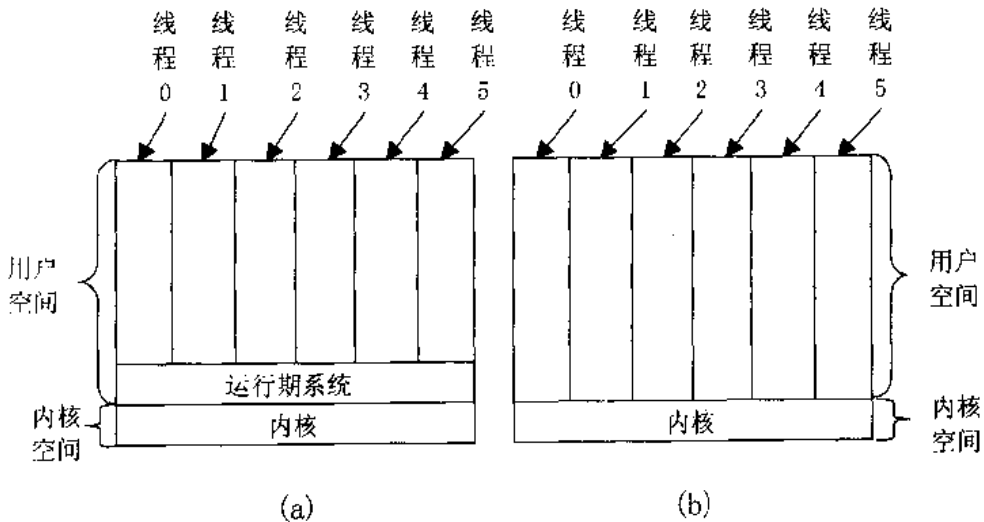


图 4-7 (a) 用户级线程包
(b) 由内核管理的线程包

用户级线程包还有其他好处。它允许每一个进程有自己定制的调度算法。对某些应用如：有垃圾收集器线程的应用，不用担心在不合适的时刻被停止，这就是一个好处。它们的可扩展性也很好，由于内核线程总是需要内核中的许多表和堆栈空间，如果存在大量的线程将产生问题。

尽管它们有较好的性能，但用户级线程包存在一些主要的问题。

首先阻塞调用是怎样实现的问题。假设线程从空管道中读数据或其他的一些操作将导致阻塞的操作。让线程发出这样的系统调用是不可行的，因为这样将终止所有线程。引入多线程的主要目的首先是允许每一个线程使用阻塞调用，但又要阻止已阻塞的线程去影响其他的线程。运用阻塞系统调用，不能实现这个目的。

可以将系统调用全部变为非阻塞系统调用（例如：读一个空管道将会失败），这对于操作系统的改变是微不足道的。除此之外，对于用户级线程的一个争论恰恰是它可运行在现存操作系统中，另外改变 READ 的语义将需要修改许多用户程序。

如果我们能事先确定一个调用是否会引起阻塞,那么有另一种替代方法。在某些 UNIX 版本中,存在一个调用 SELECT,它能告诉调用者管道是否为空等等。当这个调用出现时,库过程 READ(读)过程可被一个新的过程替换,这个新过程首先调用 SELECT;然后如果是安全的(即不阻塞)才执行 READ 调用。如果 READ 调用将要阻塞,就不执行这个调用,而是运行另一线程。运行期系统再次得到控制,它再次检查 READ 调用是否安全,这种需要重写部分系统调用库的方法不仅效率低且不方便,但这几乎无可选择。系统调用外围作检查用的代码称作外壳(jacket)。

与阻塞系统调用类似的问题是页面错,如果一个线程产生页面错时,内核甚至并不知道这些线程的存在,自然会阻塞整个进程直到所需要的页面取出为止。

用户级线程包的另一问题是如果一个线程开始运行,进程中的其他线程除非在第一个线程自愿释放 CPU 后它才能运行。在单一进程中,没有时钟中断,也不可能实现轮转法调度,除非一个线程自愿进入运行期系统,否则没有任何机会让调度程序调度它。

同步领域中无时钟中断是致命的。在分布式中有一种很普遍的现象。一线程初始化一个动作另一线程必须响应,且连续进行循环检测看响应是否发生。这种条件叫旋转锁住(spin lock)或忙等待。这种方法在期望得到快速反应时很有用,同时使用同步信号量的花费又很高。如果线程能依靠时钟中断在几个毫秒的时间内自动调度,这种方法工作得很好。然而,如果线程在阻塞前一直运行,这种方法就容易产生死锁。

解决线程一直运行的一个办法是让运行期系统每秒得到一个时钟信号中断,来使它得到控制权,这样会使程序太粗糙和混乱。更高频率的周期性时钟中断并不总是可能的。即使可能,总的费用也很大。而且,线程需要的时钟中断可能与运行期系统使用时钟相互干扰。

另一个也可能是对用户级线程构成最大的威胁是编程人员通常想在线程经常阻塞的应用中使用多线程,例如多线程文件服务器。这些线程不停地进行系统调用。一旦产生陷阱进入内核执行系统调用,如果旧线程阻塞了,内核要进行线程切换将很困难,并且将使内核不停地检查系统调用是否安全。对于那些本质上完全受限于 CPU 和几乎不阻塞的应用程序,有多线程又有什么意义?没有人会认真地提出使用多线程来计算前 n 个质数或用它来下棋,因为这样做什么好处也得不到。

在内核中实现线程

现在让我们假设内核知道并管理线程。运行期系统不再需要,如图 4-7(b)所示。取而代之的是,当一个线程想去创建一个新线程或撤消已存在的线程时,它发出一个内核调用,由它完成创建和回收工作。

为了管理所有线程,在内核中每个进程都有一张表,每个线程在表中有一个入口。每个入口保存着线程的寄存器、状态、优先权和其他信息。这些信息和用户级线程一样,但它现在是在内核中,而不是在用户空间中。这些信息类似于传统内核中维护的每一个单线程进程的信息。

所有可能阻塞线程的调用,如线程间使用信号量同步,都是按系统调用来实现的,但这将比运行期系统过程开销更大。当一线程阻塞时,内核根据它的选择,可运行同一进程中的其他线程(如果有已准备好的线程),或者另一个进程的线程。用户级线程中,运行期系统持续运行自己进程中的线程直到内核收回处理机(CPU)为止(或者没有已就绪

的线程可运行了)。

由于在内核中创建和撤消线程需更高的花费,所以有些系统采用环境补偿的方法再循环它们的线程。当线程撤消时,将它标记为不能运行,但它的内核数据结构却不受影响。稍后当它作为一个新线程创建时,原有的线程被重新激活,这样,节省了一些开销。对用户级线程来说线程再循环也是可能的。但是,由于线程管理的开销较小,因此没有理由要这样做。

内核线程不需要任何新的非阻塞系统调用,当使用旋转锁(spin lock)时,它们也不导致死锁。而且,如果一个进程中的线程产生了页面错,内核在等待来自磁盘或网络的所需页面时,能很容易地运行另一线程。它们的主要缺点是系统调用耗费很大。因此如果线程操作(创建,删除,同步等)很频繁,将引起更多的系统开销。

除了用户线程和内核线程所特有的问题外,它们还有一些相同的问题。例如,许多库函数是不可重入的。例如,可用编程为在网络上发送一条消息,它首先在一个固定的缓冲区中组装要发送的消息,然后利用内核陷阱发送出去。如果一个线程已经在缓冲区中组装好它要发送的消息,然后一个时钟中断强迫将该线程切换为另一个线程,那个线程又马上用自己的消息重写缓冲区,这时会发生什么情况呢?类似的,当一个系统调用完成后,发生了线程切换,可能改变前一线程读出的错误状态(errno,如前面所讨论的)。同样,内存分配过程(像UNIX中的malloc),可随意地访问临界表而不影响设置和使用受保护的临界区,因为它们是为单线程环境设计的,而在那种环境下根本没必要。有效地解决所有这些问题意味着重写整个库。

另一解决办法是给每个过程提供一个jacket,当过程被启动时,它锁住一个全局信号量或互斥体。通过这种方法,在库中一次仅有一线程是激活的。这样,实际上整个库成为了一个大的监视程序。

信号也是一个难题,假设某线程想捕获一特殊信号(例如:用户单击DEL键),而另一线程想让这信号终止进程。可能出现这种情况:如果一个或多个线程运行标准库过程,而其他一些是用户写的。显然,这些线程的愿望不相容。通常,在单线程环境中管理信号是很困难的,而在多线程环境下也不容易。信号,是典型的每进程概念,而不是每线程的概念,特别是当内核不知道线程的存在时。

调度者行为

许多研究者都在尝试结合用户线程(良好的性能)和内核线程(不需要很多技巧就可使之工作)的长处,下面将描述由Anderson等人设计的这样一种方法(1991),称作调度者行为(scheduler activations)。有关的工作在Edler等人(1988)和Scott等人(1990)中已作出介绍。

调度者行为的目标是模拟内核线程的功能,像用户空间内实现的线程包一样有更好的性能和有更大的灵活性。特别地,用户线程不必发出特殊的非阻塞系统调用或者事先检查发出某个系统调用是否安全。然而,当一个线程由于系统调用或页面错阻塞时,如果其他线程就绪的话,系统应该能够运行同一进程中的其他线程。

通过避免在用户空间和内核空间之间不必要的转换来实现高效率。例如:如果某线程在局部信号上阻塞,没有理由去涉及内核。用户空间运行期系统能阻塞同步线程并自主调度新线程。

当使用调度者行为时，内核分配一定数量的虚拟处理机给每个进程，并且让（用户空间）运行期系统将线程分配给处理机。这种机制也可用在下面的多处理机环境，即那些虚拟处理机可能是实在的处理机。分配给一个进程的虚拟处理机的数量初始值是一个，但进程可以要求多个，并且也可归还不再使用的处理机。内核能够回收已经分配的处理机给其他的更需要处理机的进程。

这种方案最基本的思想是当内核知道一个线程已阻塞时（例如：通过某个线程执行了一个阻塞系统调用或产生一个页面错），内核将通过将这个线程的号码和所发生事件的描述作为参数传递到堆栈，通知该进程的运行期系统，这种通知是通过内核激活已知起始地址的运行期系统发生的，有这类似 UNIX 中的一个信号。这种机制称为 **upcall**。

运行期系统一旦被激活，它就能重新调度线程，典型的是标记当前的线程为阻塞状态，然后从等待队列中调度新线程，设置它们的寄存器且重新启动运行它们。稍后，当内核得知前一进程又能重新运行时（例如：它试图读取的管道现在已有数据了，或者出错页面已经从磁盘上调入），它将向运行期系统发出另一 **upcall** 通知它。运行期系统由它自己决定，可以立即重新启动阻塞线程或者将它放入等待队列稍后运行。

当一个用户线程正在运行产生一个硬件中断时，被中断的 CPU 切换进入内核模式。如果引起中断的事件与被中断的线程无关，如另一进程完成 I/O 操作，当中断处理结束后，被中断的线程将恢复到中断前的状态。然而如果进程与中断有关，例如进程的某一线程需要的页面已到来，那么不会重新启动被中断的线程。取而代之的是：挂起被中断的线程，并且运行期系统根据堆栈中被中断线程的状态在虚拟 CPU 上启动。然后，运行期系统决定哪个线程调度到那个 CPU 上：被中断的、新就绪的或其他另一线程。

尽管调度者行为解决了怎样将控制权由进程中的阻塞线程传递给非阻塞线程的问题，但同时它产生了一个新问题。此新问题是被中断的线程在挂起时可能正执行对信号量的操作，在这种情况下，这个线程在就绪队列中可能正持有一把锁。如果由 **upcall** 启动的运行期系统企图自己得到这把锁，目的是将新的就绪线程放入队列中，它就可能失败并随之产生死锁，这种问题能通过跟踪线程什么时候在或不在临界区来解决，但这种解决方法太复杂并且很不高明。

另一个调度者行为的缺陷是它主要依靠 **upcall**，**upcall** 这个概念与任何一个分层系统的内在结构相违背。通常，第 n 层提供第 $n+1$ 层可调用的某种服务，但是第 n 层不能调用第 $n+1$ 层中的过程。

4.1.5 线程和远程过程调用（RPC）

分布式系统中普遍同时使用 RPC 和线程，因为线程是用来作为标准（重量级）进程的廉价替代物。在这种环境下，研究者自然会更仔细地研究 RPC，探究 RPC 是否也能成为更轻量级的，本节将讨论这个领域内一些有趣的工作。

Bershad 等人(1990)发现即使在分布式系统中，大量的 RPC 调用是调用与它们在同一机器上的进程（例如：调用窗口管理器）。显然，这种结果的产生依赖于系统，但它也是值得考虑的。他们提出了一个新的方案，这个方案可以使一个进程中的线程以一种比普通方法更有效的方法调用同一台机器上的另一个进程中的线程。

这种方法的主要思想是：当服务器线程 S 启动时，它输出它的接口告诉内核，这个接

口定义了哪一个过程可调用，调用的参数等等。当客户线程 C 启动时，从内核输入该接口，获得用于调用的特殊标识。内核现在知道 C 以后将调用 S，并且创建特殊的数据结构为调用作准备。

这些数据结构之一是线程 C 和 S 共享的参数堆栈，并且这个数据结构在两者（C 和 S）的地址空间中被映像为读/写。为调用服务器，C 使用普通的过程把参数压入共享栈，同时，在寄存器中放入一个特殊标识，然后产生陷阱进入内核。内核看到这个特殊标识后知道这个调用在本地（如果调用是在远端，内核将以远程调用的标准方式来处理），然后更改客户内存映像将客户放入服务器的地址空间，并且启动客户线程开始执行服务器过程。调用发出时，参数已在适当的位置，所以不需要拷贝或排列参数。结果用这种方法，本地的 RPC 可以很快地完成。

另一加速 RPC 执行的技术是基于以下观察：当一个服务器线程因等待一个新请求而阻塞时，它实际上没有任何重要的环境信息。例如，它几乎没有局部变量，通常寄存器中没有什么重要的信息。因此，当一个线程执行完请求时，它将消失并且它的堆栈和环境信息也被丢弃。

当一个新的消息进入服务器机时，内核动态创建一个新线程去为此请求服务。并且它把这条消息映像到服务器地址空间，然后建立新线程的堆栈来存取这条消息。这种方案有时称为默认接收（implicit receive），并与传统的使用系统调用接收信息的线程形成对比，自发创建的用来处理到来的 RPC 的线程有时称作弹出线程（pop-up thread）。图 4-8 解释了这种思想。

这种方法比传统的 RPC 有几个重要的优点。首先，线程不会因等待新任务而阻塞，因此没有必要保留环境。其次，创建一个新线程比存储一个存在的线程花费少。最后，由于不需在服务器线程内将到来的消息复制入缓冲区，从而节约了时间。其他许多技术使用也可减少开销。总而言之，可观地提高速度是可能的。

线程是一个正在进行研究的课题。一些其他的成果在（Marsh 等,1991: 和 Draves 等,1991）中介绍。

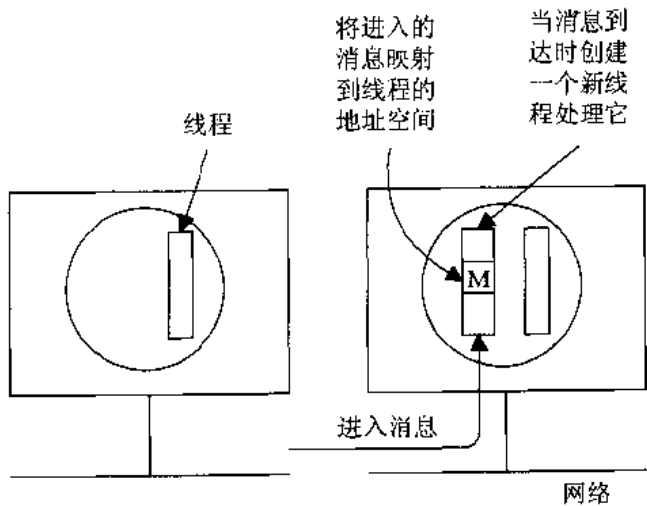


图 4-8 当消息到达时创建线程

4.2 系统模型

进程在处理机上运行，在传统系统中仅有一个处理机，因此不会出现怎样利用处理机的问题。而在有多个处理机的分布式系统中，这就成为一个主要的设计问题。分布式系统中的处理机可用多种方式组织。本节将介绍两种主要的模型：工作站模型(workstation model)和处理机池模型(processor pool model)，以及一个包含了这两个模型各自特点的混合模型。这些模型是建立在关于分布式系统的根本不同的理论体系之上的。

4.2.1 工作站模型

工作站模型是简单的，系统是由分布在建筑物中或校园中并由高速局域网连接起来的工作站（高性能终端个人计算机）构成的，如图 4-9 所示。一些工作站可能在办公室中，因而暗示这些工作站属于单独用户，而另外一些在公共环境中，因此，一天中有许多不同的用户。在两种情况下，在任意时刻，一个工作站或者有一个用户登录，因此就有一个“所有者”（虽然是暂时的），或是空闲。

在有些系统中有的工作站有本地磁盘，而另外一些没有，后者通常称为无盘工作站，但前者称为是多盘工作站(diskfull workstation)，或有盘工作站(disky workstation)，或甚至是一些更奇怪的名字。如果工作站是无磁盘的，文件系统必须由一个或多个远程文件服务器实现。读写文件的请求将发送到文件服务器，由它执行并返回结果。



图 4-9 个人工作站网络，每个工作站都具有文件系统

无盘工作站在大学或公司中流行有几个原因，并不仅因为它的价格。买大量的带有低速小容量磁盘的工作站比买一台或两台配置有高速、海量磁盘并可通过 LAN 访问的文件服务器昂贵得多。

无盘工作站流行的第二个原因是它们容易维护。当某个软件的新版本出现时（例如某种编译程序），系统管理员将很容易地将它安装在机房中很少的几台文件服务器上，但将其安装在遍布于建筑物或校园内的几十台或几百台机器上则完全是另一回事。对于一个集中放置的 5-gigabyte 容量的磁盘在备份和硬件维护上也要比散布于整个建筑物中的 50 个 100-megabyte 容量的磁盘简单。

不用磁盘的另一个原因是因为磁盘有机风扇并产生噪音，许多人感到这种噪音令人讨厌且不希望它们在办公室中出现。

最后，无盘工作站提供了对称性和灵活性。用户能走近任一工作站且登录。因为他的所有的文件都在文件服务器上，各个无盘工作站其实是一样的。相反，当所有文件存储在本地磁盘上时，使用别人的工作站意味着你容易存取他人的文件，但是访问你自己的文件将需要付出额外的努力，并且肯定与使用你自己的工作站不同。

当工作站有私人磁盘时，这些磁盘至少能以下列四种方式使用：

- (1) 分页和临时文件；
- (2) 分页，临时文件，和系统二进制文件；
- (3) 分页，临时文件，系统二进制文件和文件缓存；
- (4) 完整的局部文件系统。

第一种设计是基于这样的认识，虽然将所有的用户文件都保存在中央文件服务器可能很方便，但文件仍需要磁盘用来分页（或交换）和存储临时文件。在这种模型中本地磁盘仅用于分页和存储是临时的、不能共享的、并在登录会话结束后丢弃文件。例如，大多数编译程序包含多个检查过程，每个过程都创建一个下一级过程所要读的临时文件。当文件读过后，就将它丢弃。本地磁盘是存储这样文件的理想场所。

第二种模型是第一种模型的变型，本地磁盘仍然可以保存二进制（可执行）程序。如：编译程序、文本编辑程序和电子邮件处理程序。当调用其中的某一个程序，它将从本地磁盘而不是从文件服务器上加载，这样大大减少了网络的负载。由于这些程序很少改变，它们可以安装在本地磁盘上并可保存很长时间，当新版本的系统软件可用时，可将它发送到所有的机器上。然而，当发送这个软件时某台机器正好停机，那么它将错过这个机会，但可继续运行老的版本。因而，需要一些管理程序来跟踪用户程序使用的是哪个版本。

第三种使用本地磁盘的方法是使用它们作为缓冲区（并且将它们用于分页，临时文件和二进制文件）。在这种模型中，用户能从文件服务器下载文件到本地磁盘上，在本地读写这些文件，然后在登录会话结束之前上载修改后的文件。这种结构的目的是保持长期集中存储，但又能通过当使用文件时把它们保存在本地的方法而减少网络负载。其缺点是需保持缓存一致性，如果两个用户同时下载同一文件并以不同的方式修改它，将产生什么后果呢？这个问题不是很容易解决的，我们将在本书的后面对它进行详细的讨论。

第四种模型，每台机器都能有自己独立完备的文件系统，且能登录或存取其他机器的文件系统，在这里，主要思想是每台机器基本上是独立的并与外界进行有限的联系，这种组织结构对用户提供了统一的和固定的响应时间且网络负载很小，主要缺点是共享困难，实际上这种系统更接近于网络操作系统而不是真正的透明的分布式操作系统。

我们所讨论的一种无盘模型和四种有盘工作站模型在表 4-2 中进行了比较。表中从上到下的改进是从完全依靠文件服务器到彻底独立于服务器。

工作站模型的优点很多且显而易见。这种模型容易理解。用户有固定数量的专用计算能力，并且有保障的响应时间。能非常快的执行复杂的图形应用程序，因为它们能直接存取屏幕，每个用户有很大程度的自主性，并且能根据他的需要灵活地分配工作站资源，本地磁盘增加了这种独立性，并可使工作站在文件服务器崩溃时能继续以较小或较大的程度工作。

表 4-2 在工作站使用的磁盘

对文件服务器的 依赖程度	硬盘使用	优点	缺点
	无盘	成本低,软硬件容易维护,对称灵活	网络使用比较频繁,文件服务器可能会成为瓶颈
	分页,可擦除文件	和无盘相比,使网络的负担更轻	由于需要大量的磁盘,费用比较高
	分页,过期文件, 二进制文件	使网络的负担更轻	费用比较高,二进制的更新比较复杂
	分页,过期文件, 二进制文件,缓冲	减轻网络的负担,也减轻文件服务器的负担	费用比较高,存在 cache 一致性的问题
	完全本地文件系统	几乎没有网络负担,消除了对文件服务器的需要	失去了透明性

然而,这种模型也有两个问题。首先,随着处理机芯片价格不断下降,给每个用户 10 个然后是 100 个 CPU 不久将变得经济可行。一个办公室内有 100 台工作站将(因为拥挤)使你很难看到窗外的世界。其次,用户大多数时间不使用工作站,因此它们空闲。而其他用户可能需要更多计算能力但却不能得到。从系统的角度看,系统中一些用户分配给它们一些资源,但它们并不需要,而另一些非常需要这些资源的用户却得不到,这样系统效率非常低。

第一个问题能通过使每台工作站变成私有的多处理机来解决。例如,屏幕上的每一个窗口都能有一个专用的 CPU 运行它的程序。然而,从一些早期的个人多处理机系统如 DEC 的 Firefly 所提供的初步证据来看,所使用的 CPU 的平均个数几乎不超过一个。这是由于用户几乎并不同时运行多个进程。此外,这对资源的使用效率很低,但由于技术的进步而使 CPU 变得越来越便宜,浪费它们将变得并不那么重要。

4.2.2 使用空闲工作站

第二个问题,关于空闲工作站,已经成为大量研究的主题。这主要是因为许多大学有大量的个人工作站,其中的一部分是空闲的。统计数据显示,即使在每天中午的高峰期内,在任意时刻,仍然有 30%的工作站是闲置的,晚上空闲的则更多。已经提出了许多方案来解决空闲的工作站或未被充分利用的工作站(Litzkow 等,1988;Nichols,1987;和 Theimer 等,1985)。本节将讨论关于这个问题的基本规律。

最早的尝试允许使用空闲工作站的程序是 Berkeley UNIX 中的 *rsh* 程序,这个程序调用方式是:

```
rsh machine command
```

第一个参数是机器名字,第二个参数是指定机器上要运行的命令。*rsh* 的作用是在指定的机器上运行指定的命令。尽管这条命令被广泛使用,但这个程序有几个严重的缺陷。首先,用户必须告诉使用哪台机器,将跟踪空闲机器的任务完全交给了用户;其次,程序在远程机器的环境中执行,而这种环境通常不同于本地机器的环境。最后,如果某用户登录到有一个远程进程正在运行的空闲机器上,这个远程进程会继续运行,且新登录的用户,要么忍受较低性能,要么就使用另一台机器。

关于空闲工作站的研究集中于解决以下几个问题，这些关键问题是：

- (1) 怎样找出一台空闲机器？
- (2) 远程进程怎样透明地运行？
- (3) 如果（空闲）机器的主人回来重新使用它时怎么办？

让我们讨论这三个问题，

怎样找出一台空闲的工作站？为了便于讨论，首先我们要弄清什么是空闲的工作站。表面上看，好像是没有人从控制台上登录的工作站是空闲工作站，但对于现代计算机系统来说并不总是这么简单。在许多系统中，即使没有人登录进机器，也会有许多进程在它上面运行，如时钟、邮件、新闻等其他一些后台监视进程，另一方面，某用户在早上到达办公室后，登录进入他的机器，然后就几个小时内没有使用计算机，几乎没有给计算机施加任何额外的负载。不同的系统有不同的标准决定什么是空闲的工作站。但典型地，如果几分钟内没有人接触键盘或鼠标，并且没有用户启动的进程在运行，这台工作站就可被认为是空闲的。因此，在负载方面空闲工作站之间也是有很大的区别。例如，由于邮件量进入一台空闲工作站而未进入另一台空闲工作站。

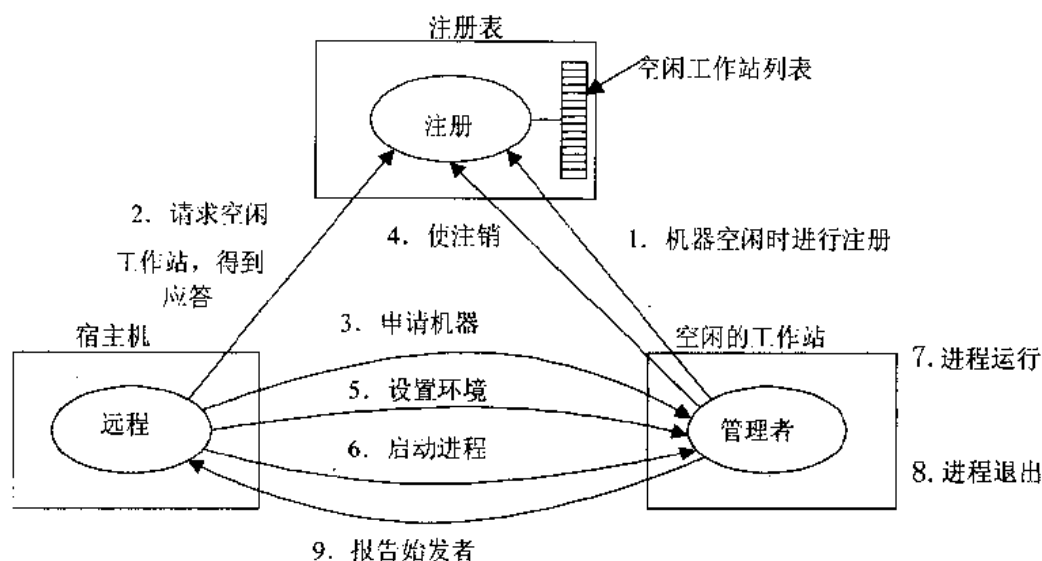
定位空闲工作站的方法分成两类：服务器驱动的和客户端驱动的。对于前者，当一台工作站空闲时，就成为一台潜在的服务器，它声明其可用性，是通过将它的名字、网络地址和属性输入到一个注册文件中（或数据库）来声明其可用性的。例如，当某用户想在空闲工作站上执行一条命令，他输入：

`remote command`

远程程序查看注册文件以寻找一个合适的空闲工作站，由于可靠性的原因，注册文件可有多份拷贝。

对于最近空闲的工作站，声明它为空闲的另一种办法是，通过在网络上广播发送一条消息。所有其他的工作站记录这条消息。实际上每台机器维护它自己私有的注册拷贝。这样做的好处是在寻找一台空闲工作站时需要较少的系统开销并且有更高的冗余性。缺点是需要所有的机器都要维护注册文件。

无论是有一个注册文件还是有多个注册文件，都可能发生潜在的竞争危险。如果两个月户同时激活 `remote` 命令，且他们发现的空闲机器是同一台，他们可能同时试图在那台机器上启动进程。为了发现和避免这种情况，`remote`（远程）程序应能检查空闲工作站，



4-10 查找与使用空闲工作站的基于注册表算法

发现空闲工作站仅是第一步，现在进程将在其上运行。移动代码是很容易的，关键技巧是设置远程进程以便使它有与本地工作站一样的环境，因此可使它与在本地进行同样的运算。

开始运行时，如果有的话，它需要同样的文件系统、同样的工作目录和同样的条件变量（外壳变量 shell variables）。这些被设置好后，程序就可开始运行。当第一次执行系统调用时（例如是 read）出现问题，内核该做些什么？答案在很大程度上取决于系统结构，如果系统是无硬盘的，所有文件都在文件服务器上，内核可以仅仅向相应的文件服务器上发送一条请求，与进程在本地机上执行的情形一样。另一方面，如果系统有本地磁盘，每个都有一个完整的文件系统，请求就须返回到宿主机上去执行。

即使所有的机器都是无盘的，一些系统调用也必须返回到宿主机。比如，读键盘和写屏幕永远不能在远程主机上运行，但是其他有一些系统调用在任何情况下都必须在远程主机上执行。比如 UNIX 系统调用 SBRK(调整数据段大小)，NICE(设置 CPU 调度优先权)和 PROFIL（程序计数器计数）不能在宿主机上运行。并且，所有查询机器状态的系统调用必须在进程真正运行的那台机器上运行。它包括询问机器名和网络地址，询问还有多少空闲内存等等。

与时间有关的系统调用会出问题，因为不同机器上的时钟可能不同步。在第 3 章里，我们看到了实现同步是多么困难，使用远程机器上的时间可能导致一些依赖时间的程序给出不正确的结果，例如：make。将所有与时间相关的调用传回到本地机会带来时间上的延迟。这同样也会导致时间上的问题。

进一步考虑复杂的情况，一些特殊的调用如创建和写一个临时文件，原来一般是传送回本地机，但它们在远程机器上执行会更有效。此外鼠标跟踪和信号传送也需仔细考虑，直接写硬件设备的程序如屏幕帧缓冲区、磁盘或磁带是绝不能远程执行的。总之，让程序运行在远程机器上，且看起来它们好像是在本地机上运行，是可能实现的，但这是一种有很高技巧和很复杂的工作。

所列举的最后一个问题是如果机器的主人回来怎么办？(如有人注册进入或前面未激活的用户、碰了鼠标或键盘)，最简单的办法是什么也不做，但这样会与“个人”工作站这个名称不相符，如果在你要使用计算机的时候，其他人可能在你的工作站上运行程序，应该保证机器对你的响应。

另一种可能的方法是杀掉正在进入的进程，最简单的方法是突然的并无警告的这样做，这种方法的缺点是所有的工作信息会丢失，并且可能造成文件系统的混乱状态。较好的方法是给一个适当的警告，这可以通过给它发一个信号来实现，使他预测到他即将到来的“命运”。让它从容地关闭（如把缓冲区的内容写回硬盘，关闭文件等等）。如果在几秒钟内它还没有退出，当然，这个程序就是等待并要处理这个信号的程序，但大多数存在的程序并不是这样的程序。

一种完全不同的方法是将这个进程移植到另一台机器上去，要么移到本地机上，要么移到另一台空闲工作站上。这种移植在实践上很少使用，因为实际的机制相当复杂，困难的是不在于移动用户的代码和数据，而是在于寻找和收集与即将移植的进程相关的内核数据结构。例如，它可能已经打开了多个文件，运行计时器，到来消息队列和其他一些散布在内核周围的散乱的少量信息，这些信息必须从源机器小心地移走并且重新安装在目的机上，这里不存在理论上的问题，但实际工程中的困难却是巨大的。为了了解更多的情况，请看 (Artsy 和 Finkel, 1989; Douglass 和 Ousterhout, 1991; 和 Zayas, 1987)。

在以上两种情况下，当将进程移走后，它应使源机器的状态与最初创建它时一致，从而避免干扰源机器的所有者。换句话说，这种要求意味着不仅该进程要移走，并且它的所有子进程以及这些子进程的子进程也需移走。除此之外，邮箱、网络连接和其他系统范围的数据结构也必须被删除，并且应制定一些规定，用来忽略在进程移走后，由于这个进程而到来的一些 RPC 应答和其他一些消息。如果是本地磁盘，必须删除临时文件，并且如果可能，必须恢复任何不得不从缓存中删去的文件。

4.2.3 处理机池模型

尽管使用空闲工作站对系统而言增强了计算能力，但它还不能解决更基本的问题，当可以提供相当于用户十倍至百倍数量的 CPU 时，怎么办？一种解决方案是给每人一个私有的多处理机环境，然而这是一种不太高效的设计。

另一种方法是建造处理机池 (processor pool)，在机柜中放满了 CPU，它们可以根据需要动态地分配给用户，处理机池模型如图 4-11 所示。在这种模型中不是给用户个人工作站，而是给他们高性能的图形终端，如 X 终端（虽然小的工作站也可用作终端）。这种想法是基于这样的观察，就是许多用户真正想要的是高质量的图形接口和良好的性能。概念上，它更像传统的分时系统，而不是个人计算机模型，尽管它是用现代技术(低价微处理机)构建成的。

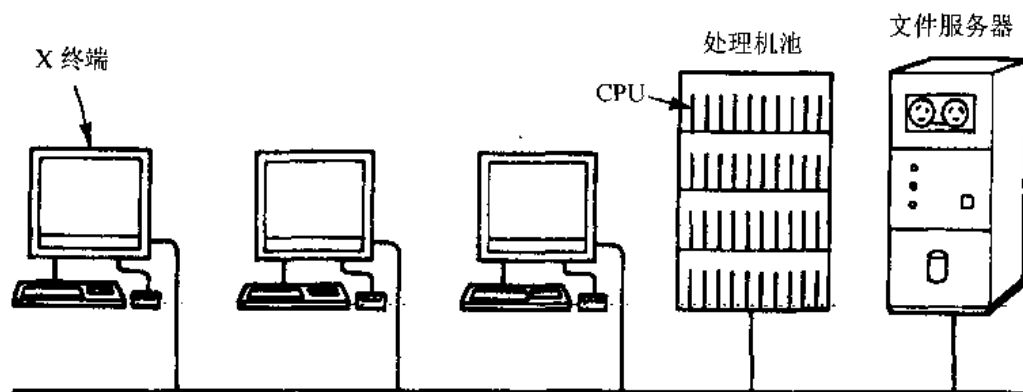


图 4-11 一个基于处理机池模型的系统

建造处理机池的动机是来自对无盘工作站的进一步考虑，如果文件系统为了规模经济而能集中在少数文件服务器上，那么计算机服务器同样也可这样做。将所有的 CPU 放在机房的架柜上，可以减少用电量和包装费用，因而可在给定资金的情况下提供更多的计算能力，更何况它允许使用更廉价的 X 终端(甚至一般的 ASCII 终端)，使用户数与工作站数目无关，从而增加了用户数。这种模型也允许简单的线性增加，如果计算负载增加了 10%，你只需多买 10% 的处理机放在池中就可满足需要。

实际上，我们是将所有的计算能力转变为能够动态访问的“空闲工作站”，用户可以在短时间内分配到他们所需要数量的 CPU。用完后，释放这些 CPU 返回处理机池中供其他用户使用，这里没有拥有的概念，所有的处理机对每个人都是平等拥有的。

将计算能力集中在一个处理机池中的最有力的根据是排队论。一个排队系统是这样一种情况：用户随机地请求服务器工作，当服务器忙时，用户排队等待服务，并按次序给予服务，排队系统的通常例子是面包房、机场登记系统、超市收款系统等等。基本思想如图 4-12 所示。

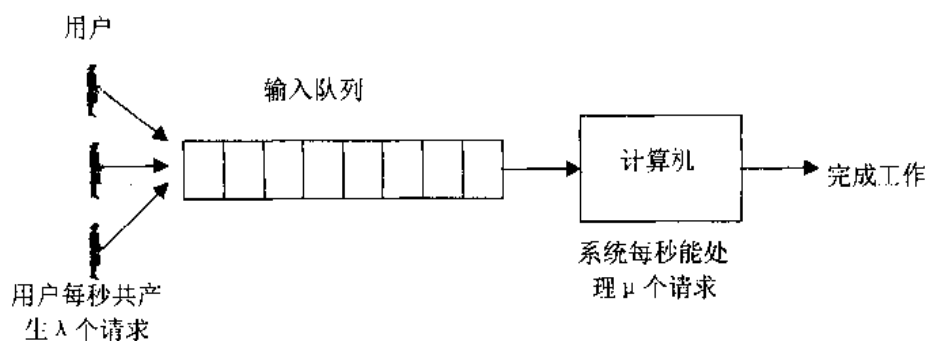


图 4-12 一个基本的排队系统

排队系统很有用，因为它可用作模型分析。我们设每秒钟从所有用户来的总的输入请求速度为 λ ，设 μ 为服务器处理请求的速度，为了达到稳定处理，必须使 $\mu > \lambda$ ，如果

服务器每秒能处理 100 个请求，但用户每秒却不断地产生 110 次请求，队列会无限的加长，(少量时间的输入速度超过服务速度是可以接受的，只要平均输入速度比服务速度慢，并且有足够的缓冲区)。

可以证明 (Kleinrock,1974)，发出请求和得到完全响应之间平均的时间间隔 T ，与 λ ， μ 的关系是

$$T = 1 / (\mu - \lambda)$$

例如：一个文件服务器每秒能处理 50 次请求，而每秒只接收到 40 次请求，那么平均响应时间为 1/10 秒或 100 毫秒，注意当 λ 为 0 时 (没有负载)，文件服务器的响应时间不是 0 而是 1/50 秒或 20 毫秒，原因显而易见，如果文件服务器能每秒处理 50 个请求，没有任何竞争，即使是它处理单个请求也得 20 毫秒，因此包含了处理时间的响应时间决不会低于 20 毫秒。

假设我们有 n 个私有多处理机，每个有多个 CPU，每个都成为一个请求到达速度为 λ 的独立的排队系统，并且 CPU 的处理速率为 μ ，平均响应时间为 T 。设想如果将所有的 CPU 放在一个处理机池中会发生什么情况？取而代之的不是 n 个小的并行运行的排队系统，而是有一个输入率为 $n\lambda$ 服务率为 $n\mu$ 的大系统。将这个合成系统的平均响应时间记为 T_1 ，根据上面的公式，我们可得到

$$T_1 = 1 / (n\mu - n\lambda) = T/n$$

这个令人惊奇的结果说明：用一个是小系统功能 n 倍的大系统来代替 n 个独立的小系统的资源可把平均响应时间减少为原来的 $1/n$ 。

这个结论很通用且已用于很多不同的系统。这也是为什么航空公司愿意每 5 小时飞一架有 300 座的 747 飞机，而不愿意每 10 分钟飞一趟有 10 人座的商业喷气机的一个主要原因，这种结果的出现是因为把处理能力划分给各个小服务器 (例如：个人工作站)，每个用户一个，这与随机到达的工作请求极不匹配。通常情况下，只有很少服务器繁忙或过载，但大多数却是空闲的，处理机池模型中就是减少了这种时间的浪费。这也是它总体性能较好的原因。使用空闲工作站的方法在回收 CPU 浪费的 CPU 周期方面很薄弱。但是正如我们所看到的，它却是很复杂的，并且有许多问题。

实际上，排队理论的结论是根本反对分布式系统的论据之一。给出两种选择，一个是集中式 1000-MIPS CPU，另一个是 100 个私有的、专用的 10-MIPS CPU，因为没有浪费 CPU 周期，前者的平均响应时间将是后者的 100 倍。机器仅仅是在用户没有任何工作要做时才闲置，这样的事实证明了尽可能的集中计算能力的好处。

然而，平均响应时间并不代表一切。分布式计算也有其优点，如费用。一个 1000-MIPS CPU 要比 100 个 10-MIPS CPU 昂贵得多，后者的性能价格比要高得多，不惜一切代价建造那样巨大的机器甚至是不可能的。

而且，个人工作站有统一的响应，不依赖于其他人正在做的事 (除非当网络或文件服务器阻塞时)，对于许多用户来说，把响应时间的恒定性比平均响应时间看得更重要。例如：在个人工作站上编辑文本时，请求显示下一页要花费 500 毫秒的时间；而在大型的、集中式共享的计算机上请求显示下一页有 95% 需要 5 毫秒，5% 需要 5 秒，20 次里有一次，

尽管平均响应时间是工作站的一半，但用户可能认为这种性能难以忍受。另一方面当用户需要运行一个很大的仿真程序，大的计算机可能会更受欢迎。

到目前为止，我们假设了一个有 n 个处理机的处理机池与一个具有处理机池中单个处理机 n 倍处理能力的一个处理机是等效的。实际上，这个假设仅当将所有的请求分配在所有的处理机上并行处理时才是正确的。假设，一个作业仅可以分为 5 部分，处理机池模型的有效服务时间只是单个处理机的 5 倍，而不是 n 倍。

同时，处理机池模型，相对于寻找空闲工作站然后悄悄登录到那个工作站，是一种更方便的方法。基于没有任何处理机属于某个特定人的假设，同时基于以下的想法我们产生了这样的设计，即从处理机池中请求处理机，使用它们，用完后放回池中。这也没有必要把发出的内容返回到“原来的”机器，因为根本就不存在这种情况。也没有机器的拥有者回来时产生的危险，因为机器就没有拥有者。

最后，所有情况都归结于工作负载的本质，如果所有人都只做简单的编辑且偶尔发几封电子邮件，使用个人工作站可能就足够了。另一方面，如果用户从事于大型软件的开发工程，经常在大目录下运行 *make* 命令，或尝试转置一个大的稀疏矩阵或做大型模拟，或运行大型人工智能程序，或不断的寻找大量空闲工作站的 VLSI 路由程序，使用个人工作站将是很麻烦的。在所有这些情况下，处理机池的思想从根本上说更简单并更有吸引力。

4.2.4 混合模型

一种可能的折衷是提供给每个用户一个私有工作站并附加有处理机池，虽然这种方法的开销比单纯的工作站模型或单纯的处理机池模型更加昂贵，它却综合了两者的优点。

为了保证响应时间，交互式的工作能在工作站上运行。然而，为了使系统设计更简单，空闲工作站放置不用。另一方面，所有的非交互式进程运行在处理机池中，就像所有通常执行繁重的计算一样。这种模型具有快速的交互响应、有效的资源利用和简单的设计的优点。

4.3 处理机分配

根据定义可知，分布式系统由多个处理机组成。这些处理机的组织形式可能是工作站集合、一个公共的处理机池、或是某种混合的形式。无论是何种形式，都要有一种算法来决定在哪台机器上运行哪个进程。对于工作站模型，问题主要是何时在本地运行一个进程以及何时去寻找一个空闲的工作站运行进程。对于处理机池模型，每个新的进程的运行都要作出决定。本节我们将要讨论处理机分配的算法，在讲述过程中，我们将遵从习惯称之为“处理机分配”，而不是“进程分配”，尽管使用后者会便于阐述（举例）。

4.3.1 分配模型

在讨论特定的算法和设计原则之前，一些关于处理机分配的基本模型、假设、及其目标的问题值得我们先讨论一下。这个领域中几乎所有的模型都假设所有的计算机都是一样的，或至少是代码兼容的，至多在速度上有所不同。只有个别模型假设系统是由多个不相关的处理机池组成，每个处理机池是同构的。这些假设通常都是有效的，并且能够简化问

题，但同时也遗留下一些问题，如：假设一个文本处理程序有适合于各种机器的可运行代码，那么启动该程序的命令行应该在 486、SPARC、还是在 MIPS 处理机上运行？

几乎所有的已有模型都假设系统是完全连接的，即每台处理机都能与其他任何一台处理机进行通信。我们也将这样假设，但这并不意味着每台机器都用连线与其他所有机器相连，只是说每对机器之间都能够进行通信连接，至于信息是如何一步一步到达目的地的问题是低层软件所要考虑的，与我们无关。一些网络支持广播通信和多址通信，有些算法就利用了这些特性。

当一个运行的进程决定派生或创建一个子进程时，新的工作随之产生。在某些情况下，进行派生的进程是命令解释程序（shell），它为应答用户输入的命令而启动新的任务；在另一些情况下，一个用户进程本身创建一个或多个子进程，例如：为了获得较高的系统性能，创建多个子进程并行运行。

处理机分配策略可以分为两大类：一类是不可迁移的，另一类是可迁移的。在不可迁移的策略下，当创建一个进程时，系统就决定为该进程分配哪台处理机。一旦分配完毕，进程将一直在这台处理机上运行，直至结束。无论这台处理机是多么严重的超载，也不管其他处理机有多少空闲，它都不能移到其他的处理机上。相反，在可迁移的策略下，还可将它即使已经开始运行的进程迁移到别的处理机上继续运行。可迁移策略能够提供更好的负载平衡，但同时也增加了系统的复杂性，并对系统的设计有很大的影响。

对于处理机分配算法，我们总是希望对它进行些优化。如果不是这样，我们只要为新的进程随机分配一个处理机或以数字顺序分配就可以了。然而，准确地说各个系统想要优化的内容各不相同，一个可能的目标是使 CPU 的利用率最大化，即：每小时中，使实际运行用户任务的 CPU 周期数最大，也即尽可能地减少 CPU 的空闲时间，让每个 CPU 都在运行。

另一个有价值的目标是使平均响应时间最小化。如图 4-13，有两个处理机和两个进程。处理机 1 以 10MIPS 的速度运行，处理机 2 以 100MIPS 的速度运行，但有一个后备进程的等待列表需耗时 5 秒才可完成其中的进程。进程 A 有 1 亿条指令，进程 B 有 3 亿条指令。每个进程在每个处理机上的响应时间（包括等待时间）在图中已显示出来。如果让处理机 1 运行进程 A，处理机 2 运行进程 B，那么平均响应时间是 $(10+8)/2=9$ 秒。如果反过来，平均等待时间是 $(30+6)/2=18$ 秒。很显然，前者方案在最小化平均响应时间上要优于后者。

最小化响应时间的一个变形是最小化响应率，响应率的定义是在一台机器上运行一个进程的时间除以这个进程在一个无负载的标准处理机上运行时应该花的时间，对于许多用户来讲，响应率是一个比响应时间更加有效的参数，因为它考虑到了大的作业会比小的作业花更多的时间这样一个事实。要更好的看清这一点，只需比较一下下面两种情况，如果一个 1 秒的任务花了 5 秒时间完成，而一个 1 分钟的任务花去了 70 秒，你认为哪个更好一些？用响应时间来算，前者要好一些，用响应率来算，则是后者要好得多，这是因为 $5/1 >> 70/60$ 。

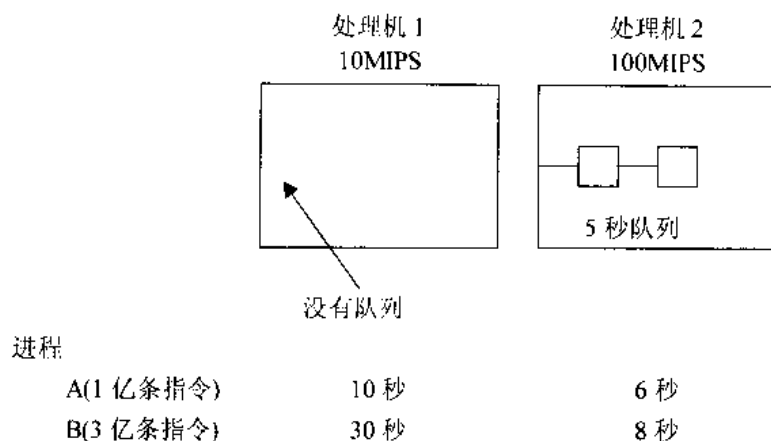


图 4-13 在两个处理机中两个进程的响应时间

4.3.2 处理机分配算法的设计问题

近年来，人们提出了大量的处理机分配算法。本节将讨论在这些算法中所涉及的一些关键问题以及一些不同的折衷方案。在设计算法时，设计者要考虑的问题可概括为以下五个方面：

- (1) 确定性与启发性（试探性）算法；
- (2) 集中式与分布式算法；
- (3) 最优与次优算法；
- (4) 本地与全局算法；
- (5) 发送者发起与接收者发起算法。

当然，还有另外一些选择，但这五个是最主要的，也是近年来广泛研究的问题。下面来分别讨论这些问题。

确定性算法适用于当进程的所有行为都是事先知道的情况。设想一下如果你有一张所有进程的清单：这些进程的计算需求、文件需求、通信需求等等，有了这些信息，那么设计一个完美的分配方案是可能的。从理论上说，可以尝试所有的可能，从而找到一个最优解。

只有极少数系统的所有信息是预先可知的，但有时可得到一个合理的近似。例如：银行、保险、航空订票系统中，每天都很相像。航空公司甚至能很好地预测在早春季节，某个星期一早上有多少人要从纽约飞往芝加哥，所以很容易精确地描绘出系统负载的特性。至少，从统计上来说，使用确定性算法是可能的。

另一个极端是系统的负载是完全不可预测的。工作请求视用户正在做什么而定，并且每个小时，甚至每分钟都有可能发生很大的变化。在这种系统中，处理机分配不能用数学的、确定性的算法来实现，而只能使用特别的启发性（试探性）算法。

第二个设计问题是集中式与分布式，这个问题反复贯穿于本书。把所有信息搜集到一个地点便于作出更好的决定，但这使得系统不够健壮，而且使中心机器的负载过重。通常倾向于采用分布式算法，但一些集中式算法仍被采用，因为缺少相应的可替代的分布式算

法。

第三个设计问题与前两个问题有关。我们是需要一个最优解呢，还是一个可接受的解就可以呢？用集中式和分布式算法都能得到最优解，但都比得到次优解的代价大得多。要得到最优解必须搜集更多的信息，并进行更全面的处理。在实际中，大多数分布式系统目标定位在启发性的、分布式的、次优的解决方案上，因为要得到最优解太困难了。

第四个设计问题与常说的转移策略有关。当创建一个新进程时，系统要决定是否让这个进程在创建它的机器上运行。如果那台机器太忙，这个新的进程应该被转移到别的机器上去运行。这时就要决定是否只根据本地的信息来决定转移策略。一种学派主张一种简单的（本地）算法：如果机器的负载低于某个阈值，就让该进程在这台机器上运行，否则就要清除该进程。另一种学派则认为这种启发性算法过于原始。最好是在对本地机器是否太忙作出决定前，能够为那个进程搜集其他地方的（全局）信息，看一看别处的负载，再决定是否在本机上运行该进程。每派都有自己的理由，局部算法简单，但远远达不到最优；然而全局算法只不过能稍好一些，但却付出了更大的代价。

最后一个设计问题与定位策略有关。一旦转移策略决定要清除一个进程，定位策略就要决定将该进程发送到何处。很明显，定位策略不能是本地的，它需要别处的负载信息来对进程传输到何处作出明智的决定。这个信息可以用两种方法来传递：一种是发送者发起，另一种是接收者来启动。

作为一个简单的例子，在图 4-14 (a) 中，这里有一台过载的机器，它向别的机器发送请求希望得到帮助，它希望将一些新进程卸载到别的机器上。在这个例子中，是由发送者主动定位其他 CPU 的。相反，在图 4-14 (b)，这里有一台空闲的或者说是欠载的机器，它向别的机器发送信息，声明自己是空闲的，可以承担一些额外的工作。它的目的是找到一台愿意给它些事情做的机器。

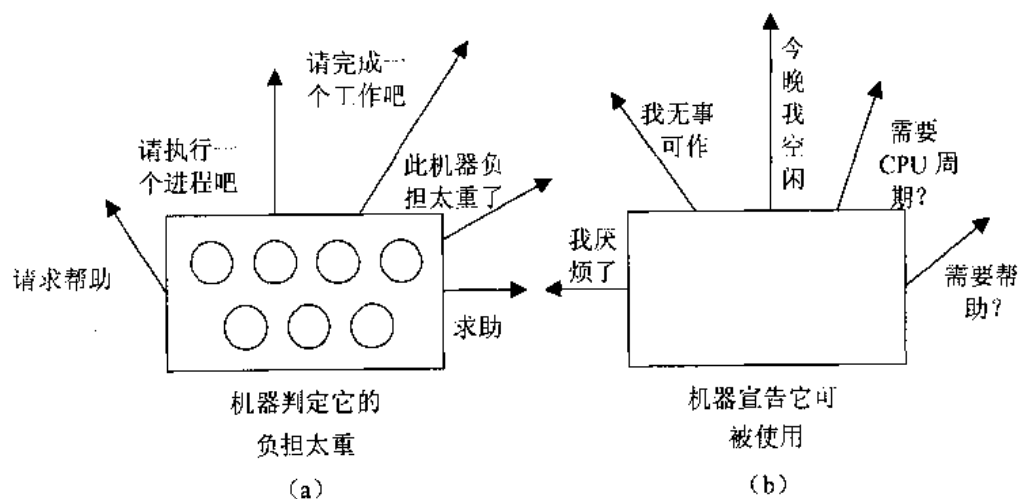


图 4-14 (a) 发送者寻找空闲机器
(b) 接收者寻找工作做

不论是发送者发起的还是接收者发起的情况，不同的算法有不同的策略来决定由谁来搜集信息，及搜集多长时间，和怎样处理结果。然而，到目前为止，两种不同方法的区别

应该很清楚了。

4.3.3 处理机分配算法的实现问题

前面几节中提到的全部问题都是理论上清晰明确的内容，它们可以引起无止境的并且很奇妙的争论。本节将介绍与实现处理机分配算法的实质性细节更相关的一些其他问题，而非细节后的那些重要的原理。

在开始之前，所有算法假定每台机器都知道自己的负载情况，能判断它们自己是否过载或欠载，并可告诉其他机器它们的状态。测量负载状态并不像看上去的那么容易，一个简单的方法是：计算每台机器上的进程数量，并将它作为负载情况。但是，正如我们前面所提到的，即使在一台空闲机器上，也会有许多进程在运行，如新闻、邮件、守护进程，窗口管理程序及其他一些进程。所以用进程数表示当前的负载情况，几乎说明不了任何问题。

进一步的解决方法是只对处于运行或就绪状态的进程计数。毕竟，每个运行的或可运行的进程都会对系统施加一些负载，即使它是一个后台进程。然而，许多守护进程周期性地被唤醒，检查是否有它感兴趣的事情发生，如果没有，则又进入睡眠状态，所以它们大多只对系统施加很小的负载。

一种更直接的测量方法是采用（虽然可能需要更多的工作）CPU 忙的时间所占的比例。很显然，一个利用率为 20% 的处理机比一个有 10% 利用率的处理机要忙，不论它运行的是一般用户程序或是守护进程。一种测量利用率的方法是设置一个计时器，并让它周期地中断机器，在每次中断时检查 CPU 的状态，这样就能得到 CPU 空闲时间所占的比例。

使用计时器中断的一个问题是当核心程序运行关键代码时，它通常会屏蔽所有中断，包括计时器中断。这样当核心代码运行时，发生计时器中断，它将延迟到核心代码运行完毕。如果核心代码正在阻塞前一个活动进程，计时器中断将在核心代码结束并进入空闲循环后才可发生。这可能会造成过低 CPU 的利用率。

另一个实现方面的问题是如何处理额外开销。许多理论上的处理机分配算法都忽略了测量处理机搜集信息和转移进程的额外开销。如果一个分配算法发现将一个新进程传输到远程机器上将使系统性能仅提高 10% 时，最好还是什么也别做，因为转移进程的花费将抵消掉所得的获益。一个好的算法应该考虑到算法本身所占用的 CPU 时间、内存的使用以及网络带宽。但现在很少有算法考虑这些因素，主要是因为这样做不太容易。

实现中还有一个要考虑的问题是复杂性。实际上，所有的研究者都以分析、仿真或计算 CPU 利用率、网络使用情况和响应时间的实验数据来衡量他们算法的好坏。很少考虑软件的复杂性对系统的性能、正确性和健壮性产生的影响。很少有人会发表一个新的算法，宣称它的性能是多么的好，然后作出结论说这个算法不值得使用，因为该算法的性能只稍好于现有的算法，但在实现上却比现有的复杂得多（或速度慢得多）。

在这一点上，Eager 等人在 1986 年所作的研究给追求复杂的、最优的算法的人们一丝希望。他们研究了三个算法，在这些算法中，所有的机器测量自己的负载以决定自己是否欠载。每当创建一个新的进程时，创建它的机器检查自己是否过载，如果是，那么它将搜索一个能够运行该进程的远程机器。这三个算法的不同之处只在于如何定位远程机器。

算法 1 随机选择一台机器，直接向它发送新进程。如果接收的机器已经过载，它也同

样随机选择一台机器，向它发送进程。这个过程一直循环，直到找到一台机器愿意接收这个进程，或者是计数器溢出，这时将不再往下转发进程。

算法 2 随机选择一台机器，然后向它发送一个询问，询问它是过载还是欠载。如果这台机器声明它是欠载，它就会得到这个进程；否则，将重新选择一台机器向它发送询问。这个过程一直循环，直到找到有一台合适的机器或者是超过了询问次数限制，这时进程将停留在创建它的机器上。

算法 3 询问 k 台机器，确定它们的确切负载情况。将新过程传送到负载最小的那台机器上。

显而易见，如果我们忽略所有询问和进程传输所带来的额外负载，人们会认为算法 3 应该是性能最好的。事实上确实是这样。但是算法 3 所得到的性能只比算法 2 稍高一点，而它的复杂程度和所需的额外工作量却要大得多。Eager 等人推断，如果使用一个简单的算法获益接近于使用昂贵并复杂得多的算法所带来的收益，那么，采用简单的算法会更好些。

最后一点是稳定性问题，它是新出现的一个问题。不同的机器都在异步地运行它们的算法，所以整个系统很少是平衡的。可能会出现机器 A 和机器 B 都没有最新信息的情况，因而它们都认为对方的负载比较轻，从而可能造成某个可怜的进程在它们之间被踢来踢去。问题在于大多数交换信息的算法在系统信息交换完毕，并且所有事情都处理完后，工作看起来很正常，但当表格正在被更新时，对其运行情况知道甚少。正是由于非平衡系统环境，才产生了这种问题。

4.3.4 处理机分配算法举例

为了使大家对算法如何实现有一个更深入的理解，本节将讨论几个不同的算法。在选择时，我们尽量使这些算法覆盖面广，但还是会有其他的情况。

图论确定性算法

一类广泛研究的算法是基于这样的一些系统，构成它的进程知道它所要求的 CPU 和内存要求，并且知道系统中每对进程之间要求的平均通信量。如果系统的 CPU 数量 k 小于进程数，则要求将多个进程指派到同一个 CPU 上。整个算法的思想是使这种指派能够使网络的通信量最小化。

整个系统可以表示为一张带权图，每个节点表示一个进程，每条边表示两个进程之间的通信量。从数学的角度来讲，整个问题就变成了如何根据特定的限制将图划分成 k 个不相连的子图（如：每个子图的总 CPU 和内存需求在一定限制之内）。对于每种满足限制的解决方案，子图内部的边意味着机器内部的通信，可以忽略。从一个子图连向另一个子图的边表示网络通信。算法的目标就是在满足所有的限制下，找到一种划分方式使网络通信量最小。图 4-15 表示了图的两种划分，这两种划分导致了两种不同的网络负载。

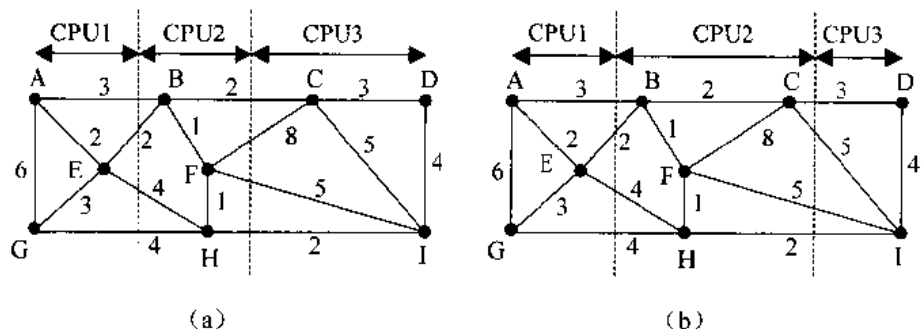


图 4-15 3 个处理机分配 9 个进程的两种方法

在图 4-15 (a) 中，我们将进程 A、E、G 划分在一个处理机上，B、F、H 在第二个处理机上，C、D、I 在第三个处理机上。整个网络的通信量等于与虚线相交的边的权值的和，为 30 单位。在图 4-15 (b) 中，由于采用了不同的划分，网络通信量为 28 单位。如果它符合 CPU 和内存的所有限制，那么这种划分要好一些，因为它要求较小的网络通信量。

显而易见，我们是在寻找紧耦合的聚集。但是这些聚集之间很少相互作用（集合内的通信量大，集合间的通信量小）。有很多文章都讨论过这一类问题，包括：Chow 和 Abraham, 1982；Stone 和 Bokhari, 1978；以及 Lo, 1984。

一个集中式的算法

刚才讨论的图论算法仅有非常有限的应用性，因为它需要事前了解完整的信息。所以让我们来看一个不需要事前了解任何信息的启发性算法。这个算法称为上-下算法（up-down 算法，Mutka 和 Livny, 1987），是一种集中式算法。算法中有一个协调者，保存着一张使用情况表，每个工作站在表中都有一个条目，初值为 0。当有重要的事件发生时，将给协调者发消息以更新使用情况表。算法将根据使用情况表决定处理机的分配。这些决定发生在调度事件发生时：有进程请求处理机、处理机进入空闲状态或者是发生了时钟中断。

这个算法的不同寻常之处，也即它是集中式的原因，就是它所考虑的是使每台工作站的用户公平地享用系统的计算能力，而不是试图使处理机的利用率最大化。而其他的算法可能会因为一个用户首先保证让每台机器都保持繁忙（即：获得很高的 CPU 利用率）而高高兴兴地给他分配所有的处理机，但此算法恰恰要避免这种情况。

当要创建进程时，如果创建该进程的机器认为这个进程应该到其他的机器上去运行，它将请求协调者给它分配处理机。如果有可分配的处理机，将准许该请求，否则将暂时拒绝该请求，并且将记录该请求。

当一个工作站用户在其他机器上运行进程时，他的罚分会增加，每秒增加一个确定值，如图 4-16 所示。这个罚分将加在它的使用情况表中的条目记录上。当有被拒绝的申请时，将从使用情况表中减去罚分。当没有等待的请求，处理机也未被使用时，表中条目记录的值将移去一个值，从而使之越来越接近 0，直到为 0。在这种情况下，分数是忽上忽下的，这就是算法名字的由来。

使用情况表中的记录值可以为正数、0 或是负数。正数表示用户纯粹是在使用系统资

源；负数表示用户需要系统资源，0 表示介于中间。

现在可以解释算法的启发性了，当一个处理机变成空闲时，首先满足分数最低的正在等待的申请。这样结果是：一个等待了很久的、没有使用任何处理机的申请将总会优先于已经占用了许多处理机的申请。这个特性就是这个算法的目标，即公平地分配系统资源。

在实践中这意味着如果一个用户已经连续使用系统了一段时间，另一个用户申请开始一个进程，较小负载的总会优先于较大负载的。模拟研究表明（Mutka 和 Livny, 1987），这个算法在各种负载情况下运行得很正常。

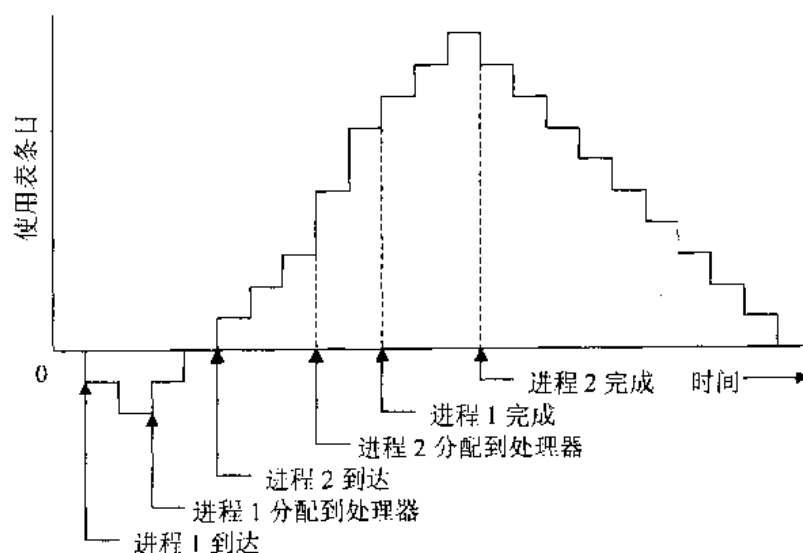


图 4-16 上-下(up-down)算法的操作

层次式的算法

集中式的算法，例如上-下算法，不能很好地适应大型系统。中央节点很快会变成系统瓶颈，更不要说是某个节点失效了。这些问题可以用层次式算法来解决。层次式算法基本保持了集中式算法的简单性，并且能够很好地适应大型系统。

人们已经提出了一个监视处理机集合的方法，就是将所有的处理机以一种与网络物理结构无关的方式组织成一个逻辑分层结构，像在 MICROS（Wittie 和 Van Tilborg, 1980）中一样。这种组织机器的方式就好像公司、军队、学校等现实世界中组织人的层次结构一样。一些机器是工作者，另一些是管理者。

对每组 k 个工作，分配给管理者（“系主任”）一个任务，即跟踪各个工作者谁正在忙，谁正空闲。如果系统很大，将会有众多的管理者，所以有的机器要成为“院长”，每个院长管理若干个系主任。如果院长太多，同样也可以用分层组织的办法来解决：设置一个“大人物”，管理若干个院长。这种层次可以这样无限地扩展下去，需要的层次数随着工作者数量呈对数级增长。因为每个处理机只需要同一个上司和少数几个下属通信，因此系统的信息流量就可控制了。

一个显而易见的问题是：如果一个系主任，或更糟糕些，一个“大人物”，停止正常工作（崩溃）了怎么办？一个方案是选择一个出错管理者的直接下级成为临时老板，取代上级的位置。这个选择可以由它的下级共同作出，也可以由它的同级作出，还可以由它的

上级作出。

为了避免单个（不稳定）管理者在层次树的最上层，可以截去最上层，而在最上层形成一个委员会，拥有最高权利，如图 4-17 所示。如果委员会中的一员无法工作，其他成员将会在下一层中选出一名成员取代其位置。

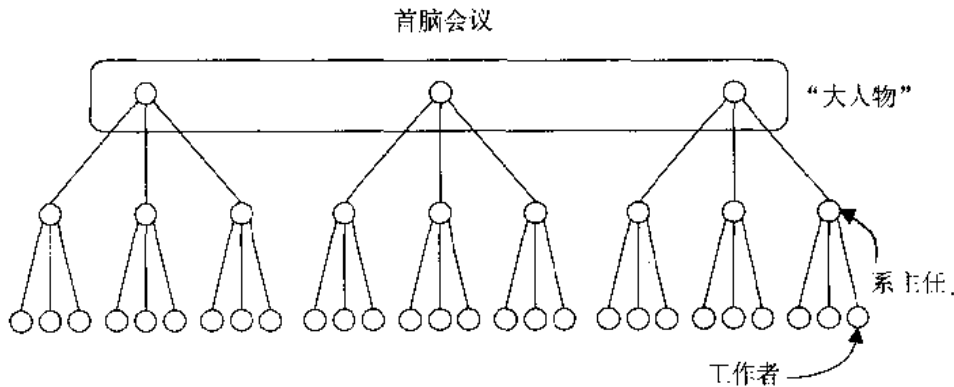


图 4-17 处理机按照原始分层划分模块

尽管这种结构并不是真正分布式的，但它是可行的，并且在实践中也工作得很好。特别是这种系统具有自修复能力，并能够经受无论是工作者还是管理者的突发崩溃，而不会造成长期影响。

在 MICORS 系统中，处理机是单进程分配的。这也就是说，如果突然出现了一个作业要求运行 S 个进程，系统必须为之分配 S 个处理机。作业可以在任意层次创建。所使用的策略是对于每个管理者，跟踪记录它所辖区域内大概有多少工作者（处理机）是可用的（可能包括它下面多层）。如果它认为有足够数量可用，它将预约 R 个处理机，其中 $R \geq S$ ，因为估算不一定十分准确，而且一些机器可能会停机。

如果收到请求的管理者认为可用处理机太少，它将申请沿树向上传递给它的上级。如果上级也无法处理，这个请求将一直向上传递，直到到达有足够可用工作者数（处理机数）的某一级。在这一级，将该申请分割成若干个申请包发送给下级，这个过程将一直持续，直到申请到达最底层。在最底层，处理机将被标记成“忙”，并且实际分配的处理机数目将一级级向上报告。

为了使算法能很好地工作， R 必须选得足够大，以确保能找到足够数量的处理机来处理整个作业。否则申请将不得不向上一层传递重新开始，这样会浪费相当多的时间和计算能力。另一方面，如果 R 选得过大，将会分配过多的处理机，浪费计算能力，直到消息传到顶层，多分配的处理机才能被释放。

由于申请可以随机地在系统的任何层次发出，因而整个系统的情况是非常复杂的。所以，在任何瞬间，分配算法的多个层次上有可能有多个请求，这可能会导致对可用处理机、竞争条件、死锁等等的过时估计，在 Van Tilborg 和 Wittie 的论文中（1981）给出了一个对此问题的数学分析，并详细讨论了这里没有提及的其他方面的一些问题。

发送者发起的分布式启发性算法

前面讨论的算法都是集中式的或半集中式的。也有一些分布式的算法。一个典型的算

法是 Eager 等人在 1986 年提出的。正如前面所提到的，在他们所研究的“最有效代价”的算法中，当创建进程时，创建进程的机器将对一个随机选取的机器发送询问，询问那台机器的负载是否低于某个阈值。如果是，将发送进程，否则，将选择另一台机器并发送询问。这个过程并不会一直持续下去，如果在 N 次询问内还没有找到合适的机器，算法将停止，新进程将在创建它的机器上运行。

已经构造并研究了一个关于该算法的分析队列模型。利用这个模型，已经确定该算法运行得很好并且在各种条件下，包括各种参数：不同的阈值、传输代价、询问次数限制等，都运行得十分稳定。

然而，应该注意到，在负载十分重的情况下，所有的机器都会不停地毫无意义地向其他机器发送询问，想找到一台愿意接受更多工作的机器。在这种情况下，几乎没有进程会被减轻负载，但却引起系统相当可观的额外开销。

接收者发起的分布式启发性算法

与上面的由过载的发送者发起的算法互补的是由欠载的接收者发起的算法。根据这个算法，当一个进程结束时，系统将检查自己是否有足够的工作可做，如果不是，将随机向一台机器申请工作。如果那台机器没有要给予的工作，系统将继续询问第二、第三台机器。如果连续 N 次询问都没有申请到工作，系统将暂时停止申请，开始处理系统队列中一个等待的进程，当这个进程结束后，再开始新一轮申请。如果系统无事可做，则将进入空闲状态，一定的时间后，它从新开始申请。

这个算法的好处就是它不会在系统非常繁忙的时候给系统增加额外的负载。在发送者发起的算法中，一台机器恰恰在系统非常繁忙，即最不能容忍时，发送大量的询问。而在接收者发起的算法中，当系统重载时，一台机器欠载的可能性很小，即使是有机器欠载，它也会很容易地找到工作来做。当然，在系统几乎没有事可做的时候，在接收者发起的算法中，因为每台空闲的机器都拼命的向别的机器申请工作，会造成相当大的询问负载。然而，在系统欠载时使系统开销增大总比在系统过载时这样做要好得多。

还有一种可能的策略是结合两种算法，让机器在过载时设法清除一些进程，而在欠载时设法申请得到一些进程。更进一步，系统内的机器可以通过保留以前的询问和进行随机的查询来决定是否有些机器一直过载或欠载，从而提高性能。是发送者还是接收者先进行查询，这取决于进行查询的进程是要清除进程（发送者）还是要请求获得工作（接收者）。

投标算法

另一类算法则试图将计算机系统变成小型的经济社会，由买卖双方和由需求关系确定的价格组成（Ferguson 等，1988）。算法的核心参与者是进程，为了完成工作，进程必须购买 CPU 时间，而 CPU 则拍卖它的处理机周期给出价最高的买主。

每个处理机在一个公共可读文件中公布他们的近似价格。这个价格并不是确定的，而是为了表示所提供服务的概略价格（事实上，这个价格是最后一个买主的出价）。根据处理机速度、内存容量、浮点运算能力、以及其他一些特性，不同的处理机有不同的价格。所提供的服务的指示，如预期的响应时间，也可以同时公布。

当一个进程要启动一个子进程时，它将查询现在有谁在提供它所需要的服务，然后确定一个它可以支付得起的处理机集。通过计算，它将从这个处理机集中选出一个最好的，根据应用程序的不同，最好的可以指最便宜的、最快的或最高性能价格比的。然后它将与

第一选中的处理机发出一个出价，所出的价格可能会高于或低于已公布的价格。

处理机收集向他们发送的所有出价，然后作出决定，可能是选出价最高的。然后通知选中的和未选中的进程，并开始执行被选中的进程。公共文件中此处理机的价格将被更新，以反映处理机的最新价格。

尽管 Ferguson 等人并没有探究细节，这样一个经济模型还是产生了各种有趣的问题。如：进程从何处得到出价所用的钱？进程有定期的薪水吗？每人的月薪都一样，还是院长高于教授呢？谁又比学生拿得多呢？如果用户增加而相应的资源没有增加，会不会造成通货膨胀（价格升高）？处理机会形成 cartel 同盟（为采取共同的行动而组成的政治或经济联盟）来向用户敲竹杠（漫天要价）吗？用户工会（联合）允许这样吗？磁盘空间使用是否也要收费？那激光打印机呢？这样的问题还有很多。

4.4 分布式系统的调度

对于分布式操作系统调度实在没有太多要说的。正常情况下，每个处理机进行自己本地的调度（假设它上面有多个进程在运行），而不管别的处理机正在干什么。在大多数情况下，这种方法工作得很好，但是当一组相关的并相互影响很强的进程在不同的处理机上运行时，独立的调度就不总是一种最有效的方法了。

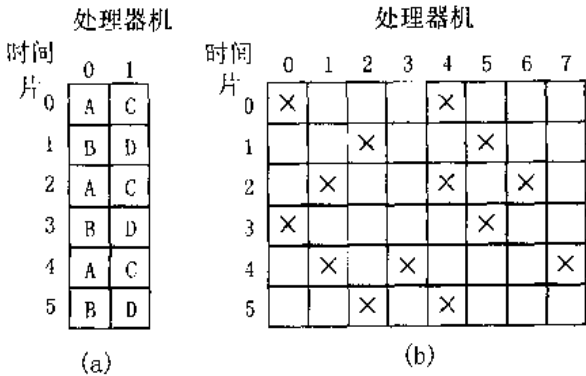


图 4-18 (a) 两个有联系的作业不协调的运行情况
(b) 8 个处理机、每个分为 8 个时间片的调度矩阵
图中 X 代表已分配的时间片

可以用下面的例子说明一种很常见的困难：进程 A 和 B 运行在一个处理机上，C 和 D 运行在另一个处理机上，每个处理机是以 100 毫秒长的时间片进行分时处理的，A 和 C 在偶数号时间片运行，B 和 D 在奇数号时间片运行，如图 4-18 (a) 所示。假设 A 向 D 发送大量的消息或对 D 进行许多远程调用，在时间片 0，A 启动运行并立即调用 D，但 D 正好不在运行，因为现在应该轮到 C 运行。100 毫秒以后，进行进程切换，D 收到了 A 的消息，执行工作并很快进行了答复。但因为现在是 B 正在运行，在 A 得到应答并能继续运行时又要过 100 毫秒。这样的结果是每次消息交换都要花费 200 毫秒。必须要有一种方

法能保证相互间经常通信的进程同时运行。

尽管动态地确定进程间的通信方式比较困难，但在大多数情况下，一组相互联系的进程会同时被启动。例如，通常 UNIX 管道中的过滤器之间的相互通信会多于它们与别的先启动的进程之间的通信。我们不妨假设进程是成组创建的，组内通信大大多于组间的通信，让我们进一步假设，有足够多的处理机可以处理最大的组，并且每个都是有 N 个进程时间片的多进程处理机（ N 路多进程）。

Ousterhout (1982) 提出了数个基于协同调度这个概念的算法，这个算法在调度时考虑了进程间的通信，以确保一个组内的所有的进程同时运行。第一种算法使用了一个概念上的矩阵，每列表示一个处理机的进程表，如图 4-18 (b) 所示。因此，列 4 表示在处理机 4 上运行的所有进程，行 3 表示处于各个处理机时间片 3 的进程，等等。算法的核心是每台处理机使用循环调度算法，启动处理机 0 中时间片 3 中的进程，然后启动处理机 1 中时间片 3 中的进程，如此不断下去。这种想法的主旨是使每个处理机使用轮转调度算法，所有的处理机首先运行时间片 0 中的进程一段固定的时间，然后处理机运行时间片 1 中的进程一段固定的时间，如此一直进行下去。用一个广播消息通知各处理机何时进行进程切换，以保持时间片同步。

将同组内的所有进程成员放置在不同处理机的相同时间片号内，并保证所有的进程同时运行，就可以获得 N 倍并行的好处，并使通信吞吐量达到最大。在图 4-18 (b) 中，为了满足最优性能，当有 4 个必须通信的进程时，应该放到处理机 1、2、3、4 的时间片 3 中。这种调度机制可以同 MICRO 系统中使用的进程管理的层次算法结合使用，只要每一系主任保存其工作者的矩阵，对矩阵中的进程分配时间片，并广播发送同步时间信号即可。Ousterhout 还描述了其他几个基于该算法的为提高系统性能的变形算法。其中一个算法将上面的矩阵切割成一行一行，并将所有的行连接成一个很长的整行。在 K 个处理机的情况下，任意连续的 K 个时间片属于不同处理机。为将一组新的进程组分配时间片，可以将一个宽度为 K 的窗口放在这一整行上，使最左边的时间片是空的，而窗口外紧挨左边沿的时间片是满的。如果窗口中有足够的空时间片，那么可以将这一组进程分配到空的时间片中；如果没有，向右移动窗口并重复上面的步骤。调度开始时窗口在整行的最左边，每个时间片向右移动大约一窗口的宽度，一组进程不能分开分配到多个窗口上。Ousterhout 的论文中对这个算法和其他的算法都作了详细的说明并给出了性能结果。

4.5 容错

如果系统不能达到它应达到的技术要求，就称为失效。有些情况，如超市分布式定货系统，失效可能导致店中罐装豆子没有存货。另如分布式控制航空交通系统，失效会引起巨大的灾难。随着计算机分布式系统在要求严格保证安全任务方面的广泛应用，防止失效的需求相应地越来越大。这一节，我们将分析一些关于系统失效的问题以及如何来避免它们。除此之外，可在 Cristian, 1991; 和 Nelson, 1990 中找到有关介绍。Gantenbein(1992) 已经编辑了关于这个问题的一个参考书目。

4.5.1 组成部件错误

由于一些部件错误会导致计算机系统失效，如处理机、存储器、I/O 设备、电缆或者软件等。这种错误是一种故障，可能是由于下列的原因引起的：一个设计差错，一个制造差错，一个编程错误，物理部件的损伤，时间的恶化，环境条件的恶劣（雪降到计算机上），意外的输入，操作员的差错，鼠类动物的啃咬，以及其他一些原因。并不是所有的错误都会（立刻）导致系统失效，但有些错误会造成系统失效。

错误通常划分为暂时性、间断性和永久性三种。暂时性错误发生一次后就消失，若重复那个操作，错误就会消失。一只鸟飞经微波传输器所发射的光束，就可能使网络中丢失许多位数据（大鸟更是如此）。如果传输超时并再次尝试传输，这次将可能会正确。

间断性错误出现后，自动消失，然后再出现，如此反复。连接器的连接部位松动常常会导致此类错误。间断性错误由于难于诊断而危害极大。特别是每当错误诊断器进行工作时，系统却又工作得很正常的情况。

永久性错误会在错误部件修理好之前一直存在。烧毁芯片、软件错误、磁盘头损害等常导致永久性错误

设计和构造容错系统的目标就是要确保：即使在错误发生的情况下，整个系统也能继续正常运行。这个目标与仅仅设计具有高可靠性的各个独立部件的目标大不相同，它允许（甚至期待着）当一个部件失效时系统仍不失效。

错误和失效在各个层次上都可能发生，如晶体管、芯片、电路板、处理机、操作系统、用户程序等。在容错领域中传统的工作大都是统计分析电子部件的错误。简单地讲，如果一个部件在规定的 t 秒时间内的失效概率为 p ，那么它连续 k 秒正常工作，然后再失效的概率为 $p(1-p)^{k-1}$ 。失效发生的期望值如下公式：

$$\text{失效发生平均时间} = \sum_{k=1}^{\infty} kp(1-p)^{k-1}$$

从 $k=1$ 开始，利用上面的无限求和公式： $\sum \alpha^k = \alpha/(1-\alpha)$ ，其中 $\alpha = 1-p$ 。两边对 p 微分并乘以 $-p$ 我们得到：

$$\text{失效发生平均时间} = 1/p$$

例如，如果失效概率是每秒 10^{-6} ，则失效发生平均时间是 10^6 秒或者大约 11.6 天。

4.5.2 系统失效

在重要的分布式系统中，我们常需要使得系统能够在部件（特别是处理机）出错时仍能正常工作，而不仅是避免发生这种出错。由于分布式系统中部件数量巨大，因此它们中某个部件出错的机会很大，所以系统可靠性显得尤为重要。

下面将讨论处理机错误或崩溃的情况，对此理解了，对进程的错误或崩溃（如由软件

错误引起的)也就能很好地理解。有两种处理机错误:

(1) Fail - silent 错误;

(2) Byzantine 错误。

对第一种错误 Fail - silent, 失效的处理机只是停止运行, 对接下来的输入不作反应也不产生进一步的输出, 即宣布它不再工作了, 称为 Fail - stop 错误。对 Byzantine 错误, 出错的处理机继续运行, 产生问题的错误答案, 并可能和其他出错的处理机一起“恶意”地工作, 给人一种它们在正常工作的假象。未检测出的软件错误常表现为 Byzantine 错误。显然, 处理 Byzantine 错误比处理 Fail - silent 错误更加困难。

4.5.3 同步系统与异步系统

如上所述, 部件失效可能是暂时性的、间断性的和永久性的。系统失效可以是 Fail - silent 或 Byzantine。第三种分类标准是在抽象意义上考虑性能。假设系统中, 一个处理机发送消息给另一个处理机, 在事先知道的时间 T 内保证会得到一个回答。若没有得到应答, 则表示接收系统已经崩溃。时间 T 包含了处理消息丢失所需(至多重新发送 n 次)的时间。

在容错系统研究领域内, 若系统工作时具有这样一个特性, 即它总能在一个已知的限定时间内对一条消息进行响应, 就称之为同步。若无此特性则叫作异步。尽管这个术语与传统的该词表达的意思有冲突, 但它在容错领域的研究者中间被广泛使用。

很显然, 异步系统要比同步系统更难对付。如果一个处理机发送消息, 并知道若在限定时间 T 内没有回答表示接收者失效, 它就能够采取正确的措施。若不知道响应发生时间的上限, 那么要判断接受者是否失效也会成为一个问题。

4.5.4 使用冗余

解决容错通常的办法是使用冗余技术。有三类可能: 信息冗余、时间冗余和物理冗余。信息冗余就是增加额外数据位以使出错的数据完全恢复。例如, 海明码可用于附加在传输数据上, 以使数据能从传输线的噪音中恢复。

时间冗余就是动作执行后, 必要时可再次执行。使用第3章中描述的原子事务处理就是本方法运用的一个例子。若一个事务处理失败, 它可无负作用地再次执行。时间冗余对于暂时性和间断性错误特别有用。

物理冗余, 要使用额外部件以使整个系统能容许一些部件的损失或失效。例如, 使用冗余的处理机, 当一些崩溃时, 系统仍能正常运行。

有两种组织冗余处理机的方法: 主动复制(active replication)和主机后备(primary backup)。考虑一个服务器的情况, 当使用主动复制时, 所有的处理机在所有的时间内都是作为服务器(并行)使用, 以彻底容错。相反, 主机后备方法仅使用一个处理机作为服务器, 当它失效后是用另一个后备机器来替换的。

下面我们将讨论这两种方法。这两种方法都有如下的问题:

- (1) 需要复制的程度;
- (2) 没有错误时, 平均情况和最坏情况下的性能;
- (3) 发生错误时, 平均情况和最坏情况下的性能。

许多容错系统的理论分析就是针对这些问题的。要了解更多的信息，请看 Schneider,1990; 和 Budhiraja 等, 1993。

4.5.5 使用主动复制方法的容错

主动复制是使用物理冗余来提供容错的一种著名的技术。它应用于生物学（哺乳类动物有两只眼，两只耳朵，两个肺叶等等），飞行器（波音 747 有四个引擎但使用三个就能飞）和体育（多名裁判以防有一个出错）。有些作者将主动复制称为状态机方法。

这种方法多年来也应用于电子电路的容错。分析图 4-19 (a) 中的电路：信号按顺序通过设备 A,B,C，若其中有一个出错，结果就可能错误。

在图 4-19 (b) 中,每个设备复制三次，结果是每级电路都设置了三个表决器，每个表决器都有三个输入和一个输出。若两个或三个输入相同，输出则等于该输入。若三个输入各不相同，输出就是不定值的。这种设计就是 TMR（三模冗余）。

假设 A2 失效，V1, V2, V3 都得到了两个好的（相同的）输入和一个坏的输入，这样每个都输出正确值到第二级。事实上，A2 出错的影响彻底地被屏蔽了，因此 B1, B2, B3 的输入值正好和没有错误发生时的输入值一样。

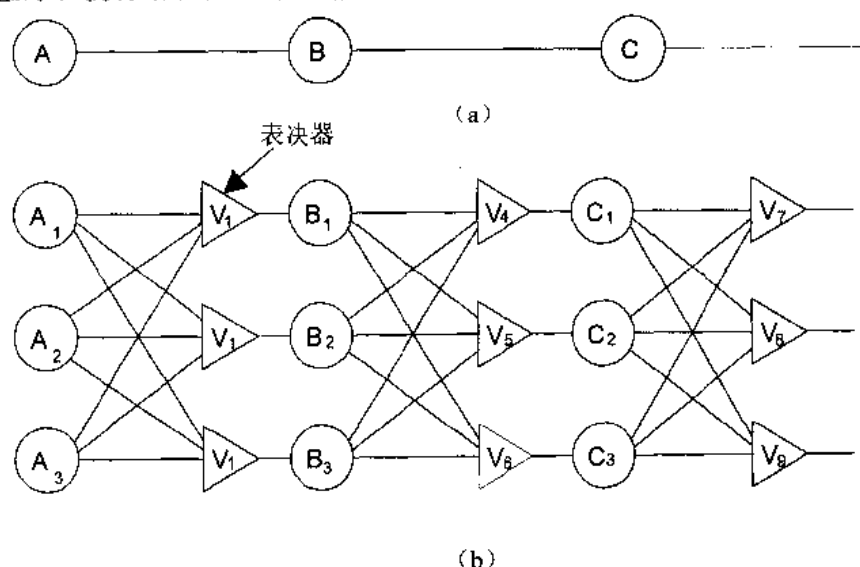


图 4-19 三模冗余

现在分析在 A2 出错的基础上，如果 B3、C1 也出错，这些影响也会被屏蔽，最终的三个输出仍是正确的。

首先，每级都用三个表决器的原因并不一目了然。毕竟一个就可检测并传递大多数的意见。然而，表决器也是一个部件，也可能出错。例如假设 V1 失灵，B1 的输入也就是错误的。但只要其他的都正常，B2、B3 将产生相同的输出，并且 V4、V5、V6 将都产生正确的输出到第三级。V1 和 B1 出错的影响在结果上没有什么不同。两种出错都使 B1 输出错误，但都将在后面的级中消除其影响。

尽管并非所有容错分布式操作系统都使用 TMR，但这项技术还是很通用的，它能给人一种清晰的概念：什么是容错系统；什么是有高可靠性的独立部件，但它们组成的系统

都不是容错的系统。当然，TMR 也可递归使用。例如，将 TMR 技术用于芯片内部而对使用芯片的设计师是隐含的，这使其具有高可靠性。

我们再回过头来看容错，特别是主动复制，在许多系统中，服务器很像一个大的有限状态机：接收请求，给出应答。读请求不改变服务器状态，但写请求改变。若将每一个用户的请求都发给每一个服务器，服务器接收这些请求并以相同的顺序运行，那么在处理完每一个请求后，所有无错误的服务器将处于同样的状态，并将给出同样的结果。客户端或表决器能综合这些结果以屏蔽掉错误。

一个很重要的问题就是需要多少份复制才合适。这个问题取决于想要达到的容错量。如果系统可在 k 个部件出错时仍能达到系统的设计要求而正常工作，那么这个系统可称为是 k 容错的。如果这些部件，比如处理机，其出错属于 fail-silent 型，那么有 $k+1$ 个这样的部件足以满足 k 容错的要求。若 k 个处理机简单停止工作，那么，可使用剩下的那个处理机的结果。

另一方面，若处理机错误是 Byzantine 型的，出错的处理机仍然运行并发出错误或随机的应答，那么至少需 $2k+1$ 个处理机才能达到 k 容错。最坏情况下， k 个失效处理机偶然（或甚至有意）地产生同样的应答。然而，剩下的 $k+1$ 个未出错的也将产生相同的应答，因此客户或表决器只要相信大多数的应答就可得到正确的结果。

当然，理论上可以认为一个系统是 k 容错的，通过由 $k+1$ 个正确相同答案得来的票数占优而消除 k 个错误相同答案的影响。但实际应用环境中很难想象你能肯定 k 个处理机会出错，而另外 $k+1$ 个就不会出错。因此，一个容错系统也需要进行一些概率分析。

与这种有限状态机模型相关的一个默认的前提是所有的请求到达所有服务器的顺序应相同，有时称之为原子广播问题（atomic broadcast problem）。实际上，由于读不会有问题而写可以交换，因此这个条件可以放宽，但基本问题仍然存在。为保证所有服务器对所有请求都能以同样的顺序执行，一个方法就是对所有请求作全局计数编号。人们已经设计了许多协议来实现这个目标。例如：可先将所有的请求发送到一个全局计数服务器，由它给请求分配一个顺序号，但同时必需制定一些规定来处理这台服务器出错时的情况（例如，可通过内部容错的方法）。

另一个可能的方法是使用第 3 章中所描述的 Lamport 逻辑时钟。如果发往服务器的每条消息都标上时间标签，并且服务器也按时间标签顺序执行，所有的请求将在所有的服务器上以同样的顺序执行。这个方法的问题在于当服务器接收到一条请求后，它并不知道是否有时间上更早的请求还没到达。事实上，大多数使用时间标签的方案都存在这一问题。简而言之，主动复制不是一件简单的事情。Schneider(1990)详细地讨论了这其中的问题和一些解决办法。

4.5.6 使用主机后备的容错

主机后备方法的基本思想是：在任一时刻都有一台服务器是主机，它完成所有的工作，若这个主服务器失效了，后备的服务器将承担其任务。理想情况下，这个切换过程应当以一种“清洁”（clean）的方式进行，并只有客户端操作系统可知，而不为用户应用程序所知。正如主动复制一样，这种方法在现实世界中也广泛使用。如用于政府（设立副总统），航空（辅助领航员），汽车业（备用轮胎），以及医院手术室的柴油动力发电机。

与主动复制相比较，主机后备容错有两个主要优点。首先，在正常工作中，消息只发

往一个服务器（主服务器）而不是到一组服务器，因此要简单得多。也不存在消息排序问题。第二，实际应用中所需机器也少得多，因为在任何时候只需一台主机和一台后备的机器（虽然当一台后备的机器投入运行成为主机时，立即需要另一台后备的机器）即可。它的缺点是，在出现 Byzantine 错误时，主机出错而它却宣称自己工作得很好从而产生假象；而且恢复一个失效的主机也是很复杂和耗时的。

作为一台主机后备方法的例子，请看图 4-20 中一个写操作协议的简单描述。客户发送消息到主机，它进行处理后再向后备机发出更新消息。后备机收到消息进行处理然后向主机发一条确认消息。当确认消息到达主机后，主机才向客户发出应答。

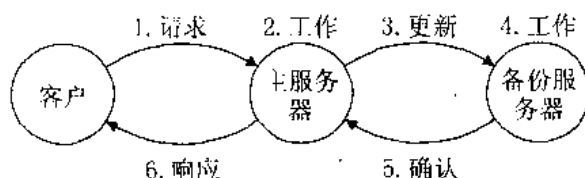


图 4-20 写操作简单主备份协议

现在考虑主机在一个 RPC 调用（远程过程调用）的不同时刻崩溃所产生的影响。若主机在进行处理（第 2 步）前崩溃，将是无害的。客户在时间片用完后会再试，直至重试足够多次后，它最终将会到达后备机，且只进行一次处理。若主机在处理完后，在发送更新消息之前崩溃，后备机就成为主机，如果客户又一次发送请求，其处理工作将再重复做第二次。若这个处理有副作用，就会产生问题。若主机在第 4 步之后并且在第 6 步之前崩溃，将做三次处理工作后结束。即主机一次：作为第 3 步的结果，后备机要做一次；在后备机成为主机后，又做一次。若请求中携带标志信息，有可能保证处理工作只进行两次，但要一次正确完成是很困难的，以致几乎是不可能的。

主机后备方法的一个理论也是一个实际的问题是：什么时候进行“主机-后备机”的切换。在上面的协议中，可通过后备机周期向主机发送“是否正常”的消息。如果在特定时间内，主机没有应答，则后备机取代主机。

但是，当主机并未崩溃而只是慢了一点时（如在异步系统中），又该如何呢？没有办法区分一个慢的主机和一个崩溃的主机。但有一点必须保证：当后备机取代主机后，原来的主机要停止充当主机的角色。理想的情况下，主机和后备机之间应有一个协议来讨论这个问题，但和一个已“死”的机器是无法协商的。最好的解决方法是设计一种硬件机制，使后备机可强制停止或重新启动主机。注意所有的主机后备方法都需要一致的协议，这是很难实现的。然而在主动复制方法中，并不总是需要一致的协议（如 TMR）。

图 4-20 中方法的一个变形是使用主机和后备机之间共享的两端口磁盘。在这种配置下，当主机得到一个请求后，在做任何事之前先将请求写入磁盘，并且也把执行的结果写入磁盘，不再需要向后备机发送和接收从后备机发来的消息了。若主机崩溃，后备机可通过读磁盘而得到整个系统现在的状态。这种方法的缺点是只有一个磁盘，因此如果它也失效了，所有消息将丢失。当然，以附加额外的设备和牺牲性能为代价，磁盘也可复制，所有的写操作将是向两个磁盘写入。

4.5.7 容错系统中的协同一致

在许多分布式系统中，进程之间存在着遵守某些协议的要求。例如，选择一个协调器，决定是否进行一个处理，在工作者（各工作站）之间分配任务，同步等等。当通信和处理机都很正常时，达到协同一致是很简单的。否则，会出现很多的问题。本节将讨论这些问题及其解决方法（或怎样避免该问题）。

通常分布式协同一致算法的目标是使所有无故障处理机对待某些问题的意见达到一致，并在有限的步骤内进行处理。依据系统的参数可有不同的情况，包括：

- (1) 消息传递总是可靠吗？
- (2) 进程会崩溃吗？若会，是 fail-silent 错误还是 Byzantine 错误？
- (3) 系统是同步还是异步？

在考虑出错处理机之前，先看一种处理机正常但通信线路可能丢失消息的简单情况。这里有一个著名的问题：两军问题。它描述了两个很正常的处理机要达成关于一位信息的协同一致，有多么的困难。红色军队有 5000 人，驻扎于山谷。两队蓝色军队各 3000 人，驻扎于周围山上，可俯瞰山谷。两队蓝色军队若能协同攻击红色军队，将会获胜。若仅一支出击，将被屠杀。蓝色军队的目的就是实现协同出击，

问题是它们只能通过一条不可靠的通道来通信：通过派遣信差，但信差经常被红色军队俘虏。

设想，蓝一军的指挥官 Alexander 将军，发信给蓝二军的指挥官 Bonaparte 将军，说：“我有一个计划——明天黎明时一起攻击”。消息传到蓝二军，Bonaparte 遣回信差并回复说：“好主意，Alex，咱们明天黎明见”。信差安全回到他的基地，把回复的消息告诉 Alexander，Alexander 就告诉他的部队准备黎明时攻击。

然而，在这之后，Alexander 意识到：Bonaparte 不知道信差是否安全返回，为此，他可能不敢出击。结果是，Alexander 再派一个信差去告诉 Bonaparte，他（Bonaparte）的应答消息已到，已经准备好战斗了。

同样，信差安全到达并传达了确认消息，但现在 Bonaparte 担心 Alexander 不知道其确认是否已到，Bonaparte 推测：如果 Alexander 认为自己的信差被俘，将对他的这次计划不确信，也许不会冒险出击。因此，Bonaparte 又让信差返回。

即使信差每次都通过敌营到达了蓝军，但很显然，不管他们派信差多少次，Alexander 和 Bonaparte 都永远不会达成协同一致。设想存在某个协议在有限步内终止使其达成一致，移去多余步以使工作协议最小化。某个消息成为最后一个消息并且它对于达成一致是必需的（由于这是最小协议）。若这个消息传递失败，战争将失败。

然而，最后这个消息的发送者不能得知其是否到达。若不能，协议将不能完成协同一致，另一将军将不会攻击。因此最后这个消息的发送者不知是否按原计划进行战斗，也就不能安全指挥军队。同样，因为最后一个消息的接收者知道发送者不确信，他也不会冒险出击，因此没有达到双方的协同一致。即使对无故障的处理机（本例中的将军），在不可靠通信的情况下，即使两个进程间达成协同一致也是不可能的。

现在让我们假设通信很正常但处理机出错的情况。这里的经典问题也发生在军队中，称为 Byzantine 将军问题（Byzantine generals problem）。在这个问题中，红军仍驻扎在山

谷,但附近山头上有 n 个蓝军将军,他们各领一支军队。交通是通过俩俩电话线安全正常的进行。但蓝军将军中有 m 个是叛徒(出错),他们通过提供给忠实将军们错误的和矛盾的消息(模仿失灵的处理机)来阻止忠诚将军们达成协同一致。现在的问题是忠诚将军们是否仍能达到协同一致。

为使问题更具普遍性,这里定义的协同一致稍有不同。每个将军假定都知道各自的军队人数,问题的目标是将军们能互相交换军队数量信息,因此在算法的最后,每个将军都有一个 n 维关于所有军队数量的向量。若 i 号将军是忠诚的,那么第 i 维元素是他的军队数。否则,它是不确定的。

Lamport 等人(1982)设计了一种递归算法可在特定条件解决这一问题,如图 4-21 所示。算法工作条件是: $n=4, m=1$ 。对这样的参数,算法运行四步。第一步,每个将军发送可靠的消息给其他所有的将军,声明他自己真实的军队人数。忠诚将军声明的是真值,叛徒则可能对其他每个将军都撒一个不同的谎。在图 4-21(a),我们可看到将军 1 报告他有 1K 军队,将军 2 报告他有 2K 军队,将军 3 对其他三个将军分别谎称有: x, y, z 军队,将军 4 报告他有 4K 军队。第二步,把第一步声明的结果收集组成向量形式,如图 4-21(b)。

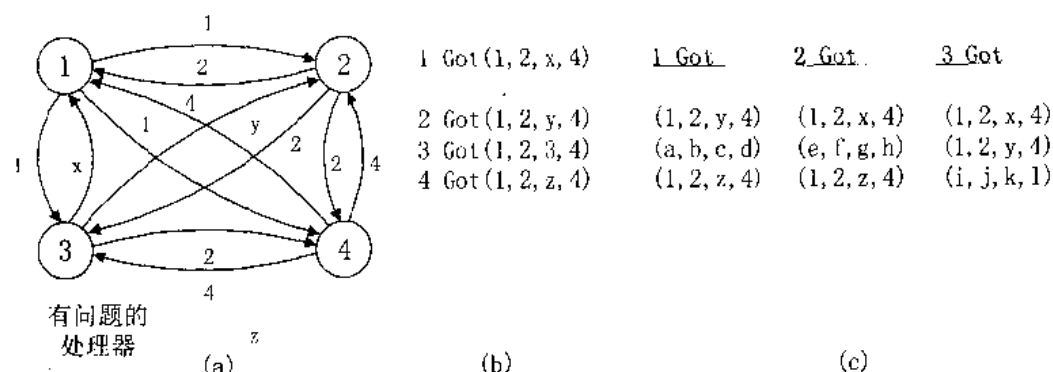


图 4-21 3 个忠诚将军和一个叛变将军的拜占廷 (Byzantine) 将军问题

- (a) 各个将军通报各自部队的战斗力 (1K 单位)
 (b) 每个将军根据 (a) 汇编出的向量
 (c) 每个将军从第二步得到的向量

第三步,每个将军把图 4-21(b)中各自的向量传递给其他每一个将军。在这里,将军 3 又谎报,使用了 12 个新值,从 a 到 l 。第三步的结果如图 4-21(c)。最后,第四步,每个将军检查所有新接收向量的第 i 个元素。若有某个值占多数,则将该值放入结果向量中。若没有一个值占多数,结果向量中的相应元素标记为不知道(UNKNOWN)。从图 4-21(c),我们看到 1, 2, 4 号将军产生协同一致向量:

(1,2,UNKNOWN,4)

这是正确结果。背叛者不能得尝所愿。

现在,让我们再来看 $m=3, n=1$ 的情况。即有两个忠诚将军,一个叛变将军,如图 4-22 所示。这里我们看图 4-22(c),没有一个忠诚将军认为元素 1, 2, 3 中占多数,因此它们都被标以不知道 (UNKNOWN)。算法失败不能产生协同一致。

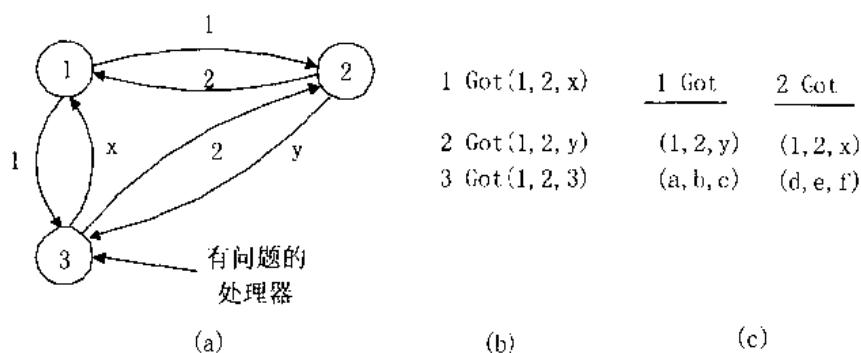


图 4-22 两个忠诚将军和一个叛变将军的拜占庭问题

在 Lamport 等人(1982)论文中, 已证明了一个有 m 个处理机出错的系统中要实现协同一致, 只有当有 $2m+1$ 个正常处理机时才可能。处理机总数为 $3m+1$ 。换种说法即是, 只有大于 $2/3$ 的处理机正常工作时, 协同一致才是可能的。

然而糟糕的是, Fischer 等人(1985)证明: 对一个具有异步式处理机和对传输时延没有限制的分布式系统, 哪怕仅一个处理机失效 (即使是 fail-silent 型), 协同都是不可能的。异步式系统的问题是区别特别慢的处理机和失效的处理机。已经有了一些关于什么情况下协同一致是可能的, 以及什么时候是不可能的其他的理论成果。这些理论成果已在 Barborak 等(1993) 和 Turek 和 Shasha(1992)中给出。

4.6 实时分布式系统

容错系统不是唯一的专用分布式系统, 实时系统是另一种。有时, 两者结合成为容错实时系统。这一节, 我们将分析实时分布式系统的各个不同方面。若要了解更多的情况, 请参看 Burns 和 Wellings,1990; Klein 等,1994 和 Shin,1991)。

4.6.1 什么是实时系统?

对大多数程序来说, 正确性仅仅依赖于指令执行的逻辑顺序, 而不是指令何时执行。如果 C 程序能够在 200-MHz 的工程工作站上正确计算出双精度浮点数的平方根, 那么它也能在基于 4.77-MHz 的 8088 个人计算机上正确执行, 只是慢了一点而已。

与这大不相同的是, 实时程序 (及系统) 同外部世界交互, 其中涉及到时间。当某个激励出现时, 系统必须以一定的方式和在限定的时间内响应它。若返回结果正确, 但已超时, 系统也认为是失效的。何时产生结果与产生什么结果同等重要。

举一个简单的例子, 一个声音压缩磁盘 (声音 CD) 播放器包含一个 CPU, 它读出磁盘上的数据, 并进行处理产生音乐。假设这个 CPU 恰好有足够的速度来做这件事。现在有一个竞争者, 想制造一个更便宜的播放器, 它使用相当于原 CPU $1/3$ 速度的 CPU。如果它先缓存所有到来的数据, 并以 $1/3$ 的速度播放, 人们将难以忍受这种声音。如果它仅每三个音符播放一个, 观众也不会心旷神怡。不像前述的平方根的例子, 这里对时间的要求本质上是正确性要求的一个特定部分。

许多涉及外部世界的其他应用，本质上也是实时的。例如，嵌入了电视、录影机的计算机，控制飞机副翼和其他部件的计算机（称作 fly-by-wire），计算机控制的汽车子系统（drive-by-wire），控制引导反坦克导弹的军用计算机（shoot-by-wire），计算机航空交通控制系统，还有从粒子加速器到脑中装有电极的心理实验鼠的科学实验，自动化的工厂，电话交换机，机器人，医疗上的重危护理病房，CAT 扫描仪，自动股票交易系统和其他众多系统。

许多实时应用系统比普通用途的分布式系统更加结构化。典型地，一个外部设备（可能是一个时钟）向计算机产生一个激励，计算机必须在时限内进行某些特定的处理。当完成所要求的工作时，系统成为空闲，直至下一个激励的到来。

通常激励是周期性的，每 ΔT 秒发生一次。如使用在 TV 或 VCR 中的计算机，每 1/60 秒得到一个帧。有时激励是非周期性的，即意味着它们是重现的，但不是周期的。比如航空交通控制器所控制的飞行空间中到来的飞机。此外，一些激励是突发的（无法预料的），如设备过热。

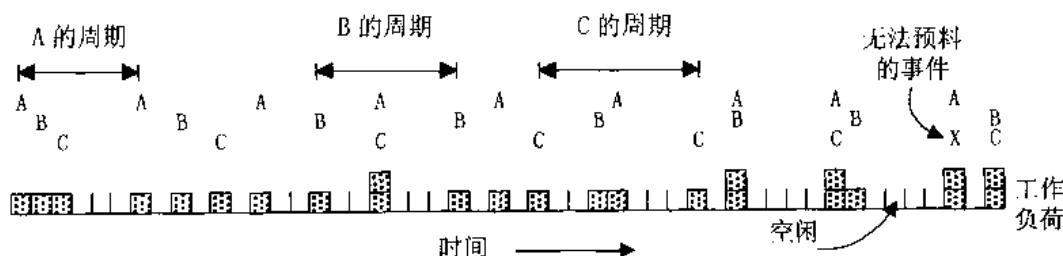


图 4-23 三个事件流加上一个突发事件的叠加情况

在大的周期性系统中，其复杂性表现于存在着多种事件，如视频输入、音频输入、马达驱动管理等，它们每个都有其固有的周期和响应的行为。如图 4-23 描绘的一种情形，图中有三个周期性事件流 A、B、C，另外还有一个突发性事件 X。

虽然 CPU 必须处理多个事件流，但对 CPU 而言：它错过了事件 B，因为当 B 发生时，它仍在处理事件 A，这种情况是不可接受的。虽然使用优先级中断的方法控制两三个输入流并不困难，但随着应用规模变得越来越大和越来越复杂（例如，有数以千计个机器人的工厂自动组装线）。对一台机器来说，要使其满足所有的时限及其他一些实时限制将变的越来越困难。

结果是，有些设计师正在试验一种新方法，在每个实时设备前放置一个专用的微处理器，只要实时设备一有输出就接收，并且以任何要求的速度给实时设备输入。当然，这并没有丧失实时特性，而是引发了一个分布式的实时系统，并且有其自身独有的特点和优势（例如实时通信）。

分布式实时系统通常可以按图 4-24 所描述的形式构造。我们可以看到，这是一个由网络连接起来的一些计算机，其中一部分计算机与外部设备相连，这些外部设备产生或接收数据或期望对它进行实时控制。这些计算机可能是一些嵌入到设备里的微型控制器，或者是独立的机器。但它们都有接收来自设备的信号的传感器和（或）向它们发送信号的执行机构，传感器和执行机构可以是数字式或模拟式的。

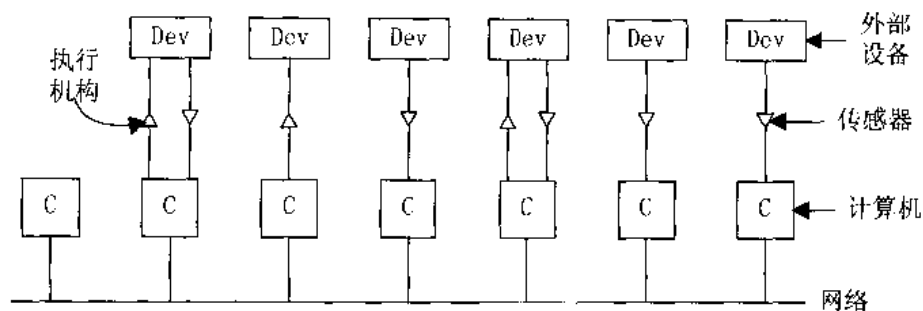


图 4-24 分布式实时计算机系统

根据时限的严格程度和超过限制时间的后果的严重性，通常将实时系统分为两类：

- (1) 软实时系统；
- (2) 硬实时系统。

软实时系统意味着可以偶而错过时限。例如，电话交换机允许在超载情况下，每 100000 次中可有 1 次丢失或串线，这仍是在系统允许的规定范围之内。相反，在硬实时系统中，哪怕一个实时请求超过时限，都是不可接受的，其后果可能人命关天或导致环境的大灾难。实际中，也有介于中间情况的系统，当超过一个时限，必须结束当前的处理，其结果却不是致命的。例如，在传送带上的苏打瓶经过了喷嘴，继续向它喷入苏打是无意义的，但结果并不是致命的。并且，在一些实时系统中，有些子系统是硬实时系统，而其他子系统却是软实时系统。

实时系统已经使用了几十年了，因而人们对它积累了大量的认识和经验，但其中许多是错误的。Stankovic (1988) 指出其中的一些，最荒谬的有如下几个：

误解之一：实时系统就是用汇编代码来编写的驱动程序。

这个在二十世纪 70 年代对于一个只连接了少数几个设备的小型机的实时系统也许是真的，但现在的实时系统太复杂了，利用汇编语言编写驱动程序已不胜任了，但对实时系统设计者来说设备驱动程序已成为最不需要担心的事了。

误解之二：实时计算是快速计算。

这不是必要的。计算机控制的天文望远镜可能不得不实时跟踪星星和星系，但地球每小时仅旋转 15 弧度，不是特别快，在这儿，我们更关心精确性。

误解之三：高速的计算机将使实时系统过时无用。

这是错误的。高速计算机只会激励人们构造在以前的发展水平之上的实时系统。心脏学家希望有一台 MRI 扫描仪给他们实时地显示正在进行运动恢复的病人的心跳图。当这个目标实现时，他们又会要求三维的全彩色的图形，并要求可以进行放缩。此外，因使用多处理机而使系统更快地引入新的必须解决的通信、同步和调度等问题。

4.6.2 设计问题

实时分布式系统有一些特有的设计问题。本节将对其中几个最重要的问题加以介绍。

时钟同步

第一个问题就是时间的自身维持问题。在多计算机情况下，每台都有自己的本地时

钟,保持时间同步是一个关键问题。我们已在第3章中讨论了这些问题,这里不再重复了。

事件触发系统与时间触发系统

在事件触发的实时系统中,当一个重要的外部事件触发时,它被传感器察觉到,并导致与传感器相连的CPU得到一个中断请求。事件触发系统因此就是中断驱动的,多数的实时系统也是以这样的方式工作的。对拥有许多剩余计算能力的软实时系统来说,这种方法是简单的,并因工作良好而被广泛应用。即使对更复杂的系统,只要系统的编译器能对程序进行分析,并在一旦发生某一事件时(即使是不知道事件何时发生),能知道系统应采取怎样的响应,在这种情况下此方法也很好。

事件触发系统的主要问题在于:当许多事件一次性发生时,在重载情况下会失效。如考虑计算机控制的核反应堆中的一个管道破裂时的情况,它会同时发生温度报警、压力报警、放射性报警及其他报警,引起大量的中断。这种事件风暴(event shower)可能压垮计算机系统,使其崩溃,然后会潜在地引起比一个管道破裂严重得多的问题。

不存在这种问题的另一种设计是时间触发实时系统。在这类系统中,每 ΔT 毫秒产生一次时钟中断。在每一次时钟滴答时,对(选定的)传感器进行采样,并且驱动(特定的)执行机构。中断仅在时钟滴答时发生。

上述管道破裂的例子中,系统在事件发生后的第一次时钟滴答时就发现了该问题,但是中断负载不会由于这个问题而改变,因而系统不会过载。在危机出现期间能正常运行增加了成功处理危机的机会。

但需强调 ΔT 必须非常小心地选择。若 ΔT 太小,系统将产生许多时钟中断,将浪费大量的时间来响应它们。若 ΔT 太大,严重的事件可能未被注意到或在被发觉时可能为时已晚。而且,对这些情况的决定都是很关键的,在每一次时钟滴答时,要检查哪些传感器,在每隔一次时钟滴答时,又应检查哪些传感器,如此下去。最后,有些事件的持续时间可能比一个时钟滴答的时段还短,因此应存储这些事件,以免丢失。它们可保存在锁存电路中或保存在嵌入外部设备的微处理机中。

举一例说明上述两种方法的不同点。考虑一个100层楼的电梯控制器的设计。假定电梯正在60层安静地等待顾客,有人在一层按下按钮。就在100毫秒之后,另一人在100层上按下按钮。在事件触发系统中,第一次按钮产生一个中断,将使得电梯启动下行,就在它作出下行的决定后,第二个按下按钮的事件到来,因此第二个事件被记录下来以作为将来的参考,但电梯还是继续下行。

现在考虑时间触发的电梯控制器,它每500毫秒采样一次。若两次按下按钮事件都在一次采样周期中出现,控制器就不得不进行决定,例如,使用最近用户优先原则,那么此时电梯将上行。

总之,事件触发的设计使得在低负载时会更快地响应,但在高负载时,会更加过载,并有可能崩溃失效。时间触发的设计有相反的特征,并且仅适用于相对静态环境,在这种环境中,事先就已经知道了系统大量的行为。哪一个方法更好取决于应用,在实时领域中,对于每一件具体的事,都会有关于该问题的十分激烈的争论。

预知性

任何实时系统的一个最重要的特性就是其行为是可预见的。理想情况下,设计时应该清楚系统能够满足的所有时限,甚至是最大负载时限。关于系统行为的统计分析在独立

事件的假定下常会令人误解，因为事件之间可能会有难以预见的联系，就像上面管道破裂例子中的温度、压力与放射性报警之间的联系。

大多数分布式系统设计者常认为随机访问共享文件的用户是独立的或是许多旅行社在不可预见的时间访问共享航线数据库。幸运的是，这种随机的行为在实时系统中很少成立。通常是，例如，检测到事件 E 时，运行进程 X ，接下来以任何顺序或并行运行进程 Y 、 Z 。此外，通常知道（或应当知道）这些进程最坏情况下的行为是什么。例如，若已知 X 需要 50 毫秒， Y 、 Z 各需 60 毫秒，进程启动需 5 毫秒，那么事先可确定一秒钟内，系统在无其他工作的情况下，可正确地处理五个周期 E 类事件。这种推理和建模导致一种确定性的系统而不是随机的系统。

容错性

许多实时系统控制着交通、医院、能源工厂等中的安全关键的设备，因此容错是非常普遍的问题。有时使用主动复制，但仅限于不存在有扩展（因而是耗时的）的协议让所有的部分在任何时间，在任何事件上都必须进行协同一致。主机后备机的方法很少使用，这是因为在主机失效后，主机后备机切换期间可能会超过时限。一种混合方法是听从“领导者”，即为：一台机器作所有的决定，其余的机器只是照吩咐去做，一有通知就进行工作。

在一个安全为最关键的系统中，系统能够对最坏情况进行处理是特别重要的。不能因为三个部件同时出错的可能性很小而将其忽略，失效不总是独立的。例如，在一次突然的停电事件中，每个人都拿起电话，就可能造成电话系统过载，尽管电话系统有自己独立的发电系统。此外，系统的最大负载经常恰恰发生在当系统中最大量设备失效的时候，因为系统通信量中许多都是与报告失效有关的。结果是，容错实时系统必须能同时处理最大量设备失效和最大量负载的情况。

有些实时系统有这样的特性：当严重出错时，就强制停止运行。例如：当一个铁路信号系统突然熄灭时，控制系统就可能通知各列火车立即停下。如果系统设计总是能使火车间的距离足够远，并且所有火车都能几乎同时刹车，灾难就可以避免，并且系统也可以在来电之后慢慢恢复。一个像这样能够停止操作而不会产生危险的系统，称为可安全停机的（fail-safe）。

语言支持

虽然许多实时系统和应用程序是用像 C 一样的通用语言编写的，但专用的实时程序设计语言可能更有用。例如，在专用的语言中，易于将一个工作作为一些短任务的集合（如，轻进程或线程），这些短任务可以独立调度，但要服从用户定义的优先权和互斥的限制。

设计这种语言应该使它在编译时能计算出每个任务的执行时间。这种要求意味着该语言不能支持一般的 while 循环语句。对循环重复执行必须要用常量参数的 for 循环，也不允许递归（如同在 FORTRAN 中使用一样）。即使是有了这些限制，也不足以事先计算出每个任务的执行时间，这是因为高速缓存没有命中、页面错、DMA 通道周期窃取等突发事件都影响性能。

实时程序设计语言也需要有一种方法处理自己的时间。首先，应有一个特殊的变量 *clock*，用时间片的多少来保存当前的时间。但要小心表示时间所用的单位，单位越小（细），*clock* 变量溢出越快。例如：若它是一个 32 位整型，变化的时间片数范围如表 4-3 所示。理想情况时 *clock* 应选择 64 位，1 纳秒的时间片。

表 4-3 32 位时钟在不同分辨率下的有效范围(溢出前)

时钟分辨率	范围
1 纳秒	4 秒
1 微妙	72 分钟
1 毫秒	50 天
1 秒	136 年

这种语言应有表达最小和最大延迟的方法。例如在 Ada 中就有延迟语句, 规定进程必需延迟的最小时间。但实际延迟因为没有界限可能会延迟很多。无法给出延迟时间的一个上限或一个要求进程必须执行的时间间隔。

还应有一种能表示当期待的事件在限定时间内未发生时该做什么的方法。例如, 一个进程占用信号量超过了某段时间, 应该使其暂停并释放信号量。相似地, 若发送出一条消息, 却没有足够快的应答, 发送者应能指出在 k 毫秒后, 它将被阻塞。

最后, 由于周期性事件在实时系统中非常重要, 下面形式的语句将会十分有用:

every (25 msec) {...}

这个语句使得花括号内的语句每 25 毫秒执行一次。然而, 更好的是, 若任务中有多个这样的语句, 编译器就能计算出每个这样的语句所需 CPU 时间的百分比, 并根据这些数据计算出运行整个程序所需的最小机器数及这些进程如何分配给各个机器。

4.6.3 实时通信

实时分布式系统中的通信与其他分布式系统中的通信是不同的。虽然高性能总是受欢迎的, 但预知性与确定性是实时系统成功的真正关键。这部分我们将介绍一些实时通信问题, 包括 LANs(局域网)和 WANs(广域网)。最后, 详细分析一个例子, 以说明它和传统的(即非实时)分布式系统的不同。其他的一些方法在 Malcolm 和 Zhao,1994; Ramanathan 和 Shin,1992 中已作介绍。

在实时系统中实现可预知性, 意味着处理机之间的通信也是可预知的。本质上是随机的 LAN 协议, 如以太网, 由于未提供传输时间的上限, 因此是不能接受的。一个想要在以太网上发送信包的机器, 可能会和一个或多个机器发生冲突。所有机器将等待一个随机的时间然后再试, 但是仍会冲突, 如此继续下去。结果是, 不可能事先给出传递信包的最大时限。

与以太网不同, 考虑一下令牌环 LAN。每当一个处理机有信包要发送, 它就等待循环的令牌经过, 捕获令牌, 而后发送信包, 然后将令牌送还到环网上, 以便下游的紧挨的机器有机会得到它。假定环网中的 k 个机器每个在捕获令牌时最多可发送含 n 个字节的信息包, 这能保证一个紧急信包可在 kn 字节传送期内到达系统中的任何地方。这种上限正是实时分布式系统所需要的。

令牌环网也能处理多优先级系统的通信问题, 这里的目的是要保证高优先级信包等待发送时, 它将在它临近的任意低优先级的信包之前发送。例如, 可在每个信包上加一个保留域, 信包经过的每一个处理机都可增加其值, 信包环行一周后, 其保留域指示下一个要发送包的优先级。当当前发送者完成发送, 它产生一个具有该优先级的令牌, 只有具有

这个优先级并且有待发送信包的处理器才能捕获它，并只发送一个信包。当然，这个方法使得最高上限 kn 字节时间仅适应于具有最高优先级的信包的情况。

替代令牌环网的一个方法是 TDMA（时分多路）协议，如图 4-25 所示。通信以固定大小的帧组织，每个帧含 n 个时隙，每个时隙被分配给一个处理器。当一个处理机的时隙到来时，它可进行发送信包。这种方法避免了冲突，延迟也是有限的。每个处理机，依赖于每帧中它所分得的时间片个数，都有一个固定的带宽。

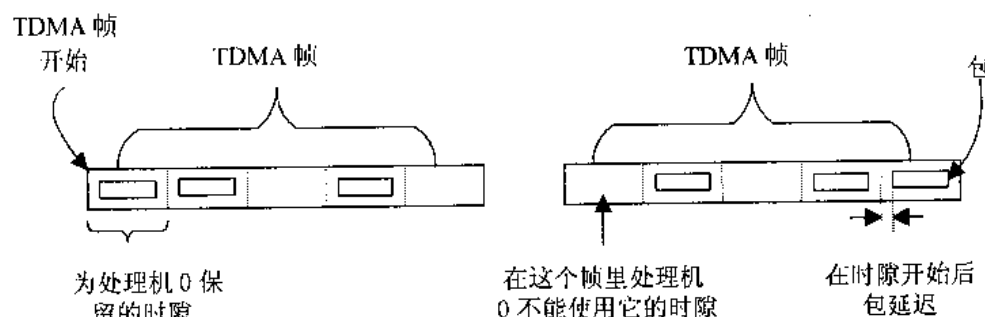


图 4-25 TDMA（时分多路访问）帧

广域网上的实时分布式系统像受限子房间或建筑物中的局域网一样，同样有可预知性的需求。这些系统中的通信都是面向连接的。通常，远距离机器之间能建立一个实时连接。当这样的连接建立时，网络服务质量是由网络用户与网络提供者事先协商的，这种服务质量可能包括确定的最大延迟、最大的抖动（信包传送的变动时间）、最小带宽和其他参数。要保证这些服务质量，在此连接的所有期间内，网络应保留有存储缓冲，表入口，CPU 周期，链接能力及其他一些资源，而不管是否使用它们，因为它们对别的连接是不可用的。

广域网实时分布系统的一个潜在的问题是它们信包丢失率相对较高。标准协议处理丢失信包的方法是在传送每一个传输信包时设置一个定时器。如果定时器在信包确认消息收到之前走完，就重发信包。实时系统中，这种无限的传输延迟是无法接受的。

发送者的一个简单的解决方法是：总是传输信包两次（或多次）。如果可选择的话，最好是通过相互独立的连接。尽管这个方法浪费了至少一半的带宽，但若十万个信包中有一个信包丢失，那么用这种方法一亿个包中，只有一次丢失两个信包。若一个信包需 1 毫秒，那么每四个月才会丢失一个信包。若每个信包传输三次，每 30,000 年才会丢失一个信包。一个信包的多副本传输从一开始，其纯效果就是较小的并几乎总是有限的延迟。

时间触发协议

考虑到对实时分布式系统的限制，其协议通常是很独特的。这节介绍这样的一种协议 TTP（时间触发协议）（Kopetz 和 Grunsteidl, 1994），它同以太网协议是不同的，就像维多利亚画室与一个狂野西部沙龙的不同一样。TTP 应用在 MARS 实时系统（Kopetz 等, 1989）中，并在许多方面与它交织在一起，因此我们在有必要时将会提及 MARS 系统的特性。

MARS 中的一个节点包含至少一个 CPU，但通常有两、三个 CPU 一起工作，对外呈现为一个单一的容错，fail-silent 型节点。MARS 系统中的节点间是通过两个可靠的和独立的 TDMA 广播网连接的。所有信包在两个网中并行发送。期望的丢失率是每三千万年

丢失一个信包。

MARS 是一个时间触发系统，因此时钟同步是很重要的。时间是离散的，每微秒产生一次。TTP 假定所有时钟在数十微秒级的精度上准确同步。由于协议本身提供连续的时钟同步，而且设计该协议为允许在硬件上以极高的精度实现，因此以上的同步精确度是可能的。

所有 MARS 中节点都知道正在其他节点上运行的程序。特别是任意节点都知道其他节点什么时候发送一个信包，并且能很容易地检测到信包存在与否。由于信包已假定不会丢失（如上所述），在某一时刻，当一个节点期待的信包未出现时就意味着发送信包的节点崩溃了。

例如，假设当检测到一个意外事件时，广播发送一个信包通知其他每个节点。节点 6 将进行一些计算，2 毫秒后它在 TDMA 帧的第 15 号时隙内广播发送一个应答。若这个消息在预期的时隙内未到来，其他节点就假定节点 6 已失效，并采取所有可能的步骤来恢复。这种紧约束和瞬时一致性不再需要时耗协同协议，并使系统既是容错的，又是实时的。

每个节点都保留了系统的全局状态，这些状态要求各个地方都一样。若有某一个节点与剩下其他的节点步调不一致，这将是一个严重的（并可测出的）错误。这个全局状态有三个部分：

- (1) 当前模式；
- (2) 全局时间；
- (3) 当前系统成员位图。

模式是由应用程序定义的，并且与系统处于哪一阶段有关。例如在航空应用程序中，倒计时、发射、飞行和着陆可能都是独立的模式，每个模式都有自己的一套进程和这套进程的运行顺序，参加运行的节点列表，TDMA 时隙分配，消息名称和格式，以及合法的后继模式。

全局状态中的第二个域是全局时间，其单位由应用定义，但在任何情况下，它都必须足够粗略，以便所有节点都能达成一致。第三个域用于跟踪节点的增添。

与 OSI 和 Internet 协议包不同，TTP 协议包含一个单独的层用来处理端对端的数据传输，时钟同步和成员组织管理。一个典型的信包格式如图 4-26 所示，它包含一个信包起始域，一个控制域，一个数据域和一个 CRC 域。

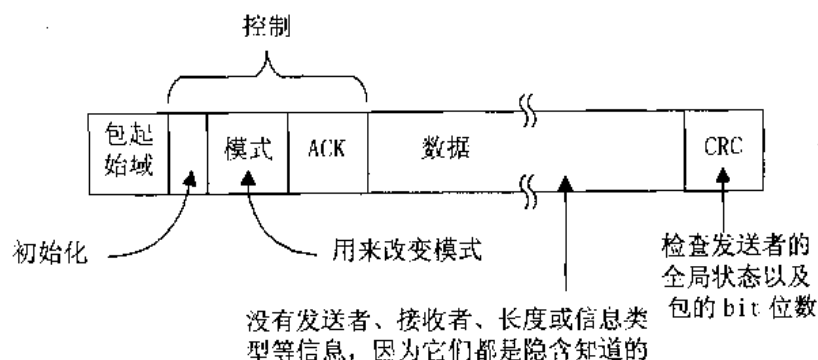


图 4-26 典型的 TTP 包

控制域包含一用于初始化系统的位（后面详述），一个变换当前模式用的子域和一个确认前一节点发送的信包的子域（根据当前的成员列表）。该域的目的是：让前面的节点知道自己正常运行并且它的信包正常地发送到网上，如果没有发生预期的确认，所有的节点标记应发确认信息的节点为无效，并将其从当前的成员状态图中删去。被删去的节点对被删除将毫无异议。

数据域包含所需的数据。CRC 域比较特别，它不仅对信包内容提供校验和，还对全局状态提供校验和。这就是说，若发送者的全局状态不正确，它发送的任何信包的 CRC 值将和接收者用自己状态计算出的全局状态值不一致。下一个发送者将不会应答确认这个信包，所有节点，包括错误状态的那一个，在成员关系位图中都将标记它为失效。

周期性的广播发送有初始化位的信包。这个信包包含当前全局状态。任何已被标记为不是一个成员，但在此模式中可认为是一个成员的节点，现在可以作为一个被动的成员加入，若某节点已被认为是一个成员，它将分配有一个 TDMA 时隙，因而在何时响应方面不存在问题（用它自己的 TDMA 时隙）。一旦它的信包被确认，所有的其他节点将再次标记它为活动（可操作）节点。

协议的最后一个有趣的方面是它处理时钟同步的方法。由于每个节点知道何时 TDMA 帧启动以及它的时隙在帧中的位置，因此它就能准确的知道何时开始发送信包。此方法避免了冲突。然而，如果一个信包在它应当发送时间之前或之后 n 微秒发送，其他每个节点都能检测到这一延迟，并把它作为本地时钟与发送者时钟之间的时间偏移估计值。通过监控每个信包的开始位置，节点会知道其偏移值，例如，每个其他节点开始传输的时间都推后 10 微秒。这时，可以合理的认为其时钟快了 10 微秒并对它进行纠正。通过记录其他信包发送的过早或过晚的平均值，不断纠正自己的时钟来与其他节点保持同步，而无须运行任何特别的时钟管理协议。

总之，TTP 协议的特别之处在于，接收者而不是发送者能够发现信包是否丢失，自动的成员关系协议，信包和全局状态的 CRC 校验码，以及时钟同步的方法。

4.6.4 实时调度

实时系统通常设计成为一些短任务（进程或线程）的集合，每个短任务都有明确的功能和有限的运行时间，对某个激励的响应可能需要多个任务来运行，并通常对它们有执行顺序上的限制。另外，必需决定哪一个任务在哪一个处理机上运行。本节讨论一些关于实时系统任务调度的问题。

实时调度算法的特征可用下面几个参数表示：

- (1) 硬实时与软实时；
- (2) 抢占的与非抢占的；
- (3) 动态的与静态的；
- (4) 集中的与分散的。

硬实时算法必须保证所有时限都要满足。软实时算法可忍受一种最大近似的方法。最重要的方法是硬实时。

抢占式调度允许在更高优先级的任务到来时暂时挂起当前的任务，以后在没有更高优先级的任务要运行时对恢复它的处理。非抢占式调度运行每个任务直至完成，即一旦一

个任务启动后，它将占有处理机直到执行结束。这两种调度策略都在使用。

动态算法是在运行期间进行调度。当检测到一事件时，动态抢占式算法立即决定是运行与这个事件相连的（第一个）任务还是继续运行当前的任务。而对于动态非抢占式算法，它仅知道有另一个任务可以运行，在当前任务结束后，它才在就绪的任务中选择一个来执行。

与上面相反，静态调度算法，不管是否是抢占式的，都是在运行前事先就调度好的。当一个事件发生时，运行的调度程序仅仅到一个表中查看应该执行什么操作。

最后，调度可以使用集中式的，用一个机器来收集所有的信息并作出所有的决定，或者使用分散式的，各个处理机可以独立做出决定。在集中式的情况下，同时可将任务分配给处理机。在分散的情况下，任务如何分配给各处理机与决定把哪一个任务分配给一个给定的处理机来运行是不同的。

所有实时系统的设计者所面临的一个关键问题是：是否有可能满足所有的限制。若一个系统有一个处理机，每秒有 60 个中断，每个需要 50 毫秒，设计者将会面临一个大问题。

假设一个周期性实时分布式系统有 M 个任务运行在 N 个处理机上。 C_i 表示任务 i 所需 CPU 时间， P_i 表示其周期，即两个连续的的中断之间的间隔时间。为使之可行，系统的利用率 μ 与 N 相关，符合如下等式：

$$\mu = \sum_{i=1}^m \frac{C_i}{P_i} \leq N$$

例如，若任务每 20 毫秒做一次，每次运行 10 毫秒，占用 0.5（50%）的 CPUs。五个这样的任务就得 3 个 CPU 来做。一个任务集合能满足上式的要求我们就说它们是可调度的。注意，上面的等式给出的所需 CPU 数较小于真正需要的 CPU 数，因为未考虑任务的转换时间、消息传输和其他一些额外开销，并假定可以进行最优化调度。

下面两部分将分别对周期性任务集合的动态和静态调度进行介绍。

动态调度

让我们先看一些著名的动态调度算法。这些算法在程序运行期间决定下面运行哪一个进程。比率单调算法(Liu 和 Layland,1973)，它是设计用来抢占式调度一个单一处理机上的没有顺序和互斥限制的周期性任务的。它是这样工作的。事先给每个任务分配一个与其执行频率相等的优先级。例如，一个每 20 毫秒运行一次的任务，分配其优先级为 50，一个每 100 毫秒运行一次的任务，分配其优先级为 10。运行时，调度程序总是选择优先级最高的任务运行，如果有必要，可先暂停运行当前任务。Liu 和 Layland 证明该算法是最优的，也证明了一任务集合若满足利用率条件：

$$\mu = \sum_{i=1}^m \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

则对该算法来说是可调度的。等式右边当 m 趋向于无穷大时，趋于 $\ln 2$ （约等于 0.693）。实际上，该限制是很保守的，因它通常对 μ 为 0.88 的任务集合也是可调度的。

另一个常用的抢占式动态调度算法是最早时限优先法（earliest deadline first）。每当

检测到一事件时，调度程序就将其加入等待任务队列中。这个等待队列根据这些任务的时限排序，最近的时限在最前面（对周期性任务时限就是它下一次发生时间）。然后，调度程序就从列表选择第一个任务调度，就是距它最后时限最近的一个。同比率单调算法一样，它也产生最优结果，即使是对 $\mu=1$ 的任务集合也是一样的。

第三种抢占式动态算法先计算出每个任务的剩余时间量，称为余度（松弛度）。对于一个必须在 200 毫秒内完成的任务，若还需运行 150 毫秒，则余度是 50。这种算法叫作最小余度法(least laxity)，即选择有最小余度的任务，即选择有最小喘息余地的任务来运行。

上述没有一个算法在分布式系统中证明它是最优的，但它们常作为启发式算法。同样它们也没有考虑到顺序和互斥的限制，哪怕是对一个单一的处理机，这使得它们只在理论上很有用，而在实践中不那么有效。因此，许多实用的系统，当它们事先可掌握有足够的信息时，通常使用静态调度算法。不仅是因为静态算法能考虑到那些额外限制，而且还能使额外的系统开销非常低。

静态调度

静态调度是在系统开始运行前就已进行。算法的输入包含了所有任务的列表及它们各自的运行时间。目标是将任务分配到各个处理机，并对每一处理机给出所要运行任务的静态运行顺序。理论上，调度算法可以穷举所有的调度方案以找到最优方案，但这一搜索时间随任务数成指数增长（Ullman,1976），因此，通常使用上述的各种启发式方法。在这里不再给出其他的启发式算法，而是给出一个详细的例子，以说明非抢占式静态调度的实时分布式系统中调度与通信的相互影响。

先假定，每次检测到一个特定的事件。任务 1 就在处理机 A 上启动，如图 4-27 所示。这个任务依次在本地和另一个处理机 B 上启动其他任务。为了简化，假定将任务分配给处理机是由外部因素决定的（哪一个任务需要访问哪一个 I/O 设备），并且在这里它不是一个参数。所有的任务都需要一个 CPU 时间单位。

任务 1 启动本地机上的任务 2 和 3，同时也启动 B 处理机上的任务 7。这三个任务又都启动另一个任务，以此类推，如图 4-27 中所示。箭头表示任务间传递的消息。在这个简单的例子中， $X \rightarrow Y$ 表示 Y 不能运行，除非从 X 发出的消息已经传到它那了。有些任务，如任务 8，它启动运行前，要求有两条消息。当任务 10 运行完，并对最初的激励产生预期的响应时，整个周期就结束了。

任务 1 完成后，任务 2 和 3 都是可运行的。调度程序要选择哪一个接着运行。假定调度程序选择任务 2，则又要在任务 3 和 4 之间选择一个成为任务 2 的后继任务。若选择任务 3，就还会有在任务 4 和 5 之间的选择。然而，若选择任务 4 而不选择任务 3，就必需在任务 4 后运行任务 3，不能启动任务 6，因为只有任务 5 和 9 都执行后它才可运行。

同时，B 处理机上的工作也可并行进行：任务 1 一启动它，B 上的任务 7 就运行（可以和任务 2 或 3 同时运行）。当任务 5 和 7 都完成时，任务 8 开始运行，如此进行下去。注意，任务 6 需要从任务 4、5 和 9 来的输入，并产生输出给任务 10。

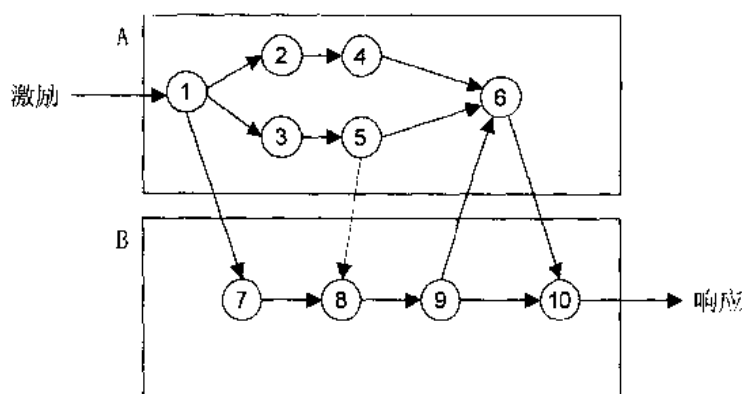


图 4-27 在两个处理机上执行 10 个实时任务

如图 4-28 (a) 和 (b) 是两种可能的调度方法，图中不同处理机上的任务间的消息用箭头表示。同一机器上的任务间的消息在机器内部处理，没有标出。在所描述的两中调度方法中，图 4-28 (b) 更好些，因为它让任务 5 提早运行，从而使得任务 8 可以更早地运行。若任务 5 延后执行，如图 4-28 (a)，那么任务 8 和 9 就会延后执行，从而使任务 6 以及后来的任务 10 也都被延后执行。

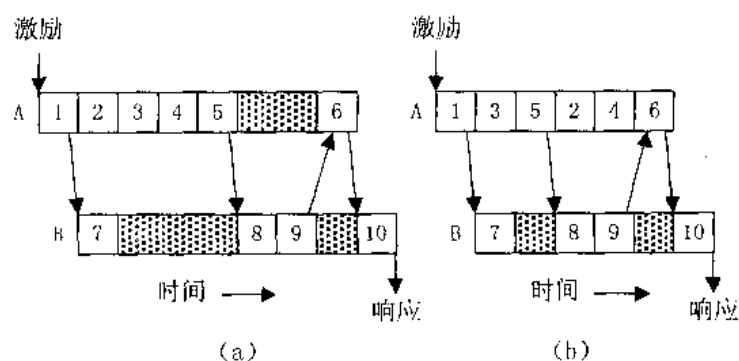


图 4-28 图 4-27 中的两种可能任务调度方式

对于静态调度，在系统启动前，调度程序就要事先决定是使用上面所述的两个调度策略中的一个还是使用其他的替代调度策略。调度程序对图 4-27 进行分析，并使用所有任务的运行时间作为输入信息，然后用一些启发式算法找到一个较好的调度策略，一旦调度策略确定，所作的选择将分析并入表中。因此运行时，使用一个简单的分配器就可用较小的开销实现调度。

现在我们将分析上述任务的调度问题，但这次要考虑通信的问题。我们将使用 TDMA 通信法，每个 TDMA 帧有 8 个时隙，本例中，一个 TDMA 时隙的大小等于一个任务执行时间的 1/4。我们人为地将时隙 1 分配给处理机 A，时隙 5 分配给处理机 B。将 TDMA 时隙分配给各个处理机是由静态调度算法决定的，可随程序所处的阶段的不同而不同。

如图 4-29，图中表示了对图 4-28 的两种调度策略，但现在对 TDMA 时隙的使用进行了考虑。一个任务只有在它的处理机的时隙到来时，才可发送消息。因此，任务 5 直到循环到下一个 TDMA 帧的第一个时隙时，才可向任务 8 发送消息，因而在图 4-29 (a) 中出

现了前面图中所没有的任务 8 的启动延迟。

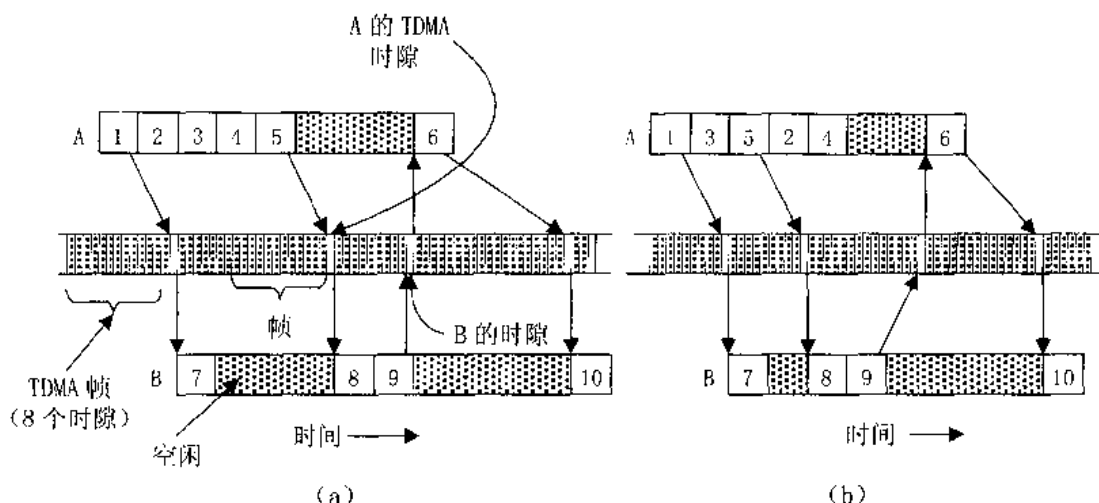


图 4-29 包括处理和通信的两种调度

在这个例子中，值得注意的是系统运行行为是完全确定性的，甚至是在程序开始执行前就已经知道了。只要通信和处理机不出错，系统总能满足它的实时时限的要求。处理机失效可通过每个节点设置两个或更多的 CPU，并使它们彼此互相动态跟踪来屏蔽掉。若有必要，额外时间可静态地分配在任务之间的间隙期内，以用于恢复。对于丢失的或出错的信包可以这样解决：每个信包一开始就发送两次，或者是通过不相连的网络发送或者是在一个网络上使 TDMA 时隙等于两个信包宽度的方法来发送。

到目前为止已经很清楚，实时系统并不是要“榨出”硬件的最后“一滴”性能，而是要用附加额外的资源来确保系统在任何条件下都能满足各种实时限制。然而在我们的例子中，对通信带宽使用相对较低的情况下并不具有代表性。造成这种情况是因为这个例子仅用了两个处理机，且通信要求适中。实际的实时系统会有许多处理机和大量的通信需求。

动态与静态调度的对比

选择用动态调度还是静态调度是很重要的，对系统也会产生深远的影响。静态调度对时间触发系统的设计很适合，而动态调度对事件触发系统的设计很适合。静态调度必需事先仔细设计，并要花很大的力气考虑选择各种各样的参数。动态调度不要求事先做多少工作，它是在执行期间动态地作出决定。

在资源利用方面动态调度比静态调度有更大的潜力。对于静态调度，系统通常需要过大范围的设计，以能够处理最不可能的事件。可是，在硬实时系统中，浪费资源通常是为保证满足系统所有的时限而付出的代价。

另一方面，若给定足够的处理能力，对静态系统一个最优或次优的调度策略可事先获得。例如，对一个反应堆控制的应用，花费数月的 CPU 时间来找出最好的调度策略是值得的。一个动态系统在运行期间无法承受复杂的调度计算花费。因此为了安全，就不得不在过大范围内设计，即使如此，也不能保证系统能够满足对它的指标要求。取而代之的是，需要大量的测试。

最后应指出，我们讨论的是大大简单化了的情况。例如，各个任务可能需要访问共享变量，因此，这些变量要预先进行保存。通常的一些调度限制，我们也忽略了。最后要说明的一点是，一些系统在运行中间会作一些事先的计划，这使它们成为介于动态和静态之间的混合体。

4.7 小结

尽管线程不是分布式系统所固有的，但大多数的分布式系统都支持线程包，所以在本章中我们学习了线程。线程是一种轻量级的进程，它可以与其他的一个或多个线程共享地址空间。各个线程有自己的程序计数器和堆栈，它独立于其他线程而调度。当一个线程发出一个系统阻塞调用时，其他处于同一地址空间的线程不会受到影响。线程打包可以在用户空间内或系统核心内实现，但两种方法都有自己要解决的一些问题。使用轻量级线程导致了轻量级 RPC（远程过程调用）的一些有趣的结果。弹出式（POP-UP）线程也是一个重要的技术。

有两种经常使用的组织处理机的模型：工作站模型和处理机池模型。在工作站模型中，每个用户都有自己的工作站，有时可以在别的空闲工作站上运行进程。在处理机池模型中，所有的处理能力是一个共享的资源。处理机通常根据需要动态地分配给用户，当工作完成后将处理机收回至处理机池中。还有另一种模型是混合模型。

给定一个处理机的集合，需要有一个算法来决定如何将进程分配给各个处理机。这个算法可以是确定性的或是启发式的、集中式的或分布式的、最优的或次优的、局部的或全局的、发送者启动的或接收者启动的。

尽管进程通常都是独立进行调度的，可是使用协同调度（co-scheduling），可以保证必需进行通信的进程同时运行，从而提高了系统性能。

对于大多数分布式系统来说，容错性是很重要的。它可以用三模冗余、主动复制或主机-后备机复制来实现。两军（Two-army）问题在不可靠通信的情况下是无法解决的，但是对于 Byzantine 将军问题，如果超过 $2/3$ 的处理机是无错的，问题是可以解决的。

最后，实时分布式系统也是很重要的。有两种类型：硬实时系统和软实时系统。事件触发的系统是中断驱动的，而时间触发的系统可以以固定的时间间隔对外围设备进行采样。实时通信要求使用可预测的协议，如令牌环或 TDMA。任务的动态或静态调度都是可以实现的。动态调度在运行时进行，静态调度在运行前就完成了。

习 题

- (1) 本问题是要比较使用单线程文件服务器与多线程文件服务器读取一个文件的不同之处。如果要求的数据在缓存中，收到任务请求，分配这个工作，并作相应的处理需要 15 毫秒；如果要进行磁盘读写，并且进行读写占用 $1/3$ 的时间，则需要多使用 75 毫秒，在盘操作时，线程休眠。请分别计算单线程文件服务器和多线程文件服务器每秒各能处理多少条请求？
- (2) 在图 4-3 中，寄存器集是以每线程为一个单位被列出的而不是以每进程为一个单位列

出,为什么?要知道,整台机器只有一套寄存器。

- (3) 在本章中,我们描述了多线程文件服务器比单线程服务器和有限状态机要好。试举例说明是否在某些情况下,单线程服务器要优于多线程服务器?
- (4) 在关于线程中的全局变量的讨论中,我们使用了一个 *create_global* 过程来为指向全局变量的指针分配内存而不是给全局变量本身。这是必需的吗?或这个过程直接为变量本身分配内存是可行的吗?
- (5) 试考虑下面的系统:线程完全在用户空间内实现,运行系统每秒得到一个时钟中断。如果时钟中断在某个线程正在系统中运行时发生,可能会出现什么问题?你能够给出一种解决办法吗?
- (6) 如果一个操作系统不提供任何类似 SELECT 系统调用的机制来事先判断读写一个文件,管道,或设备是否安全,但系统允许设置告警时钟来中断阻塞的系统调用。在这样的条件下可以在用户空间中实现线程包吗?请讨论。
- (7) 在一个基于工作站的系统中,工作站有自己的磁盘存放系统的可执行程序。当发布一个新的可执行程序后,它被发至每个工作站,但这时有些工作站可能停机(或关机了)。试设计一个算法使这种更新可以自动执行,即使是工作站会偶然停机。
- (8) 如第7题所描述的是工作站类型,试列出其他种类的能够安全存放于用户工作站上的文件。
- (9) Bershad 等人提出的加速本地 RPC(远程过程调用)的方案在其他每进程仅含一个线程的系统中仍然可行吗?Peregrine 的方案呢?
- (10) 在图 4-10 中,如果两个用户同时检查注册表,他们很有可能会选中同一台空闲工作站。如何改进算法以避免这种竞争问题?
- (11) 假设一个进程远程运行于一个以前空闲的工作站,并且它与其他所有的工作站一样都是无盘的。对于以下每一个 UNIX 系统调用,请判断是否应该将调用返回主机:
READ (从文件读数据)
IOCTL (改变主控终端的模式)
GETPID (取进程号)
- (12) 以处理机 1 为基准处理机,计算图 4-13 中的响应率。如何分配进程,以使平均响应率达到最小?
- (13) 我们讨论处理机分配算法时,指出一种是在集中式和分布式之间选择,另一种是在最优的和次优的之间选择。试给出两种最优分配算法,一个分布式的、一个集中式的。
- (14) 在图 4-15 中,我们讨论了两种不同的分配算法,它们的网络通信量也不相同。还有其他更好的分配算法吗?假设每台处理机最多运行 4 个进程。
- (15) 正文中讨论的上-下算法是一个集中式的算法,用来平均分配处理机。试设计一种集中式算法,以始终一致的方式分配负载而不是平均分配。
- (16) 当某个分布式系统过载时,系统进行 m 个尝试,以寻找一个空闲的工作站来减轻其负载。一个工作站有 k 个进程的概率符合泊松分布:

$$P(k) = \frac{\lambda^k e^{-\lambda}}{k!} \quad (\lambda \text{ 为每个工作站的平均进程数})。$$

计算过载工作站在 m 次请求内能够找到空闲工作站的概率(即 $k=0$)。

计算在 $m=3, \lambda=1,2,3,4$ 时的结果。

- (17) 使用图 4-18 中的数据,最多可以调度多长的 UNIX 管道?

- (18) 三模冗余可以处理 Byzantine 错误问题吗？
- (19) 图 4-19 最多可以处理多少个失效的部件（设备和表决器）？试给出一个可被屏蔽掉的最坏情况的例子。
- (20) TMR 可以扩展到每组五个部件而不是三个吗？如果可以，它有何特性？
- (21) Eloise 住在 Plaza 酒店，她喜欢的活动是站在大厅里几个小时不停地按电梯按钮。Plaza 酒店正在安装新的电梯系统，若他们可在事件触发系统和时间触发系统中选择，那他们应选择哪一个呢？
- (22) 一个实时系统中有下面四个周期出现的进程（前面是计算所需时间，后面是周期）：
P1：需要 20 毫秒，每 40 毫秒出现一次
P2：需要 60 毫秒，每 500 毫秒出现一次
P3：需要 5 毫秒，每 20 毫秒出现一次
P4：需要 15 毫秒，每 100 毫秒出现一次
请问该系统在单 CPU 上是否可调度？
- (23) 在前一个问题中，是否能够运用比率单调算法给每个进程分配优先级？如果可以，试给出优先级；否则给出原因，指出还缺少什么信息。
- (24) 一个网络有两条并行线路：一条正向，信包由左至右；另一条反向，信包由右至左。两条线路两端都有一个发生器产生连续的信包帧流，每一个信包中有“空/满”位表示信包是否包含信息（初始为“空”）。所有的计算机都在这两条线路之间，并与两条线都相连。要发送信包时，计算机先根据目的地是在左边还是在右边来决定使用哪条线路，然后等待一个空帧到来，将一个信包放入帧中，并标记该帧为“满”。这种网络符合实时系统的要求吗？试解释。
- (25) 图 4-29 中给进程分配时限过于随意，还有其他的分配方法。试为例二找出一个能提高系统性能的更好的分配方法。

第 5 章 分布式文件系统

对任何分布式系统来说，文件系统都是其中的一个关键组成部分。与在单处理机系统中一样，分布式系统中文件系统的工作也是存储程序和数据，在需要时可使用它们。分布式文件系统在许多方面类似于传统的文件系统，在这里就不再赘述了。相反地，我们将着重讨论分布式文件系统不同于集中式文件系统的那些方面。

首先，在分布式系统中，区分文件服务和文件服务器这两个概念是非常重要的。文件服务是文件系统提供给客户内容的详细说明。它描述了可用的原语，以及原语所需的参数和执行的动作。对于客户来说，文件服务精确地定义了他们所期望的服务，但并不涉及这些服务是如何实现的。实际上，文件服务说明了文件系统对客户接口。

相反，文件服务器是运行在某台机器上的一个有助于实现文件服务的进程。一个系统可以有一个或多个文件服务器。特别是，客户可能不知道有多少个文件服务器以及每个服务器的位置和功能。他们所知道的是当它调用文件服务中的具体过程时，所要求的工作以某种方式执行，并返回所需的结果。实际上，客户甚至不知道文件服务是分布的。理想情况下，它可能看上去和通常的单处理机文件系统一样。

由于文件服务器通常仅仅是运行在某台机器上的一个用户进程（有时是一个核心过程），所以一个系统可包含多台文件服务器，每台服务器提供不同的文件服务。例如，一个分布式系统可以有两个文件服务器，分别提供 UNIX 文件服务和 MS-DOS 文件服务，每个用户进程可以分别地使用适合它的文件服务器。在这种方式下，分布式系统的终端就可能有多窗口，一些 UNIX 程序运行在一些窗口，而一些 MS-DOS 程序运行在另一些窗口而互不冲突。服务器是否提供特殊的文件服务，如 UNIX 或 MS-DOS，或者更多的一般性服务取决于系统设计者。可用的文件服务的类型和数量甚至可以随着系统的发展而改变。

5.1 分布式文件系统设计

分布式文件系统通常包括两个截然不同的部分：真正的文件服务和目录服务。前者涉及单个文件上的操作，例如读、写和追加，而后者涉及创建和管理目录，在目录中增加和删除文件等。本节我们讨论文件服务接口，下一节再讨论目录服务接口。

5.1.1 文件服务接口

无论是在单处理机系统中还是在分布式系统中，对于任何文件服务，最基本的问题是：什么是文件？在许多系统中，如 UNIX 和 MS-DOS，文件是一个未经解释的字节序列。文件中信息的含义和结构完全取决于应用程序，操作系统对此不感兴趣。

然而，在主机上存在着多种类型的文件，每种类型的文件都有各自不同的特性。一个文件可以组织成一个记录系列，例如，根据操作系统要求来读写一个特定的记录。这个记录通常由给定它的记录号（即在文件中的位置）或者某一字段的值来确定。在后一种情况下，操作系统既可以把文件作为 B 型树来保存，也可以将文件作为其他合适的数据结构来保存，或者使用哈希表（Chash tables）来迅速定位记录。由于大多数分布式系统都是运行在 UNIX 或者 MS-DOS 环境，所以大部分文件服务器都支持文件作为一字节序列的概念，而不支持文件作为一段关键字的记录序列的概念。

文件有多个属性，这些属性都是关于文件的一部分信息，而不是文件本身的一部分。典型的属性有：所有者、大小、创建日期和访问权限。文件服务通常提供读写某些属性的原语，例如，有可能改变访问权限而不用改变文件大小（除非对文件追加数据）。在少数高级系统中，可以建立和使用除标准属性以外的用户自定义属性。

文件模型的另一个重要方面是文件创建后能否修改，通常是可行的。但在某些分布式系统中，对文件的操作只有 CREATE 和 READ。一旦文件创建了，就不能改变它。这样的文件称为是不可变的（immutable）。保持文件的不可变性，使得支持文件高速缓存和复制变得更为容易，因为它消除了有关文件改变时必须修改所有文件拷贝的全部问题。

分布式系统的保护基本上使用了与单处理机系统相同的技术：权能（capability）和存取控制表。就权能而言，每个用户拥有访问每个对象的某种门票，称作权能。权能指定了允许的访问类型（例如，允许读但不允许写）。

所有的存取控制表模式将每个文件与可以访问它的用户以及访问方式联系起来。UNIX 模式就是一个简化了的存取控制表，它通过使用二进制位来分别控制所有者、所有者组、以及其他每个人对每个文件的读、写、以及运行。

根据是否支持上载/下载（upload/download）模式或远程访问模式，文件服务可以分成两大类。在上载/下载模式中，如图 5-1(a)所示，文件服务只提供两种主要的操作：读文件和写文件。前一个操作是将整个文件从一个文件服务器传送到提出请求的客户；后一个操作是将整个文件从客户传送到服务器。因此这种概念模式是在任一方向上传送整个文件。这些文件可以存储在内存或本地的硬盘，视需要而定。

上载/下载的优点是概念简单。应用程序取得它们所需的文件，然后在本地使用它们。任何修改过的文件或新创建的文件在程序结束时都要将它回写。使用这种模式不需要掌握复杂的文件接口，而且，整个文件传送也是高效的。但是，客户端必须具有足够大的存储空间来存储所需的所有文件。而且，如果只需要文件的一小部分，移动整个文件是很浪费的。

文件服务的另一种类型是远程访问模式，如图 5-1(b)所示。在这种模式中，文件服务提供了大量的操作用于打开和关闭文件，读写文件的一部分，在文件中来回移动 (LSEEK)，检查和改变文件属性等等。而在上载/下载模式中，文件服务只提供物理存储和传送，在这里文件系统运行在服务器上而不是运行在客户端。远程存取模式的优点是在客户端不需要很大的空间，当仅需要文件的一小部分时，不需要传送整个文件。

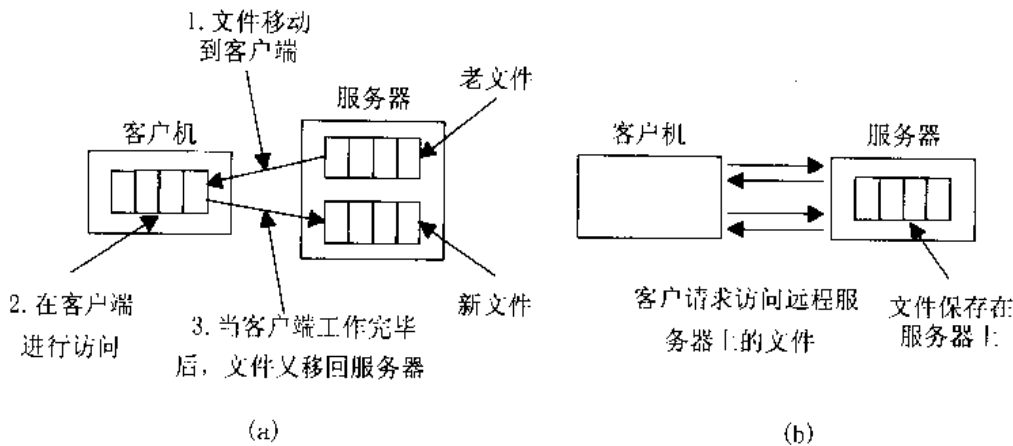


图 5-1 (a) 上传/下载模式
(b) 远程访问模式

5.1.2 目录服务器接口

文件服务的另一部分是目录服务，它提供诸如创建和删除目录，命名和重命名文件以及将文件从一个目录移动到另一个目录等操作。目录服务的性质并不依赖于单个文件是整体传送还是远程访问。

目录服务定义了构成文件(目录)名的某种字母表和语法。文件名通常是从 1 到某一最大数的字母、数字和某些特殊字符。有些系统将文件名分成两个部分，通常用一个点分开，如用 *prog.c* 表示一个 C 程序或 *man.txt* 表示一个文本文件。文件名的第二部分叫作文件扩展，标识文件的类型。其他的系统使用一个显式属性来达到此目的，而不是在文件名上添加一个扩展名。

所有的分布式系统都允许目录包含子目录，这使得用户可以把相关文件组合到一起。相应地，系统提供了创建和删除目录的操作，也提供了在目录中插入、移动和查找文件的操作。通常，每个子目录包含一个项目的所有文件，如一个大程序或文档(如一本书)。当显示该(子)目录时，只显示相关的文件，无关文件在其他(子)目录中，这样就不致使显示列表凌乱。子目录可以包含它们自己的子目录，以此类推，这样就形成一棵目录树，通常称为分层文件系统。图 5-2(a)表示了一棵有五个目录的树。

在某些系统中，可以对任意目录建立连接或指针。这种连接和指针可以放在任一目录中，使得不仅可以按树结构组织目录，而且可以将目录组织成更强有力的任意目录图。在分布式系统中，树和图的区别特别重要。

由图 5-2(b)的目录图可见难点的性质。在这张图中，目录 D 连接到目录 B 上。当删除 A 到 B 的连接时，就出现问题了。在树型层次结构中，仅当目录为空时，才能删除指向该目录的连接。在目录图中，一个目录只要存在至少另一条链接就可以删除指向该目录的链接。通过保存一个参考计数，如图 5-2(b)中的每个目录的右上角所示，就可以确定什么时候删除的链接是最后一个链接。

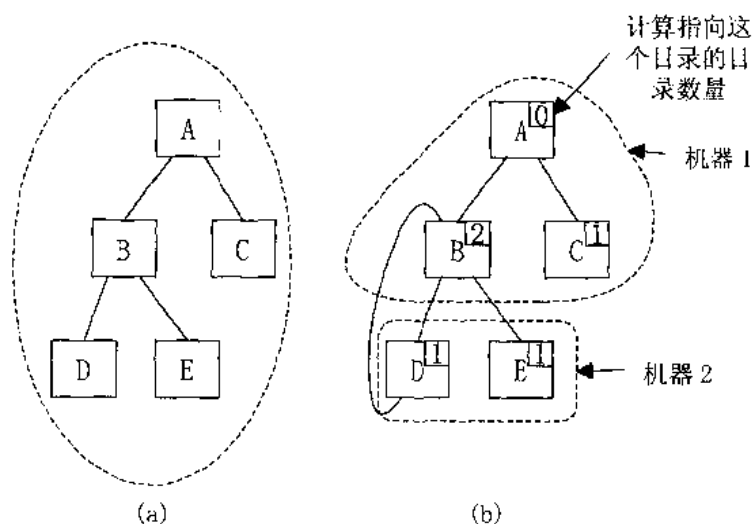


图 5-2 (a) 包含在一台机器中的目录树
(b) 在两台机器中的目录图

删除 A 到 B 的连接之后, B 的参考计数从 2 减为 1, 这在理论上是行得通的。然而, 现在从文件系统 (A) 的根已不能达到 B 。 B 、 D 和 E 三个目录以及它们所有的文件实际上是孤立了。

这样的问题在集中式系统中也存在, 但在分布式系统中显得更加严重。如果所有内容都在一台机器上, 虽然花销有点昂贵, 但是当发现孤立目录时, 因为所有的信息都在一个地方, 就可以停止所有的文件活动。并且从图的根开始遍历, 标记所有可达到的目录。当这个过程结束时, 所有未标记的目录将认为是不可达到的。在分布式系统中, 涉及到多台机器, 不能停止所有的活动, 因此得到一个“瞬态图”即使有可能也很困难。

在设计任何分布式系统时, 一个关键的问题是: 是否全部机器(和进程)都应该具有完全相同的分层目录结构视图。为说明这一问题, 给出一个例子 (参考图 5-3)。在图 5-3(a) 中我们给出了两个文件服务器, 每个文件服务器有三个目录和一些文件; 在图 5-3(b) 中, 我们有一个系统, 其中所有的客户(和其他机器)具有相同的分布式文件系统视图, 如果路径 $/D/E/x$ 在一台机器上有效, 则在所有的机器上都有效。

相反, 在图 5-3(c) 中, 不同的机器可能具有不同的文件系统视图。重复前面的例子, 路径 $/D/E/x$ 在客户 1 可能是有效的, 而在客户 2 上可能是无效的。在通过远程安装管理多个文件服务器的系统中, 图 5-3(c) 是一个标准。其实现是灵活且简单的, 但其缺点是不能使整个系统的行为变成像一个单个过时的分时系统一样。在分时系统中, 对任何进程, 文件系统看起来都是一样的 (即 5-3(b) 所示的模型)。这个特性使得系统容易编程和理解。

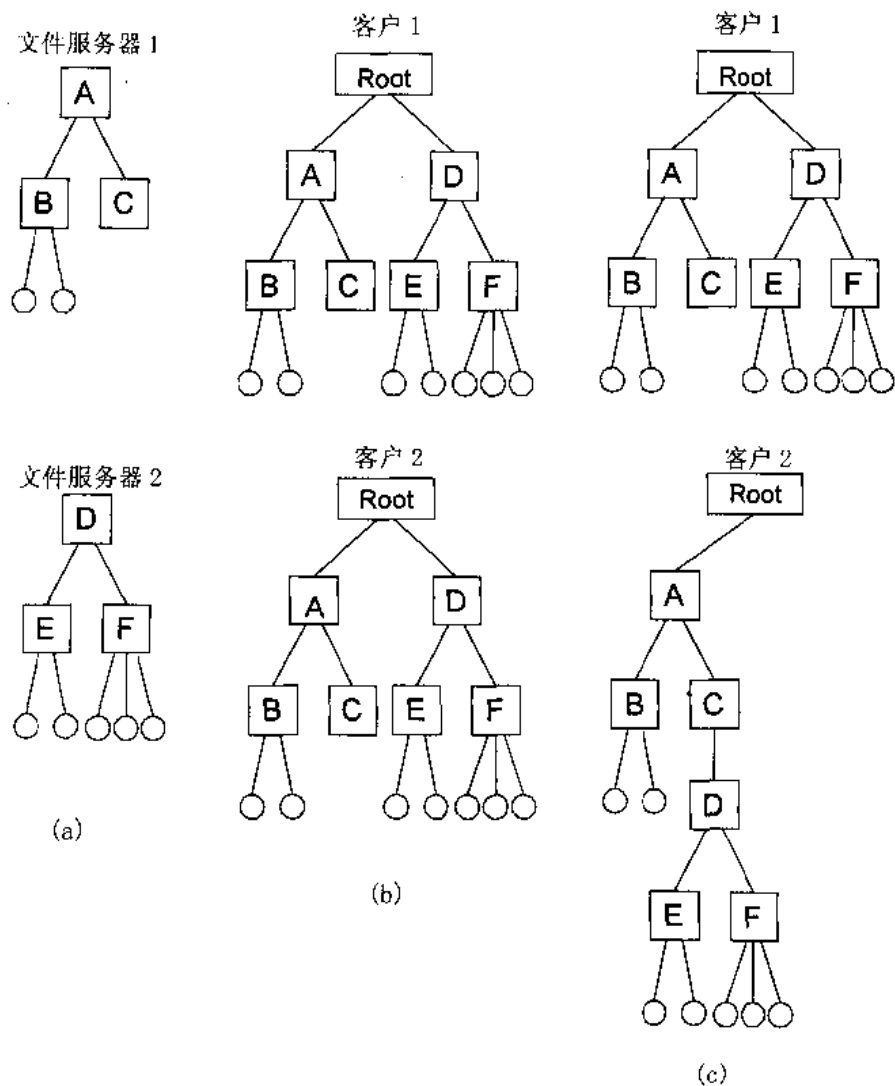


图 5-3 (a) 两文件服务器
 (b) 一个所有用户有同样文件系统图的系统
 (c) 一个不同的客户有不同的文件系统的系统
 注：方框是目录，圆圈是文件

基于此，一个密切相关的问题是，是否存在一个全局根目录，所有的机器都认为其为根。设置全局根目录的一个方法是使这个根只包含每个服务器的一个入口。在这种情况下，路径的形式为：/服务器/路径，虽有它自身的缺点，但至少在系统中任何地方都是一样的。

命名透明性

基于此，这种命名形成的主要问题就是：它不是完全透明的。在上下文中有两种形式的透明性是相关的，且有必要加以区分。第一种是位置透明性，是指路径名并不能对文件（或其他对象）存储位置给出任何提示。像路径/server1/dir1/dir2/x 说明 x 位于服务器 1

上,但是它并没有说明服务器 1 的位置。服务器在网络中可以自由地移动到它想去的任何地方而无需改变路径名。因此,这样的系统具有位置透明性。

然而,假设文件 *x* 非常大而且服务器 1 上的空间又很紧张。进一步,假设在服务器 2 上有足够的空间。系统很可能自动地把文件 *x* 移到服务器 2 上。不幸的是当所有路径名的第一部分是服务器时,尽管 *dir1* 和 *dir2* 在两个服务器上都存在,但系统不能自动地将文件移动到其他的服务器上去。问题在于自动的移动文件要将路径名由 */server1/dir1/dir2/x* 变为 */server2/dir1/dir2/x*。如果路径改变,具有前一个路径字符串的程序将停止工作。在一个系统中文件能够移动而无需改变其文件名,这样的系统称之为具有位置独立性。一个明确将机器或服务器名嵌入路径名的分布式系统不是位置独立的系统。基于远程安装的系统也不具备位置独立性,因为它不可能将一个文件从一个文件组(安装单元)移动到另一个文件组并仍可使用旧路径名。位置独立性是不容易实现的,但它是分布式系统所期望拥有的一个特性。

总结前面所述的内容,在分布式系统中用于文件和目录命名有以下三种常见的方法:

- (1) 机器+路径命名,如 */machine/path* 或 *machine:path*;
- (2) 安装远程文件系统到本地文件分层结构;
- (3) 一个在所有机器上看上去都一样的单个名字空间。

前两种方法容易实现,特别是作为一种并非为分布式系统设计的连接现存系统的方法。实现第 3 种方法很困难并且需要仔细设计,但如果要实现使分布式系统像单个计算机一样工作的目标,这种方法是需要的。

两级命名

大多数分布式系统都是使用某种形式的两级命名。文件(和其他对象)具有符号名(比如, *prog.c*),供用户使用,但它们还可具有内部的二进制名供系统自己使用。目录实际上起的作用是提供这两个命名层次之间的一个变换(映射)。用户和程序使用符号(ASCII)名是很方便的,但在系统内部使用,这些名字太长且不方便。这样当用户打开一个文件或另外引用一个符号名时,系统将立即在适当的目录中查找符号名以得到其用于定位该文件二进制名。有时,二进制名对于用户是可见的,而有时候是不可见的。

二进制名的特性随系统的不同而有显著的变化。在一个由多个文件服务器组成的系统中,其中每个文件服务器都是自含的(即它不引用其他文件服务器中的目录或文件),二进制名可以像在 UNIX 系统中一样仅仅是一个局部的 *i*-节点数。

一个更常见的命名模式是让这个二进制名同时指明服务器和在它上面的一个具体文件。这种方法允许一个服务器上的目录拥有一个不同的服务器上的文件。完成此功能有时被首选另一种方法使用符号连接。符号连接是一个目录项目,它映射到一个(服务器,文件名)字符串上,此串可以在为查找其二进制名而命名的服务器上查到。符号连接本身只是一个路径名。

而另一种思想是使用权限作为二进制名。在这种方法中,查找一个 ASCII 名就产生一个权限,这种权限具有多种形式。例如,它可以包含一个物理的或逻辑的机器号或者某个适当的服务器的网络地址,还可以是一个表明所需文件的数字。物理地址常用于向服务

器发送消息而无须作进一步的解释。逻辑地址可通过广播或通过在一个名字服务器上查找来定位。

一种有时在分布式系统中出现，而在集中式系统中不会出现的最新情况是：当查找一个 ASCII 名时，有可能得到多个二进制名(i-节点，权限等)。这些二进制名通常代表了原文件和它的所有备份。拥有多个二进制名，就可对其中一个文件进行定位。如果由于某种原因那个文件不可用，还可以另试一个文件。这种方法通过冗余而实现了一定程度上的容错。

5.1.3 文件共享的语义

当两个或更多用户共享相同的文件时，必须精确地定义读和写的语义以避免产生问题。在允许进程共享文件的单处理机系统中，如 UNIX 这种语义通常规定，在 READ 操作紧跟在 WRITE 操作后执行时，READ 操作返回刚刚写入的值，如图 5-4(a)所示。同样地，当一个 READ 操作跟在两个紧相连发生的 WRITE 操作后，读入的值就是后一个写操作写入的值。实际上，系统对所有操作都强制一个绝对时间顺序，并且总是返回最近的值。我们称这种模式为 UNIX 语义。这种模式易于理解且容易实现。

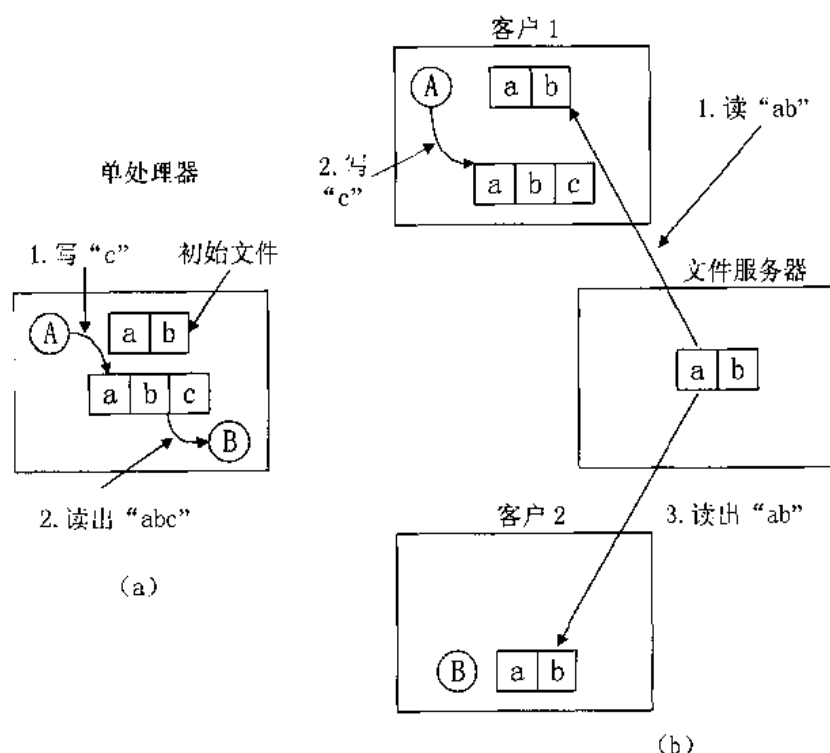


图 5-4 (a) 在单处理机中，当 READ 操作紧跟 WRITE 操作执行时，READ 操作返回刚刚写入的值
(b) 在分布式系统中，可能返回已过时的值

在分布式系统中，只要仅有一个文件服务器并且客户不缓存文件，那么 UNIX 语义是很容易实现的。所有 READ 和 WRITE 操作直接送到文件服务器上，文件服务器严格地

按顺序处理它们。这种方法表达了 UNIX 语义(除了由于网络延迟导致一个在写操作后仅一微秒就产生的读操作先到达服务器从而取得原有值的小问题外)。

然而, 实际上, 如果分布式系统中所有的文件请求都必须送到某个服务器上去执行, 这个系统的运行效率常常是很差的。解决这一问题的常用办法是允许客户在他们自己的高速缓存中保留经常使用的文件的局部拷贝。尽管稍后我们将讨论文件高速缓存的具体细节, 但现在也可明确地指出, 如果一个客户在本地修改了一个高速缓存文件且紧接着另一个客户从这个服务器上读取这一文件, 那么第二个客户就会得到一个过时的文件, 如图 5-4(b)所示。

解决这一问题的一个方法是立即将高速缓存文件的所有修改回写到服务器上。这种方法在概念上简单, 但效率很低。另一种方法是放宽文件共享的语义。为了替代一个读操作需看先前所有写操作的结果的需求, 这里介绍一个新的规则: “对一个打开文件的修改仅对修改该文件的进程(也可能是机器)是初始可见的, 仅当文件关闭时, 其修改才对其他进程(或机器)可见。”这个规则的采用并不能改变图 5-4(b)中的情形, 但它确实正确地重新定义了实际的操作行为(*B* 得到了文件的初始值)。当 *A* 关闭文件时, 它把拷贝发送到服务器, 这样, 后面的读操作得到了所需的新值, 这个规则已得到广泛应用并称之为对话语义。

如果两个或者多个客户同时进行高速缓存和修改同一文件, 使用对话语义就会发生问题。一个解决办法是: 当各个文件连续关闭时, 将它的值送回到服务器上, 这样, 最后的结果取决于哪个文件最后关闭。一种稍差但更易实现的方法是最后的结果是候选者之一, 但并未指明是哪一个。

使用高速缓存和对话语义的一个最终困难是: 除了没有让所有的读操作都返回最近回写的那个值之外, 它还违背了 UNIX 语义的另一个方面。在 UNIX 中, 联系每一个打开文件的是指出文件当前位置的指针。读操作从该位置开始取数据, 写操作从该位置开始写数据。该指针在打开这个文件的进程和其所有子进程之间共享。使用对话语义, 当其子进程在不同的机器上运行时, 这种共享就无法实现。为了清楚了解抛弃共享文件指针的后果是什么, 请看如下命令:

```
run>out
```

这里, *run* 是一个外壳程序, 它执行两个程序 *a* 和 *b*, 一个跟在另一个后面。如果两个程序都产生输出, 则希望在 *out* 中, 由 *b* 产生的输出直接跟在由 *a* 产生的输出后面。实现这种输出的方法是当 *b* 启动时, 它就继承来自 *a* 的文件指针, 这个文件指针是由外壳和两个进程共享的。这样 *b* 写的第一个字节就直接跟在 *a* 写的最后一个字节之后。如果使用对话语义而不用共享文件指针, 那就需要一个完全不同的机制来产生外壳程序和使用共享文件指针工作的类似结构。由于没有解决这个问题的通用方法, 所以每个系统都必须用特定的方式来处理它。

在分布式系统中, 对文件共享语义的一个完全不同的方法是使所有文件都不可变。这样, 就没有办法打开一个文件来进行写操作。事实上, 对文件的操作仅有创建和读。

创建一个全新的文件并用以前存在的一个文件名进入目录系统是可能的, 但还是以

前存在的那个文件就不可访问了（至少以该文件名）。这样，虽然不可能去修改文件 x ，还是有可能用一个新文件来自动地替代 x 。换句话说，虽然文件不能更新，但目录是可以更新的。一旦我们确定文件不能够被修改，有关如何处理两个进程的问题，即写文件进程和读入文件的进程，就不出现了。

遗留的问题是：当两个进程试图在同一时刻去替换同一文件时，将发生什么问题。对于会话语义，最好的解决办法似乎是让其中一个新文件去替换旧文件：要么用最后一个文件，要么不确定的。

更困难的问题是如果一个文件被代替而另一个进程正在忙于读它时，应如何处理。一种解决的办法是：以某种方式安排读进程继续使用原有文件，即使它不再处于任何目录中，类似于 UNIX 允许进程继续使用它所打开的文件，即使该文件已从所有的目录中删去。另一种解决的办法是检测已改变的文件并使随后对它的读操作失败。

在分布式系统中，处理文件共享的第四种方法是使用原子事务处理，正如在第 3 章中详细讨论的那样。简而言之，为了存取一个文件或一组文件，进程首先执行某种类型的开始事务处理原语，以指示跟在其后的操作是不可分的。然后通过系统调用来读写一个或多个文件。当此工作完成后，执行结束事务处理原语。这个方法的关键特性是保证了包含于事务处理中的所有调用都将按顺序完成，而不能有其他任何同时存在的事务处理的干扰。如果两个或多个事务处理同时启动，那么系统保证最后的结果和它们在某一（未规定）顺序执行时是相同的。

事务处理使编程更加容易的典型的例子是在银行系统中。设想某一银行帐号有 100 美元，有两个进程都想往该帐号上加 50 美元。在非约束的系统中，每一个进程都可能同时读包含当前余额 (100) 的那个文件，各自计算新的余额 (150)，并成功地用这个新值写入文件。最后的结果可能是 150，也可能是 200，这取决于读和写时的精确计时。通过将所有这些操作组成一个事务处理，这种交错就不会出现，最后的结果总是 200。

表 5-1 给出了在分布式系统中处理共享文件的四种方法。

表 5-1 在分布式系统中处理共享文件的四种方法

方 法	说 明
UNIX 语义	对一个文件的任何操作对所有进程都是及时可见的
会话语义	在文件关闭之前，对文件的修改对其他进程是不可见的
不可更改文件	不能进行更改，只是简单的共享和复制
事务	所有的更改要么都完成，要么都不能完成

5.2 分布式文件系统的实现

前一节，我们从用户的角度讨论了分布式文件系统的各个方面，即它们是呈现给用户的。本节将讨论文件系统是如何实现的。首先，给出文件使用的一些经验信息；然后再讨论系统结构，高速缓存实现，分布式系统的复制以及并发控制；最后简单地讨论一下经验

教训。

5.2.1 文件的使用

在实现分布式或者任何其他系统之前，对如何使用它有一个好的想法并保证最常用的操作有效是非常有用的。为此，Satyanarayanan(1981)对文件使用方式进行了研究。我们将在下面给出他的主要结果。

然而，首先有必要提请读者注意的是关于这些结果和类似的测量。一些测量是静态的，即这些测量只表示系统某一时刻的瞬态。静态测量是通过检查硬盘(磁盘)内容来进行的。这些测量包括：文件的大小和类型的分布，以及各种类型和大小的文件所占存储空间的总量。另外一些测量是动态的，通过对文件系统的修改，将各种操作记录到一个日志文件中以用于以后的分析。这些数据将产生关于各种操作的相对频率、任何时刻文件打开次数和发生的共享数的信息。尽管静态和动态测量在根本上是不同的，但将它们结合起来使用，可以对文件系统的使用得到一个更好的描述。

测量任何现存系统总会产生的一个问题是要知道所观察的使用者具有怎样的典型性。Satyanarayanan 的测量是在一所大学里进行的。它们可否运用于工业研究实验室？办公自动化工程？银行业务处理系统？在这些系统未装备和测量之前，没有人能确切地知道。

在进行测量时，另一个固有的问题是要注意被测系统的情况 (artifacts)。看一个简单的例子：当看到一个 MS-DOS 系统中的文件名分配时，可以迅速得到这样的结论：文件名决不会超过 8 个字符(加一个可选的三个字符的扩展名)。但是，由于没有人曾使用过超过 8 个字符的文件名，因此得到 8 个字符就足够的结论可能是错误的。由于 MS-DOS 不允许文件名超过 8 个字符，所以就无法得知当用户不受这一限制时他们会如何去做。

最后，Satyanarayanan 的测量基本是在传统的 UNIX 系统上进行的。这些测量能否转移到或推广到分布式系统上还不能确切知道。

为此，最重要的结论列于表 5-2 中。根据观测资料，可得出一些结论。首先，大多数文件都小于 10K，这与 Mullender 和 Tanenbaum(1984)在不同环境下所得出的结果是一致的。这个结果表明，在服务器和客户之间传送整个文件而不是磁盘块是可行的。由于整个文件传送明显更简单且更有效，因此这个思想是值得考虑的。当然，有些文件很大，所以对它们还必须作一些规定。还有，一个好的指导方针对正常情况可以起到优化作用，而对非正常情况要特殊处理。

一个有意义的观测结果是：大多数文件的生命周期都不长。一般的模式是：建立文件，读取它(也许只一次)，然后删除它。编译程序就是一个典型的应用，它建立临时文件用于再运行期间传送信息。在这里暗指：在客户边建立文件并保存直到删除，可能是一个好主意。这样做可以消除客户和服务器之间大量的不必要的通信量。

文件很少共享的事实为客户高速缓存提供了依据。如我们已看到的那样，高速缓存使得语义更加复杂，但如果文件很少共享，那么最好进行客户高速缓存，并接受对话语义的结果作为更好运行效率的回报。

最后，由于明显存在截然不同的文件类型，建议采用不同的机制来处理不同的文件类型。系统二进制数普遍需要，但很难改变，所以它们也许会被广泛地复制，即使这意味着偶然的修改也是很复杂的。编译程序和其他临时文件很小，不可共享，而且消失很快，因此它们应该尽可能地保留在本地。电子邮件信箱经常更新但极少共享，所以复制也无益。普通的数据文件可以共享，因此它们仍需要其他的处理。

表 5-2 被测文件系统的特性

大多数文件比较小（小于 10K）
对文件的读操作多于对文件的写操作
对文件的读写是顺序进行的，随机访问非常少
大多数文件寿命比较短
文件共享非常少
一般进程只是使用某几个文件
用不同的属性区分文件类型

5.2.2 系统结构

在这一节，我们将讨论文件服务器和目录服务器内部的某些组织方式，要特别注意那些抉择方式。我们首先从一个简单的问题开始：客户和服务器不同吗？非常令人吃惊的是，对于这个问题没有一致的回答。

在有些系统中，客户和服务器之间没有什么差别。所有的机器都执行基本相同的软件，因此，任何机器都可自由地为公众提供文件服务。提供文件服务就是输出所选择的目录名，以便其他的机器可访问它。

在其他一些系统中，文件服务器和目录服务器仅是用户程序，因此，可根据需要配置一个系统以便在同一机器上或者不同机器上执行客户和服务器软件。最后，另一个极端是：系统根据硬件或者软件将客户和服务器分配在完全不同的机器上。服务器甚至可以执行来自客户的不同操作系统的版本。虽然功能分开看起来更简洁些，但并没有根本理由说明一种方式优于其他的方式。

系统不同的第二个实现问题是文件和目录服务是怎样构造的。一种结构是将两者合并成一个服务器来处理所有的目录和文件自身的调用，而另一种可能是保持它们独立。在后一种情况下，打开一个文件需要进入目录服务器，以便将它们的符号名变换成二进制名（例如，机器+i 节点），然后用二进制名进入文件服务器对文件进行读写。

赞成分开的理由是两个功能确实不相关，保持独立会更加灵活些。例如，我们可以实现 MS-DOS 目录服务器和 UNIX 目录服务器，两者都使用相同的文件处理机供物理存储。功能既可独立又能产生较简单的软件。反对这个主张的理由是：两个服务器要求更多的通信。

我们首先分析目录和文件服务器独立的情况。在正常情况下，客户发送一个符号名

给目录服务器，接着目录服务器就返回一个文件服务器能理解的二进制名。然而，目录分层在多个服务器之间分配是可能的（如图 5-5 所示），例如：假设有一个系统，系统中的当前目录在服务器 1 上，它包含在服务器 2 上的另一个目录的入口 *a*。同样，这个目录对另一个在服务器 3 上的目录包含一个入口 *b*。第三个目录包含文件 *c* 的入口和它的二进制名。

为了查找 *a/b/c*，客户发送一个消息给管理其当前目录的服务器 1。服务器 1 找到 *a*，并发现其二进制名指向另一个服务器。现在可以选择：或者它告诉客户哪个服务器保存 *b*，并让它自己去查找 *b/c*，如图 5-5(a)所示；或者它将请求的剩余部分发送给服务器 2，并不作回答，如图 5-5(b)所示。前一种方法要求客户知道哪个服务器保存着哪个目录和更多的消息。后一种方法更有效，但不可能用正常的 RPC（远程进程调用）来处理，这是因为接收客户所发送消息的进程不是发送应答的进程。

总是查找路径名，特别是涉及到多个目录服务器时，其代价是昂贵的。有些系统试用通过保存高速缓存提示来改善其性能，这些提示是最近查找的名字和这些查找的结果。当一个文件打开时，先检查高速缓存中是否有该路径名。如果有，那么就不用逐个目录去查找，从高速缓存中直接读取其二进制地址。如果没有，再进行目录查找。

为使用名字高速缓存，基本的作法是：当无意中使用一个已过时的文件时，我们应设法通知客户，以便它能通过逐一查找目录的方式找到文件并更新高速缓存。而且，为了使高速缓存提示（hint caching）最具有使用价值，必须保证提示在大部分时间里是正确的。当这些条件具备时，高速缓存提示将成为许多分布式操作系统都适用的一种强有力的技术。

我们将讨论的最后的结构问题是：文件、目录和其他的服务器是否应该保留客户的状态信息。这个问题在一定程度上还是有争论的，现在还存在两种派别思想的对抗。

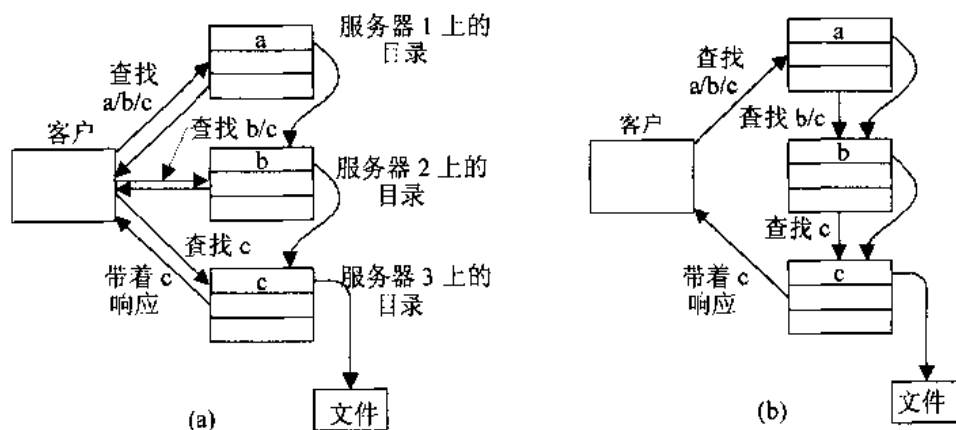


图 5-5 (a) 迭代查找 *a/b/c*
(b) 自动查找

一种派别认为：服务器应该是无状态的。换句话说，当客户发送一个请求给服务器，服务器完成请求，发送一个应答，然后从内部表中移出关于该请求的所有信息。在请求之

间，服务器不保存具体客户的信息。另一种派别则坚持：服务器应该保存两个请求之间的客户的状态信息。毕竟，集中式操作系统保存了关于活动进程的状态信息，为什么这一传统的作法突然变得不可接受了呢？

为了更好地理解这个差别，考虑到拥有打开、读、写和关闭文件通用命令的文件服务器。文件打开之后，服务器必须保存哪个客户打开了哪个文件的信息。典型地，当一个文件打开时，客户将获得一个文件描述符或其他用于后续调用的编号，以便识别这个文件。当有请求到来时，服务器就使用文件描述符来确定需要哪个文件。将文件描述符变换成文件本身的表就是状态信息。

对于不保留状态信息的服务器，每一个请求必须是独立的。为了使服务器能够工作，它必须包含全文件名和文件中的偏移量。此信息增加了消息的长度。

研究状态信息的另一种方法是：如果服务器坏了并且它的所有表都永久性丢失了，将会发生什么情况。当服务器重新启动时，它已不能再知道哪些客户打开了哪些文件。以后对已打开文件进行读与写操作就会失败，而且如果有可能的话，将完全由客户决定恢复。其结果是：不保留状态的服务器比保留状态的服务器有助于更好的容错。这是赞成前者的理由之一。

表 5-3 无状态和有状态服务器比较

无状态服务器的优点	有状态服务器的优点
容错	请求消息比较短
不需要 OPEN/CLOSE 调用	更好的性能
没有服务器表空间的浪费	可以预读
没有打开文件数目的限制	易于幂等性
客户崩溃时不会造成服务器错误	可以对文件加锁

表 5-3 中总结了两种方式的优点。正如我们已提到的那样，无状态服务器在本质上有更多的容错。不需要 OPEN 和 CLOSE 调用，这就减少了消息编号，特别对于那些整个文件用一次就可读出的普通情况，服务器不用浪费空间来存放表。使用表时，如果太多的客户一次打开太多的文件，则将表填满，就不能打开新的文件。最后对于状态服务器，如在文件打开时客户出了故障，服务器就会处于困境中。如果它对此束手无策，它的表最终将充满垃圾。如果它超时了还未打开文件，那么客户因两个请求之间等待时间太长将被拒绝服务，而校正程序就会失去校正功能。无状态服务就不存在这些问题。

有状态服务器也可以对这些问题进行处理。由于 READ 和 WRITE 消息并不是必须包含文件名，所以它可以更短些，这样就使用更小的网络带宽。由于关于打开文件(在 UNIX 项中，i-节点)的信息在文件关闭之前都可保存在主存储器中，所以有较好的性能。由于大多数文件都是按顺序读的，可以预先读信息块减少延迟。如果一个客户已用完时间片并且两次发送了同一请求，例如 APPEND，那么用状态信息(通过每一条消息的次序号)就可以很容易地检测到。在无状态操作的不可靠通信前，实现幂等性需要更多的努力。最后，在一个真正的无状态系统中，要使文件锁定是不可能的，因为建立锁定的唯一效果是使状态

进入系统。在无状态系统中，文件锁定必须通过一个特定的锁定服务器来完成。

5.2.3 高速缓存(caching, 超缓存)

在一个各自有主存和磁盘的客户-服务器系统中，有四个地方可用来存储文件或存储部分文件：服务器磁盘，服务器主存，客户磁盘(如果可用的话)或者客户主存，如图 5-6 所示。我们将看到，不同的存储位置具有不同的特性。

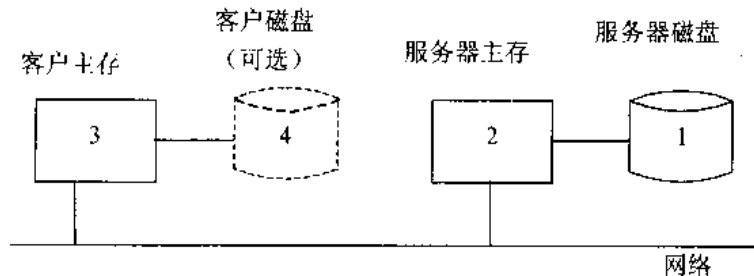


图 5-6 四个存储文件或部分文件的地方

存储所有文件最直接的地方是在服务器磁盘上。通常那里具有充足的空间，存放在那里的所有文件对所有客户都是可访问的。而且，由于每一个文件只有一个拷贝，所以不会产生一致性的问题。

使用服务器磁盘所产生的问题是运行效率低。在客户读一个文件之前，文件必须从服务器磁盘传送到服务器主存中，然后再通过网络传送到客户的主存中。两次传送都需要花费时间。

获取可观的运行效率的一种方法是使用高速缓存(即保存)，通过将最近使用过的文件保留在服务器的主存中来实现。客户读取一个刚好存放在服务器主存中的文件，就消除了磁盘传送，但网络传送仍然必须进行。由于主存总是比磁盘小得多，所以需要某些算法来确定哪些文件或文件的哪些部分应该保存在高速缓存中。

这个算法有两个问题需要解决。首先，什么是高速缓存管理单元？它可以是整个文件或磁盘块。如果整个文件需高速缓存，它应该连续存放在磁盘上(至少在一个很大块中)，以使得在存储器 and 磁盘之间进行高速传送，并获得较好的运行效率。但是，磁盘块超高速缓存比使用高速缓存和磁盘空间时更有效。

其次，这个算法必须决定当高速缓存填满，且必须去掉一些内容时该做什么。这里可以使用任何标准的高速缓存算法，但因为高速缓存访问比起主存访问来说更为少见，所以使用链接表的 LRU (最近最小使用) 一般是可行的。当必须去掉一些内容时，应选择最陈旧的内容。如果磁盘上存在最新的拷贝，则就要除去高速缓存拷贝。否则，首先要更新磁盘。

在服务器主存中设置高速缓存是很容易且对客户是完全透明的。因为服务器可以保持其主存和磁盘拷贝同步，从客户的观点来看，每个文件只有一个拷贝，所以不会产生一致性问题。

尽管服务器高速缓存消除了每一次访问时的磁盘传送，但它仍然需要网络访问。消除网络访问的唯一方法是在客户端进行高速缓存，这是产生所有问题的地方。使用客户主存还是其磁盘取决于对空间和效率的考虑。磁盘保存数据多但速度慢。当面对服务器主存的高速缓存与客户磁盘之间的选择时，前者通常更快些，而且总是简单得多。当然，如果使用大量的数据，客户磁盘高速缓存可能更好些。总之，大多数进行客户高速缓存的系统都是在客户主存中设置高速缓存，所以我们将集中讨论这种方式。

如果设计者决定在客户主存中设置高速缓存，有三种可使用的选择来精确定义高速缓存的位置。最简单的选择是在每个用户进程自己的地址空间直接进行文件高速缓存，如图 5-7(b)所示。典型地，高速缓存由系统调用库管理。当打开、关闭、读和写文件时，该库只是保存最常用的文件，这样当重新使用文件时，它已经是可用的了。当该进程退出时，所有被修改过的文件都回写到服务器中。尽管这种模式的系统开销很小，但它仅当单个进程重复地打开和关闭文件时才有效率。一个数据库管理程序进程可能满足这种要求，但是在通常的程序开发环境中，大多数进程对每个文件只读一次，因此库中的高速缓存是无益可获的。

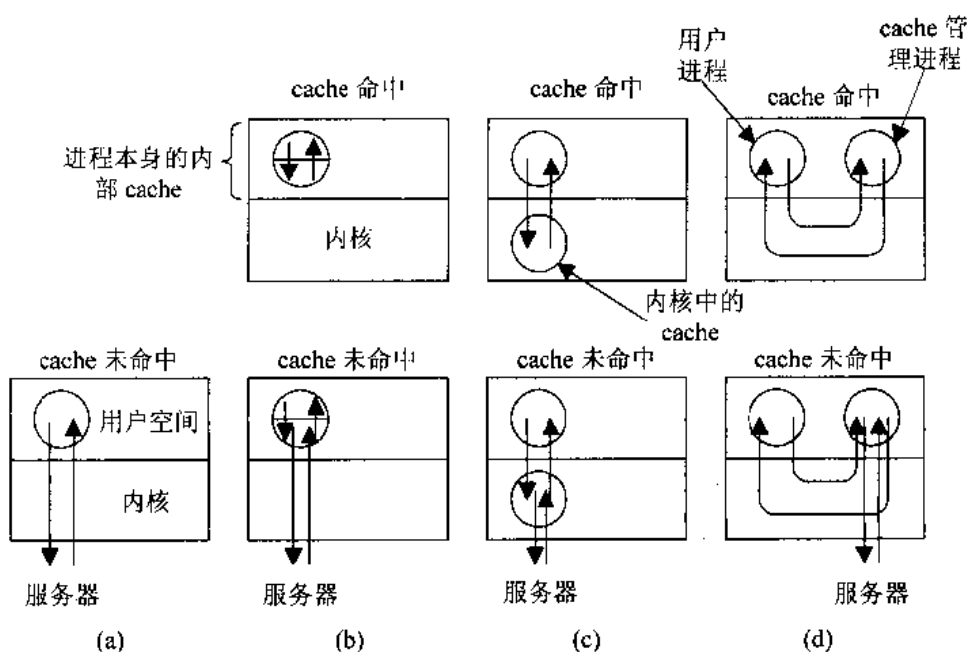


图 5-7 在客户存储器中进行高速缓存的不同方法

- (a) 没有高速缓存
- (b) 每个进程都有高速缓存
- (c) 内核中有高速缓存
- (d) 高速缓存管理作为一个用户进程

第二个设置高速缓存的地方是在内核中，如图 5-7(c)所示。这里的缺点是在所有情况下都需要内核调用，甚至对于高速缓存命中，但事实是，高速缓存使进程获益比补偿多。例如，假设一个两遍扫描编译器作为两个进程运行。第一遍扫描写一个中间文件而由第二

遍扫描来读。在图 5-7(c) 中，当第一遍扫描进程结束后，该中间文件可能在高速缓存中，所以在第二遍扫描进程读入时没有必要进行服务器调用。

设置高速缓存的第三个地方是在一个单独的用户级高速缓存管理者进程中，如图 5-7(d)所示。用户级高速缓存管理者的优点是：它保持了（微）内核独立于文件系统编码，因为它是完全孤立的，因此易于编程，并且更加灵活。

另一方面，当内核管理高速缓存时，它就能够动态地决定多少内存用于保存程序，多少内存用于高速缓存。用户级高速缓存管理程序运行在一个具有虚拟内存的机器上时，可以确信的是由内核来决定把高速缓存的部分或全部内容分页写入磁盘，所谓的“高速缓存命中”就需要换入一页或几页。显然，这就完全挫败了客户高速缓存的思想，但是，如果高速缓存器管理程序可以在内存中分配和锁定一些页码号，这个具有讽刺意义的情形就可避免。

评价高速缓存的价值时，重要的是注意在图 5-7(a)中，它无论如何都要严密地通过一个 RPC 来产生一个文件请求。在图 5-7(c)和图 5-7(d)中，它既可取一个也可取两个，这取决于在没有高速缓存器时，这个请求是否能满足。因此，当使用高速缓存时，RPC 的平均数总是要大一些。在 RPC 快而网络传送慢（快 CPU，慢网络）的情况下，高速缓存可以获得可观的运行效率。但是，如果网络传送特别快（例如，用超高速光纤网络），网络传送时间是很小的，因此额外的 RPC 可能损失大部分提高的运行效率。这样，由高速缓存所获得的运行效率的提高在某种程度上取决于所可用的 CPU 和网络技术，当然，这也取决于应用。

高速缓存一致性

在计算机科学中很普遍的规律是，得到任何东西都是有代价的。客户高速缓存为系统带来了不一致性。如果两个客户同时读一相同文件，然后又都去修改它，就会出现几个问题。一是，当第三个进程从服务器读该文件时，它将得到原来的值，而不是两个新文件中的一个。此问题可用对话语义来定义它（正规的陈述是对文件的修改结果在文件关闭之前是不允许全程可见的）。换句话说，这种“不正确”的行为可简单地声明为“正确”的行为。当然，如果用户期待的是 UNIX 语义，这种把戏也就无意义了。

不幸的是，另一个问题是当两个文件已回写到服务器时，后回写的会覆盖掉前面的那个。这意味着客户高速缓存必须相当小心地设计。下面我们将讨论这些问题的解决方案。

解决一致性问题的一个方法是使用直接写算法（**WRITE-THROUGH 算法**）。当修改一个高速缓存项（文件或块）时，新的值保存在高速缓存中，并立即回写到服务器。结果是，当另一个进程读该文件时，它将得到最近的值。

但是，出现了下面的问题：假设在机器 A 上一个的客户进程读文件 f 。客户结束但该机器在其高速缓存中保存 f 。随后，机器 B 上的一个客户读同一个文件，修改它，并将它写到服务器上。最后，机器 A 上的一个新的客户进程开始启动。它做的第一件事就是打开并读入 f ，而 f 是从该机器的高速缓存得到的。不幸的是，该值已经过时了。

一种摆脱此困境的方法是要求高速缓存管理者在从高速缓存向任何客户提供文件

时，须先与服务器进行核对。核对可通过将高速缓存中版本的最后修改时间和服务器中版本的最后修改时间进行比较。如果它们是一样的，那么高速缓存中内容是最新的。否则，当前的版本必须从服务器中提取。除了使用日期外，也可以使用版本号或校验和。尽管与服务器核对日期，版本号或校验和需要使用一个 RPC，交换的数据量还是很小的。当然，这也需花一些时间。

WRITE-THROUGH 算法的另一问题是：尽管它对读是有益的，但对写来说，网络传输是相同的，就像根本没有高速缓存一样。许多系统设计者发现这种算法是不可接受的，而且带有欺骗性。替代在写操作完成的瞬间去访问服务器，客户只是记录一下该文件已被修改。大约每隔 30 秒，所有更新文件聚集在一起一次向服务器发送。一个单独的大量的写操作通常比许多小的写操作更有效。

除此之外，许多程序还需创建擦除文件，写入它们，读回它们，然后删除它们，所有操作都紧连着进行。如果这整个过程发生在将所有被修改过的文件回写到服务器之前，则正在被删除的文件就没有必要回写到服务器中。临时文件不使用文件服务器是获得运行效率的一个主要方法。

当然，延迟写操作使得语义变得不清楚，因为当另一个进程读此文件时，它所得结果取决于时间选择。这样，延迟写只好在运行效率和清晰的语义（它可使编程更为容易）之间的权衡。

沿这一方向的下一步是使用对话语义，且仅当文件关闭后才将文件回写到服务器。这种算法称为关闭时写（WRITE-ON-CLOSE）。更好的是，再等 30 秒看是否要删除文件。如我们前面讲到的，走这条路意味着两个已放入高速缓存的文件被依次回写，第二个会覆盖第一个。解决此问题的唯一方法是注意它是否比第一次出现时已好得多了。在单 CPU 系统中，两个进程可以在各自的地址空间打开和读文件，并修改它，然后将它回写。因此，使用对话语义的 WRITE-ON-CLOSE 并不比单 CPU 系统糟糕。

一种解决一致性问题的完全不同的方法是集中的控制算法。当打开一个文件时，打开该文件的机器向服务器发送一条消息来宣告这一事实。服务器保存谁打开了哪个文件以及打开是为了读，还是写或二者皆有。如果文件是为读打开，其他进程也为读而打开它就不会有问题，但必须避免为写而打开它。同样地，如果某个进程为写而打开一个文件，就必须禁止所有其他的访问。当关闭文件时，必须报告这一事实，这样服务器就可以更新哪个客户打开了哪个文件的表。这时，被修改的文件就可回写到服务器上了。

若一个客户试图打开一个文件，而该文件已在系统其他地方打开了，则新的请求或者被拒绝或者要排队。作为选择，服务器可向所有有打开文件的客户发送一个自发消息（unsolicited message），告诉它们应将该文件从它们的高速缓存中移去，并对那个文件的高速缓存无效。在这种方式下，多个读者和写者可同时进行，其结果和在单 CPU 系统中情况不相上下。

尽管发送自发消息是肯定可行的，但却非常生硬。因为它颠倒了客户和服务器的角色。通常，服务器不会自发地给客户发送消息，或用它们初始化 RPC。如果客户是多线程的，可永久地分配一个线程来等待服务器的请求。否则，自发消息肯定会产生中断。

即使有了这些防御措施，也必须小心。特别是，如果一台机器打开高速缓存、关闭一个文件，当再次打开该文件时，高速缓存管理程序必须检查其高速缓存是否有效。毕竟是因为某些其他进程有可能随后已打开、修改、并关闭了该文件。这时使用不同的语义，许多集中式控制算法的变体都是可行的。例如，服务器可以跟踪已被高速缓存的文件，而不是跟踪被打开的文件。所有这些方法都存在单点失败并且没有一种方法可以满足大系统的要求。

表 5-4 管理客户文件高速缓存的四种算法

方 法	描 述
直接写	有效，但不影响写流量
延迟写	效率较高，但可能语义不清
关闭时写	与会话语义相配
集中控制	UNIX 语义，但不健壮，不能规模化

表 5-4 总结了上面讨论过的四种高速缓存管理算法。下面从总体上总结一下高速缓存这个主题。不管客户高速缓存是否存在，服务器高速缓存容易实现且大多数是值得去做的。从客户的角度来看，服务器高速缓存对文件系统的语义没有影响。相反，客户高速缓存以增加复杂性和可能更为模糊的语义为代价来提供更好的性能。它的价值取决于设计者如何看待运行性能、复杂性，以及编程的难易。

在本章的前面，当我们讨论分布式系统的语义时，曾指出一种设计是选择不可变文件。不可变文件的一个最具吸引力的地方是机器 A 对它进行高速缓存时不用担心机器 B 会改变它，改变也是不允许的。当然，有可能创建了一个新的文件，并且取和高速缓存文件相同的符号名，但是一旦高速缓存文件被重新打开时，就可以进行核对。这种模型与前述 RPC 方法花费同样的开销，但是，语义要清楚一些。

5.2.4 复制

分布式系统通常提供文件复制作为对客户的一种服务。换句话说，系统保持所选文件的多个拷贝，每个拷贝存放在一台单独的文件服务器上。提供这样一种服务的原因是多样的，但主要的原因有：

- (1) 通过对每个文件的独立备份来增加系统的可靠性。即使一个服务器出现问题，或永久性地损坏，数据也不会丢失。对许多应用系统来说，这一点是非常重要的。
- (2) 当一个文件服务器出现问题时，仍允许进行文件访问。这里的座右铭是：演出必须继续进行。在重新启动系统之前，一个服务器的崩溃不应导致整个系统的瘫痪。
- (3) 将工作量分配到多个服务器上。随着系统规模的增大，将所有文件放在一个服务器上会造成运行性能上的瓶颈。通过将文件复制到两个或多个服务器上，可使用负载较轻的服务器。

前面两条与改善可靠性和有效性有关；第三条关系到系统性能，这些都很重要。

有关复制的一个主要问题是透明性（通常）。用户了解文件复制到何种程度？他们在复制进程中起作用，还是由系统完全自动处理的呢？一个极端，用户完全了解复制进程，甚至能够控制它。另一个极端，系统在他们背后做一切事情。在后一种情况下，我们称系统是复制透明的。

图 5-8 说明了实现复制的三种方法。第一种方法是显式复制（如图 5-8 (a) 所示），是为编程人员控制整个进程所用。当一个进程产生一个文件时，它只在一台指定的服务器上进行。然后，如果需要的话，它可以在其他的服务器上生成另外的拷贝。如果目录服务器允许一个文件有多个拷贝，则所有拷贝的网络地址都可以和这个文件名联系起来（如图 5-8 (a) 的下方所示），因此当用该文件名进行查找时，能找到它所有的拷贝。当该文件随后被打开时，它将以某种顺序对其拷贝依次进行尝试，直到找到一个可用的为止。

为了更熟悉显式复制的概念，考虑一个在 UNIX 中如何实现基于远程安装的系统。假设一个编程人员的根目录是 *machine1/usr/ast*。创建一个文件后，比如文件 */machine1/usr/ast/xyz*，编程人员、进程或库都可使用命令 *cp*（或等价者）来产生拷贝 */machine2/usr/ast/xyz* 和 */machine3/usr/ast/xyz*。命令 *cp* 可以使程序接受像 */usr/ast/xyz* 的字符串作为参数，并依次去打开拷贝，直到成功。虽然这种模式可行，但非常麻烦。由于这个原因，分布式系统可能会更好一些。

在图 5-8(b)中，我们可以看到另一种可选择的方法，懒惰复制（*lazy replication*）。只要在某个服务器上建立每个文件的一个拷贝，在编程人员后来不知道的情况下，服务器自己在其他的服务器上也可自动地生成副本。系统必须足够聪明以便能够从这些拷贝中找到任何一个所需的拷贝。在这种背景下产生拷贝时，我们必须注意到这种可能性：在产生拷贝之前，文件有可能已经改变。

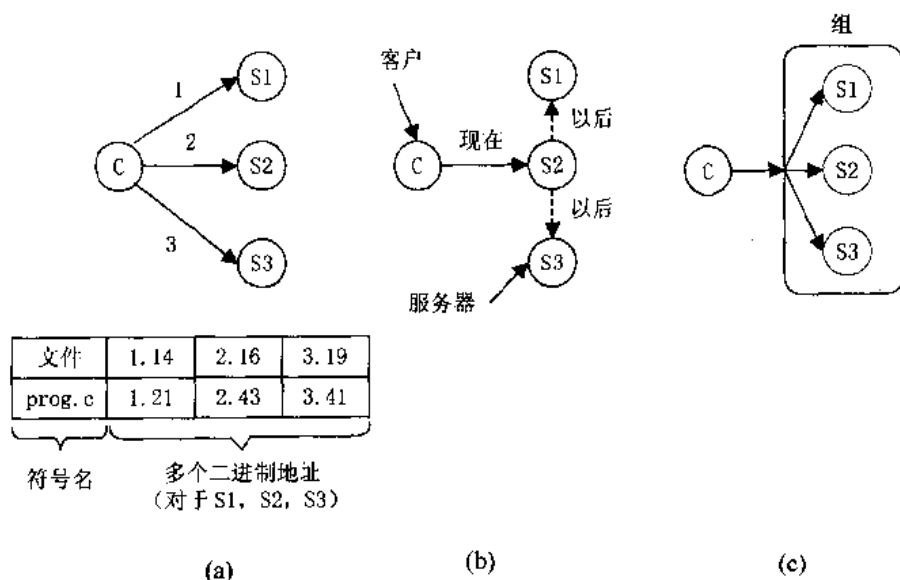


图 5-8 (a) 显式文件复制
(b) 懒惰复制
(c) 用组复制文件

最后一个方法是使用组通信（如图 5-8 (c) 所示）。在此方案中，所有的写系统调用都同时传送到所有的服务器，这样，其他的拷贝在原文件产生的时候就产生了。懒惰复制和使用组复制有两个主要区别。首先，使用懒惰复制时，只有一个服务器被寻址而不是一个组。其次，懒惰复制是当服务器有一些空闲时，在后台进行，而使用组通信时，所有的拷贝都同时产生。

更新协议

上面我们讨论了复制文件是怎样创建的。现在让我们来看看怎样修改已存在的文件。仅仅按顺序向每个拷贝发送一个更新消息并不是一个好主意，因为进行更新的进程如果半途而废，这样某些拷贝被改变了，而其他的则没被改变。其结果是，某些随后的读操作可能一些得到的是原有值，而另一些得到的是新值，这肯定不是希望出现的情形。现在来看一下解决这个问题两个著名的算法。

第一个算法称为主拷贝复制（**primary copy replication**）。使用时，指定一个服务器为主服务器，其他所有的服务器则为从服务器。当要更新一个复制文件时，我们就将该改变送至主服务器，在本地完成修改，然后向各从服务器发出命令，命令它们也完成修改。这样，可在任何一个（主或从）拷贝中进行读操作。

为了防止主服务器在向从服务器发出命令之前崩溃，在改变主拷贝之前，应将更新写在稳定存储的介质上。周期改变其主拷贝，当一个服务器在崩溃后重新启动时用这种方法，将进行检查，看崩溃时是否有任何更新正在进行。如果有，它们将继续进行。所有的从服务器迟早都将被更新。

虽然这种方法简单，但它也有缺点：当主服务器停机时，所有的更新将不能进行。为避免这种不对称性，Gifford(1979)推荐了一种更健全的方法，即表决（**voting**）。其基本思想是在读或写一个复制文件之前要求申请并获得多个服务器的允许。

下面举一简单的例子来说明该算法是如何工作的。假设一个文件在 N 个服务器上都有复制。建立一个更新文件的规则：首先客户必须和超过半数的服务器联系，并让它们同意它进行更新。如果它们同意，就改变文件，并将一个新的版本号和新文件联系起来。该版本号用来标识该文件的版本，这对所有新近更新的文件都是一样的。

当读一个已有复制的文件时，客户必须和超过半数的服务器联系，并请求它们发送与该文件相联系的版本号。如果所有版本号相一致，则说明这是最新的版本，如果企图只更新剩余服务器，会因为数目不足而失败。

例如，有 5 个服务器，客户已确定它们中的三个有版本号 8，则其他两个的版本号不可能是 9。因为任何从版本号 8 到版本号 9 的成功更新需要 3 个服务器同意，而不是两个。

Gifford 的方案实际上比这更普通一些。在该方案中，读一个已有 N 个复制存在的文件时，客户需要获得一个读法定数（**read quorum**），它是任何 N_r 个或更多服务器的任一集合。同样地，修改一个文件需要一个至少 N_w 个服务器的写法定数（**write quorum**）。 N_r 和 N_w 的值必须满足约束条件 $N_r + N_w > N$ 。只有在适当数目的服务器同意参与时，文件才能读或写文件。

为弄清该算法是如何工作的，请看图 5-9 (a)。 $N_r=3$ ， $N_w=10$ 。假设最近的写法定数

由从 C 到 L 的 10 个服务器组成，所有这些服务器得到了新版本和新版本号，任何随后的由 3 个服务器组成的读法定数中将至少包含一个该集合中的成员。当客户查看版本号时，它将知道哪一个是最新的并得到它。

图 5-9(b)和图 5-9(c)给出了另外两个例子。后者特别有趣，因为它把 N_r 设成了 1。这样，它可以通过查找任何一个拷贝并使用它来读一个已复制的文件。但是，其代价是写更新需要获得所有的拷贝。

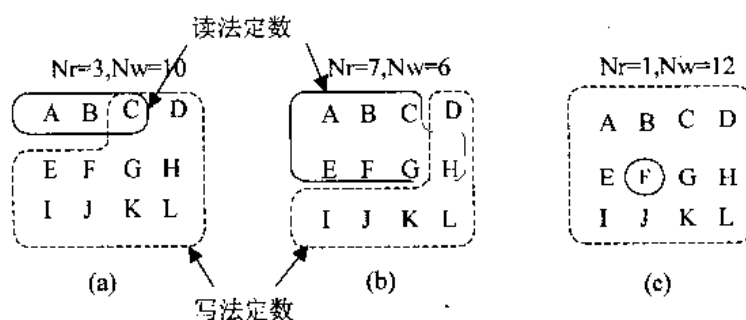


图 5-9 表决算法三个例子

表决法的一个有趣的变种是虚像表决 (**Voting with ghosts**) (Van Renesse and Tanenbaum, 1988)。在许多应用中，读比写平常得多，因此， N_r 通常是非常小的数字，而 N_w 几乎接近 N 。这意味着如果少数几个服务器崩溃时，要获得写法定数是不可能的。

虚像表决通过为每个已崩溃的服务器建立一个没有存储器的虚拟服务器解决了这个问题。虚设者不允许出现在读法定数中（毕竟它没有任何文件），但它可以加入写法定数中，在这种情况下，它只需去掉写入的文件即可。只要有一个真实的服务器存在，写操作就能成功。

当一个崩溃的服务器重新启动时，它必须获得一个读法定数来找到最新的版本。在它开始正常工作之前，它将为自己拷贝一份该拷贝。由于该算法和基本的表决算法具有相同的特性，因此，它是可行的。也就是说，所选择的 N_r 和 N_w 使得它不可能同时获得读法定数和写法定数。这里唯一的区别是允许死的机器加入写法定数，但要满足一旦死机恢复，它们在进行服务之前应立即获得当前的版本的条件。

其他的复制算法可以参见 Bernstein 和 Goodman, 1984; Brereton, 1986; Pu 等, 1986; Purdin 等, 1987。

5.2.5 例子：SUN 公司的网络文件系统

本节我们看一个实例：Sun Microsystem 公司的网络文件系统(**Network File System**)，通常称为 **NFS**。NFS 最初由 Sun Microsystem 公司设计并实现，它基于 UNIX 的工作站上。UNIX 和其他操作系统（包括 MS-DOS）的其他制造商也支持它。NFS 支持不同种类的系统，例如，MS-DOS 客户可以使用 UNIX 服务器。它甚至也不要求所有的机器使用相同的硬件。我们可以经常发现运行在 Intel 386 CPU 上的 MS-DOS 客户可以得到运行于 Motorola

68030 或 Sun SPARC CPU 上的文件服务器的服务。

NFS 有三个方面吸引人：结构、协议和实现。下面将依次讨论。

NFS 结构

NFS 背后的基本思想是让客户和服务器的任一集合共享一个公用的文件系统。大多数情况下，所有的客户和服务器的在同一个局域网内，但这并不是必须要求的。NFS 也可以在广域网上运行。为了简单起见，我们将所谈及到的客户和服务器的当作它们好像在不同的机器上一样，实际上，NFS 允许每一台机器同时既是客户又是服务器。

每个 NFS 服务器可向远程访问的客户输出它的一个或多个目录。当有一个目录可访问时，所有它的子目录都可访问，因此，实际上，整个目录树通常作为一个单元而输出。服务器输出的目录列表被保存在文件 */etc/exports* 中，这样无论服务器何时重新启动，将自动输出这些目录。

客户通过安装方式访问输出的目录。当客户安装一个（远程）目录时，它就成为该目录层次结构的一部分 5-10。许多 SUN 的工作站是无盘的。如果需要的话，无盘的客户可以登到远程文件系统的根目录下，这样将导致整个文件系统完全由远程的文件服务器来支持。那些有盘工作站从本地目录结构的顶部的任何需要的地方安装远程目录，这样，文件系统一部分在本地，一部分在远程。对运行在客户上的程序来说，位于远程文件服务器上的文件和位于本地磁盘上的文件几乎没有什么区别。

这样，NFS 的基本的机构特征是服务器输出目录，客户从远程安装它们。如果两个或多个客户同时安装同一个目录，它们可以通过共享它们公用目录中的文件来进行通信。一个客户上的程序可以创建一个文件，而另一个客户上的程序可以读该文件。一旦安装工作完成，就可实现共享。共享文件在多台机器的目录结构中，可用通常的方式读写它们。这一简单特性是 NFS 最具有吸引力的地方之一。

NFS 协议

由于 NFS 的目标之一是要支持异构系统，其客户和服务器的可能在不同的硬件平台上运行不同的操作系统，因此，必须定义好客户和服务器的之间的接口。只有这样，对任何人来说，才可能写出一个新的客户实现程序，并期望已有的服务器正常工作，反之亦然。NFS 通过定义两个客户-服务器协议来实现这一目标。

NFS 通过定义客户-服务器两个协议实现 NFS 的目录，一个协议是客户向服务器发出的一组请求，该请求带有服务器返回给客户的相应的应答。（协议是分布式系统中一个重要主题，后面我们将返回来更详细的讨论它。）只要服务器能够识别和处理协议中的所有要求，它没有必要了解它的客户的任何情况。同样地，客户也可以只将服务器当作可以接受和处理一组特殊请求的“黑盒子”。它们怎么做是它们自己的事。

第一个 NFS 协议处理安装。客户可以向服务器发送一个路径名并请求安装其目录结构中某个地方的目录。要安装的地方并不包含在消息中，因为服务器并不关心在哪儿安装。如果文件名合法，并且所指出的目录已经输出，服务器将向客户返回一个文件句柄（**file handle**）。文件句柄包含唯一标识文件系统类型、磁盘、目录的 i-节点号和安全信息的字段。以后在已安装的目录中对文件的读写调用将使用文件句柄。

许多客户都可以使用无须人工干预便可安装到指定的远程目录中的配置。通常情况下，这些客户必需包含一个远程安装命令的外壳程序的文件 `/etc/rc`。这个外壳程序在客户启动时将自动执行。

另外，SUN 的 UNIX 版本也支持自动安装 (automounting)。这一特性使得一组远程目录和一个本地目录联系起来。当客户启动时并没有安装这些远程目录（和它们的服务器也没有联系）中的任何一个。相反，打开一个远程文件时，操作系统向每一个服务器发送一消息，第一个回答的服务器的目录就被安装。自动安装与通过 `/etc/rc` 文件的静态安装相比较有两个主要优点。首先，如果一个在 `/etc/rc` 中指明的服务器崩溃，客户就不可能启动，至少会有一些困难（延迟和一些错误消息）。如果在那时用不着那个服务器，那么所有的工作都白费了。其次，允许客户去并行尝试一组服务器，可以实现某种程度上的容错（因为只需它们中的一个可用就可以了），而且可以提高性能（通过选择最先应答的服务器——有可能是负载最轻的那个）。

另一方面，默认指定作为自动安装文件系统的替代者是唯一的。由于 NFS 没有提供文件或目录的复制，所以由用户来安排，使得所有文件系统是一样的。结果，自动安装最常用于包含系统二进制文件和其他很少改变的文件的只读文件系统。

第二个协议是关于目录和文件访问。客户可向服务器发送一条消息来操纵目录和读写文件。此外，它们也可以访问文件的属性，如文件类型、大小、最后一次修改的时间。NFS 支持大多数的 UNIX 调用，可能让人意外的是不包括 OPEN 和 CLOSE。

省略 OPEN 和 CLOSE 并不是一个意外，这完全是有意。在读一个文件之前没有必要打开它，操作完成后也没有必要关闭它。相反，在读文件时，客户向服务器发送一个包含文件名的消息并要求查找该文件，返回一个文件句柄，它是标识该文件的一个结构。不像 OPEN 调用，这个查找操作并不向内部系统表拷贝任何信息。读调用包含要读的文件句柄，开始读的位置在文件中的偏移量和所需的字节数。每个这样的消息都是自含的。这种模式的优点是服务器不用记住对它的调用之间的打开连接。这样，如果服务器崩溃，然后又重启时，就不会丢失任何打开文件的信息，因为服务器上根本就没有。像这样的不用保留打开文件的状态信息的服务器被称为是无状态的。

相反，在 UNIX 系统 V 中，远程文件系统 (RFS) 则要求在读写文件之前必须先打开该文件。然后，服务器建立一个表的项目用来记录该文件已被打开和当前读者的位置，这样，每个请求中就不需要包含偏移量了。这种模式的缺点是，如果服务器崩溃，然后又迅速重新启动，所有的打开连接就都会丢失，并且客户程序会失败。而 NFS 没有这样的特性。

不幸的是，用 NFS 方式实现精确的 UNIX 文件语义是有困难的。例如，在 UNIX 中，可以打开并锁定一个文件，这样其他进程就不能访问它。当文件关闭时，锁就释放了。在无状态的服务器中（如 NFS 中），打开文件不能与加锁联系起来，因为服务器并不知道打开了哪些文件。因此，NFS 需要一个单独的、另外的机制来处理加锁。

NFS 使用 UNIX 保护机制，对所有者、组和其他使用 `rwX` 位。最初，每个请求消息只包含调用者的用户和组的标识符，NFS 用它来对访问进行有效性检查。实际上，它相信

客户不会欺骗它。多年的经验已经充分表明这样的假设是天真的，我们应该怎么办呢？现在，可以使用公共键密码技术来建立一个保密键以便对每一次请求和应答的客户和服务器的有效性检查。当这种方法起作用时，一个带恶意的客户就不能模仿另一个客户，因为它不知道那个客户的保密键。随便说一下，键只是用来鉴别参与者的真伪，而数据本身并没有加密。

所有用于鉴别真伪的键和其他信息一样由网络信息服务 **NIS (Network Information Service)** 保存。NIS 以前被称为黄页。它的作用是用来存放（键，值）对。当给它提供一个键时，它返回相应的值。它不仅处理加密键，而且存储用户名和（已加密的）口令之间的映射，还有机器名和网络地址之间的映射和其他项。

人们使用主/从排列来对网络信息服务器进行复制。读取数据时，进程可以使用主拷贝或任何一个从拷贝。所有的修改只能由主服务器进行，然后由主服务器将修改传送到从服务器。每一次更新后将有一个短暂的间隔，在这个间隔中数据库的数据是不一致的。

NFS 实现

尽管客户和服务器的实现是独立于 NFS 协议的，但是，浏览一下 SUN 公司的实现也是很有意思的：它包括三层，如图 5-10 所示。顶层是系统调用层。它处理像 **OPEN**、**READ** 和 **CLOSE** 这样的调用。在对调用进行了语法分析并检验了参数之后，就调用第二层——虚拟文件系统（VFS）层。

VFS 层的任务是维持一张表，表中每一项对应一个打开文件的表，类似于 UNIX 中为每个打开文件设置的 **i** 节点表。在普通 UNIX 中，用（设备，**i** 节点号）对来唯一标识一个 **i** 节点。相应地，VFS 层对每一个打开文件有一个表项目，称为 **v-节点（虚拟 i 节点）**。**v-节点**用来说明文件是本地的还是远程的。对于远程文件，只有提供足够的信息才能访问它们。

为了了解 **v-节点**是如何使用的，我们来跟踪一下 **MOUNT**、**OPEN** 和 **READ** 序列的系统调用。为了安装一个远程文件系统，系统管理员调用安装程序来详细说明远程目录，要安装的本地目录和其他信息。远程安装程序对要安装的远程目录进行语法分析，并找到存放该目录的机器名。然后和该机器联系并向它索取一个该目录的文件句柄。如果该目录存在并且可用于远程安装，服务器将返回一个该目录的文件句柄。最后，它产生一个 **MOUNT** 系统调用，将处理交给内核。

然后，内核为该远程目录构造一个 **v-节点**并要求在图 5-10 NFS 客户代码在其内部表中创建一个 **r 节点（远程 i-节点）**来存放该文件句柄。**v-节点**指向 **r 节点**。VFS 层的每一个 **v-节点**最终将包含一个指针或者指向 VFS 客户代码中的一个 **r 节点**，或者指向本地操作系统的一个 **i 节点**（见图 5-10）。这样，通过 **v-节点**就可以知道一个文件或目录是本地的还是远程的。如果是远程的，就可以找到它的文件句柄。

当打开一个远程文件，在对路径名进行语法分析的某个时刻，内核找到安装到远程文件系统的那个目录。它会发现这个目录是远程的并且在该目录的 **v-节点**中找到指向 **r 节点**的指针，然后，它请求客户代码打开该文件。NFS 客户代码在与已安装的目录相连接的服务器上查找路径名的剩余部分，并为它取回一个文件句柄。它在其表中为远程文件建立一个 **r 节点**并返回给 VFS 层，VFS 层在其表中为该文件插入一个指向 **r 节点**的 **v-节点**。

这里我们又看到每一个打开的文件或目录都有一个指向 r 节点或 i 节点的 v-节点。

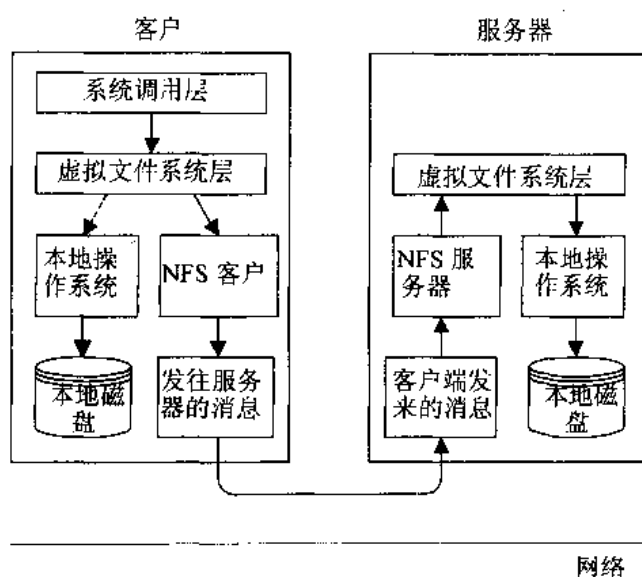


图 5-10 NFS 层结构

给调用者提供一个该远程文件的文件描述符。该描述符由 VFS 层的表映射到 v-节点上。注意，这在服务器方不产生任何表项。尽管服务器准备根据请求来提供文件句柄，但它从不记录哪些文件有未提供完全的文件句柄，哪些没有。当有一个用于文件访问的文件句柄发送给它时，服务器对它进行检查，如果该文件句柄有效，便使用它。如果安全性可用的话，有效性就包括证实一个包含在 RPC 头中的认证。

当该文件描述符在随后的系统调用中使用（例如，READ），VFS 层便定位相应的 v-节点，并通过它确定该文件是本地的还是远程的，以及哪个 i 节点或 r 节点描述它。

由于效率的原因，客户和服务端之间的传输以大的块进行，即使需要很少的字节通常也是 8192 个字节。当客户 VFS 层取得了所需的 8K 的块后，它自动发出传送下一个块的请求，这样它将得到它可能很快就需要的内容。这种特性称为超前读，它大大地提高了运行效率。

对于写操作，遵循的策略是类似的。如果一个 WRITE 系统调用提供的数据少于 8192 个字节，这些数据就仅在本地积累，只有当整个 8K 的块满了才发送到服务器。但是，当文件关闭时，它的所有数据必须立即发送到服务器。

提高性能的另一技术是使用高速缓存，这和普通的 UNIX 中一样。服务器使用高速缓存来避免磁盘访问，但这对客户是不可见的。客户维持两个高速缓存，一个用于文件属性（i 节点），一个用于文件数据。当需要一个 i 节点或文件块时，可进行检查，看能否用高速缓存来满足。如果可以，就可避免网络传输。

虽然客户高速缓存可以大大地提高性能，但它也带来一些令人讨厌的问题。假设两个客户对同一个文件块进行了高速缓存，并且其中一个对它进行了修改。当另外一个客户读该块时，得到的是原有值，高速缓存不能保持一致。在早期的多处理机中也存在同样的

问题。这一问题可通过在总线上设置高速缓存探视来对所有的写操作进行检查，并相应地对高速缓存项进行废除或更新。由于导致客户高速缓存命中的文件写并不产生任何网络传输，因此使用文件高速缓存是不可能的。即使可能，使用当前的硬件也几乎不能在网络上进行探视。

由于这一问题的潜在的严重性，NFS 实现采取了几个措施来减轻了其危险。第一，设置一个与每个高速缓存块相联系的计时器。当计时器到期，该项就被抛弃。通常，计时器对数据块是 3 秒，对目录块是 30 秒。这样做将减少一点危险。另外，当打开高速缓存文件时，计时器发送一条消息给服务器以便查出文件的最后修改时间。如果最后修改时间在本地拷贝被高速缓存之后，则抛弃该高速缓存拷贝并从服务器中提取一份新的拷贝。最后，一旦 30 秒的计时器到期，高速缓存中所有修改过的块都发送到服务器。

NFS 由于没有实现合适的 UNIX 语义仍受到广泛的批评。当一个客户读文件时，另一个客户上的对文件的写操作可能是可见的，也可能是不可见的，这依赖于计时。而且，当一个文件被创建时，有可能在 30 秒之内它对外是不可见的。类似的问题也存在。

通过这个例子我们可以看到，尽管 NFS 提供了一个共享的文件系统，但因为最后的系统是 UNIX 的一种修补，文件访问的语义并没有得到完全很好的定义，并且如果时间不同，重新运行一组相互协作的程序可能得到不同的结果。而且，NFS 所处理的主要问题是文件系统。其他的问题，如进程执行根本就没有涉及到。但不管怎么样，NFS 是很受欢迎的并得到了广泛的应用。

5.2.6 学到的教训

Satyanarayanan(1990b)根据他从事各种分布式文件系统的经验，提出了一些他认为分布式文件系统设计者应遵循的一般准则。我们在表 5-5 总结了这些准则。第一条准则是，工作站应有足够的 CPU 能力，在任何可能的地方使用工作站是很明智的。特别是，要选择是在工作站上还是在服务器上做事情时，应选择工作站。因为服务器的周期是很珍贵的，而工作站周期则不是很珍贵。

第二个准则是使用高速缓存。它们经常可节约大量的计算时间和网络带宽。

第三个准则是利用使用特性。例如，在一个典型的 UNIX 系统中，大约有三分之一是对临时文件进行访问的。它们的生命期短，且从来不被共享。对这些文件进行特殊处理有可能获得可观的性能。公平地说，还有另一学派的一种说法：“选择一个单一的机制并坚持它。不要用五种方法做同一件事情。”每个人持哪种观点取决于他是否追求效率或简单性。

表 5-5 分布式文件系统设计原理

工作站可以循环分担作业
如果可能就使用高速缓存
利用使用特性
降低系统的广度和变化
信任最少的可能实体
在可能的地方运用批处理

最小化系统范围的知识 and 变化对制定系统规模是很重要的。层次式设计在这方面有帮助。

相信最少的可能实体是一条在安全世界中早已建立的准则。如果系统的正确运行依赖于 10000 台工作站都去做指定给它们去做的事情，这个系统就有大问题。

最后，批处理是获得运行效率的主要方法。一次传送一个 50K 的文件比将传送 50 个 1K 块的传送的效率要好得多。

5.3 分布式文件系统的发展趋势

尽管计算机工业自诞生以来，迅速的变化已是它的组成部分之一，但近年来，新的发展，在硬件和软件领域都来得更快。许多硬件方面的变化可能对未来分布式文件系统的发展有主要影响。除了所有技术上的进步外，变化的用户期望和应用也可能是一个主要影响。本节我们将对在可预见的将来所期望出现的变化作一概括，并讨论这些变化对文件系统的某些潜在的影响。这一节将提出的问题比将要回答的问题更多，但这将为以后的研究提供一些更为有趣的方向。

5.3.1 新的硬件

在我们讨论新的硬件之前，让我们先以新的价格来看看旧的硬件。随着内存储器越来越便宜，文件服务器在组织方式上发生了革命性的变化。现在，所有的文件服务器都使用磁盘进行存储。主存储器主要用于服务器高速缓存，但这只是为获得更好的运行效率的一种优化，并不是本质。

近几年来，内存变得如此便宜，以至于甚至一些小型组织都能够为他们的文件服务器装配若干个 G 的物理内存。结果是，文件系统可永久地驻留内存，而不需要磁盘。这一步使得运行效率大大地提高，并大大简化了文件系统的结构。

大多数当前的文件系统都将文件组织成块的集合，或者是树（如 UNIX），或者是连接列表（如 MS-DOS）。使用驻留内存的文件系统，在内存中连续存放每个文件可能简单得多，而不是将它分成块。连续存放的文件更易于跟踪，而且在网上传输也更快。在磁盘上不使用连续文件的原因是，当文件增大时，将文件移到磁盘中一个足够大的区域的代价太昂贵。相反，将文件移到内存的另一个区域是可行的。

但是，主存文件服务器产生了一个严重的错误。如果断电，所有的文件都会丢失。不像磁盘在掉电时不会丢失任何信息，主存在停电时将被刷新。解决的办法是在录像带上做连续的或至少是增量的备份。使用当前的技术可以在一盘价值不操过 10 美元的录像带上存放大约 5 个 G 的信息。但是，访问时间很长，如果由于停电需要访问来进行恢复的次数一年只要一两次的话，这种模式也可能很具有诱惑力。

对文件系统可能产生影响的一个硬件发展是光盘。最初，这些设备的特性是只能写入一次（通过用激光在表面灼孔），但以后就不可改变了。它们有时被称为 **WORM (Write Once Read Many)** 设备。现在有些光盘可用激光去影响盘的晶体结构，但并不损坏它们，

这样，就可以对它们进行擦除。

光盘有三个重要的性质：

- (1) 速度慢；
- (2) 容量大；
- (3) 可随机访问。

尽管光盘比录像带贵，但还是相当便宜的。前面两个特性和录像带相同，但第三个特性则使得可以进行以下的事情。假设一个文件服务器在主存中有一个 n 个 G 的文件系统和一个 n 个 G 的光盘作备份。当创建文件时，将文件存放在主存中，并标记还没备份。我们使用主存进行所有的访问。当工作负荷很低时，将未作备份的文件传送到后台的光盘上，内存中第 k 个字节放在光盘的第 k 个字节。和第一种模式一样，我们这里是一个主存文件服务器，但多了一个与内存一一对应的更为方便的备份设备。

另一个令人感兴趣的硬件发展是快速的光纤网络。如我们在前面讨论的，进行客户高速缓存的原因和其所有固有的问题都是为了避免从服务器到客户的缓慢传输。但是，假设我们为系统装配一个主存文件服务器和一个快速的光纤网络，就有可能不用客户高速缓存和服务器的磁盘，而只利用有光盘作备份的服务器的内存进行操作。这样肯定能使软件简化。

在研究客户高速缓存的时候，我们看到，如果两个客户同时高速缓存同一文件，其中一个对它进行了修改，而另一个并不知道，这就导致了不一致性。稍微分析一下就可以发现这种情况和多处理机系统中的内存高速缓存非常类似。只是在那里，当一个处理机对一个共享字作修改时，通过总线向其他高速缓存发送一个硬件信号，以便让它们作废或更新该共享字。而在分布式文件系统中并不这样做。

实际上为什么不呢？原因是当前的网络接口不支持这种信号。不过，也有可能用那样的方法构造网络接口。举一个非常简单的例子，如图 5-11 所示的系统，其中网络接口有一个位图，每一个高速缓存了的文件对应一位。为修改文件，处理机对接口中相应的位进行设置，如果该位为 0，则说明当前没有处理机对该文件进行更新。对位的设置可使接口建立并发送一个绕环的包，该包可对所有接口中的相应的位进行检查和设置。如果该包绕一圈后一直未发现任何其他机器试图使用该文件，就将接口中的某一其他的寄存器设置为 1，否则，设置为 0。实际上，这种机制提供了一种在几微秒之内全程锁定文件的方法。

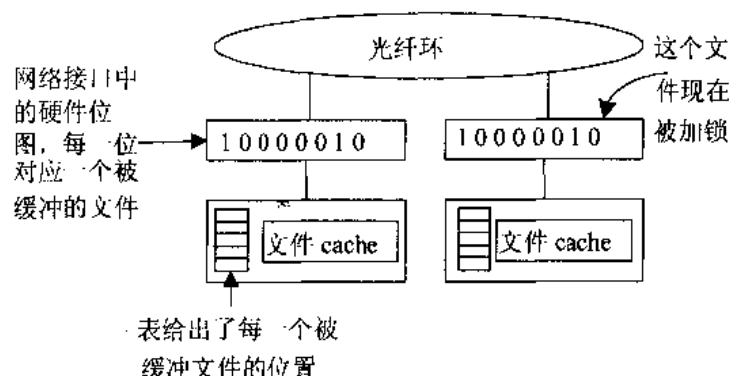


图 5-11 更新共享文件的硬件方案

设置锁定后，处理机就可对文件进行更新，对修改过的每个文件块作标识（例如，使用页表中的位）。当更新完成后，处理机清楚位图中的该位，这就使得网络接口使用内存中的表去定位文件，并自动将所有修改过的块传送到其他机器的适当位置。当文件在所有地方都被更新后，位图中的该位在所有机器上也都被清除。

很显然，这是一个在许多方面都可进行改进的简单的方法，但它说明了如何用少量的设计良好的硬件来解决那些用软件难以解决的问题。各种专门的硬件将对以后的操作系统提供支援。

5.3.2 规模

分布式系统的一个明确趋势是朝着越来越大的系统发展。这个观测对分布式文件系统的设计有提示作用。在有 100 台机器的系统上工作良好的算法，可能在有 1000 台机器的系统上就会工作很差，甚至在有 10000 台机器的系统上根本就不能用。一开始，集中式算法就没有规划好。如果每打开一个文件都要和一个单独的集中式服务器联系用来记录该文件已打开的事实，那么随着系统的增大，这个服务器最终将成为瓶颈。

解决这一问题的一般方法是将系统分成较小的单元，并尽量使它们相互独立。每个单元一个服务器要比单独的一个服务器规划起来好得多。在这样的环境下，即使让服务器记录所有的打开操作也是可接受的。

广播是另一个问题领域。如果每台机器每秒钟发出一个广播，有 n 台机器，每秒钟就有总共 n 个广播出现在网络上，一共会产生 n^2 次中断。显然，随着 n 的增大，这最终将成为问题。

资源和算法在用户数方面并不呈线性关系，因此，让服务器维护一个用户的线型列表用于保护或其他目的并不是一个好主意。由于访问时间或多或少是一个常数，几乎和项目数的多少无关，所以哈希表（HASH）是可接受的。

一般来说，严格的语义，如 UNIX 语义，在系统增大时更难以实现。更低的承诺要容易实现得多。显然，由于编程人员更喜欢定义良好的语义，但实际是严格的语义不好规划，因此，这里存在一个平衡的问题。

在非常大的系统中，可能必须重新检查一下单个 UNIX 似的文件树的概念。随着系统的增大，不可避免的是路径名的长度也会增大，系统的开销也会增多。从某种意义上看，有必要将树分成更小的树。

5.3.3 广域网

关于分布式系统，许多当前的工作都集中在以局域网为基础的系统上。将来，许多以局域网为基础的分布式系统将内联以形成透明的跨国跨洲的分布式系统。举一个例子，法国的 PTT 在法国的每一个公寓和房间都放置了一台小型计算机。尽管最初的目标是为了消除信息操作员和电话预定的需求，但到某个时候，就会有人问，是否可能将遍布法国的一千万或更多的计算机连成一个单独的透明的系统（这类应用目前还没有）。将需要何种类型的文件系统来为整个法国服务？为整个欧洲？为整个世界？目前，还没有人知道。

尽管法国的机器都是相同的，但在广域网中，会遇到许多不同类型的设备。当遇到只有不同预算和目标的多个购买者时，这种多样性是不可避免的，而且这种购买在技术迅速进步的年代会跨越许多个年头。这样一个广域的分布式系统必须能够处理异构。这就会产生诸如并不是每个人都用 ASCII 时，你应怎样存储一个字符文件，或当可用多种表示形式时，应用何种格式存储一个包含浮点数的文件等的问题。

在应用中所期望的变化也是很重要的。在大学里开发的大多数实验型分布式系统将在一个类似 UNIX 环境下的编程视为规范的应用，因为这是研究者们整天要做的事情（至少当他们不在开委员会或写助学金提议时）。最初的资料表明，并不是所有 5 千万法国人都列出 C 程序设计作为他们的主要活动。随着分布式系统越来越普及，我们很可能会看到它向电子邮件、电子银行、对数据库的访问以及娱乐活动的转移，这种转移将会改变文件使用、访问模式和大量的我们目前还不了解的方式。

大型分布式系统的一个固有问题是网络带宽极低。如果用电话线作为主连接线，要达到 64Kbps 以上的传输速率是不可能的。将光纤连到每个人的家还须几十年，且要花掉几十亿。另一方面，巨量的数据可廉价地存储在压缩磁盘或录像带上。和安装到电话公司的计算机上去查找一个电话号码相比，给每个人发一个包含整个数据库的磁盘或磁带可能更便宜。我们可能应该开发将静态的、只读的信息（如，电话预定）和动态的信息（如，电子邮件）进行区别的文件系统。这种区别可能成为整个文件系统的基础。

5.3.4 移动用户

便携式计算机已成为计算机商业中增长最快的部分。膝上型计算机、笔记本计算机和袖珍型计算机近年来随处可见，而且增长很快。尽管在驾驶时使用很困难，但飞行时已不成问题。现在，在飞机上装电话已很普遍，所以，在飞行时使用 FAX 和移动的调制解调器的日子还会很远吗？然而，从飞机到地面所需的总的带宽是相当低的，而且用户要去的许多地方根本没有在线连接。

必然的结果是，在大部分时间里，用户处于脱机状态，与文件系统失去连接。尽管 Satyanarayanan(1990b)已经报道在这个方向已进行了一些初始研究，但现在几乎没有为这样的使用而设计的系统。

任何解决方法可能都是以高速缓存为基础的。连接后，用户将那些以后要使用的文件下载到便携机上，这些文件在断开连接后使用。当重新连接时，高速缓存中的文件将和文件树中的文件进行合并。由于断开经常持续几个小时或几天，因此保存高速缓存的一致性比在联机系统中更为严重。

还有一个问题是，当重新连接时，用户可能在离他家非常遥远的城市。给家里的机器安一个电话是重新获得同步的一个方法。但电话的带宽很低。此外，在一个真正的分布式系统中，和本地的文件服务器连接可能就足够了。对一个世界范围的、完全透明的、供几百万移动的且经常断开的用户同时使用的分布式系统的设计留给读者作练习。

5.3.5 容错

当前的计算机系统，除了像空中运输控制那样的专门系统外，都不是容错的。当计算机出现问题时，将希望用户像平凡生活事实那样接受它。不幸的是，一般人还是期望事情能做下去。如果电视频道、电话系统或电力公司中断半个小时的服务，第二天就会出现许多不愉快的人。随着分布式系统越来越普及，要求系统根本不会失败的需求也会增长。当前的系统还不能满足这样的需求。

显然，这样的系统要求硬件和通信基础设施方面有可观的冗余。但也需要软件特别是数据的冗余。文件复制，当前分布式系统的一种常用的事后措施，将成为今后的系统的一个必须部分。由于坚持所有数据在任何时间都可用不会导致容错，系统也将被设计成当只有一部分数据可用时还能有效发挥作用。系统崩溃的时间对编程人员和有经验的用户来说，认为是可以接受的，但当随着计算机应用扩展到非专业人员，系统崩溃时间将变得越来越不可接受。

5.3.6 多媒体

新的应用，特别是那些涉及到实时视频或多媒体的应用，对今后的分布式文件系统将产生巨大的影响。文本文件几乎不会超过 1M 字节，但视频文件很容易就超过 1G 字节。要处理如视频需求类的应用，就需要一个完全不同的文件系统。

5.4 小结

任何分布式系统的核心都是分布式文件系统。这样一个文件系统的设计是从接口开始的：文件的模型是什么，提供什么样的功能？通常，文件的本质对分布的情形和单处理机的情形没有什么不同。一般来说，接口的一个重要部分是文件命名和目录系统。命名很快带来了透明性问题。文件的名字和它的位置联系到什么程度？系统能否在自己上面移动一个单独的文件而不需改变文件名，不同的系统对这些问题有不同的回答。

文件共享在分布式系统中是一个复杂而又重要的主题。我们推荐了各种语义模型，包括 UNIX 语义、对话语义、不可变文件和事务语义。每个都有各自的优点和弱点。UNIX 语义直观，大多数的编程人员（甚至非 UNIX 的程序设计者）熟悉它，但实现起来很昂贵。对话语义确定性小，但效率更高。不可更改文件语义大多数人不熟悉且修改文件困难。事务语义通常是不用的。

实现一个分布式文件系统要作许多决策。这包括系统是无状态的还是有状态的，是否使用高速缓存以及怎样实现高速缓存，怎样进行文件复制。每个这样的决策对设计者和用户都将产生深远的影响。NFS 展示了建立分布式文件系统的一种方法。

将来的分布式文件系统将要考虑在硬件技术、规模、广域系统、移动用户、容错的进展以及多媒体的引入。许多激动人心的挑战正等待着我们。

习 题

- (1) 使用上载/下载模式的文件服务系统与使用远程访问模式的文件服务系统之间有什么区别?
- (2) 一个文件系统允许从目录连接到另一个目录。用这种方法,一个目录可以“包含”一个子目录。在这章里,以什么样的准则来区分一个树结构的目录系统与一个普通图形结构的目录系统?
- (3) 在正文中指出共享文件指针不能想当然的在会话语义中实现。但在提供 UNIX 语义的单个文件服务上是否能实现?
- (4) 试说出不可更改文件语义的两个有用的特性?
- (5) 为什么有些分布式系统使用两级命名法?
- (6) 在无状态服务器中,每个请求为什么要包括一个文件区段?在有状态服务器中是否也需要?
- (7) 在正文中一个有争议的论点是对打开的文件可以存放一个 i-节点,从而减少磁盘操作。设计一个实现方案,以无状态服务器形式实现所有相同的性能指标。如果可以,在哪方面,你的设计优于或差于有状态服务器?
- (8) 当使用会话语义时,常常出现:修改一个文件立即被修改它的进程所知但却不为其他机器上的进程所知晓。然而,就有一个问题,这些文件是否能被同一台机器上的其他进程所知晓呢?给出每一方法及其理由。
- (9) 为什么文件高速缓存技术可以使用 LRU 而虚拟存储页算法不可以?试以相关图形进行说明。
- (10) 在 cache 一致性一节中,建议的方法是与服务器连接,我们讨论了客户机 cache 管理者如何知道其 cache 中的文件仍然有效。建议的方法是与服务器联系比较客户机与服务器的时间即可。但如果客户机与服务器的时钟不一致,该方法是否就失效呢?
- (11) 考虑这样一个系统,客户机高速缓存使用的是 write-through 算法,而且存储的仅是个别模块而不是整个文件。假设一个客户机相继读取整个文件,某些模块存放在 cache 中而有一些没有,会出现什么问题,又如何解决呢?
- (12) 设想一个分布式文件使用客户机高速缓存技术,运用延迟回写策略。一台机器打开,修改并关闭一个文件。30 秒后,另一机器从服务器读取文件,它得到的是哪一版本。
- (13) 一些分布式文件系统中利用客户机高速缓存技术使用延迟回写算法或 write-on-close 算法,若加上语义问题,该系统又会导致另一个问题,这个问题是什么?
- (14) 测试表明,许多文件只有一个很短暂的寿命,这对客户机高速缓存技术又有什么意义呢?
- (15) 如我们在这章里所讨论的,一些分布式文件使用两级命名法,ASCII 码和二进制码;有些则不然,而全部使用 ASCII 码。类似的,有些文件服务器是无状态的,有些是有状态的,给出这两种特征的四种结合方式。其中一种结合方式和其他方式相比几

乎是不可取的，是哪种，为什么不可取？

- (16) 文件系统复制文件时，一般不复制所有的文件，举例说出哪一类文件是不用复制的？
- (17) 一个文件在 10 个服务器上复制，试列举表决算法所允许的读数定额与写数定额的所有组合。
- (18) 文件服务器的一个主存连续地存储文件，当文件增大超出了当前所在单元，该文件要进行备份。假设一般文件长为 20M 字节，每拷贝 32 字节的字需要 200 纳秒 (10^{-9} 秒)，那么一秒钟能拷贝多少文件？你能给出一种方法，备份文件而又不一直占用文件服务器 CPU 的时间？
- (19) 在 NFS 中，当打开一个文件时，则返回一个文件句柄，类似的在 UNIX 中有一个文件描述符被返回。假设在将文件句柄送回给用户后，NFS 服务器崩溃。当服务器重新启动时，该文件句柄是否有效。如果有效，又如何工作？如果无效，这是否又违反无状态模式的原理呢？
- (20) 在图 5-11 的二进制映射方案中，是否所有机器必需使用同一表记录来存取一个给定的文件，如果可以，又如何实施？

第 6 章 分布式共享存储器

在第 1 章中，我们看到存在着多处理机和多计算机这样两种多处理机系统。在多处理机系统中，两个或更多的 CPU 共享一个存储器。任何处理机上的任何进程只需将数据移入移出就可读写共享存储器中的数据。而在多计算机系统中，每个 CPU 有各自的存储器，并不共享。

以农业为例，一个多处理机可以看作这样一个系统：一群猪（进程）用一个食槽（共享存储器）进食。而多计算机则是每只猪拥有自己的食槽。以教育为例，多处理机就是教室前面的一块黑板，所有学生都可以看到，而多计算机则是每个学生看自己的笔记本。尽管这种区别看起来微不足道，其影响却是深远的。

其结果涉及到软件和硬件两个方面。首先看看对硬件的影响。设计一种使多个处理机同时使用同一存储器的机器是非常困难的。如 1.3.1 节所述，由于总线会成为一种瓶颈，基于总线的多处理机总数就最多只能有几个。1.3.2 节介绍的交换式多处理机可应用于大系统，但相比之下过于昂贵、缓慢、复杂以及难以维护。

相反，大的多计算机系统更易于建立。单主板计算机（包含一个 CPU，一个存储器，一个网络接口）可以相互连接在一起，而不受数量的限制。许多生产商都提供包含数以千计处理机的多计算机系统。从硬件设计者的角度看，多计算机系统比多处理机系统更优越。

现在看看它对软件的影响。用于多处理机编程的技术很多。为实现通信，一个进程将数据写到存储器，其他进程将之读出。为保持同步，可以使用临界区（critical region），信号量或管程（monitor）提供必要的互斥。很多文章都讨论了在共享存储器的计算机上实现通信与同步的问题。在过去的 20 年中，每一本操作系统的教科书都有几个章节讨论这个问题。简而言之，大量的理论和实践知识可用于多处理机编程。

对多计算机系统，情况正好相反。通信一般使用消息传递，这使输入/输出更为抽象。消息传递带来了许多复杂的问题，包括：流控制、信包丢失、缓冲区设置和拥塞。尽管已提出了不同的解决方案，但消息传递的编程仍很复杂。

为回避消息传递中的一些困难，Birrell 和 Nelson（1984）提议采用远程过程调用。在他们的方案中，实际的通信隐藏在过程库中。为了使用远程服务，进程调用相应的库过程，它将操作码和参数组装成一条消息，通过网络送出，等待应答。但是像图像和其他包含指针的复杂数据结构不易发送。且它不能用于使用全局变量的程序。另外，用这种方法传送大数组也相当昂贵，这是因为它们必须通过值而不是引用来传送。

总之，从软件设计者的角度来看，多处理机系统要比多计算机系统优越。这两种方法各有优缺点：多计算机系统易于建立但难于编程；多处理机系统正相反，易于编程但难于建立。我们需要既易于建立又易于编程的系统。本章便试图建立这样的系统。

6.1 简介

在早期的分布式系统中,人们认为程序运行在不同的地址空间,没有物理上的共享存储器(例如多计算机)。在这种思维模式下,通信自然被看作不相关地址空间的消息传递。1986年,Li提出了不同方案,这就是现在所说的分布式共享存储器(DSM)(Li,1986;Li及Hudak,1989)。简言之,Li和Hudak提出让一组由局域网互连的工作站共享一个分页的虚拟地址空间。在最简单的变形中,每一页刚好在一台机器上。对本地页的访问以及存储器速度由硬件实现。试图访问其他机器上的页将导致缺页错误,它激活操作系统的陷阱程序。操作系统向远程机器发送消息,远程机器查找所需页面,将它发送给提出请求的处理机。失败的操作将重新开始并得以完成。

这种设计与传统的虚拟存储器系统有类似之处:当一进程访问一未驻留存储器的页时,激活陷阱,操作系统获取相应页并将其调入。不同之处在于:操作系统不是从磁盘中而是通过网络从另一个处理机中获取页。对用户进程来说,系统更像传统的多处理机、多个进程随意地读写共享存储器。所有的通信和同步都通过存储器来完成,对用户进程不可见。实际上,Li和Hudak设计了一种既易于编程又易于建立的系统。

然而,十全十美是不可能的。尽管这个系统既易于编程又易于建立,但是许多应用程序在这个系统上运行的情况并不乐观。页在网上频繁地调进调出,这种情况与单处理机虚拟存储器系统的抖动类似。近几年来,如何使这种分布式共享存储器系统更为有效已成为一个有待深入研究的领域,并研究开发了大量的新技术。所有这些都是为了缓解网络冲突、减少存储器请求与应答之间的时间。

另一种方法是不共享整个地址空间,而只共享其中所选择的一部分,即那些由多个进程引用的变量和数据结构。在这种模式下,每台机器并不直接访问一般存储器而是访问共享的变量集,这是一种更高层次上的抽象。这种策略不仅极大地减小了必须共享的数据量,而且在大多数情况下,共享数据的大量信息,比如类型,也是可得的,这有助于优化系统性能。

还有一种可能的优化方法是在多台计算机上复制共享变量,通过共享复制的变量而不是整个页,模拟处理机的问题可简化为保证一组数据结构的多个拷贝一致性的问题。一般来说读操作可以在本地进行而不引起任何网络通信,写则可以通过多拷贝更新协议完成。这种协议在分布式数据库中已得到广泛应用。

进一步研究地址空间的构成,我们不仅可以共享变量,还可以共享通常被称为对象的封装的数据类型。与共享变量不同,每个对象不仅有数据还有称之为方法的过程。程序不能直接访问数据而必须通过调用方法来访问。通过这种限制访问数据,我们有可能采用各种新算法。

和硬件分页相比,仅由软件实现分布式共享内存既有优点,也有缺点。我们通常倾向于在程序里加入一定限制来获得较好的性能。本章后面将涉及到这一点。

在详细介绍分布式共享存储器之前,我们首先回顾一下什么是共享存储器以及共享存储器的多处理机系统是怎样工作的。由于共享的含义很微妙,所以在后面我们将分析共享的语义。最后,我们将回到分布式共享存储器的设计上来。因为分布式共享存储器与计算机结构、操作系统、实时系统,甚至程序设计语言都有关系,所有这些在本章中都将提到。

6.2 什么是共享存储器？

这一节我们将比较几种共享存储器（内存）的多处理机，范围从简单的单总线多处理机到具有高级缓存方案的多处理机。DSM 的很多研究工作都受到了多处理机结构发展的启发，所以理解这些机制对理解分布式共享存储器（DSM）是很重要的。而且，许多算法是如此的相似以至于很难区分出一台高档计算机是多处理机还是应用了由硬件实现分布式共享存储器的多计算机，我们将比较各种多处理机结构和分布式共享存储器系统，从硬件到软件，提出各种可能的设计方案。通过分析这一系列方案我们会了解 DSM 更适用于哪种场合。

6.2.1 芯片存储器

尽管大多数计算机都用和 CPU 分离的存储器，但仍然存在着包含 CPU 和存储器的芯片。这种芯片生产量数以百万计，广泛应用于汽车、电气用具、甚至玩具。在芯片里，CPU 和存储器通过地址线 and 数据线直接相连，图 6-1 (a) 是这种芯片的一个简图。

可以设想，只要将这种芯片简单扩展，多个 CPU 就可以共享同一存储器，如图 6-1 (b) 所示。构建这样一个芯片不仅复杂，而且非常昂贵和不实用。如果想构建这样一个单芯片多处理机，想使 100 个 CPU 可以同时访问一个存储器，在工程上是不可实现的。我们需要采用其他方法共享所需存储器。

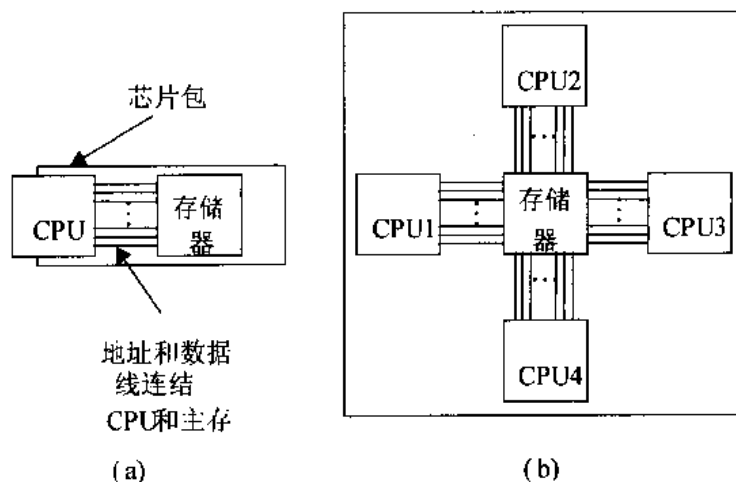


图 6-1 (a) 单片计算机
(b) 理想的共享存储器多处理机

6.2.2 基于总线的多处理机

从图 6-1(a) 中可以看出 CPU 和存储器由并行线连接，一些具有 CPU 要读写的地址，一些接收和发送数据，其余用于传输控制，这些线称为总线。这里的总线是在芯片内，但在大多数系统中总线都是外置的，用于连接包含 CPU、存储器和 I/O 控制器在内的印刷电路板。

在台式计算机中，总线一般蚀刻在主板上，主板上插有 CPU、存储器和 I/O 控制卡。

基于总线相连的多个 CPU 是构建多处理器的一种简单有效的方法。图 6-2 (a) 中描述的系统具有 3 个 CPU 且它们共享一个存储器。当任一 CPU 要从存储器中读取数据时，它将数据的地址放在总线上，并在控制总线上加上“读”信号。存储器读出需要的数据，将其放在总线上，在控制总线上加“准备好”信号，CPU 即可读到相应的数据。写的情况类似。

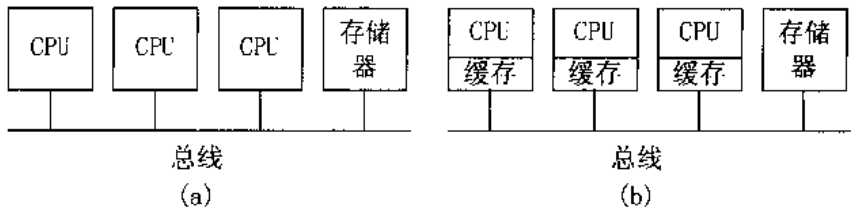


图 6-2 (a) 多处理器
(b) 带缓存的多处理器

为防止多个 CPU 同时访问存储器，需要总线仲裁机制。现已采用了多种方案。例如 CPU 要获取总线控制权，首先要提出申请，获得允许后方能使用。这种申请可以采用集中式即总线仲裁机构来仲裁；也可采用非集中方式即先申请的 CPU 先获得总线控制权。

单一总线的不利之处是在有 3-4 个 CPU 时就可能导致超载。降低总线负载的一般方法是给每个 CPU 提供一个监听高速缓存 (snooping cache 或 snoopy cache)。称之为监听，是因为它在总线上“监听”信号。监听高速缓存如图 6-2 (b) 所示。这已经成为许多研究机构近年来的研究方向。(Agarwal 等,1988; Agarwal 和 Cherian,1989; Archibald 和 Baer, 1986; Cheong 和 Veidenbaum,1988; Dahlgern 等,1994; Eggers 和 Katz, 1989a,1989b; Nayfeh 和 Olukotun,1994; Przybylski 等,1988; Scheurich 和 Dubois,1987; Thekkath 和 Eggers,1994; Vernon 等 1988; Weber 和 Gupta,1989)。这些文章都类似地提出了缓存一致性协议，即保证不同的缓存对于相同的存储器位置具有相同的值。

其中一种特别简单、通用的协议称为通写 (write-through)。当 CPU 从存储器中首次读取某字时，该字通过总线取出并存在提出请求的 CPU 缓存中。当再次需要这个字时，CPU 不再提出访问存储器的请求，而是直接从缓存中获取，这减少了总线流量。表 6-1 的前两行说明了读取的两种情况，读失败 (该字不在 Cache 中) 和读命中 (该字在 Cache 中)。在简单的系统中，只将请求的字放在 Cache 中，但在大部分系统中，通常第一次访问后，将一数据块即 16 或 32 字节传送并置于 Cache 中，便于下次访问。

表 6-1 通写缓存一致性协议

事件	缓存响应本地 CPU 操作时执行的动作	缓存响应远程 CPU 操作时执行的动作
读失败 (Read miss)	从存储器中取得数据并存储到缓存中	(无动作)
读命中 (Read hit)	从本地缓存中取得数据	(无动作)
写失败 (Write miss)	更新存储器中的数据并存储到缓存中	(无动作)
写命中 (Write hit)	更新存储器和缓存	使缓存项无效

说明：第三列中的“命中”项说明监听 CPU 的缓存中有该字，而不是请求的 CPU

由于每个 CPU 都独立完成缓存操作, 因此也可能两个或更多 CPU 同时对一个字执行了缓存操作。我们来看看执行写操作时的情况。如果某字不在 CPU 的缓存中, 那么仅仅更新存储器, 就像从未使用缓存一样。这个操作需要一个普通的总线周期。如果执行写操作的 CPU 拥有字的唯一拷贝, 则缓存和存储器通过总线同时更新。

如果情况仅仅是这样, 就不会发生错误。但当某 CPU 要写的字也在其他 CPU 的缓存中时, 就可能会出现问題。当 CPU 对在缓存中的字执行写操作时, Cache 中的项便被更新。不管怎样, 存储器都通过总线更新。其他 CPU 的缓存监听到写操作 (它们总在总线上监听), 检查自身是否有这个被更改的字。如果有, 则将该字的缓存项置无效。当写操作完成时, 存储器便被更新并且只有一个 CPU 的缓存中有这个字。

除了将其他的缓存项置无效, 也可将它们全部更新。在大多数情况下, 更新要比置无效慢得多。置无效只需提供地址, 更新还需要提供新的值。如果这两项顺序出现在总线上, 就可能会出现多余的循环。即使将地址和数据同时写入总线, 但若缓存块长于一个字, 更新整个数据块也得需要多个总线循环。这种置无效和更新的问题在所有的缓存协议以及 DSM 系统中都存在。

表 6-1 概括了整个协议。第一列列出了可能发生的四种基本事件。第二列说明缓存如何响应相关 CPU 的操作。第三列说明缓存发现 (通过监听) 其他 CPU 的读写操作时如何反应。例如, 当 Cache S (监听者) 发现其他 CPU 写入的字在其 Cache 中 (从 S 的角度看是写命中) 时, S 必须采取措施。S 的响应是从本缓存中删除这个字。

通写协议 (*write-through protocol*) 易于理解和使用, 其不利之处在于所有的写操作必须通过总线。尽管总线流量在某种程度上确实是减小了, 但允许挂在单一总线上的 CPU 数量仍很少, 不能满足大型多处理机的需要。

可喜的是, 对大多数实际的程序来说, 一旦 CPU 写了某字, 它很可能再次需要这个字, 而其他 CPU 不大可能立即需要该字。这种情况表明如果让正在使用某字的 CPU 暂时拥有该字的拥有权, 直到别的 CPU 需要该字, 那么就可以避免更新存储器的顺序写操作。这种缓存协议是存在的。1983 年, Goodman 设计了第一个这样的协议: *write once*。然而, 他设计的协议考虑了与当时的现有总线协作问题, 因此比所需的更为复杂。下面我们将给出一个它的简化版本, 这个版本在所有权协议中具有典型性。Archibald 和 Baer(1986)描述及比较了其他协议。

该协议管理缓存块, 每个块处于以下三种状态之一:

- (1) 无效 —— 本缓存块无有效数据;
- (2) 干净 —— 存储器被更新, 该块可能在别的缓存中;
- (3) 脏 —— 存储器错误, 该数据块不在其他缓存中。

基本思想是允许正被多个 CPU 读取的字出现在它们所有的缓存中。而仅被一个 CPU 经常写的字将只保存在它的缓存中, 为减少总线流量, 不必每次都写回存储器。

下面举例说明协议的操作过程。为简化本例, 我们假定每个缓冲块只含一个字。首先, 如图 6-3 (a) 的缓存块中有一个字 W , 地址为 W 。存储器 B 拥有 W 的有效拷贝。图 6-3(b) 中, A 提出请求并从存储器中得到 W 的一个拷贝。 B 监听到此读操作, 不作反应。

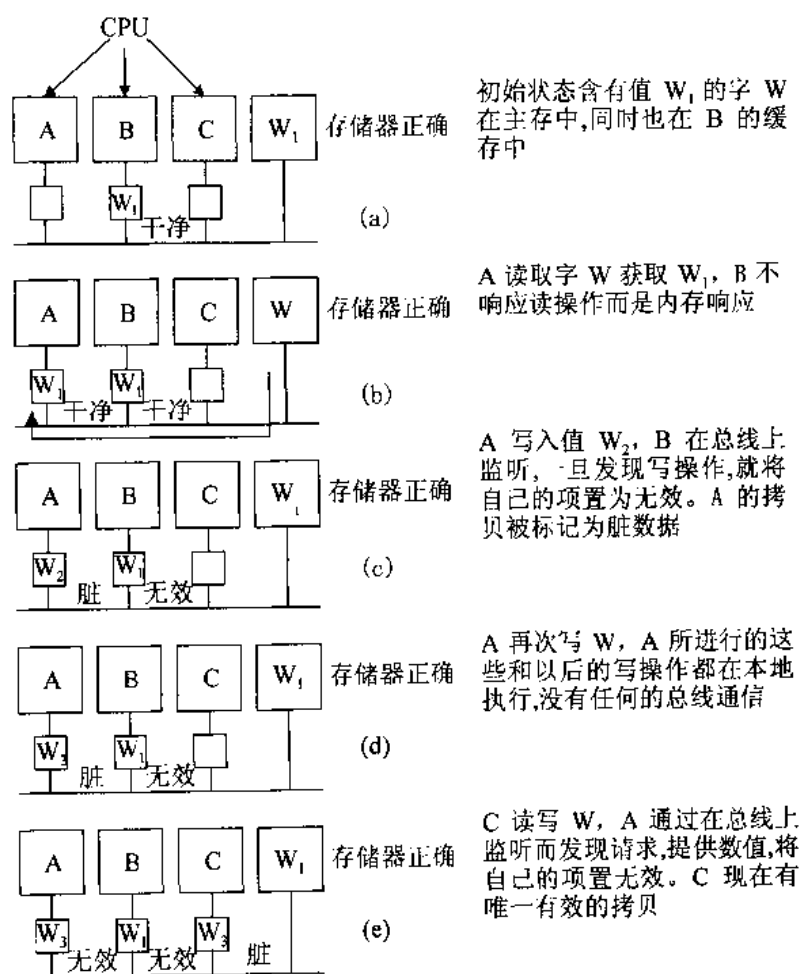


图 6-3 缓存拥有权协议工作的例子

现在 A 将新值 W_2 写入 W, B 监听到写请求, 将自己的缓存项置无效。如图 6-3(c), A 的缓存块状态变为 DIRTY。这说明 A 持有 W 的唯一拷贝, 存储器中的 W 值已经失效。

这时, 如图 6-3 (d), A 又一次重写了该字。写操作在本地缓存完成, 不需总线通信。此后一系列写操作都不用更新存储器。

此后, 其他 CPU, 如图 6-3(e)中的 C 访问了这个字。A 发现总线上的访问请求则发信号禁止存储器响应。然后, A 提供 C 所需的字并将自己的项置为无效。C 发现该字来自其他的缓存而不是存储器, 且其状态为 DIRTY, 则也相应的标示了自己的缓存项。C 现在是该字的拥有者, 这意味着 C 可以不通过总线请求而直接读写该字。当然它还要监听其他 CPU 的请求, 并提供服务。直到缓存缺乏空间而清除该字, 该字状态始终为 DIRTY。那时, 该字从所有的缓存中消失, 回写到存储器。

许多小型多处理机采用与此类似的缓存一致性协议, 它有三个重要属性:

- (1) 缓存对总线的监听保证了一致性;
- (2) 协议建立在存储器管理单元中;
- (3) 整个算法在一个存储器周期中完成。

后面我们将会看到，大型交换式多处理机系统并不具有某些属性，DSM 则不具有上面所有属性。

6.2.3 基于环的多处理机

通向 DSM 系统的下一步是基于环的多处理机，例证由 Memnet 提出 (Delp,1988; Delp 等,1991; Tam 等,1990)。在 Memnet 的例子中，一个独立的地址空间被分成一个私有区和一个共享区。私有区被分成段，使得每个机器都有一个段用来存放堆栈和其他非共享的数据和代码。共享区对所有的机器都是一样的，保持其一致性的协议类似于基于总线多处理机的。共享存储器被分成 32 字的块，是机器间传输的单元。

Memnet 中的机器由一个改进的令牌环连接。该环包括 20 条并行线，允许每 100nsec 发送 16 位数据和 4 位控制信号，数据传输率为 160Mbps。环如图 6-4(a)所示。环接口，MMU (存储器管理单元)、缓存和一部分存储器集成在一个称为 Memnet 设备(Memnet Device)的器件中，如图 6-4(b)中上面三分之一所示。

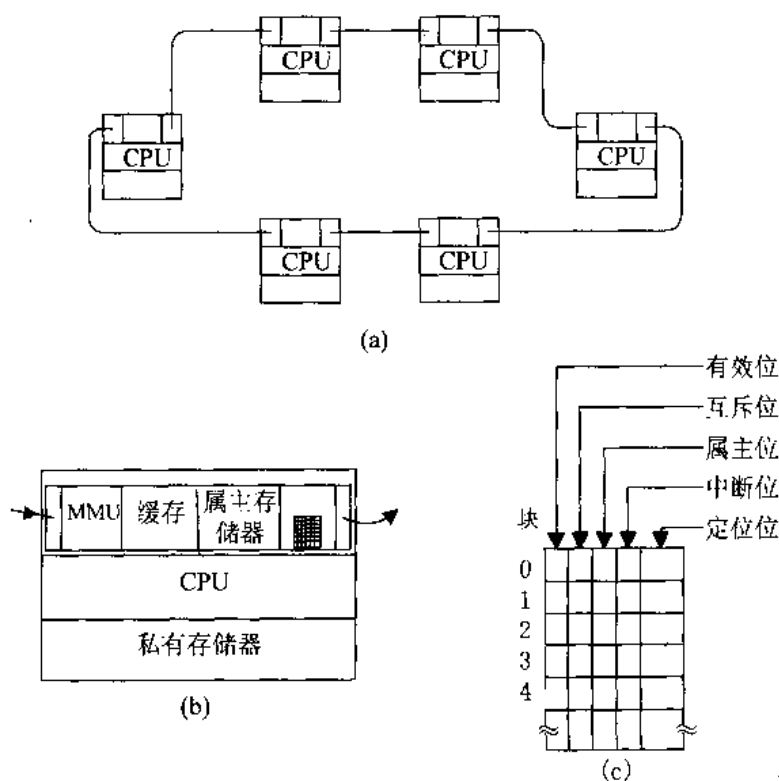


图 6-4 (a) Memnet 环
(b) 单一主机 (c) 块表

不同于图 6-2 所示的基于总线的多处理机，在 Memnet 中没有集中式全局存储器。共享地址空间中的每个 32 字节数据块都有一属主机器 (home machine)，该机器在其属主存储器 (home memory) 域中始终为这 32 位数据块保留物理存储器，如图 6-4 (b) 所示。数据块可被置于除属主机器以外的任何机器的缓存中。cache 和 home memory 区实际上

是共享缓冲池的，因它们的使用稍有不同，所以将它们分为两部分来对待。只读块可以出现在多个机器上；可读可写块只能出现在一台机器上。在这两种情况中，块都不需要置于属主机器中。属主机器所做的是保证该块不在其他机器的缓存时，有一个确定的地点保存。由于没有全局存储器，这一点是必要的。实际上，全局存储器已散布到整个系统中。

如图 6-4 (c)，每个机器的 Memnet 设备上有一个表，包括每个块在共享地址空间的入口，以块序号为索引。每个表项包含一个有效位，判断块是否在缓存中且已被更新；一个互斥位，确定本地的拷贝是否唯一；一个属主位，只有当本机是该块的属主机器时才置位；一个中断位，用于强制中断；一个定位位，表明若块存在并有效，它定位在缓存中何处。

看过 Memnet 的结构，再来看一下它所用的协议。当 CPU 要从共享存储器中读一个字时，该存储器地址传送给 Memnet 设备，Memnet 设备检查表项以确定该块是否存在。若存在，读请求立即得到满足。否则，Memnet 设备等待直到俘获循环的令牌，将请求信包送入环中，并挂起 CPU。请求信包包含目标地址和 32 字节哑域 (dummy field)。

信包在环中传递的过程中，每个 Memnet 设备都检查自己是否有所需的块，若有则将块置入哑域，修改信包头使下一机器不再重复此工作。若块中的互斥位置位，则清除它。因为某数据块一定位于系统某处，因此当信包返回发送点时，一定包含了所需的块。CPU 发出请求并存储块，满足请求以后释放 CPU。

若发出请求的机器没有足够的缓存容纳到来的块，就会出现这个问题。为了得到空间，它随机取一缓存块送入属主机器中，释放该缓存块。因为属主位已置位的块已在属主机器中，所以不再被选取。

写操作与读操作有些不同。这里有三点需要区别。

若包含要写字的块存在于本地并且是系统中的唯一拷贝 (互斥位置位)，字就在本地写入。

若所需块在本地存在但不是系统中的唯一拷贝，CPU 将在环中发出置无效信包，强制其他机器抛弃此块的拷贝。当置无效信包返回时，将该块的互斥位置位，字在本地写入。

若本地不存在所需块，CPU 则发出包含读请求和置无效请求的信包。第一个有此数据块的机器将该块拷入信包，同时抛弃自己的拷贝。其余的机器仅从自己的缓存中抛弃该数据块。当信包返回到发送者时，本地存储该数据块并执行写操作。

在许多方面，Memnet 与基于总线的多处理机有相似之处。在这两种设计中，读操作返回的都是最近写入的值。读操作时，数据块可能在一个缓存中，也可能在多个缓存中。写操作时，数据块只能在一个缓存中。两者的协议也很相似，但 Memnet 没有集中式全局存储器。

基于总线的多处理机和像 Memnet 这样的基于环的多处理机的最大的区别是：前者是紧耦合的，CPU 通常在一个框架上。而使用基于环的多处理机的机器耦合得较松一些，甚至有可能像局域网上的机器那样在一座楼的桌面上分布，尽管这种松耦合会影响性能。再者，不像基于总线的多处理机，像 Memnet 这样基于环的多处理机没有集中式全局存储器。在这两方面，基于环的多处理机几乎就是 DSM 的硬件实现。

有人试图说明基于环的处理机在理论上是不应该存在的，因为它结合了两种被认为互斥类别的特性 (多处理机和 DSM 机对应哺乳动物与鸟的关系)。然而，它确实存在，说明这两种类别并不像人们想像的那样不同。

6.2.4 交换式多处理机

尽管基于总线和基于环的多处理机对于小系统（不超过 64 个 CPU）很适用，但对于有着成百上千个 CPU 的系统来说，情况就不容乐观。CPU 增加到一定数量时，总线或环的带宽达到饱和，再增加额外的 CPU 也不会提高系统性能。

我们一般采用两种方法解决带宽不足的问题：

- (1) 减少通信流量；
- (2) 增加通信容量。

我们已经看到试图通过缓冲减少通信流量的例子。这方面的工作集中在改善缓冲协议，优化块大小，重组程序，以提高存储器访问的本地命中率。

但是，总有这样的一天：不知足的设计者需要添加更多的 CPU，虽然使用了书中的所有技术，但也不再剩余总线带宽了。唯一的办法只有增加总线带宽。一种措施是改变拓扑结构，例如从一根总线到两根总线到树状结构到网状结构。通过改变互联网络的拓扑结构，可能会增加通信容量。

另一种方法是为系统建立层次结构。仍在一根总线上挂一些 CPU，但将整个单元（CPU 加总线）看作一个簇。如图 6-5 (a)，整个系统中有多个簇，用簇间总线连接。只要大多数 CPU 主要在簇内通信，就可保证簇间的通信流量相对较小。若一条簇间总线不够，可以增加第二条簇间总线，或者将簇排列成树状或网状。若仍需要更多的带宽，则将这些以总线相连的或是树状、网状的簇组成超级簇，将系统分成若干超级簇，超级簇也可以用单总线相连或组织成树状、网状结构。图 6-5 (b) 是一个三级总线的系统。

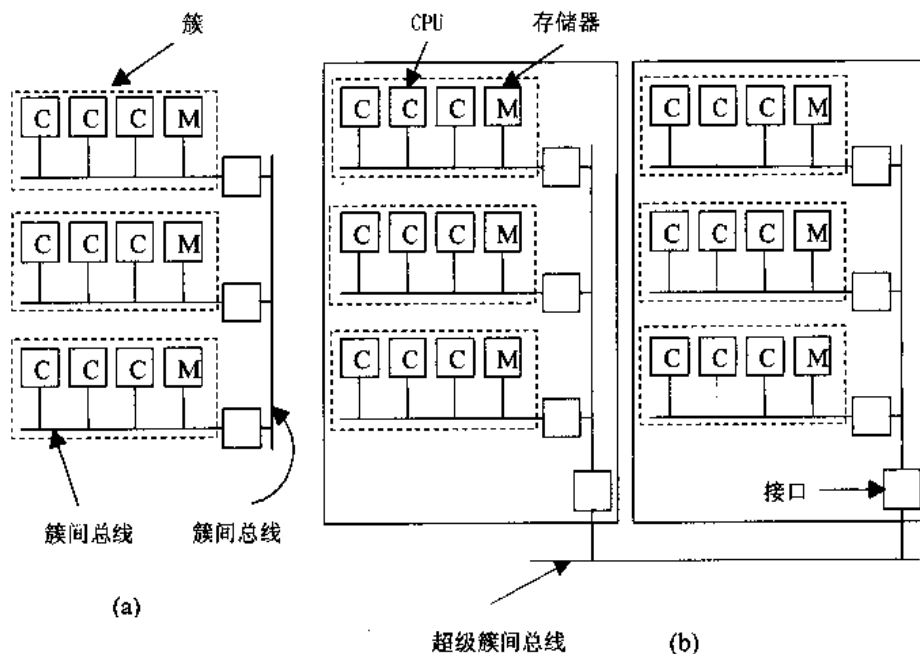


图 6-5 (a) 三个簇由簇间总线连接组成一个超级簇
(b) 由超级簇间总线相连的两个超级簇

本节我们来看看基于网状簇的分层设计。Dash 机是斯坦福大学的一项研究课题 (Lenoski 等,1992)。尽管许多学者都在做着类似的工作, Dash 机仍是比较典型的一个。下面将着重讨论一个已实际建立的具有 64 个 CPU 的原型。它的设计原理经过严格筛选以便于建立更大规模的版本。为避免一些不必要的细节,在下面的描述中,我们简化了一些地方。

图 6-6(a) 是 Dash 原型的简化拓扑结构图。它包括 16 个簇,每簇有一条总线,4 个 CPU, 16M 全局存储器和一些 I/O 设备(如磁盘等)。为避免结构图过于杂乱,每簇省略掉 2 个 CPU 和 I/O 设备。每个 CPU 可监听本地总线,如图 6-6 (b),却不能监听其他总线。

原型中总共可用的地址空间为 256M,分为 16 块,每块 16M。簇 0 的全局存储器地址是 0~16M,簇 1 的全局存储器地址是 16M~32M,以此类推。存储器以 16 字节的块为单位设置缓冲和传送。因此每一簇的地址空间中有 1M 存储器块。

目录

每一簇都以目录来记录哪些簇现在拥有哪些块的拷贝。因为每簇有 1M 存储器块,在它的目录中就有 1M 个项,每个块 1 项。每一簇的每个表项保留一个一位的位图,以判定簇是否将该块放入缓冲区。表项中还有 2 位表明块的状态。下面我们会看到目录对于 Dash 的操作很重要。实际上, Dash 的名字就来源于“共享存储器的目录结构”(Directory Architecture for Shared memory)。

每簇拥有 1M 18 位的项意味着每个目录的大小超过 2M。16 簇的目录所需存储器超过 36M,几乎是 256M 的 14%。若每簇的 CPU 数量增加,目录所需存储器的数量不变。每簇拥有更多 CPU 使得目录开销分摊到大量的 CPU 上,减少每个 CPU 开销。同时,每个 CPU 的总线控制和目录开销也减小了。理论上,1 个 CPU 为一簇最好,但总线和目录的开销对一个 CPU 来说太大了。

位图不是跟踪哪一簇拥有那些缓存块的唯一方法。另一方法是以列表形式组织每个目录项,判断哪些块拥有相应的缓冲块。若有共享的数据较少,则列表法需要较少位,若有大量共享,则需很多位。列表法的不足之处在于变长度的数据结构,但这可以解决。例如 M.I.T. Alewife 多处理机 (Agarwal 等, 1991; Kranz 等, 1993) 与 Dash 有许多相似之处,但 M.I.T. Alewife 在目录的记录上使用表格而不是位图,目录溢出处理也是采用软件的方法。

Dash 中的每个簇通过一个接口连接,这个接口用于簇间通信。如图 6-6 (a),接口由簇间链(简单的总线)连接成一个四方网格。系统中增加簇后,也增加了更多簇间链,带宽和系统规模增大。簇间链系统使用蛀孔式路由 (Wormhole Routing),这意味着在整个信包接收完之前,已可以处理第一部分信包,从而减少了每步的延迟。实际系统有两套簇间链,一套用于发送请求信包,一套用于发送应答信包,图中没有标出。簇间链不可被监听。

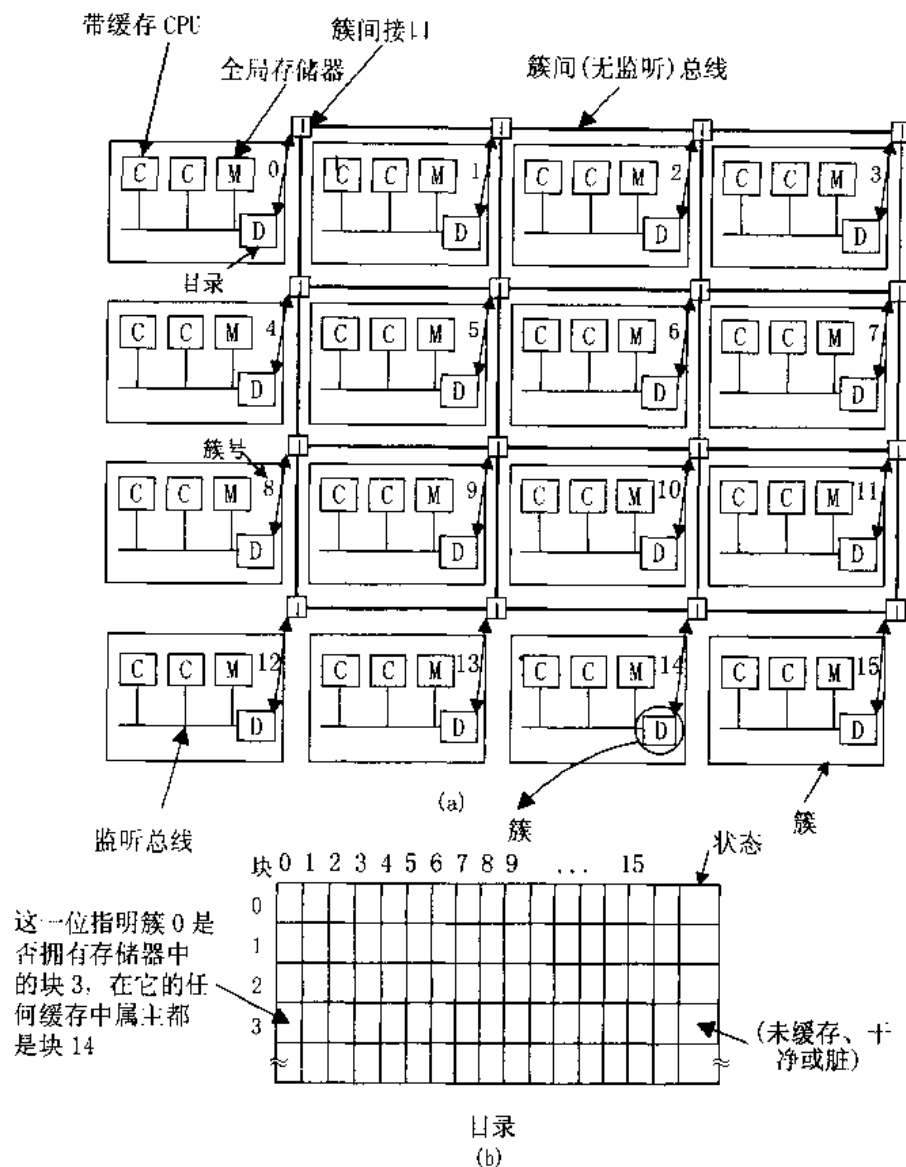


图 6-6 (a) Dash 体系结构的简单表示, 每一个簇实际上有四个 CPU, 这里只显示了两个
(b) Dash 目录

缓存 (caching)

缓存分为两级: 第一级缓存和更大的第二级缓存。第一级缓存是第二级缓存的子集。所以这里我们只关心后者。每个缓存 (第二级) 都用类似于图 6-3 的缓存拥有权协议监听本地总线。

每个缓存块有下列三个状态之一:

- (1) 未缓存(UNCACHED)——存储器中有此块的唯一拷贝;
- (2) 干净(CLEAN)——存储器已更新, 块可能在若干缓存中;
- (3) 脏(DIRTY)——存储器错误, 块仅在一个缓存中。

如图 6-6 (b) 所示, 每个缓存块的状态存于其目录项的状态域中。

协议 (protocol)

Dash 协议是基于拥有权和置无效的。在任何时候每个缓存块都有唯一拥有者。对于状态为 UNCACHED 或 CLEAN 的块, 块的属主所在簇 (Home Cluster) 为拥有者。对于状态为 DIRTY 的块, 拥有唯一块拷贝的簇为拥有者。对一个状态为 CLEAN 的块执行写操作需要首先找到所有拷贝并将之置无效。这就是引入目录结构的原因。

为说明这种机制怎样运行, 我们首先看看 CPU 如何读取一个存储器中的字。CPU 首先检查自己的缓存。若缓存中无此字, 则在本地簇内发出请求, 询问本地簇的其他 CPU 缓存中是否有此字。若有, 该数据块从该缓存传送到发出申请的 CPU 缓存中。若数据块的状态是 CLEAN, 拷贝此数据块。若块的状态是 DIRTY, 其目录将此块标记为 CLEAN, 使之共享。无论如何, 只要命中缓存就能满足指令, 但不影响任何目录的位图。

如果数据块不在任何簇的缓存中, CPU 将发一请求信包给该块的属主所在簇, 这可以由存储器地址的高 4 位来确定。属主所在簇可能就是请求者所在的簇。在这种情况下消息并没有被实际发送。属主所在簇的目录管理硬件检查它的表以确定块的状态。若为 UNCACHED 或 CLEAN, 硬件从全局存储器中取出块, 将其发送到请求簇, 该簇更新其目录, 表明该块已在缓冲区中 (如果还没有这样标记)。

如所需块的状态为 DIRTY, 目录硬件查找拥有该块的簇的标志, 该簇响应请求。拥有 DIRTY 块的簇将数据块发给请求簇, 并因为数据块已共享, 将其状态改为 CLEAN; 它还要给属主所在簇发回一个拷贝以更新存储器, 这时块的状态被置为 CLEAN。如图 6-7 (a) 所示, 若块被置为一个新状态, 因为目录跟踪状态的变化, 其目录也将随之改变。

写操作与此不同。在写以前, 执行写操作的 CPU 必须确定它拥有系统中该缓存块的唯一拷贝。若该 CPU 的缓存中已有此数据块且状态为 DIRTY, 可以立即写; 若缓存中有数据, 但状态为 CLEAN, 必须发送信包给属主所在簇, 记录所有其他拷贝并将之置无效。

若发送请求的 CPU 没有数据的缓存块, 它在本地总线发送请求察看邻近 CPU 的缓存中是否有该数据块。如果有, 执行缓冲区到缓冲区 (存储器到缓冲区) 的传送。如果块的状态为 CLEAN, 属主所在簇的其他所有拷贝必须被置无效。

数据块的位置

块状态	R 的缓存	邻近者缓存	属主所在簇的存储器	簇缓存
未缓存			发送块到 R; 标记为干净, 仅在 R 簇的缓存中	
干净	使用块	复制块到 R 的缓存中	从存储器到 R 拷贝块; 标记表明仍然在 R 簇的缓存中	
脏	使用块	发送块到 R 和属主所在簇, 通知属主所在簇将其标记为干净并在 R 簇的缓存中		发送块到 R 和属主所在簇 (如果在其他地方缓存过), 通知属主所在簇将其标记为干净且它在 R 簇的缓存中

(a)

数据块的位置

块状态	R 的缓存	邻近者缓存	属主所在簇的存储器	簇缓存
未缓存			发送块到 R; 标记为脏, 只有在 R 簇中才缓存	
干净	发送信息给属主所在簇, 请求脏状态的唯一拥有权; 如果被允许, 使用块	复制块并使其无效, 发送信息给属主所在簇, 请求脏状态的唯一拥有权	发送块给 R; 将所有的缓存置无效; 将它标记为脏并只存在于 R 簇的缓存中	
脏	使用块	缓存到缓存地传送到 R 中, 将邻近的拷贝置无效		直接将块发送给 R; 将缓存置无效; 属主所在簇将它标记为脏并只存在于 R 簇的缓存中

(b)

图 6-7 Dash 协议。 (a) 读

(b) 写

说明: 列指出数据块的位置, 行指出所处状态。方框中的内容指出执行动作。R 指请求 CPU, 空框表明这种情况不可能

如果本地查找失败, 数据块在其他地方, CPU 发送信包到其属主所在簇。这里有三种情况: 如果块为 UNCACHED, 标记为 DIRTY 并发送给请求者。如果块为 CLEAN, 所有拷贝置无效, 然后执行上述操作。如果块为 DIRTY, 请求传送到拥有该数据块拷贝的远程簇 (需要的话), 该簇将自己的拷贝置为无效并满足请求。图 6-7(b)说明了这几种不同情况。

显然, 在 Dash(或其他大型多处理机)中维持存储器一致性完全不同于图 6-1(b)的简单模型。一次简单的存储器访问也可能要求传送大量信包。而且, 为保持存储器一致, 所有信包被确认后访存才能结束, 这极大地影响了性能。Dash 采用几种特殊技术避免这个问题: 如两套簇间链, 管道写入, 存储器多重语义。以后我们还将继续讨论这些问题。现在, 共享存储器的实现需要大型的数据库 (目录), 相当大的计算能力 (目录管理硬件), 可能的大量请求和确认信包。后面我们将看到, 实现分布式共享存储器也有同样的特点。两者区别在于实现的技术, 而不是思想、结构或算法。

6.2.5 NUMA 多处理机

我们现在可以十分清楚地看到在大型多处理机上实现硬件缓存并不简单。硬件必须维护复杂的数据结构, 像图 6-7 那样复杂的协议必须放入缓冲控制器或 MMU 中。无疑, 这样的大型多处理机价格昂贵, 不可能得到广泛应用。

然而, 研究人员已经花费大量的精力努力寻求不需要复杂缓存机制的替代设计方案。其中一种结构就是 NUMA (NonUniform Memory Access) 多处理机。和传统 UMA (Uniform Memory Access) 多处理机一样, NUMA 机的虚拟地址空间对所有 CPU 都可见。任何 CPU 在地址 a 写入值, 接下来别的处理机对 a 的读操作将读取刚刚写入的值。

UMA 与 NUMA 的区别不是在语义上, 而是在性能上。在 NUMA 机上, 访问远程存

存储器要比访问本地存储器慢得多，而硬件缓存并不试图掩盖这个问题。访问远程与访问本地的比例通常为 10:1。尽管 CPU 可以直接运行驻留在远程存储器的程序，但比运行驻留本地的程序慢得多。

NUMA 机的例子

为了更清晰地理解 NUMA 的概念，我们来看看图 6-8(a)所示的例子。Cm*是第一台 NUMA 机 (Jones 等,1977)。Cm*包含多个簇，每簇包含一个 CPU、一个可执行微程序的 MMU、一个存储器模块和一些需要的输入输出设备，这些均以总线相连。其中没有缓存，也没有总线监听。簇由簇间总线相连，图中只显示了一条。

当 CPU 访问存储器时，请求到达 CPU 的 MMU，MMU 检查高位地址以确定需要哪块存储器。若为本地地址，MMU 仅在本地总线发送请求。如果是远程存储器，MMU 建立一个包括地址的请求信包（写请求还包括要写入的数据）。通过簇间总线发送到目的簇。目的 MMU 收到信包后，执行要求的操作并返回字（读操作）或确认信息（写操作）。尽管 CPU 可以完全在远程存储器中运行，但通过发送信包读写每个字大大减慢了操作速度。

图 6-8 (b) 说明了另一种 NUMA 机制即 BBN 蝶形算法。在这种设计中，每个 CPU 和一块存储器直接相连。图中每个小方块代表一个 CPU 和一个存储器对。图中右边的 CPU 与存储器对和左边的相同。它们通过 8 个交换器相连，每个交换器有 4 个入口，4 个出口。本地访问的请求直接完成。如要访问远程存储器，发送请求信包，通过交换网络将信包送入相应存储器。这里，程序也可以在远地运行，性能相对较好。

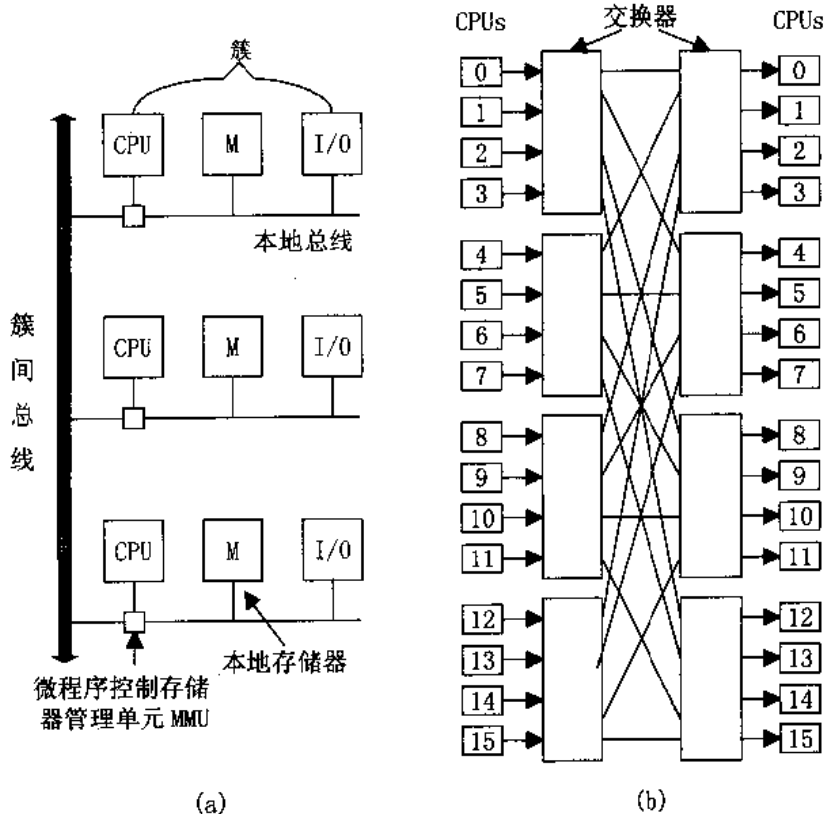


图 6-8 (a) Cm*系统的简单表示
(b) BBN 蝶形.右边的 CPU 对和左边的完全一样(也就是说,体系结构是一个真正的柱面)

这两例中都没有全局存储器。NUMA 机器还可以配备不与任何 CPU 相连的存储器。例如, Bolosky 等人(1989) 就描述过一种基于总线的 NUMA 机, 它的全局存储器不与任何 CPU 相连, 但可以被所有 CPU 访问 (作为本地存储器的补充)。

NUMA 多处理机的属性

NUMA 机器有我们关心的三个主要属性:

- (1) 可以访问远程存储器;
- (2) 访问远程存储器比访问本地存储器慢;
- (3) 没有缓冲机制隐藏访问远程存储器的时间。

前两点比较容易解释。第三点需要些说明。在 Dash 和大多数其他现代的 UMA 多处理机中, 访问远程存储器也比本地慢。缓存的存在使这一点可以忍受。当访问远程的字时, 它所在的存储器块都将移入提出请求的处理机的缓存中。因此, 以后的访问是高速的。尽管处理缓存错误时可能会耽误一些时间, 但访问远程存储器只在某些时候比访问本地存储器代价高。这一发现的结果是页处于哪块存储器并不重要: 代码和数据由硬件自动移入所需地方 (尽管在 Dash 中, 如果错误地选择了属主所在簇, 就会导致额外的开销)。

NUMA 没有这一特性, 因此将页放在哪块存储器 (即哪个机器上) 是很重要的。NUMA 软件的一个重要问题就是决定将每页放在何处以寻求最好的性能。下面我们将简单介绍一些 LaRowe 和 Ellis (1991) 的思想。Cox 和 Fowler (1989); LaRowe 等, (1991) 及 Ramanathan 和 Ni (1991) 描述了其他的研究。

运行 NUMA 机器上的一个程序时, 页可能被预置到也可能不被预置到某一特定处理机的机器上 (它们的属主处理机)。在任何一种情况下, 当 CPU 试图访问一个还没有调入的页时, 出现缺页错误。操作系统俘获错误, 并根据不同的情况作出相应的处理。若页为只读, 复制该页 (即在本机复制该页而不改变初始页) 或将虚拟页映射到远程存储器, 从而实现该页中所有地址的远程访问。若页为可读写的, 则将该页移入出错处理机 (使初始页无效) 或将虚拟页映射到远程存储器。

这里涉及的方案很简单。若本地拷贝 (复制或移入) 了某页且该页没有经常使用, 则取页所需的时间就大量的浪费了。另一方面, 若不作拷贝而将页映射到远程存储器, 随后的许多访问存储器操作将缓慢下来。因此, 操作系统必须猜测该页今后是否会被多次使用。若猜错, 系统性能将受到影响。

不管作出怎样选择, 页都将映射到本地或远程, 出错的指令重新开始。接下来对那页的访问由硬件来做, 无需软件介入。但若再无其他操作, 则操作系统的决策错误将永远不能更正。

NUMA 算法

为允许改正错误, 使系统适应访问模式的改变, NUMA 系统都有一个称为页面扫描器的 daemon 进程在后台运行。页面扫描器在硬件的帮助下, 周期性地 (例如每 4 秒) 收集有关本地和远程访问的使用数据。每运行 n 次, 页面扫描器重新估算先前选择的策略: 拷贝页或映射到远程存储器。若使用统计表明页放错位置, 页面扫描器就将取消该页映射, 下一次访问存储器而引起缺页错误时, 将页面重置。若在短间隔内该页频繁移动, 该页会被冻结。除非发生特别事件 (例如几秒钟过去以后), 禁止页移动。

许多策略可以用于 NUMA 机器, 区别在于算法: 或是扫描器 (scanner) 使用的页无

效算法,或是页错误之后的重置决策。一种可行的扫描器算法是将那些访问远程存储器多于本地存储器的页置为无效。一种有效测试是在扫描器运行最近 k 次内, 如果访问远程存储器的次数大大多于本地存储器的次数, 则将页置无效。另一种方法是若经过 t 秒或在一定间隔内访问远程存储器的次数在一定数量上多于本地存储器时, 解冻冻结页。

若发生页面错误, 有许多包括复制/移入或不包括复制/移入的算法可供选择。一种高级算法是当页非冻结时复制或迁移它。同时也应考虑最近的使用模式, 以及该页是否在属主机器上。

LaRome 和 Ellis (1991) 比较许多算法后认为没有一种措施是最好的。机器结构、远程存储器的页面大小及程序的访存模式都对算法选择很重要。

6.2.6 分布式共享系统的比较

共享存储器的系统包括很多种: 从完全由硬件维护一致性的系统到完全由软件来实现的系统。我们已经较详细地研究了由硬件维护一致性的系统, 并简要地讨论了完全由软件来实现的系统(基于分页的分布式共享存储器和基于对象的分布式共享存储器)。图 6-9 详细地表示了这一系列系统。

图 6-9 的左边是单总线式多处理机, 它有硬件缓存, 并通过监听总线保持存储器一致, 这是最简单的共享存储器机制, 并且完全由硬件完成。由 Sequent 和其他供应商提供的各种机制以及试验性的 DEC Firefly 工作站也在这一类中。它对于中小规模数量的 CPU 运行适宜, 但总线饱和后性能很快下降。

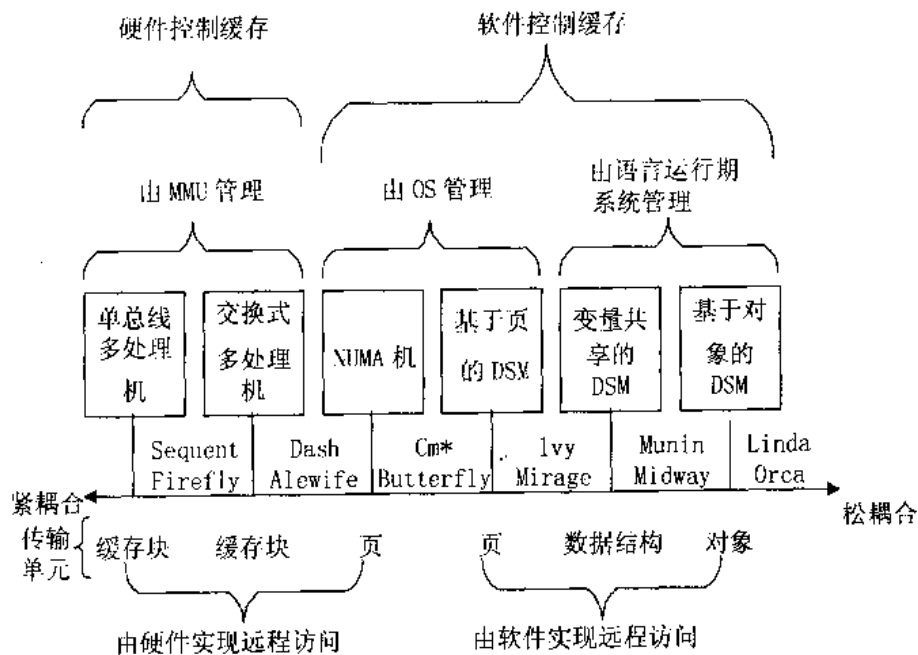


图 6-9 共享存储器机器的范围

接下来是交换式多处理机, 如斯坦福的 Dash 机和 M.I.T. Alewife 机。它们也有硬件缓存, 但是通过目录和别的数据结构来记录哪个 CPU 或簇保存哪个缓存块。为维护存储器

一致性使用了许多复杂算法。但由于它们都以 MMU 微码保存（用软件处理的例外），它应该属于硬件实现的技术。

下面是 NUMA 机制。它界于硬件控制和软件控制之间。像在多处理机中一样，每个 NUMA 的 CPU 可以通过读写操作访问普通虚拟地址空间的所有字。不同于多处理机中，缓存（即页的放置和移入）是由软件（操作系统）而不是由硬件（MMU）来控制的。Cm* (Jones 等, 1977) 和 BBN 蝶形机制就是 NUMA 的例子。

下面是基于分页的分布式共享存储器系统，如 IVY (Li, 1986) 和 Mirage (Fleisch 和 Popek, 1989)。在这样的系统中，每个 CPU 有自己私有存储器，但是不同于 NUMA 和 UMA 机制，它们不能直接访问远程存储器。若 CPU 访问的字所在的地址空间在另一台机器的页上时，操作系统激活陷阱中断，将由软件来获取所需页。操作系统通过发送信包从该页驻留的机器上获取所需页。这样，定位和访存都是由软件完成。

下面我们来看看这样的机器，它们仅共享地址空间中被选择的一部分，也就是共享变量和其他数据结构。Munin (Bennett 等, 1990) 和 Mideay 机 (Bershed 等, 1990) 就是这样的。用户必须提供信息以确定哪些变量共享，哪些不需共享。在这个系统中，重点不再是体现普通单一的存储器，而是如何维持一系列复制的分布式数据结构的一致性，有可能是在所有共享数据的机器上执行更新。在一些情况下，分页硬件能检测到写操作，这有助于有效地维护系统一致性；在另一些情况下，分页硬件不用来管理一致性。

最后，我们讨论基于对象的分布式共享存储器系统。和以上几种机制不同的是，这里的程序不能直接访问共享数据。它们必须通过受保护的方法来访问，这意味着运行期可以控制每次访问以维护系统一致性。系统完全由软件实现，没有任何硬件支持。Orca (Bal, 1992) 是这种设计的样例，Linda (Carriero 和 Gelernter, 1989) 在一些重要方面与其有类似之处。

表 6-2 总结了这 6 种系统的区别，左边是紧耦合的硬件，右边是松耦合的软件。前 4 种提供的存储器模式包含标准存储器、分页存储器、虚拟地址空间存储器。前 2 个为通常的多处理机，接下来的 2 个与它们类似。前 4 种机器类似于多处理机，唯一的可能操作就是读写存储器字。第 5 个使用了特别的共享变量，但仍是通过通常的读写来访问。基于对象的系统通过封装的数据和方法，可提供更多的操作，并比自然状态的存储器有着更高层次的抽象。

多处理机与 DSM 系统的实际区别在于是否能够通过地址直接访问到远程数据。所有的多处理机系统都是可以的，而 DSM 不行：它总需要软件支持。同样的，独立的全局存储器，即未与任何 CPU 相连的存储器在多处理机中是允许的，而 DSM 系统不行（因为后者是通过网络互连的分散的计算机的集合）。

在多处理机系统中，当发现 CPU 访问远程存储器时，缓存控制器或 MMU 发送消息给远程存储器。而 DSM 系统中由操作系统或运行时系统发送消息。它们所采用的介质也不一样。多处理机采用高速总线（或总线集），而 DSM 采用传统的 LAN（虽然有时总线和局域网的区别是含糊的，区别主要与金属线的数量有关）。

表 6-2 六种共享处理器系统的比较

项	多处理机			DSM		
	单总线	交换	NUMA	基于页式	共享变量	基于对象
是线性、共享虚拟地址空间吗?	是	是	是	是	否	否
可能的操作	读/写	读/写	读/写	读/写	读/写	常规
有封装及方法吗?	否	否	否	否	否	是
是否由硬件执行远程访问?	是	是	是	否	否	否
有独立存储器吗?	是	是	是	否	否	否
谁将远程存储器的访问转化为消息?	MMU	MMU	MMU	操作系统	运行期系统	运行期系统
传输介质	总线	总线	总线	网络	网络	网络
由谁执行数据迁移	硬件	硬件	软件	软件	软件	软件
传输单元	块	块	页	页	共享变量	对象

还有一个问题是：需要时由谁来作数据迁移。这里 NUMA 机制和分布式共享存储器系统类似：两者都由软件实现而不是硬件，硬件负责在机器之间移动数据。最后，六个系统的数据传输单元不同。UMA 多处理机是缓存块，NUMA 机和页式 DSM 系统是页，后两种采用变量或对象。

6.3 一致性模型

尽管现代的多处理机和分布式共享存储器系统有很多共性，我们仍将离开多处理机的话题向前更进一步。本章的开始我们简略介绍了 DSM 系统，在 DSM 系统中，每个只读页可以有多个拷贝，而每个可写页只能有一个拷贝。在最简单的实现方案中，机器远程访问一可写页会激活陷阱，然后获取该页。由于可写页经常共享，因此只允许有一个拷贝可能会带来严重的性能瓶颈。

只要能更新所有拷贝，允许多重拷贝就可以简化性能问题。但这又引起了新的问题：即如何保证所有拷贝的一致性。当拷贝分布于不同机器上，而机器间通信只能通过慢速的（和存储器速度相比）网络发送信包来完成时，维护绝对的一致性就显得尤其困难。在一些 DSM（以及多处理机）系统中，解决办法是为了高性能而降低一致性的要求。一致性究竟意味着什么？多大程度上的不一致是程序可以接受的？这是 DSM 研究者面临的主要问题。

一致性模型本质上是软件与存储器间的协约问题（Adve 和 Hill,1990）。它指的是，如果软件遵守约定的规则，存储器就能工作正常。如果软件违反了这些规定，存储器就不再保证操作的正确性。研究者已经设立了很大范围的一系列约定，其中有些对软件只有少量限制，而有些则使普通编程几乎成为不可能。当然，限制少的模型没有限制多的模型执行效果好。这正如同生活本身。本节我们主要学习 DSM 系统中的各种一致性模型。更多的信息可以参阅 Mosberger 的文章（1993）。

6.3.1 严格一致性 (Strict Consistency)

最严格的一致性模型称为严格一致性，它由下述条件定义：

从存储器地址 X 处读出的值为最近写入 X 的值。

这个定义自然而明白，由于它假定了绝对全局时间（就像在牛顿物理中）的存在，“最近”访问的意义是明确的。传统意义上，单一处理机遵守严格一致性，单处理机的编程者正希望这样。在一系统中编程如下：

```
A=1; A=2; PRINT (A);
```

打印 1 或 2 以外的任何值将很快引起编程人员们的热烈讨论（本章中，*print* 是一个打印其参数的过程）。

在 DSM 系统中，问题更为复杂。假设 X 是存在 B 机器上的变量。 A 机器上的进程在 T_1 时刻读取 X ，即发送信包到 B 以读取 X 。稍后，在 T_2 时刻， B 机器上的进程写 X 。若遵守严格一致性，不管机器在哪里，也不管 T_1 和 T_2 相距多远， A 都应该读出原来的值。然而，若 $T_2 - T_1 = 1\text{ns}$ ，而机器距离 3 米，从 A 到 B 传送读操作并使之先于写操作，信号则必须以十倍光速的速度传递，而这与爱因斯坦相对论矛盾。编程人员有理由在违反了物理原则的情况下要求严格一致性吗？

这就给我们带来了存储器和软件的协约问题。若协约或明显或暗示地要求严格一致性，存储器则最好传送它。另一方面，确实希望严格一致性的编程人员，当他不在现场时他的程序出了错，他就会面临危险。即使在小型多处理机上，一处理机在 A 处写值，一纳秒后另一处理机读 a ，也很可能在本地 cache 中读出原来的值。在这种情况下编程失败者应该回到学校继续学习，并牢记这一点。

还有一个更实际的例子。我们可以想像这样一个系统，它为体育迷提供世界范围的体育事件的比分，比分以分为单位更新（而不用精确到纳秒）。在这种情况下，前后相差 2 纳秒的查询是可接受的，尤其当它允许数据多重拷贝时会带来更为优越的性能。这里，并不保证传送、或需要的严格一致性。

为深入研究一致性，我们将给出很多例子。为使举例精确，我们需要一些特定符号。举例如下。先规定一些符号： P_1, P_2, \dots 代表不同的进程，在图中以不同的高度表示，每一进程完成的操作水平的显示时间轴向右增加，直线分割不同进程。符号 $W(X)a$ 和 $R(Y)b$ 分别表示在 x 处写 a 和从 y 处读出值赋给 b 。本章中所有图表中变量初值均为 0。如图 6-10(a)， P_1 在 x 处写 1 之后， P_2 读 x ，返回 1。对于严格一致性存储器，这个操作是正确的。

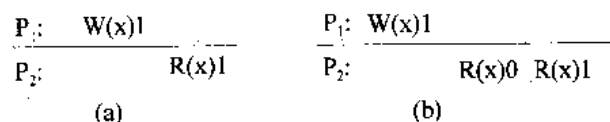


图 6-10 两个进程的动作。横轴时间

(a) 严格一致存储器

(b) 不严格一致存储器

相反的,如图 6-10(b), P_2 在写之后读(或许是 1 纳秒之后,但仍在写之后),返回值为 0。之后又执行一次读操作,返回值为 1。对于严格一致性的存储器,这个操作是不正确的。

总之,对于严格一致性的存储器,写操作在任一时刻对所有进程都是可见的,同时维护一个绝对全局时间。一旦存储器中的值改变,不管读写之间的事件间隔多小,不管哪个进程执行读操作,也不管进程在何处,以后读出的都是新更改的值。同样,不管后面的写操作有多迅速,执行读操作仍应读出原来的值。

6.3.2 顺序一致性(Sequential Consistency)

尽管严格一致性是理想的编程模式,但在分布式系统中,这几乎不可能实现。经验表明,编程人员通常对弱模式掌握得较好。比如,所有操作系统课本中都谈到临界区和同步互斥的问题。人们认为编写这种正确的并行程序(如生产者—消费者问题)不应考虑进程间的相对速度以及语句的执行在时间上如何交错。如果一个进程的两个事件发生如此之快,以致其他进程不能在此之间执行任何操作,那可能会带来麻烦。相反,读者的编程方式是:语句执行顺序(实际上是存储器访问)并不重要。如果事件顺序是必要的,必须要用到信号量或其他同步操作。接受这种意见意味着采用弱模式。经过几次实践,大多数并行程序编写人员都能适应这种模式。

顺序一致是比严格一致稍弱的存储器模式,Lamport (1979)首先定义了顺序一致存储器应符合的条件:

如果所有进程以一定的顺序执行操作,每一进程的操作都以程序规定的顺序出现,则任何操作的结果都是一样的。

这个定义说明:对于在不同机器上(甚至在伪并行的分时系统上)并行运行的进程,任何有效的交错都是可以接受的行为,但所有进程必须遵守同一访问存储器顺序。在一块存储器中,若一个进程(或处理机)看到一种交错,另一进程看到另一个交错,这就不是顺序一致存储器。注意,这与时间无关,没有最近存入的概念。在这里,进程可以看到所有进程写,但只能看到本进程读。

从图 6-11 可以看出这里不考虑时间。图 6-11(a)表示的存储器行为是顺序一致的,尽管 P_2 执行的第一次读操作返回初值 0 而不是新值 1。

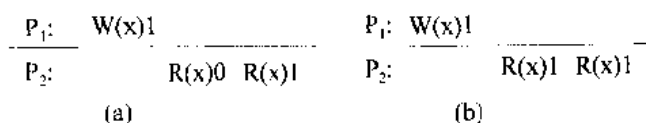


图 6-11 运行同一程序得到的两个可能的结果

顺序一致存储器不保证读返回的值是 1ns、1ms 甚至 1 分钟以前另一进程写入的。它只保证所有进程以相同的顺序看见存储器访问。如果根据图 6-11(a)编写的程序在一次运行后,或许会得到图 6-11(b)的结果。结果是不确定的。如果缺少明确的同步操作,在此运行程序或许不能得到相同的结果。

a=1; print(b,c);	b=1; print(a,c);	c=1; print(a,b);
(a)	(b)	(c)

图 6-12 三个并行进程

为使这点更为明确，我们考虑一下图 6-12 的例子 (Dubois 等,1988)。我们可以看到三个并行运行的进程的代码。三个进程共享相同的顺序一致分布式共享存储器，并都访问变量 a, b, c 。从存储器访问的角度看，赋值应被看作写操作，打印应被看作同时读取它的两个参数。所有语句都是原子操作。

各种交错执行的顺序都是可能的。六个独立的语句，将会有 720 (6!) 种可能的执行顺序，尽管它们可能会破坏程序顺序。从 $a=1$ 开始的考虑的顺序有 120 (5!) 种。其中一半将在 $b=1$ 前执行 $print(a, c)$ ，这就打乱了程序次序。还有一半将在 $c=1$ 前执行 $print(a, b)$ ，也打乱程序次序。120 中顺序中只有 1/4 即 30 个是有效的。另外 30 个有效顺序是以 $b=1$ 开头的，还有 30 个以 $c=1$ 开头。共有 90 个有效执行顺序。图 6-15 表示出 4 种。

如图 6-13(a)，三个进程以 P_1, P_2, P_3 的次序运行，另三个例子则不同，但同样是有有效的，语句按时间交错执行，三个进程都打印两个参数。由于每个变量只可能是初值 0 或被赋值为 1，每个进程产生一个二位字串。在 $prints$ 操作之后的数字是在输出设备上的实际输出。

a=1; print(b,c); b=1; print(a,c); c=1; print(a,c);	a=1; b=1; print(a,c); print(b,c); c=1; print(a,b);	b=1; c=1; print(a,b); print(a,c); a=1; print(b,c);	b=1; a=1; c=1; print(a,c); print(b,c); print(a,b);
Prints:001011	Prints:101011	Prints:010111	Prints:111111
Signature:00101	Signature:101011	Signature:110101	Signature:111111
(a)	(b)	(c)	(d)

图 6-13 在图 6-12 中的四个有效的执行序列。竖轴方向是时间轴，自上往下增长

若以这种次序顺序输出 P_1, P_2, P_3 的结果，将得到六位字串，这标明了语句（以及存储器访问）的一个特定交错。这就是图 6-13 中的标记项 (Signature) 所写的字符串。下面可以标记项而不用打印输出表示每个次序。

并非所有 64 种标记项都是允许的。如 000000 就不允许，因为这意味着打印语句先于赋值执行，违反了 Lamport 的关于语句必须按照程序顺序执行的规定。另一微妙的例子是 001001，前两位 00 即当 P_1 打印时， b, c 都是 0。这种情况只发生在 P_1 在 P_2, P_3 开始

前执行所有语句。下面三位 10, 即 P_2 必须在 P_1 之后 P_3 之前开始执行, 最后两位 01, 即 P_3 必须在开始之前完成。但我们已经看到 P_1 必须首先运行, 因此 001001 是不可能的。

总之, 在假定顺序一致的情况下, 90 种不同的有效语句次序导致了不同的 (但少于 64 种) 运行结果, 这些都是允许的。软件与存储器在这里的约定是: 软件必须承认它们都是有效结果。也就是软件必须接受图 6-13 的 4 个结果以及其他有效结果为正确答案。若任何一种情况发生, 这就仍须正常工作。有些程序只能在部分结果下正常工作, 而在另外的结果下不行, 这就没有遵守与存储器的约定, 是不正确的。

顺序一致存储器可在 DSM 或多处理机系统上实现, 在保证前一存储器操作完成后才开始后一存储器操作时, 它复制可写页。例如, 在一个有效全序的广播机制系统中, 可以将所有共享变量组合在一页或更多页上。对共享页的操作可以广播发送。只要所有进程遵守在共享存储器上的所有操作顺序, 操作交错的确切次序就显得不重要了。

有很多正式系统使用了顺序存储器 (和其他模式) 思想。让我们简单看一下 Ahamad 等 (1993) 的系统。在这种系统中, 进程 i 读写操作顺序由 $H_i(P_i)$ 的历史) 确定, 图 6-10b 显示两种次序 H_1 和 H_2 , 分别对应于 P_1 和 P_2 , 表示如下:

$$\begin{aligned} H_1 &= W(x)1 \\ H_2 &= R(x)0 R(x)1 \end{aligned}$$

这种次序的集合称为 H 。

为得到操作执行的相对顺序, 必须合并 H 中的操作中为一个单独串 S , 这样 H 中每个操作在 S 中出现一次。 S 直观地给出了在假设单一集中存储器的条件下, 操作本应该执行的顺序。 S 的合法值须遵守两个限制:

- (1) 维持程序次序;
- (2) 保证存储相关性 (Memory coherence)。

第一条限制即: 如果在 H 的一个字串中, 读或写访问 A 出现在另一访问 B 之前, 那么 S 中 A 也应出现于 B 之前。若所有操作遵守这个规定, 则 S 中不会出现违反程序的执行顺序。

第二条限制称为存储相关性, 这意味着在地址 x 处读出的值必须是最新写入的 x , 即在 $R(x)$ 之前由最近的 $W(x)v$ 写入的值 v 。存储相关性不考虑别的位置, 检查的是每一个孤立位置以及作用在其上的操作顺序。相反, 一致性处理不同地址上的写操作以及他们的顺序。

图 6-10(b)中, S 只有一个合法值:

$$S = R(x)0 W(x)1 R(x)1,$$

对于更复杂的例子, S 可能会有更多的合法值, 若操作顺序是按照 S 的一些合法值进行的, 则程序运行被认为是正确的。

虽然顺序一致性是对编程人员友好的模式, 但它有严重的性能问题。Lipton 和 Sandberg (1988) 证明, 若读时间为 r , 写时间为 w , 节点间信包传输最小时间为 t , 则总有 $r+w \geq t$ 。换言之也就是对所有顺序一致存储器来说, 改变协议试图提高读性能只能导致写性能下降, 反之亦然。因此, 研究者研究了其他 (更弱) 模式。下面几节, 我们将讨论其中

的一些。

6.3.3 因果一致性(Causal Consistency)

因果一致模型(Hutto 和 Ahamad, 1990)是顺序一致的淡化, 它按有无可能的因果联系区分各事件。

为了说明因果的含义, 我们看看生活中的一个例子 (关于计算机科学家)。在一次关于不同编程语言的相对优点的 USENET 新闻组讨论中, 有一些头脑发热的人发消息: 用 FORTRAN 编程的人该杀。很快, 另一冷静的人写道: 即使违反了好的品位, 我也反对死刑。由于不同传输路径的不同延时, 第三个订阅者可能先看到回答的消息而觉得迷惑, 这里的问题是破坏了因果关系。如果较早事件 A 引起或影响了 B , 因果性要求大家先看到 A , 再看到 B 。

现在看看存储器的例子。假设进程 P_1 写变量 x , 然后 P_2 读出 x , 写入 y 。这里读出 x 和写入 y 之间可能有潜在的因果联系, 因为 y 的计算很可能决定于 P_2 读到的 x 值 (即 P_1 写入的值)。另一方面, 若两进程自然而同时地写两个变量, 就没有因果联系。先有读操作之后执行写操作, 两个事件就可能有因果联系。相似的, 读和提供所读数据的写有因果关系。没有因果关系的操作称为并发的 (concurrent)。

因果一致的存储器应遵守以下条件:

可能因果相关的写操作应对所有进程可见, 且顺序一致。并发写操作在不同机器看来顺序可能是不同的。

图 6-14 是一个因果一致性的例子。这里我们有一个在因果一致存储器下允许的事件顺序, 在顺序一致存储器和严格一致存储器中这是禁止的。要注意的是, $W(x)2$ 和 $W(x)3$ 是并发的, 所以不须所有进程将他们视为同一顺序。若不同进程以不同次序看待并发事件, 而导致软件失败, 则违反了因果存储器协议。

现在看一下第二个的例子。图 6-15(a)中, $W(x)2$ 可能决定于 $W(x)1$, 因为 2 可能是 $R(x)1$ 所读的值计算的结果。两个写操作是因果联系的, 所有进程必须视它们为同一顺序。因此 6-15(a)不正确。6-15(b)中, 读被去掉, $W(x)1$ 和 $W(x)2$ 变为并发事件。因果一致性存储器不要求并发写有全局一致的次序, 因此 6-15(b)是正确的。

P_1 :	$W(x)1$	$W(x)3$		
P_2 :	$R(x)1$	$W(x)2$		
P_3 :	$R(x)1$		$R(x)3$	$R(x)2$
P_4 :	$R(x)1$		$R(x)2$	$R(x)3$

图 6-14 因果一致性存储器允许的序列, 但是顺序一致存储器和严格一致性存储器不允许

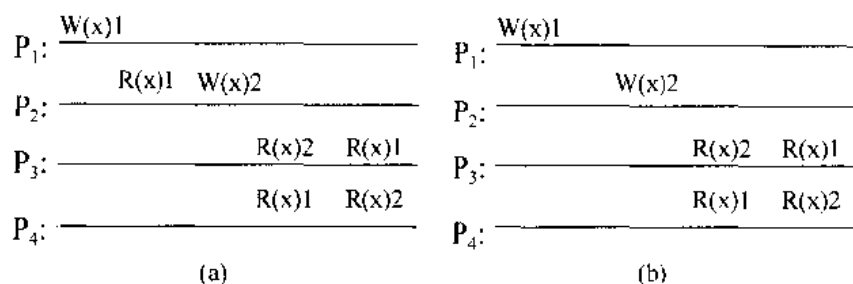


图 6-15 (a) 因果一致存储器的反例

(b) 因果一致存储器中的正确事件序列

实现因果一致性需要由记录来跟踪哪个进程看到哪个写操作。这要建立和维护一个依赖图：即一个操作依赖于其他什么操作。这样做需要一些额外的开销。

6.3.4 PRAM 一致性(PRAM Consistency)和处理器一致性(Processor Consistency)

在因果一致性存储器中，所有机器看到的因果相关写操作的顺序是一致的，而并发写操作的顺序可以不同。进一步放松对存储器限制可以去掉前面的要求。这样就给出了 PRAM 一致（管道 RAM），它要遵守的条件是：

一个进程的写操作可以被其他进程以指定的顺序接收到，但不同进程的写操作在不同进程看来次序可以是不同的。

PRAM 一致性由 Lipton 和 Sandberg（1988）提出，PRAM 代表管道 RAM，由于一个进程的写操作可以是流水线的，即进程不必在开始下一个操作之前闲等待一操作结束。PRAM 一致和因果一致的对比如图 6-16 所示。这里的时间顺序在 PRAM 一致的存储器中是允许的，而在其他更强的模式下都不允许。

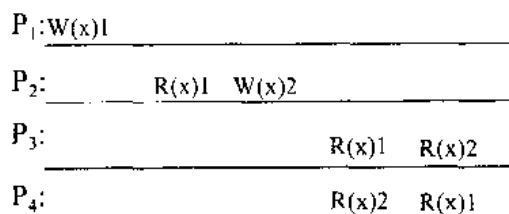


图 6-16 PRAM 一致性的有效事件序

PRAM 一致由于易于应用而得到关注。事实上它不保证不同进程看到的写操作顺序是一致的，除非是一个源的一个或多个写操作，才必须按次序到达，就好像在管道中。换言之，在这种模式中，由不同进程产生的写操作是并发的。

再来看图 6-17，在 PRAM 一致下，不同进程所看到的语句执行顺序不同，如图 6-17(a)显示 P₁ 怎样看到事件，而 6-17(b)显示 P₂ 所看到的，6-17(c)则是 P₃ 所见。对于顺序一致存储器，三个不同显示是不允许的。

a=1;	a=1;	b=1;
*print(b,c);	b=1;	print(a,c);
b=1	*print(a,c);	c=1;
print(a,c);	print(b,c);	print(a,b);
c=1	c=1;	a=1;
print(a,b);	print(a,b);	print(b,c);
Prints:00	Prints:10	Prints:01
(a)	(b)	(c)

图 6-17 三个进程的语句执行情况,用星号标记的语句产生实际输出

我们使三个进程的输出顺序相接,得到结果为 001001,正如我们早先看到的,这在顺序一致性下是不可能的。顺序一致与 PRAM 一致的关键不同在于:前者尽管未确定语句(和存储器访问)的执行顺序,但至少所有进程都遵守共同的顺序。后者就不遵守。不同进程看到的操作顺序不同。

有时,PRAM 一致导致的结果不那么直观。在下面的例子中,Goodman (1989)提出了一种略微不同但仍支持 PRAM 一致的存储器形式(下面将加以讨论)。如图 6-18,有人可能会天真地期待出现三种结果之一: P_1 被 KILL; P_2 被 KILL;或两者都不被 KILL(如果两个赋值语句先执行了)。在 PRAM 一致下,可能两个进程都被 KILL,即若 P_1 在看到 P_2 中 b 赋值之前读取 b , P_2 在看到 P_1 中 a 赋值之前读取 a 。在顺序一致存储器下可能有 6 中语句交错,没有一种可以导致两进程都被 KILL。

a=1;	b=1;
if (b==0) kill (P2);	if (a==0) kill (P1);
(a)	(b)

图 6-18 两个并行的进程。(a) P_1 (b) P_2

Goodman (1989) 模式叫作处理机一致性 (processor consistency), 它与 PRAM 一致如此接近,以至于有的作者认为他们实际是一样的(如, Attiya 和 Friedman, 1992; 以及 Bitar, 1990)。Goodman 通过一个例子证明在处理器一致的存储器还有一个附加条件,即存储相关性。如上所述,对于任意存储器地址 x ,对于写入 x 的顺序有个全局约定。写入不同地址,对于不同进程来看,不需要相同顺序。Gharachorloo 等(1990)描述了在 Dash 多处理机上使用了处理器一致。但他与 Goodman 的定义有些微不同。PRAM 和两个处理器一致模型的区别是很小的, Ahamad 等(1993)描述了这一点。

6.3.5 弱一致性 (Weak Consistency)

尽管 PRAM 一致性和处理器一致性的性能比强模式的更好,但它仍对很多应用程序作

了不必要的限制,即要求一个进程的写操作以一定顺序被所有进程看到。并非所有应用程序要看到所有写操作。比如在临界区中一个进程循环读写变量的例子。即使直到一个进程已离开临界区后其他进程才能访问变量,存储器也无从得知何时进程在临界区,何时不在。因此它必须以普通方法将所有写操作传播到所有存储器。

、更好的解决办法是让进程完成临界区操作以后,将最后结果发送到各处,而不用太关心甚至不用关心中间结果是否也顺序发送到所有存储器。这一切都可通过引入一种新变量:同步变量来解决,它用于同步的目的。对它的操作用于同步存储器。一旦完成一次同步,那台机器上所有写操作就均被广播出去,其他机器上的所有写操作都被引入,即所有(共享)存储器被同步了。

Dubois 等(1986)定义这种模式为弱一致性,它有三个属性:

- (1) 对同步变量的访问是顺序一致的;
- (2) 在所有先前的写操作完成之前,不能访问同步变量;
- (3) 在先前所有同步变量的访问完成前,不能访问(读或写)数据。

第一点指出所有进程以相同顺序看到对同步变量的访问。实际上,一个同步变量的访问将立即被广播出去。在广播发送结束之前,任意其他进程不能访问其他同步变量。

第二点指出访问同步变量“刷新了管道”,它强制所有程序中的写操作部分结束或在某些存储器上结束。同步访问开始前,应保证先前所有的写操作已完成。在更新共享数据后做同步操作,进程可将新值传遍所有存储器。

第三点指出访问一般(即非同步)变量,不管是读是写,只有在所有前序的同步操作结束后方可进行。在读共享数据前做同步操作,可以保证进程读到最新值。

需要指出的是:在 DSM 系统中,“执行完”这个词之后隐藏着相当的复杂性。当其后的写操作不会影响读的返回值时读操作被认为已执行,当所有后续读操作均返回刚写入的值时写操作被认为已执行。当更新了所有共享变量时同步被认为已执行。我们还可以比较在局部和全局执行操作的不同。Dubois 等(1988)详细讨论了这一点。

从应用的角度来说,软件和存储器间的协约指出只有当访问同步变量时,存储器才需更新;新的写操作可以在前一写操作完成之前开始,在一些情况下可完全避免写。当然,此协约加重了编程人员的负担,但换来了优越的性能。不同于以往的存储器模型,它是对一组操作的一致性约束,而不是单独的读或写。当很少单独而基本上以簇的形式(短时间内有很多访问,每一访问时间都不长)访问共享变量时,这个模型更为有用。

让存储器出错的思想并不是新近才提出的。许多编译器均搞欺骗。如图 6-19 中的程序片段,所有变量均初始化为合适的值。一个优化的编译器可以在寄存器中计算 a 、 b ,并在那里保存片刻,而不更新存储器,只有当调用函数 f 后才将 a 和 b 当前值返还存储器,因为 f 可能需要访问他们。

这里允许存储器出错是因为编译器知道它所做的(即因为软件不坚持存储器被更新)。很明显,若有另外一个进程可随意读取存储器的话,这种设计就不可行了。例如,在给 d 赋值时,另一进程读取 a 、 b 、 c ,可能会得到不一致的值(原来的 a 、 b 值,新的 c 值)。我们可以设想一种避免混乱的方法,即先让编译器写一标志位说明存储器没有更新。若另一进程访问 a ,它会在标志位上等待。在软件执行同步且各部分都遵守规定的条件下,使用这种方式可以完全保证一致性。


```

int a,b,c,d,e,x,y;      /*变量*/
int *p,*q;              /*指针*/
int f(int *p,int*q)     /*函数属性*/

a=x*x;                  /*存储在寄存器中*/
b=y*y;                  /*同上*/
c=a*a+a+b*b+a*b;       /*留待后用*/
d=a*a*c;                /*留待后用*/
p=&a;                    /*获取的地址*/
q=&b;                    /*获取的地址*/
e=f(p,q);               /*函数调用*/

```

图 6-19 程序片断中的一些变量保存在寄存器中

让我们考虑一个较牵强的例子。在图 6-20 (a) 中, P1 进程对一个普通变量写了两次, 然后执行同步 (如字母 S 所示)。若 P2、P3 还未同步, 则不保证它们看到的顺序, 所以这种事件顺序是允许的。

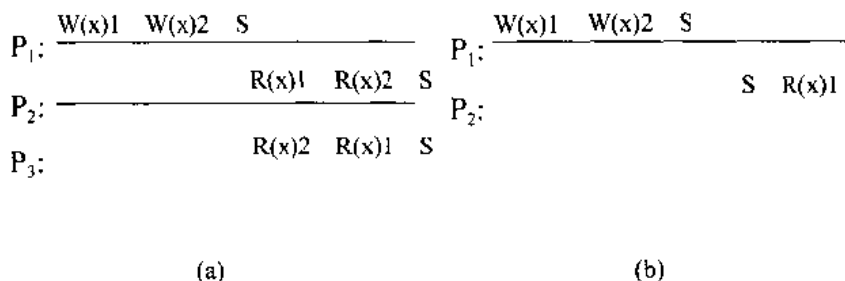


图 6-20 (a) 弱一致性事件有效序列

(b) 弱一致性事件无效序列

图 6-20 (b) 与此不同。P₂ 已被同步, 这意味着存储器已被更新。当它读 x 时, 读取的值应是 2。如图所示, 它得到 1, 这在弱一致性中是不允许的。

6.3.6 释放一致性 (Release Consistency)

弱一致性存在这样的问题, 即当访问同步变量时, 存储器并不知道这是因为进程已完成对共享变量的写操作还是要开始读共享变量。因此, 它必须执行这两种情况要求的操作, 也就是它要确保本地启动的写操作都已完成 (即发送到其他所有机器上), 同时还要收集其他机器执行的写操作。若存储器能够区分进入还是离开临界区的话, 应用起来会更有效。鉴于此, 我们需要两个而非一个同步变量或操作。

释放一致性 (Gharachorloo 等, 1990) 提供了这样两种访问: 获取 (acquire) 访问用于通知存储器系统临界区已就绪。释放 (release) 访问表明临界区刚退出。这些访问既可以通过特殊变量上的普通操作实现, 也可以作为特殊操作。在这两种情况下, 编程人员都需要在程序中编写明确的代码说明何时做这些操作, 例如, 是调用获取 (acquire) 和释放 (release) 这样的库过程还是进入临界区 (enter_critical_region) 和离开临界区 (leave_critical_region) 这样的过程。

在释放一致性下, 也可以使用屏障 (barrier) 代替临界区。屏障是一种同步机制, 在

所有进程完成程序的 n 段之前，它禁止任何进程开始 $n+1$ 段。当一个进程遇到屏障，它必须等待所有进程都运行到此处。当最后一个进程到达此处后，所有共享变量被同步，所有进程重新开始。从屏障离开的是获取访问，进入屏障是释放访问。

除了同步访问，读写共享变量也是允许的。获取和释放并不需要应用于整个存储器。相反，它们只是保护特殊共享变量，这时也只有那些变量保持一致。保持一致的共享变量称为被保护的。

存储器和软件的协约说明，当软件执行获取访问时，存储器会根据需要确保被保护变量的所有本地拷贝被更新，并和远程变量保持一致。当执行释放访问时，修改的被保护变量要传送到其他机器上。执行获取访问不保证本地所作的修改立即传送到其他机器上。相似的，执行释放访问不需要从其他的机器上引入所做的改变。

图 6-21 是释放一致性下的一个有效事件顺序。进程 P_1 执行获取访问，两次改变共享变量，然后执行释放操作。进程 P_2 执行获取访问，读取 x 。它应在释放访问时得到 x 的值为 2（除非 P_2 的获取访问先于 P_1 的）。若 P_2 的获取访问在 P_1 的释放访问之前，则它必须等待到执行了释放访问。因为 P_3 在读共享变量前没有执行获取访问，存储器不必给它 x 的当前值，因此返回 1 是允许的。

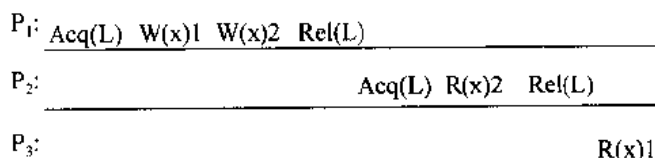


图 6-21 释放一致性的有效序列

为使释放一致性更为清晰，让我们简略地描述一个 DSM 中的简单应用（释放一致性实际上是为 Dash 多处理机而设计的，但尽管实现方法不同，基本思想是一样的）。为了执行获取操作，进程将消息发送给同步管理员，要求在一特殊的锁定上执行获取访问。在没有竞争时，请求获准，获取访问完成。对共享数据的读写就可以在本地开始了。它们不用传送到其他机器。当执行释放访问时，修改过的数据被传送到使用它们的机器上。当每个机器确认收到数据后，同步管理员被告知可以执行释放了。这样，就可以任意次数的读写共享变量，而开销是固定的。不同锁定上的获取和释放是各自独立进行的。

以上所述的集中算法是可行的但不是唯一的。通常，分布式共享存储器在遵守以下规定时就是释放一致的：

- (1) 在访问共享变量前，进程所有先前的获取访问都必须成功地完成；
- (2) 在允许释放访问前，进程先前的所有读写操作都必须结束；
- (3) 获取访问和释放访问必须是处理器一致的。

在上述情况下，如果进程正确使用获取和释放访问（即在获取-释放对时），任何操作都与在顺序一致存储器下的相同。实际上，为防止交错，获取和释放原语将共享变量的访问块变为原子的。

释放一致性的另一应用是懒惰释放一致性（lazy release consistency）（Keleher 等,1992）。通常的释放一致性，我们下面将称为急迫释放一致性(eager release consistency)，和它的区别在于变量是否为“懒惰”的，当执行了释放操作，执行此操作的处理机将所有

修改的数据传给所有那些已经有其缓冲拷贝且可能需要它的处理机。因为没有办法判断它们是否确实需要它,为安全起见,它们将获得所有修改过的数据。

尽管这样将所有数据传输出去的方法很直接,但通常不够有效。在懒惰释放一致性中,在执行释放时,不发送任何数据。相反,在执行获取操作时,处理机试图从拥有这些变量的机器上取得它们的最新值。时间戳协议可用来确定需传送哪些变量。

在许多程序中,临界区在一个循环中定位。在急迫释放一致性中,每经过一次循环,就执行一次释放访问,所有修改过的数据就要被发送到拥有其拷贝的所有机器上。这种算法浪费带宽,导致不必要的延时。在懒惰释放一致性中,执行释放访问时不做任何工作。在下次获取访问时,处理机发现它已有所需的所有数据,所以也不需要产生任何消息。除非其他处理机执行获取访问,使用懒惰释放一致性根本不产生网络流量。在没有外部竞争时,同一处理机执行的重复的获取-释放对是没有开销的。

6.3.7 入口一致性 (Entry Consistency)

另一种使用临界区的一致模型是入口一致性 (Bershad 等,1993)。和释放一致性的两个变体一样,它也需要编程人员或编译器在临界区的首尾分别使用获取和释放访问。然而,与释放一致性不同的是,入口一致性要求每个普通共享变量与同步变量如锁 (lock) 或屏障 (barrier) 保持一定联系。若并行访问数组元素,不同的数组元素就需要与不同的锁相联系。在对一个同步变量执行获取访问时,只保证与这个同步变量有关的共享变量一致。入口一致性与懒惰释放一致性不同在于,后者没有将共享变量与锁或屏障相联系,做获取访问时必须凭经验确认它需要哪些变量。

将每个同步变量与一组共享变量相连,可减少获取访问和释放访问一个同步变量时的开销,因为只有极少数共享变量需同步。它还允许包含不相关共享变量的多个临界区同时执行同步,增加了并行度。代价是将每个共享数据变量与同步变量相联系,这要求额外的开销,且增加了复杂度。这样编程更为复杂且更易于出错。

同步变量的使用如下。每个同步变量有一个当前拥有者,即最近对执行它获取访问的进程。拥有者可以反复进出临界区,而不必在网上发送消息。同步变量的非拥有者进程试图获取访问它时,必须在网上发送消息给拥有者,要求拥有权,获得相关变量的当前值。在非互斥方式下,几个进程同时拥有一个同步变量更是可能的,也就是说,它们只能读,而不能写相关的数据变量。

当存储器满足下列所有条件时,就成为入口一致性存储器 (Bershad 和 Zekauskas, 1991):

- (1) 只有某一进程的保护共享变量全部被更新以后,该进程才允许执行同步变量的获取访问;

- (2) 在一进程以互斥模式访问该进程的同步变量之前,不允许其他进程持有此同步变量,即使在非互斥模式下;

- (3) 在结束互斥模式下对一个同步变量的访问后,任意其他进程必须与该变量的拥有者核查,才能试图以非互斥模式访问该同步变量。

第一个条件说明,当进程执行获取访问时,该操作在所有保护的共享变量均更新后才能结束 (即将控制权交给下一条语句)。也就是说,在获取操作时,所有被保护数据的

远程修改都必须可见的。

第二个条件说明，在更新一个共享变量前，进程必须以互斥方式进入临界区，以确定没有其他进程同时更新它。

第三个条件说明，若进程试图以非互斥方式进入临界区，它必须与保护此临界区的同步变量的拥有者核查，以获得被保护共享变量的最新拷贝。

6.3.8 一致性模型总结

尽管还有其他的一致性模型，以上讨论的是主要的几种。它们的区别在于限制的不同、应用复杂度、编程难易程度及性能的不同。严格一致性最为严格，但因为它在 DSM 系统中的应用几乎不可能，所以从未使用过。

顺序一致性是可行的，在编程人员中很流行且广泛应用。但是它的性能很差。解决这个问题方法是放宽一致性的模型。图 6-22 (a) 以限制程度递减的顺序给出了几种可能的模型。

一致性	说明
严格	所有的共享访问事件都有绝对时间顺序
顺序	所有进程都以相同的顺序检测到所有的共享访问事件
因果	所有进程都以相同的顺序检测到所有因果联系的事件
处理器	PRAM 一致性+存储相关性
PRAM	所有的进程按照预定的顺序检测到来自一个处理器的写操作，来自其他处理器的写操作不必以相同的顺序出现

(a)

弱	同步完成后，共享数据才可能保持一致
释放	当离开临界区时，共享数据就保持一致
入口	当进入临界区时，和该临界区相关的共享数据保持一致

(b)

图 6-22 (a) 不用同步操作的一致性模型

(b) 使用同步操作的一致性模型

因果一致性，处理器一致性和 PRAM 一致性都代表了较弱的模式，即不再有一个全局的所有进程认可的操作次序。不同进程所见操作顺序不同，这三种区别在于哪种顺序是许可的，而哪种不允许，但在所有的情况下，都需要编程人员来避免程序只在顺序一致存储器下才能正确运行的情况。

另一措施是引入了明确的同步变量，正如弱一致性，释放一致性和入口一致性所做的那样。如图 6-22 (b) 总结了这三种模式。当进程对普通共享数据变量执行操作时，不能保证它们何时对于其他进程是可见的。只有当访问同步变量后，变化才能传播出去。这三种模型的不同在于其同步机制如何工作。但在这三种情况中，它们都可在一个临界区中执行多重的读写操作，而不引起数据传输。当临界区的操作完成后，最后结果或者广播发送给其他进程或准备就绪在其他进程需要时再发送出去。

总之，弱一致性、释放一致性和入口一致性需要额外的编程结构，这样使用时，允

许编程人员将存储器假设为顺序一致的，但实际上并非如此。理论上，这三种模型使用了明确的同步，性能应是最好的，但不同的应用程序可能会有不同的结果。在得出结论前，我们还需要做很多研究工作。

6.4 基于分页的分布式共享存储器

我们已经研究了分布式共享存储器系统的基本原理，现在让我们回到系统本身。本节我们将研究“经典”的分布式共享存储器(DSM)，首先是 IVY(Li 1986; Li 和 Hudak 1989)。这些系统建立在多计算机之上，也就是说，或是通过特殊的基于消息传送的网络相连的处理机，或是局域网上的工作站，或是与之类似的设计。这里的基本条件是任何处理机都不能直接访问其他处理机的存储器。这样的系统有时称为 NORMA(NO Remote Memory Access) 系统，以区别于 NUMA 系统。

NUMA 和 NORMA 的最大区别在于，在 NUMA 系统中，每一处理机可以通过读写操作直接访问全局地址空间的每一字。页可以随机分布在存储器中，而不影响程序的运行结果。当处理机访问远程页时，系统可以选择是复制页还是远程访问它。这个决定影响到系统性能，但不影响正确性。NUMA 是真正的多处理机系统——硬件允许任一处理机访问地址空间的每一字节，不需要软件的干涉。

局域网上的工作站从根本上不同于多处理机。处理机只能访问本地存储器。不像在 NUMA 或 UMA 多处理机系统中，这里没有全局共享变量的概念。然而，DSM 的目的就是在系统中增加软件，以允许多计算机系统运行多处理机上的程序。这样，当一处理机访问远程页时，必须取回该页。这里不需作出在 NUMA 系统中的选择。

DSM 系统的早期研究工作集中于如何在多计算机系统上运行现有的多处理机程序。有时称为“dusty deck”问题。它的主要思想是给通过在新的 DSM 系统上运行原来的程序，给它们注入新的活力。这个概念对那些希望得到所有可得的 CPU 周期的应用程序尤为有利，所以这些程序的编写者对大规模多计算机而不是小规模多处理机更感兴趣。

因为在多处理机上编写程序时通常假设存储器是顺序一致的，DSM 的早期工作主要是提供顺序一致的存储器，这样原有的多处理机程序就可以不加修改地运行。以后的经验表明，使用限制更松的存储器模式可以获得更好的性能，其代价是重新改写已存在的应用或以不同的形式重写应用。后面我们将重新回到这一点，这里我们首先将看一看 IVY 类型的经典 DSM 系统设计中的主要问题。

6.4.1 基本设计

DSM 系统的思想是简单的：试图使用 MMU 或操作系统软件模仿多处理机中的缓存。在 DSM 系统中，地址空间被分为大的页块(chunks)，散布在系统中所有处理机上。如处理机访问的地址不在本地，激活陷阱程序，DSM 软件将包括该地址的页块取出，并重新开始执行出错的指令，该指令现在将成功地完成。图 6-23 (a) 讲述了这个概念，图中地址空间有 16 个页块，4 个处理机，每一处理机可以有 4 块。

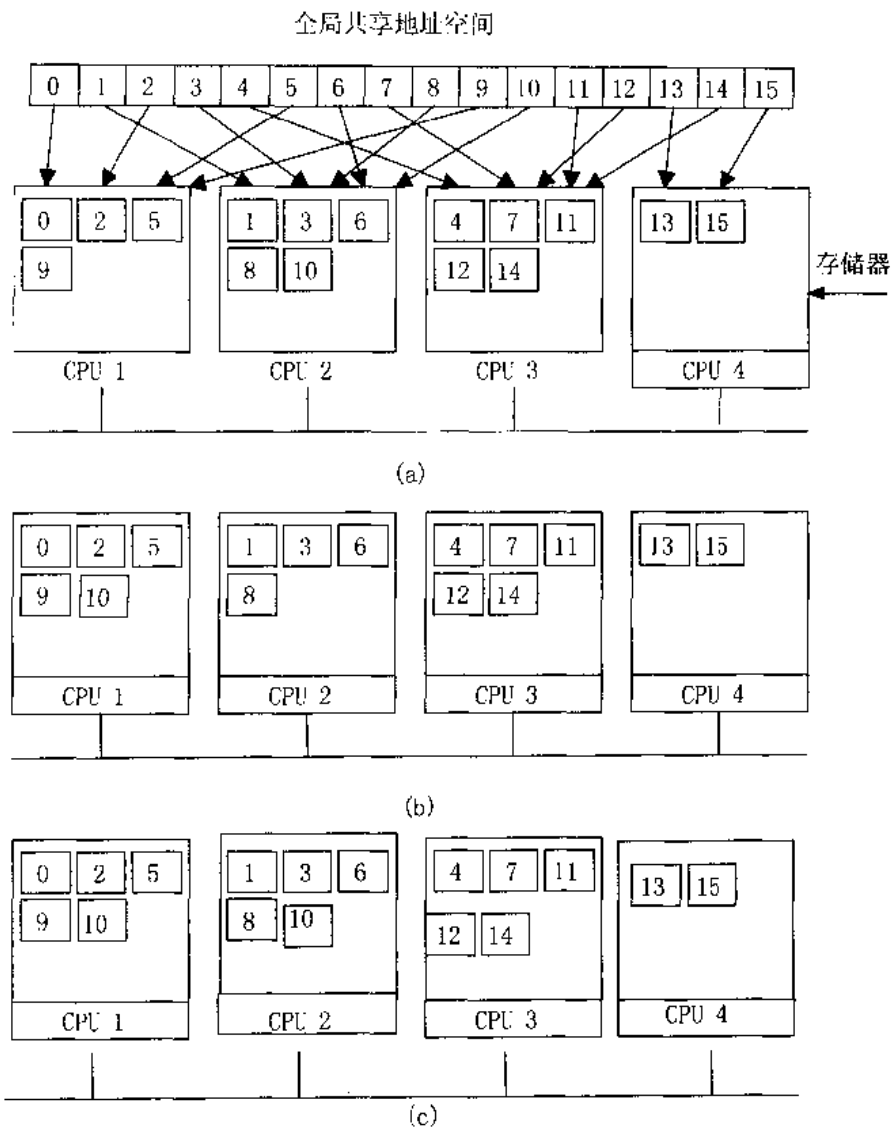


图 6-23 (a) 分布在四台机器上的地址块
(b) 当 CPU 1 访问地址块 10 后的情形
(c) 地址块 10 为只读且使用复制后的情形

在这个例子中，如果处理机 1 访问的指令或数据在块 0、2、5 或 9 中，访问就在本地执行。访问其他的块会激活陷阱程序。例如，访问块 10 中的地址会激活 DSM 系统软件的陷阱程序，软件将块 10 从机器 2 移到机器 1 上。如 6-23(b)所示。

6.4.2 复制

复制只读块，例如程序文本、固定的内容以及其他的只读数据结构，这大大提高了基本系统的性能。例如，图 6-23 中的块 10 是一程序文本的一部分，它可以发送给处理机 1 一个复制块，使处理机 1 可以使用它，而不使用处理机 2 的存储器中的原始块，如图 6-23 (c) 所示。这样，处理机 1 和 2 都可以经常访问块 10，而不激活陷阱程序以获取不在

本地的存储器块。

另一种可能的方案是不仅复制只读块，而是复制所有页块。如果只是执行了读操作，那么复制只读块和复制读写块没有实质区别。然而，如果复制的块突然被修改，则必须采取特殊的方法以避免存在多个不一致的拷贝。下节我们要讨论如何防止不一致性的问题。

6.4.3 粒度 (Granularity)

DSM 系统和多处理机在主要方面相类似。在这两个系统中，当访问非本地字时，系统将该字的一块存储器从所在地址取出，放在执行访问的机器上（分别是存储器和缓存）。一个重要的设计问题是页块的大小应是多大。可能的方案是字、块（少量字）、页或段（多个页）。

在多处理机中，取一个字或一些字节是可行的，因为 MMU 确切知道访问地址而且总线传输的速度可以以纳秒计。尽管 Mennet 不是严格的多处理机，但也使用了较小的页块（32 字节）。在 DSM 系统中，因为采用了 MMU 的工作方式，所以这样的页块大小是很难实现或根本不可能实现的。

当进程访问不在存储器中的字时，就产生缺页中断。一个明显的选择是调入所需的整个页。而且，因为使用了相同的单元，页，具有虚拟存储器的集成 DSM 系统使整个设计更为简单。在缺页中断时，所缺页从另一机器而不是从磁盘中换入，处理缺页中断的代码和传统的一样。

然而，另一种可能的选择是使用较大的单元，即包括所需页的 2、4 或 8 页的一个段。实际上，这相当于模仿了较大的页面。在 DSM 系统中使用较大的块既有优点也有缺点。最大的优点是因为网络传输的开始时间较长，传输 1024 字节的时间和传输 512 字节的差不多。当要移动大量地址空间时，使用大单元传输数据可以减少传输的次数。因为很多程序显示了访问的本地性，即如果程序访问了程序中的某个字，很可能它很快就要访问该页中的其他字，这一点尤为重要。

另一方面，大量的传输使网络连接更为紧密，可能会阻塞其他进程的缺页中断。有效页太大引起的新问题，称为“错误共享”（false sharing），如图 6-24 所示。这里，我们有一页包含了两个无关的共享变量 A 和 B。进程 1 经常对 A 执行读写操作。相似的，进程 2 经常使用 B。这时，包含这两个变量的页将经常在两个进程之间传送。

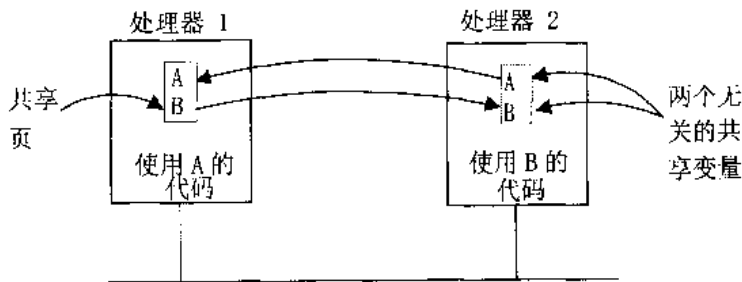


图 6-24 包含两个无关变量的页的错误共享

这里的问题是尽管变量是无关的，它们也会偶然出现在同一页上，当一进程使用它们之一时，进程也得到了其他的变量。有效页越大，发生错误共享的可能性越大，相反的，共享页越小，发生的可能性就越小。在一般的虚拟存储器系统中没有与此相似的情况。

可以理解问题并相应地将变量置于不同地址空间的智能化编译器可以减少错误共享,提高系统性能。然而,真正实现它并不像说的那样简单。而且,如果错误共享包括进程1使用了一数组中的一个元素,而进程2使用了该数组的另一元素,即使是智能化编译器也很难解决这种问题。

6.4.4 实现顺序一致性 (Achieving Sequential Consistency)

如果不复制页,实现一致性不是问题。每一页只有一个拷贝并根据需要来回调动。因为每一页只有一个拷贝,所以不用担心不同的拷贝有不同的值。

如果只复制只读页,那还没有问题。只读页永远不会被改变,所有的拷贝总是一样的。每一读/写页只保存一个拷贝,所以不一致性在这里也是不可能的。

有趣的事情是复制读写页。在许多 DSM 系统中,当进程试图读取一远程页时,由于系统不知道页上有什么以及是否可写,就复制本地拷贝。本地拷贝(实际上是所有的拷贝)以及原始页都在各自的 MMU 中设为只读。只要所有的访问都是读操作,就不会出现问题。

然而,因为不能改变一个拷贝而不管其他拷贝,所以如果一进程试图在一复制页上写,就出现了潜在的 inconsistency 问题。在多处理机系统中,当一处理机试图修改在其他处理机缓存中的字时,也会发生类似的情况。所以让我们回顾一下多处理机在这种情况下是如何处理的。

多处理机一般有两种处理方法:更新或置无效。通过更新,写操作允许在本地执行,且允许被修改字所在的地址在总线上同时广播发送到其他各缓存。每一拥有该字的缓存发现它被修改,则从总线拷贝新值到缓存,覆盖原来的值。最后的结果是更新前拥有这个字的缓存在更新后仍旧拥有这个字,并且都获得了新的值。

多处理机采用的另一种方法是置无效。一旦使用了这种方法,就只有被更新字的地址而不包括新值在总线上广播发送。当缓存发现其中的字被更新时,它将包括该字的缓存块置无效,即将它从缓存移走。最后的结果是只有一个缓存拥有被更新的值,所以这避免了一致性问题。如果其中一进程只拥有缓存块的无效拷贝而试图使用它,它就会产生缓存错误(cache miss)并从拥有有效拷贝的处理机中获得该数据块。

尽管这两种方案在多处理机中实现的难易程度差不多,但在 DSM 系统中它们很不一样。不像在多处理机系统中,MMU 知道要写哪个字且新值是什么,在 DSM 系统中,软件不知道被写的字是什么,或新值是什么。为发现这一点,它要对要修改的页做一秘密的拷贝(页号是已知的),使该页可写,并设置硬件陷阱位,该位在每一条指令后激活陷阱程序,使出错进程重新开始运行。一条指令以后,它捕获了陷阱,并将当前页和它刚刚所做的秘密拷贝相比较,以确定是否有字改变了。它可以在网络上广播发送一个短信包,给出地址和新的值。进程接收到信包后,检查自己是否有不一致的页,如果有,就更新他。

这里的工作量是很大的,更为糟糕的是,这一方案并不十分安全。如果几个源于不同处理机的更新同时发生,不同的处理机或许会看到不同的顺序,所以存储器可能不是顺序一致的。在多处理机系统中,因为总线的广播完全可靠(不会丢失信息),顺序是明确的,这样的问题不会发生。

另一问题是,因为很多程序显示了访问的本地性,所以进程可以上千次地连续写同一

让我们现在看一看每一例中的操作。在图 6-25 (a) 的前四种情况中, P 仅做读操作。在这四例中, 页被映射到它的地址空间, 所以读操作由硬件完成。不会激活陷阱程序。在第五第六例中, 页没有被调进来, 发生缺页错误, DSM 的软件获得控制权。它发送消息给拥有者以获得一份拷贝。当拷贝传送过来时, 页被映射, 出错的指令继续执行。如果拥有者的页处于 W 状态, 它必须将之降级为 R 状态, 但还可以保留此页。在这个协议中, 其他进程是拥有者, 但在其他一些略有不同的协议中, 这也可以变化。

如图 6-25 (b) 所示, 写有不同的处理方法。在第一种情况中, 因为页映射为只读模式, 所以写操作只是发生了, 而不激活陷阱程序。在第二种情况中 (没其他拷贝), 页被改为写状态并被写入。在第三种情况中, 该页有其他拷贝, 所以在写以前, 必须先置无效这些拷贝。

在后面三种情况中, 当 P 要执行写操作时, 其他进程是该页的拥有者。在所有这三例中, P 必须要求现在的拥有者将任何已存在拷贝置无效, 将拥有权传给 P , 除非 P 已经有了一个拷贝, 将该页的一个拷贝也传给 P 。只有这样写操作才能开始。在这三例中, P 最后拥有该页的唯一拷贝, 其状态为 W 。

在所有六例中, 在写操作执行以前, 协议保证在要写的进程的地址空间中只有页的一个拷贝存在。这样, 可以保证一致性。

6.4.5 寻找拥有者 (Finding The Owner)

在以上的描述中, 我们忽略了几点。其中一点是如何寻找某页的拥有者。最简单的方法是通过广播, 要求某一页的拥有者响应。一旦拥有者以这种方式确定下来, 则协议如上所述继续进行。

一个明显的优化是不询问谁是拥有者, 而判断发送请求者要求读还是写, 是否需要该页的一个拷贝。拥有者可以根据需要发送一条消息传送拥有权和页。

广播的缺点是打断了每一个 CPU, 强制它检查请求包。对于除拥有者以外的进程来说, 处理中断基本上是浪费时间。根据硬件的不同, 广播耗费了大量的网络带宽。

Li 和 Hudak (1989) 还提到了几种其他的可能方法。首先, 指定一个进程为页管理者, 跟踪哪个进程有哪些页。当一进程 P 想读写它不拥有的页时, 就发送消息给页管理者, 说明要在何页上做何操作。管理者回送消息, 指出拥有者是谁。 P 现在和拥有者联系, 按需要获取页或拥有权。如图 6-26 (a), 这个协议需要四条消息。

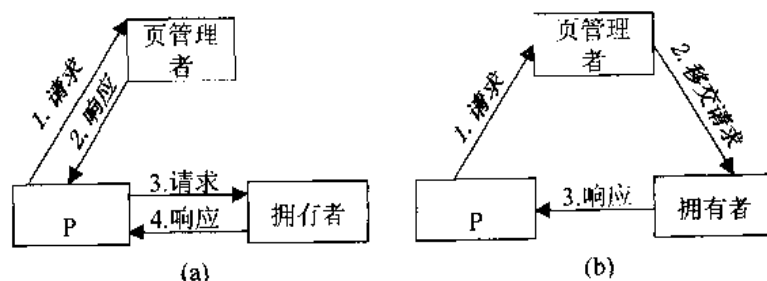


图 6-26 使用集中管理者的拥有权定位

(a) 四消息协议

(b) 三消息协议

图 6-26 (b) 是拥有者定位协议(ownership location protocol)的一个优化。这里页管理者直接将请求发送给拥有者, 拥有者则直接将应答发送给 P , 节省了一条消息的传送。

这个协议的一个问题是, 页管理者要处理所有到来的请求, 负担可能很大。虽然用多个页管理者代替一个可以减少这种问题, 但可能会引起新问题, 即如何找到管理者。一个简单的解决方法是用页号的低字节作为管理者表的索引。这样, 例如有 8 个管理者, 所有由 000 结尾的页由管理者 0 处理, 以 001 结尾的页由管理者 1 处理, 以此类推。其他的映射方法, 如使用哈希函数, 也是可以的。页管理者不仅为到来的请求提供应答, 而且跟踪拥有权的改变。当进程要求在一页上写时, 管理者将该进程指定为新的拥有者。

另一个可能的算法是让每个进程(更可能的是每个处理机)跟踪每一页的可能拥有者。拥有权的请求被发送到可能的拥有者, 如果拥有权改变则继续传送请求。如果拥有权被改变数次, 请求信息会被转发数次。在执行的开始或拥有权改变 n 次以后, 新拥有者的位置必须被广播发送, 以允许所有处理机更新可能拥有者的表。

在多处理机系统如 Dash 和 Memnet 中, 管理者定位的问题也存在。在这些系统中, 解决方法是将地址空间分为若干区域, 并将每个区域指定给一固定的管理者, 这基本上采用了和上述解决多管理者的方法的相同技术。

6.4.6 寻找拷贝

另一重要的细节是当所有的拷贝需要置无效时, 如何找到它们。有两种方法是可能的。一是广播发送包含该页号的一条信息, 要求所有拥有该页的处理机将它置为无效。这种方法只有在广播完全可靠, 不会丢失数据时才能正常工作。

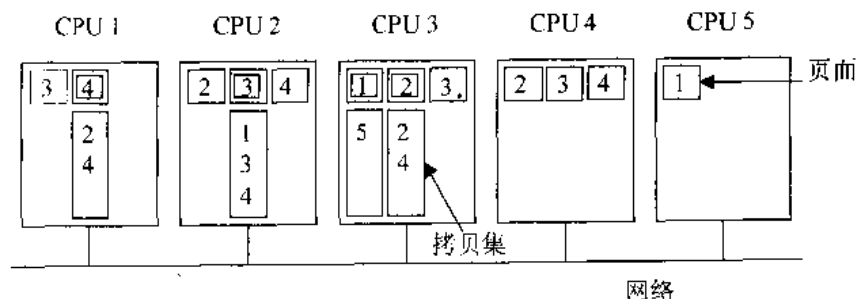


图 6-27 每个页面的拥有者通过拷贝集得知哪个 CPU 正共享该页面。页面拥有者用双线框表示

另一可能的方法是使拥有者或页管理者维护一张表或拷贝集, 判断哪个处理机有哪些页, 如图 6-27 所示。例如页 4 由 CPU1 的一个进程拥有。图中 4 上用双线框表示这一点。因为 CPU2 和 CPU4 都有页 4 的拷贝, 拷贝集包括 2 和 4。

当一页必须被置无效, 原来的拥有者、新拥有者或页管理者给每一拥有该页拷贝的处理机发送消息并等待确认。当每条消息都得到确认, 置无效就完成了。

当一进程突然要求写时, Dash 和 Memnet 也需要将该页置无效, 但使用的方法不一样。Dash 用目录结构, 写进程发送信包到目录(用我们的术语是页管理器), 从它的位图中找到所有拷贝, 给每个发送一置无效信包, 收集所有的应答。Memnet 将所需页取出, 通过在环上广播发送一置无效信包, 将所有拷贝置无效。拷贝所在的第一个处理机将它打

包，并设置一起始位标明拷贝在此。下面的处理机仅仅将拷贝置无效。当信包在环中传送而到达发送者时，发送者获得数据且所有的拷贝都已置无效了。实际上，Memnet 以硬件实现了 DSM。

6.4.7 页面置换 (Page Replacement)

正如使用虚拟存储器的任何系统，在 DSM 系统中可能会发生这样的情况，即需要某页时存储器空间没有空余的页块可以容纳它。这时，为了给所需页让出地方必须将某页换出存储器。这就产生了以下两个问题：哪一页被换出及该页置于何处。

在很大的程度上，我们可以用传统的虚拟存储器算法，例如类似于最近最少使用 (LRU) 的算法，决定将何页换出。DSM 系统的一个复杂性是页可以被任意的置无效（由于其他进程的活动），这影响了可能的选择。但是，通过维护刚被置有效的那些页的估算的 LRU 顺序，可以使用任何传统的算法。

使用传统的算法时，跟踪哪些页是“干净”哪些页是“脏”是很有必要的。在 DSM 系统中，因为它知道另一拷贝存在，要调出的首选页是另一进程拥有其拷贝的页。这样，该页不用被保存在任意地方。如果用目录跟踪各拷贝，必须将该改变通知拥有者或页管理者。如果页由广播定位，也可以立即将它丢弃。

第二个最好的选择是置换进程拥有的那些复制页。它可以将拥有权传给其他拷贝之一，可根据具体情况通过通知那个拥有拷贝的进程，页管理者或这两者来实现。因为页本身不被传送，所以只有较小的信息流量。

如果没有合适的复制页作为替换，则必须选择非复制页，例如，最近最少使用的页。调出该页有两种可能的方法：一是将它写回磁盘，另一是将它传送给其他进程。

选择接收调出页的处理机有若干种方法。例如，每一页都可被指定一个必须接收它的属主机器，尽管这可能意味着为那些将来可能被送回的页保留了大量的平时空闲的空间。另外，还可以在每个发送的消息上加上空闲页块数，这样每个处理机就可知道空闲存储器如何在网络中被使用。存储器块的确切信息由一个随机的广播信息给出，使其数目保持最新。

另外，选择复制页（可能被丢弃）和选择在很长时间内没有被访问的页（它可能只有唯一拷贝）可能会发生冲突。在传统虚拟存储器系统中存在着相同的问题，所以，它们采用了同样的妥协和搜索算法。

当不同机器上的进程经常共享一可写页时，不论是错误共享或真正共享，都会引起网络流量问题，这是仅存在于 DSM 系统中的问题。减少网络流量的一种特定方法是强制执行这样的规则：当一页到达任一处理机后，必须在那里停留一段时间 ΔT 。如果要求某页从其他机器调入，请求消息只能排队，直到计时器到期，然后允许本地进程多次访问存储器而不发生冲突。

通常，在多处理机系统中看见页面如何置换是有益的。在 Dash 中，当缓冲区满时，始终存在将某块写回存储器的可能。尽管用磁盘作为页的最终储藏库并不是经常可行的，但在 DSM 系统中不存在这种可能性。在 Memnet 中，每一缓存块都有一属主机器为之提供存储空间。尽管这在 Memnet 和 Dash 系统中并不经济，但在 DSM 系统中，这种设计还是可行的。

6.4.8 同步

和在多处理机系统中一样，在 DSM 系统中，进程也经常需要同步它们的操作。一个普通的例子是互斥，即在某一时间只能由一个进程执行代码的一部分。在多处理机系统中，经常使用 TEST-AND-SET-LOCK(TSL)指令完成互斥操作。通常，如果没有进程在临界区，则一变量被设为 0，如果有进程则设为 1。TSL 指令在一个原子操作中读出变量，并将之置为 1。如果读出的变量为 1，进程就不断执行 TSL 指令，直到进程退出缓冲区，并将变量置为 0。

在 DSM 系统中，尽管代码是正确的，还是存在着潜在的性能问题。如果进程 A 在临界区中而另一进程 B（在另一机器上）希望进入临界区，B 会始终循环，等待该变量变为 0。包含该变量的页可能在 B 机器上。当 A 试图退出缓冲区，并将变量写为 0 时，会出现缺页错误，则将包含该变量的页调入。然而，B 立即出现缺页错误而将页调回。但这种情况是可以接受的。

当其他几个进程也想进入缓冲区时就会出现这个问题。记住 TSL 指令每次执行时都修改了存储器（将同步变量写为 1）。这样每次进程执行 TSL 指令，它必须将包含同步变量的整个页从拥有者那里取出。如果有多个处理机，每几百纳秒内执行一次 TSL 指令，网络流量将不可忍受。

基于这些原因，同步通常需要一附加机制。一种可能的方法是让同步管理器接受进入和离开临界区的消息，锁或解锁变量，如果工作正常则发送应答。当不能进入临界区或不能加锁变量时，不立即发送应答并阻塞发送消息的进程。当临界区可用或变量可以加锁时消息才被送回。这样，同步可以在网络流量较小的情况下实现，但代价是要集中控制每次加锁解锁的操作。

6.5 共享变量的分布式共享存储器

基于分页的 DSM 采用普通的线性地址，允许页根据需要动态地调动。通过普通的读写指令，进程可以访问所有的内存，而不用了解何时发生缺页中断及网络传输。MMU 检测远程数据的访问并为之提供保护。

另一个更为结构化的方法是仅仅共享那些被多个进程使用的变量和数据结构。这样，问题由如何在网络中分页变为如何维护包含所需变量且可被复制的分布式数据库。这里应用了不同的技术，通常可以带来较大的性能提高。

第一个必须被提到的问题是共享变量是否被复制，如果复制的话，是全部复制还是部分复制。和基于分页的 DSM 系统相比，如果它们被复制，更可能要用到更新算法，使得对单个共享变量的写是隔离的。

使用单独处理的共享变量可以大大减少错误共享。系统可以更新一个变量而不影响其余变量，这样，变量在页上的布局是不重要的。下面将描述这种系统中两个最为有趣的例子，Munin 和 Midway。

6.5.1 Munin

Munin 是基本基于软件对象的一个 DSM 系统,但它可以将每一变量放置在分散的页上,所以可以用硬件 MMU 检测对共享对象的访问 (Bennett 等,1990;Carter 等,1991,1993)。Munin 使用的基本模式是多处理机,每一处理机有一分页线性地址空间,多个线程可以在那里运行仅作了少量改动的多处理机程序。Munin 工程的目标是采用现存的多处理机程序,加以少量修改,使之有效地运行在使用 DSM 模式的多计算机系统上。下面描述的不同技术都获得了良好的性能,包括使用释放一致性代替顺序一致性。

由于在共享变量的声明中用关键字 “shared” 加以注解,编译器可以认出它们。我们还可以给出有关期望使用模式的信息,使之能被认出并优化某些重要的特殊事件。在缺省情况下,编译器将每个共享变量放在一页上。尽管大的共享变量如数组可以占据多个页,程序也可以将多个相同 Munin 类型的共享变量放在同一页上。但是由于某页的一致性协议由其上变量的类型决定,所以类型不能混合。

已编译的程序从一处理机的根进程开始运行。此进程可以在其他处理机中产生新进程,子进程像一般的多处理机进程一样,可和根进程并行运行,并通过共享变量与根进程通信并被彼此通信。一旦一进程在某一处理机上运行,它就不能被移动。

使用 CPU 的一般读写指令可以访问共享变量。这里不要求使用特殊的读写模式。试图使用不在内存的共享变量会发生缺页中断,这时 Munin 获得控制权。

互斥同步的特殊处理方法和内存一致性模式紧密相关。系统可以声明锁变量,可以提供库过程加锁或解锁它们。该系统还支持屏障 (barrier)、条件变量和其他同步变量。

释放一致性

Munin 是基于软件实现 (eager) 释放一致性。Gharachorloo 等(1990)的文章叙述了这样的理论。Munin 所做的就是为用户提供在缓冲区构建程序的工具,使之可以通过获得 (acquire) 和释放 (release) 调用动态地定义。共享变量的写操作必须在缓冲区内进行;读操作则不一定。当进程在缓冲区内被激活,系统不保证共享变量的一致性,但当其退出缓冲区,最后一次释放的访问时修改的共享变量将在所有机器上被更新。只要遵守这种编程模式,程序在分布式共享内存中产生的结果就和顺序一致性的一样。

Munin 区分三种变量:

- (1) 普通变量;
- (2) 共享数据变量;
- (3) 同步变量。

普通变量只能被创建它们的进程读写,不能共享。共享数据变量对多个进程是可见的并显示为顺序一致的,所有进程只能在临界区使用它们。它们必须被声明为共享变量,但可用普通的读写指令访问。像锁和屏障这样的同步变量,只能通过系统提供的访问过程,如锁的加锁解锁,屏障的递增和等待,来访问。正是这些过程使分布式共享系统正常工作。

图 6-28 揭示了 Munin 的释放一致内存的基本操作,图中有三个协作的进程,运行在不同的机器上。在某一时刻,进程 1 想通过加锁 L 进入代码保护的临界区 (所有临界区通过同步变量被保护)。加锁语句保证没有其他进程在缓冲区内执行。其后,用普通的机器指令就可以访问这三个共享变量 a, b, c 。最后调用解锁,结果被传送到其他各个拥有 a, b, c

拷贝的机器上。这些改变被打包为少量信息。当进程 1 还在临界区中时，其他机器访问这些变量产生的结果是不确定的。

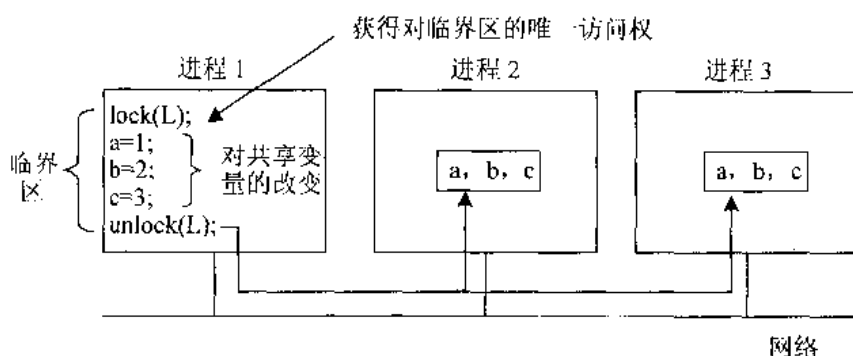


图 6-28 Munin 中的释放一致性

多重协议

除了使用释放一致性，Munin 还使用其他技术以提高系统性能。主要技术是允许编程人员在定义共享变量时加以注释，将它们分为以下四种情况之一：

- (1) 只读 (Read-only);
- (2) 迁移 (Migratory);
- (3) 写共享 (Write-shared);
- (4) 常规 (Conventional)。

Munin 本来还支持其他的类型，但经验表明它们仅是一些边界值，所以被取消了。每一机器都有一目录列出所有变量，并根据其他信息判断它们属于哪一类。每一类都使用了不同的协议。

只读变量是最简单的。当访问只读变量而引起缺页错误时，Munin 就在变量目录表中寻找该变量，以找出谁拥有该变量，并向拥有者请求变量所在页的拷贝。因为该页的只读变量不会被改变（初始化以后），所以不存在一致性问题。MMU 硬件保护只读变量。试图写共享变量会发生致命的错误。

迁移共享变量使用图 6-28 中锁的获得/释放协议。它们在临界区中使用，而且必须被同步变量保护。当进入和退出临界区时，这些变量就从一机器迁移到另一机器。它们不用被复制。

使用迁移共享变量，必须先得到锁。读该变量时，访问它的机器就复制该页而将原来的拷贝删除。一种优化是将迁移共享变量和特定锁相关联，所以当锁被传送，数据也相应被传送了，消除了多余的信包。

当编程人员表明多个进程同时写某变量是安全的时，可以使用写共享的变量。例如，不同进程同时访问一个数组的不同子数组时。初始化时，拥有写共享变量的页被标记为只读，该页可能同时在所有机器上。当写操作发生时，错误处理程序复制该页，称为孪生 (twin)，并将该页标记为 “dirty” 状态，设置 MMU 以允许以后一系列写。图 6-29 说明了这些步骤，某字初始化为 6，而后改变为 8。

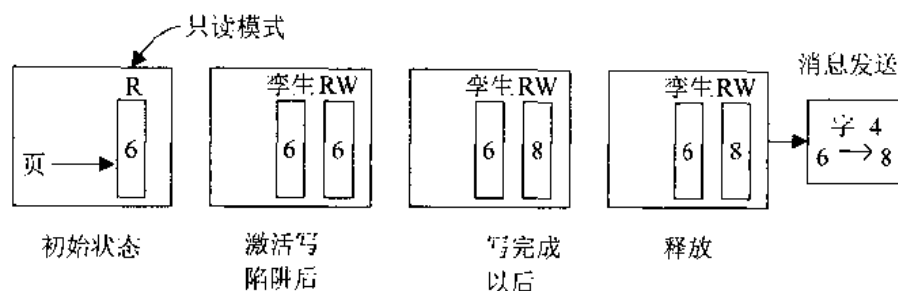


图 6-29 在 Munin 中使用孪生页

当执行释放操作时，Munin 将逐字比较每一状态为“dirty”的共享页和其孪生页，并将不同之处发送（和迁移页一起）给所有需要它们的进程。然后将页重置为只读。

当这一系列的差异到达一进程时，接收者检查每一页以确定本地是否也修改了页。如果某页没有被修改，则接受到来的改变。如果本地修改了页，则逐字比较本地拷贝、孪生页及传送来的页。如果本地页被修改，而传送来的没修改，传送来的就覆盖本地页。如果两者都修改了，就给出运行错误的信号。如果没有这样的冲突，传送来的页就覆盖本地页，程序继续执行。

没有标注为以上几种类型的共享变量和在传统的基于分页的 DSM 系统中一样：可写页只允许一个拷贝，按需要从一进程传送到另一进程。只读页按需要可以有多个拷贝。

现在让我们看一个多重写协议如何工作的例子。考虑一下图 6-30 (a) 和 (b) 所示的程序。两个进程都对同一数组的元素执行递增操作。进程 1 用函数 f 递增下标为偶数的元素，进程 2 用函数 g 递增下标为奇数的元素。在开始某一阶段之前，一进程块在一屏障前等待另一进程到达。完成这一阶段后，进程块仍在下一屏障处等待另一进程块的到来。然后继续执行程序的剩余部分。快速排序的并行程序和快速傅立叶变换就使用了这种方法。

在单纯的顺序一致内存中，两个进程都在如图 6-30 (c) 所示的屏障处停顿。屏障可以这样实现，即让每一进程发送消息给屏障管理器并阻塞进程直到应答到来。屏障管理器在所有进程都到达屏障时才发送应答信息。

通过屏障以后，进程 1 可以开始写 $a[0]$ 。接着进程 2 试图写 $a[1]$ ，由于发生缺页错误，将包含数组的页取回。之后，进程 1 试图写 $a[2]$ ，这引起了另一缺页错误。很不幸的是，每次写入都会引起一整页被传送，从而引起了很大的网络流量。

在释放一致内存中，情况如图 6-30 (d) 所示。两个进程开始都通过了屏障。对 $a[0]$ 的写入为进程 1 创建了一孪生页。相似的，对 $a[2]$ 的写入为进程 2 创建了一孪生页。这些不需要页在机器之间传送。以后，两个进程都在其私有拷贝中任意写 a ，而不会产生缺页错误。

当每一进程到达第二个屏障语句时，比较 a 的现有值和初始值（存在孪生页中）的差别。这些差异被送到所有和该页有关的进程中。这些进程又依次将它们传送到其他对之感兴趣而改变源不知道的进程。冲突会导致运行时的错误。

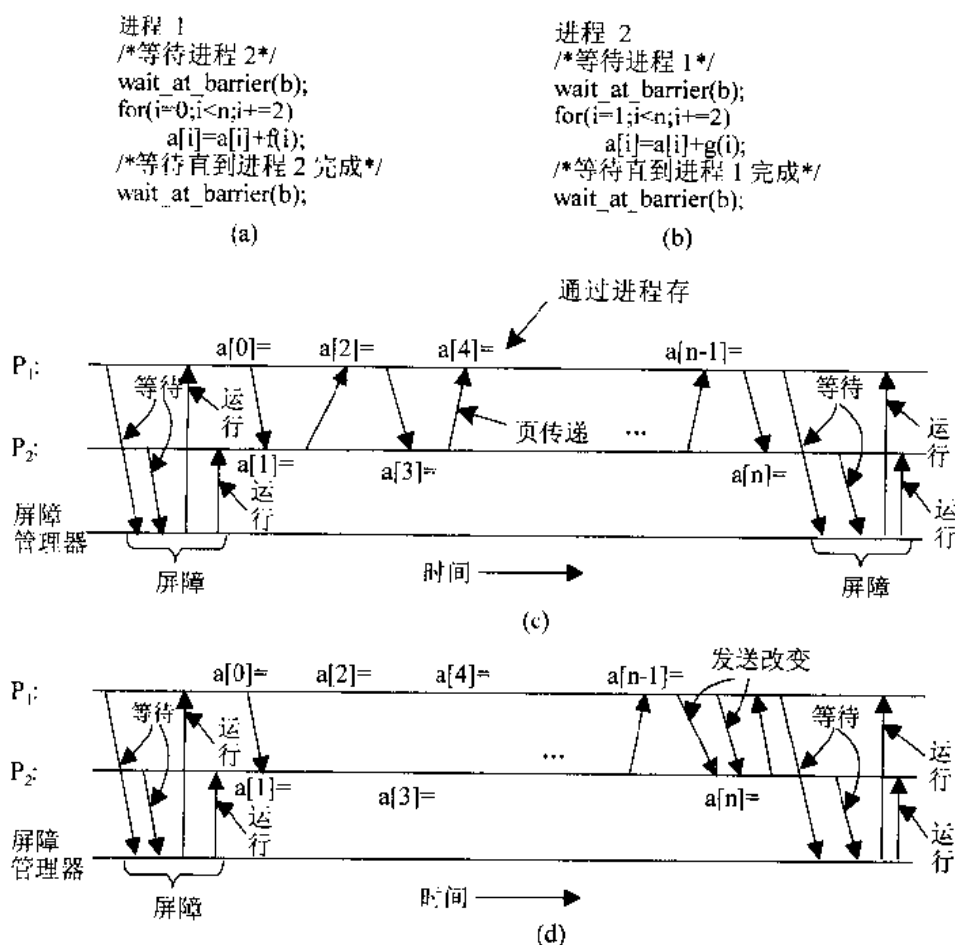


图 6-30 (a) 使用数组 a 的一个程序
(b) 使用数组 a 的另一程序
(c) 顺序一致存储器的消息传送
(d) 释放一致存储器的消息传送

当进程以这种方式报告了差异之后，它发送消息给屏障管理者并等待响应。当所有进程将它们的更新传送出并到达屏障时，屏障管理者送出应答，每一个进程可以继续执行。这样，只有当进程到达屏障时才需要传送页。

目录

Munin 使用目录定位包含共享变量的页。当访问共享变量出错时，Munin 用哈希算法排列导致出错的虚拟地址以找到共享变量目录中变量项。从该项中，它找到变量的种类 (category) 是否有本地拷贝，可能的拥有者是谁。对于普通共享变量，拥有者是请求写访问的最后进程。对迁移共享变量 (migratory shared variable)，拥有者是现在拥有它的进程。

可能拥有者的定位可以使用下列算法。当 Munin 进程开始，根进程拥有所有共享变量。之后，进程 P_1 访问共享变量，发生缺页错误，它发送一条消息给根进程要求该页。根进程将所需页传送过去，并将 P_1 设为拥有者。如果 P_2 请求该页，根进程告知它 P_1 现在是拥有者。 P_2 向 P_1 要求该变量并得到它。如果 P_2 要执行写操作，或变量的类型是迁移的，

P_2 就成为新的拥有者，并由 P_1 记录下来。图 6-31(a)显示了可写或迁移变量此时拥有者的状态。

现在假设 P_3 和 P_4 相继要求该页。它们也随着这条链，产生了图 6-31 (b) 所示的结果。现在 P_1 又需要变量了。因为它认为 P_2 是拥有者，所以就给 P_2 发消息，从而得知 P_2 认为 P_3 是拥有者。根据这条链， P_1 得到该页，该链如图 6-31 (c) 所示。这样，每一个进程最后都得知谁是可能的拥有者，并以这种方法从这条链找到真正的拥有者。

目录还用于跟踪拷贝集。然而，拷贝集不需要十分一致。例如，假设 P_1 和 P_2 都有些写共享变量，并互相了解另一进程所有的哪些变量。之后， P_3 向 P_1 要求并得到了拷贝。因为拥有其一个拷贝， P_3 记录了 P_1 ，但 P_2 并不知道。此后， P_4 认为 P_2 是拥有者，向它要求一个拷贝，并将 P_2 的拷贝集更新为包括 P_4 。这样，没有一个进程拥有页的拥有情况的完整列表。

尽管如此，保持一致性是可能的。假设 P_4 现在释放了一个锁，它将更新传送到 P_2 。 P_2 到 P_4 的确认信息包括一条注解即 P_1 也拥有一个拷贝。当 P_4 和 P_1 联系时就得知了 P_3 。这样，它最终得到了所有的拷贝集，所有的拷贝可以被更新且它可以更新自己的拷贝集。

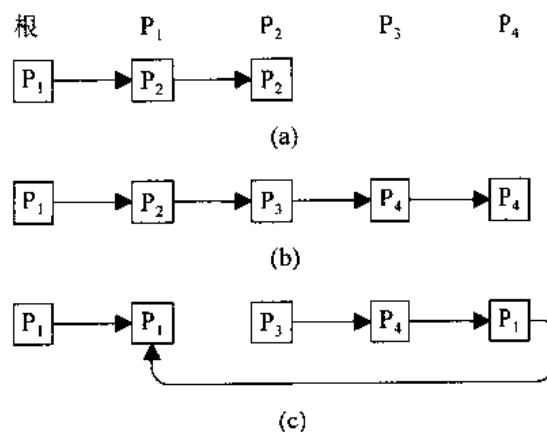


图 6-31 在任一时刻，一进程可以认为其他进程是一些页的可能拥有者

为节省开销，可以不再把更新发送给不再使用某一特殊写共享页的进程，这使用了基于计时器的算法。如果进程拥有一页，但在一段时间内没有访问它，并收到一个更新后，就丢弃该页。下一次它收到被丢弃页的更新时该进程告诉更新进程不再有此拷贝，所以更新者在拷贝集中减去它。可能拥有者链标明了最后一个拷贝，除非找到新的拥有者或写入磁盘，它不能被丢弃。这种机制保证某页不能被所有进程丢弃，以免丢失。

同步

Munin 为同步变量维护了第二个目录。同步变量的定位方法和普通共享变量的定位方法差不多。从概念上来说，锁操作和在集中式中的类似，但实际上，分布式的实现通常要避免向任何机器传送过多的流量。

当一进程要得到锁时，它先检查是否已拥有锁本身。如果是这样而且锁为空闲，它就请求被批准。如果锁不在本地，它就用同步目录定位，同步目录跟踪可能的拥有者。如果锁不空闲，就请求被加在队列尾。当锁被释放时，拥有者将它传给队列中的下一个进程。

屏障通过集中式服务实现。当屏障被创建时，它要求一定数量的进程必须在此等待，

直到全部被释放。当一进程完成计算的一阶段时，它可以发送一条消息给屏障服务器要求等待。如果所有的进程都在等待，屏障服务器就可以发送消息，全部释放这些进程。

6.5.2 Midway

Midway 是基于共享独立数据结构的分布式共享内存系统。它虽在一些方面和 Munin 类似，但本身有一些有趣的新特性。它的目标是只将代码作少量改动，就允许现有的和新的多处理机程序有效地在多计算机系统上运行。关于 Midway 的更多信息可以参见 (Bershad 和 Zekauskas, 1991; 以及 Bershad 等, 1993)。

Midway 的程序基本是用传统的 C、C++ 或 ML 编写的程序，编程人员提供了一定的附加信息。Midway 程序为实现并行机制使用了 Mach C-线程包。一个线程可以生成一个或多个线程。子线程之间及子线程和父线程并行运行（即将每一线程看作一独立进程）。所有线程共享相同的包括私有数据和共享数据的线性地址空间。Midway 的工作是有效地保证共享变量的一致性。

入口一致性

所有对共享的变量和数据结构的访问请求都要在特殊的 Midway 所知的临界区中执行，这样保证了一致性。每一个这样的临界区都有特殊的同步变量保护，一般是锁或屏障。在临界区中访问的每一共享变量必须通过过程调用与临界区的锁（或屏障）明确相联系。这样，当进入或退出临界区时，Midway 精确地知道哪个共享变量可能被访问或已经被访问。

Midway 支持入口一致性，它以如下方式工作：为访问共享变量，进程一般通过调用库过程，即以 lock 变量为参数的锁，进入临界区。该调用还要明确表示要求唯一锁还是非唯一锁。当一个或多个共享变量要更新时需要唯一锁。如果只是读取而非修改共享变量，非唯一锁就足够了。它允许多个进程同时进入同一临界区。因为没有共享变量被修改，所以不会有危险。

调用了锁，Midway 运行期系统就要求该锁同时更新所有和该锁相关联的共享变量。这样做可能要求给其他进程发送信息以获得最新值。当接收到所有应答，锁就被授权（假设没有冲突），进程开始在临界区运行。当进程完成了临界区内的操作时，它释放锁。和释放一致性不一样，在释放时没有通信发生，即修改的共享变量不被传送到使用它们的所有机器中。只有当这些进程之一以后请求该锁并需要当前值时才发生通信。

为使入口一致性正常工作，Midway 要求程序具有多处理机程序没有的三个特性：

- (1) 用新的关键字 shared 声明共享变量；
- (2) 每一共享变量必须和锁或屏障相联系；
- (3) 共享变量只能在临界区中被访问。

做到这些要求编程人员额外的努力。如果没有完全遵守这些规则，就不会产生错误信息，但程序可能会得出不正确的结果。因为这样编程在某种程度上易于出错，尤其是运行原有的不再有人懂的多处理机程序时，所以 Midway 还支持顺序一致性和释放一致性。这些模式只要求较少的细节就可以正确执行。

Midway 要求的额外信息应该被认为是我们早先在一致性下研究的软件和内存协约的一部分。实际上，如果程序遵守以上的那些规则，内存就保证正常工作。否则，所有约定

都无效。

实现

当进入临界区时, Midway 运行期系统必须首先得到相应的锁。为得到唯一锁, 必须定位锁拥有者的地址, 即最后一个唯一得到它的进程。每一进程使用与 IVY 和 Munin 中一样的方法跟踪可能的拥有者, 随着一系列拥有者的分布的链, 找到现在的拥有者。如果该进程当时不使用该锁, 则将它传送出去。如果锁在使用, 请求的进程就等待, 直到锁空闲。为得到非唯一锁, 需要和所有现在拥有它的进程通信。屏障由集中的屏障管理器管理。

在请求得到锁的同时, 请求的进程将更新所有共享变量的拷贝。在简单的协议中, 原来的拥有者仅仅将它们都发送出去。Midway 使用优化算法降低必须传送的数据量。假设某请求在时间 T_1 完成, 相同进程的前一请求在 T_0 完成。因为请求者已经有了其余的值, 所以只有那些在 T_0 以后修改的变量才需被传输。

这种策略产生的问题是系统如何跟踪哪些变量在何时被修改。为跟踪哪些共享变量发生了改变, 可以使用一特殊的编译器编译代码以维护一张运行时间表, 每个共享变量在其中都有一项。无论何时共享变量被更新, 改变都在表中标示出来。如果不能使用这种特殊编译器, 和在 Munin 中一样, MMU 硬件检测对共享变量的写操作。

我们用基于 Lamport(1978) “发生以前 (happens before)” 的时间戳协议来跟踪何时发生变化。每个机器维护一逻辑时钟, 无论何时, 只要一条信息发送时钟就加一, 并将此时钟包含在该信息中。当一条信息到来时, 接收者将逻辑时钟设置为发送者的时钟及其本身时钟中较大的。通过这些时钟, 时间被有效地分为由信息传送定义的时间段。当执行了请求时, 请求的处理器给出以前访问的时间, 并要求在此之后改变的所有共享变量。

这样, 使用入口一致性实现可能会带来很好的性能, 因为只有在进程发出请求时才有通信发生。而且, 只有那些失效的共享变量需要传输。特殊的是, 如果一进程进入、离开、再进入临界区, 则都没有通信发生。这种模式在并行程序的编程中很普通, 所以潜在的获利是很可能的。获得这种性能的代价是: 和其他一致性模型相比, 编程人员的接口更为复杂且易于出错。

6.6 基于对象的分布共享内存

我们研究的基于分页的 DSM 系统应用 MMU 硬件激活陷阱, 实现对所缺页的访问。这种方法有利有弊。特别是, 在许多程序语言中, 数据被组织对象、信包、模块或其他数据结构, 它们彼此独立存在。若进程访问对象的一部分, 在许多情况下需要整个对象, 因此以对象而不是页为单元在网上传送数据是有意义的。

共享变量方法, 如 Munin 和 Midway 提出的, 朝着用更为结构化的方法组织内存的方向上迈出了一步, 但这仅是第一步。在这两个系统中, 编程者必须提供信息以确定哪些变量要共享, 哪些不用, 而且在 Munin 中要提供协议信息, 在 Midway 中要提供关联信息。错误的注解会带来严重后果。

在高级编程模式的方向上更进一步, 就可以使 DSM 系统更为简单, 编程更不易出错。它们可以使共享变量和同步变量的访问结合得更为清晰。在一些情况下, 这种优化在较为具体的编程模式下更难完成。

6.6.1 对象

如图 6-32, 对象 (object) 是编程人员定义的封装的数据结构。它包含内部数据, 即对象状态(object state), 和被称为方法(method)或操作(operations)的过程, 它们对对象状态进行操作。为访问或操作内部数据, 程序必须调用其中某一方法。方法可以改变内部状态, 返回 (部分) 状态及其他。系统不允许直接访问内部状态。这个特性被称为消息隐蔽 (information hiding) (Parnas, 1972)。它迫使所有数据的访问都必须通过方法, 使程序结构更为模块化。

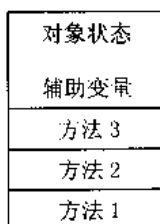


图 6-32 一个对象

在基于对象的分布式共享内存中, 多个机器上的进程共享一充满了共享对象的抽象空间, 如图 6-33 所示。对象的定位和管理由运行期系统自动处理。这个模型与基于分页的 DSM 系统如 IVY 的不同在于, IVY 仅提供从 0 到某一值的自然状态的线性内存。

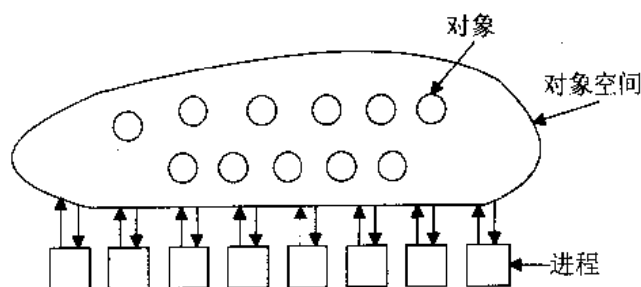


图 6-33 在基于对象的分布式共享存储器中, 进程通信通过调用共享对象的方法完成

不管进程和对象在哪, 任何进程都可调用对象的方法。这个工作由操作系统和运行期系统完成。由于进程不能直接访问任何共享对象的内部数据, 一些在基于分页的 DSM 系统中不可行的优化算法在这里是可行的。例如, 因为内部数据的访问是严格控制的, 所以甚至在编程者不知道的情况下, 放松内存一致性协议也许是可能的。

一旦决定以单独对象的集合而不是线性地址空间来组织共享内存, 就可以作出许多其他的选择。最重要的问题可能是对象是否可被复制。如果不允许复制, 对所有对象的访问只能通过唯一拷贝实现, 这很简单, 但这会导致性能下降。允许对象按需从一机器迁移到另一机器, 可能会减少性能的损失。

另一方面, 若允许复制对象, 当一个拷贝更新后要做些什么? 一种方法是使所有其他拷贝无效, 这样只保留更新的拷贝。别的拷贝可在需要时通过请求创建。另一方法是不将拷贝置无效, 而是更新它们。共享变量的分布式共享内存系统就是这样, 但对基于分页

的 DSM, 置无效是唯一可行的选择。相似的是, 两者都消除了大部分的错误共享。

总之, 基于对象的分布式共享内存有三个优越于其他方法的地方:

- (1) 比其他技术更为模块化;
- (2) 由于访问受到控制, 实现方法更为灵活;
- (3) 同步和访问可以清晰地结合起来。

基于对象的 DSM 也有不足之处。其中一点是, 它不能运行原来的“dusty deck”多处理机程序, 那些程序假定共享线性地址空间的存在, 任何进程都可随机读写。然而, 由于多处理机的概念相对较新, 很少有人关注已有的那些多处理机程序。

第二个缺陷是, 由于对共享对象的访问均需调用方法, 当可被直接访问的共享页不在内存时, 会导致额外的开销。然而, 许多软件工程的专家都推荐将对象作为结构化工具 (甚至在单机上也这样做), 并认为以额外的开销为代价是值得的。

下面将研究两种相当不同的基于对象的 DSM: Linda 和 ORCA。当然也存在其他的基于对象的分布式系统, 包括 Amber(Chase 等, 1989), Emerald(Jul 等, 1988)以及 COOL(Lea 等, 1993)。

6.6.2 Linda

Linda 为运行在多台机器上的进程提供了高度结构化的分布式共享内存。内存的访问是通过少量与原始操作的集合实现的, 原始操作可以加入到现有语言中, 如 C 和 FORTRAN, 从而形成并行语言 C-Linda 和 FORTRAN-Linda。下面我们将着重于介绍 C-Linda, 但概念上它们差异很小。在 (Carriero 和 Gelernter, 1986, 1989; 及 Gelernter, 1985) 中可以找到有关 Linda 的具体信息。

这种方法与引入一种新语言相比, 有若干好处。最主要的优点是用户不用再学习一套新语言。这一点不容低估。第二点是简单, 将 X 语言变成 X-Linda 只需在库中增加几条原语, 调整 Linda 的预处理器, 将 Linda 程序提供给编译器。最后, Linda 系统在操作系统和机器结构中具有高度的可移植性, 可以应用在许多分布和并行系统中。

元组空间 (Tuple Space)

Linda 的独特概念是抽象的元组空间。元组空间在整个系统中是全局的, 任何机器上的进程都可以从元组空间中移入移出元组, 而不管它们存于何处, 如何存放。对用户来说, 元组空间就像我们在图 6-33 中所见的大的全局共享内存。实际应用将牵涉到多个机器上的多个服务, 这一点将在后面描述。

元组类似于 C 中的结构或 PASCAL 中的记录。它包括一个或多个字段, 字段中值的类型由母语支持。对于 C-Linda, 字段类型包括整型、长整型、浮点型以及像数组 (包括字符串) 和结构 (但不是其他元组) 的组合类型。图 6-34 给出了 3 个元组的例子。

```
("abc", 2, 5)
("matrix-1", 1, 6, 3, 14)
("family", "is-sister", "Carolyn", "Elinor")
```

图 6-34 三个 Linda 元组

元组上的操作

Linda 并不是一个完全意义上的基于对象的系统,它只提供几种确定的内置操作,而不能定义新的操作。系统为元组提供了 4 种操作。第一种是 *out*,将元组送入元组空间,例如

```
out ("abc", 2, 5);
```

将元组 ("abc", 2, 5) 放入元组空间。*out* 的字段通常是常量、变量或表达式,如在以下操作中:

```
out ("matrix-1", i, j, 3.14);
```

它输出有四个字段的元组,第二个、第三个字段由变量 *i*, *j* 的当前值决定。

in 原语从元组空间取出元组。它们是通过内容而不是地址或名称定位的。*in* 的字段可以是表达式或普通参数。考虑下面的例子:

```
in ("abc", 2, ? I);
```

这个操作在元组空间“查找”包括字符串“abc”、整型 2、及第三个字段为任意整型(假设 *i* 为整型)的元组。如果找到,该元组就被移出元组空间,并将第三个字段的值赋给变量 *i*。匹配和移动都是原子操作,所以如果同时执行相同的 *in* 操作,只有其中一个会成功,除非有两个或多个元组匹配。元组空间甚至可以包含同一元组的多个拷贝。

in 的匹配算法是顺向的。在 *in* 原语中的字段,称为属性单元 (template),和元组空间每一元组的相应字段相比(概念上),当以下三个条件都满足时,则发生了匹配:

- (1) 属性单元和元组有相同的字段数;
- (2) 相应字段的类型相同;
- (3) 属性单元中的每个常量或变量和元组字段中的相应项匹配。

形式参数,以问号为前导的变量名或类型,不参与匹配(除非类型检验),但匹配成功后将对变量名赋值。

若不存在匹配的元组,调用进程将等待其他进程插入所需元组,这时进程自动激活,获得新元组。进程自动阻塞或解除阻塞的事实意味着当一个进程要输出元组而另一个进程要输入时,谁先执行是没有关系的。区别仅在于如果 *in* 先于 *out* 执行,那么在元组可以移动之前会有一段延迟。

当所需元组不存在时进程阻塞的事实可应用于很多地方。例如,它可用于实现信号量。为了在信号量 *S* 上创建或执行 *UP* (*V*),进程可以执行

```
out ("semaphore S");
```

执行 *DOWN* (*P*),进程执行

```
in ("semaphore S");
```

信号量 *S* 的状态由元组空间中 ("semaphore S") 元组的数量决定。如果不存在,进程阻塞直到其他进程提供一个。

除了 *in*, *out*, Linda 还提供原语 *read*,除了它不从元组空间中取出元组,其他和 *in* 一样。还有原语 *eval*,系统并行计算其参数,并将结果元组存放在元组空间中。使用这种机

制可执行任意的计算，这就是在 Linda 中产生并行进程的方法。

Linda 中的一个普通编程范例是重复工人模型 (replicated worker model)。这个模型是基于任务包 (task bag) 思想的，任务包中包含所有要执行的工作。主进程以执行包括下述的循环开始：

```
out ("task-bag", job);
```

在每个循环中将一个不同的工作描述输出到元组空间。通过每个工人执行：

```
in ("task-bag", ?job);
```

得到一个描述工作的元组，接着开始此项工作。结束后获取另一个。在执行期间，新工作可以加入任务包。以这种简单的方式，工作在工人间动态地分配，每个工人始终在工作，而开销很小。

在某些方面，Linda 与 Prolog 相似，并在很小的程度上是基于 Prolog 之上。它们都支持一种类似于数据库的抽象空间。在 Prolog 中，该空间包括事实(fact)和规则(rules)，而 Linda 中包含元组。在这两种情况下，进程都提供能与数据库内容相匹配的属性单元。

除了这些相同点，两系统间仍在很多方面有所不同。Prolog 是在单机上用于人工智能编程的，而 Linda 用于多计算机上的普通编程。Prolog 有一个包括一致性和回溯的复杂的模式匹配机制；Linda 的匹配机制相对简单。Linda 中一次成功的匹配将从元组空间中取出匹配的元组。Prolog 中不是这样。最后，Linda 进程无法定位所需元组块，元组块是进程间同步的基础。在 Prolog 中没有进程，程序也不会阻塞。

Linda 的实现

Linda 可以有效地在许多不同硬件上实现。下面我们将讨论最为有趣的几种。在所有的实现方法中，预处理器 (preprocessor) 扫描 Linda 程序，抽取有用信息并在需要的地方将之转换为母语言 (如字符串 "? int" 不能作为 C 或 FORTRAN 的参数)。从元组空间插入和移出元组的实际工作是由 Linda 的运行期系统在运行时执行的。

一个有效的 Linda 应用需解决下面两个问题：

- (1) 不通过大量查找，怎样模拟相关的寻址；
- (2) 怎样在机器间分配元组并如何定位。

这两个问题的关键在于每个元组都有类型签字 (type signature)，它包括字段(fields)类型的 (顺序) 列表。而且，每一元组的第一个字段通常是字符串，它有效地将元组空间划分为以字符串命名的不相交的子空间。将元组空间划分为各子空间，每个子空间的元组有相同的类型签字和相同的第一个字段，这样简化了编程，并使某种优化成为可能。

例如，若 in 或 out 调用的第一个参数是常量字符串，在编译时就可确定调用操作在哪个子空间。若第一个参数为变量，则在运行时方可确定。在这两种情况下，这种划分都意味着只需查找一部分元组空间。图 6-35 是 4 个元组和 4 个属性单元。它们共同形成 4 个子空间。每次 in 或 out 调用，都可在编译时确定需要哪个子空间和元组服务。若初始字符串为变量，则需要待运行时才能确定相应子空间。

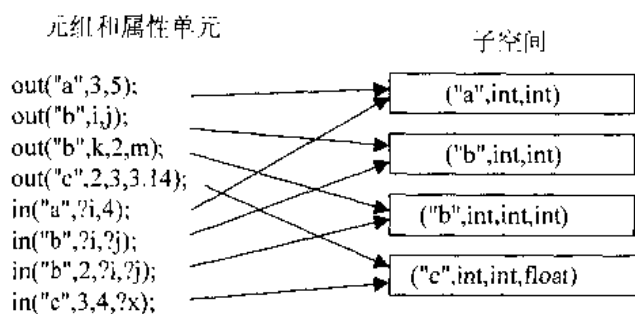


图 6-35 元组和属性单元可通过子空间相关联

而且，每个子空间可组成一个哈希表，用其第 i 个字段作为哈希表的关键字。如果第 i 个字段为常量或变量（而不是形式参数），调用 *in*、*out* 时就可以通过计算第 i 个字段的哈希值找到元组在表中的位置。知道了子空间和表中的位置就省去了全部查找。因为要是某个 *in* 调用的第 i 个字段为形式参数，就不可能用哈希表，所以除了一些特殊情况，需要在整个子空间内查找。通过仔细选择适当的字段做为哈希表的关键字，预处理器在大多数时间通常可避免查找。除哈希表以外，在一些特殊情况中，其他的子空间组织方式也是可行的（如，当有读者和写者时的队列）。

系统还需使用其他的一些优化。例如，以上所述的哈希法将给定子空间的元组分成箱（bin），以限制在一个箱里查找。不同箱可以放在不同机器上，这样不但使负担更分散，而且更加有利于定位。若哈希函数是机器数量的主要参数，则箱的规模与系统大小成线性关系。

现在让我们检查一下不同硬件上的不同应用技术。在多处理机上，元组子空间可通过全局内存的哈希表实现，每个子空间一个。当执行 *in*、*out* 调用时，锁定相应子空间，移入移出相应元组，然后解锁子空间。

在多处理机上，通信结构决定了什么是最好的选择。如果可以使用可靠的广播，最好的选择是在所有机器上复制所有子空间，如图 6-36。当调用 *out* 时，新的元组被广播发送，并写入每个机器的相应子空间。当调用 *in* 时，查找本地子空间。然而，因为 *in* 的成功调用要从元组空间中移出元组，所以需要删除协议(delete protocol)将它从所有机器上移出。为防止竞争和死锁，可以使用两段提交协议(two-phase commit protocol)。

这种设计很直接，但因为每个元组都必须存在每个机器上，所以当系统规模扩大后可能不能适应。另一方面，元组空间的总体大小一般是中等的，所以除非在大系统中，一般不会出现这样的问题。因为 S/NET Linda 系统有一条快速可靠的并行的数据总线用于广播，所以它采用这种方法（Carriero 和 Gelernter,1986）。

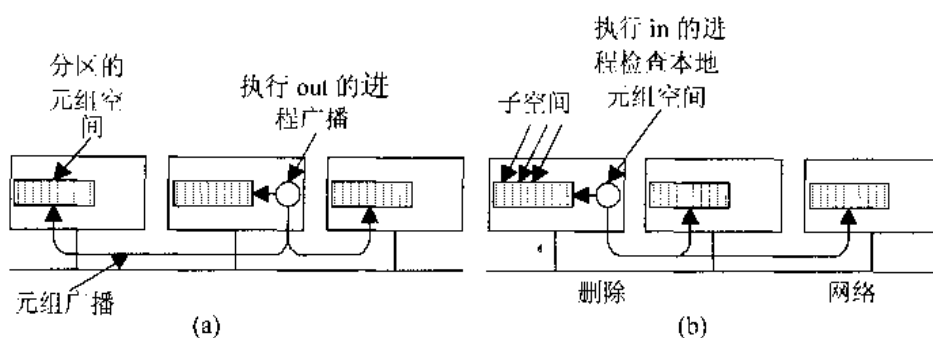


图 6-36 元组空间可在所有的机器上被复制。虚线表示将元组空间分为子空间

(a) 执行 out 时元组被广播

(b) 调用 in 在本地执行，但删除必须被广播

相反的设计是在本地调用 out，仅将元组存储在创建它的机器上，如图 6-37。调用 in 时，进程需广播属性单元。每个接受者检查是否有匹配者，若有则发送应答。

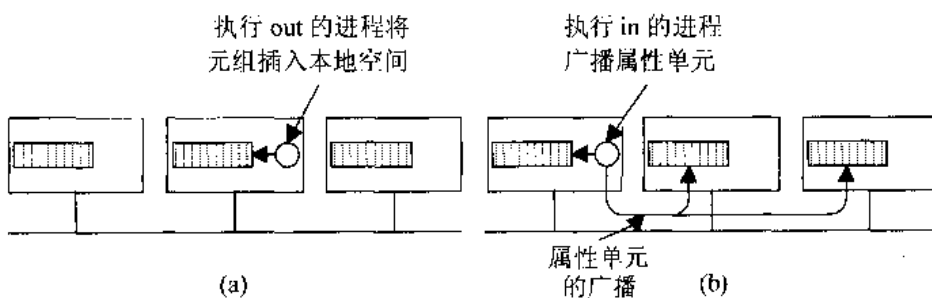


图 6-37 非复制的元组空间

(a) 本地执行 out

(b) 调用 in 时广播属性单元以找到元组

如果元组不存在，或者有元组的机器未收到广播，请求的机器将再次发送广播，并增大两次广播的间隔，直到出现合适的元组，要求得以满足为止。如果要发送 2 个或更多元组，将调用 out，将元组从拥有它们的机器上移入发出请求的机器。实际上，运行期系统甚至会移动元组来平衡负担。Carriero 等人(1986)使用这种方法在 LAN 实现 Linda。

这两个方法结合使用可以生成一个有部分应答的系统。一个简单的例子是假设所有的机器在逻辑上形成一个矩形，如图 6-38 所示。当一机器 A 上进程要调用 out 时，它将元组广播发送（或点到点发送）到与它同行的机器上。当一机器 B 上的进程要调用 in 时，它将属性单元广播发送到与它同列的机器上。根据几何学原理，总有一个机器会同时看到元组和属性单元（本例中是 C 机器），该机器将之匹配，并将元组发给请求的进程。Krishnaswamy(1991)在硬件 Linda 协处理器中使用了这种方法。

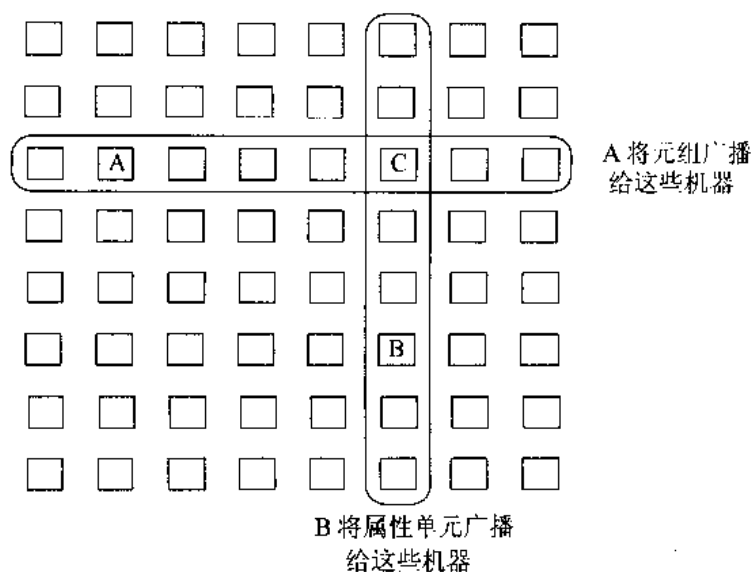


图 6-38 元组和属性单元的部分广播

最后看一下 Linda 在完全没有广播能力的系统中是如何实现的 (Bjornson,1993)。其基本思想是将元组空间分成不关联的子空间, 首先按每个类型签字划分子空间, 再按第一个字段进一步划分。每个分区都可以在不同机器上, 由自己的元组服务器控制, 并将负担分散。当调用 *in* 或 *out* 时, 确定相应的分区, 发送一条消息给该机器, 在那里存放或获取一个元组。

Linda 的经验表明, 使用不同于我们以上研究的基于分页系统中整页移动的方法, 也可以处理分布式共享内存。这和释放及入口一致性中变量共享很不一样。因为以后的系统将更加庞大且功能更强, 所以类似于此的新措施将给我们展现一个新的视野, 让我们了解在这些系统中如何更简单的编程。

6.6.3 Orca

Orca 是一个并行的编程系统, 它允许不同机器上的进程对包含保护对象的分布式共享内存的执行受控制的访问 (Bal,1991;Bal 等,1990,1992)。这些对象和 Linda 中元组相比, 功能更为强大 (且更为复杂), 它支持任意的操作而不是仅有 *in* 和 *out*。另一区别在于 Linda 元组在执行期间大量地被创建, 而 Orca 对象不是。Linda 元组主要用于通信, 而 Orca 对象还用于计算且通常更为重要。

Orca 系统包括语言、编译器和运行期系统, 它在执行期间管理共享对象。尽管语言、编译器和运行期系统被设计为协同工作, 但因为运行期系统独立于编译器, 所以也可以用于别的语言。介绍完 Orca 语言之后, 我们将描述运行期系统如何应用于基于对象的分布式共享内存的。

Orca 语言

在某些方面, Orca 是一种传统语言, 它的顺序语句基本上是基于 Modula-2 的。然而, 它是一种没有指针和别名的类型保护语言。数组范围在运行时检查 (除非在编译时可以检查)。这些及其他一些相似的特征消除或检测了许多普通编程错误, 如在内存中存入数据

过多等。我们应仔细选择语言的特征，使优化算法更易于实现。

Orca 中对于分布式编程最重要的两个特征是共享数据对象（或仅仅是对象）和 `fork` 语句。对象是一种抽象数据类型，类似于 Ada 中的包(package)。它将内部数据结构和用户写的过程封装，对内部数据结构的操作称为操作（或方法）。对象是被动的，即它们不包含将信包发送到何处的线索。相反，进程通过调用操作来访问对象内部数据。一对象不从其他对象中继承属性，所以 Orca 是一个基于对象的语言，而非面向对象的语言。

每个操作包括一个一系列（guard,语句块）对。guard 是一个布尔表达式，guard 的缺省值为 *true*。当调用一个操作时，就以某种不确定的次序计算所有 guard 表达式的值。若均为 *false*，调用进程将延迟到其中一个变为 *true*。如果某个 guard 的值为 *true*，则执行下面的语句块。图 6-39 描述了一个有两个操作 *push* 和 *pop* 的栈对象。

```
object implementation stack;
  top:integer;                                #声明 top 变量
  stack: array[integer0..N-1] of             #栈的存储
integer;
  operation push(item:integer);               #没有返回值的函数
  begin
    stack[top]:=item;                         #将项压入堆栈
    top:=top+1;                               #递增栈指针
  end;
  operation pop():integer;                    #返回整型的函数
  begin
    guard top > 0 do                          #如栈为空则挂花
      top:=top-1;                             #递减栈指针
      return stack[top];                     #返回最上面的项
    od;
  end;
begin
  top:=0;                                    #初始化
end;
```

图 6-39 简化的堆栈对象，它包含内部数据和两个对象

定义一个栈后，可以声明这种类型的变量，如下

```
s, t: STACK
```

就生成了两个栈对象，并将它们的 *top* 变量初始化为 0。整形变量 *k* 可通过下面语句压入栈 *s*，

```
s$PUSH (k);
```

pop 操作有个 guard，所以试图从空栈中弹出变量将挂起调用者，直到其他进程将一些变量压入栈中。

Orca 用 `fork` 语句在用户定义的处理器中创建新进程。新进程运行在 `fork` 中命名的过程。参数包括对象都可以传给新进程。这样，对象就可以分布在机器中了，例如，语句

```
for i in 1..n do fork foobar(s) on i;od;
```

在 1 到 n 的每个机器上生成一个新进程，它们都运行进程 **foobar**。因为这 n 个新进程（及父进程）并行执行，所以它们可以在共享栈 s 上压入或弹出数据项，就像在共享内存多处理机上运行一样。这是运行期系统维持了共享内存的假象，而实际上并不存在。

共享对象上的操作是原语级且顺序一致的。系统保证，假如多个进程几乎同时在同一共享对象上执行操作，操作看起来是严格顺序的，前一操作完成后新操作方可进行。

而且，对于所有进程，操作都是顺序出现的。例如，我们在扩充栈对象时，通过新的操作 *peek* 检查栈顶的数据项。若两个独立进程同时压入 3 和 4，之后所有进程用 *peek* 操作检查栈顶，系统保证要么所有的进程得到 3，要么得到 4。在多处理机系统或基于分页的 DSM 系统中，不可能有些进程看到 3 而另一些看到 4，在 Orca 中也不可能。如果仅有栈的一个拷贝，这很容易做到，若栈在所有机器上均有拷贝，如下所述需要做更多工作。

尽管我们没有强调，Orca 使用了不同于基于分页的 DSM 的方法，综合了共享数据和同步。并行程序中需要两种同步。第一种是互斥同步，防止两个进程同时在同一临界区中执行。在 Orca 中，共享对象上的每个操作很像临界区，因为系统保证最后结果是一样的，就像每一时刻执行一个临界区的操作一样（即顺序的）。在这一点上，Orca 对象类似于 monitor 的分布形式 (Hoare, 1975)。

另一种同步是条件同步，一进程块等待一些条件被满足。在 Orca 中，条件同步是通过 *guard* 实现的。在图 6-39 的例子中，一进程试图从空栈中弹出一数据项，它将被挂起直到栈不为空。

Orca 中共享对象的管理

Orca 中共享对象的管理是由运行期系统控制的。它在广播式（或多播通信）及点对点的网络中都可运行。运行期系统控制对象的复制、迁移、一致性及调用操作。

每个对象可以有两种状态：单个拷贝和复制。只有单个拷贝的对象只存在于一台机器上。复制的对象可以在使用它的进程的所有机器上。因为不要求所有对象都是同一状态，所以进程使用的一些对象可以复制，而另一些为单个拷贝。在执行期间，对象状态可以从单个拷贝改变为复制或反之。

在每台机器上都复制对象的最大优点是读可以不需要通过网络在本地完成，也没有延迟。如果不复制对象，那么所有操作都要发送给对象，调用者必须阻塞等待应答。第二点好处是提高了并行度，可以同时进行多个读操作。而对于单一拷贝，同时只能执行一个操作。复制的主要缺点是为保持一致性而带来的额外开销。

当程序在对象上执行一个操作时，编译器调用运行期系统的过程 *invoke-op*，确定对象、操作和参数以及判断对象是否可被修改（调用写）或不可修改（调用读）的标志位。*invoke-op* 执行的操作取决于对象是否被复制，拷贝是否在本地可得，读还是写，底层系统是否支持有序可靠的广播。如图 6-40 解释了以上区分的 4 种不同情况。

在图 6-40 (a) 中，进程要在单一拷贝的对象上执行操作，该对象恰好在本地。它只是加锁对象，调用操作，解锁对象。加锁对象是程序为了在执行本地操作时，暂时禁止远程调用。

在图 6-40 (b) 中是一个不在本地的单拷贝对象。运行期系统执行 RPC，请求远程机

器执行操作。如果当请求到达时对象被加锁，即可能会有一些延迟。在这两种情况下，读写没有区别（除非写可以唤醒阻塞的进程）。

若对象是复制的，如图 6-40 (c)，(d)，本地则始终有拷贝，但必须注意是读操作还是写操作。读操作在本地完成，不需要通过网络，也没有额外开销。

在复制对象上执行写需要些技巧。如果底层系统提供可靠有序的广播，运行期系统广播对象名，操作，参数和数据块，直到广播结束。所有机器包括它本身，都计算新的值。

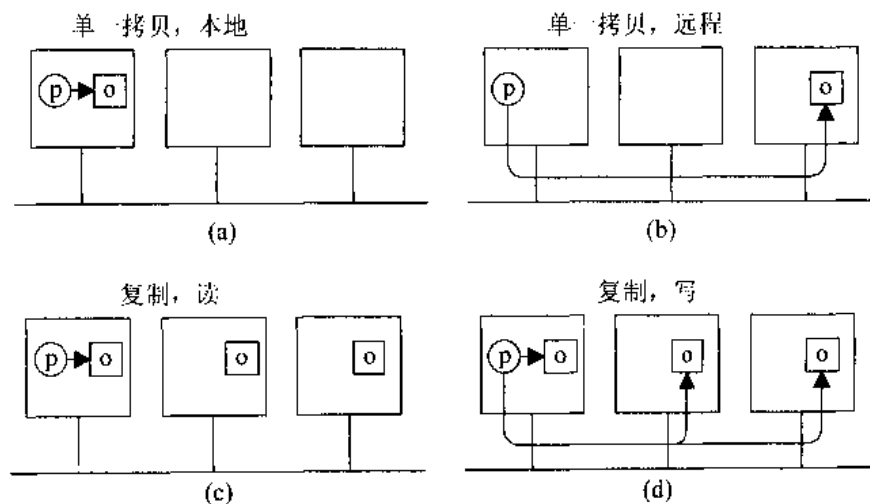


图 6-40 执行对象 O 上的操作的四种情况

注意广播原语必须是可靠的，也就是说低层自动探测并恢复丢失的消息。Amoeba 系统是 Orca 的原形，它就具有这个特征。算法将在第 7 章详细说明，我们先在这里简单总结一下。每条要广播发送的消息都发送给特殊的称为定序器 (sequencer) 的进程，它给该消息分配一个序列号，然后用不可靠的硬件将之广播出去。如果进程发现序列号出现间隔，就说明丢失了消息，它将采取措施恢复。

若系统无可靠广播（或根本没有广播），则更新通过两阶段的初始拷贝算法来实现。执行更新的进程首先给对象的初始拷贝发送消息，锁定并更新它。初始拷贝分别给所有拥有该对象的其他机器发送消息，要求它们锁定拷贝。当所有机器都确认设置了锁时，初始进程进入第二阶段，发送另一个消息告诉这些机器更新并解锁对象。

即使两个进程试图同时更新同一对象也不会发生死锁，因为其中之一会首先得到初始拷贝并锁定它，其他请求必须排队直到对象被释放。在更新过程中，对象的所有拷贝均被锁定，所以其他进程不会读到原来的值。这种锁定保证了以全局唯一的次序执行所有操作。

注意到这个运行期系统使用更新算法，而不像大多数基于分页的 DSM 系统那样使用置无效算法。大部分对象相对较小，因此发送一条更新消息（新值和参数）的代价不会高于发送置无效消息的代价。更新的最大优点是允许一系列的远程读操作，而不需获取对象或远程执行操作。

下面让我们简单看一下如何决定对象为单一拷贝状态还是复制状态。开始，Orca 包含一个进程，它拥有所有对象。当它执行 *fork* 操作时，所有其他机器就获知该事件，并

得到所有子孙的共享参数的当前拷贝。每个运行期系统计算复制每个对象与否的期望代价。

计算时需要知道读对写 (reads to writes) 的期望比率。编译器通过检查程序估算这个值,并考虑到循环中的访问次数大于其他的访问次数而 if 语句中的访问次数则是小于其他的访问次数。通信代价也是要平衡的一个因素。例如,读写比率为 10 的对象在广播网络中应被复制,但读写比率为 1 的对象在点对点式网络中的状态应为单一拷贝,机器上的单一拷贝可完成大部分写操作。更多详细信息请见 (Bal 和 Kaashoek,1993)。

所有运行期系统都执行相同的计算,因此它们得出相同的结论。如果所有机器需要一个只存在于一台机器上的对象,它就被传播出去。如果它已被复制但不再为最佳选择时,那么除一台以外的所有机器抛弃其拷贝。对象可通过这种机制迁移。

现在让我们看一下如何获得顺序一致性。对于单一拷贝状态的对象,所有操作都是顺序的,这就获得了顺序一致性。对于复制对象,不管是通过可靠有序广播还是通过初始拷贝算法,写操作都是完全有序的。不管怎样,它们对于写操作的顺序都有一致的认同。读是本地的并可以用任意的方式和写交叉而不影响顺序一致性。

假设可靠有序的广播可通过发送两条消息实现,就如在 Amoeba 中,Orca 方法效率很高。不管某操作改变了多少本地变量,都是操作结束后才发送结果。如果将每个操作看作临界区,效果和释放一致性下的一样——在每个临界区结束时广播。

我们可以使用不同优化算法。例如,不同于操作后同步,可以在操作开始时同步,正如在 entry 和 lazy 释放一致中一样。这样做的优点是如果进程在一个共享对象上重复执行操作(如循环),直到其他进程需要这个对象才需广播发送它。

另一优化算法是当写操作后的广播没有返回值时(如堆栈例子中的 push),不挂起调用者。当然,这种算法应该透明。由编译器提供的信息使其他优化算法成为可能。

总之,分布式共享内存的 Orca 模型综合了好的软件工程经验(封装的对象),共享数据,简单语义和自然方式的同步。在许多情况下,这可能会和在释放一致性中一样有效。当底层硬件和操作系统提供有效有序的广播且应用程序中对共享对象的访问读写比率较高时运行会更好。

6.7 比较

让我们简单比较一下以上所述的不同系统。IVY 只是试图通过在网络中而不是磁盘中分页来模仿多处理机。它提供了一种常见的内存模式——顺序一致性,而且可以不加修改地运行现存的多处理机程序。唯一的问题是性能。

Munin 和 Midway 要求编程人员标记共享变量并使用更弱的一致性模型,以此来提高性能。Munin 是基于释放一致性的,在每次调用释放操作时,将所有修改的页传送到共享那些页的其他进程上,而不是在锁改变拥有权时才通信。

Midway 只支持一种共享数据变量类型,而 Minun 有四种(只读、迁移、写共享和普通)。另一面,Midway 支持三种一致性模型(入口、释放和处理器),而 Munin 仅支持释放一致性。有多种共享数据类型和多种协议究竟好不好还可以继续讨论。在我们完全理解这一点之前还应作更多研究。

最后,共享变量的写操作的检测方法不同。Munin 使用 MMU 硬件捕获写操作,而在 Midway 中,系统可以提供特殊的编译器纪录写操作或和 Minun 一样使用 MMU 硬件。不需要缺页错误流,特别是在临界区中,无疑是 Midway 的一个优点。

现在让我们把 Munin 及 Midway 和基于对象的共享内存的 Linda-Orca 变体作比较。在 Munin 和 Midway 中,实现同步和数据访问是编程人员的任务,而在 Linda-Orca 中,它们紧密的结合在一起。在 Linda 中,因为 *in* 和 *out* 在其内部处理自己的同步,所以编程人员犯同步错误的危险性就减少了。相似的,在 Orca 中调用共享对象的操作时,运行期系统完全控制了锁,编程人员甚至不用知道它。条件同步(被认为是互斥同步)不是 Munin 和 Midway 模型的一部分,所以要求编程人员清晰地执行所有的操作。相反,这是 Linda 模型(当元组不在时阻塞)和 Orca 模型(根据 *guard* 表达式阻塞进程)的一个组成部分。

简而言之,Munin 和 Midway 的编程人员在同步和一致性的领域得到的支持较少,所以必须做更多的工作以保证其正确性。那里没有封装和方法保护共享变量,但它们在 Linda 和 Orca 中存在。另一方面,Munin 和 Midway 允许使用只作了少量修改的 C,C++ 语言,但 Linda 中的通信和一般的不一樣,Orca 就是一种新语言。就效率来说,Midway 在传输信息的数量和大小上是最优的,尽管在这三种情况下使用基本上不同的编程模型(开放 C 代码、对象和元组)会导致基本不同的算法,它们可能会影响效率。

6.8 小结

多 CPU 的计算机系统使用两种策略:共享内存或不共享。共享内存机制(多处理机)易于编程但较难建立,而不共享内存的机制(多计算机)难于编程但易于建立。分布式共享内存是这样一种机制:它在多计算机上模拟共享内存使多计算机的编程变得简单。

小型的多处理机多是基于总线的,但较大的系统是交换式的。大规模系统使用的协议要求复杂的数据结构和保持缓存一致性的协议。NUMA 多处理器强制软件决定在哪个机器上放置哪页,从而避免了这种复杂性。

正如 IVY 中所做的,DSM 系统的直接实现是可能的,但其性能通常低于多处理机,所以,研究者们研究了各种内存模式,试图获得更好的性能。衡量所用模型的标准是顺序一致性,即所有进程以相同的顺序看见所有内存访问。因果一致性、PARM 一致性和处理器一致性都弱化了这样的概念,即所有的进程都应以相同的顺序看见所有内存访问。

另一种方法包括弱一致性、释放一致性和入口一致性,也就是说在任意时间内存不一定一致,但是编程人员可以采取一些措施,如进入或离开临界区,使内存一致。

为提供 DSM,一般使用了三种技术。第一是模拟多处理机内存模式,给每个机器一个线性分页的内存。根据需要页可在机器间移动。第二是使用对共享变量进行注释的普通编程语言。第三是基于诸如元组和对象这样的高层编程语言。

本章中,我们研究了通向 DSM 的五种不同途径。IVY 本质上和虚拟内存一致,当发生缺页错误时页从一台机器移动到另一台机器。Minun 使用多重协议和释放一致性,允许共享单独变量。Midway 和 Minun 类似,但它使用入口一致性而不是普通情况下的释放一致性。Linda 显示了这几种方法的另一端,它使用了抽象的元组空间,远不同于具体的分页。Orca 支持这样的模型:数据对象可在多个机器上复制并通过保护的方法访问,对编程人

员来说,对象看起来是顺序一致的。

习 题

- (1) 一 Dash 系统的内存有 b 字节,分为 n 簇。每一簇中 p 个进程。缓存块的大小是 c 字节。给出一个公式计算目录所需的内存量(每个目录项不包括两个状态位)。
- (2) Dash 有必要区分 CLEAR 和 UNCACHED 状态吗?能不能去除其中一个状态?不管怎样,在这两种情况下,内存都应有数据块的更新拷贝。
- (3) 书中指出图 6-5 的缓冲拥有权协议有很多差别很小的变体。描述这样的变体,并说明和书中的相比,优点是什么。
- (4) 为什么在 Memnet 中需要“属主内存”的概念,而 Dash 中不需要?
- (5) 在一 NUMA 多处理机系统中,本地内存访问需要 100ns,远程访问需要 800ns。某程序在执行过程中共访问内存 N 次,其中百分之一是访问页 P 。该页本来在远程,拷贝到本地要用 C ns。不考虑其他进程的使用,在怎样的情况下页应被拷贝到本地?
- (6) 在内存一致性模型的讨论中,我们经常提到软件和内存的协约。为什么需要这样的协约?
- (7) 一多处理机有一条总线。能不能实现严格一致性内存?
- (8) 图 6-11 (a) 说明了顺序一致性内存的一个例子。对 P_2 作少量改动,使它破坏顺序一致性。
- (9) 在图 6-12 中,001110 是不是顺序一致性内存的合法输出?并加以解释。
- (10) 在 6.2.2 节的最后,我们讨论了一个形式模型,顺序一致性内存上的每一个操作集可通过一字符串 S 模拟, S 可以表示所有单独进程顺序。对在图 6-14 中的进程 P_1 和 P_2 ,给出 S 的所有可能值。忽略进程 P_3 和 P_4 及它们对进程的访问。
- (11) 在图 6-18 中,顺序一致性允许六种可能的语句交错。将它们一一列出。
- (12) 为什么在图 6-10 (b) 中 $W(x)1 R(x)0 R(x)1$ 是不合法的?
- (13) 在图 6-12 中,000000 是不是 PARM 一致性内存的合法输出?解释原因。
- (14) DSM 系统中,(eager)释放一致性的大多数实现方法是在调用 release 时同步共享变量,而不是在调用获取时同步。为什么还需要获取调用?
- (15) 在图 6-25 中,假设页拥有者通过广播定位。在什么样的情况下,页因为读操作而被发送?
- (16) 在图 6-26 中,进程可通过页管理者和页拥有者接触。它或许需要拥有权或页本身,它们是独立的量。所有这四种情况是否存在(当然不包括请求者不要求页和所有权的情况)。
- (17) 假设两个变量 A 和 B ,恰好位于基于分页的 DSM 系统的同一页上。然而,它们都不是共享变量。是否会发生错误共享?
- (18) 为什么 IVY 用置无效的机制而不是更新机制实现一致性?
- (19) 一些机器在一个原子操作中只有一条指令,它用内存中的一个字交换寄存器的值。用这种指令,可以实现保护临界区的信号量。使用这种指令的程序是否能工作于基于分页的 DSM 系统?如果可以,在什么样的条件下它们能有效工作?

-
- (20) 如果 Minun 进程在临界区外修改共享变量，会发生什么情况？
 - (21) 在 Linda 中调用 *in*，匹配时不需要任何查找或哈希算法。举一例说明。
 - (22) 当 Linda 通过在多个机器上复制元组实现时，需要一协议删除元组。给出一协议的例子，在该协议中，当两个进程要同时删除同一元组时不产生竞争。
 - (23) 考虑运行在具有硬件广播的网络上的 Orca 系统。为什么读操作和写操作的比例影响性能？

第7章 实例研究 1: Amoeba

本章将讨论分布式操作系统的第一个实例: Amoeba。下一章, 将研究第二个实例: Mach。Amoeba 是一个分布式操作系统, 它使得多个 CPU 和输入/输出设备“集中”在一起, 像一台计算机一样工作。同时, 它还在需要的地方为并行编程提供便利条件。本章将讲述 Amoeba 系统的目标、设计以及系统的实现。要了解 Amoeba 系统的更详细信息, 请参阅论文: Mullender 等, 1990; 和 Tanenbaum 等, 1990。

7.1 Amoeba 介绍

本节将对 Amoeba 系统进行初步介绍。首先简要介绍它的历史和当前研究目标, 然后将看一个典型的 Amoeba 系统的结构。最后, 将开始学习 Amoeba 系统的软件, 包括系统内核和服务程序。

7.1.1 Amoeba 的发展史

Amoeba 系统起源于 1981 年荷兰阿姆斯特丹 Vrije 大学的一项关于分布式计算和并行计算的研究课题。虽然有很多人参与了这个系统的设计和实现工作, 但它主要是由 Andrew S.Tanenbaum 和他的三位博士生 Frans Kaashoek、Sape J.Mullender 和 Robbert van Renesse 设计的。到 1983 年, Amoeba 的最初原型 Amoeba 1.0 投入使用。

从 1984 年开始, Amoeba 研究组解体, 在阿姆斯特丹的数学和计算机科学中心成立了第二个小组, 由 Mullender 领导。在随后的几年里, 在一个由欧共体资助的广域分布式系统的课题里, 合作研究工作扩展到英国和挪威的一些节点 (site)。这些研究使用的是 Amoeba 3.0 系统, 该版本不同于早期的版本, 它是基于远程进程调用(RPC)的。通过使用 Amoeba 3.0, 使位于 Tromsø 的客户能透明地访问设在阿姆斯特丹的服务器; 反之亦然 (反向也一样)。

经过多年的发展, Amoeba 系统已具有部分 UNIX 模拟、组通信和一个新的初级协议 (low-level-protocol) 等特征。本章讨论的版本为 Amoeba 5.2 系统。

7.1.2 研究目标

通常在分布式操作系统方面的研究项目一般从一个已存在的系统着手 (如 UNIX 系统), 然后增加诸如网络和共享文件系统等特征, 从而使系统具有分布式系统的某些特性。Amoeba 采用了一个不同的研究方法, 它是开发一个全新的系统, 从而不必考虑对现存系统的兼容性。后来为了避免重写大量代码的烦琐工作, 加入了 UNIX 仿真软件包。

该研究项目的最初目标是建立一个透明的分布式操作系统。从一般用户看来, 使用 Amoeba 系统好像在使用一个像 UNIX 那样的分时操作系统。用户可以完成登录、编辑和编译程序、拷贝文件等操作。不同之处在于这些操作都是通过网络利用多台计算机来实现

的。这些计算机包括进程服务器、文件服务器、目录服务器、计算服务器以及其他的服务
器，但用户并不会意识到这些细节。在终端用户看来，它就好像是一个分时操作系统。

Amoeba 与其他分布式操作系统的一个重要区别在于它没有“宿主机”的概念。当
一个用户登录后，他就进入了整个分布式系统，而不是进入某台特定的计算机。系统中的
机器没有所有者。登录时启动的初始外壳是运行在任意一台机器上，但是当启动命令时，
一般来说它们不和外壳运行在同一台机器上，系统将自动查找负载最轻的一台机器来运行
每一条新的命令。当然，在一个比较长的终端对话期间，根据负载情况，使得任何一个用
户其运行的进程将在系统内的所有机器上统一地被分布。在这方面，Amoeba 是高度位置
透明的。

换句话说，系统把所有资源看作一个整体并管理它们。除了在某个很短的时间段是
为特定的进程使用之外，它们并不属于特定用户的。这种模式试图提供一种所有分布式系
统设计人员都希望实现的最高的透明度。

以 *amake* 命令为例，该命令与 UNIX 系统 *make* 命令相对应。当用户键入 *amake* 命
令后，由系统（不是用户）决定所有的编译操作是顺序执行还是并行执行，以及在哪个或哪
些计算机上执行。但这些操作对用户来说都是透明的。

Amoeba 系统的第二个目标是提供一个分布式并行程序设计的测试环境。虽然某些
Amoeba 用户像使用分时操作系统一样使用它，但大多数用户利用它进行分布式并行计
算、语言、工具及应用的实验。Amoeba 系统为用户提供了基本的并行环境。实际上，Amoeba
用户主要是以各种形式进行分布式并行计算的研究。Orca 就是为此而在 Amoeba 上设计
并实现的一种语言，有关 Orca 及其应用可参考：Bal, 1991; Bal 等, 1990; Tanenbaum
等, 1992。Amoeba 系统是用 C 语言编写的。

7.1.3 Amoeba 的系统结构

在讨论 Amoeba 结构之前，先描述一下 Amoeba 系统的硬件配置情况，因为它和目前
多数系统的组织多少有些不同。Amoeba 主要是针对如下两个硬件假设设计的：

- (1) 系统拥有大量的 CPU；
- (2) 每个 CPU 有数十兆的内存。

这两个假设目前在某些地方已经成为现实。而且在不久的将来，也许几乎在所有的
公司、研究机构或政府都将采用这种结构。

存在这种系统结构的依据是实现一种直接以简单的方法满足协同大量 CPU 的需要。
也就是说，当你能向每个用户提供 10 个或 100 个 CPU 时你将采取何种方案。其中一种方
案就是给每个用户单独分配具有 10 个或 100 个节点的多处理机。

虽然给每人一个多处理机是可能的，但在经费有限的情况下这并不是一种有效的方
法。在大多数时间中，几乎所有的处理机都处于空闲状态。而某些用户可能希望运行巨大
的并行程序，却不能利用空闲的 CPU 时间片，因为这些 CPU 在其他用户的个人计算机中。

Amoeba 没有采用这种个人多处理机解决方法，而是以图 7-1 所示的模型为基础。这
种模型把所有的计算资源存放在一个或多个处理机池中。一个处理机池由大量的 CPU 组
成，每个处理机有其自己的内存和网络连接。它们不需要共享内存，如果它是现存的，它
可能用于在通过内存到内存的拷贝来传输消息和通过网络发送消息之间进行最佳选择。

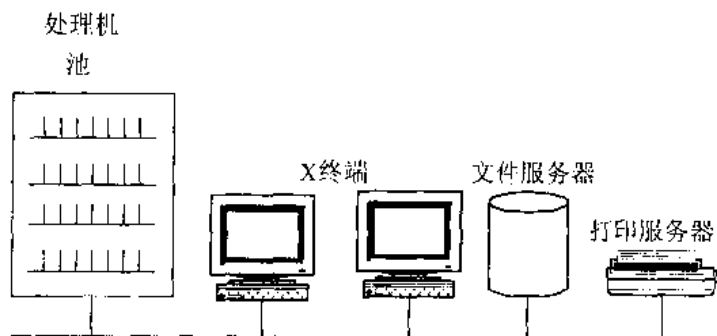


图 7-1 Amoeba 系统体系结构

在 CPU 池中的 CPU 体系结构可以不同，例如这些 CPU 可能由 680x0, 386 或 SPARC 芯片组成。Amoeba 设计成可以处理多种结构和异构的系统，也可以让进程的某个子进程在不同体系结构上运行。

池中的处理机 (pool processor, 下面简称为“池处理机”) 并不属于某个用户。当用户键入一条命令后，操作系统自动选择一个或多个处理机运行该命令。当命令执行完后，终止该进程，将其资源收回并放入池中，并等待下一个命令，或下一个用户。如果处理机不能满足需求，则每个处理机是分时的，新的进程将分配给负载最轻的处理机执行。注意，这个模型和目前使用的系统是完全不同的，在当前的系统中，每个用户拥有自己的个人工作站来从事他所有的计算。

假设将来的系统必须具有大容量内存，这一点将会影响系统在许多方面的设计。为了得到高性能系统，不惜以更多的内存为代价，以空间来换取时间。在后面我们将讲到这方面的例子。

Amoeba 系统的第二个要素是终端。用户通过终端访问系统。通常 Amoeba 终端是 X 终端，包括一个大的位图屏幕和一个鼠标。此处，运行 X-windows 的个人计算机或工作站也可用作终端。虽然 Amoeba 并不禁止在终端上运行程序，但该模型的用意是将计算集中放到一个公用的池中，使它们能够被充分利用，而终端应该相对便宜些。

池处理机仅由一个带有网络连接的控制板组成，没有键盘、显示器或鼠标，并且电源装置可由许多控制板分享，因而比工作站相对便宜。利用为 100 个用户购买 100 台高性能工作站的价钱可以购买 50 个高性能池处理机和 100 台 X 终端(当然这与经济状况有关)。由于池处理机只是在需要时才进行分配，所以一个空闲的用户只连接一台便宜的 X 终端，而不是一台昂贵的工作站。池处理机模型和工作站模型的适应性比较已在第 4 章中讨论过。

池处理机并不必须是单板机。现存的个人计算机或工作站的一部分也可被指定作为池处理机，它们也可放在一个不同的地方。实际上，池处理机是与其地理位置无关的。就像我们将要讨论的那样，池处理机甚至可以位于不同的国家。

Amoeba 系统的一个重要组成部分是专用服务器，如文件服务器，它是由于硬件或软件的需要运行在一个单独的处理机上的。有时一个服务器可以运行在池处理机上，可以在需要时才启动它，但考虑到性能因素，最好让它一直处于运行状态。

服务器提供服务。服务是对服务器能为客户所作工作的一个抽象定义。它描述了客

户的请求和服务器的返回，但并没有详细说明多少台服务器可以一起工作来提供该服务。通过这种方式，系统可以采用多台服务器协同工作，提供容错服务。

以目录服务器为例。系统并不阻止用户在查找一个文件名时在池处理机上启动一个新的目录服务器，但是这样做效率极低。因此为了提高性能，一个或多个目录服务器在专用的计算机上一直保持运行状态。系统管理员决定一直运行的服务器个数。

7.1.4 Amoeba 微内核

在讨论完 Amoeba 硬件模型后，我们将讨论 Amoeba 的软件模型。Amoeba 软件由两部分组成：运行在每一个处理机上的微内核，以及提供大多数传统操作系统服务的服务器。

Amoeba 内核运行在系统的所有计算机上，同样的内核可以运行在处理机池、终端(看成是一台计算机,而不是 X 终端)以及特定的服务器上。如图 7-2 所示，内核有四个主要功能：

- (1) 管理进程和线程；
- (2) 提供低级内存管理支持；
- (3) 支持通信；
- (4) 处理低级 I/O。

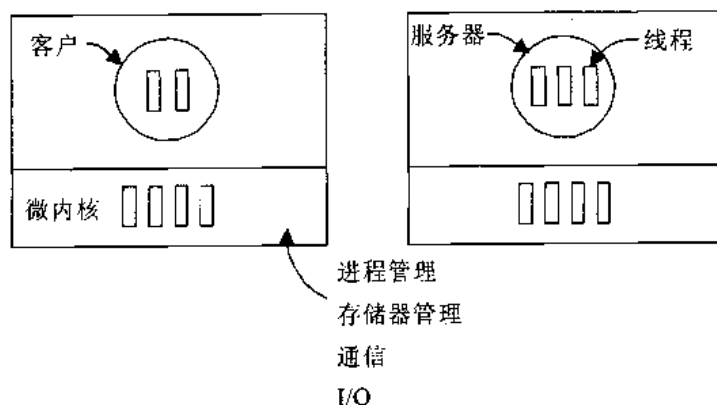


图 7-2 Amoeba 软件结构

我们将依次介绍上述四个功能。

像大多数操作系统一样，Amoeba 支持进程的概念。此外，Amoeba 也支持在一个单独的地址空间中对多个线程的控制。拥有一个线程的进程基本上和 UNIX 中的进程一样。这种进程包含一个地址空间、多个寄存器、一个程序计数器和一个堆栈。

对拥有多个线程的进程来说，虽然它有一个被所有线程共享的地址空间，但逻辑上每个线程有它自己的寄存器、程序计数器和堆栈。事实上，一个进程中线程的集合类似于 UNIX 中独立进程的集合，只不过这些线程共享一个独立的地址空间，而 UNIX 不是。

文件服务器是一个典型的多线程应用。对文件的每个请求都分配一个独立的线程来执行。该线程可能首先处理该文件的请求，接着因等待磁盘操作而阻塞，然后继续执行。通过将服务器分割成多个线程，虽然每个线程可能因 I/O 而阻塞，但每个线程都是顺序执行的。然而，所有的线程可以访问一个共享的软件缓存。线程之间可以通过信号量或互斥

的方法来防止同时访问一个共享的缓存。

内核的第二个任务是提供低级内存管理。线程可以按段 (segment) 来实现内存块的分配和释放。段是可以读和写的, 并且可以映射入或映射出调用线程的进程的地址空间。一个进程必须有一段内存, 也可以有多段。段可以根据进程的需要作为正文段、数据段或堆栈段。操作系统不按某一个固定的方法进行段的分配。

内核的第三个任务是处理进程间通信。它提供两种通信方式: 点对点通信 (point-to-point communication) 和组通信 (group communication)。这两种方式的紧密集成使它们极其相似。

点对点通信基于如下模型: 客户机向服务器发送一条消息, 然后阻塞, 直到服务器返回一条应答消息。这种请求/回答交换方式是基础, 在这个基础上, 可以建立进程之间的通信。

另一通信形式是组通信。它允许一条消息从一个源头发送到多个目的地。在消息丢失或其他错误的情况下, 软件协议为用户进程提供了可靠的、容错的组通信。

内核的第四个功能是管理低级 I/O。每个连接到计算机上的 I/O 设备, 在内核中都对应有一个设备驱动程序。设备驱动程序管理设备的所有 I/O 操作。驱动程序和内核的连接, 不能动态加载。

设备驱动程序通过标准的请求和应答消息与系统的其他部分进行通信。需要和磁盘驱动程序进行通信的进程, 如文件服务器, 发出它的请求消息, 然后收到其应答消息。一般用户并不感知它正在和一个设备驱动程序进行通信。就这点而言, 它只是在和某处的一个线程通信。

点对点通信和组通信均使用了 FLIP 特殊协议。这个协议是网络分层协议, 它是为满足分布式计算的需要而专门设计的。在复杂的网际技术中, 它既可以处理单一计算, 又可以处理复合计算。我们将在后面的章节中予以讨论。

7.1.5 Amoeba 服务器

服务器进程完成内核不能完成的所有工作。这种设计缩小了内核的大小, 同时增强了它的灵活性。由于没有将文件系统和其他标准服务建立在内核中, 因此它们可以很容易地被改动, 并且针对不同的用户数可以同时运行多个版本。

Amoeba 是基于客户/服务器模型的。通常客户程序由用户编写, 而服务器程序由系统编程人员编写。但如果用户需要的话, 用户也可以自由编写服务器程序。整个软件设计的核心是对象概念。对象是一种抽象数据类型, 每个对象由一些封装的数据和定义在这些数据上的操作组成。例如, 文件对象包括 READ 等操作。

对象由服务器管理。当进程创建一个对象时, 管理该对象的服务器为该客户提供加密能力。如以后使用该对象, 必须使用正确的权限。系统中的所有对象, 包括硬件和软件, 均以加密的方式被命名、保护 and 管理的。以这种方式支持的对象包括文件、目录、内存段、屏幕窗口、处理机、磁盘和磁带驱动器。对所有对象提供一个统一接口, 从而对对象的管理变得规范和简单。

所有标准的服务器程序在开发包中有存根过程 (stub procedure)。当使用服务器时, 客户通常仅仅调用存根过程, 存根过程处理参数, 发送消息, 然后一直阻塞到响应返回。

这种机制对用户隐藏了所有的实现细节。系统为需要编写自己的存根过程的用户提供了存根编译器。

所有服务器程序中，最重要的可能要算文件服务器，称为子弹服务器（bullet server）。它提供管理文件的各种原语，包括创建文件、读文件、删除文件等操作。和大多数文件系统不同，子弹服务器所创建的文件是不可改变的，文件一旦创建，它就不能修改，只能删除。不可改变的文件避免了在进行文件复制时的竞争，使得文件复制变得更容易了。

另一个重要的服务器是目录服务器（directory server），由于历史的原因也叫 soap 服务器。目录服务器管理目录和路径名，并将它们映射到权能上。进程读文件时，首先请求目录服务器寻找路径名。一旦查找成功，目录服务器返回该文件（或其他对象）的权能。以后对文件的操作就不再通过目录服务器，而是直接通过文件服务器。将文件系统分成两部分增加了灵活性，同时由于每一部分仅仅管理一种类型的对象（目录或文件），使实现变得更为简单了。

其他的标准服务器可以实现对象复制，启动进程，错误监控服务器及和外界通信等。用户服务器可以根据特定的应用任务完成特定的功能。

本章其余部分编排如下。首先讨论对象和权能，因为它们是整个系统的中心。然后我们将研究内核，重点研究进程管理、内存管理和通信。最后将探讨几个主要服务器，包括子弹服务器、目录服务器、复制服务器和管理服务器。

7.2 Amoeba 中的对象（object）和权能（capabilities）

支持 Amoeba 服务器和它们提供的服务的基础是对象这一统一的概念。对象由一组数据及一组对数据进行的操作组成。本质上，对象是一种抽象数据类型。对象是被动（无源）的。它们不包含处理、方法或其他活动实体。每个对象由服务器进程管理。

要完成对一对象的操作，客户机和服务器之间要进行一次远程过程调用，指定对象和完成的操作及其所需的可选参数。服务器完成这些工作并返回应答。这些操作是同步执行的，即当客户机在初始化一个对服务器的 RPC 调用后，客户线程一直阻塞直到返回应答。然而，同一进程中的其他线程仍保持运行。

客户机不知道它所使用的对象的位置，也不知道管理这些对象的服务器的位置。服务器和客户机还可能在同一台计算机上运行，也可能在局域网的不同机器上运行，甚至可能在相隔数千里之外的机器上运行。此外，虽然多数服务器以用户进程的形式运行，但一些低级的服务器，如段服务器和进程服务器，在内核中以线程的方式运行，客户也不知道这一区别。无论对用户服务器还是内核服务器，本地服务器还是远地服务器，其 RPC 协议在所有情况下都是相同的。一个客户机只参与它要做的事情，而不关心对象存储在什么地方以及服务器在哪里运行。一个特定的目录包含可以访问的所有文件服务器，同时指定一个缺省的文件服务器。如果出错，用户可以覆盖该缺省值。通常，系统管理员将缺省值设置为本地。

7.2.1 权能（Capability）

系统通过使用权能这种统一的方法来命名和保护对象。为了创建一个对象，客户机

与某一服务器之间进行 RPC 调用，说明它想做什么。服务器创建该对象并将其权能返回给客户。客户在对该对象的后续操作中，必须通过权能识别该对象。一个权能实际上是一个长的二进制数。在 Amoeba5.2 中，权能的格式如图 7-3 所示。

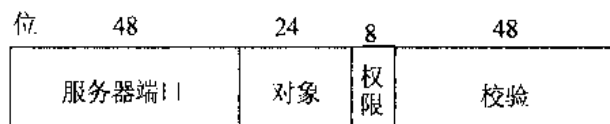


图 7-3 Amoeba 中的一个权能

当客户想操作对象时，它调用一个存根过程，该过程建立一条包含对象权能在内的消息，然后捕获到内核。内核从权能中提取服务器端口值，并在缓存中查找它以确定服务器在哪台计算机上。如果端口不在缓存中，通过广播确定位置。在后面的章节中将讨论这个问题。端口号是可以到达服务器的有效逻辑地址。服务器端口和某一特殊的服务器（或一系列服务器）相对应，而不是与某一台的计算机相对应。如果服务器移到一台新的计算机上，其服务器端口仍不变。许多服务，如文件服务器，它是众所周知的，而且总保持不变。定位服务器的唯一方法就是通过它的端口。

内核忽略权能中的其他信息，而将它们传给服务器由服务器处理。对象域用作服务器识别特定的对象。例如，一个文件服务器可能管理上万个文件，对象号指定操作的是哪一个文件。在某种意义上，文件权能中的对象域和 UNIX 系统的 i 节点相似。

权限（Right）域规定了权能的拥有者可以对对象进行的操作。例如，虽然一个对象支持读和写，一个特定的权能通过 Right 域规定不能读。

校验（Check）域用作检查权能的有效性。用户进程直接操作权能。如果没有保护，就没有办法防止用户进程伪造权能。

7.2.2 对象保护

下面讲述用于对象保护的基本算法。当服务器创建一个对象时，随机选择校验域，将它保存在权能以及服务器的表中。在新权能中的权限位被初始化后，将该权能返回给客户机。当客户请求操作对象时，又将权能回传给服务器，这是服务器验证校验域。

为了创建一个保密的权能，客户机将权能传回给服务器，并附带一些权限位。服务器从内部表中得到原始的校验域，将它和新的权限位进行异或操作（它一定是权能中权限的子集），然后利用一个单向函数运算结果。单向函数，如 $y=f(x)$ ，给出参数 x 值容易求出 y 值，但如果只给出 y ，却不容易找到 x 值。

服务器建立一个新权能，它和对象字段含有相同的值，不包括权限字段的权限表和单向函数在校验字段的输出。然后，新权能返回给调用者。客户机可以将新权能发送给另一个进程，因为权能完全由用户空间管理。

图 7-4 说明了生成受限权能的方法。在本例中，权能的拥有者只保留了一个权限位，关闭了其他权限。例如，该受限的权能可能只允许对对象进行读操作，但是没有其他权限。由于对对象的逻辑操作因对象的不同而不同，因此 Rights 域的含义也随对象的不同而不同。

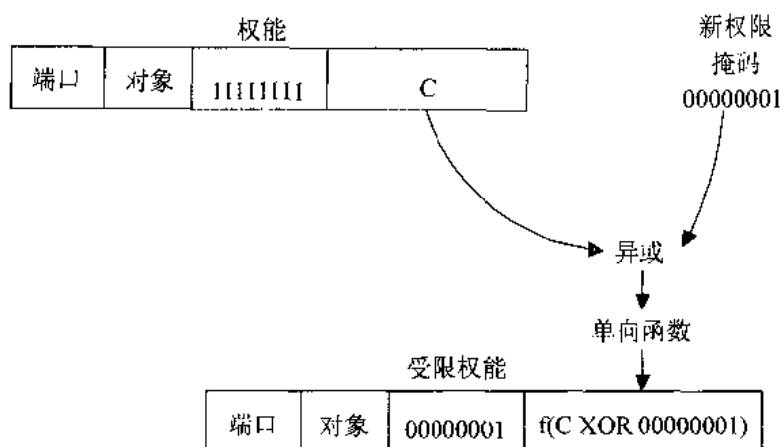


图 7-4 权能拥有者生成受限权能

当限定权能返回给服务器，服务器检查权能的权限位，由于权限域至少有一位无效，该权能不是所有者权能。服务器接着从表中取出原来的随机数，与限定权能的 *Rights* 位进行异或，并通过一个单向函数计算其结果，如果结果和检查域值相同，权能就为有效的。

显然，如果一个用户企图使用他不具有的权限，该算法只需简单把该权能置为无效。将保密权能中的 *Check* 字段反过来转化为自变量（图 7-4 中的 $C \text{ XOR } 00000001$ ）是不可能的，因为函数 f 是个单向函数（“单向”意思是反过来算法不存在）。通过加密算法保护权能不被篡改。

在整个 Amoeba 系统中，权能始终被用作命名对象并保护它们。该机制实现了统一的命名和保护模式，它也是完全位置透明的。对一对象进行操作，不必知道对象的所在地。事实上，即使能得到该信息，也没有方法利用它们。

Amoeba 并不使用访问控制矩阵进行认证。该保护模式几乎不需要任何管理。然而，在不安全的网络环境下可能需要进行加密，使权能在网上不会意外地泄露出去。

7.2.3 标准操作

虽然许多对象的操作取决于对象的类型，但有些操作适合于大多数对象，如表 7-1 所示。其中一些操作需要设置特定的权限位，其他的操作可被任何客户利用一个该对象有效的权能进行调用。

在 Amoeba 系统中，创建一个对象后又把它放弃是可能的，因此需要某一机制清除某些不再被访问的老对象。采用的方法是让服务器定期运行一个垃圾收集程序，清除已经经过 n 次垃圾收集循环仍未被使用的对象。AGE 调用启动一个新的垃圾收集循环。TOUCH 调用告诉服务器该对象仍在被使用。当对象进入目录服务器后，它们被定期引用，使垃圾收集者不能收集它。某些标准操作的权限，如 AGE，只在系统管理员拥有的权能中出现。

表 7-1 适合于对大多数对象的标准操作

调用	说明
Age	执行一次垃圾收集周期
Copy	复制对象、返回副本的权能

(续表)

调用	说明
Destro	销毁对象、释放空间
Getpar	根据服务器获取参数
Info	取得简要说明目标的 ASCII 字
Restri	生成对象新的受限能力
Setpar	根据服务器设置参数
Status	从服务器获取当前状态信息
Touch	伪装对象刚被使用

COPY 操作是快速复制一个对象，但不传输对象。如果没有该操作，拷贝一个文件需要在网络上发送两次：从服务器发送到客户机，再从客户机返回服务器。COPY 操作也能获取一个远程对象或把对象发送到一个远程机上。

DESTROY 操作删除对象。显然，它需要适当的权限。

GETPARAMS 和 SETPARAMS 调用将服务器看作一个整体，而不是看成一个特殊的对象来处理。通过这两个操作，系统管理员可以读或写那些控制服务器操作的参数。例如，通过这种机制，可以确定进行处理机选择时使用的算法。

INFO 和 STATUS 调用返回状态信息。前者返回一个描述对象的简短的 ASCII 字符串。串中的信息取决于服务器，但总体上，它指明对象的类型和其他有用的信息（例如，告诉文件的大小）。STATUS 得到服务器的总体信息，比如空闲内存数量。这些信息帮助系统管理员更好管理系统。

RESTRICT 调用为对象产生一个具有某些权限的新权能。

7.3 Amoeba 中的进程管理

Amoeba 中的进程基本上由一段地址空间和在该地址空间中运行的线程集合组成。一个仅有一个线程的进程，根据它的行为和它能做的事情，和 UNIX 中或 MS-DOS 中的进程大致上相似。本节将解释进程和线程是如何工作的，以及它们是怎样完成的。

7.3.1 进程

Amoeba 中，进程是一个对象。父进程创建一子进程时，和创建其他对象一样，父进程得到该子进程的句柄。利用该句柄，父进程可以挂起、重新启动、激活以及释放该子进程。

Amoeba 中进程的创建和 UNIX 不同。在 UNIX 中，通过复制父进程的方式创建子进程，这种方式在分布式环境下是不合适的。FORK 建立一个拷贝开销较大，而 EXEC 是直接替换原进程。在 Amoeba 中，在一个指定的处理机上建立一个随即启动的内存影像是可能的。这一点，Amoeba 的内存创建和 MS-DOS 非常类似。和 MS-DOS 不同的是，父进程能和子进程并行运行，且可以创建任意数量的子进程。子进程又可再创建它自己的子进程，从而形成一个进程树。

Amoeba 从三个不同的层次来管理进程。最低一层是进程服务器，它们是运行在每一台计算机上的核心线程。为了在一给定计算机上建立一进程，另一进程对那台计算机上的进程服务器执行 RPC 调用，并提供必须的创建信息。

其上一层是一组库过程，它为用户程序提供更方便的接口，提供几种不同风格的界面。它们是通过调用下一层的接口来实现的。

最后，创建进程的一个最简单的方法是使用运行服务器，它决定新进程运行在哪台计算机上。在后面的章节我们将讨论运行服务器。

某些进程管理调用使用一个称为进程描述符的数据结构，此数据结构提供运行进程的信息。进程权能（见图 7-5）中的一个域表明该进程可以运行的 CPU 结构。在异构环境中，该域保证 386 的二进制程序不会运行在 SPARC 芯片上，以此类推。

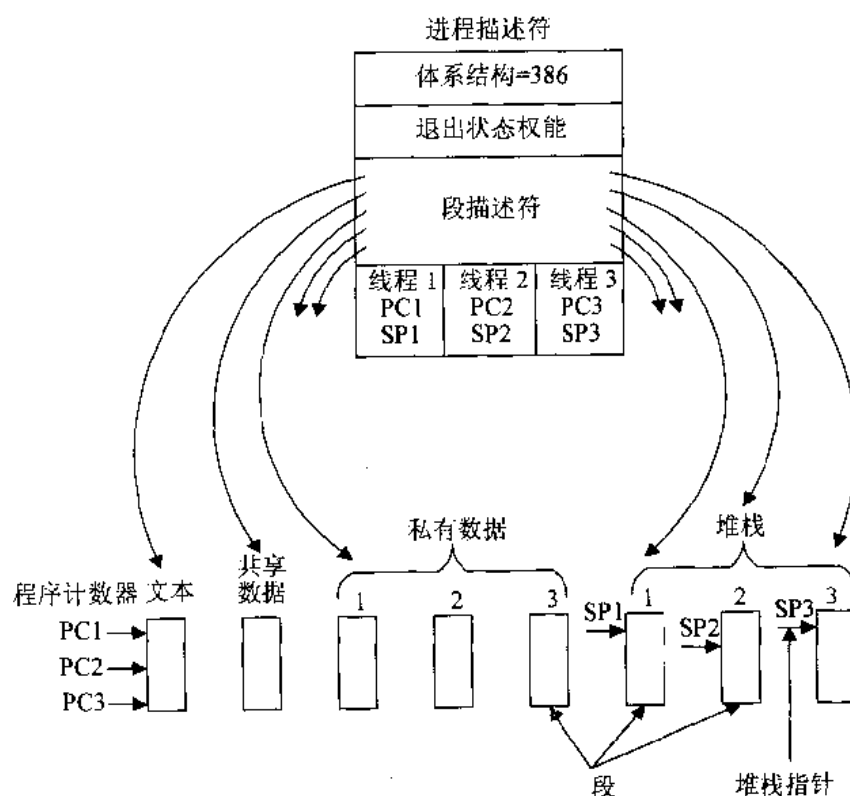


图 7-5 进程描述符

另一个域包含进程拥有者权能。当一进程终止或休眠时，RPC 将利用该权能报告该事件。它也包含所有进程段的权能，总起来定义它的地址空间，还包含所有线程的描述符。

最后，进程描述符中包含该进程的每个线程的描述符。线程描述符的内容是与结构无关的，它包含线程的程序计数器和栈指针，也包含运行该线程所必须的其他附加信息，包括寄存器、线程状态和各种标志。新的进程在它们的进程描述符中只包含一个线程，但是进程可能在调用 *stun* 过程之前已建立了附加线程。

低层进程接口大约由 6 个库过程组成。这里我们考虑 3 个过程。第一个是 *exec* 过程，它是其中最重要的一个，它有两个输入参数，以及进程服务器权能和进程描述符。它的功

能是对指定的进程服务器执行 RPC 调用, 要求服务器运行该进程。如果调用成功, 新进程的权能将返回给调用者, 这样调用者就可以通过该权能控制该进程。

第二个库过程是 *getload*。在任何时刻它返回有关 CPU 速度的信息、当前负载情况及空闲内存的数量。运行服务器利用这些信息决定执行运行新进程的地方。

第三个库过程是 *stun*。父进程通过 *stun* 调用挂起子进程。父进程可以将进程权能传给调试器, 在交互调试过程中, 调试器可以休眠或重启该进程。系统支持两种休眠: 正常休眠和紧急休眠。它们的不同之处是在中断时, 如果在一个或多个远程进程调用中进程被阻塞, 所作出的反应不同。正常休眠, 进程向服务器发送一条它正在等待的消息, 信息内容是: “我被停止。立即完成你的工作并给我一个答复。” 如果该服务器阻塞, 等待另一服务器, 则消息被进一步传播直到末端服务器。最末端的服务器直接返回一个规定的错误消息。通过这种方式, 所有的挂起的 RPC 被终止, 服务器也正常结束。

紧急休眠立即停止进程, 也不向正在为该进程工作的服务器发送任何消息。由服务器完成的计算变为孤独的。当服务器完成并返回应答后, 这些应答最终将被抛弃。

高级的进程接口不需要进程完全成型的描述符。*newproc* 调用使用三个参数, 分别为二进制文件名、参数指针和环境数据指针, 这一点和 UNIX 类似。附加参数为初始状态提供更详细的控制。

7.3.2 线程

Amoeba 支持一个简单的线程模型。当一个进程启动后, 它就拥有一个线程。进程在执行过程中, 可以创建其他线程, 已存在的线程可能被终止。因此, 线程的数量完全是动态的。当建立一个新线程后, 调用的参数规定了要运行的过程和初始堆栈的大小。

虽然在一个进程中所有的线程共享同一进程正文和公共数据, 但是每个线程有其自己的堆栈、堆栈指针和寄存器拷贝。此外, 如果一个线程打算建立并使用变量, 且该变量对所有过程是公用的, 但不能被其他线程所使用, 那么可以利用一些提供的库例程。这些变量称为全局变量。一个库例程请求分配一个一定大小的全局内存块, 并返回指针。全局内存块通过整数引用。线程可以利用一个系统调用得到其全局指针。

对线程提供三种同步方法: 信号、互斥和信号量。信号是同一进程的不同线程之间发送的异步中断。在概念上和 UNIX 信号类似, 不同之处在于它是线程之间, 而不是 UNIX 的进程之间。信号可以产生, 获取或忽略。进程之间的异步中断使用休眠 (*stun*) 机制。

线程间通信的第二种形式是互斥。互斥像二进制的信号量一样, 它可以处于加锁或解锁状态。对一个未加锁的互斥加锁, 则该互斥变成加锁状态, 调用线程继续执行。对一个已加锁的互斥加锁将引起调用的线程阻塞, 直到另一个线程打开互斥。如果多个线程等待该锁, 则只有一个线程能继续运行。除了对加锁和未加锁的互斥调用外, 还有一个试图对互斥的调用, 但是如果在一定的时间间隔内做不到, 时间将溢出并返回一个错误码。互斥是公平的, 同时考虑了线程的优先级。

线程通信的第三种方法是通过信号量计数。信号量的速度比互斥慢, 但当需要它们时是有时间的。它们在通常方式下工作, 除非有一个附加的调用使得当一段时间内不能执行完毕时, 允许 DOWN 操作。

所有的线程由内核管理。这种设计的优点是当一个线程执行 RPC 时, 内核可以阻塞

该线程，同时调度该进程的另一已准备好的线程。线程调度是通过优先级实现的，内核线程具有比用户线程更高的优先级。正如进程所希望的，线程可以实现抢先式的或非抢先式的（例如，线程继续运行，直到阻塞）。

7.4 Amoeba 中的内存管理

Amoeba 有一个非常简单的内存模型。一个进程可以获得它所需的任意数量的内存段，这些内存段可以位于进程虚拟地址空间的任何地方。内存段不能交换或采用请求页式算法，因此，一个进程必须完全驻留在内存中才能运行。此外，虽然使用 MMU 硬件，但每个内存段在内存中是连续存放的。

虽然此种设计在现在看来可能有些不常见，但它考虑了性能、简单性以及经济性三个方面。让整个进程一直驻留在内存中，可以使 RPC 调用的运行速度更快些。当需要发送大的块数据时，系统不仅知道所有数据在虚存中是连续存放的，而且在物理内存中也是连续存放的。这种设计节省了检查含缓冲区的所有页此刻是否都在的时间，如果它们不在，可以消除对它们的检查所必须的等待。同样，就输入而言，由于存放数据的缓冲区总是在内存中，这样输入的数据就很容易存放，且不会出现页面错误。这种设计思想保证 Amoeba 在进行大量 RPC 调用时具有很高的传输速度。

内存管理的第二个设计目标是简单性。不需进行换页或交换使得系统简单得多，内核更小且更便于管理。但是，第三个原因才使得前两个原因成为可行。内存变得更加便宜，这样在几年后，所有的 Amoeba 计算机将拥有数十兆的内存。如此大的内存可以大大地减少换页和交换的需要，即当把大程序装入小计算机内运行时。

7.4.1 段 (segment)

进程可以通过一些它们可用的调用来管理段。在这些调用中最重要的是创建段、释放段及读段和写段。当创建一个段时，调用者得到一个该段的权能。通过段权能可以对该段进行读写操作和其他涉及该段的调用操作。

段在创建时有一个初始大小。其大小可以在进程运行期间改变。段也可能有一个初值，它来自另一个段或一个文件中的数据。

由于段可读可写，因此可以利用段来构造一个主存文件服务器。首先，服务器创建一个尽可能大的内存段。段的最大长度可以通过请求内核来确定。该段将当作一个虚拟硬盘使用。然后服务器按照文件系统对该段进行格式化，放入任何跟踪文件所需的数据结构。然后打开它进行工作，接受客户机的请求。

7.4.2 段映射 (mapped segments)

Amoeba 中虚拟地址空间由段组成。当启动一个进程时，它必须至少拥有一个段。然而，一旦进程开始运行，它可以创建其他段，并将它们映射到它的任何未使用的虚拟地址空间。图 7-6 给出了一个有 3 个当前已映射的内存段的进程。

进程也可以释放段映射。此外，进程还可以指定虚拟地址范围，请求释放该范围内的段映射，这样一来那些地址就不再合法了。当一个段或一个地址范围被释放时，返回一

个段权能，但该段仍可被访问，或甚至以后再次映射，可能映射在一个不同的虚拟地址空间。

一个段可以同时映射到两个或多个进程的地址空间。这样允许这些进程可在一个共享内存中运行。然而，如需要共享内存，通常一个进程创建多个线程会更好。使用不同进程的主要原因是为了更好地保护，但是，如果两个进程共享一内存，保护通常不太有必要。

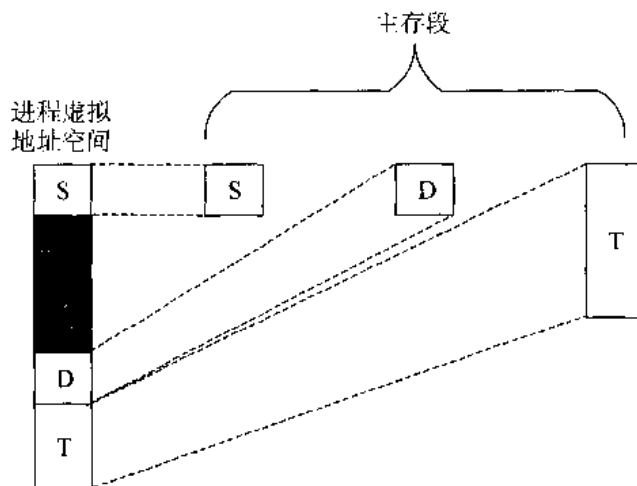


图 7-6 进程有三个段映射到自己的虚拟地址空间

7.5 Amoeba 的通信

Amoeba 支持两种形式的通信：使用点到点消息传递的 RPC 和组（group）通信。在最底层，RPC 由一个带应答消息的请求消息组成。组通信使用硬件广播或如可能的话使用多播（multicasting）。然而内核用一个独立的消息透明地模拟组通信。本节将讨论 Amoeba RPC 和 Amoeba 组通信，以及用来支持它们的 FLIP 协议。

7.5.1 远程过程调用(RPC)

在 Amoeba 系统中，通常的点到点通信是由客户发送给服务器一个消息而后服务器发回给客户一个应答消息组成的。客户不可能仅发送一个消息，然后处理其他的工作，除非绕过 RPC 接口，那只能在一些特殊环境下才可行。发送请求消息的 RPC 原语，自动阻塞调用者直到应答返回。这样在程序上强制一定数量的结构。单独的 *send* 和 *receive* 原语可看作是分布式系统对 *goto* 语句的应答：并行的意大利通心面（非直线）式编程。可以通过用户程序来避免它们，而且它们仅用于通信需求独特的语言运行时期系统。

每个标准服务器定义了一个客户可以调用的过程接口。这些过程库例程是一些将参数打包成消息然后调用内核原语来发送消息的存根。在消息传递的过程中，存根过程及调用线程被阻塞。一旦返回应答，存根过程将状态和结果返回给客户。虽然内核级原语实际上是涉及消息传递，但利用存根过程使得这种机制对编程人员来说更像 RPC。我们称这种基本的通信原语为 RPC，而不是更为精确一些的“请求/应答报文交换”。

客户线程要利用服务器线程进行一个 RPC 调用，它必须知道服务器的地址。寻址可以通过允许任何一个线程选择一个 48 位的随机数，称端口，这个随机数用于作为接收消息的地址。进程中的不同线程可以使用不同的端口。端口仅仅是一个线程的逻辑地址，它没有相应的数据结构和数据。另一方面，它和 IP 地址或以太网地址相似，除了它不被捆绑到某一特殊的物理位置上外。在每个权能的第一个域给定了管理对象的服务器端口（参见图 7-3）。

RPC 原语

RPC 机制使用下面三个主要的内核原语：

- (1) `get_request` —— 表明服务器是否准备好监听一个端口；
- (2) `put_reply` —— 当服务器要发送一个应答消息时由服务器执行；
- (3) `trans` —— 从客户向服务器发送一个消息，然后等待一个应答。

服务器使用前两个原语。第三个原语由客户用于传输一个消息并等待应答。三个都是真正的系统调用，也就是说，它们不是通过发送一个消息给一个通信服务器线程来工作的。（如果进程能够发送消息，为什么它们为发送消息而不得和一个服务器联系呢？）然而，和通常一样，用户通过库过程来访问系统调用。

当服务器为了等待一个即将进来的请求而想休眠时，它调用 `get_request`。这个过程有三个参数：

```
get_request(&header, buffer, bytes)
```

第一个参数指向一个消息头，第二个参数指向一个数据缓冲区，第三个参数表示该缓冲区的大小。该调用和 UNIX 或 DOS 中的类似。

```
read(fd, buffer, bytes)
```

第一个参数指示将要读的内容，第二个参数表示用于存放数据的缓冲区，第三个参数表示该缓冲区的大小。

消息在网络上传输时，它包含一个头和一个（可选的）数据缓冲区。头是一个如图 7-7 所示的固定的 32 位数据结构。`get_request` 的第一个参数告诉内核即将到来的头的存放位置。此外，在调用 `get_request` 前，服务器必须初始化头的端口域以包含它将要监听的端口。这样，内核就知道哪个服务器将监听哪个端口了。即将到来的头覆盖掉由服务器初始化的头。

消息一到达，服务器就解除阻塞状态。服务器通常首先检查消息头，以便从中发现更多的关于请求的信息。签名域为认证而被保留，但目前不会使用。

RPC 协议不指定其他域，因此服务器和客户可以协商以任何它们希望的方式去使用它们。通常的约定如下所述。大多数对服务器的请求包含一个权能以指明将要操作的对象。许多应答也将权能作为一个返回值。私有部分通常用来存放权能的最右边三个域。

大多数服务器支持对它们对象的多个操作，如：读，写和释放操作（destroying）。在请求中的命令域用来指示需要何种操作。在应答中它用来告诉该操作是否成功，如果失败，它返回出错原因。

端口(6)
签名(6)
私有部分(10)
命令(2)
偏移量(4)
大小(2)
附加(2)

图 7-7 Amoeba 中所有请求和应答消息的头格式
(圆括号中的数字给出了域的字节大小)

最后三个域存放参数，如果需要的话。例如，当读一个段或文件时，它们表示开始读取数据的位置在对象中的偏移量和所要读取的字节数。

注意，在许多操作中，不需要或不使用缓冲区。再一次请求读操作时，对象权能、偏移量和大小都存放在消息头中。请求写操作时，缓冲区包含要写的数据。另一方面，对读操作的应答包含一个缓冲区，而对写操作的应答则不包含。

服务器在处理完请求后，它调用

```
put_reply(&header, buffer, bytes)
```

向客户返回应答。第一个参数指出消息头，第二个指出缓冲区，第三个指出该缓冲区的大小。如果服务器调用 *put_reply* 时，先前没有调用一个相匹配的 *get_request*，则 *put_reply* 调用就将因错误而失败。同样，两个连续的 *get_request* 也会失败。这两个调用必须正确匹配。

现在让我们从服务器转向客户。为了执行一个 RPC，客户应产生如下调用的存根过程：

```
trans(&header1,buffer1,bytes1,&header2,buffer2,bytes2)
```

前三个参数描述即将离开的请求的消息头和缓冲区的信息，后三个参数描述即将到来的应答的同样的信息。*trans* 调用发送请求后，阻塞客户直到应答返回。这样迫使进程严格遵守客户/服务器 RPC 通信模式，类似于结构化程序设计技术阻止编程人员做那些会导致程序的结构很差的事情（如使用无限制的 *goto* 语句）。

如果 Amoeba 真正完全按上述描述工作，网络入侵者可能模仿一个服务器通过在该服务器的端口上调用 *get_request*。所有端口毕竟是公开的，因为客户要建立与服务器的连接必须知道它们。Amoeba 采用加密方法解决了这个问题。每个端口实际上是一对端口。*get-port* 是一私有的，只有服务器知道，*put-port* 对全世界都是公开的。这两个端口通过一个单向函数根据关系：

$put_port = F(get_port)$

建立联系。这个单向函数实际上和用于权能保护的函数是一样的，但没有必要一样，因为这两个概念是不相关的。

当服务器调用 *get_request* 时，内核计算对应的 *put-port*，然后存放在正在被监听的端口表中。所有的 *trans* 请求使用 *put-port* 端口，当一个包到达一个机器时，内核将信息头中的 *put-port* 与端口中 *put-port* 进行比较，检查是否一致。由于 *get-port* 从不出现在网络上，也不能通过公众所知的 *put-port* 得到，因此这种模式是安全的。这种模式如图 7-8 所示，在 Tanenbaum 等人 1986 年发表的论文中描述得更为详细。

Amoeba 至多支持一次语义。换句话说，当一个 RPC 执行时，系统保证一个 RPC 永远不会第二次执行，即使服务器崩溃并快速重启。

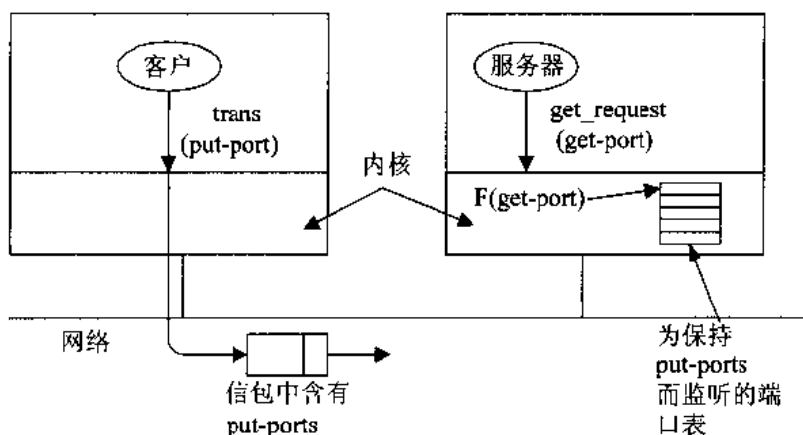


图 7-8 输入端口和输出端口的关系

7.5.2 Amoeba 的组通信

RPC 不是 Amoeba 支持的唯一通信形式。它 also 支持组通信。Amoeba 中的组由一个或多个进程组成，这些进程共同完成某项任务或提供某种服务。一个进程可以同时属于多个组。组是封闭的，也就是只有组成员可以向组广播。客户通常利用组的一个成员执行一个 RPC 调用来访问该组所提供的服务。然后那个成员在组中利用组通信决定哪个组成员将完成什么任务，如果有必要的话。

这种设计比开放式组结构提供了更大程度上的透明性。这种设计背后的思想是客户通常通过 RPC 和独立的服务器进行交互，因此它们也应该可通过 RPC 和组进行交互。替代物——开放式组使用 RPC 同单个服务器进行交互，而使用组通信同组服务器进行交互——透明性要差得多。（如果都使用组通信将损失许多前面已讨论过的 RPC 调用的优点。一旦决定组外客户用 RPC 实现和组的通信（实际上是同组中的一个进程通信），开（放）式组就不必要了，闭组完全能满足这种需要，而且易于实现。在 Amoeba 中，用于组通信的可用操作如表 7-2 所示。

表 7-2 Amoeba 组通信原语

调用	说明
CreateGroup	创建新组并设置参数
JoinGroup	将调用者加入组中
LeaveGroup	将调用者从组中移走
SendToGroup	给组中所有成员可靠地发送
ReceiveFromGr	阻塞直到从组来的信息到达
ResetGroup	进程崩溃后复位恢复

组通信原语

CreateGroup 创建一个组，并返回一个组标识符，这个组标识符在其他的调用中用来标识所需的组。调用参数指定了组的各种大小和所要求的容错级（组能容忍多少个不发挥作用的成员并能继续正确工作）。

JoinGroup 和 *LeaveGroup* 允许进程加入组和离开已经存在的组。*JoinGroup* 的参数之一是一个小消息，这个消息发送给所有该组的成员以申明新成员的存在。类似地，*LeaveGroup* 的参数之一是另一个小消息，该消息发送给所有成员向它们说再见并祝它们在未来的活动中好运。发送小消息的意义是使得所有的成员了解它们的同事，万一它们感兴趣的话，例如，如果某些成员崩溃，可以重新构造组。当组的最后一个成员调用 *LeaveGroup* 时，它关掉灯，该组就将释放。

SendToGroup 自动向一个指定组的所有成员广播一个消息，不管消息的丢失、有限的缓冲区和处理机的崩溃。Amoeba 支持全局时序，如果两个进程同时调用 *SendToGroup*，系统保证所有的组成员以同样的顺序收到消息。保证了这一点，编程人员就可以指望它了。如果两个调用是完全并发的，在 LAN 上成功发送的第一个包所对应的调用就宣称是第一个。根据第 6 章中讨论的语义，这种模式符合顺序一致性，而不是严格的一致性。

ReceiveFromGroup 尽力从它的一个参数所指定的组获取一个消息。如果消息没有到达（即，当前被内核缓冲），则该调用者一直阻塞到有消息到达为止。消息一到达，调用者马上取得该消息。协议保证在没有处理机崩溃时，消息不会不可恢复地丢失。

最后一个调用，*ResetGroup*，用于从崩溃中恢复组。它指定新组必须有的最少成员个数。如果内核能够和必不可少个数的进程建立连接并重建该组，该调用返回新组的大小。否则，该调用失败。在这种情况下，组的恢复就指望用户程序。

Amoeba 可靠的广播协议

现在让我们来看 Amoeba 是如何实现组通信的。Amoeba 系统特别适合运行在支持多播或广播的局域网上（或像以太网，都可以）。为简单起见，我们仅讨论广播，尽管实际上，利用多播实现可以避免干扰那些对发送消息不感兴趣的计算机。我们假设硬件支持广播是不错的，但并不是万无一失的。在现实中，丢包是很少见的，但接收溢出偶尔会发生。由于不能轻易地忽视这些错误发生，因此必须设计协议来处理它们。

形成组通信的实现基础的主要思想是可靠的广播发送。通过这种方式，意味着当一个用户进程广播发送一个消息时（如，用 *SendToGroup*），用户提供的消息就正确地传递给该组的所有成员，尽管硬件可能丢失包。为简单起见，我们假设每一个消息适合一个单独的包。暂

时, 我们假设处理机不会崩溃。以后我们将考虑不可靠的处理机的情况。下面给出一个大概的描述。详细请参见(Kaashoek 和 Tanenbaum, 1991; Kaashoek 等, 1989)。其他的可靠广播发送协议在如下资料中讨论: Birman 和 Joseph, 1987a; Chang 和 Maxemchuk, 1984; Garcia-Molina 和 Spauster, 1991; Luan 和 Gligor, 1990; Melliar-Smith 等, 1990; Tseung, 1989。

在可靠广播发送中所需的硬件/软件配置如图 7-9 所示。所有机器的硬件通常是相同的, 而且它们运行完全相同的内核。然而, 当应用程序启动时, 其中一个机器就被选为定序器 (就像一个委员会选一个主席一样)。如果该定序器机器后来崩溃了, 则余下的机器就重选一个新的定序器。有许多选举算法, 如选择网络地址最高的进程。在本章的后面我们将讨论容错性。

可用来实现可靠广播发送的事件的顺序总结如下:

- (1) 用户进程发生对内核的陷阱并向它传送消息;
- (2) 内核接收消息并阻塞用户进程;
- (3) 内核给定序器发送一个点对点的消息;
- (4) 当定序器获得消息后, 它分配下一个可用的顺序号, 将该顺序号存放在为保存该顺序号的一个消息头域中, 并广播发送该消息 (和该顺序号);
- (5) 当发送内核看到该广播消息时, 它释放调用进程, 让它继续执行。

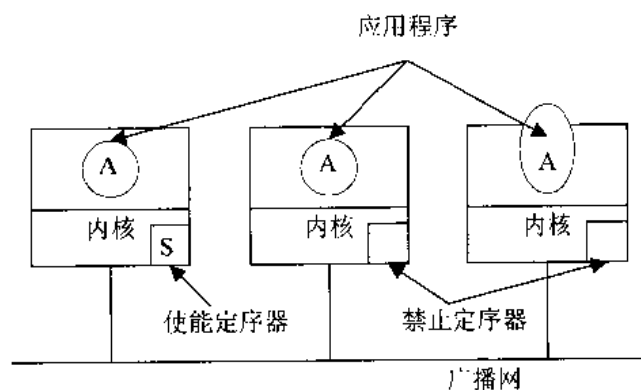


图 7-9 Amoeba 中的组通信系统结构

现在让我们来详细地看一看这些步骤。当一个应用程序进程执行一个广播原语时, 如, *SendToGroup*, 发生一个对内核的陷阱。内核然后阻塞该调用者并构造一个包含内核提供的消息头和应用程序提供的数据的消息。消息包含消息类型 (这种情况是 *Request for Broadcast*), 内核接收的最后的广播的序号 (通常称为捎带确认响应), 和一些其他的信息。

内核给定序器发送消息使用正常的点对点消息, 并同时打开一个记时器。如果广播消息在记时器有效时间之内返回, 发送内核就关闭记时器并将控制权返回给调用者。在实践中, 百分之九十九以上的时间这种情况都会发生, 因为现代的局域网是非常可靠的。

另一方面, 如果广播在记时器有效时间内没有返回, 内核就认为该消息或者广播已经丢失。任何一种情况下, 内核都重新传递消息。如果原始消息丢失, 不会造成任何影响, 第二次 (后来的) 努力将以通常的方式触发广播发送。如果消息到达定序器并且该消息是广播消息, 但是发送丢失了该广播消息, 定序器将检测出重新传递的是一个复本 (从消息标识符) 并只告诉发送者一切正常。该消息就不会再一次被广播发送了。

第三种可能是广播在超时之前返回，但它是错误的广播。当两个进程同时试图广播时，就会发生这种情况。其中之一 *A*，先到达定序器，它的消息被广播。*A* 看到广播释放它的应用程序。但是，它的竞争者 *B* 看到 *A* 的广播，意识到它不能先广播。但是 *B* 知道它的消息可能会到达定序器（因为很少会丢失消息），它将会排队等候，并在其后广播。这样，*B* 接受 *A* 的广播，并继续等待它自己的广播返回或者它的定时器超时。

现在考虑，当 *Request for Broadcast* 到达定序器时，会发生什么情况。首先检查消息是否是重发的，如果是，就像前面所提到的一样，告知发送者已经广播了。如果消息是新的（通常情况），就将下一个序列号分配给它，并为下一次将定序器计数器加 1。消息和它的标识符被存在历史缓冲区中，然后广播消息。消息也会传到正运行在定序器机器上的应用程序（因为广播不能在发送广播的机器上引起中断）。

最后，让我们考虑一下当内核收到广播时会发生什么。首先，将它的序列号与最近一次收到的广播的序列号比较。如果比上一次的大 1，就没有丢失任何广播，因此就把消息传给应用程序，假设该应用程序正在等待。如果该应用程序不在等待，就把它压入缓冲区，直到应用程序调用 *ReceiveFromGroup*。

假设新收到的广播的序列号是 25，而前一个序列号是 23。内核马上察觉它丢失了 24 号广播，它就会给定序器发送一个点到点消息，请求它秘密重发丢失的那个消息。定序器从历史缓冲区取出这个消息后发送出去。当消息到达时，内核处理 24 和 25，并按数字的顺序将它们传到应用程序。这样，丢失消息的唯一影响是一个微小的时延。即使有一些消息丢失，所有的应用程序也会以相同的顺序看到所有的广播。

图 7-10 说明了可靠的广播协议。这里运行在机器 *A* 上的应用程序向它的内核发送一个要广播的消息 *M*。内核发送该消息给定序器，这个消息分配的编号为 25。现在就向所有机器广播消息（包含编号 25），也传送到运行在定序器上的应用程序。图中这个广播消息用 *M25* 代表。

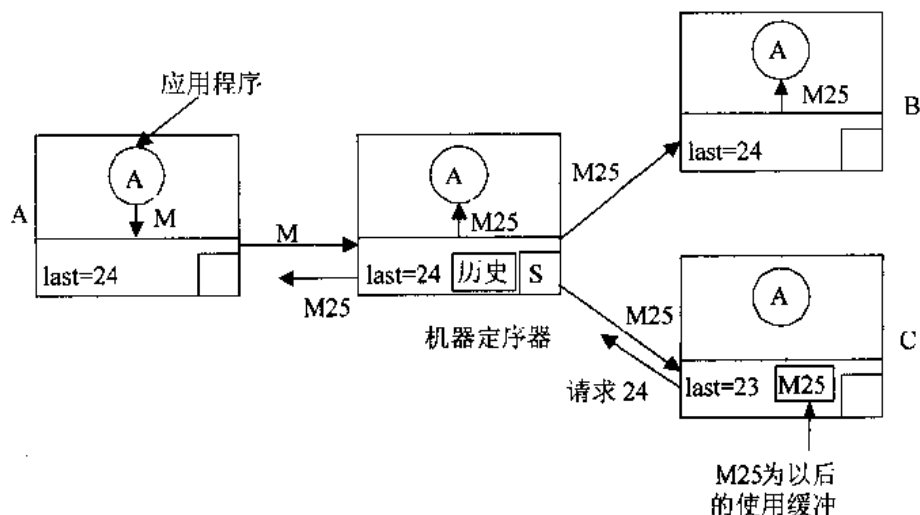


图 7-10 机器 *A* 的应用程序发送信息给定序器，定序器在信息上增加了一个序号（25）并将其广播。在 *B* 端正常接收，但是在 *C* 端将一直等待 24，这时已经丢失了，但可以从定序器再次获得

消息 M_{25} 到达机器 B 和 C 。在机器 B 上, 它的内核看到它已经处理了包括 24 在内的所有广播, 所以它马上把 M_{25} 传给应用程序。而在 C 中, 上一个到达的广播是 23 (24 一定丢了), 它就将 25 缓存在内核中, 并向定序器发送一个请求重发 24 的点到点消息。只有在应答返回并传给了应用程序后, M_{25} 才会接着传送。

现在看一下历史缓冲区的管理。如果不采取任何措施, 历史缓冲区很快就会被充满。但是如果定序器知道所有的机器都已经正确地收到了 0 到 23 的广播, 它就可以将这些广播从它的历史缓冲区中删除。

定序器为发现这个信息提供了几种机制。基本的机制是, 送到定序器的每个 *Request for Broadcast* 消息携带一个捎带确认响应 k , 意味着包括 k 在内的所有广播都被正确接收了。这样, 定序器保留一个捎带确认表, 这个表用机器号来索引, 为每个机器说明最后一次收到的是哪一个广播。无论何时历史缓冲区被充满, 定序器都能够通过查找该表, 找到最小值。然后它就可以安全地丢弃这个值及其以下的所有消息。

如果一个机器很长一段时间没有请求, 定序器就不知道它的状态。为了让定序器知道它的状态, 当在一段时间内它没有发送任何广播消息, 需要发送一个短的确认消息。而且, 定序器可以发送 *Request for Status* 消息, 让所有其他机器给它发送消息, 该消息给出按顺序接收的广播的最大数。这样, 定序器就可以更新捎带确认表, 然后缩短历史缓冲区。

实际上, *Request for Status* 消息很少出现, 但确实也会出现。这样, 即使没有消息丢失, 可靠的广播所要求的平均消息数就比 2 稍大了。当机器数增加时, 影响会稍微大一些。

有一个关于该协议的细微设计点需要澄清。发送广播有两种方法。方法 1 (上述的) 中, 用户给定序器发送点到点消息, 然后由定序器广播该消息。在方法 2 中, 用户广播消息, 消息包含一个唯一的标识符。当定序器看到时, 就广播一个包括该唯一标识符和新分配的序列号的特殊的 *Accept* 消息。只有发出 *Accept* 消息后, 广播才是正式的。图 7-11 比较了这两种方法。

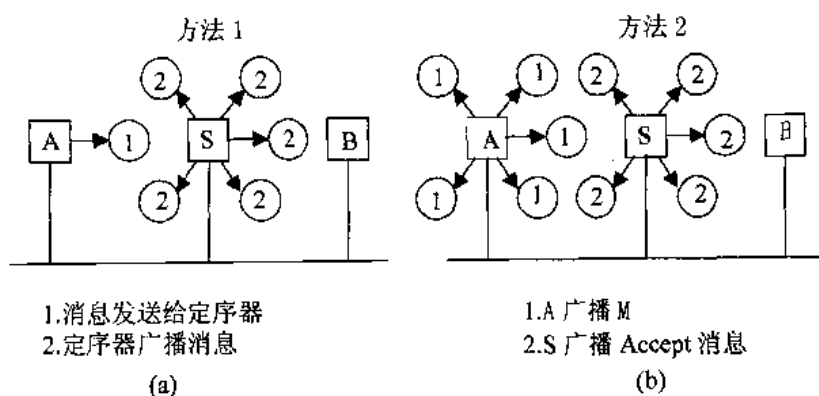


图 7-11 实现可靠广播的两种方法

这些协议在逻辑上是等价的, 但有不同的性能特点。在方法 1 中, 每个消息完整地出现在网络上两次: 一次是发往定序器, 一次是由定序器发出。这样一个 m 字节长的消息要占用 $2m$ 字节的网络带宽。然而, 只有第二个被广播了, 所以每个用户机器只被中断一次 (被第二个消息中断)。

在方法 2 中,完整的消息只在网络上出现一次,再加上一个来自定序器的很短的 *Accept* 消息,所以只占用一半的带宽。另一方面,每个机器被中断两次,一次是被消息中断,另一次是被 *Accept* 中断。这样,和方法 2 比较,方法 1 牺牲带宽来减少中断次数。可以根据消息的平均大小来选择用哪一种方法。

总的说来,这个协议允许在不可靠的网络上用刚刚超过两个的消息实现一个可靠的广播。每个广播都是不可分的,所有的应用程序以同一顺序接收所有的消息,不管丢失多少消息。可能发生的最差情况是当消息丢失时,会引起一定的时延,但这种情况很少发生。当两个进程同时试图广播时,先到达定序器的获胜。另一个看到它的竞争者的广播从定序器发出,意识到它的请求被排序并且不久后会广播,因此,它只是等待。

容错组通信 (Fault-Tolerant Group Communicate)

到目前为止,我们一直假定处理机不会崩溃。事实上,这个协议设计来支持任意组合的 k 个处理机(包括定序器)的崩溃, k (恢复度)的值在组创建时确定。 k 值越大,所需的冗余就越多,在正常情况下操作速度就越慢。因此用户必须认真选择 k 值。下面我们将概要介绍恢复算法。详细描述请见 (Kaashoek and Tanenbaum,1991)。

当一个处理机崩溃时,刚开始没有内核检测到该事件。但是,迟早某个内核会注意到发送到已崩溃计算机的消息不能得到确认,这时内核就标记该崩溃处理机为无效状态,所属的组为不可用。在该组被重建前,所有对那台计算机的组通信原语均失败(返回一个错误状态)。

发现问题后不久,得到错误返回值的用户进程调用 *ResetGroup* 进行恢复。恢复过程分为两个阶段 (Garcia-Molina,1982)。第一阶段选取一个进程作为协调者,第二阶段协调者重建组并更新所有的其他进程。那时,就可以继续正常的操作了。

在图 7-12 (a) 中,我们看到一个有六台计算机的组,5 号机是定序器,它刚刚崩溃。方框中的数字表示每台计算机正确收到的最后一个消息。0 号机和 1 号机同时检测到定序器的崩溃,都调用了 *ResetGroup* 开始恢复。这个调用导致内核给其他所有计算机发送消息,邀请它们参加恢复过程,并请它们返回它们所见的最高消息的序列号。此时,发现两个进程声称自己是协调者。见到具有最高序列号的消息的进程获胜。如果见到的序号相同,具有最高网络地址的进程获胜。这样就保证只有一个协调者。如图 7-12 (b) 所示。

一旦协调者确定,则该协调者从其他成员收集可能丢失的消息。现在,协调者是最新的,已能够成为新的定序器。它构造一个 *Result* 消息,声明自己是定序器,然后告诉其他计算机当前的最高序列号。每台计算机现在就可以请求它丢失的任何消息了。当一个组成员已是最新的时,它给新的定序器返回一个确认。当定序器收到来自所有有效成员的确认后,它知道所有消息已经被依次正确地发送到应用程序了,所以,定序器清除历史缓冲区,恢复正常的操作。

另一个问题仍然存在:如果定序器崩溃,协调者是如何得到丢失的消息的呢?答案在于 k 的值,恢复度 k 的值是在组创建时确定的。当 k 为 0 (无容错情况) 时,只有定序器维护一个历史缓冲区。然而,如果 k 大于 0, $k+1$ 台计算机连续维护最新的历史缓冲区。这样即使任意组合的 k 台计算机崩溃,仍可保证至少有一个历史缓冲区存在,并且就是这个历史缓冲区为协调者提供所需的任何消息。额外的机器只要监视网络就可以维护它们的历史记录。

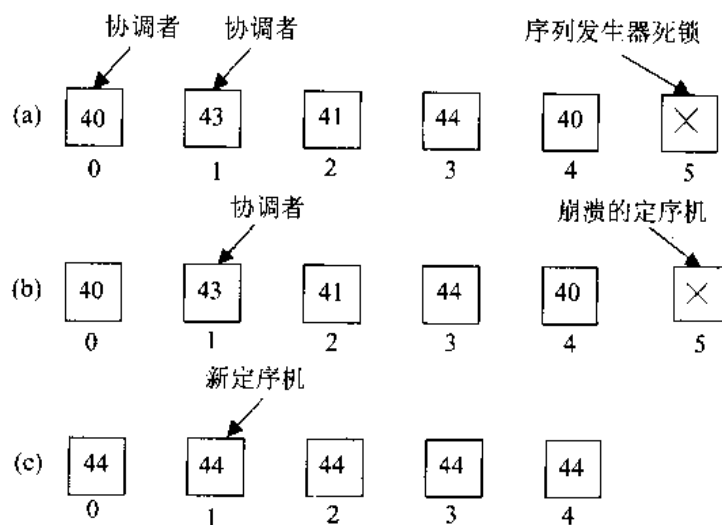


图 7-12 (a) 序列发生器崩溃
(b) 选择一个协调者
(c) 恢复

另外还有一个问题要解决。通常，当定序器已经接收并广播或证实这一消息时，*SendToGroup* 就成功结束了。如果 $k > 0$ ，该协议就不能够容忍任意 k 个机器崩溃。采用方法 2 的稍微改进的版本作为替代。当定序器看到刚刚被广播的消息 M 时，它不会像 $k=0$ 时那样马上广播一个 *Accept* 消息。相反的，它会一直等到 k 个最低序数的内核承认它们已经看到并存储了该消息，才广播 *Accept* 消息。因为现在已知有 $k+1$ 个机器（包括定序器）在它们的历史缓冲区存储了 M ，即使 k 个机器崩溃， M 也不会丢失。

和在通常情况下一样，在看到 *Accept* 消息之前，内核不会将 M 传到它的应用程序。因为只有 $k+1$ 个机器存储了 M 后才可能生成 *Accept* 消息，这就保证了如果一台机器得到了 M ，所有这些机器都会得到。这样，从任意 k 台机器的损失中恢复都总是可能的。另外，为加速 $K > 0$ 时的操作，无论何时输入进入历史缓冲区，都广播一个短的控制包以向世界宣布这个事件。

总的说来，Amoeba 的组通信模式保证了即使面临任意 k 个机器崩溃，也能以全局时序实现原子广播，这里 k 是在组创建时由用户选择的。这一机制为进行分布式编程提供了一个容易理解的基础。在 Amoeba 中它用于为 Orac 编程语言和其他设备提供基于对象的分布式共享内存的支持。它也可以有效地实现。对在一个 10-Mbps 以太网上的 68030 CPUs 的测试表明每秒连续地处理 800 个广播是可能的（Tanenbaum 等，1992）。

7.5.3 快速本地互联协议 FLIP (The Fast Local Internet Protocol)

Amoeba 使用称为 FLIP 的定制协议进行实际的消息传输。该协议处理 RPC 和组通信，在协议层次中，该协议位于 RPC 和组通信之下。在 OSI 协议中，FLIP 为网络层协议，而 RPC 更像无连接的传输或会话协议（确切的位置是有争议的，因为 OSI 是为基于连接网络设计的）。概念上，FLIP 可以被任一网络层协议替代，如 IP，尽管这样做可能造成 Amoeba

的透明性的一些损失。虽然 FLIP 是为 Amoeba 而设计的，但在其他操作系统中也是有用的。本节我们讨论它的设计和实现。

分布式系统的协议需求

在深入 FLIP 细节前，理解设计的目的是相当有用的。毕竟已有许多现存的协议，引入一个新协议无疑必须进行论证。表 7-3 列出了为分布式系统设计的协议必须满足的主要条件。首先，协议必须有效支持 RPC 和组通信。如果底层网络具有硬件广播或多播的功能，例如像以太网那样，协议将使用这一功能实现组通信。另一方面，如果网络没有其中任何一种特性，组通信也应该以完全相同的方式工作，尽管实现的方法肯定不同。

表 7-3 分布式系统协议必须满足的特性

项	说明
RPC	协议必须支持 RPC
组通信	协议必须支持组通信
进程迁移	进程必须能够携带他们的地址
安全性	进程一定不能模仿其他进程
网络管理	自动重新配置是必须的
广域网	协议必须能够在广域网上工作

协议的一个越来越重要的特征是支持进程迁移。一个进程应该能不被察觉地从一台计算机迁移到另一台计算机，甚至是到不同网络中的另一台计算机。如 OSI, X.25 和 TCP/IP 等使用机器地址标志进程的协议，使得进程迁移变得困难。因为进程不能在它移动时带上它的地址。

安全性也是一个问题。虽然 get-ports 和 put-ports 为 Amoeba 提供了安全性，但在包一级协议也应该提供一种安全机制，这样该机制可以在不具有加密安全地址的操作系统中使用。

还有一点就是网络管理，在该点上许多现存的协议都做得很差。没有必要使用繁杂的配置表来表示哪个网络连接到另外哪个网络上了。而且，如果由于网关的停止或重新启动引起配置改变，协议应该能自动适应新的配置。

最后，协议应该在局域网和广域网上都能工作。特别是相同的协议应该在两者上都是有用的。

FLIP 接口

FLIP 协议和它的相关结构满足所有这些要求。一个典型的 FLIP 配置如图 7-13 所示。图中共有五台计算机，两台在以太网上，四台在令牌环网上。每台计算机上都有一个用户进程，从 A 到 E。其中一台计算机连接到两个网，从而它自动充当网关。网关和其他节点一样，也可以运行客户和服务程序。

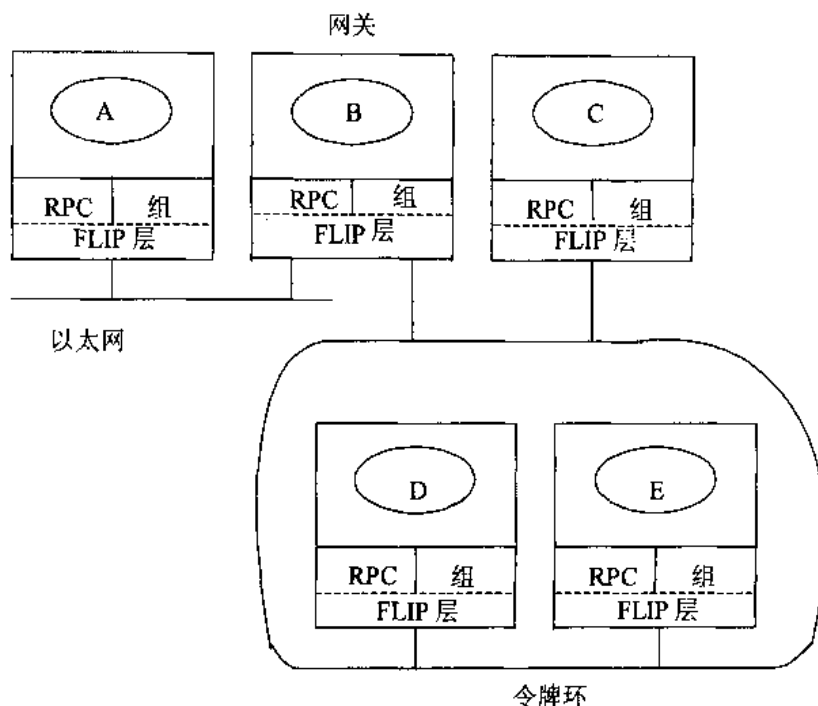


图 7-13 一个拥有五台机器两个网络的 FLIP 系统

软件如图 7-13 进行构造。内核包含两层，上层处理来自用户进程的 RPC 或组通信服务调用。底层处理 FLIP 协议。例如，当客户调用 *trans*，它发生一个对内核的陷阱。RPC 层检查消息头和缓冲区，通过它建立一个消息并将消息传递到 FLIP 层进行传输。

Amoeba 的所有低级通信都是建立在 FLIP 地址的基础之上。每个进程有且只有一个 FLIP 地址，FLIP 地址是在进程创建时由系统生成的一个 64 位的随机数。如果进程可以迁移，该进程就带上它的 FLIP 地址。如果网络被重新配置，从而所有的计算机被重新分配（硬件）网络地址或网络号，但 FLIP 地址仍保持不变。FLIP 唯一地标识了一个进程，而不是一台计算机，这样使得在 Amoeba 中的通信与网络拓扑和网络地址的改变无关。

一个 FLIP 地址实际上是两个地址：一个公有地址和一个私有地址，它们之间的关系如下：

$$\text{public-address} = \text{DES}(\text{private-address})$$

这里 DES 表示数据加密标准(Data Encryption Standard)。为从私有地址计算出公有地址，私有地址用作一个 DES key 来加密一个 64-bit 的 0 块。给定公有地址，在计算上来说找到相应的私有地址是不可能的。服务器监听私有地址，客户却发送到公有地址，和 put-ports, get-ports 的工作方式类似，只是在较低的层次上进行。

FLIP 不仅适合 Amoeba 系统，也适合其他操作系统。已经有 FLIP 的一个 UNIX 版本，没有理由不能产生一个 DOS 版本。私有地址提供安全性，公有地址模式也可以工作在利用 FLIP 进行的 UNIX 之间的通信，而与 Amoeba 无关。

而且 FLIP 已经设计好了，因此，它可以固化在硬件中，例如作为网络接口芯片的一

部分。为此，就必须指定一个与它的上一层的精确接口。FLIP 层和它的上一层（我们将称之为 RPC 层）之间的接口定义有 9 个原语，其中 7 个是用于输出信息，两个是用于输入信息。每个原语都有一个库例程来调用它。表 7-4 列出了这 9 个原语。

表 7-4 FLIP 层支持的调用

调用	说明	方向
Init	分配表插槽	↓
End	返回表插槽	↓
Register	监听 FLIP 地址	↓
Unregister	停止监听	↓
Unicast	发送点到点的信息	↓
Multicast	发送多点传送信息	↓
Broadcast	发送广播信息	↓
Receive	报文接收	↑
Notdeliver	无法投递的报文返回接收	↑

第一个，*Init*，容许 RPC 分配一个表插槽并用指向两个过程（或者是在硬件实现中的两个中断向量）的指针初始化它。当正常的或无法投递的数据包到达后，这些过程就分别被调用。在关闭计算机的过程中，*End* 调用释放插槽。

Register 调用来向 FLIP 层声明进程的 FLIP 地址。当进程启动时调用它（或者至少在第一次试图接收或发送消息时）。FLIP 层立即利用 DES 函数对私有地址进行运算，将公有地址存放在地址表中。如果一个输入数据包地址寻址到该公有 FLIP 地址，该数据包就将传到 RPC 层进行传递。*unregister* 调用从 FLIP 层的地址表中删除一项。

下面的三个调用 *unicast*、*multicast* 和 *broadcast* 分别用作发送点到点消息、多播消息和广播消息。这些调用均不提供传输的可靠性。要保证 RPC 的可靠性，必须采取认证机制。为了保证组通信的可靠性，即使面对丢失的包，仍要使用上面讨论的定序器协议。

最后两个调用 *recieve* 和 *notdeliver* 用于输入信息。第一个 *receive* 用于接收到一个其他地方产生并直接发送到该计算机的信息包。*notdeliver* 是接收一个由本计算机发出的，但因无法投递而返回的信息包。

虽然 FLIP 接口是为内核的 RPC 和广播层设计的，但对用户进程也是可见的。用户可以利用 FLIP 接口完成一些低级的通信功能。

FLIP 层的操作

通过 RPC 层或组通信层向 FLIP 层传递的包的地址是 FLIP 地址，因此 FLIP 层必须能够为实际传输将 FLIP 地址转换成网络地址。为了实现这一功能，FLIP 层维护如图 7-14 所示的一张路由表。当前该表在软件中维护，将来芯片设计者可以在硬件中实现它。

无论何时输入包到达任何一台计算机时，首先由 FLIP 层处理，FLIP 层从中获取 FLIP 地址和发送方的网络地址。包传递的跳数也已被记录。由于当数据包向前传递经过网关时跳数才增加，所以通过跳数可以得到包经过的网关数。因此跳数可以粗略地衡量数据源有多远（实际上，情况比这个要理想一些，因为可以使用慢速网络作为多个跳数。）。如果 FLIP 地址不在路由表中，则将该 FLIP 地址加入路由表。这样以后就可以使用这个表目将包发

送给该 FLIP 地址了，因为现在已经知道该 FLIP 地址的网络号和网络地址了。

FLIP 地址	网络地址	跳数	受托位	寿命

图 7-14 FLIP 路由表

在包中有一附加位，表示包已经过的路径是否都是信任网络。该位是由网关管理的。如果包通过了一个或多个非信任网络，如果需要绝对安全的话，到目的地址的包需要加密。而在信任网络中，没必要加密。

每个路由表目的最后一项表示该表目存在的时间长短。当从相应 FLIP 地址收到一个包后，它就被初始化为 0。所有的时间定期增加。通过该域，在地址表满时，FLIP 层可以清除一个合适的表目（大的数字表明该地址已有很长一段时间没有通信量了。）。

定位 put-ports

为了了解 Amoeba 中 FLIP 是如何工作的，让我们用图 7-13 的配置来看一个简单的例子。A 是客户，B 是服务器。在 FLIP 协议中，任何与两个或两个以上网络相连的机器都自动地成为一个网关，因此 B 只是碰巧运行在一个网关机器上。

当创建 B 时，内核为它选择一个新的任意的 FLIP 地址，并将它注册到 FLIP 层。启动之后，B 自己初始化，并用 *get_request* 调用它自己的 *get-port*，产生一个对内核的陷阱。RPC 层在它的 *get-port* 到 *put-port* 缓存中查找 *put-port*（或者计算它如果找不到任何表目），并记录下来一个进程正在监听哪个端口。然后就阻塞直到一个请求来临。

后来，A 在 *put-port* 调用 *trans*。它的 RPC 层在表中查找，看它是否知道监听 *put-port* 的服务器进程的 FLIP 地址。既然它不知道，RPC 层就发送一个特殊的广播包寻找这个地址。这个包有一个最大跳数设置，以保证广播局限在自己的网络中。（当网关发现一个包的当前跳数已经等于它的最大跳数，这个包就被丢弃，而不是继续向前传送。）如果广播失败，发送 RPC 层超时，就用一个更大的跳数来试，直到能够定位服务器。

当广播包到达 B 的机器时，它的 RPC 层返回一个应答宣告它的 FLIP 地址。像所有的输入包一样，这个包使得在把应答包传送到 RPC 层之前 A 的 FLIP 层要为那个 FLIP 地址创建一个表目。RPC 层现在在自己的表中创建了一个表目，将 *put-port* 映射到 FLIP 地址。然后就向服务器发请求信息。由于现在 FLIP 层已经有了服务器的 FLIP 地址的表目，它就可以建立一个包含正确的网络地址的包，并发送出去，再没有任何麻烦。对服务器的 *put-port* 的后续请求使用 RPC 层的缓存得到 FLIP 地址，使用 FLIP 层的路由表得到网络地址。这样，广播只在服务器首次被连接时使用。以后，就由内核表提供必要的信息。

总而言之，定位一次 *put-ports* 需要两次映射：

- （1）从 *put-port* 到 FLIP 地址的映射（由 RPC 层实现）；
- （2）从 FLIP 地址到网络地址的映射（由 FLIP 层实现）。

两步处理的原因是双重的。首先，FLIP 协议设计成通用协议是为了在分布式系统（包

括非 Amoeba 分布式系统)中应用。这些系统通常不使用 Amoeba 式的端口,因此从 put-port 到 FLIP 地址的映射就没有固定到 FLIP 层上。FLIP 的其他用户可能只是直接使用 FLIP 地址。

量 LAN 组成的 WAN 中效果不错。实际上, 服务器很少移动, 因此一旦服务器被广播定位, 后续的请求就可以使用缓冲表目了。利用该方法, 遍布世界的大量计算机可以以一种透明的方式协同工作。对位于调用者地址空间的线程的一个 RPC 调用和对位于世界之遙的线程的 RPC 调用是以完全相同的方式执行的。

组通信也使用 FLIP。当一个消息发向多个目的地时, FLIP 在有多播或广播功能的网络中使用硬件多播或广播。在不支持多播或广播功能的网络上, 通过发送单个消息模拟广播, 就像我们在广域网中所看到的一样。机制的选择由 FLIP 层完成, 并且在所有情况下使用相同的用户语义。

7.6 Amoeba 服务器

在 Amoeba 系统中, 大多数传统的操作系统服务 (如文件服务器) 是以服务器进程的形式实现的。尽管有可能把任意组合的服务器放在一起, 其中每个服务器有自己的模式, 但很早就决定提供一个服务器所需的专一模式以便达到一致和实现简单性。尽管是自愿的, 但多数服务器都遵守这一约定。本节将讲述这一模型和关键的 Amoeba 服务器的一些例子。

在 Amoeba 中所有标准服务器都是用一组存根过程定义的。旧的存根过程是用 C 语言手工写的, 而新的存根过程是用 AIL (Amoeba Interface Language, Amoeba 接口语言) 定义的。存根定义通过 AIL 编译器产生存根过程, 存根过程放进库中以便客户使用。实际上, 存根精确定义了服务器所提供的服务和调用服务所需的参数。在下面的讨论中, 我们将经常提到存根。

7.6.1 子弹服务器 (Bullet Server)

和所有操作系统一样, Amoeba 也有一个文件系统。与其他操作系统不同的是, Amoeba 的文件系统的选择不是由操作系统指示的。文件系统作为服务器进程的集合而运行。如果用户不喜欢标准的服务器进程, 可以编写自己的服务器进程。内核不知道, 也不会关心谁是真正的文件系统。实际上, 不同的用户可以同时使用不同的且不兼容的文件系统。

标准的文件系统由三个服务器组成: 子弹服务器 (bullet server), 它处理文件存储; 目录服务器 (directory server), 它实现文件命名和目录管理; 复制服务器 (replication server), 它处理文件复制。将文件系统分成这几个独立的部分, 可以提高灵活性并且可使每个服务器实现简单。本节我们将讨论子弹服务器, 在后续章节我们将讨论另外两个服务器。

客户进程可非常简单地调用 *create* 来生成文件。子弹服务器返回一个权能, 在后续的调用中可使用该权能来读取整个或部分文件。大多数情况下, 用户将给文件一个 ASCII 名称, 目录服务器其 (ASCII 名称, 权能) 存在目录中, 但这一操作与子弹服务器无关。

正如其名字一样, 子弹服务器被设计成相当快。其设计目标主要是运行在具有大量内存和大容量磁盘的计算机上, 而不是运行在内存很少的低档计算机上。它的结构与传统的文件服务器的结构大不相同。特别是, 它的文件是不可变的。一旦文件被创建, 它就不

能随便被修改。文件可以被删除，在它的位置可重新创建一个新文件，但该新文件具有与原文件不同的权能。我们将看到，这一特性简化了文件的自动复制操作。同时，它也特别适合在大容量、一次写的光盘上使用。

由于文件在创建后不能被修改，文件的大小在创建时就已经知道了。这一特性使得文件可以在磁盘中也可以在主存的缓存中连续存储。通过连续存储文件，一次单独的磁盘操作就可以将文件读进内存，然后通过一个单独的 RPC 应答消息就可将文件发送给用户。这些简化大大提高了性能。

该文件系统基于如下概念模型：客户在自己的内存中创建一个完整的文件，然后通过一个单独的 RPC 调用将文件传到了服务器，服务器保存文件，然后给客户返回该文件的权能，客户以后就通过该文件权能访问该文件。为了修改文件（如编辑一程序或文档），客户传回权能请求文件，然后（理想情况下）服务器通过一个单独的 RPC 将文件发送到客户的内存中。客户便可以在本地任意修改该文件。在修改结束后，客户通过一个 RPC 向服务器发送该文件，这样服务器就创建一个新文件，并返回一个新的权能。此时，客户可以请求服务器删除或将原文件作为一个备份。

作为对现实的让步，服务器也支持内存太少而不能通过一个单独的 RPC 来发送或接收整个文件的客户。当读文件时，客户可以通过指定文件的偏移量和字节数来读取文件的一部分。这一特性使客户方便地以任意大小的单元读取文件。

由于文件不能被修改，因此用几个操作来写一个文件就变得复杂了。系统通过引入两种类型的文件来解决这个问题，未提交文件（uncommitted files），它处于正在创建的过程中；提交文件（committed files），它是永久文件。未提交文件可以被修改，而提交文件则不能。执行创建操作的 RPC 调用必须指定该文件是否被立即提交。

无论哪种文件，在服务器上必须拥有该文件的拷贝，同时返回该文件的权能。如果文件没被提交，它可以被后续的 RPC 操作修改；特别是，它可以被追加。当所有的追加和修改完成后，文件就被提交了，这时文件就成为不可变的了。为了强调未提交文件的暂时性，未提交文件不可读，只有提交后的文件可读。

子弹服务器接口(the bullet server interface)

子弹服务器支持表 7-5 所示的六个操作和为系统管理员保留的另外三个操作，另外还支持表 7-1 所示的一些相关的标准操作。所有这些操作是通过从库中调用存根过程来访问的。

表 7-5 Bullet 服务调用

调用	说明
Create	创建新文件；也可以提交它
Read	读特定文件的全部或部分
Size	返回特定文件的大小
Modify	覆盖未提交文件的 n 个字节
Insert	在尚未提交文件中插入或追加 n 个字节
Delete	从尚未提交文件中删除 n 个字节

Create 调用创建新文件，提供一些该新文件的数据，该新文件的权能在应答中返回。

如果文件已经提交（由一个参数确定），则该文件就可读而不可修改了。如果文件未提交，在提交前，不能读该文件，但可以修改或追加该文件。

Read 调用可读取任何提交文件的部分或全部。它通过提供该文件的权能来指定要读的文件。权能的出现是操作被允许的证明。子弹服务器不能对客户的身分作任何检查：它甚至不知道客户的身分。*Size* 调用以权能作为参数，返回相应文件的大小。

最后三个调用都是针对未提交文件的。通过这些调用可以重写、插入或删除文件的字节。可以依次执行多个调用。最后一个调用通过一个参数指示它想提交文件。

子弹服务器也支持系统管理员的三个特殊调用，系统管理员必须提供一个特殊的超级权能。这些调用将主存缓冲区中的内容移到磁盘上并冲洗主存缓冲区，允许磁盘压缩和修复损坏的文件系统。

子弹服务器产生和使用的权能通过 *Rights* 域实现操作的保护。这样，例如，可以产生一个允许读而不能破坏文件的权能。

子弹服务器的实现

子弹服务器维护一个每个文件一个表目的文件表，与 UNIX 的 *i* 节点表类似，如图 7-16 所示。当子弹服务器启动时，整个表被读入内存，并且只要子弹服务器在运行，表就一直保存在那里。

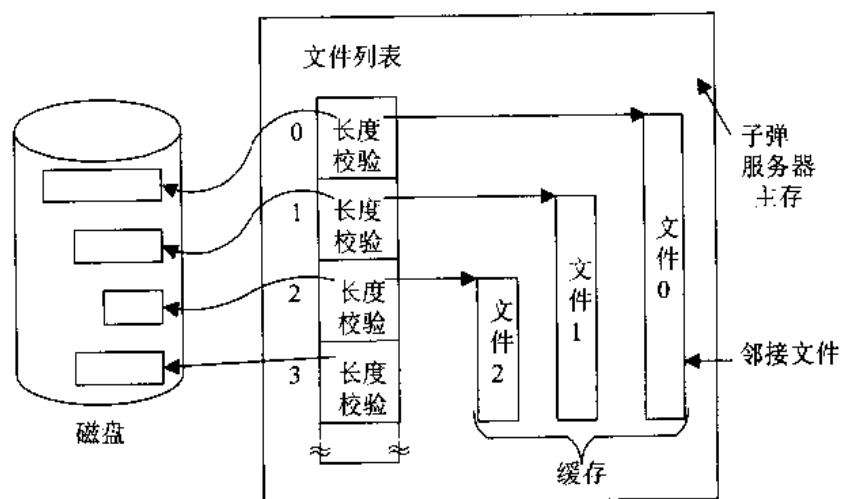


图 7-16 子弹服务器的实现

粗略地说，每个表目包含两个指针和一个长度，再加上其他附加的信息。一个指针给出文件的磁盘地址，另一个给出主存地址，如果那时文件恰好在主存缓冲区中。所有文件在磁盘上和缓存中都是邻接存放的，所以一个指针和一个长度就足够了。与 UNIX 不同的是，不需要直接或间接的块。

尽管由于内存和磁盘的外部碎片的存在，这种策略会浪费空间，但它有非常简单和高效的优点。磁盘上的一个文件可以以磁盘的最大速度，在一次操作中读入内存，也可以以网络的最大速度在网络上传输。随着内存和磁盘变得更大更便宜，与所提供的速度相比，内存浪费的代价有可能是可以接受的。

当客户进程想读一个文件时，它就将该文件的权能送到子弹服务器。服务器从权能

中得到对象号码 (object number), 把它作为文件表的索引来定位该文件的表目。表目包括权能的 check 域中所用的随机数, 然后就用它来检验权能是否有效。如果无效, 操作将以一个错误代码结束。如果有效, 整个文件就从磁盘送到缓存, 除非它已经在缓存中了。缓存空间用 LRU 管理, 但暗含的假设是缓存通常足够大, 能够保存所有正在使用的文件。

如果文件已创建且权能丢失, 则这个文件就再也不能访问了, 但它将永远存在。为了避免这种情况, 采用了超时 (timeouts) 技术。一个 10 分钟内未被访问的未提交文件将被删除, 它的表目也被释放。如果该表日后被另一个文件使用, 但是 15 分钟后出现了原来的权能, check 域将检测出文件已经改变, 将拒绝对旧文件的操作。由于通常文件的未提交状态只存在几秒钟, 所以这种方案是可接受的。

对于提交的文件, 采用一种不太严格的方法。与每个文件 (在文件表目中) 相联系的是一个计数器, 该计数器被初始化为 `MAX_LIFETIME`。后台程序定期利用子弹服务器执行 RPC, 要求它执行标准 `age` 操作 (见表 7-1)。这一操作使子弹服务器遍历文件表, 将每个计数器减 1。任何计数器为 0 的文件将被删除, 它在磁盘, 表和缓存所占的空间被释放。

为了防止这种机制删除正在使用的文件, 提供了另一种操作 `touch`。`age` 用于所有文件的操作, 而 `touch` 操作只用于一个具体文件, 它的功能是将计数器复位为 `MAX_LIFETIME`。对任何目录下的所有文件定期执行 `touch` 操作以防它们超时。通常, 每个文件每小时执行一次 `touch` 操作, 如果一个文件已有 24 小时未执行 `touch` 操作, 就删除该文件。这种机制可以删除丢失的文件 (即不在任何目录的文件)。

子弹服务器可以作为普通进程在用户空间运行。但是, 如果它运行在没有其他进程的专用机上, 可通过将它放在内核中稍微改善性能。这一变化不影响语法, 客户甚至不知道它在哪里。

7.6.2 目录服务器

子弹服务器, 如我们所看到的一样, 仅处理文件存储。文件和其他对象的命名由目录服务器处理。目录服务器的主要功能是提供从人们可读的 (ASCII) 名字到权能的映射。进程可以创建一个或多个目录, 其中每个目录包含多行。每一行描述一个对象, 并包含该对象的对象名和权能。系统提供创建和删除目录, 增加或删除一行, 以及在目录中查找名字等操作。和子弹文件不同, 目录是可以修改的。可以给已经存在的目录增加一个项或从已经存在的目录中删除一个项。

目录本身也是对象, 和其他对象一样, 它通过权能得到保护。在目录上的操作, 诸如名字查找和增加入口, 是以通常的方式通过权限域的位得到保护的。目录权能可存放在其他目录中, 允许层次目录树和更为一般的结构。

尽管目录服务器可以被简单用作存放(文件名,权能)对, 它 also 支持更通用的模型。第一, 目录项可以是任何可由权能描述的对象, 而不仅仅是文件或目录。目录服务器不知道也不关心它的权能所控制的对象种类。一个目录中的项可能是各种不同类型的对象, 这些对象可能分散在世界的任一地方。在一个目录中所有对象的类型不必完全相同, 也不必由同一服务器管理。当获得一个权能时, 通过广播定位该权能对应的服务器的位置, 如前面所述。

第二, 每行可以包含不止一个权能, 而是权能一个完整集合, 如图 7-17 所示。通常,

这些权能是对象的同一拷贝，它们由不同的服务器管理。当进程查找一个名字时，就给这个进程整个权能集合。为了了解这个性质应用，看看打开文件的 *open* 库例程。该进程查找一个文件并得到返回的一个权能集，然后进程就可以依次尝试每个权能，直到找到一个可用的服务器。通过这种方法，如果一个对象不可用，可以用另一个对象来替代它。应该明确的是，当文件为不可改变文件时，这种机制效果最好，所以不存在当文件创建后可能改变的危险。

ASCII 字符串	权能集	属主	组	其他
Mail	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>	1111	0000	0000
Games	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>	1111	1110	1110
Exams	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>	1111	0000	0000
Papers	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>	1111	1100	1000
Committees	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>	1111	1010	0010

图 7-17 由目录服务器组织的典型目录

第三，每行可以包含多列，每行形成不同的保护域并有不同的权限。例如，模仿 UNIX 保护模式，一个目录可能有一列针对所有者，一列针对所有者的组，一列针对其他用户。目录的一个权能实际是目录中特定列的权能，使得所有者、所有者组和其他用户具有不同的权限。由于对一行的所有列来说，潜在权能集是相同的，所以只需要为每一列存储权限位即可。实际的权能可以按要求来计算。

图 7-17 给出了一个有五项的示例目录的格式。这个目录用五行来存储存在其中的五个文件名。这个目录有三列，每一列代表一个不同的保护域，在本例中即为所有者、所有者组和其他。如果目录所有者发送最后一列的权能，接收者就无法访问前面两列的更大权能的权能。

和我们在前面所提到的一样，目录中可以包含其他目录的权能，这种能力使得我们不仅可以以它们完全的一般性来建造目录树，而且可以建造目录图。这种能力的一个明显应用是，一个文件的权能可以放在一个或多个目录中，从而生成对它的多个连接。这些权能也可以有不同的权限，这样使得多个用户可以以不同的访问权限共享一个文件，这一点在 UNIX 中是不可能的。

在任何分布式系统中，特别是在为在 WAN 上应用所设计的分布式操作系统中，使用任何单个的全局的根目录的概念都很困难。Amoeba 中，每个用户有它自己的根目录，如图 7-18 所示。它包含的权能不仅可以是用户的私有子目录的权能，也可以是包含系统程序或其他共享文件的各种公共目录的权能。

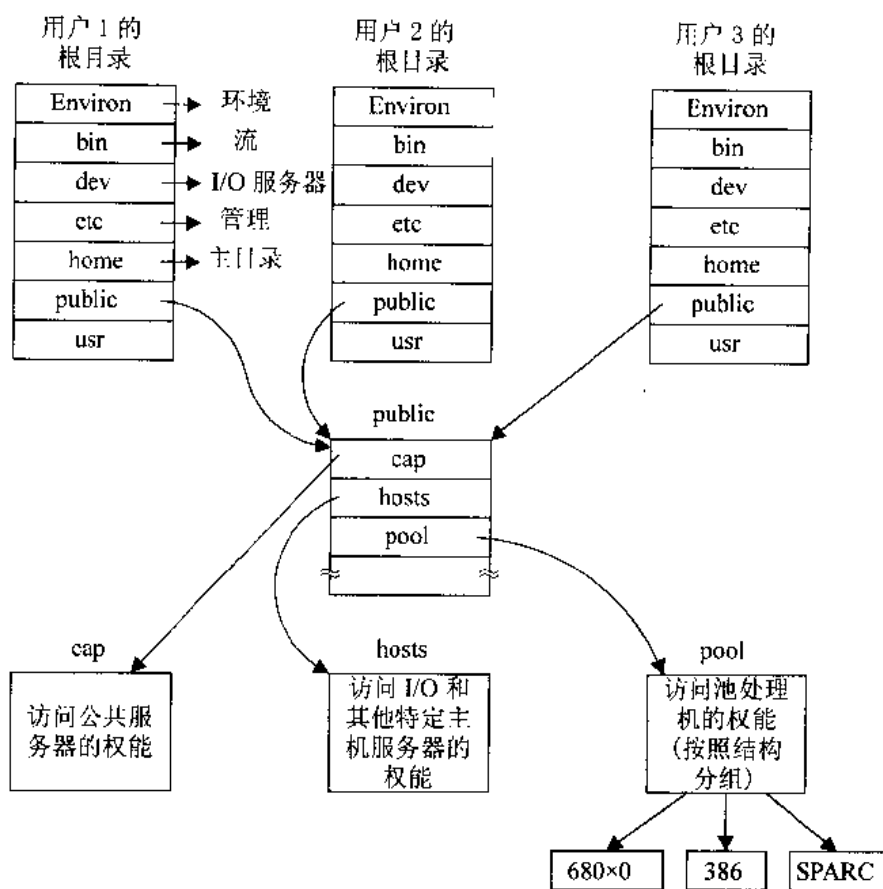


图 7-18 Amoeba 目录层次的简单表示

用户根目录下的某些目录和 UNIX 系统中的那些目录类似，比如 *bin*、*dev* 和 *etc*。但是，其他的目录根本不同。这些目录中有一个是 *home* 目录，它是用户的本地目录。

另一个是 *public* 目录，它包含共享公共目录树的起点。在其他目录中，可以找到 *cap*、*hosts* 和 *pool*。例如，当一个进程要联系子弹服务器、目录服务器或其他服务器以便创建一个对象或进行其他操作时，它必须拥有为和那个服务器进行谈话的通用权能，这些权能存放在 */public/cap* 目录下。

在 *public* 的另一个目录是 *hosts*，它包含了一个系统中所有主机的目录。这个目录包含可在主机上发现的各种服务器的权能，如磁盘服务器、终端服务器、进程服务器和随机数服务器等。最后，*pool* 目录包含由 CPU 结构归类的池处理机的权能。有一种机制将每个用户限制到一个特定的池处理机集合中。

目录服务器接口

目录服务器的主要调用如表 7-6 所示。最前面的两个 *create* 和 *delete* 分别用来创建和删除目录。目录创建后，和创建其他对象一样，系统返回其权能。该权能随后可以插入到其他目录中，从而形成一个层次结构。低级接口提供了对命名图 (naming graph) 的外形的最大控制，由于许多程序都乐意使用传统的目录树，因此，库程序包更易于实现这一点。

表 7-6 基本目录服务调用

调用	说明
Create	创建新的目录
Delete	删除目录或目录中的项
Append	增加新的目录项到特定目录中
Replace	替换单个的目录项
Lookup	返回一特定名字的权能集
Getmasks	为特定项返回权限掩码
Chmod	改变现存目录项的权限的位

删除一个目录项和释放对象本身并不完全相同，这一点值得注意。如果一个权能从目录中移走，该对象本身仍继续存在。举例来说，其权能可以被放到另一个目录中。为了清除该对象，必须显示释放该对象或收集到垃圾桶里。

利用 *append* 可以为文件、目录或其他种类的对象在目录中加入一个新项。和多数目录服务器调用一样，它指定了所操作的目录的权能以及放入该目录中的权能和所有列的权限位。利用 *replace* 可以替换一个存在的项，比如当文件被编辑后，要使用新文件而非旧文件时就要执行 *replace* 调用。

最常用的目录操作是 *lookup*。它的参数包括一个目录（列）权能和一个 ASCII 串。它返回对应的权能集。打开一个文件来读首先需要查找它的权能。

所列的最后两个操作 *getmasks* 和 *chmod* 实现对由它的串所指定行的所有列的权限位掩码的读和写。

还有几个其他的目录操作。它们主要是涉及对多个文件的同时查找或替换。它们在实际涉及多个文件的原子事务中很有用。

目录服务器的实现

目录服务器是 Amoeba 系统的一个关键组成部分，因此它是以容错方式实现的。基本的数据结构是存储在原始的磁盘分区上的权能对的数组。这个数组不用子弹服务器，因为它需要时常更新，如果用子弹服务器，开销很大。

当目录创建时，放入它的权能里的对象数是进入该数组的索引。当给出一个目录权能，服务器检查包含在其中的对象编号，并利用它从数组中得到相应的权能对。这些权能是同一文件的，存储在不同的子弹服务器中，每一权能都包含目录和用来核实目录权能的真实性的 *Check* 域。

当目录改变时，将为它产生一个新的子弹文件，原始磁盘分区上的数组将重写。稍后，后台线程将产生第二个拷贝。然后原来的目录就被删除了。尽管这种机制有一些额外的开销，但传统的文件系统相比，它所提供的可靠性高得多。另外，目录服务器通常成对出现，每一个带有自己的权能对数组（在不同的磁盘上），以防止一个原始磁盘分区被破坏所造成的灾难性后果。两个服务器进行通信以保持同步。也可以只运行一个。双服务器模式如图 7-19 所示。

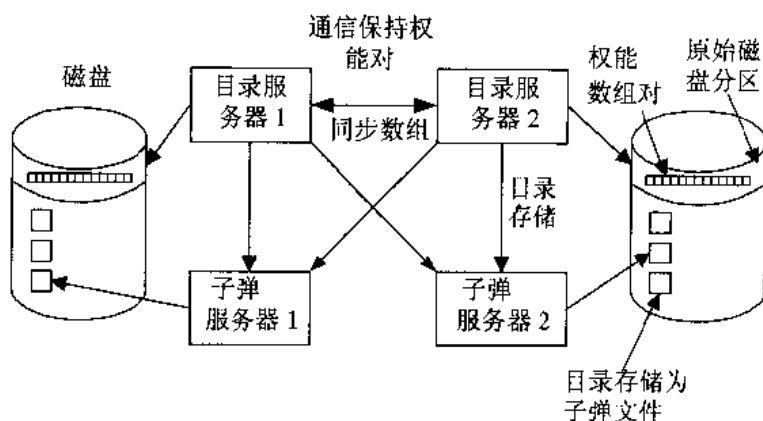


图 7-19 一对目录服务器所有的数据都在不同的子弹服务器上存储

在图 7-17 中，权能集每行只存储一次，尽管其中有很多列。在实际中使用这种结构。在大多数情况下，*owner* 列包含全为 1 的权限位，因此在该集中的权能是真正的所有者权能（即 *Check* 域还没有经过单向函数的运算）。当查找另外一列的一个名字时，通过异或目录项的 *rights* 域和所有者权能的 *Check* 域，目录服务器自己可以计算出限定的权能。然后，这个结果通过单向函数的运算后返回给调用者。

使用这种方法消除了存储大量的权能的必要。而且，目录服务器缓存大量已使用过权能来避免使用不必要的单向函数。如果权能集不包括所有者权能，就要激活服务器来计算限定的权能，因为目录服务器那时无法访问原始的 *Check* 域。

7.6.3 复制服务器

目录服务器管理的对象可以使用复制服务器自动复制。它执行的复制称为懒惰复制。这意味着当一个文件或其他对象刚创建后，最初只形成一个拷贝。然后可以激活复制服务器产生一个相同的副本，如果复制服务器有时间的话。复制服务器一直在后台运行，定期扫描目录系统的特定部分，而不是直接调用它。一旦它发现一个目录项中实际的权能比所假定的 n 个少时，它就联系相关的服务器并安排形成其他的拷贝。虽然复制服务器可以用来复制任何种类的对象，但它对不可变对象，如子弹文件，效果最好。不可变对象的优点在于它在复制过程中不会改变，保证了操作在后台运行的安全性，即使该过程需要大量的时间。由于可变对象在复制过程中可能修改，故增加了为保持一致性导致的复杂度。

此外，目录服务器也采用子弹服务器及其他服务器所采用的过期和垃圾收集机制。它定期接触目录服务器控制的每个对象，防止它们超时。它也可以向服务器发送 *age* 消息，让服务器减少所有的对象计数器，垃圾箱则收集所有计数器值为 0 的对象。

7.6.4 运行服务器

当用户在终端键入一个命令（如 *sort*）后，系统必须作出如下两个决定：

- (1) 进程将运行在何种结构的 CPU 上？
- (2) 应该选择哪个 CPU？

第一个问题与该进程是否能运行在 386、SPARC 或 680x0 等 CPU 上有关。第二个问

题与 CPU 的选择有关，并依赖于当前候选 CPU 的负载和候选处理机的内存的可用性。运行服务器帮助作出这些决定。

每个运行服务器管理一个或多个处理机池。一个称为 `pooldir` 的目录表示一个处理机池，`pooldir` 包括所支持的每种 CPU 结构的子目录。子目录包含访问池中每台计算机上的进程服务器的权能。图 7-20 给出了一种范例排列方式。其他排列方式也是可以的，包括混合池和重叠池，以及将池分为子池。

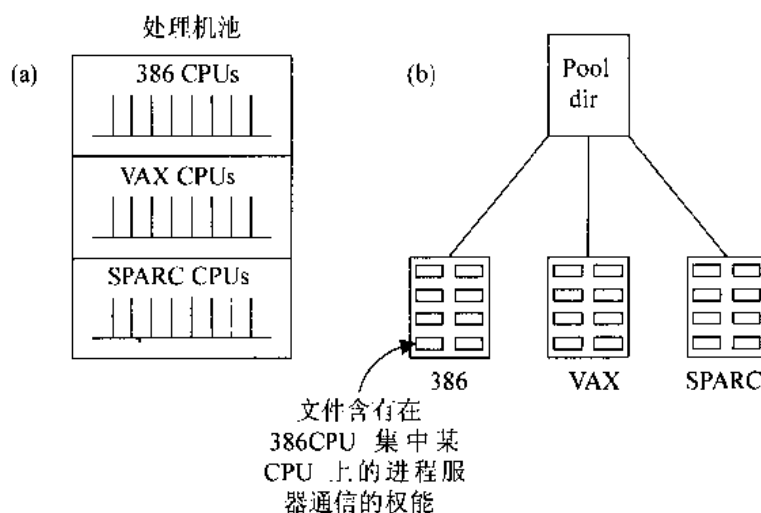


图 7-20 (a) 处理机池
(b) 通信池目录

当外壳要运行一个程序时，它查找 `/bin` 并找到，假设为 `sort`。如果 `sort` 对多种结构可用，它就不是单独一个文件而是目录，该目录包含的各可用种结构上的可执行文件。然后外壳利用运行服务器执行一次 RPC，它向运行服务器发送所有有效的进程描述符，并要求它选择一个结构和一个特定的 CPU。

运行服务器在它的 `pooldir` 中查找，以了解它提供的东西。选择过程大致如下所述。首先，计算进程描述符与池处理机的交集。如果有 386、SPARC 和 680x0 的进程描述符（即，二进制程序），而这个运行服务器管理 386、SPARC 和 VAX 池处理机，则只有 386 可能，排除其他机器作为候选。

其次，运行服务器检查哪一台候选机器有足够的内存来运行该程序。没有足够内存的也被排除掉。运行服务器跟踪它的每一个池处理机的内存和 CPU 的使用情况，通过执行 `getload` 调用定期取得它的池中的每个处理机的这些值，因此运行服务器表中的数字是经常刷新的。

最后，对每个余下的机器，可以得到对其投入新的程序的计算能力的估计。每个 CPU 作出自己的估计。所用的启发策略是将已知的 CPU 的总的计算能力和在 CPU 上并行的活动的线程的数目作为输入。例如，如果一个 20-MIPS 的机器现在有四个活动线程，加入第五个就意味着包括新加入的在内的每个线程平均会得到 4MIPS。如果另一个处理机有 10MIPS 和一个活动线程，在这台机器上，新程序可以期望得到 5MIPS。运行服务器选择能够给出最大 MIPS 的处理机，并将与进程服务器谈话的权能返还给调用者。然后调用者

用这个权能来创建进程，和 7.3 节讨论过的一样。

7.6.5 引导服务器 (the boot server)

作为 Amoeba 服务器的另一个例子，让我们来看看引导服务器。通过检查各个假设正在运行的服务器是否正在运行并且对非运行的服务器采取校正措施，引导服务器为 Amoeba 系统提供了一定的容错能力。对崩溃恢复感兴趣的服务器可包含在引导服务器的配置文件中。每个项表明引导服务器轮询的周期和轮询的方式。只要服务器响应正确，引导服务器就不采取任何措施。

然而，如果经过一定次数的尝试后，服务器仍不能响应，引导服务器则宣布它死亡，并试图重新引导它。如果引导失败，则分配一个新的池处理机并运行死亡服务器的一个新拷贝。通过这种方式，重要的服务即使失败也可以自动重启。引导服务器可以自我复制，以防自身的失败。

7.6.6 TCP/IP 服务器

尽管 Amoeba 在内部使用 FLIP 协议来获得高性能，但有时也需要采用 TCP/IP 协议。例如，和 X 终端通信，给非 Amoeba 计算机发送邮件或接收来自非 Amoeba 计算机的邮件，通过 Internet 和其他 Amoeba 系统交互。TCP/IP 服务器可提供服务让 Amoeba 去做这些事。

为了建立一个连接，Amoeba 进程利用 TCP/IP 服务器执行一个 RPC 调用给该服务器发送一个 TCP/IP 地址。这时，调用者一直被阻塞直到连接建立或被拒绝。在应答中，TCP/IP 服务器提供了使用该连接的权能。后续的 RPC 可以从远程计算机上发送或接收包，Amoeba 进程不必知道使用的是 TCP/IP 协议。该机制不如 FLIP 效率高，所以只有在不能用 FLIP 的情况下才使用它。

7.6.7 其他服务器

Amoeba 支持其他各种服务器。这些服务器包括磁盘服务器（被目录服务器用来存放它的权能对数组）、各种 I/O 服务器、时钟服务器以及随机数服务器（对产生端口，权能，和 FLIP 地址有用）。所谓的 Swiss Army Knife 服务器处理那些必须通过在未来某时启动进程完成的各种活动。邮件服务器处理电子邮件的收发。

7.7 小结

Amoeba 是一个新的操作系统，它将一些独立的计算机集合起来，而对用户来说，它看起来是一个单独的分时系统。通常，用户不知道他们的进程在何处（甚至在何种 CPU 上）运行，文件的存放地以及为可用性和性能而维护的拷贝数。然而，对并行程序很感兴趣的用户可以利用存在的多个 CPU 把单个任务分配到多个 CPU 上执行。

Amoeba 是基于微内核的，该微内核处理低级进程和内存管理，通信和 I/O 管理。文件系统和操作系统的其他部分作为用户进程运行。这种分工保证内核小且简单。

Amoeba 有一个单独的机制实现对所有对象的命名和保护——权能。权能中包含了说明利用该权能能进行的操作的权限。利用单向函数加密保护权能。每个权能包含一个确保

该权能的安全性的 Checksum 域。

Amoeba 支持三种通信机制: RPC 和为点到点通信设计的原始 FLIP,以及为多播通信设计的可靠的组通信。RPC 保证最多一次语义。由定序器算法所提供的可靠广播是组通信的基础。FLIP 协议的上层支持这两种机制,且这两种机制紧密结合。原始 FLIP 只有在特殊环境下才使用。

Amoeba 文件系统由三个服务器组成:处理文件存储的子弹服务器,处理文件命名的目录服务器,处理文件复制的复制服务器。子弹服务器维护在磁盘或缓存中是邻接存放的不可变文件。目录服务器具有容错功能,它将 ASCII 串映射到权能。复制服务器处理懒惰复制。

习 题

- (1) Amoeba 系统的设计者假设大容量、价格低的内存很快就会成为现实,这个假设对于设计有什么影响?
- (2) 说出处理机池模型与个人多处理机模型相比较的一个优点和一个缺点。
- (3) 列举 Amoeba 微内核的三个功能。
- (4) 一些 Amoeba 服务器可以运行在内核中,也可以运行在用户空间中。它们的客户端并不能区分(除非对它们进行时测)。Amoeba 中的哪些措施使客户端不能区分服务器运行在内核还是用户空间?
- (5) 一个恶意的用户试图通过选择一个 48 位的随机数,猜测子弹服务器的 get-port,并运行著名的单向函数,以获知是否得到 put-port。每毫秒试一次。问一般来说要用多长时间才能试出 get-port?
- (6) 一个服务器怎样区分一个权能(capability)是权能拥有者(owner capability)而不是受限权能(restricted capability)?怎样判定 Owner capability?
- (7) 如果一个 capability 不是 owner capability,服务器怎样验证它的有效性?
- (8) 解释什么是全局变量。
- (9) 为什么 trans 调用对于发送和接收都有参数?用两个不同的调用, send_request 用来发送和 get_reply 用来接收,会不会更好和更简单呢?
- (10) Amoeba 宣称保证 RPC 至多一次的语义解释,假设三个文件服务器提供相同的服务,一个客户端使用 RPC 调用其中的一个服务器,这个服务器执行请求然后崩溃,而后这个 RPC 又调用另一个服务器,导致它的请求被执行了两次,这是可能的吗?如果可能,那么 Amoeba 系统“保证”的是什么?如果不可能,它是怎样被防止的?
- (11) 为什么定序器需要一个历史(history)缓冲区?
- (12) 文中给出了两个在 Amoeba 系统中实现广播发送的算法。方法 1 中发送者发送一个点对点消息给定序器,定序器然后广播发送。方法 2 中,发送者广播发送,定序器然后广播发送确认包。假设网络速度 10-Mbps 并且处理一个到达信包(任意大小)的中断需要 500 微妙。如果所有的数据包是 1K 字节,确认信包是 100 字节,对于 1000 个广播发送,两种方法各需多少带宽和 CPU 时间?
- (13) FLIP 寻址的什么特性使处理进程迁移和以简单的方法进行网络自动重新设置成为可

能？

- (14) 子弹服务器对它的用户支持不可修改的文件，子弹服务器自己的表也是不变的吗？
- (15) 为什么子弹服务器有提交的和未提交的文件？
- (16) 在 Amoeba，与一个文件的连接可通过将有不同权限的权能（capability）放入不同的目录中。这可以给不同用户不同的权限。在 UNIX 中不存在这种特性，为什么？

第 8 章 实例研究 2: Mach

现代基于微内核操作系统的第二个例子是 Mach。我们将以它的历史以及如何从早期系统中发展出来作为开始。然后我们将以进程与线程、内存管理和通信为重点详细介绍微内核。最后我们将讨论一下该系统的 UNIX 仿真。关于 Mach 的更详细的信息可以从 (Accetta 等, 1986; Baron 等, 1985; Black 等, 1992; Boykin 等, 1993; Draves 等, 1991; Rashid, 1986a; Rashid, 1986b; 以及 Sansom 等, 1986) 得到。

8.1 有关 Mach 的介绍

在本节中我们将对 Mach 进行简要介绍。首先我们要介绍一下它的历史与设计目标, 然后再描述 Mach 微内核的概念以及在微内核上运行的主 (princial) 服务器。

8.1.1 Mach 的发展历史

Mach 的根最早可以溯源到一个名为 RIG (Rochester Intelligent Gateway) 的系统。该系统是 Rochester 大学从 1975 年开始研制的 (Ball 等, 1976)。RIG 开始是为一种 16 位数据通用小型机 (Data General Minicomputer) Eclipse 编写的。它的主要研究目的是为了验证操作系统可以以一种模块化的形式进行设计以及包括网络中的不同进程在内的一组进程之间的通信可以通过消息传递来实现。这个系统的设计与开发成功从实际上证明了这种形式的操作系统是可以实现的。

当这个系统的设计人之一, Richard Rashid, 在 1979 年离开 Rochester 大学来到 Carnegie-Mellon 大学之后, 他希望能够在更先进的硬件设备上继续实现基于消息传递的操作系统。他在许多不同的设备之间进行选择, 最后确定为 PERQ。PERQ 它是一种具有位图式的显示器、鼠标以及网络连接在内的早期工程工作站。当然在它上面是可微程序再编程的。这种为 PERQ 编写的新操作系统称为 Accent。它在 RIG 的基础上增加了保护、网络透明操作、32 位虚拟内存以及其他的一些特性。该系统的原始版本在 1981 年编制完毕并投入运行。

在 1984 年, Accent 系统运行在具有 150 台 PERQ 机器的环境下性能明显不如 UNIX 系统。这种情况促使 Richard 开始实施称为 Mach 的第三代操作系统项目。他希望通过使 Mach 同 UNIX 兼容来利用 UNIX 系统中的大量现有软件。除此之外, 他还在 Accent 的基础上改进了许多特性, 主要包括线程、更好的进程通信机制、对多处理机的支持以及一个高度抽象的虚拟内存系统。

与此同时, DARPA, 美国国防部的高级研究计划局正在四处寻找支持多处理机的操作系统来作为它初始战略计算的一部分。他们选择了 CMU (Carnegie-Mellon University: 卡内基-梅隆大学)。于是在 DAPRA 的充足资金的支持下, Mach 系统得到了更进一步的发展。Mach 开始由 BSD4.1 的一个版本通过增加一些有关通信以及内存管理部分来实现。

随着 BSD 4.2 以及 BSD4.3 的出现, Mach 也通过推出同它们绑定的相应版本来进行升级。虽然这种版本导致了一个较大的系统内核,但是它确实保证了同 Berkeley UNIX 系统的绝对兼容,而这正是 DARPA 的一个重要目标。

Mach 的最早的版本是 1986 年推出的运行在具有四个 CPU 的 VAX 11/784 系统上的版本。在这之后不久就完成了向 IBM PC/RT 以及 Sun 3 的移植工作。在 1987 年, Mach 也被移植到 Encore 以及 Sequent 多处理机环境。虽然 Mach 具有网络应用程序,但是当时它们主要被设计为在单机或者多处理机系统中运行的系统,而不是在由一组机器组成的 LAN 上运行的透明分布式系统。

在这之后不久,在计算机供应商 IBM、DEC 和 Hewlett Packard 的联合领导下成立了开放软件基金会 (Open Software Foundation) 联盟,它的主要目的是改变 UNIX 对他们用户的控制,而 UNIX 的属有者 AT&T 公司当时正同 Sun Microsystem 紧密合作开发 System V Release 4。OSF 的成员们害怕这两家的联盟会使 Sun 比他们更具有竞争优势。在经过一些错误的选择之后, OSF 选择了 Mach2.5 来作为它的第一个操作系统 OSF/1 的基础。虽然 Mach2.5 和 OSF/1 都包含了大量的 Berkeley 和 AT&T 代码,实际上是希望 OSF 至少可以控制 UNIX 的发展方向。

在 1988 年, Mach2.5 内核比较庞大而且非常单一,这主要是由系统内核中存在大量 Berkeley UNIX 代码的缘故。在 1988 年, CMU 将所有的 Berkeley 代码都从内核移出,并将它们放到用户空间中去。这样就使系统具有微内核以及纯粹由 Mach 组成的特点。本章中我们将集中介绍 Mach3 微内核以及一个为 BSD UNIX 使用的用户级的操作系统仿真器。然而一个比较重要的问题是 Mach 仍然处于正在开发的阶段,所以对于它的任何描述最多只能是一个快照。而幸运的是本章中所讨论的大多数基本概念是相对比较稳定的,但是它们中的一些细节也会随着时间的推移而改变。

8.1.2 Mach 的设计目标

Mach 是从它的最早的实现系统——RIG 中发展起来的。这个项目的目标也随着时间的推移而改变。目前的主要目的可以总结如下:

- (1) 为建造其他操作系统而提供一个基础 (如 UNIX);
- (2) 支持大型稀疏地址空间;
- (3) 允许对网络资源的透明访问;
- (4) 从系统与应用两个方面开发并行性;
- (5) 使 Mach 可以被移植到有更多数量机器的系统中。

这些目标中既包含了研究也包含了开发。主要思想是在仿真现有系统,如 UNIX、MS-DOS 和 Macintosh 操作系统的基础上探究多处理机和分布式系统。

Mach 的早期工作集中在单一和多处理机系统中。在 Mach 的设计阶段,很少有系统能够支持多处理机。即使现在,除了 Mach 之外也很少有多处理机系统是和设备无关的。

8.1.3 Mach 微内核

Mach 的微内核的建立目标是作为一个可以仿真 UNIX 以及其他操作系统的基础存在的。这种仿真可以通过运行在内核之外的软件层来实现的,如图 8-1 所示。

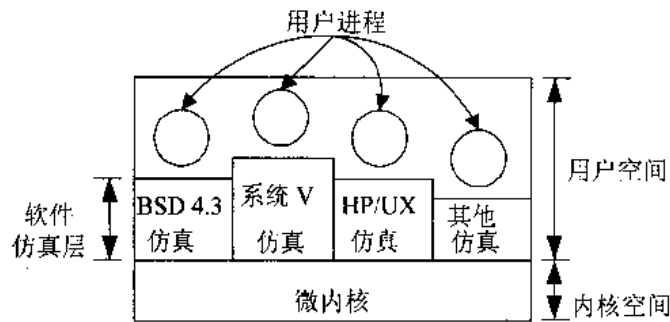


图 8-1 使用 Mach 的 UNIX 仿真的抽象模型

每一个仿真器都由一个呈现它的应用程序的地址空间部分以及一个或者多个独立于应用程序的服务器组成。我们需要强调的是多个仿真器可以同时运行，所以在同一台机器上同时运行 BSD 4.3, System V 和 MS-DOS 程序是可能的。

同其他微内核非常相似，Mach 内核提供了进程管理、内存管理、通信以及 I/O 服务。文件、目录以及其他传统的操作系统功能是在用户空间中处理的。Mach 内核的核心思想是提供可以让系统工作的必要机制，而由用户级过程来决定具体的策略。

内核管理五个主抽象对象：

- (1) 进程；
- (2) 线程；
- (3) 内存对象；
- (4) 端口；
- (5) 消息。

除此之外，内核还管理着其他几个与这几个有关，或者不那么重要的抽象对象。

进程是操作可以得到执行的基本环境。它拥有一段包含程序和数据地址空间，通常也包含一个或者多个堆栈。进程是资源配置的最基本单位。例如，一个通信通道总是被一个单一进程所“占有”。

另外，在本章中我们也遵循传统的术语，尽管这可能意味着对于 Mach 文档中所使用的术语的背离（例如，在 Mach 中的进程（process）称为任务（task））。

Mach 中的线程是一个可执行的实体。它自身具有程序计数器和一组与它相关联的寄存器。每一个线程都是且仅是一个进程的一部分。一个具有一个线程的进程非常与传统的（如 UNIX）进程相似。

一个 Mach 所特有的概念是内存对象（memory object），它是一种可以映射到一个进程的地址空间的数据结构。内存对象占据一个或者多个页面，形成 Mach 虚拟内存系统的基础。当一个进程要求访问一个当前不在物理主存中的内存对象时会发生页面故障。而在所有操作系统中内核都可以捕获页面故障。然而与其他系统不同的是 Mach 内核会通过给一个用户级的服务器发送消息来获取缺页。

Mach 中的进程间通信是基于消息传递的。为了接收一个消息，一个用户进程要求内核为它创建一种受保护的邮箱，这被称为端口（port）。端口是在内核中保存的，它能够以队列的形式保存任何经过排序的消息队列。队列的长度并不是固定的，但为了合理进行

流量控制 (flow control) 缘故, 如果队列中具有超过 n 个消息, 那么一个企图向这个端口发送消息的进程将挂起, 以使这个端口有清空这个队列的机会。而参数 n 也是可以根据端口进行设定的。

一个进程可以拥有将它的一个端口发送给另外一个端口 (或者从另外端口接收) 的权能。这种允许是以权能的形式出现的, 它不仅仅包括一个指向端口的指针, 还包含一个说明其他进程对该端口所具有的权限 (如发送权限) 的列表。一旦这种许可建立完毕, 其他进程就可以给这个端口发送消息, 然后第一个进程就可以读取这些消息。Mach 中的所有通信都使用这种机制。Mach 并不支持一个全权能机制, 端口是唯一的权能存在的地方。

8.1.4 Mach 的 BSD UNIX 服务器

正如我们前面所描述的, Mach 的设计者已经将 Berkeley UNIX 移到用户空间中作为一个应用程序存在。这种结构相较于单内核结构有许多明显的优势。首先, 通过将系统分解成一个处理资源管理的部分 (内核) 和一个处理系统调用的部分 (UNIX 服务器) 可以使这两个部分都变得更简单和容易维护。从某些方面来说, 这种分解很类似于 IBM 的主机操作系统 VM/370 的任务划分。在该方式下, 内核仿真一组 370 裸机, 其中每一个都运行一个单用户操作系统。

其次, 通过将 UNIX 移到用户空间可以使系统具有很强的机器无关性, 使得系统向更多的不同种类机器的可移植性大大增强。所有的机器相关性都可以从 UNIX 中删除, 只在 Mach 的内核中隐藏实现。

第三, 前面我们提到过多种操作系统可以同时运行。例如在 386 机器上, Mach 可以同时运行一个 UNIX 程序和一个 MS-DOS 程序。类似的, 我们也可以在测试一个新的试验操作系统的同时运行其他的若干操作系统产品。

第四, 系统的实时操作特性也可以得到加强。这主要是因为传统 UNIX 的所有对实时工作的障碍, 如封锁中断以更新关键表等, 或全部清除或者移到用户空间中。于是内核就可以设计成为对实时应用程序没有任何阻碍的。

最后, 这种安排还可以用来加强进程之间的安全保证, 如果这是必要的话。如果每一个进程都有它自己的 UNIX 版本号, 那么一个进程偷窃其他进程的文件将会十分困难。

8.2 Mach 中的进程管理

Mach 中的进程管理处理进程、线程以及它们的调度。在本节中我们将逐一进行介绍。

8.2.1 进程

Mach 中的进程主要由一个地址空间和运行在这个地址空间之内的若干个线程组成。进程是被动的, 它的执行依赖于线程。进程主要作为一种方便的容器来收集一组协同工作的线程所需要的所有资源。

图 8-2 就是一个 Mach 的进程。除了一个地址空间和线程之外它还有一些端口和其他特性。在图 8-2 中所显示的端口有一些特殊的功能。进程端口 (process port) 主要用来同内核进行通信。许多进程所需要的内核服务都是通过给进程端口发消息来完成的, 而不是通过

系统调用完成。整个 Mach 系统中都使用这种机制来使实际的系统调用减少到几乎没有有的地步。本章中将讨论其中的一些，以使读者对它们有一些概念。

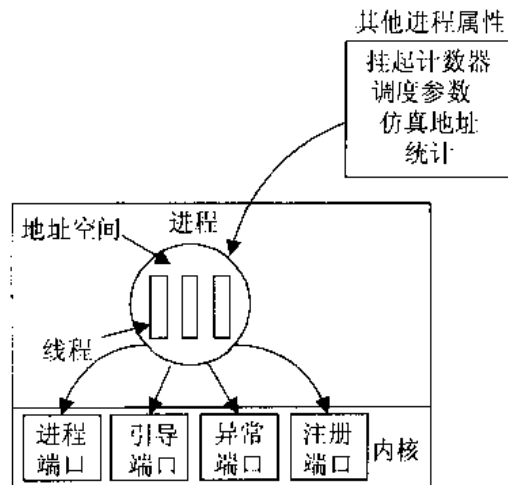


图 8-2 一个 Mach 进程

在一般情况下，编程人员甚至不知道一个服务是否需要系统进行调用。包括系统调用所访问的和消息传递所访问在内的所有服务都在类库中有存根过程。这些过程在用户手册中详细描述以供应用程序调用。而这些过程都由 MIG (Mach Interface Generator: Mach 接口生成器) 编译器的服务定义生成。

引导端口 (bootstrap port) 用来进行进程开始的初始化工作。最开始的进程读入引导端口来获取提供必要服务的内核端口的名称。UNIX 进程也使用它来同 UNIX 仿真器进行通信。

异常端口 (exception port) 用来报告进程所引起的异常。典型的异常有被零除和执行了非法指令。该端口告诉系统应该向哪里发送异常消息。调试器 (debugger) 也使用异常端口。

注册端口 (registered port) 通常用来为进程提供一种同标准系统服务器进行通信的途径。例如，命名服务器使通过提供一个字符串来获取某个基本服务器的相应端口成为可能。

进程也可能有其他属性。一个进程可以处于就绪或者阻塞状态，而与它的线程状态无关。如果一个进程是可以运行的，那么其中线程就是可运行的，可以被调度和运行。如果该进程处于阻塞状态，那么它的线程就不能运行，而不管它们处于什么状态。

进程相关的属性也包括调度参数。这些属性包括指定线程运行的处理机的权能。这在多处理机系统中特别有用。例如，进程可以利用这种权能去强制所有线程运行在不同的处理机上，或者强制它们运行在相同的处理机上，或者这两者之间的其他情况。除此之外，每一个进程也有可以进行设定的缺省优先级。当一个线程创建时就赋给这种优先级。当然，所有现有线程的优先级也都可以改变。

通过设定仿真地址可以通知内核系统使用进程空间的哪一部分来定位系统调用处理程序 (handler)。系统必须知道这些地址来处理需要被仿真的 UNIX 系统调用。而且这些是在仿真器一开始启动就设定的，所有仿真器的子孙 (就是所有的 UNIX 进程) 都继承这个参数。

最后，每一个进程还拥有同它相联系的诸如内存消耗、线程运行时间等统计参数。需要这种信息的进程可以通过给相应进程端口发送消息来获取。

另外也值得一提的是一个 Mach 进程所缺乏的特性。一个进程不包含 uid、gid、信号掩码、根目录、工作目录或者文件描述符数组 (array)；而这些都是 UNIX 进程所拥有的。所有这些信息都由仿真器包来管理，因而内核对它们一无所知。

进程管理原语

Mach 系统提供了少量用来进行进程管理的原语。它们中的绝大多数都是通过进程端口给内核发送消息来实现的，而不是使用真正的系统调用。表 8-1 中列出了这些原语中最重要的一些。这些调用同所有 Mach 调用一样都由前缀来指明它们所归属的组，但是为了简明起见，我们在这里（以及之后的所有其他地方）省略了这些前缀。

表 8-1 部分 Mach 进程管理调用

调用	说明
Create	创建继承某些属性的新进程
Terminate	杀死一个特定的进程
Suspend	挂起计数器加 1
Resume	挂起计数器减 1。如果结果为 0 则将进程从阻塞状态恢复
Priority	为当前或后来的线程设定优先级
Assign	说明新线程应该运行在哪个处理机上
Info	返回执行时间、内存消耗等信息
Threads	返回进程所包含的线程列表

表 8-1 中的前两个调用分别是进程的建立与清除。进程建立调用指定了进程的一个原型进程，但它不一定与调用进程相同。除了调用拥有一个指明孩子进程是否继承父进程地址空间的参数之外，孩子进程基本上是原型的一个拷贝。如果它不继承父进程的地址空间，以后也可以对这些对象（如文本、初始数据以及一个堆栈）进行映射。孩子进程开始没有线程，但是它会自动获得一个进程端口、一个引导端口以及一个异常端口。由于其他端口可能只有一个读者，所以它们并不是自动继承的。

进程可以在程序的控制下挂起或者继续。每一个进程都有一个计数器，*suspend* 调用会给它增 1 而 *resume* 调用则正好相反，这样就可以阻塞一个进程或者对它进行恢复。当计数器为 0 时进程就可以运行，而当它为正时进程就被阻塞。使用计数器要比仅使用一个位应用广泛，而且还可以避免出现竞争情况。

priority 和 *assign* 调用可以让编程人员决定他的线程在多处理机系统中的运行方式与位置。由于通过使用优先级可以实现 CPU 调度，所以编程人员可以很好地控制哪一个线程是最重要和哪一个线程最不重要。*assign* 调用可以使线程在指定的 CPU 或 CPU 组上运行。

表 8-1 中的最后两个调用主要用于返回进程的信息。前一个调用给出了一些统计信息，而后者则返回该进程所拥有的所有线程列表。

8.2.2 线程

Mach 的活动实体是线程。它们执行指令，管理它们的寄存器和地址空间。每一个线程都属于而且仅属于一个进程。除非进程拥有一个或者多个线程，否则它不能进行任何工

作。

一个进程中的所有线程都共享地址空间和图 8-2 所描述的所有进程资源。然而线程也有自己的私有资源。其中之一就是与进程端口非常相似的线程端口 (thread port)。每一个线程都拥有自己的线程端口, 用它来触发线程的内核服务, 如线程结束退出。由于端口是进程范围的资源, 每一个线程都可以访问它的兄弟端口, 所以在需要的情况下任何线程都可以控制同进程内的所有其他线程。

Mach 线程是由内核进行管理的, 这也就是说它们是我们有时所讲的重量线程而不是轻量线程 (每用户空间的线程)。线程的建立和清除都由内核完成, 而且其中也包含了对内核数据结构的更新。它们实现了在单一地址空间内处理多个活动的基本机制。而用户如何使用这些机制则由用户自己决定。

在一个单 CPU 系统中线程是时分复用的, 一个进程首先运行, 然后是下一个。而在一个多处理机系统中, 几个线程可以同时处于活动状态。由于在这种情况下性能与正确性成为一个主要问题, 所以这种并行互斥、同步和调用要比平常重要得多。由于 Mach 系统的目的就是为了解决在多处理机系统上运行, 所以对这些问题都进行了专门考虑。

与一个进程一样, 一个线程也可以处于可以运行或者阻塞状态。而且机制也非常相似: 每一个线程都拥有一个可以增减的计数器。当该计数器为零时, 该线程是可以运行的。当它为正时, 就必须等其他线程来将他降低到零为止。这种机制允许线程控制其他线程的活动。

还有一些其他的不同原语。基本的内核接口提供了大约二十多个线程原语, 这些原语中许多都是关于调度控制细节的。在这些原语之上可以建立不同的线程包。

Mach 的解决方法是 C 线程 (C threads) 包。这个包的目的是使内核线程原语可以以一种简单和方便的形式提供给用户使用。虽然它并不具有内核接口所提供的所有功能, 但是对于普通编程人员来说是足够的。将它设计为可以移植到许多不同的操作系统和体系结构中去。

C 线程包提供了十六个直接进行线程操作的调用。其中最重要的都已在表 8-2 中列出。第一个调用 *fork*, 在与调用线程相同的地址空间上建立一个新线程。它运行一个参数所指定的过程而不是父线程的代码。在调用之后, 父线程继续同子线程并行运行。线程开始以一个优先级在一个进程的调度参数所决定的处理机上运行, 就如同前面所讨论的那样。

表 8-2 直接线程管理的主 C 线程调用

调用	说明
Fork	建立一个与父线程运行相同代码的新线程
Exit	终止调用线程
Join	挂起线程直到特定的线程退出 (exit) 为止
Detach	声明该线程永远不能被 joint (被等待)
Yield	自动放弃 CPU
Self	返回调用线程的身份

当一个线程的工作结束之后, 它就调用 *exit* 调用。如果父线程需要等待线程结束, 它就可以调用 *join* 来阻塞它自己直到一个特定的子线程终止为止。这三个调用大致类似于 UNIX 的 FORK、EXIT 和 WAITPID 系统调用。

第四个系统调用 *detach*，在 UNIX 系统中不存在。它提供了一种声明一个特定的进程是永远不能被等待的途径。如果这个线程存在，它的堆栈以及其他状态信息会立即被删除。这种清除工作一般只有在父线程成功地执行了 *join* 之后才会进行。在一个服务器上，有可能需要为每一个到达的请求开辟一个新的线程。当请求结束之后，线程同时结束。由于初始线程没有必要去等待这种线程，所以这些服务器线程就应该执行 *detach* 调用。

yield 调用提醒调度器当前该线程没有必要的操作，它必须等待某个事件发生之后才可以继续。一个智能调度器将采纳这个提醒并运行其他线程。在 Mach 系统中，线程的调度通常采用抢先机制，*yield* 只是一种优化。在不采用抢先调度策略的系统中，没有工作的线程释放 CPU 就非常有必要，这样可以使其他线程有机会运行。

最后，*self* 返回调用者的身份，类似于 UNIX 的 *GETPID*。

剩余的调用（没有在表中给出的）实现的功能包括允许对线程进行命名、允许程序控制自身线程的数量和堆栈的大小，以及提供访问内核线程和消息传递机制的接口。

同步是通过互斥体和条件变量来实现的。互斥原语包括 *lock*、*trylock* 和 *unlock*。系统也提供了申请和释放互斥体的原语。互斥体工作方式类似于二进制的信号量，提供了一种相互排斥，但是并没有传送任何信息。

对条件参数进行的操作包括 *signal*、*wait* 和 *broadcast*，它允许线程在一定条件下阻塞直到后来有其他线程实现这个条件后再被唤醒。

Mach 中 C 线程的实现

Mach 中对于 C 线程的实现有许多不同的方法。最开始的方法是在用户空间中的一个单一进程中完成所有工作。这种方法使所有的 C 线程在一个内核线程上以时分复用的方式运行，如图 8-3 (a) 所示。这种方法也可以应用在 UNIX 或者其他任何没有内核支持的系统中。线程以协同程序 (coroutines) 的方式运行，这就意味着调度的方式是非抢先机制。一个线程可以在需要或者可能的情况下占有一个 CPU 任意长时间。对于生产者-消费者类型应用来说，生产者可能填满缓存后阻塞，然后让消费者运行。然而在其他应用中，线程将不得不经常调用 *yield* 来给其他线程一个运行的机会。

开始的实现包也被大多数没有内核支持的用户空间线程包所固有的问题所困扰。如果一个线程阻塞了系统调用，如从终端上执行读取，那么整个进程都会被阻塞。为了避免这种情况，编程人员必须避免阻塞系统调用。在 Berkeley UNIX 系统中可以使用 *SELECT* 调用来表明是否所有线程都挂起，但是整个情况仍然十分混乱。

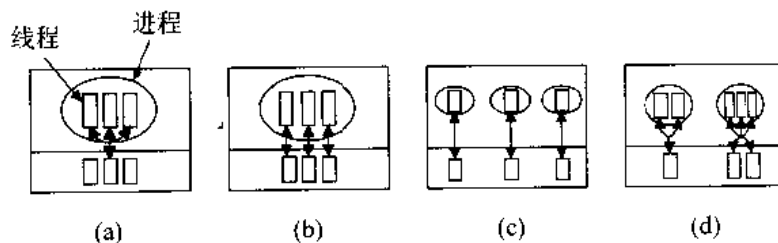


图 8-3 (a) 所有 C 线程使用一个内核线程
(b) 每一个 C 线程有自己的内核线程
(c) 每一个 C 线程都有自己的单线程进程
(d) 用户线程与内核线程的任意映射

第二个实现方法是每一个 C 线程使用一个 Mach 线程,如图 8-3 (b) 所示。这些线程采用抢先机制进行调度。而且在多处理机系统中它们可以在不同的 CPU 上并行运行。实际上也可以在 n 个内核线程上运行 m 个用户线程,虽然最常见的情况是 $m=n$ 。

第三种实现包中每一个进程只有一个线程,如图 8-3 (c) 所示。这些进程被设置成都映射到相同的物理内存空间,像前面的实现方法一样共享内存。只有在使用特殊的虚拟内存的情况下才会使用这种解决方法。这种方法的缺陷在于端口、UNIX 文件以及其他单进程相关的资源都是不能共享的,这极大地减小了这种方法的值。

第四种包允许任意数量的用户进程与任意数量的内核进程之间的映射,如图 8-3 (d) 所示。

第一种解决方法的实际价值主要在于实际中并没有真正的并行,串行运行可以得到可重复结果以及调试比较容易。第二种方法的优势主要在于简单而且已经使用了很长时间。第三种方法并不经常使用。第四种方法虽然最复杂,但是它具有最大的灵活性,同时也是目前最为常用的一种方法。

8.2.3 调度

运行于多处理机系统的目标对 Mach 系统的调度影响很大。由于单处理机系统实际上是一个多处理系统的一个特例(只有一个 CPU),所以我们对调度的讨论将集中在多处理机系统上。如果需要更多的信息请参考(Black, 1990)。

多处理机中的 CPU 可以被软件指定为处理机集(processor set)。每一个 CPU 都属于而且仅属于一个处理机集。线程也可以被软件指定到处理机集。因而每一个处理机集都有一个可供分配的 CPU 集合和一个需要计算权能的线程集合。调度算法的工作就是将线程均衡有效地指定到 CPU 上。为了调度需要,每一个处理机集都是拥有同其他处理机集无关的专有资源和消费者的一个封闭世界。

这种机制给了进程控制自己线程的很多权力。一个进程可以将一个重要的线程指定到一个没有其他线程运行的 CPU 上,因而可以保证这个进程可以一直运行。它也可以随着工作的进展将线程重新分配到 CPU 集上以保证工作负载的均衡。一般的编译器不太可能使用这种应用,而数据库管理系统或者实时系统则可以很好地利用它。

Mach 中的线程调度主要基于优先级。优先级是从 0 开始到某一个数字(经常是 31 或者 127)的整数,其中 0 代表最高的优先级而 31 或 127 代表最低的优先级。这种倒相的优先级来自 UNIX 系统中的优先级。每一个线程有三种优先级。第一种是线程可以在一定范围内设定的基本优先级。第二种优先级是线程可以自行设定的基本优先级的最低数值。由于使用较高的优先机会降低服务质量,所以除非一个进程有意延缓其他进程的运行,它一般将基本优先级设定成为允许的最低值。第三种优先级是当前优先级,主要供调度使用。它由内核通过给基本优先级增加一个基于线程的当前 CPU 使用量的函数决定。

Mach 的线程对于内核是可见的,至少在使用图 8-3 (b) 模式时是这样。每一个线程都同其他线程竞争使用 CPU 资源,而与线程所在的进程无关。当进行调度决策时,内核并不关心线程所从属的进程。

同每一个处理机集相关联的是一个运行队列数组,如图 8-4 所示。这个矩阵中有 32 个队列,对应着当前优先级从 0 到 31 的线程。当一个优先级为 n 的线程变为可运行状态

时，它将被添加到队列 n 的尾部。一个当前不能运行的线程并不在任何一个队列中出现。

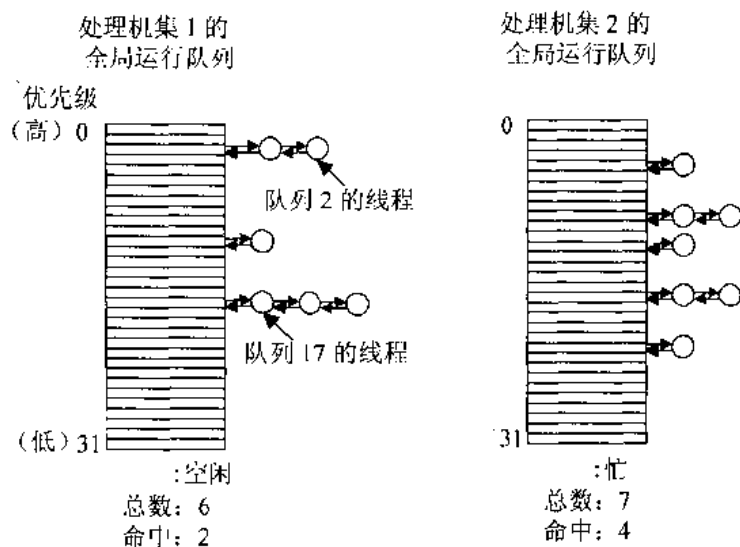


图 8-4 包含两个处理机集的系统的全局运行队列

每一个运行队列有三个变量。第一个变量是用于实现数据结构锁定的互斥体。它用来保证一次只有一个 CPU 操作队列。第二个变量是所有队列的线程数之和。一旦这个变量变为 0 就没有任何工作。第三个变量是指向优先级最高的线程的线程。它保证了没有线程具有更高的优先级，但是当前最高优先级的线程有可能是较低的优先级。这个线程可以帮助系统寻找优先级最高的线程以避免查找数组头部空队列的情况。

除了图 8-4 所示的全局运行队列之外，每一个 CPU 也有它自己的本地运行队列。每一个本地队列中保持着所有始终绑定到这个 CPU 的线程，这有可能它们是从属于某个 CPU 的 I/O 设备的驱动器。由于它们只能在某个 CPU 上运行，所以将它们放入全局运行队列是不正确的（一个“错误”的 CPU 有可能会选择并执行这些线程）。

我们下面介绍基本调度算法。当一个线程阻塞、退出或者耗尽它的时限，该线程所在的 CPU 首先查找本地队列看是否有活动的线程。这种检查仅仅是察看本地运行队列的计数器。如果它非 0 则 CPU 开始在队列头部查找线索所指向的优先级最高的线程。如果本地运行队列为空，就在全局运行队列中使用相同的算法，唯一的区别就在于全局队列在进行查找之前必须先要进行封锁。如果在这两个队列中都没有可以运行的线程，那么系统会运行一个特殊的 idle 线程直到有某个线程等待运行为止。

如果找到了一个可以运行的线程，它就被调度并运行一个量程 (quantum)。在量程结束时，系统会检查本地和全局运行队列来寻找具有相同或更高优先级的线程，前提条件是所有本地运行队列线程的优先级要比所有全局运行队列线程的优先级高。如果找到了合适的线程，系统就会进行线程交换。如果没有找到，那么这个线程就可以再运行一个量程。线程也可以被抢占。在多处理机中，量程的长度可以根据可以运行的线程数量的不同而不同。可以运行的线程数量越多和 CPU 的数量越少，量程的长度就越短。这种算法即使在重负载的系统中对于短请求也具有良好的响应时间，而在轻负载的系统中则效率很高（即在这种系统中量程很长）。

在每一个时钟滴答，CPU 都会将当前运行线程的优先级计数器递增一个很小的量。随着数值的增长，线程的优先级不断降低，最终该线程会移到优先级数值更高（即优先级更低）的队列中去。优先级计数器也会随着时间的流逝而降低。

对于一些应用来说，有可能会出出现许多线程共同工作来解决一个问题的情况，这时就需要对调度进行详细控制。Mach 提供了一个钩子（hook）来实现对线程调度的一些附加控制（除了处理机集和优先级之外）。钩子是允许一个线程将自己的优先级降到绝对最低并保持若干秒的系统调用。这个操作给其他线程一个运行的机会。当时间耗尽之后，进程的优先级会被恢复到原始值。

这个系统调用还有另外一个有趣的属性：如果需要的话，它可以指定继任者。例如，通过给另外一个线程发送消息，发送消息的线程可以放弃 CPU 而要求接收消息的线程继续运行。这种被称为传递调度（handoff scheduling）的机制完全旁路了运行队列。如果使用得好的话，这种机制可以加强系统的效率。内核也可以在某些情况下作为一种优化途径使用这种方法。

Mach 系统中还可以配置为相关调度，但是通常情况下这个选项是关闭的。当这个选项打开时，内核将一个线程尽量调度到它上次运行的 CPU 上去，这样做的原因是这个线程的部分地址空间可能还在这个 CPU 的缓存中。相关调度只有在多处理机中才可以应用。

最后，系统的某些版本还支持包括供实时应用使用的其他一些调度算法。

8.3 Mach 的存储管理

Mach 有一个强大（Powerful）、精细和高度灵活的基于分页的存储管理系统，它具有其他少数（few）操作系统才具有的一些特点。但是特别之处在于它以一种极为清楚和特殊的方式将存储管理系统中机器相关部分和机器无关部分区分开来。这种区分使得存储管理的移植性要比其他系统强得多。而且存储管理系统同后面将要介绍的通信系统的联系也非常紧密。

Mach 存储管理与众不同的方面主要在于它的代码分为三部分。第一部分是 *pmap* 模块，它在内核中运行以管理 MMU。它设定了 MMU 的寄存器和硬件分页表，同时还俘获所有缺页中断。这部分代码是同 MMU 结构有关的，对于每一个 Mach 所必须支持的新 MMU，这部分代码都必须进行重写。

第二部分是机器无关的内核代码，主要用来进行缺页中断处理、地址映射管理和页面替换。

存储管理代码的第三部分是作为用户进程运行的内存管理器（memory manager），或者也可以称为外部分页器（external pager）。它处理内存管理系统的逻辑（同物理相对应）部分，主要是对回写（到磁盘）的管理。如虚拟页的当前使用情况，主内存中的页面使用情况以及没有在主内存中的页面存放位置等都是由内存管理器管理的。

内核和存储管理器之间的通信是通过一种定义明确的协议进行的，这就使用户可以自己编写内存管理器。这种任务的分工让用户可以为满足特殊要求而编写专门的分页系统。而且这种方法还具有因大量代码移到用户空间而使内核更小、更简单的潜在优势。另一方面，它还具有可以使系统更复杂的潜在优势。由于内核必须保护自己不受存在缺陷或

恶意的内存管理器的伤害，在内存管理中使用两个实体就造成了竞争情况的出现。

8.3.1 虚拟内存

Mach 用户进程所见到的内存概念模型是一个巨大的线性虚拟地址空间。对于大多数 32 位芯片来说，由于内核自己使用高的一半地址空间，所以用户地址空间一般是从地址 0 到地址 $2^{31}-1$ 。地址空间使用分页技术。由于分页技术是为了使用户使用虚存可获得同使用常规内存一样的效果，但实际上并不仅仅是这样。Mach 的虚拟内存管理从原理上没有什么需要特别阐述的方面。

在实际中则还有许多需要说明的地方。Mach 系统为虚拟页面的应用方式提供了很多细粒度的控制（进程非常关心这一点）。开始，地址空间可以稀疏使用。例如一个进程可能使用许多虚拟地址空间节（section），而这些节之间的距离可能有几兆，也就是在节之间存在巨大的未使用地址空洞（hole）。

从理论上讲，任何虚拟地址空间都可以以这种方式使用。所以使用许多广泛分布的节的权能并不真正是虚拟地址空间结构的一个特性。换一句话说，任何 32 位系统都应该允许一个进程在从 0 到极限 4GB 的空间上平均每 100MB 有一个 50KB 大小的数据节。然而在许多实现系统中，内核存储器中保存着一个从 0 到使用的最高页的线性页表。在一个页面大小为 1K 的机器中，这种配置需要有四百万个页表条目。即使还可能实现的话，这无疑也是非常昂贵的。即就是在多级页表结构中，这种稀疏使用也是非常不方便的。Mach 系统的目标是全面支持稀疏地址空间。

为了判断哪些虚拟地址正在被使用而哪些没有被使用，Mach 提供了一种称为区域（region）的分配（allocate）和回收（deallocate）虚拟地址空间节的方法。分配调用可以指定一个基地址和大小，在这种情况下就指定了区域，或者也可以只指定大小，在这种情况下系统会找到一个合适的地址区间并返回基地址。一个虚拟地址只有处于一个分配后的区域内才是有效的。任何使用分配后的区域之间的地址空间的尝试都会引起陷阱，如果进程需要的话可以捕获这些陷阱。

同虚拟地址空间的使用有关的一个关键概念是存储对象（memory object）。一个存储对象可以是一个页面或者一个页面集，但是它也可以是一个文件或其他更加专用的数据结构。一个存储对象可以被映射到虚拟地址空间的未使用部分，形成一个如图 8-5 所示的新区域。当一个文件被映射到虚拟地址空间之后，它就可以通过普通的机器指令来进行读写。映射后的文件采用普通的分页方式。当一个进程结束后，它的映射文件自动带着所有在映射期间的改变重新在文件系统出现。也可以按需要精确地解除文件或者其他存储对象的映射，释放它们所占用的虚拟内存并使它们可以为后来的分配或映射所使用。

除此之外，文件映射并不是唯一的文件访问方法，也可以通过常规方法访问它们。然而即便是这样，库函数也可能将它们进行映射而不是通过 I/O 系统来访问它们。这种方式允许文件页使用虚拟存储系统，而不是使用系统的其他专用缓存。

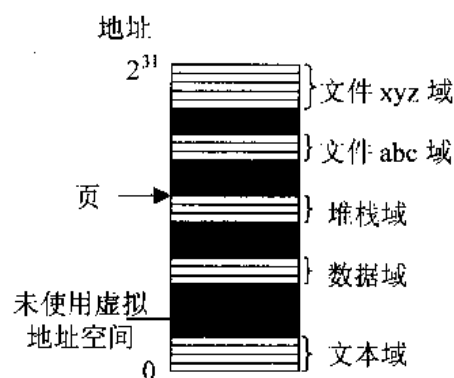


图 8-5 一个具有分配后的区域、已映射对象及未使用地址的地址空间

Mach 支持一些操作虚拟存储空间的调用。表 8-3 中列出了主要的调用。它们中没有一个是真正的系统调用。相反它们都向调用进程端口写消息。

表 8-3 管理虚拟存储的部分 Mach 调用

调用	说明
Allocate	建立虚拟地址空间的一个区域
Deallocate	使虚拟地址空间上的一个区域失效
Map	将一个存储对象映射到虚拟地址空间
Copy	在其他虚拟地址上建立一个区域的拷贝
Inherit	设定一个区域的继承属性
Read	从其他进程的虚拟地址空间读取数据
Write	向其他进程的虚拟地址空间写入数据

第一个调用 *allocate*，可以使一个虚拟地址空间上的区域处于可用状态。一个进程可能继承已经分配的虚拟地址空间，同时也可以分配更多的地址空间，但是任何引用没有被分配地址的企图都会失败。第二个调用 *deallocate*，使区域失效（也就是从存储映射中删除它），因而它也就可以被重新申请或者使用 *map* 调用映射其他东西。

Copy 调用将一个存储对象拷贝到一个新的区域，而原存储对象保持不变。一个存储对象可以通过这种形式在地址空间中多次出现。从概念上讲，调用 *copy* 同让该对象被循环程序所拷贝并没有区别。然而 *copy* 调用的实现经过了页面共享优化，这就避免了物理拷贝。

Inherit 调用影响着新进程建立时区域的继承方式。系统可以通过设定地址空间来使一些区域被继承而另外一些区域不被继承。我们将在下一节中讨论这个问题。

Read 和 *write* 允许一个线程访问属于其他进程的虚拟存储空间。这些调用要求调用者拥有操作远程进程端口的权力，而这种权力是进程在需要的情况下可以赋予它的伙伴（*friend*）进程的。

除了表 8-3 中所列出的之外还有少量其他调用。这些调用的作用主要是获取或者设定属性、保护模式以及各种统计信息。

8.3.2 存储共享

共享在 Mach 中具有非常重要的地位。一个进程中的线程共享对象不需要特殊的机制：它们都自动可以看到相同的地址空间。如果它们中的一个具有访问一个数据块的权力，那么大家也都有这个权力。更让人感兴趣的是两个或者多个进程如何共享相同的存储对象，或者在需要的时候只共享数据页。有些时候在单 CPU 系统中实现共享是非常重要的。例如在典型的生产者-消费者问题中，比较好的解决办法是让生产者和消费者各自具有不同的进程，但是它们共享一个共同的缓冲器以使生产者可以向缓存器中放入数据而消费者从中取出数据。

在多处理机系统中，两个或者多个进程之间对象的共享经常更加重要。在许多情况下都是一个由一组在不同 CPU 上并行运行（同单 CPU 系统中的时分复用相对应）的协同进程共同解决的。这些进程可能需要不断地访问共同的缓存、表或者其他数据结构以进行它们的工作。操作系统允许这种共享的发生是非常必要的。如早期的 UNIX 版本就不具有这种权能，虽然后来它增加了这种权能。

例如我们可以考虑一下处理卫星发送的地球图像的实时系统。这种分析不但非常费时，而且同样的图片还需要被用来进行气象预报、作物收成预测以及污染跟踪。当收到一张图像时，它是作为一个文件存储的。

多处理机系统适合进行这种分析。由于气象、农业以及环境程序存在很大差异，而且是由不同人员编写的，所以它们不应该是同一进程的不同线程。相反，它们都是独立的进程，而且每一个都将当前图像映射到它的私有地址空间，如图 8-6 所示。在这里我们要注意保存图像的文件可能在每个进程中被映射到不同的虚拟地址上。虽然每一个页在内存中只出现一次，但是它在每一个进程中的页面映射可能都不同。通过这种方式所有的三个进程都可以非常方便地同时对同一文件进行操作。

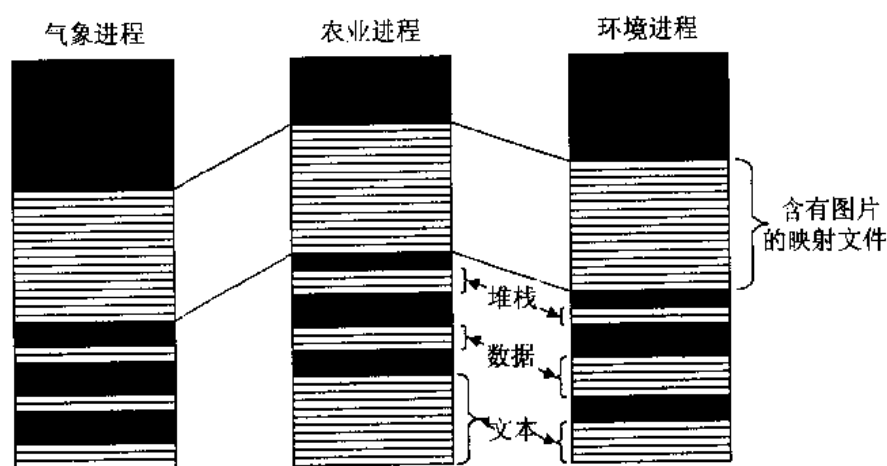


图 8-6 三个进程共享一个映射文件

共享的另外一个重要用途是进程创建。在 UNIX 和 Mach 中创建新进程的基本方式都是作为一个现存进程的拷贝。在 UNIX 中，一个拷贝总是执行 FORK 系统调用的进程的克隆 (clone)，而在 Mach 中子进程则可以是其他进程 (原型) 的复制。但是在这两种方

式下子进程都是某个进程的拷贝。

一种建立子进程的方法是拷贝所有必要的页面，然后再将这些拷贝映射到子进程的地址空间。虽然这种方式是可行的，但是它的代价过高。程序文本一般是只读的，所以它是不变的，而且部分数据也有可能是只读的。对于这些只读页面是没有必要进行拷贝的，因为仅仅将它们都映射到两个进程就可以完成任务。可写的页面并不总是能够共享，这主要是因为建立进程（至少在 UNIX 中是这样）的语义要求虽然在建立时父子进程是相同的，但是之后的各自的变化对于对方的地址空间来说就是不可见的。

而且子进程也有可能不需要一些区域（如某个文件映射）。如果子进程不需要这些区域那么为什么要费事为它们指定呢？

为了获得这些不同目标，Mach 允许进程指定其地址空间的所有区域的继承属性（inheritance attribute）。不同的区域可能有不同的属性。系统提供了三种属性：

- (1) 子进程中不使用这个区域；
- (2) 该区域在原型进程和子进程之间共享；
- (3) 子进程的区域是原型进程相应区域的一个拷贝。

如果一个区域的属性值是第一个，那么子进程中的相应区域没有被分配。对于它的引用同对于任何其他没有分配的引用一样会引起陷阱。子进程可以自由的分配这个区域或者在其上映射一个存储对象。

第二个选项是真正共享。区域内的页在原型以及子进程的地址空间中同时出现。只要有一个产生了改变，另一个就会知道。实现 UNIX 系统调用不采用这种选择，但是经常用于其他目的。

第三种情况是，拷贝区域中的所有页，并把它们映射到地址空间。FORK 就使用这个选项。实际上 Mach 并不真正拷贝这些页，而是使用了一个称为写拷贝（copy-on-write）的小花招。它将所有必需的页放到子进程的虚拟存储表中，但是将它们都标注为只读，如图 8-7 所示。只要子进程对这些页面只进行读引用，整个系统就可以正常工作。

然而如果子进程需要在任何页面上进行写操作时就会发生写保护错误。操作系统这时就为这个页面建立一个拷贝，然后将这个拷贝映射到子进程的用户空间以替代那里的只读页面。新页面被标记为可读写。在图 8-7 (b) 中，子进程要求写页面 7。这个动作造成页面 7 被拷贝到页面 8，然后页面 8 被映射到子进程中以替代页面 7。由于页面 8 是可以读写的，所以随后的写操作并不会引起陷阱。

写拷贝同当新进程建立时直接进行所有拷贝工作相比较具有一些优点。首先，一些页面是只读的，所以没有必要复制它们。第二，其他页面中有一些可能永远也不会被引用，所以它们即是可写也没有必要进行拷贝。第三，还有其他一些页面在子进程中是需要进行释放而不是使用的，所以在这里避免进行拷贝页面是值得的。在这种方式下，只有子进程真正进行写操作的页面才会被复制。

写拷贝也有一些缺点。首先管理将更加复杂，因为系统必须知道哪一些页面是真正只读，对它执行写操作将引起程序错误，以及哪一些是需要经过拷贝并完成写操作的页面。另外，写拷贝需要多内核陷阱，每一个最终写的页面都有一个。根据硬件的不同，一个具有多个页面拷贝的内核陷阱的代价要比具有单页拷贝的多内核陷阱代价可能高不了多少。最后，写拷贝方式不能在网络上运行，它总是需要进行物理传输。

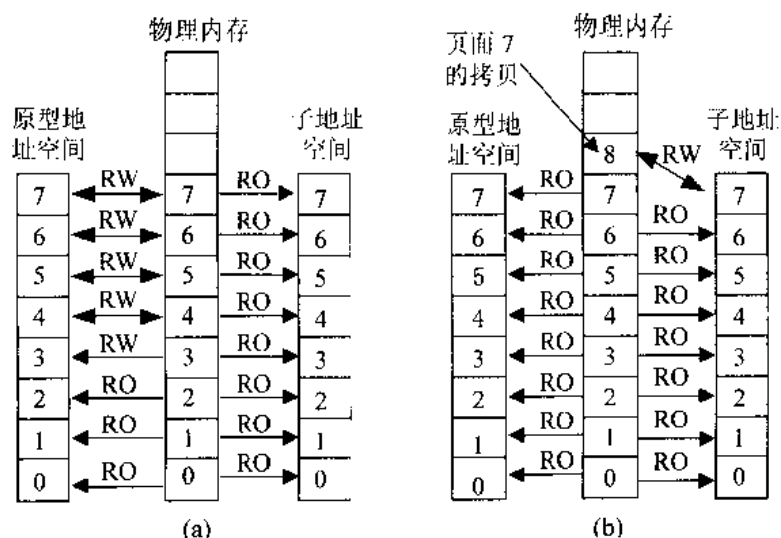


图 8-7 写拷贝操作

- (a) 在 `FORK` 之后，所有子进程的页面被标注为只读
- (b) 当子进程写页面 7 时对它进行拷贝

8.3.3 外部存储管理器

在对 Mach 存储管理讨论的一开始，我们就简要提到过用户级的存储管理器的存在。现在让我们来详细对它进行讨论。每一个映射到一个进程的地址空间的存储对象都必须有一个外部存储对象来控制它。不同的存储对象类由不同存储管理器进行处理。每一个都可以实现它自己的语义、可以决定在哪里存储不在内存中的页面以及提供当它们的映射清除时对象的变化规则。

为了将一个对象映射到一个进程的地址空间，这个进程给一个存储管理器发送消息要求它来进行映射。执行这个工作需要三个端口。首先存储管理器建立一个对象端口 (object port) 以供内核通知存储管理器缺页中断以及其他与对象相关的事件。第二个是内核自己建立的控制端口 (control port)，这样存储管理器就可以对这些事件进行响应 (许多都需要存储管理器进行一些工作)。使用不同端口的原因主要在于端口没有方向这一事实。对象端口是内核写，存储管理器读的；而控制端口的工作方式则正相反。第三个端口，命名端口 (name port)，是作为一个名字来标识一个对象。例如一个线程可以给内核一个虚拟地址，然后请求它所从属的区域。返回的结果就是指向一个命名端口的指针。如果地址属于相同的区域，那么用这个相同的命名端口进行标识它们。

当存储管理器映射一个对象时，它把权能作为参数之一提供给对象端口。内核接着会根据这个端口建立其他两个端口，然后给对象端口发送一个初始化消息来告诉它管理端口和命名端口。存储管理器给内核返回有关对象属性的应答，同时告知内核当结束映射之后这个对象是否需要在它的缓存中继续保留。开始时所有的对象页面都被标注为不可读/不可写，用以在第一次使用时强制产生一个陷阱。

在这时存储管理器对对象端口执行一个读操作然后阻塞。存储管理器保持处于空闲

状态直到内核通过给对象端口写消息进行申请为止。映射在对象内的线程现在就会解除阻塞而处于可以运行的状态。

线程无疑早晚会要求读或者写一个属于存储对象的页。这个操作会引起一个缺页中断以及内核陷阱。内核然后就会通过对象端口给存储管理器发送一个消息，这个消息告知它哪一个页被引用了并且要求它提供这个页。这个消息是异步的，因为内核不能为了等待一个有可能不会回应的用户进程而阻塞它自己的任何线程。在等待回应的同时，内核挂起失效线程并调度其他线程运行。

当存储管理器接收到缺页中断的消息，它首先检查这种引用是否合法。如果不合法的话它就会给内核返回一个错误消息。如果合法的话存储管理器就会以出现申请的对象所适合的方法来获取这个页。如果这个对象是一个文件的话，存储管理器会查找正确的地址并将页面读入它自己的地址空间。然后它会给内核返回一个指向这个页的指针来作为回应。内核将这个页映射到出现失效的线程的地址空间，这个线程现在就可以恢复。这个过程可以按需要重复以获取所有需要的页。

为了保证系统中有稳定的空闲页帧，内核中的一个分页后台线程会不时被唤醒来检查内存中的状态。如果系统中没有足够的空闲页整，它就会使用二次机会算法释放一个页。如果这个页是“清洁”的，它通常会被丢弃。如果这个页是“脏”的，后台线程会将它送到管理该页的存储管理器中。存储管理器应该将这个页写回磁盘然后在工作结束后发送消息。如果该页属于一个文件，存储管理器将先查找该页在文件中的偏移然后将它写回到那里。

页面可以被标注为珍贵 (precious)，在这种情况下它们将永远不会被丢弃，即使它们是清洁的也是如此。它们将总是被送回它们的内存管理器。珍贵页可以被使用，例如可以作为网络上仅有一份不能丢弃的拷贝共享。通信也可以由存储管理器初始化，如当 SYNC 系统调用执行到冲洗缓存回写到磁盘。

这个分页后台线程作为内核的一部分出现并没有什么意义。虽然页置换算法是完全机器相关的，但是在充满分属于不同存储管理器的页面的存储器中并没有一个很好的算法去要求它们中的一个去决定选择一个页面放弃。可行的唯一办法是在不同的管理对象之间静态地分配页面，然后由管理对象在自己的页面集合中进行页置换。然而由于全局算法一般要比局部算法效率高，所以一般并不采纳这种解决办法。后来在这个问题上继续进行了一些工作，请参考 (Harty 与 Cheriton, 1992; 以及 Subramian, 1991)。

除了文件映射对象和其他专用对象之外还有一种处理“普通”分页存储器的默认存储管理对象。当一个进程通过 *allocate* 调用分配一个区域时，它实际上通过一个缺省管理器来映射对象。这个管理器在需要的情况下可以用零来填充页面。它使用一个临时文件来作为缓冲空间，而不是像 UNIX 那样使用独立的交换区。

为了使外部存储管理器的思想真正能够实现，内核与存储管理器之间的通信还需要严格遵守一种协议。这个协议由一些内核可以发送到存储管理器的消息以及一些存储管理器给内核的应答组成。所有的通信都是由内核为某个存储对象的对象端口以异步消息形式进行初始化的。存储管理器随后会在控制端口上返回异步应答。

表 8-4 列出了内核发送给存储管理器的主要消息类型。当一个被映射的对象使用表 8-3 中的 *map* 调用时，内核会给合适的存储管理器发送 *init* 消息使其初始化。消息中还指定

了后来与对象进行通信的端口。内核在申请页面以及传送页面时分别使用 *data_request* 和 *data_write*。这些处理了页面的双向通信，因而诸如此类的调用是最重要的。

表 8-4 部分从内核到外部存储管理器的消息类型

调用	说明
Init	初始化一个新映射存储对象
Data_request	为内核传递一个可以处理缺页中断的特殊页
Data_write	从内存中获取一个页并将它写回磁盘
Data_unlock	为页面解锁以使内核可以使用它
Lock_completed	前一个加锁要求已经完成
Terminate	接到这个对象已经不再使用的通知

Data_unlock 是一个内核用来请求存储管理器给一个加锁的页进行解锁的消息，它可以使这个页供其他进程使用。*Lock_completed* 标志着 *lock_request* 的结束，下面还将对它进行讨论。最后，*terminate* 告诉存储管理器消息中所指明的对象已经不再使用而且可以被移出内存。实际中还存在专为缺省存储管理器使用的同名调用，其他少量管理属性以及错误处理调用也同样如此。

表 8-4 所列出的消息是从内核到存储对象的。表 8-5 列出了从存储管理器到内核的反方向应答。存储管理器可以用它们作为上述请求的应答。

表 8-5 部分从外部存储管理器到内核的消息类型

调用	说明
Set_attribtes	Init 的应答
Data_provided	请求的页在这里(对 <i>Data_request</i> 的应答)
Data_unavailable	没有可以使用的页(对 <i>Data_request</i> 的应答)
Lock_request	请求内核清空、冲洗 (flush) 或者锁定页面
Destroy	清除一个已经不再需要的页面

第一个，*set_attribtes*，是 *init* 的应答。它告诉内核现在可以处理新的映射对象。即使没有任何映射对象，应答中也会包含是否对对象进行缓存的状态位。下两个是 *date_request* 的应答。这个调用请求存储管理器提供一个页面。这两个应答就分别取决于它是否能够提供这个页。前一个可以提供，而后一个则不行。

Lock_request 可以供存储管理器请求内核清空某个页面，也就是说给它发送可写回磁盘的页面。这个调用也可以用来改变页面的保护模式(读、写、执行)。最后，*destroy* 告诉内核某个对象已经不再需要了。

值得一提的是当内核给存储管理器发消息时，内核就执行有效的上行调用 (upcall)。虽然这种方式可以获得一定的灵活性，但是一些系统设计人员认为这并不是内核调用用户程序为其提供服务的最佳方法，因为一般是低层为高层提供服务而不是相反。

8.3.4 Mach 中的分布式共享存储

Mach 外存储管理器的概念可以实现一个很好的基于页面的分布式共享存储。在本节中我们将简要介绍一下这方面的一些工作。详细信息请参考 (Forin 等, 1989)。作为基本概念的回溯, 我们的核心思想是使用能够被所有进程共享的单一线性虚拟地址空间, 而这个地址空间则在没有物理共享内存的机器上运行。当线程需要引用一个不存在的页面时就会引起一个缺页中断。最后, 该页将在定位后传送到发生缺页中断的机器中进行安装, 该线程就可以继续工作了。

由于 Mach 中的不同对象类已经有不同的存储管理器, 所以自然就需要引入一个新的共享页面存储对象。共享页面是由一个或者多个特殊存储管理器所管理的。一种可能的方式是让一个存储管理器管理所有的共享页面。而让每一个页面或页面组都由不同的存储管理对象来进行管理以获得负载均衡也是另外一种解决方法。

还有一种可能性是让所有不同的页面存储管理器都有不同的语义。如一个存储管理器可以保证完整的存储一致性, 这就意味着在写操作之后的所有读操作都可以得到最新的数据。而另外一个存储管理器则可能会提供稍弱一些的语义, 如一个读操作永远不会返回过期超过 30 秒的数据。

让我们来考虑一下最基本的情况: 一个共享页、集中式控制以及完全的存储一致性。所有其他页面都是某台机器的本地页面。为了实现这种模式, 我们需要一个为系统中所有机器提供服务的存储管理器。我们将它称为 DSM (Distributed Shared Memory 分布式共享存储) 服务器。DSM 服务器处理所有对共享页面的引用。而传统的存储管理器则处理其他的页面请求。直到现在为止, 我们一直都将某台机器服务的一些存储管理器或者一个管理器默认为是这个机器本地的。实际上由于 Mach 中的通信是透明的, 所以一个存储管理器并不一定要位于它所管理的存储空间所在的机器上。

共享页面总是可读或可写的。如果它是可读的话, 它将在多台机器上进行复制。如果它是可写的话, 一般只有一个拷贝。DSM 服务器总是知道共享页面的状态以及它当前所处的机器。如果该页面是可读的, DSM 自己也有该页面的一份有效拷贝。

假设一个页面是可读的, 而且在某处有一个线程在尝试对它进行读取。DSM 服务器会给这个机器发送共享页面的一个拷贝, 同时更新自己的表以记录这个读者并结束工作。这个页面将在做读操作的新的机器上进行映射。

现在让我们假设有一个线程需要写这个页面。DSM 服务器会发送消息给所有具有这个页面拷贝的内核来收回这个页面。在这时并不需要真正的传送这个页面, 因为 DSM 服务器自己就有一份有效的拷贝。所需要确认的只是该页面已经不再使用了。当所有的内核都已经释放该页面之后, 写操作者被赋给使用这个页面的一个拷贝的排他权限 (供写操作使用)。

如果这时有其他线程申请这个页面 (当它仍然是可写时), DSM 服务器将要求当前的使用者停止使用这个页面并将它发送回来。当该页面被发送回来之后, 它就可以供一个或者多个读者或者一个写操作者使用了。这种集中方式还有其他一些实现方法, 如当前机器占有这个页面的时间没有达到某个最小值, 则服务器不能收回一个页面。也可以实现分布式的解决方案。

8.4 Mach 中的通信

Mach 中通信目标是以可靠而灵活的方式支持不同风格的通信 (Draves, 1990)。它可以处理异步消息传递、RPC、位流以及其他的通信方式。Mach 的进程间通信机制是基于它先辈 RIG 和 Accent 的。根据这种演化, 这种机制已经为本地使用 (单节点) 而不是远程使用 (分布式系统) 进行了优化。

我们首先介绍一下单节点情况, 然后回到如何对它进行网络扩展。应该注意的是在这些情况下, 一个多处理机是一个单节点, 所以在同一多处理机不同 CPU 上进程之间的通信属于本地通信。

8.4.1 端口

Mach 中所有通信的基础是内核中一个称为端口 (port) 的数据结构。一个端口实质上是一个受保护的邮箱。当进程中的一个线程需要同其他进程的线程进行通信时, 发送方向端口中写入信息而接收方从端口获取信息。系统保护每一个端口以保证只有经过授权的进程才可以向它发送或者取出消息。

端口支持单向通信, 非常类似于 UNIX 中的管道。一个客户端用来向服务器发送请求的端口不能被服务器用来发送回应, 而需要另外一个端口来进行这个工作。

端口支持可靠的顺序消息流。如果一个线程向一个端口发送一个消息, 系统将保证这个端口一定可以接收到这个消息。消息不会因为错误、溢出或者其他原因丢失 (至少是在系统没有发生崩溃的情况下)。从一个单线程发出的消息系统保证按序投送。如果两个线程以交叉形式轮流写同一个端口, 系统则不会为这些消息提供任何顺序保证, 这主要是因为内核中可能由于加锁和其他一些因素而使用缓存。

同管道不相同的是端口支持消息流, 而不是字节流。消息是永远不会被连接的。如果一个线程向一个端口写入了五个 100 字节的消息, 消息接收者永远会看到五个不同的消息, 而不是看到一个 500 字节的消息。当然如果不重要的话, 高级软件也可以忽略消息的边界。

图 8-8 中显示了一个端口。当建立一个端口时, 系统内核将为它申请 64 个字节内核存储空间并一直保留到端口被清除时为止。这种清除可能是明确说明的, 也可能是在某种条件下隐含说明的, 如当所有使用这个端口的进程都结束的情况。图 8-8 列出了端口所包含的大部分域。

消息实际上并不是由端口自己保存的, 而是保存在内核中一个称为消息队列 (message queue) 的数据结构中。端口中包含了当前在消息队列中所出现的消息个数以及所允许的最大消息个数。如果该端口属于一个端口集合, 在端口中还将有一个指向端口集合数据结构的指针。我们前面已经提到过一个进程可以将使用它自己端口的权能赋给其他进程。出于许多方面的考虑, 内核必须知道每一种类型中的权能授权数量, 所以端口就必须保存这个数字。

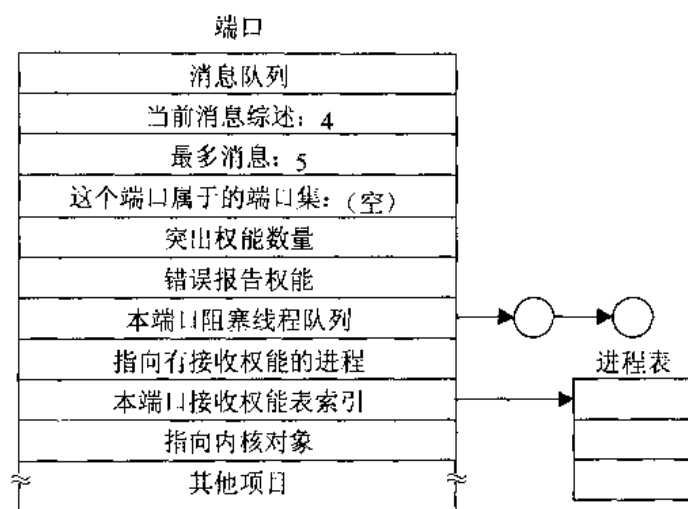


图 8-8 Mach 端口

如果在使用端口时发生了某种错误，它们是通过给持有权能的进程端口发送消息来进行报告的。在读一个端口时线程可以被阻塞，所以端口还包含了一个指向阻塞线程列表的指针。由于找到具有读权能的进程（只能有一个）对于端口来说意义非常重大，所以端口中也包含了这个信息。如果端口是一个进程端口，那么 `next` 域是一个指向端口所从属的进程的指针。如果端口是一个线程端口，这个域将保存指向内核中该线程数据结构的指针。其他一些必须的域在这里没有进行说明。

当一个线程建立端口时，它首先获取一个类似 UNIX 中文件描述符的标志端口的整数。这个整数在随后的读写端口调用中用来标志所使用的端口。由于端口的使用情况是按进程来记录的，而不是按线程，所以如果一个线程建立了一个标志为 3 的端口，那么这个进程中的其他进程将不可能再建立标志为 3 的端口。内核实际上甚至并不记录线程建立端口的情况。

一个线程可以将端口访问权限赋给其他进程的线程。但是很明显，它不可能通过将端口标志放在消息中完成这个操作，就像在 UNIX 系统中进程可以通过给管道中写入 1 来将一个文件描述符作为标准输出。实际所使用的模式是由内核进行保护的，我们后面将对它进行详细讨论。在目前我们只需要知道它是可以实现的就足够了。

在图 8-9 中我们可以看到一个具有两个进程 A 和 B 的情况，它们都有访问一个端口的权力。A 刚刚给端口发送过一个消息，而 B 也刚刚读过消息。消息头与消息体开始从进程 A 拷贝到端口，后来又从端口拷贝到进程 B。

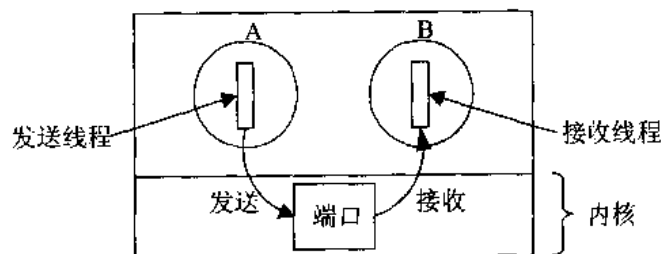


图 8-9 通过端口进行的消息传递

为了方便起见，端口可以分成端口集。一个端口最多只能从属于一个端口集，而且也有可能从一个端口集进行读取（但不是向它写）。例如一个服务器可以采用这种方式同时从许多端口中读取信息。内核从端口集中某个端口中返回一个消息。但是端口的选择是不一定的。如果所有的端口都为空，则这个服务器被阻塞。服务器可以通过这种方式来维护它所支持的每一种对象的端口，同时可以在不为每一种对象建立专用线程的情况下接收它们的所有消息。当前的实现是将所有端口集的消息统一成单一链表形式。而在实际中从一个端口中接收消息和从一个端口集中接收消息的唯一不同之处在于接收者需要知道后者的实际端口而前者则不必。

一些端口的使用方式比较特殊。每个进程都有一个特殊的用来同内核进行通信的进程端口（process port）。同进程有关的大多数“系统调用”都是通过向这种端口中写消息实现的。线程自己同样也有专门实现同线程有关的“系统调用”的端口。同 I/O 驱动通信也使用端口机制。

权能（capability）

作为进程的第一个大致描述，内核中有一个表记录着一个进程所能够访问的所有端口。这个表在内核中被安全保存，任何用户进程都不能对它进行访问。进程通过它们在这个表中的位置来引用这个表中的端口，也就是条目 1 或条目 2 等等。这些表的条目就是典型的有效权能。我们将它们称为权能，这个保存权能的表称为权能列表（capability list）。

每一个进程都有且仅有一个权能列表。当一个线程请求内核为它建立一个端口时，内核执行这个操作并且将它作为一个权能条目插入线程所属进程的权能列表。调用线程同该进程内的所有其他线程对权能具有相同的访问权能。返回线程的用来标志权能的整数通常是权能列表的索引（但它也可能会是一个大的整数，如一个机器地址）。我们将这个整数称为权能名（capability name）。或者有时只简单的将它称为权能，条件是上下文环境必须很明确我们是指索引，而不是权能自身。它总是一个 32 位整数，而不是一个字符串。

每一个权能并不仅仅包括一个指向端口的指针，而且还包括权能的拥有者对端口所具有的访问权限。进程中的所有线程都被认为是进程权能的同等拥有者。现在主要有三种权限：RECEIVE、SEND 和 SEND-ONCE。RECEIVE 使拥有者具有从端口中读取消息的权限。前面我们已经提到过 Mach 中的通信是单向的。这实际上意味着在任何时候都只有一个进程能够拥有一个进程的 RECEIVE 权限，即将相应权能从权能表中移出。RECEIVE 权限可以从一个进程传递到另一个进程，但是这意味着传送者将不再拥有 RECEIVE 权限。因而每一个端口都只有一个唯一的潜在接收者。

具有 SEND 权限的权能允许拥有者向一个特定端口写消息。具有向一个端口写消息的权能的进程可以有許多。这种情况类似于许多国家的银行系统：任何知道银行帐户的人都可以向其中存钱，但是只有真正的主人才可以取钱。

SEND-ONCE 权限允许而且只允许进程向端口发送一次消息。在发送结束后内核将收回这个权能。这种机制用来实现请求-响应协议。例如，一个客户需要向服务器请求一些东西，于是它将为回应消息建立一个端口。它然后就可以给服务器发送包含对回应端口具有 SEND-ONCE 权限的（受保护的）权能在内的请求。当服务器发送应答之后，这个权能将从权能列表中清除，而这个权能名在以后还可以重复使用。

这种权能名只在单个进程内才有效。有可能会出现两个进程具有访问相同端口但是

用不同权能名的情况出现，这就像两个 UNIX 进程可能访问同一个打开文件但是使用不同的文件描述符来进行读操作一样。在图 8-10 中，两个进程都拥有端口 Y 的发送权能，但是在 A 进程中是权能 3，而在 B 进程中是权能 4。

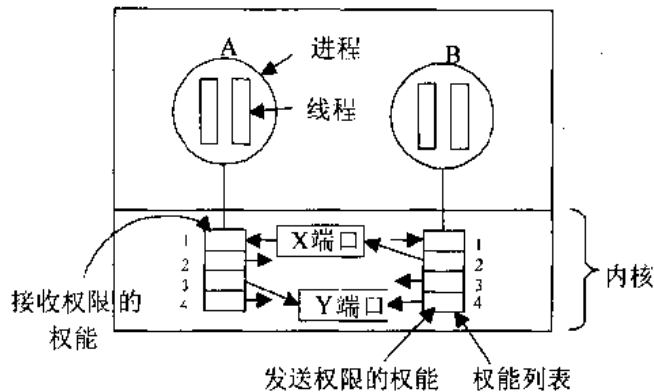


图 8-10 权能列表

一个权能列表是同特定进程相绑定的。当一个进程退出或者被杀死时，它的权能列表也将清除。该进程权能列表中所有具有 **RECEIVE** 权限的端口也同时失效，因而会被清除，而不管它们可能还包含有没有送到（现在也不可能送到）的消息。

如果一个进程中的不同线程多次请求相同的权能，权能列表中只会有一个条目。为了记录这种申请的次数，内核为每一个端口保留了一个引用值。当需要删除一个权能时，首先会递减这个参考值。只有在参考值为 0 时这个权能条目才会真正从权能列表中释放。由于不同线程之间可能独立进行权能的申请与释放，所以这种机制就非常重要了。例如 UNIX 仿真库和正在运行的程序之间的关系就是如此。

每一个权能列表条目都是以下四个项目之一：

- (1) 一个端口的权能；
- (2) 一个端口集的权能；
- (3) 一个空条目；
- (4) 一个指示该端口当前已经死去的代码。

第一种情况我们已经详细地进行了讨论。第二种情况允许一个线程从一组端口中读消息而不用知道这个权能名是由端口集来记录而不是单个端口记录的。第三种情况是表明当前该条目处于空闲状态。如果分配给一个端口的权能条目被释放，那么它就会被空条目所取代以表明它处于空闲状态。

最后，第四种选项表示端口已经不再存在但是仍然有具有 **SEND** 权限权能的端口存在。例如当端口被删除时，由于包含 **RECEIVE** 权能的进程已经退出，内核将查找所有具有 **SEND** 权能条目并将它们标注为死亡。任何对空或者已经死亡的权能所进行的发送都将得到适当的错误代码。当一个端口的所有 **SEND** 权能无论因为什么原因都失去了，内核（可选）会给接收者发送一个消息告诉它没有发送者存在，以后也不会再有任何消息。

端口管理原语

Mach 提供了大约 20 条用来管理端口的调用。它们都是通过给进程端口发送消息来实现的。表 8-6 列出了其中的一些重要调用。

表 8-6 Mach 中的部分端口管理调用

调用	说明
Allocate	建立一个端口并将它的权能插入权能列表
Destroy	清除一个端口并在列表中清除所有的权能
Deallocate	从权能列表中清除一个权能
Extract_right	从其他进程析取第 n 个权能
Insert_right	在其他进程的权能列表中插入一个权能
Move_member	将一个权能移入权能集中
Set_qlimit	设定一个端口可以保持的消息数量

第一个调用 *allocate*，建立一个新的端口并将它的权能插入到调用者的权能列表中。该权能是从端口中进行读取。系统返回一个可以使端口可供使用的权能名。

下两个调用进行的工作同第一个正相反。*Destroy* 删除一个权能。如果它是一个 *RECEIVE* 权能，则端口会被清除，同时该端口的所有进程所具有的其他权能被标注为死亡。*Deallocate* 调用则递减权能的参考值。如果这个参考值为 0，这个权能就会在不影响端口的前提下被清除。*Deallocate* 只能用来清除 *SEND*、*SEND-ONCE* 权能或标注为死亡的权能。

Extract_right 允许一个线程从其他进程的权能列表中选择一个权能并将其插入自己的权能列表中。当然，调用线程必须具有访问控制其他进程的进程端口的权能（例如它自己的子进程）。*Insert_right* 则正好相反。它允许一个进程将它自己的一个权能取出并将它插入到（例如）一个孩子的权能列表中。

Move_member 调用用来管理端口集。它可以给一个端口集中增加端口或者从端口集中删除端口。最后，*set_qlimit* 决定了一个端口所能够容纳的消息数量。当端口建立时，缺省值是 5 个消息，但是这个调用可以对这个数字进行增减。消息的长度在这里并没有限制，因为它们并不在端口上实际保存这些消息。

8.4.2 消息的发送与接收

端口存在的目的是为了发送消息。在本节中我们将介绍消息是如何发送以及如何接收的。Mach 系统中有一个单一的用于消息发送与接收的系统调用。这个系统调用是封装在一个称为 *mach_msg* 的库函数中。它有七个参数和许多可选项。为了让大家对它的复杂性有更进一步的了解，光它所能够返回的不同错误信息就有 35 种。下面我们将简要介绍它的一些主要的权能情况。幸运的是它主要是在存根编译器生成的过程中使用的，而不是人工编程使用。

消息的发送和接收都使用 *mach_msg* 调用。它可以将一个消息发送到端口后立即将控制权返回给调用者，这样调用者就可以在不影响发送的情况下对消息缓冲区进行修改。它也可以从一个端口读取消息，如果该端口为空则它被阻塞或者在一段时间之后放弃读取操作。最后，它还可以同时进行这两种操作，也就是先发送消息，然后阻塞直到应答返回为止。在后一种情况下，*mach_msg* 可以当成 *RPC* 使用。

一个典型的 *mach_msg* 调用形式如下：

```
mach_msg (&hdr, options, send_size, rcv_size, rcv_port,
timeout, notify_port);
```

第一个参数 *hdr* 是一个指针，它指向将要发送的消息位置，或者指向将要接收的消息的存储位置，也可以同时表明两者。消息由一个固定的头部和紧随其后的消息体组成。图 8-11 中显示了这种结构。我们后面将详细解释消息格式，但是现在我们只需要注意消息头中包含了目标端口的权能名。内核需要用这些信息来指明消息的去向。当执行一个纯粹的 **RECEIVE** 操作时，并没有填写消息头，因为它将会被接收的消息全部覆盖。

Mach_msg 调用的第二个参数 *options*，包含了一个指示是否为读消息的位以及一个指示是否为写消息的位。如果这两个位都处于打开状态，则执行一个 **RPC** 操作。另外还有一个位用于表示超时，超时参数由 *timeout* 参数以毫秒为单位给出。如果一个请求在 *timeout* 所指定的时间段内无法完成的话，调用就将返回一个错误代码。如果 **RPC** 操作的 **SEND** 部分超时（主要是因为目标端口的消息队列处于满的状态时间太长），那么 **RECEIVE** 操作就甚至不会执行。

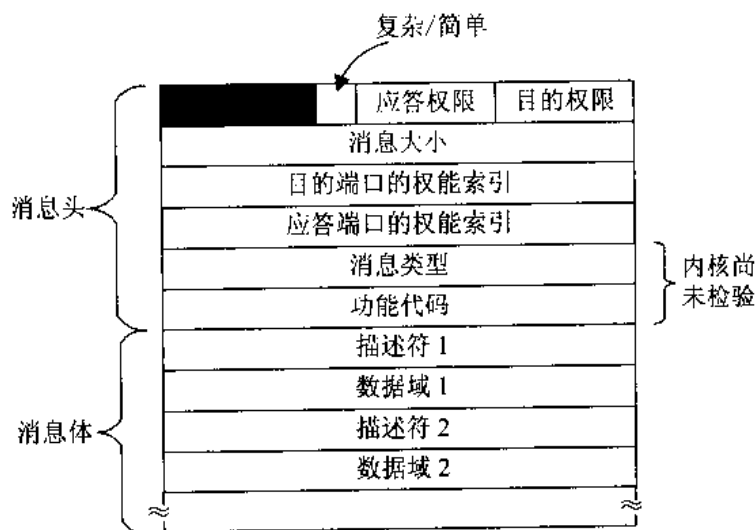


图 8-11 Mach 的消息格式

Option 中的另一个位允许无法完成任务的 **SEND** 操作立即返回控制权，而后再给 *notify_report* 端口发送消息报告状态。如果 *notify_report* 端口的权能不适合或者在状态通知到达之前被改变，则系统可能会出现任何错误。这个调用甚至会损坏 *notify_report* 本身（调用可能会造成巨大的副作用，就像我们后面将要介绍的一样）。

Mach_msg 调用可能会在执行的中途被应用程序中断，所以 *options* 中有一个位说明发生这种情况后应该放弃还是应该重试。

Send_size 和 *rcv_size* 分别说明输出的消息长度以及为接收消息准备的空间大小。*Rcv_port* 是接收消息时使用的。它是需要监听的端口或端口集的权能名。

下面让我们研究一下图 8-11 所示的消息格式。第一个字包含了一个说明消息是简单消息还是复杂消息的位。两者的不同之处在于简单消息不能携带权能或者受保护的指针，而复杂消息则可以。简单消息所需要的内核工作量较小，因而效率更高。这两种消息都拥

有下面描述的系统决定的结构。*Message size* 消息域说明了消息的全部长度。这个信息是消息的发送者和接收者都需要的。

下面两个是权能名（也就是发送者权能列表的索引）。第一个指出了目标端口；第二个可以指出应答端口。例如在客户-服务器结构的 RPC 中，目标域决定了服务器，而应答域则说明了服务器发送应答的端口。

最后两个消息头的域，内核并没有使用。高级应用软件可以按它们的需要使用这两个域。根据惯例，它们用来指定消息的类型以及一个功能码或操作码（也就是对于一个服务器来说，这个请求是要读还是要写）。这种使用方法在以后一定会发生变化。

当一个消息被发送并成功接收之后，它将拷贝到目标端口的地址空间中。但是实际中可能发生目标端口队列已满的情况。这种情况的解决随选项以及对目标端口所具有的权限不同而不同。一种可能性是发送者被阻塞到端口出现空间为止。另一种情况是发送者超时。在某些情况下则可以超过队列应有长度而继续发送。

另外值得一提的是同消息接收有关的问题。例如，如果新接收的消息比缓冲区大怎么办？在这种情况下有两种选择：抛弃这条消息或者使 *mash_msg* 调用失败并返回空间大小，这样就可以使调用者在申请到更大空间之后继续尝试。

如果有多个线程阻塞在读一个端口上，那么当一个消息到达时，系统会选择其中一个线程并使其获得消息。剩余的线程继续保持阻塞状态。如果所读取的端口实际上是一个端口集，那么在一个或者多个线程阻塞在读该端口集时，这个端口集的端口组成仍然可以发生变化。在这里我们只需要说明系统中有明确的规则来控制所有与之类似的情况就足够了。

消息格式

一个消息体既可以简单，也可以复杂，它主要由前面所提到的文件头进行控制。复杂消息的结构如图 8-11 所示。一个复杂消息体由一组（描述符、数据区）属性对组成。每一个描述符都说明了紧随其后的数据区的属性。描述符的形式主要有两种，区别仅在于域所包含的位的不同。图 8-12 显示了普通的描述符格式。它指定了后面数据项的类型、长度和个数（一个数据域有可能包含多个类型相同的项目）。现有的类型包括位和字节、不同长度的整数、非结构化机器字、布尔型、浮点型、字符串性以及权能。通过使用这些信息，系统就可以在这些具有不同表达方式的不同机器之间进行转换。这种转换不是由内核进行，而是由网络消息服务器完成的（下面将进行介绍）。它甚至还可以完成简单消息的节点间传输（同样也是由网络消息服务器完成的）。

数据域中所能够容纳的更有趣的一个项目是权能。通过使用复杂消息，它就能够将权能进行拷贝或者从一个进程传送到另一个进程。由于在 Mash 中权能是受保护的内核对象，所以需要有一个保护机制来保护它们的移动。

这种机制主要如下。一个描述符可以指定在消息中紧随其后的一个字中包含着发送者的一个权能的名字，这个权能被传递给接收进程并将插入到接收进程的权能列表中。这个描述符同时还指出该权能是被拷贝（原始权能保持不动）还是被移动（原始权能被删除）。

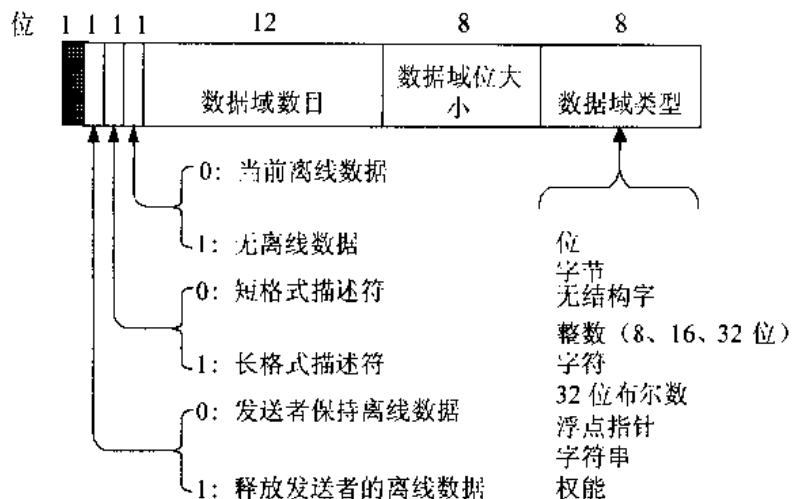


图 8-12 复杂消息的域描述符

数据域类型 (*Data file tyre*) 中的某个值可以更进一步要求内核在拷贝或移动过程中改变权能的权限。例如一个 **RECEIVE** 权能可以被改变为 **SEND** 或者 **SEND-ONCE** 权能，这样接收者就可以向发送者具有 **RECEIVE** 权能的端口中发送应答。实际上，两个进程建立通信的常规做法就是其中一个建立端口，然后将该端口的 **RECEIVE** 权能发送给另外一个，同时在传送过程中将其转为 **SEND** 权能。

为了进一步说明权能是如何进行传递的，我们可以考虑如图 8-13 所示的情况。在图中我们可以看到分别具有三个权能和一个权能的进程 *A* 和 *B*。所有的权能都是 **RECEIVE** 权能。由于条目 0 主要用于空端口，所有计数都是从 1 开始。*A* 中的一个线程给 *B* 发送一个具有权能 3 的消息。

当消息到达之后，内核会检查该消息的消息头并发现它是一个复杂消息。然后它 will 开始按顺序处理消息体中的描述符。在这个例子中只有一个权能描述符，同时还有将它转换成为 **SEND** (或者也可能是 **SEND-ONCE**) 权能的指令。内核会在接收者的权能列表里申请一个空位，在本例中为 *slot2*，同时将消息中紧跟描述符的字改为 2 而不是 3。当接收者获得消息之后，它会发现它拥有一个名字 (索引) 为 2 的新权能。它然后就可以立即使用这个权能 (如发送应答消息)。

图 8-12 中我们还没有讨论的最后一个方面是 **out-of-line data** (离线数据)。Mach 中提供了一种不用进行任何拷贝而实现大量数据从发送者向接收者进行传送的方法 (在同一机器或者多处理机中)。如果描述符中的 **out-of-line data** 被置位，那么紧随该描述符后的字中包含了一个地址，而描述符的大小以及数字域则给出了一个 20 位的计数器。这些信息一起指定了发送者的一段虚拟地址空间。对于更大的区间则可以使用描述符的长模式。

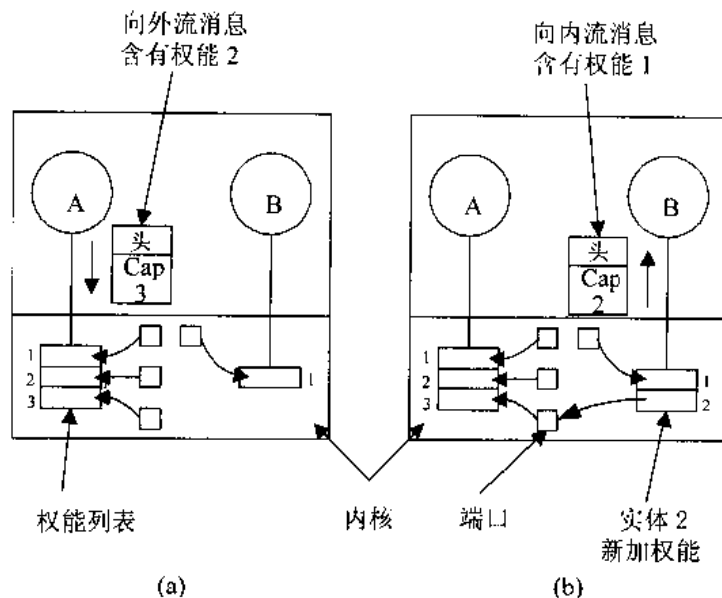


图 8-13 (a) 权能发送前的情况
(b) 权能到达后的情况

当消息到达接收者时，内核会申请一段与 out-of-line data 同样大小的虚拟内存，然后将发送者的页面映射在这段空间上，将它们标注为写拷贝。紧随描述符后的字将会被改变为指向接收者的地址空间的区域。这种机制可以实现大量数据的高速移动，因为除了消息头和两个消息体字（描述符和地址）之外不需要进行任何拷贝。根据描述符中一个位的设置不同，该区域或者从发送者的地址空间中删除，或者继续保留。

虽然这种方法在同一机器（或者多处理机的不同 CPU 之间）的不同进程之间的拷贝具有很高的效率，但是在网络上不同进程之间的拷贝方面就不是这么有效了，这主要是因为如果页面被使用（甚至只是读操作），那么就必须对它们进行拷贝。因而在逻辑上而无须在物理上进行数据传送的权能就不复存在。写拷贝同时也要求消息定位在页面边界上，同时是一个整数页面以获得最佳效果。部分页面会使接收者在 out-of-line data 前后看到它所不应该看到的数据。

8.4.3 网络消息服务器

到目前为止我们有关 Mach 中的通信都是局限于单个节点内通信的，这个节点或者是一个 CPU 或者是一个多处理机节点。网络上的通信是由一个称为网络消息服务器（network message server）的用户层服务器实现的，它类似于我们前面讨论过的外部存储管理器。Mach 分布式系统中的每一台机器，运行一个网络消息服务器。许多网络消息服务器协同工作以处理不同机器之间的信息传递，同时使其尽量与同一机器内部的消息传递相一致。

一个网络消息服务器可以是一个执行多种功能的多线程进程。这些功能包括同本地线程的接口、网络上的消息转递、具有不同表现方式的机器之间的数据类型转换、权能的安全管理、远程通知(Rmotify)的实现、网络上简单名字查找服务以及其他网络消息服务器认证的处理。网络消息服务器可以根据所连接网络的不同而使用不同的协议。

图 8-14 中显示了通过网络传递消息的基本方法。这里我们在机器 A 上有一个客户，在机器 B 上有一个服务器。在客户同服务器进行通信之前，在机器 A 上必须建立一个端口以作为服务器的代理。网络消息服务器具有该端口的 RECEIVE 权能，同时服务器中一个线程不断监听这个端口（以及所有其他以端口集形式出现的远程端口）。这个端口在图中以 A 内核中的一个方框（box）代表。

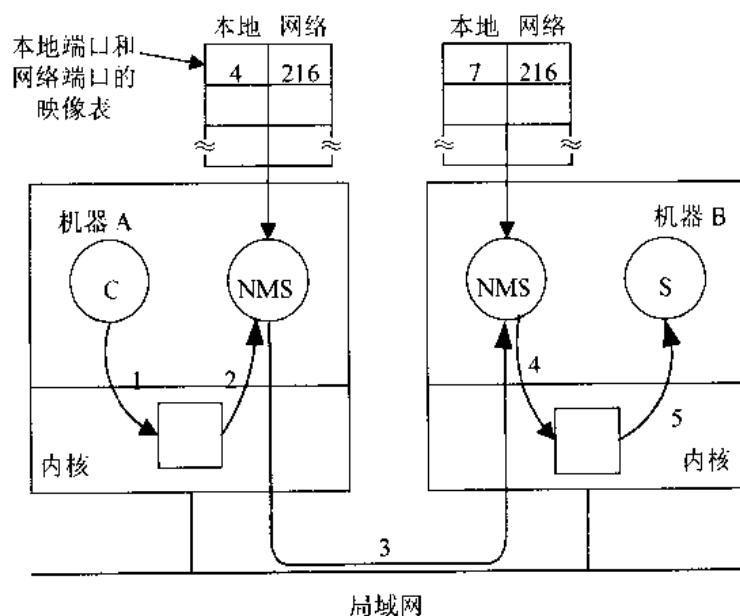


图 8-14 Mach 中的机器间通信需要五个步骤

如图 8-14 所示，从客户到服务器的消息传递需要五个步骤，分别标记为 1 到 5。首先，客户给服务器的代理端口发送一个消息。然后网络消息服务器会获得这个消息。由于这个消息是严格的本地消息，所以就可以使用 out-of-line data，写拷贝也可以正常工作。第三，网络消息服务器在将代理端口映射成为网络端口（network ports）的表中检查本地端口，在本例中为 4。一旦知道了网络端口，网络消息服务器就可以在另一个表中查找它的位置。随后它会建立一个包含本地消息和所有 out-of-line data 的网络消息，通过 LAN 将这个�息发送到服务器所在的机器上。在一些情况下，网络消息服务器之间的通信必须加密以确保安全。传送模块负责将它们分割成消息包，然后将它们按合适的协议打包加密。

当远程网络消息服务器获得这个消息之后，检查消息中所包含的网络端口号并将该端口号映射成为本地端口号。在步骤 4 中，它将消息写入刚查找到的端口中。最后，服务器从本地端口中读取消息并按要求进行执行。应答则沿相同路线反方向进行。

复杂消息则需要更复杂一些的工作。对于普通数据域来说，位于服务器上的网络消息服务器必须在必要的情况下进行转换工作，如两个机器的字节排序顺序不同的情况。同时还必须对权能进行处理。当一个权能通过网络进行传递时，必须分配一个网络端口号，而且发送者和接收者都必须在它们的映射表中添加这个网络端口的条目。如果这些机器之间不存在相互信任关系，那么还需要一个详细的认证过程来确信相互之间的身份。

虽然从一个机器到另外一个机器之间的消息传递通过用户层服务转接的思想提供了一定的灵活性，但是系统同纯内核消息传递实现相比较在性能上付出了潜在代价，而其他

大多数分布式系统的消息传递都采用纯内核消息传递机制。为了解决这个问题，开发了一个运行在内核中的新通信包（NORMA 模式）来获得更高的通信速度。最终它将替代网络消息服务器。

8.5 Mach 的 UNIX 仿真

Mach 上运行着许多不同的服务器。其中最重要的可能就是包含着大量 Berkeley UNIX（如必须的全部文件系统代码）代码的程序。这个服务器是主 UNIX 仿真器（Golub 等，1990）。这种设计是 Mach 从 Berkeley UNIX 的历史版本改进中的遗留部分。

如图 8-15 所示，Mach 系统中的 UNIX 仿真器实现主要由两部分组成，UNIX 服务器和系统调用仿真库。当一个系统开始运行时，UNIX 服务器指示内核获取所有系统调用陷阱并将它们指向执行系统调用进程的仿真库。从这开始，UNIX 进程所发出的任何系统调用都将归结为控制暂时转向内核，然后立即转向它的仿真库。当控制转到仿真库时，机器的所有寄存器都被赋以陷阱发生时的值。这种通过内核反射回用户空间的方法有时称为蹦床机制（trampoline mechanism）。

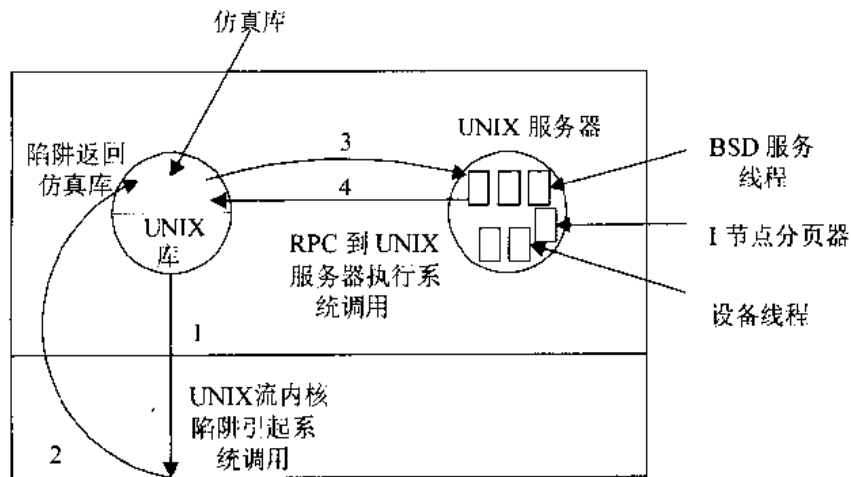


图 8-15 Mach 使用蹦床机制的 UNIX 仿真

一旦仿真库获得控制权，它将检查寄存器以决定应该触发哪个系统调用。然后它 will 向其他进程、UNIX 服务器发送一个 RPC 来完成所需要的工作。当这个过程结束之后，用户程序将再一次获得控制权。这种控制权的转移不需要通过内核。

当 *init* 进程产生子进程时，它们将自动继承仿真库和蹦床机制，所以它们也可以执行 UNIX 系统调用。EXEC 系统调用也被修改为只改变 UNIX 程序的地址空间部分，而不改变仿真库。

UNIX 服务器的实现方式是一组 C 线程。虽然其中一些线程用于处理时钟、网络以及其他 I/O 设备，但是大多数都用于处理 BSD 系统调用，执行 UNIX 进程仿真器所提出的要求。仿真库通过普通的 Mach 进程间通信机制同这些线程进行通信。

当一个消息到达 UNIX 服务器时，一个空闲线程会接受这个消息，判断发送该消息

的进程，从消息中获得需要的系统调用号以及参数，执行它后返回应答。大多数消息都只对应一个 BSD 系统调用。

工作方式比较特殊的一组系统调用是 I/O 调用。它们虽然可以以相同的方式进行实现，但是出于性能的考虑采用了不同的实现方法。当一个文件被打开之后，它直接映射到调用者的地址空间中，因而仿真库也就可以直接获取它，而不用向 UNIX 服务器执行 RPC 调用。例如，为了实现一个 READ 调用，仿真库会在需要进行读操作的文件中找到相应的位，然后定位用户缓存，最后以尽可能快的方式将前者拷贝到后者。

在拷贝过程中如果需要拷贝的页面不在内存中则产生页面出错。每一个失效都会引发 Mach 给保存映射文件的外部存储管理器发送一个消息。这个存储管理器是 UNIX 服务器中一个称为 I-节点分页器 (I-node pager) 的线程。它从磁盘上获取文件页并将它们映射到用户的地址空间中。它也可以同步不同 UNIX 进程同时打开的文件上的操作。

虽然这种运行 UNIX 程序的方法看起来很笨拙，但是测试结果表明它可以同传统的单内核实现相抗衡 (Golub 等, 1990)。今后的工作将集中于将 UNIX 服务器分解为更加专用的多服务器。虽然这在很大程度上取决于多服务器的发展情况，但是最终单 UNIX 服务器一定会消失。

8.6 小结

Mach 是一种基于微内核的操作系统。它被设计为建立新操作系统以及仿真现有操作系统的基础。它也提供了一种灵活的将 UNIX 扩展为多处理机和分布式系统的途径。

Mach 基于进程、线程、端口以及消息等概念。一个 Mach 进程是一个地址空间以及一组运行于其上的线程。活动实体是线程，而进程只是容纳它们的容器。每一个进程和线程都拥有可以通过写消息来让内核执行的端口，从而避免了直接系统调用的需要。

Mach 拥有一个精心设计的虚拟存储系统，它以可以被映射到地址空间、从地址空间中解除映射和由外部的用户级存储服务器进行备份的存储对象为特征。例如，文件可以通过这种方式直接被设定为可以读或写。存储对象可以包括写拷贝 (copy-on-write) 在内的许多方式进行共享。而继承属性则决定了地址空间的哪一部分将被传递给它的孩子。

Mach 中的通信基于端口，它是承载信息的内核对象。所有的消息都被导向端口。端口通过使用存储在内核中的权能进行访问，这些权能通过一般为权能列表索引的 32 位整数所确定。

BSD UNIX 仿真是通过存在于每一个 UNIX 进程地址空间的仿真库实现的。它的工作是捕获所有系统调用并将它们反映到内核中，然后再传递给真正执行它们的 UNIX 服务器。有一些调用在进程的地址空间中进行本地处理。其他 UNIX 仿真器也处于开发阶段。

Amoeba 和 Mach 有许多共同点，但是也有许多不同之处。它们都有进程和线程，都基于消息传递。Amoeba 有 Mach 没有的可靠广播原语，但是 Mach 也有 Amoeba 所没有的请求分页。从总体上来说，Amoeba 更倾向于使一组分布的机器像一台单机一样工作，而 Mach 则倾向于多处理机的有效利用。这两个系统都在不断发展，而且必将随着时间的发展而改变。

习 题

- (1) 有两个线程的进程和各有一个线程的两个进程共享同一个地址空间（也就是说同一套页面集）。指出它们的一项不同。
- (2) 对你自己的进程调用 *Join* 过程会发生什么？
- (3) 一个 Mach 线程生成两个新线程 *A* 和 *B* 作为它的子线程。*A* 线程调用了一个 *detach* 过程；而 *B* 没有。两个线程都退出了，父线程调用 *join* 过程。这时将发生什么？
- (4) 图 8-4 所示的全局运行队列在被搜索前必须先锁定。局部运行队列是否也这样？说明理由。
- (5) 每个全局运行队列都有一个互斥体来锁定它。假设一个专门的多处理机有一个全局时钟，它能同时使所有的 CPU 产生时钟中断。这会对 Mach 调度器产生什么影响？
- (6) Mach 支持处理机集概念。这个概念多用在什么类型的机器上？用来做什么？
- (7) Mach 为虚存空间的区提供了三个继承属性。哪些在正确做 UNIX FORK 系统调用时是必需的？
- (8) 一个小进程的所有的页都在内存中，并有足够的空闲内存来对它做十个以上的拷贝。它执行 *fork* 生成了一个子进程。这个子进程能得到一个页或防止错误吗？
- (9) 为什么你认为有一个调用能拷贝虚存的一个区域（见表 8-3）？毕竟所有的线程只能通过安置在一个紧密的拷贝环中来拷贝它。
- (10) 为什么页面置换算法在内核中运行而不在外存管理器中运行？
- (11) 举例证明线程在它的虚存地址空间里释放一个对象是值得的。
- (12) 两个进程能同时拥有对同一个端口的 *RECEIVE* 权能吗？*SEND* 权能呢？
- (13) 进程能否知道它正在读的端口实际上是一个端口集？这有关系吗？
- (14) Mach 支持两种类型的消息：简单的和复杂的。实际上是否需要复杂消息，或这只是一个优化形式？
- (15) 现在来回答一下有 *SEND-ONCE* 权能和离线 (*out-of-line*) 消息的问题。即它们对正确实现 Mach 的功能是否必要？
- (16) 在图 8-10 中同一个端口在不同的进程中有不同的名字。这可能会造成什么问题？
- (17) Mach 有一个系统调用允许进程通过获得一个特殊的句柄来请求非 Mach 陷阱，而不是使进程被杀死。这个系统调用的好处是什么？

第9章 实例研究 3: Chorus

第三个实例是 Chorus。它是一个现代的、基于微内核的操作系统。本章的结构和前两章类似：首先是简单的历史回顾，然后对微内核作一概述，接下来对进程管理、内存管理和通信进行详细的讨论。在这之后，将研究 Chorus 是如何实现 UNIX 仿真的。接下来的一节讨论的是 Chorus 中分布式面向对象编程的有关内容。最后将对 Amoeba、Mach 和 Chorus 作一个简单的比较作为本章的结束。关于 Chorus 更多的信息可以在 (Abrossimov 等, 1989, 1992; Armand 和 Dean, 1992; Batlivala, 等, 1992; Bricker 等, 1991; Gien 和 Grob, 1992; 和 Rozier 等, 1988) 中找到。

9.1 Chorus 简介

这一节将总结 Chorus 这些年来发展情况，简要地讨论它的设计目标，然后对它的微内核和两个子系统进行技术介绍。在后面小节里将对内核和子系统作更详细的描述。Chorus 的文献用的是一种非标准的术语，在这一章将采用标准化的术语，同时在圆括弧中给出 Chorus 术语。

9.1.1 Chorus 的发展史

Chorus 在 1980 年作为分布式系统的研究项目，诞生在法国 INRIA 学会中，它已经推出了四个版本：版本 0 ~ 版本 4。版本 0 的想法是建立一个分布式应用的模型，这个模型是一个包含多个“actors”（本质上是进程结构，译成“活动”）的集合，每一个“活动”都在执行原子事务和执行通信之间切换。从效果上看，每一个“活动”都是一个微观上的有限状态自动机。系统中的每一个机器都运行着相同的内核，这个内核管理着活动、通信、文件和 I/O 设备。版本 0 是用解释型 UCSD Pascal 编写的，它的运行环境是由环行网连接的 8086 机组。它在 1982 年中期得到应用。

版本 1，存在于 1982 年到 1984 年，集中于多处理机研究上。它是为法国 SM90 多处理机机器而做的，SM90 由 8 个 Motorola 68020 CPU 组成，这 8 个 CPU 连接在一条公共总线上，其中一个运行 UNIX，其他 7 个运行 Chorus，用运行 UNIX 的 CPU 来进行系统服务和管理输入输出。多台 SM90 机器由以太网连接。它的软件和版本 0 相似，只是增加了一些结构化的消息和对容错性的支持。版本 1 是由编译型（而不是解释型）的 Pascal 语言来编写，曾试用于大约 12 个大学和公司的机器上。

版本 2(1984~1986)是系统的一次重要的重新编写，是用 C 语言编写的。它被设计为一个在系统调用的源代码水平上与 UNIX 兼容的系统，这也就意味着现存的 UNIX 程序在 Chorus 中只要重新编译就可运行。版本 2 的内核被进行彻底的重新设计，它将尽可能多的功能从内核中移出到了用户那里，这样就将内核变成了微内核。UNIX 的仿真由多个进程完成，它们分别负责控制进程管理、文件管理和设备管理。对分布式应用的支持也有所

增加, 包括远程处理、分布式命名及定位 (location) 协议。

版本 3 开始于 1987 年。这个版本标志着 Chorus 从一个研究性的系统变成了一个商业性的产品, 这也是因为 Chorus 的设计者 (为了 Chorus 的发展及市场化) 离开了 INRIA 而自组了公司 Chorus Systems。在版本 3 中, 有好几个技术上的变化, 包括更为精干的微内核及它与系统其他部分的关系, “活动”模型及它的原子事务的最后痕迹也消失了。RPC (远程过程调用) 作为一个常规通信模式被推出, 内核方式进程开始出现。

为了使 Chorus 成为一个通用的商业性产品, UNIX 的仿真能力得到了增强, 增加了二进制代码级的相容性, 这样现存 UNIX 程序无须重新编译就可直接运行。UNIX 仿真机制的一部分 (在微内核中的那一部分) 被移植到了仿真子系统中, 同时它更趋于模块化。异常处理被改进为可以正确接收 UNIX 的异常信号。

这样, 它的性能得到进一步增强。系统的一部分用 C++ 语言进行了重写, 使它的可移植性得到了增强, 且能够在多种不同的体系结构下实现。版本 3 也吸收了其他分布式系统微内核的许多思想, 比较重要的有: 进程间的通信系统、虚拟存储设计以及从 Mach 学来的外部分页技术 (Pager) 和从 Amoeba 学来的对全局命名稀疏权限化和保护的使用。

9.1.2 Chorus 的设计目标

Chorus 项目的目标随着系统本身的发展, 也在不断地发展变化着。最初, 它是一个纯学术性的研究项目, 主要应用于探究基于 “活动” (进程) 模型的分布式计算的新思想, 随着时间的推移, 它变得更加商业化, 侧重点也发生了变化。目前的目标可被粗略地概括如下:

- (1) 高性能 UNIX 仿真;
- (2) 应用于分布式系统;
- (3) 实时应用;
- (4) 把面向对象编程集成到 Chorus 中。

作为一个商业系统, 大量的工作集中在跟踪不断发展的 UNIX 标准以及移植到装有新的 CPU 芯片的系统和提高性能上。公司想要使 Chorus 看起来像 AT&T UNIX 的一个变体, 可重复使用, 更容易维护, 且面向未来用户的需要。

第二个目标是分布式的需要。Chorus 目标是要允许 UNIX 程序在一个由网络连接的多机器环境下运行。为了支持分布式应用, 对编程模型进行了大量的扩充, 这其中如基于消息的通信, 很容易引入到现在的模型中, 其他的, 如线程的引入需要对现有的特征重新思考 (例如 UNIX 的信号处理)。

第三个目标是对实时应用的支持。这里所采用的方法是允许实时程序在内核方式下运行 (部分地), 可直接对微内核进行访问 (指用这种方式不需任何软件)。另外, 在实时应用时, 用户控制中断的方式和进程调度算法也是很重要的。

最后一个目标是通过一种简捷的方法、无需改动现有的子系统及其应用引入面向对象编程, 这个目标的实现将在这一章稍后描述。

9.1.3 Chorus 系统结构

Chorus 是层次结构的操作系统, 如图 9-1 所示。在底部是微内核 (在 Chorus 文献中

被称为 nucleus)，它提供对名称、进程、线程、存储和通信的最低程度管理。对这些服务进行访问要通过调用（和微内核直接相通）实现，共有 100 多个调用。更高一层的进程提供剩余的操作系统服务。基于 Chorus 的分布式系统中的每一个机器都运行着一个完全相同的 Chorus 微内核拷贝（copy）。

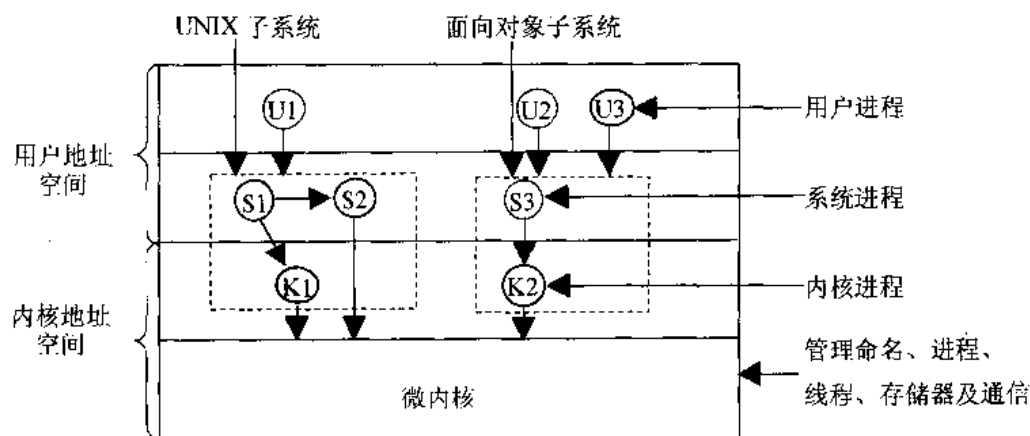


图 9-1 Chorus 的分层结构：微内核、子系统、用户进程

在微内核的上一层（也以内核方式操作）是内核进程。这些进程在系统运行过程中可被动态地加载和卸载，它们提供了一种在不增加微内核的规模和复杂性的情况下扩展其功能的方法。既然这些进程和微内核以及它们彼此之间分享了内核空间，它们在被加载以后必须重新定位（relocated）。另外，它们可以对微内核进行功能调用，同时彼此之间也可互相调用。

例如，中断处理程序是作为内核进程编写的。在一个有磁盘驱动器的机器上，当系统启动的时候，磁盘中断进程将被加载，当磁盘中断发生时，它们将通过这个进程而被截获，在没有硬盘的工作站上，由于不需要磁盘中断处理程序，它就不被加载。动态装载与卸载内核进程的能力使得通过设置系统软件来匹配硬件成为可能，无须再重新编译和连接微内核。

下一层包含系统进程。这些进程运行在用户态，但也可给内核进程传递消息（或彼此之间），还能对微内核进行调用，如图 9-1 所示的那样，内核进程和系统进程的结合体可以在一起工作，从而组成一个子系统。在图 9-1 中，进程 S1、S2 和 K1 组成了一个子系统，进程 S3 和 K2 组成第二个子系统。一个子系统存在着一个良好定义的面向用户的接口，就像 UNIX 系统调用接口那样。每个子系统有一个进程是管理者起控制子系统操作的作用。

在子系统的上一层是用户进程。例如，由用户进程 U1 所做的系统调用可能由 K1 来完成，并通过 S1 和 S2 处理，在需要的地方这些都可以依次用到内核服务。子系统使得在微内核的上层用标准方法建立新的（或旧的）操作系统成为可能，而且允许在同一个机器上在同一时间存在多个操作系统接口。

微内核知道子系统的每一个用户进程是否正在运行，而且保证每一个用户进程只能对子系统作系统调用。由用户进程直接对微内核作调用是不允许的，除非那些调用被子系统定义为合法的。实时进程像系统进程那样运行（而不是像用户进程那样），这样它就能

对微内核直接操作而不受干涉。

9.1.4 内核概念

内核（微内核）提供并管理七个关键的概念，它们在一起构成了 Chorus 的基础。这些概念是：进程、线程、区、消息、端口、端口组、唯一标识符，如图 9-2 所示。Chorus 进程（在 Chorus 的文献中仍称为 actors）和其他操作系统的进程完全一样，它们是可用资源的拥有者，一个进程拥有一份资源，但进程被取消时，资源也就被释放了。

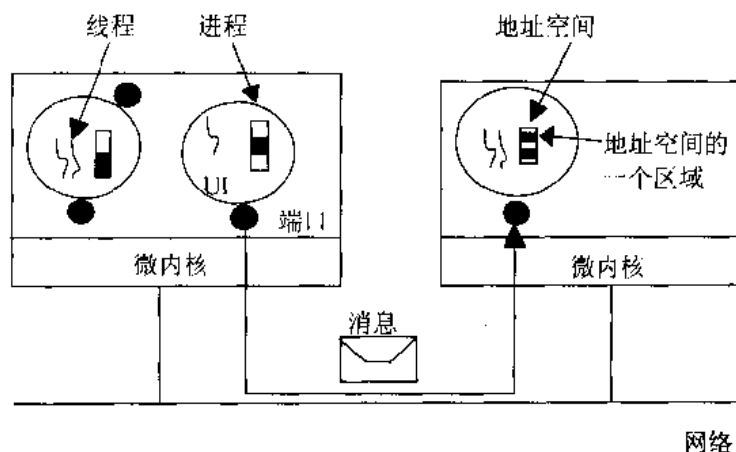


图 9-2 进程、线程、区域、消息以及由 UIs 标识的端口

在一个进程中存在一个或多个线程。在拥有自己的堆栈、堆栈指针、程序计数器和寄存器方面线程和进程很相似，但是，同一进程中的线程共用着进程范围地址空间和资源。从原则上说，一个进程中的线程彼此之间是独立的，在一个多处理机中，可能在不同的 CPU 上会同时运行多个线程。上面提到的三种进程都可以是多线程的。

每一个进程都有一块地址空间，一般从 0 到某一个最大地址，例如 $2^{32}-1$ 。一个进程中的所有线程都可对这个地址空间进行访问。一块连续的地址被称作区域（region），每一个区域由某种数据段组成，比如一个程序或一个文件。在支持虚拟存储和页式存储的系统中，区域是分页的，在 Chorus 中区域对存储管理起主要作用。

不同进程（可能在不同机器上）中的线程通过消息来通信。消息可以有一个固定的部分和一个大小可变的正文部分，这两个部分都是可选的。正文是非固定模式的，可能由发送者放入其中的任意信息组成。消息并不寄往线程，而是被寄往一个叫作端口（port）的媒介（中介）结构（Intermedia Structure）。一个端口是接受并保存从进程那里得到的还未读过的消息的缓冲区。和线程、存储区以及其他资源一样，每一个端口都属于某个进程，只有这个进程能读它里面的消息。端口可以被集中在一起成为端口组，这将在以后讨论。

最后一个内核概念与命名有关。大多数内核资源（比如进程、端口）都由一个 64 位的唯一标识符（UI）来命名。一旦 UI 被分配给一个资源时，即使一年后当资源被用在另一个机器上时，也将被保证该 UI 不会被另一个资源再次使用。这种唯一性由 UI 中的如下编码保证：这个编码由 UI 被生成时的站点（机器或多处理机）以及时间序号（epoch number）再加上这个时间序号的有效计数组成，系统的每一次引导时间序号也要作加工

修正。

唯一标识符是一个二进制的数，它本身是不被保护的，进程可以通过消息将 UI 传递给其他进程或者将它存储到文件中。当一个唯一标识符通过网络传输时，接收者努力去访问对应的对象，UI 中的位置信息被用于指明这个对象的位置。

因为 UI 地址太长，使用起来开销太大，在一个进程内部用局部标识符（LIs）来标识资源，这同在 UNIX 中用小整数代替文件描述符来识别打开的文件很相似。

内核概念并不是 Chorus 所使用的唯一的東西。由内核和子系统共同管理的其他三个概念也是重要的，它们是权能、保护标识符和段。权能是由子系统所管理的一种资源名，有时它由内核管理。它包括一个 64 位的属于某个子系统的端口唯一标识符，一个 64 位的由子系统分配的关键词。如图 9-3 所示。

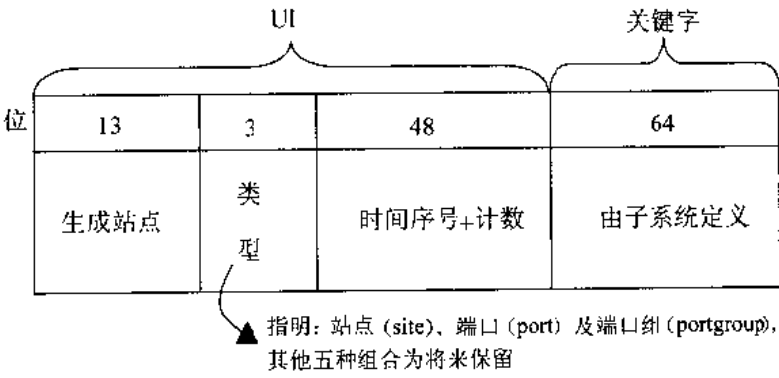


图 9-3 Chorus 的权能

当一个子系统生成一个对象，比如生成一个文件时，它返回给调用者一个这个对象的权能。通过这个权能，这个进程或者接下来的其他得到了这个权能的进程可以找到对应端口的 UI，从而向这个端口发送消息以请求操作对象。这个 64 位的关键词必须被包括在这样的信息中，该信息用来告诉子系统：它属于子系统的许多对象中的哪个对象。这 64 位信息包含了一个指向子系统表的索引，而子系统表是用来标识对象的。其他位是用随机数产生的以使猜出有效的“权能”变得困难。和 UI 一样，权能通过消息和文件传送。这个命名方案是从 Amoeba 那里学来的 (Tanenbaum 等，1986)。

Chorus 中的存储管理建立在两个概念的基础上，即前面已经讲过的区域和段。段是通过权能来标识的由字节组成的线性序列，当一个段被映射到一个区域上时，这个段中的字节对于这个区域内的进程的线程是可用的，只要它们在这个区域里进行读/写操作。程序、文件和其他形式的数据在 Chorus 中以段的形式存储，可以被映射到区域里。还可以通过系统调用对段进行读和写（甚至当一个段还没有被映射到区域上的时候）。关于地址空间的一个例子如图 9-4 所示。

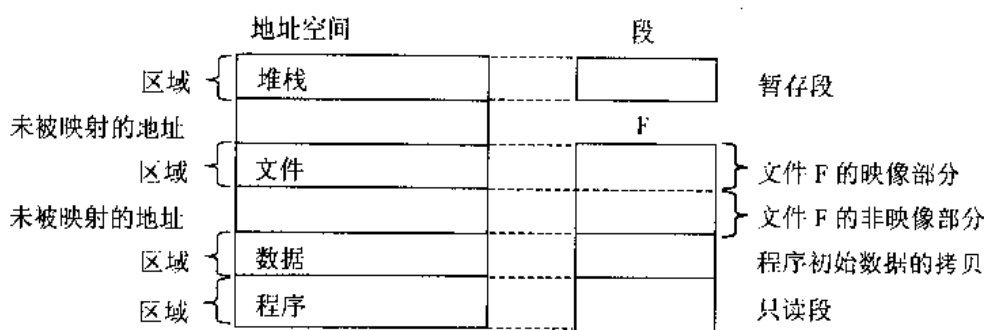


图 9-4 有四个被映射的区域的地址空间

9.1.5 内核结构

在对 Chorus 内核所提供的主要概念进行描述之后，我们来简要地看看内核内部是怎样的结构。内核包括四个部分，如图 9-5 所示。底部是监控程序（Supervisor），它管理着裸硬件，俘获陷阱、异常、中断以及其他的硬件细节，并且处理现场交换（context switching）。它有一部分是用汇编语言写的，所以当 Chorus 内核被移植到新的硬件上时不得被重写。

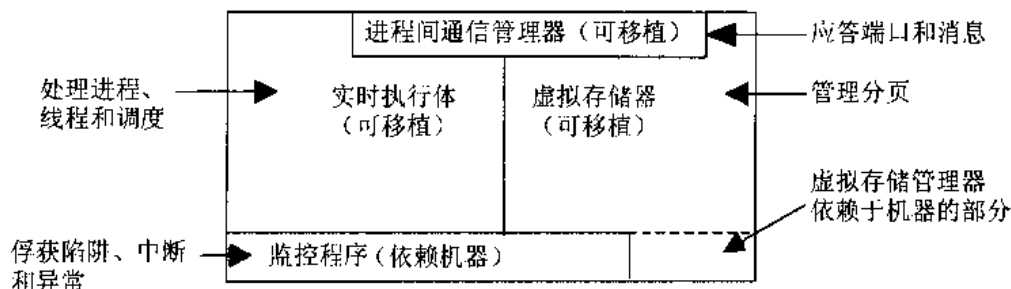


图 9-5 Chorus 内核的结构

下一个要讨论的是虚拟存储管理器，它对最低层的页系统进行处理。它的最大的一部分处理分页缓冲管理和其他逻辑概念，它是独立于机器的。但是它还有一小部分必需要知道怎样访问 MMU 寄存器，这一部分是依赖于机器的。在把 Chorus 移植到一个新的机器上时必须修正。

虚拟存储管理器的这两部分合在一起依然不能做完分页式管理系统的所有工作。第三部分，也就是内核外的映射器，将完成高层部分的工作。在虚拟存储管理器和映射之间的通信协议是一个良好定义了的协议，用户可以编写自己专用的映射器。

内核的第三部分是实时执行体，它对进程、线程和调度的管理作响应，它还解决发生冲突的（或因为其他目的）线程之间的同步问题。

最后，还有进程间通信管理器，它操作 UIs、端口以及用一种透明的方式发送消息。它使用实时执行体和虚拟内存管理所提供的服务来做它自己的工作。它是完全可以移植的，当 Chorus 被移植到一个新平台时完全不需改变。内核的这四个部分是用模块化的方式构造的，所以其中之一发生改变一般不会对其他部分产生影响。

9.1.6 UNIX 子系统

既然 Chorus 现在是一个商业化的产品，它就必须给出客户们想要的东西。而客户们想要的（至少在高端工作站的范围中）是和 UNIX 之间的兼容性。Chorus 通过提供一个标准子系统来达到这个目标，这个标准子系统叫 MiX，和 UNIX 的 System V 版本兼容，这种兼容性是在二进制代码（也就是说，由于有着相同的体系结构，在真正的 UNIX 系统中编译过的可执行程序无须修改即可在 Chorus 上运行）和源代码（也就是说，UNIX 源程序能够在 Chorus 上编译后运行）两个层次上的。MiX 的一个较早的版本（3.2）同 4.2 BSD 兼容，但是这一章将讨论一个更新的版本。

MiX 与 UNIX 在许多方面兼容，例如文件系统兼容，这样 Chorus 可以读 UNIX 的磁盘，而且，Chorus 和 UNIX 设备驱动程序有着兼容的接口，这样如果有某些 UNIX 的设备驱动程序，几乎不需要做什么相关的工作就可以应用到 Chorus 上。

MiX 子系统比 UNIX 系统更加模块化。它由四个进程构成，一个是进程管理，一个是文件管理，一个是设备管理，一个用来流（Stream）和进程间的通信。这些进程不共享任何变量和存储器，只通过远程过程调用来通信。本章稍后，将详细描述它们是怎样进行工作的。

9.1.7 面向对象子系统

作为一项研究性的实验，第二个子系统已经实现了，它是为面向对象编程服务的。它由三层组成，最底层用一种类的方法来进行对象管理，它是面向对象系统的一个高效的微内核，中间一层提供一个通用运行时间（runtime）系统，最顶层是语言运行时间系统。这个子系统叫作 COOL，也将在本章稍后进行讨论。

9.2 Chorus 中的进程管理

在这一节中我们将讨论 Chorus 进程和线程是如何进行工作的，如何处理异常以及如何完成调用。并将通过对一些主要的进程管理简短的描述来给出全部要用的内核调用。

9.2.1 进程

一个 Chorus 的进程是一些主动的和被动的元素集合，它们一起完成某个计算任务。主动元素是线程，被动元素是地址空间（包括区域）和端口集（用来接收消息和发送消息）。只有一个线程的进程就像传统的 UNIX 进程。一个没有线程的进程不能做任何有用的事，而且当它被创建时只能存在一段很短的时间。

三种进程在特权数量和它们是否有信誉（trust）上有很大的不同，如表 9-1 所列的那样。特权是指运行 I/O 和其他保护指令的能力，信誉意味着进程被允许直接调用内核。

内核进程权限最大，它们在内核态下运行，且全部和内核及彼此之间共享地址空间。它们能够在执行时被加载和卸载，不仅如此，它们还可被看作是微内核的扩展。内核进程可以用一种专用的轻量 RPC（它对其他进程是不可用的）通信。

表 9-1 Chorus 中的三种进程

类型	信任度	特权	状态	空间
用户	非信誉	非特权	用户态	用户
系统	有信誉	非特权	用户态	用户
内核	有信誉	有特权	内核态	内核

每一个系统进程都在它自己的地址空间上运行。系统是非特权级的（即在用户态运行），这样它就不能直接操作 I/O 和其他保护指令。但是内核相信它们，允许它们做内核调用，这样系统进程就无须通过中介可直接获得系统服务。

用户进程是非信誉和非特权的。它们不能直接操作 I/O 设备，也不能调用内核，除非它们的子系统给了它们这样的权利。每一个用户进程都有两个部分：常规用户部分和陷入后被调用的系统部分。这种安排和 UNIX 的工作方式很相似。

每一个进程（和端口）都有一个相应的保护标识符（Protection identifier），如果进程有子进程，它的子进程将继承相同的保护标识符。这个标识符只是一个位串，不含任何可让内核能知道的内容。保护标识符提供了认证机制，例如，UNIX 子系统给每一个进程都分派了用户标识符 UID，可应用 Chorus 保护标识符来实现 UIDs。

9.2.2 线程

Chorus 中每一个主动的进程都有一个或多个执行代码的线程，每一个线程都有它专用的上下文（即栈、程序计数器和寄存器），当线程因为等待事件而阻塞时，它被存储起来，当线程恢复时它就还原。一个线程总是和生成它的进程捆绑在一起的，不能被移到其他进程中。

Chorus 线程对内核来说是可知的，由内核调度，所以生成和取消它都需要作内核调用。内核线程（相对于运行在用户空间且无内核知识的线程包）的一个优点就是当一个线程因等待事件（例如一个消息的到来）而停止时，内核可以调用其他线程，它还有一个优点是当用的是多处理机时可以使不同的线程同时在不同的 CPU 上运行。内核线程的缺点是需要一份额外的开销来管理它们，当然，用户依然可以自由地在单独的一个内核线程里实现用户态的线程包。

线程彼此之间通过收发消息来通信，即使线程在同一过程或在不同的机器上时也并不影响。在所有的情况下通信的含义都是一样的。如果两个线程在同一个进程里，它们也可以通过共享内存来通信，但是那样系统以后就不能重新设置运行不同进程中的线程了。

下面是线程一些不同状态的区分，但它们并不是互相排斥的：

- （1）活动状态（active）——线程在逻辑上可运行；
- （2）挂起状态（suspended）——线程被暂时挂起；
- （3）停止状态（stoped）——线程所在的进程被挂起；
- （4）等待状态（waiting）——线程正在等待某事件的发生。

一个处于活动状态的线程既可以是正在运行也可以是正在等待一个空闲的 CPU，在这两种情况下，逻辑上它都是没有阻碍而可以运行的。一个处于挂起状态的线程是被另一个线程（或它自己）进行内核调用请求内核挂起它的，类似的，当一个内核调用停止一个

进程时，这个进程中所有处于活动状态的线程都被置于停止状态直到这个进程被重新启动。最后，当一个线程执行一项不能立即实现的阻塞操作（blocking operation）时，这个线程就被置于等待状态，直到事件发生。

一个线程可以同时处于多个状态。例如，一个处于挂起状态的线程可能以后又成为停止状态，如果它的进程被挂起的话。从理论上说，每一个线程都有三个独立的位和它相联系，每一个分别对应挂起状态、停止状态和等待状态，只有当所有三位全为 0 时，线程才能运行。

线程在对应进程的“态”和空间内运行，换句话说，内核进程的线程在核心态下运行，用户进程的线程在用户态下运行。

内核提供两种线程能够使用的同步机制。第一种是传统的（计数）信号量，它有 UP（或 V）和 DOWN（或 P）两个操作，这些操作总是由内核调用来实现的，所以不便于应用；第二种机制是互斥体，它基本上是一个取值限制在 0 和 1 之间的信号量。互斥体是专为互斥服务的，它的优点是调用者不会因同步操作而阻塞，同步操作可以完全地在其自己的空间内实现，这样节省了内核调用造成的额外开销。

在每一个基于线程的系统里都有可能提出的问题是：如何对每一个线程的专有数据进行管理，比如它的堆栈。Chorus 通过给每个线程分配两个软件寄存器来解决这个问题。其中之一存储着当线程处于用户态时指向它的专有数据的指针，另一个是当线程被陷入内核中正在执行一个内核调用时，存储着指向它的专有数据的指针。这两个寄存器都是线程状态的一部分，当线程被停止或启动时跟随硬件寄存器存储或恢复。通过索引这些寄存器，一个线程能够按照约定访问对于同一进程中其他线程来说无法使用的数据。

9.2.3 调度

CPU 调度是通过使用基于线程的优先级来实现的。每一个进程都有一个优先级，每一个线程也都有一个它的进程内的相对优先级。一个线程的绝对优先级是它的进程的优先级和它自己的相对优先级的和。内核跟踪着每个处于活动态的线程的优先级，它总是使那个有着最高绝对优先级的线程运行。在一个有 k 个处理机的处理机上，将有 k 个最高优先级的线程运行。

但是，为了适合实时处理，一个附加的功能被添加到算法里。有一个特定的级，它将所有的优先级高于这一级的线程和低于这一级的线程分成了两类，处于高优先级的线程，比如 A 和 B（如图 9-6(a)）是非时间片的，当这样的线程开始运行时，它会一直运行直到它自愿释放 CPU（例如由于信号量被阻塞），或者一个更高级别的线程进入活动状态（因为 I/O 操作完成或其他事件发生的结果）。一般来说，它不会仅仅因为运行了很久而停止。

另一类线程，如图 9-6(b)所示，线程 C 将投入运行，当它运行完一个 CPU 时间片后被放到相应优先级线程队列的末尾，而线程 D 将要得到一个时间片。在没有 CPU 竞争机制的情况下，它们将不断地按时间片切换。

这种机制为大多数实时应用程序提供了足够的办法。系统调用能改变进程和线程的优先级，这样应用程序能告诉系统哪些程序是最重要的，哪些是不重要的。附加的调度算法能支持 System V 的实时处理和系统进程。

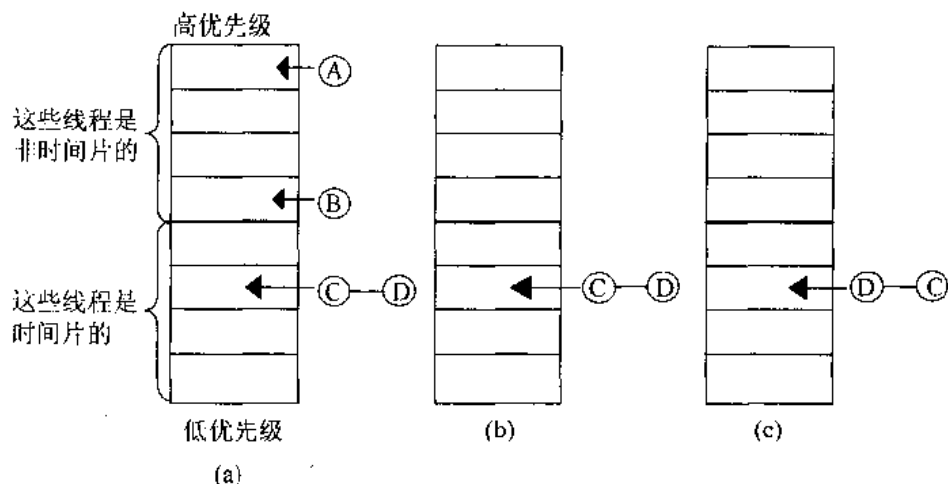


图 9-6 (a) 线程 A 一直运行到结束或被阻塞

(b)~(c) 线程 C 和 D 定时轮流交换执行

9.2.4 陷阱、异常和中断

Chorus 软件对这三种进入内核的方式作了区分。陷阱是为了请求服务对内核或子系统的一种有意的调用，程序通过调用一个系统调用库过程（库例程）来引发陷阱。系统支持两种管理陷阱的方法，第一种方法是：一个特殊陷阱向量的所有陷阱进入一个事先决定控制该向量的内核线程。第二种方法是：每一个陷阱向量都被捆绑在一个内核线程数组上，通过内核监控程序使用某一特定寄存器所存内容来在这一个内核线程数组中选出一个线程。后一种机制允许所有的系统调用使用相同的陷阱向量，利用寄存器所存系统调用号来选择一个处理程序（句柄：handler）

异常是由事故所引起的而非主动要求的事件。例如，除零，浮点溢出或者页错误。安排某一内核线程专用于管理异常是可能的，如果这个线程能完成这次处理，它将返回一个特殊代码，这次异常处理就结束了，否则（或者没有内核线程被分派），内核将把引起异常的线程挂起，同时送一个消息给这个线程所属进程的一个特定的异常处理端口。一般情况下，会有其他的某个线程在端口这里等待消息，它将给出某种进程所需的操作。如果没有异常端口存在，这个错误线程将被取消（杀死）。

中断是由于异步事件引起的，比如时钟点（clock tick）或 I/O 请求的完成。它们并非一定和某个当前线程正做的事有关，所以，让当前线程所属的进程来管理它们就不大可能。相反，当一个中断在某一中断向量上发生时（例如一个特殊设备），事先就安排好一个新线程来处理它是可能的。如果在第一个中断结束前指向同一中断向量的中断再次发生时，还可生成第二个线程，由此类推。所有除了时钟以外的 I/O 中断都可用这种方法来管理。时钟中断由监控程序自己来截获，但是如果需要它也可以被设置为向一个用户线程报告。中断只能请求一定范围内的内核服务，这是因为当它们被启动时系统处于一种未知状态。它们都能在信号量和互斥体上操作，还可寄一些微消息（minmessages）到微端口上（miniports）。

9.2.5 进程管理的内核调用

想要知道一个内核或操作系统实际上在干什么的最好的方法是去检查它的接口，所谓接口就是提供给用户的系统调用。在这一节，将介绍用于系统进程的最重要 Chorus 内核调用。不太重要的调用和仅用于内核线程的保护调用将忽略。

表 9-2 Chorus 内核所支持的部分进程调用

调用	说明
actorCreate	创建一个新进程
actorDelete	撤除一个进程
actorStop	停止一个进程，把它的线程置于停止状态
actorStart	从停止状态重新启动一个进程
actorpriority	读出或写入一个进程的优先级
actorExcept	获得建立一个用于异常处理的端口

下面开始讨论如表 9-2 所列出的进程调用。**actorCreate** 创建一个新进程并且将它的权能返回给调用者。这个新进程继承父进程的优先级、保护标识符和异常端口。参数说明这个新进程是用户，系统，还是内核进程，还说明它开始于什么状态。当它刚被创建时，这个新进程是空的，没有线程也没有区域，只有一个端口，即缺省端口。注意 **actorCreate** 在正确拼写上比 UNIX 有一个进步，“Create”在末尾加了一个“e”。

actorDelete 调用杀死一个进程。这个被杀死的进程是由一个作为参数传递的权能指明的。**ActorStop** 冻结一个进程，将它所属的线程都置于停止状态。只有当做 **actorStart** 调用时，这些线程才能再次运行。一个进程可以停止它自己。这类调用被典型地应用在调试上。例如，如果一个线程遇到了一个断点，调试程序将使用 **actorStop** 来停止这个进程的其他线程。

actorPriority 调用允许一个进程读另一个进程的优先级，而且，有时还可赋给它一个新值。虽然 Chorus 是位置无关的，但它并没有完全的做到这一点。有一些调用，包括 **actorpriority**，仅当目标进程在调用者的机器上时才是可行的，换句话说，去读取或重新设置异地进程的优先级是不可能的。

actorExcept 调用用于读取或改变调用者的异常端口，也可读取或改变调用者拥有权能的进程的异常端口。它还可以用于移去异常端口，这时当一个异常被送到端口所在进程该进程被杀死。

下一组内核调用是有关线程的，它们如表 9-3 所示。同样的，**threadCreate** 和 **threadDelete** 生成和关闭某个进程（不一定是调用者）中的线程。给 **threadCreate** 提供的参数指明特权水平、初始状态、优先级、入口和栈指针。

thraedSuspend 和 **threadResume** 挂起和重新启动目标进程中的线程。**threadPriority** 返回一个目标线程的当前相对优先级，如愿意还可以按照参数给它重新赋值。

我们的最后三个调用是管理线程的专用上下文的。**threadLoad** 和 **threadStore** 调用是分别读取和填入当前上下文寄存器的。这个寄存器指向线程的上下文，它的上下文包含有它的局部变量。**threadContext** 调用可有选择地复制线程原有上下文到缓冲区，也可有选择地

从另一缓冲区取出新的上下文。

表 9-3 一些 Chorus 内核支持的线程调用

调用	说明
threadCreate	生成某进程中的线程
threadDelete	关闭某进程中的线程
threadSuspend	挂起某进程中的线程
threadResume	重新启动被挂起的线程
threadPriority	读取和设置线程的优先级
threadLoiad	读取线程的上下文指针
threadStore	设置线程的上下文指针
threadContext	读取和设置线程运行的上下文

同步操作如表 9-4 所示。这些调用被用来初始化、请求及释放互斥体和信号量。它们都以普通方式工作。

表 9-4 一些 Chorus 内核支持的同步调用

调用	说明
mutexInit	初始化互斥体
mutexGet	请求一个互斥体
mutexRel	释放互斥体
semInit	初始化信号量
semP	在信号量上做 DOWN 操作
semV	在信号等上做 UP 操作

9.3 Chorus 的内存管理

Chorus 的内存管理从 Mach 中吸收了许多思想，当然，其中也有 Mach 所没有的思想。这一节将讨论它的基本概念和使用方法。

9.3.1 区域和段

Chorus 内存管理包含的主要概念是区域和段。“区域 (region)”是一段连续范围的虚拟地址，例如，1024 到 6143。从理论上来说，区域可以起始和结束于任何一个虚拟地址，但从实际角度来说，区域应该是页对齐的，应该有同整数页所拥有字节数相等的长度。一个区域中的所有字节一定有相同的保护特性（例如只读）。区域是进程的所有物，一个进程中所有的线程看到的是同样的区域，其中若一个进程有两个区域则它们是不能重叠的。

段是一个命名的字节的连续集合，它被一个权能保护。文件和磁盘上的交换区 (Swap areas) 是最普通的段。段可以用系统调用来读和写，这些系统调用提供段的权能、偏移地址、字节数、缓冲区和传输方向。这些调用被用作文件上的传统 I/O 操作。

但是，另一种可能性是映射段到区域上，如图 9-4 所示。一个段不必恰好和它所映射

的区域有相同的大小，如果这个段比区域大，就只有段的映射在区域上的那一部分才在地址空间上是可见的，选择哪一部分可见可以通过重新映射来实现。如果段比区域小，要读一个未映射地址则取决于映像程序，比如，它可以产生一个异常，或返回一个“0”或扩展段。

映射段通常是请求页式交换的（除非这个功能被屏蔽掉了，例如实时程序）。当一个进程首次调用一个新映射段时，一个页错误发生，于是被访问的地址相关的段页会被调入，这个错误指令也会被重新启动。用这个方法，一般虚拟内存就可以实现，而且一个进程可以在它的虚拟地址空间使一个或更多的文件可见，这样它就可以直接访问它们而不必用系统调用来读或写了。

内核支持一些特殊 I/O 段访问那些带有内存映射设备寄存器的机器上的 I/O 寄存器，使用这些段，内核进程可以直接通过读写内存来操作 I/O 设备。

9.3.2 映像程序 (Mapper)

Chorus 支持 Mach 风格的外部分页，它被叫作映像程序。每一个映像程序都控制着一个或几个被映射到区域上的段，一个段可以同时被映射到多个不同地址空间的区上，如图 9-7 所示。这里 S1 和 S2 都被映射到进程 A 和 B 上，A 和 B 在同一机器上，但地址不同。如果进程 A 在地址 A1 处写，那么 S1 的第一个字会被改变，如果进程 B 后来又在 B1 处读，它读到的将是 A 所写入的值。而且，如果 S1 是文件，当这两个进程都终止时，磁盘上的文件也会被修改。

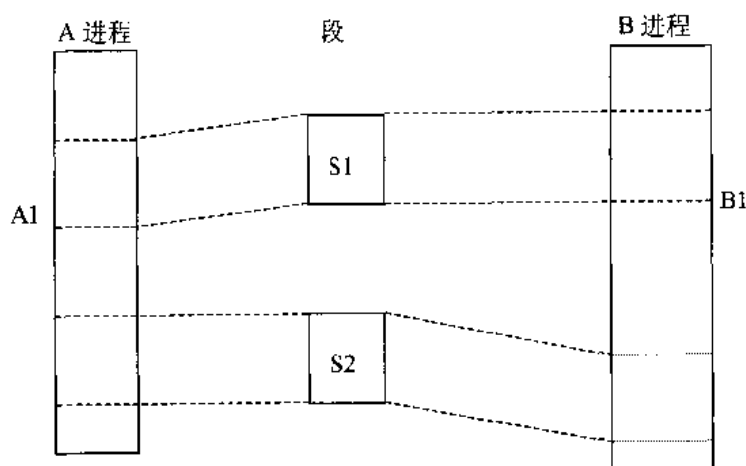


图 9-7 段可以被同时映射到多个地址空间

每个内核的虚拟内存管理程序都有一个页高速缓存 (page cache)，并且始终跟踪哪个页属于哪个段。本地高速缓存 (local cache) 里的页也可以属于可命名的段，例如文件，也可属于无名的段，例如交换区。内核还跟踪哪些页是未被写过的，哪些页是被写过的。内核可以按意愿取消未被写过的页，但是当要使用写过的页所占地址时，它必须先把它们映射到相关的映像程序中才能回收它的空间。

内核与映像程序间的协议决定页流 (flow of pages) 是两个方向的。当一个页错误发生时，内核首先检查被请求页是否在高速缓存里，如果不在，内核会发送一个消息给控制

这个页段的映像程序来请求这个页（有可能刚好是相邻的页）。这个错误线程会被挂起直到页被送过来。

当映像程序收到请求时，它检查被请求的页是否在它自己的缓冲区里（即它自己的地址空间里）如果不在，它将会发送一个消息给管理磁盘的线程做 I/O 操作以请求这一页。当这一页已调过来时（或它已在场），这个映像程序会通知内核，内核然后接受这一页，并调整 MMU 页表，然后重新启动出错线程。

映像程序也可以主动要求内核把写过的页交回给它。当内核交回它们后，映像程序可以把其中一部分放在自己的高速缓存里，将其他的写入磁盘。有几个调用供映像程序使用，允许它辨别它想要回的那些页。因为当空间满了的时候，内核可以自由地释放未被写过的页，还可以将写过的页送还给映像程序，所以大多数映像程序都无法跟踪它们的段拥多少个页帧。

作为映射段一部分的页和作为被 I/O 段读写的页一样，它们用的是相同的缓存和管理机制。这种相通性保证了如果一个进程通过在它上面写而修改了一个被映射的页，当随后另一个进程去读作为这个页的一部分的文件时，第二个进程会得到新数据，这是因为在内存中只有一个页的拷贝存在。

9.3.3 分布式共享存储器

Chorus 支持 IVY 形式的页式分布共享内存，就像第 6 章讨论过的那样。它用的是动态分散算法，也就是说，不同的管理器管理不同的页，每个页的管理器随着页（可写页）在系统中的运动而变动。

在多个机器上共享的单元是段。段被分裂成一或几页的片（fragments）。在任何时候，每个片要么在多个多处理机上是只读的，要么只在一个机器上是可读/写的。

9.3.4 内存管理的内核调用

Chorus 中的内存管理包括 26 个不同的系统调用和内核对映像程序的向上调用（upcall）。在这一节，将简短地描述其中一些重要的。要描述的调用是关于区域管理（有前缀 *rgn*）、段管理（有前缀 *sg*）及对映像程序的调用（有前缀 *Mp*，注意不是 *mp*，*mp* 用于小端口调用，以后会有论述的）。在这里我们不描述有关管理本地高速缓存（有前缀 *lc*）和虚拟内存（有前缀 *vm*）的调用。

表 9-5 给出了有关区域管理的调用。*rgnAllocate* 通过将某个区域的权能给予一个进程而标明了这个区域，权能包含起始地址、大小和多个选项。如果没有同别的区域冲突和其他问题，这个区域将被创建。选择包括初始化区域，即把字节全置为 0，设置它的读/写可执行位，固定它使其不能被页交换。*RgnFree* 归还一个以前被分配了的区域，这样地址空间中它所占的部分就成为空闲，从而可供其他段或区域的分配使用。

rgnInit 和 *rgnAllocate* 很相似，只是在分配区域后用一个段（它的权能是调用的参数）来填充它。还有几个同 *rgnInit* 相似的调用，它们用不同的方式填充区域。

剩下的三个调用用不同的方法改变一个区域的属性。*rgnSetInherit* 同一个区域今后被拷贝的可能性有关，标明这拷贝是获得自己的页还是共享原来的区域的页。*RgnSetPaging* 主要用于那些兴趣在于提供实时响应而不能允许页错误或者被交换出去的进程。当没有足

够内存时如果进程很想分配一个被捆绑的不能页交换的区域，它也会告诉进程该怎样做。*rgnSetProtect* 改变区域的读/写/可执行位，并可使区域设定成只对内核来说是可访问的。最后，*rgnStat* 告诉调用者区域的大小以及其他信息。

表 9-5 一些 Chorus 内核支持的管理区的调用

调用	说明
<i>rgnAllocate</i>	分配一个区域并设置它的属性
<i>rgnFree</i>	释放以前被分配了的区域
<i>rgnLnit</i>	分配一个区域并用一给定段填充它
<i>rgnSetInherit</i>	设置一个区域的继承属性
<i>rgnSetpaging</i>	设置一个区域的分页交换属性
<i>rgnSetProtect</i>	设置一个区域的保护选项
<i>rgnStat</i>	取得一个区域的有关典型统计量

段 I/O 调用如表 9-6 所示。*sgRead* 和 *sgWrite* 是允许进程读写段的基本 I/O 调用。段由一个描述符标识。段与缓冲区之间的复制进行时，偏移地址与字节数是必须要提供的。*sgStat* 允许调用者，典型的如映像程序，得到有关页高速缓存的统计信息。*sgFlush*（以及几个有关的调用）允许映像程序请求内核发送给它们页，这样这些页就可以被缓冲到映像程序的地址空间，或者写回它们在磁盘上的段。这个机制是需要的，比如，可保证一组集体支持分布式共享内存的映像程序能够从所有机器而不是一个机器中去掉可写页。这组中的其他调用与内存中的锁定和未锁定的个人页有关，给出了页系统的不同尺寸和其他信息。

表 9-6 一些和段有关的调用

调用	说明
<i>sgRead</i>	从段中读数据
<i>sgWrite</i>	向段里写数据
<i>sgStat</i>	请求某页高速缓存的消息
<i>sgFlush</i>	映像程序向内核请求写过的页

表 9-7 的调用是一些对映像程序的调用，即可以是内核发出的也可以是应用程序发出的，它要求映像程序为它们做某些事情。第一个，*MpCreate* 用于内核或者程序想要交换出一个段而需要分配硬盘空间给它，映像程序响应后，即在磁盘上分配一个新的段并且返回它一个权能标识。

表 9-7 映像程序调用

调用	说明
<i>MpCreate</i>	请求创建一个空段来交换
<i>MpRelease</i>	请求释放一个以前建立的段
<i>MpPullin</i>	请求一页或多页
<i>MpPushOut</i>	请求映像程序接受一页或多页

MpRelease 调用返回一个用 *MpCreate* 创建的段。*MpPullIn* 用于内核从一个新被创建并存在的页中取出数据。映像程序需要发出一个包含所需页的消息来响应它。由于使用一个精巧的 MMU 编程, 所需页不用物理拷贝。

MpPushOut 调用是用于另一传输方向(从内核到映像程序)的, 它既可以是对一个 *sgFlush* (或类似) 调用的响应, 也可以是内核想要交换出一个段到自己上面。虽然上面调用说明的表并不完整, 但它确实给出了 Chorus 中的内存管理是如何工作的概况图。

9.4 Chorus 中的通信

在 Chorus 中最基本的通信范例就是消息传送。早在第一版使用的期间里, 当研究集中在多处理机上时, 就已经考虑过运用共享内存作为通信的基本方法了, 但对此的反应是它还不太适用。在这一节中, 将讨论消息、端口以及通信操作, 包括对用于通信的内核调用的总结。

9.4.1 消息

每一条消息包括一个头部(只被微内核内部使用), 一个可选的固定部分和一个可选的正文部分。头部表明了来源与目标并包括几个保护标识符和标志位。固定部分使用时, 通常为 64 字节, 并且完全由用户控制; 正文有不同的大小, 最大为 64K 字节, 并且也完全在用户控制下。从内核的观点看, 在内核并不关心其固定部分和正文之内究竟有什么的情况下, 它们都是无类型字节数组。

当一个消息被传递到一个不同的机器的线程时, 它通常被拷贝下来。然而, 当它被送到相同机器的线程时, 那么, 在实际拷贝它和仅仅是将它映射到接收者的地址空间就有一个选择。在后一种情况下, 如果接收者在一个映射页上做写操作的话, 那么在现场就有一个真正的拷贝形成(也就是 Mach 的写-复制机理)。当一个消息表示的大小不是在整数页上时, 并且它又被映射, 一些超出缓冲器的(或在其之前的)数据就会在最后(或最前)页被映射时丢失。

消息的另一种形式是微消息(minimessage), 仅仅用于内核进程之间的短的同步消息, 特别是用在标明一个中断发生时。微消息被送往特殊的低开销小端口(miniports)处。

9.4.2 端口

消息被寄往端口, 每个端口都有一个能存一定数目消息的存储器。如果一个消息被送往一个已满了的端口, 寄出者将被挂起直到有空间可用。当一个端口被创建时, 一个唯一标识符和一个局部标识符都返回给调用者。前者还可被发往其他进程, 从而给这个端口发送消息, 后者用于创建它的进程在内部直接访问它。只有拥有这个端口的进程的线程可以读端口内的内容(端口可以迁移)。

当一个进程被创建时, 它自动获得一个缺省的端口, 内核通常向这个端口发送异常消息。它也可以加建它所需要的端口。这些后加上的端口可移到其他进程那里(缺省端口不行), 甚至是异地进程。当一个端口被移动时, 它上面所有消息会被一起移动。端口转移是很有用的, 例如, 当某个机器需要维修, 而它上面的一个服务程序想要求其他机器的

服务程序接替它的工作时。用这个办法，服务可以以一种透明的方式保留，尤其在服务机下线以后又上线时。

Chorus 提供了一种把几个端口集合为一个端口组的方法。为了做到这一点，进程先创建一个空的端口组，并为其分配一个权能。用这个权能可以向这个组里加端口，以后它还可以用这个权能从组里取消端口。一个端口可在多个端口组中存在，如图 9-8 所示。

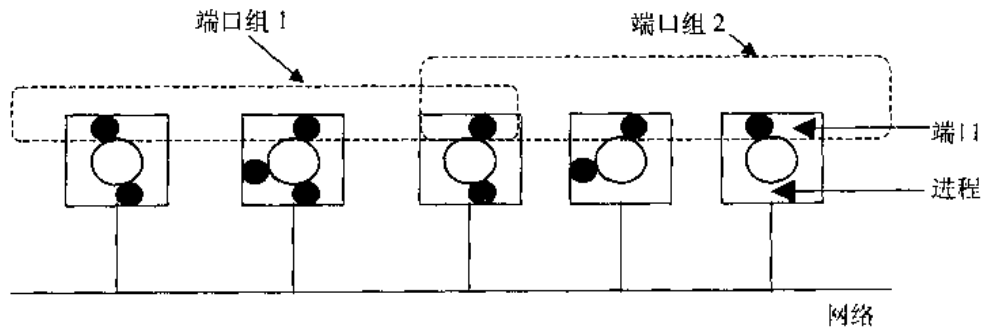


图 9-8 一个端口可能是多个端口组的成员

端口组通常用于提供可重新设置的服务。开始某个服务程序集属于一个组，这个组提供某种服务，客户可以给这个组发消息，而无须知道哪些服务程序来做这个工作。服务程序可以加入这个组，而且可以去掉，这个过程不会干扰服务，客户也不知道系统已经被重新设置了。

9.4.3 通信操作

Chorus 提供两种通信操作：异步传输和 RPC。异步传输让一个线程简单地发送消息到端口。这里对消息能否到达没有保证，如果出了问题也没有提示。这是一种数据报（datagram）传输的最单纯的形式，允许用户在其上层建立任意的通信模型。另一种通信操作是 RPC，当一个进程做 RPC 操作时，它会被自动阻塞直到有回复到来或 RPC 计时结束，这时发送者将不再阻塞。解除阻塞的消息是对发送进程请求的响应。任何不产生 RPC 事务标识的消息都将存在端口里以备将来使用，这种消息不唤醒发送进程。

RPC（远程过程调用）使用至多一次语义学，也就是说，当一次通信未完成或进程失败时，系统保证 RPC 会返回一个错误代码，但并不提供机会再做一次操作。

将消息发给端口组也是有可能的。有好几个可选择方式，如图 9-9 所示，这些选项决定有多少个消息要发送和被发送到哪些端口。

图 9-9（a）中是将消息寄往组里的所有端口。为了高可靠的存储，进程可能想把这个数据存储在每个文件服务器中。图 9-9（b）是将消息寄往一个端口，但是让系统自己选择寄往哪一个，当进程只想要某一个服务时，比如当前日期，但从不在乎它从哪里来，这个选择是最好的选择，因为系统自己能选择最有效的方式提供这个服务。

其他两个选择也是只寄往一个端口，但限制系统的选择。在图 9-9（c）中，调用者能标明指定位置的端口，例如，为了平衡系统负载而规定它的选择。而图 9-9（d）意思是任何不在指定位置上的端口都可以使用。使用这个选择，可能是为了强行命令一个文件向回拷贝到一个与原拷贝不同的机器上。

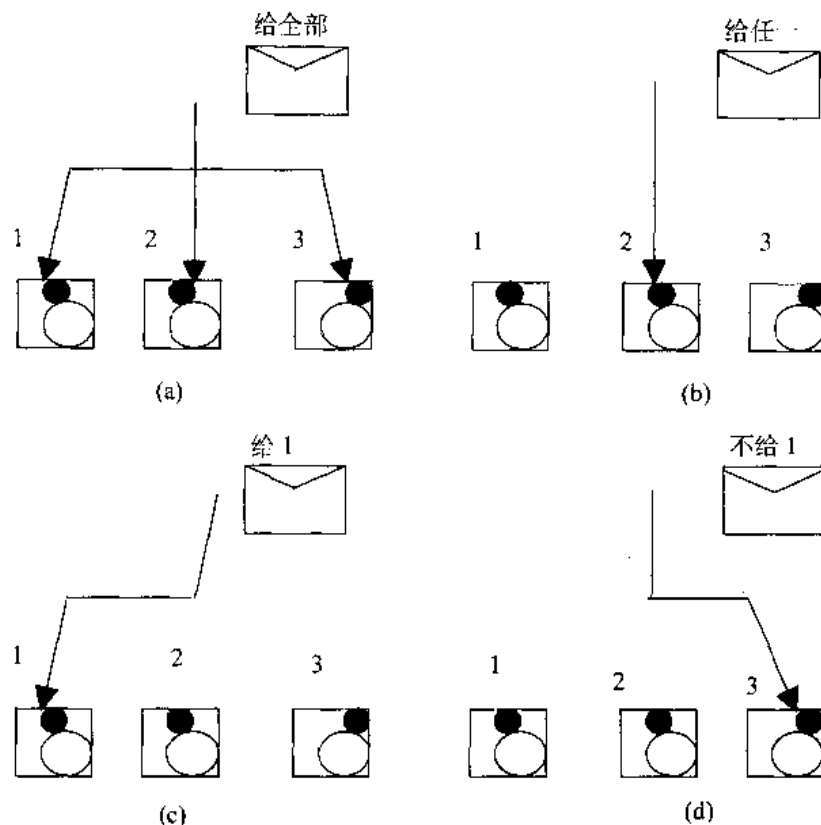


图 9-9 给端口组发送信息的选择

- (a) 给所有成员
- (b) 给任一成员(如给 2)
- (c) 给在同一位置指定了的端口
- (d) 给在非特殊位的端口

向端口组发送消息使用异步发送。广播发送（即对所有成员）不是流程控制的，如果需要流程控制，它必须由用户提供。

为了收到消息，线程要作一个内核调用，说明它想要哪个端口收消息。如果消息可用，它的固定部分会被拷贝到调用者的地址空间，而正文，无论哪一种，也会被拷贝或映射，这要看选项。如果没有消息可用，调用线程会被挂起直到消息到达或用户指定计时器到时。

而且，进程还可以指明它想要从它自己的端口中接收消息，但它并不在意是哪一个端口。这个选择还可通过使某些端口不可用而被精简，在这种情况下只有可用端口才可被选而满足请求。最后，端口还可被赋予优先级，这意味着如果不止一个可用端口有消息，那么选择具有最高优先级端口。端口可以动态设置为可用或不可用，而且它们的优先级也是可随意改变的。

9.4.4 通信的内核调用

端口管理调用如表 9-8 所示。前四个好理解，分别用来创建、取消、使其可用或不可

用。最后一个标明进程和端口。在这个调用完成后，端口不再属于它原先的所有者（不一定是调用者）而属于目标进程。只有它才能从这个端口读消息。

表 9-8 一些端口管理调用

调用	说明
portCreate	创建一个端口并送回它的权能
portDelete	取消一个端口
portEnable	使一个端口可用，这样它就可以从所有端口接收消息
portDisable	使端口不可用
portMigrate	移动端口到另一进程

有三个管理端口组的调用，它们在表 9-9 中列出。第一个，*grpallocate*，创建一个新组，并且把它的权能提供给调用者，用这个权能，这个调用进程或其他后来得到这个权能的进程能对这个组增加或撤除端口。

表 9-9 管理端口组调用

调用	说明
grpallocate	创建端口组
grpPortInsert	增加一个新端口给现存端口组
grpPortRemove	从端口组中撤除一个端口

最后一组内核调用是用来管理实际的投寄和接收消息的。这些在表 9-10 中列出。*ipcSend* 异步地投寄出一个消息给指定的端口或端口组。*ipcReceive* 阻塞直到从某一指定端口收到消息。消息可能被直接寄往指定端口，或端口组中的指定端口，或者所有可用端口（假定指定端口可用）。要被拷贝的固定部分的调用地址必须被提供，而正文的地址是可选项，因为它的大小事先可能不知道。如果正文没有被提供给缓冲区，*ipcGetData* 调用会被执行以从内核中获得正文（大小现在是可知的，因为它正好被 *ipcReceive* 调用收到了）。回应消息用 *ipcReply* 来寄出。最后，*ipcCall* 是用来作远程调用的。

表 9-10 选择通信调用

调用	说明
ipcSend	异步投寄出一个信号
ipcReceived	阻塞直到消息到来
ipcGetData	获取当前消息正文
ipcReply	寄一个回执给当前消息
ipcCall	作远程过程调用

9.5 Chorus 中的 UNIX 仿真

与 UNIX 相像并非 Chorus 的初衷（它有完全不同的界面并且用 UCSD Pascal 写成）。

但是随着第 3 版的来临, UNIX 兼容性已成了 Chorus 的主要目标。Chorus 采取的办法是让内核成为操作系统的核心, 另外再加上一个子系统使已有的 UNIX 二进制程序不用修改就能在其上运行。这个子系统一部分由写 Chorus 的人写出新代码, 一部分由系统 V UNIX 代码本身构成。UNIX 代码由 UNIX 系统实验室授权。在本节将讲述 Chorus 的 UNIX 是什么样子, 如何实现。UNIX 子系统 MiX, 代表 Modular UNIX。

9.5.1 UNIX 进程的结构

UNIX 进程作为 Chorus 用户进程运行在 UNIX 子系统之上。UNIX 进程拥有 Chorus 进程的全部特点, 尽管已有 UNIX 程序不会用到这些特点。标准文本、数据、堆栈段由 Chorus 映射段实现。

总体上, Chorus 工作方式并没有对 UNIX 运行期系统和库隐藏 (但是对于程序是隐藏的)。例如, 当程序打开一个文件时, 库过程 (例程) *open* 激活子系统, 最终获得一个文件权能。系统将文件权能存于内部, 当用户调用 *read*, *write* 和其他打开文件的过程时就使用文件权能。

打开文件并不是唯一的映射到 Chorus 资源的 UNIX 概念。打开目录 (包括工作目录和根目录), 打开设备、管道、正在用的段在内部都是代表相应的 Chorus 资源权能。UNIX 子系统在它们上的操作都是将权能传给适当的服务器。为了保持对 UNIX 的二进制兼容性, 用户进程绝对不能看到权能。

9.5.2 对 UNIX 的扩展

Chorus 提供了许多扩展的 UNIX 进程以使分布式编程更容易。大多数都是可见的 Chorus 属性。例如, 利用 Chorus 线程包, UNIX 进程可以创建、撤消新线程。这些线程并发运行 (在多处处理机上, 实际上并行运行)。

由于 Chorus 有内核管理, 所以当一一个线程阻塞时, 例如一个系统调用, 内核可以在同一进程里调度运行另外一个线程。然而, 对于大多数系统调用, 在同一进程里的两个线程中通常是不可能同时执行的。因为处理系统调用的老 UNIX 代码是不可重入的。在大多数情况下, 在启动系统调用之前, 第二个线程被透明地挂起, 一直到第一个结束。

由于线程的加入, 就必须重新考虑如何处理信号。在原来的 UNIX 中, 对整个进程来说, 所有的信号处理都一样。在扩展的 UNIX 中取代它的是同步信号 (由线程动作引起的信号) 与指定线程联系, 而异步信号 (比如用户敲击 DEL 键) 在进程范围内联系。比如, ALARM 系统调用在指定的秒数后中断调用线程。别的线程则不受影响。

类似地, 一个线程也可获取浮点溢出和类似的异常, 而另一个忽视这些异常, 第三个线程则不捕获这些异常 (这意味着, 如果发生一个异常则被杀死)。

当然, 其他信号并不确定于一个线程。另一个进程发出的终止信号, 或者键盘发出的 SIGINT 或 SIGQUIT 被送给所有的线程, 每一个都能自由地捕获或忽视这个信号。如果没有一个线程捕获或忽视一个信号, 则整个进程被终止。

信号由进程本身处理。UNIX 子系统中的一个控制线程与每一个 UNIX 进程都相连。它用于监听异常端口。当一个信号被激发后, 这个平常停止的控制线程就被唤醒。然后, 这个线程获得送到控制端口的消息, 检查其内部表, 执行请求动作, 也就是中断适当的线

程。

对 UNIX 所作的第三处扩展是使其分布化。进程可以使用一个系统调用指示新进程不在本地机上创建，而是在指定的远程机器上创建。当一个新进程分叉时，它和父进程在同一台机器上开始，但是当新进程执行 EXEC 系统调用时，它就在远程机上开始。

用户进程用 UNIX 子系统可以像其他 Chorus 进程一样创建端口和端口组，收发消息。它们也能创建区并且把段映射到上面。一般情况下，所有的与进程管理，内存管理和进程间通信相关的 Chorus 特性对 UNIX 进程都是有效的。

9.5.3 Chorus 上 UNIX 的实现

UNIX 子系统的实现由四个主要部件构成：进程管理器、对象管理器、流管理器和进程间通信管理器，如图 9-10 所示。每一部分都有一个专门的仿真功能。进程管理器捕获系统调用执行进程管理。对象管理器处理文件系统调用和分页活动。流管理器处理 I/O。进程间通信管理器执行系统 V IPC。进程管理器是新代码。其他的大部分从 UNIX 本身移植过来，以最大限度减少设计者的工作并达到最大的兼容性。这四个管理器都能处理多任务，所以不管有多少用户登录，在给定站点 (site) 只能存在其中一个。

在最初的设计中，这四个进程要求要么运行在用户态要么运行在核心态。然而，由于加入了越来越多的特权代码，这越来越难实现。在实用中，为了获得可以接受的性能，一般都运行在核心态。

为了了解这些部件之间的关系，考察一下系统调用是如何执行的。在系统初始化期间进程管理器告诉内核，它想处理标准 AT&T UNIX 用于产生系统调用的陷阱数字（为了获得二进制兼容）。当其后一个 UNIX 进程通过内核陷阱来发布系统调用时，进程管理器中的一个线程就获取控制，如图 9-10 的第 (1) 步所示。这个线程开始工作，就好像它是那个调用线程一样。传统 UNIX 系统中的系统调用也与此类似。基于系统调用，进程管理器或许亲自执行系统调用请求，或者如图 9-10 所示，给对象管理器发一条消息，叫对象管理器去做。对于 I/O 调用，流管理器会被激活。对于 IPC 调用，则是通信管理器的事。

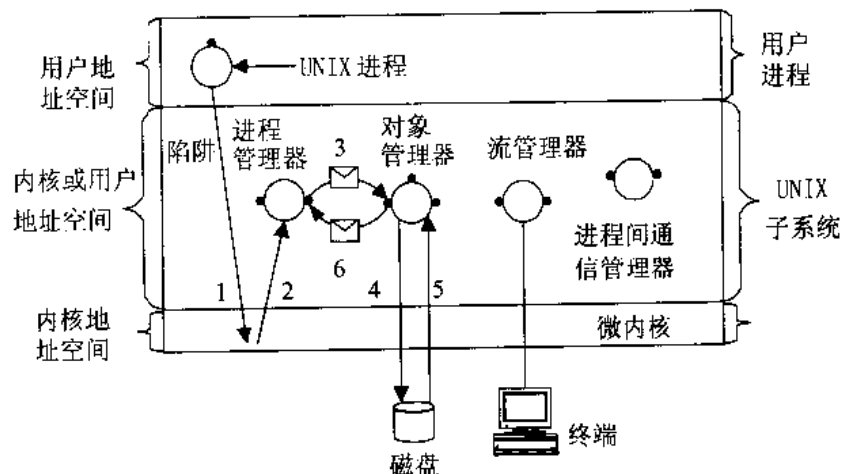


图 9-10 Chorus UNIX 子系统结构。数字显示了典型文件操作步骤序列

在这个例子中，对象管理器执行了一个磁盘操作，还给进程管理器发了一条回应消

息。这条消息携带适当的返回值并释放被阻塞的 UNIX 进程。

进程管理器 (PM)

进程管理器是仿真中的核心部件。它捕获所有的系统调用，并决定如何处理。它还处理进程管理（包括创建、终止进程），信号（生成信号、接收信号）和命名。当一个系统调用不能处理时，进程管理器就给对象管理器或者流管理器执行一个 RPC。也可以让核心调用来处理。例如，当一个新进程分叉时，内核就做大部分工作。

如果要在远程机上创建一个新进程，则需要一个复杂的机制，如图 9-11 所示。此时，进程管理器捕获一个系统调用，但是并没有让本地内核去创建一个 Chorus 新进程，而是如图 9-11 第 (3) 步所示，向目标机上的进程管理器执行一个 RPC。然后如第 (4) 步所示，远程进程管理器叫它的内核创建一个新进程。每一个进程管理器都有一个确定的端口，这个端口就是其他进程的 RPC 所指的端口。

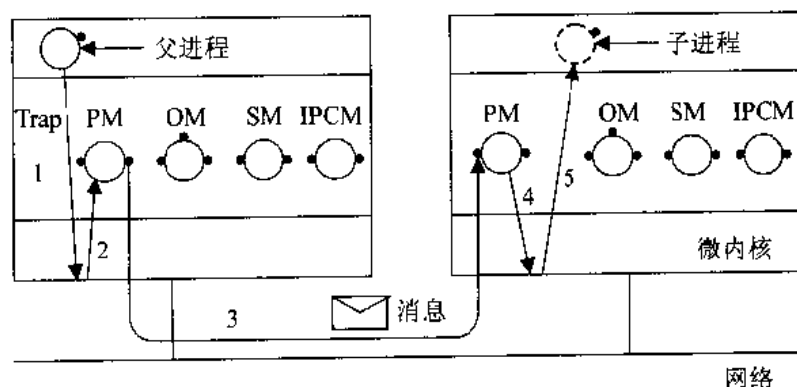


图 9-11 在远程机器上创建一个进程。答复消息没有显示

进程管理器有多个线程。例如，当一个用户线程设置一条警报时，进程管理器线程就进入休眠直到计时器到时间。然后它就向那个线程的进程异常端口发一条消息。

进程管理器也管理唯一标识符。它维护一张表，将 UIs 映射到相应的资源上。

对象管理器 (OM)

对象管理器处理文件、交换空间及其他形式的实际信息。它还可能包括磁盘驱动器。除了异常端口外，对象管理器还有一个接收分页请求的端口和一个接收本地或远程进程管理器请求的端口。当有请求来到时，就有一个线程去处理它。多个对象管理器线程可以同时活动。

对象管理器对于它所控制的文件就像一个映像程序，它在指定的端口接收页面错误请求，做必要的磁盘 I/O，并发送适当的回应消息。

对象管理器用以权能命名的段来工作，换句话说，就是以 Chorus 方式来工作。当一个 UNIX 进程调用一个文件描述符时，运行时系统就激活进程管理器。进程管理器以描述符作为索引在一张表中找到与权能相应文件的段。

逻辑上来说，当一个进程从一个文件中做读操作时，这个请求应被进程管理器捕获，然后用平常的 RPC 机制传给对象管理器。然而，由于读、写操作十分重要，因而要用优化策略来改善性能。进程管理器维护一张记录所有打开文件段的权能表。它产生一条 *sgRead* 调用给内核，来获得所需数据。如果数据有效，内核就直接将数据拷贝到用户的

缓冲区上。如果数据无效，内核就对适当的映像程序（通常是给对象管理器）产生一条 *MpPullIn* 调用。映像程序按所需发布一条或多条磁盘读操作。当页面有效时，映像程序就把页面给内核，内核把所需的数据拷贝到用户的缓冲区上完成系统调用。

流管理器（SM）

流管理器处理所有的系统 V 的流，包括键盘、显示、鼠标和磁带设备。在系统初始化期间，流管理器向对象管理器发送一条消息，宣布它的端口，并且通告它准备处理哪些设备。对这些设备的 I/O 请求就被送到流管理器。

流管理器还处理 Berkeley 套接字和连网（networking）。用 Chorus 进程可以用 TCP/IP 和其他网络协议进行通信。管道与命名管道也在这里处理。

进程间通信管理器（IPCM）

这个进程处理有关系统 V 消息（不是 Chorus 消息），系统 V 信号量（不是 Chorus 信号量）和系统 V 共享内存（不是 Chorus 共享内存）的系统调用。这些系统调用不好，代码绝大部分来自系统 V 本身。对这些讲得越少越好。

可配置性

UNIX 子系统负载分配使得一个机器集群以不同的方式进行配置相对简单一些。这样可使每一台机器只运行它所需的软件。多处理机的节点也可以有不同的配置。所有的机器都需要进程管理器，但其他的管理器是可选的。具体需要哪一个由实际应用决定。

下面简要地讨论几个用 Chorus 构造的不同配置。图 9-12（a）显示了一个全配置，这可用于连接在网络上的工作站（带有硬盘）。所有的 4 种 UNIX 子系统进程都需要并配齐了。

对象管理器只是在包含磁盘的机器上才需要。图 9-12（b）所示的配置对于无盘工作站是最合适的。当这台机器上的进程读写文件时，进程管理器把请求直接给了用户文件服务器上的对象管理器。原则上，无论是用户机器还是文件服务器都能捕获消息，但不能一起进行，除非是以只读方式打开或映射的段。

对于专门的应用，比如一台 X 终端，事先知道程序能用什么系统调用，不能用什么系统调用。如果不需要本地文件系统也不需要系统 V IPC，则对象管理器和进程间通信管理器可以删除，如图 9-12（c）所示。

最后，对于一个非连接的嵌入式系统，比如汽车、电视或会说话的玩具熊，则只需要进程管理器。其他的都可以删除以减少 ROM 大小。甚至特定的应用程序可以直接运行在微内核的顶部（即进程管理器也可以不要），如 9-12（d）所示。

配置可以动态地进行。当系统启动时，它会检测环境并决定需要哪些管理器。如果机器上有磁盘，则加载对象管理器；否则不加载。别的也以此类推。另外一种方法，则可以用配置文件。同样，每一个管理器由一条线程创建。有请求时就动态地生成另外的线程。表空间，例如进程表，动态地从池中分配。因此没有必要对小系统和大系统分别产生不同的内核代码（小系统只有很少的进程，而大系统则有数千个进程）。

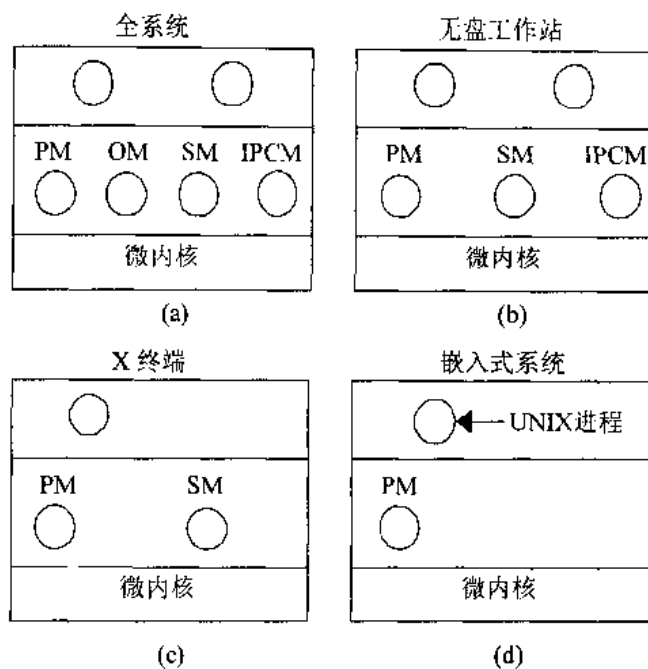


图 9-12 使用不同的配置不同的应用可以获得最佳处理

实时应用

无论有无 UNIX，Chorus 都可处理实时应用。Chorus 调度机构尤其反映这一特点。优先级从 1 到 255，同 UNIX 一样，数字越小，优先级越高。

如图 9-13 所示，原来的 UNIX 应用进程运行在 128 到 255 级上。在这些优先级上，当进程用完了它的 CPU 时间片时，它就被放到同级队列的末尾。组成 UNIX 子系统的子系统进程运行在 64 到 68 级上。这样，实时进程就有很多比 UNIX 子系统高或低的优先级可用。

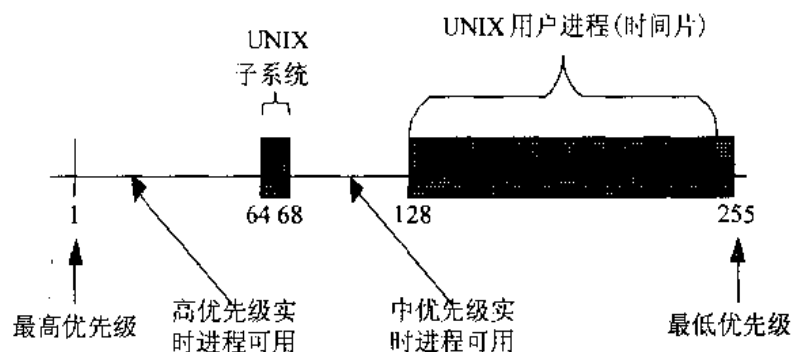


图 9-13 优先级和实时进程

另外一个对于实时工作非常重要的特性就是减少中断后 CPU 被禁止时间的能力。一般情况下，当一个中断发生时对象管理器或流管理器线程立即处理请求事务。然而另外一种方法是，中断线程让另一个低优先级的线程去做实际工作。这样中断就几乎可以立即终

止。这样做减少了中断后的限时，但是却增加了处理中断的开销。因此这种特性要小心使用。

这些特性与 UNIX 都结合得很好。所以就可以在 UNIX 下以通常的方法来调试一个应用程序，但在实际应用时，改变系统配置就可使其像实时进程一样运行。

9.6 COOL: 一个面向对象的子系统

UNIX 子系统只不过是标志成子系统的一些 Chorus 进程的集合。由此，也可以用另外一个子系统与其同时运行。Chorus 开发的第二个子系统就是 COOL (Chorus Object-Oriented Layer)。它是为研究面向对象程序而设计的，并且在粗粒度系统对象（例如文件）和细粒度语言对象（例如结构、记录）之间架起一座桥梁。本节将讲述 COOL。详细的资料请看 (Lea 等,1991,1993)。

第 1 版 COOL-1 开始于 1988 年。其目的是为细粒度面向对象的语言和应用提供系统级支持，并且要求新的 COOL 程序和老的 UNIX 程序能在同一台机器上一起运行而互不干扰。

COOL-1 中的所有对象都由两个 Chorus 段组成，一个是数据，一个是代码。程序并不直接存取数据段，而是激活位于对象代码段中的过程，该过程称为“方法”。这样，对象内部模型对于用户是隐藏的，从而使对象编辑者可以自由地使用看上去最好的模型（比如一个数组或连接表）。这些描述以后可以改变，而使用其对象的程序却不必要知道。为了节省内存，同一类的多个对象共享同一个代码段。

简而言之，这个系统在性能、资源使用等方面的表现使人失望。粗粒度系统对象和细粒度语言对象之间的桥梁也没能架起来。在 1990 年，设计者又开始设计 COOL-2。一年后该系统投入运行。以下讲述它的体系结构和实现。

9.6.1 COOL 的体系结构

从概念上说，COOL 提供一个可以跨越机器边界的 COOL 基层。这个 COOL 基层提供了一种地址空间形式，它可使所有的 COOL 进程无论在哪里都能运行。这就好像分布式文件系统提供一种全局文件空间一样。在基层之上是通用运行时系统，这也是整个系统范围内的。在通用运行时系统之上是语言运行期系统，再上是用户程序，如图 9-14 所示。

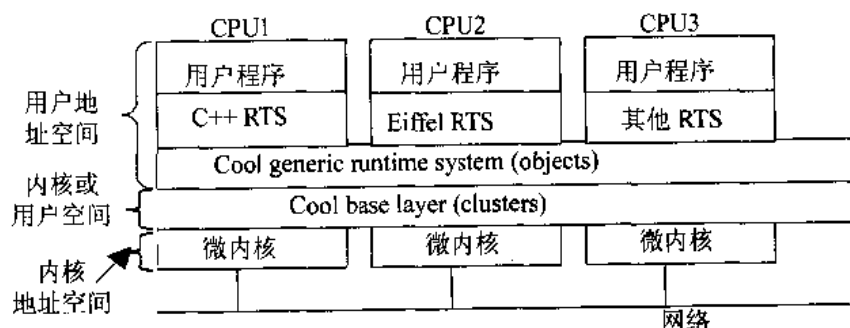


图 9-14 Cool 体系结构

9.6.2 COOL 基层

COOL 基层给 COOL 用户进程，尤其是与每一个 COOL 进程都相连的 COOL 通用库提供一套服务。最重要的服务是内存抽象，大体上就像分布式共享内存一样，但是更偏向于面向对象编程。这种抽象基于“簇”(cluster)，就是由段所支持的一组 Chours 区。每一个簇一般都有一组相关的对象，如属于同一类的对象。上层的软件决定哪一个对象进入哪一个簇。

一个簇可以映射到多个进程（也许在多台机器上）的地址空间。一个簇总是起始于一个页边界，并且拥有正在使用该簇当前所有进程的地址。簇中的区域不必在一个连续的虚拟地址空间内。比如，一个簇可拥有地址 1024~2047, 4096~8191 和 14336~16535（假定页面大小是 1K）。

COOL 基层支持的第二个概念是上下文空间，也就是可能在多台机器上的一些地址空间的集合。图 9-15 显示了一个系统有五个地址空间（在五台机器上）和两个上下文空间。第一个上下文空间跨越所有的机器，还包含了一个有三个区域的簇。在这些地址空间内的线程都能存取簇内的对象，就像它们在一个共享内存一样。第二个上下文空间只跨越三台机器，含有一个只有一个区域的簇。在地址空间 1 和 2 内的线程不能存取这个簇内的对象。在不同地址空间内的相应簇必须被映射至同一虚拟地址。在映射时，内部指针会重定位。

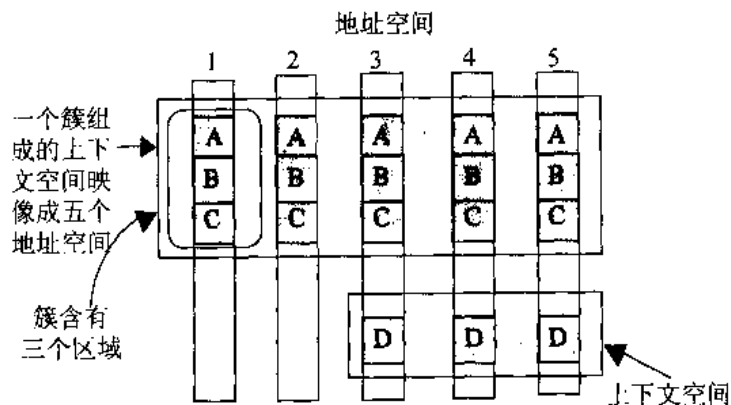


图 9-15 上下文空间和簇

簇不能复制。这意味着尽管一个簇可以同时出现在多个地址空间上，但实际上每个簇只有一份物理拷贝。当一个用户线程试图激活它的地址空间内的对象方法，而该对象的物理拷贝并不在调用线程的机器上，COOL 基层就会产生一个陷阱。然后基层要么用远程调用把请求传给实际拥有该簇的机器，要么把簇迁移到本地机上使用本地调用。

9.6.3 COOL 通用运行时系统

COOL 通用运行时系统使用簇和上下文空间来管理对象。在没有进程映射时，对象一直存在于磁盘上。语言运行时系统可以创建、删除对象，将对象映射进或映射出地址空间，还能激活对象方法。当一个对象激活位于同一簇内另一对象的方法时，使用本地过程调用。当被激活对象属于另一个簇时，通用运行时系统便请求 COOL 基层使用远程调用。由于

簇内调用的费用比簇间调用的费用小得多，所以将彼此激活非常多的对象放在一个簇里可以大大降低系统开销。在 COOL-1 中，所有的调用都是簇间操作，实际证明那样代价太高了。依赖于语言的运行时系统调用通用运行时系统，通用运行时系统给运行时系统提供了一个标准界面。这个界面基于向上调用。通用运行时系统使用向上调用来调用语言运行时系统去找对象属性。例如，当一个对象映射进地址空间时，为了重定位去找内部指针。

9.6.4 语言运行时系统

一般情况下，当程序员定义对象时，是在一个确定的界面下用确定的语言去定义。这些定义随后被编译成在运行时能够被调用的对象去调用其他对象。这些界面对象随后会决定对远程对象是进行远程调用还是将其迁到本地机上调用，界面对象中的方法可使程序控制此策略。

9.6.5 COOL 的实现

COOL 子系统像 UNIX 子系统一样，运行在 Chorus 微内核之上。如图 9-16 所示，COOL 子系统由一个实现 COOL 基层的进程和与每一个 COOL 程序(还有语言库)都相连的 COOL 通用运行时系统构成。这种设计使得 COOL 进程和 UNIX 在同时调用其子系统时不发生相互干扰。这种模块化是微内核设计的直接结果。当然，Amoeba 和 Mach 也是如此。

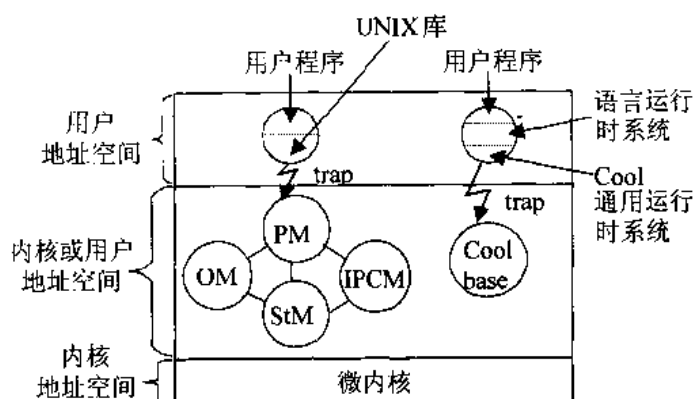


图 9-16 COOL 的实现

9.7 Amoeba、Mach 和 Chorus 的比较

在这一章和前两章中，对三种基于微内核的分布式操作系统：Amoeba、Mach 和 Chorus 作了相当详细的介绍。尽管它们有很多共同点，但是 Amoeba 和 Mach 在技术细节上有很多不同。Chorus 从 Amoeba 和 Mach 中借鉴了许多思想，所以它处于一种中间地位。在本节中我们将对这三种系统同时分析，以说明设计者可作的多种选择。

9.7.1 指导思想

Amoeba、Mach 和 Chorus 有不同的历史和不同的指导思想。Amoeba 从一开始就被设

计成一种通过 LAN 互连的 CPU 集合上使用的分布式系统。后来，又加进了多处理机和 WAN。Mach（实际上是 RIG）开始是一种面对单处理机的操作系统，后来才加进了多处理机和 LAN。Chorus 开始是作为一项分布式操作系统的研究工程，跟 UNIX 根本不沾边。但经过三次重要修订后，越来越靠近 UNIX。在这些系统中仍然能看到不同背景所留下的痕迹。

Amoeba 基于处理机池模式。用户并不在某一特定的机器上登录，而是将系统当作一个整体来登录。操作系统根据当前的负载来决定在哪里执行命令。因而使用多处理机也就很正常了。两个相继的命令很少在同一处理机上执行。这里也就没有主机的概念。

与此相反，Mach 和 Chorus（UNIX）是在这样一种指导思想下设计的：用户必须在确定的机器上登录，在缺省情况下，用户所有的程序都是在该机上运行。系统并不试图在尽可能多的机器上分配每个用户的工作（尽管在多处理机上，工作会被自动地分配给多处理机的所有 CPU）。尽管可以远程运行，但是指导思想上的不同之处在于，每个用户都得有一台主机（比如一个工作站），并且它的大多数工作都在主机上运行。然而，当 Mach 移植到 Intel Paragon（一种由处理机池构成的机器）之后，这种区别逐渐模糊了。

另一种指导思想上的差别在于，关于什么是微内核。Amoeba 遵从著名的法国飞行家和作家 Antoine de St. Exupery 的名言：“完美不是再没有什么东西可以加入，而是再没有什么东西可以去除”。当要向内核增加新特性时，决定性的问题是：我们能否去掉它？这种指导思想导致了一个最小的内核，并且大多数代码都运行在用户级服务器上。

与此相反，Mach 的设计者试图在内核中提供足够的功能来满足尽可能最大范围内的程序。在许多领域，Amoeba 只有一种方法，而 Mach 有两种或三种，每一种在不同的环境下都有其方便或高效之处。结果，Mach 的内核大得多，它所含的系统调用（包括对内核线程的调用）比 Amoeba 多四倍。Chorus 处于 Amoeba 和 Mach 之间，但它的系统调用仍然比 BSD UNIX 4.2（这几乎也是一个微内核）多。表 9-11 是各系统之间的比较。

Amoeba 和 Mach 还有一种指导思想上的差别。Amoeba 对远程事件（通过网络通信）进行优化，而 Mach 则对本地事件（通过内存通信）进行了优化。例如，Amoeba 在网络上有非常快的 RPC，而 Mach 的写拷贝机制则提供了单个节点上的高速通信。Chorus 主要强调单 CPU 和多处理机系统。但是由于通信管理在内核中处理（如同 Amoeba 一样），并不在用户进程中（Mach 方式），因此它实际上也有良好的 RPC 表现。

表 9-11 系统调用数目和选定系统中内核线程调用

系统	内核调用
Amoeba	30
Version 7UNIX	45
4.2 BSD	84
Chorus	112
Mach	153
SunOS	165

9.7.2 对象

对象是 Amoeba 中的核心概念。对象少部分是内建的，像一些线程和进程，但大部分是用户定义的（比如文件）并且拥有任意操作。大约十来个通用操作（比如获取状态）几乎对所有的对象都有，还有各种各样对象专用的操作。

相反，Mach 直接支持的对象只有线程、进程、端口和内存对象，每一种都有一套固定的操作。高级软件可以用这些概念来建其他的对象，但其本质上与内存对象等内建对象是不同的。

在三种系统中，对象都以权能来命名、寻址和保护。在 Amoeba 中，权能在用户空间内管理，并且用单向功能来保护。系统定义对象（如进程）的权能和用户定义对象（如目录）的权能都统一对待，并且出现在用户级目录上，用以对所有对象的命名和寻址。Amoeba 的权能在原则上是世界范围的，也就是说，一个目录可以拥有任何地方的文件和其他对象。对象通过广播来定位，其结果缓存起来以备将来之用。

Mach 也有权能，但只用于端口。内核用一张权能表来管理，每个进程一个权能。与 Amoeba 不同，Mach 中没有用于进程和其他系统、用户定义对象的权能，它们也不直接由应用程序使用。端口权能在进程之间以一种受控方式传递，所以 Mach 能通过查寻内核表来找到它们。

Chorus 也支持和 Mach 几乎相同的内建对象，但是也使用 Amoeba 的权能系统，以允许子系统定义新的受保护对象。与 Amoeba 不同，Chorus 的权能没有直接（密码保护）字段给予许可权限。与 Amoeba 相似，而与 Mach 不同的是，Chorus 的权能够从一个进程传到另一个进程，这仅仅简单地将它们写进一个共享文件或是包含在一条消息里就行。

9.7.3 进程

三种系统都支持每个进程里有多个线程，并且线程都由内核来管理、调度，虽然用户级线程包也可建于其上。Amoeba 不支持用户控制线程调度，而 Mach 和 Chorus 允许进程在软件里设置线程优先级和策略。Mach 比其他两种提供更完善的多进程支持。

在 Amoeba 和 Chorus 中，线程同步是通过互斥体和信号量实现的。在 Mach 中，是由互斥体和条件变量来做的。Amoeba 和 Mach 支持某种形式的全局变量。Chorus 通过软件登记来定义每个线程的私有上下文。

Amoeba、Mach 和 Chorus 都工作在多处理机上。但是如何处理这种机器上的线程，三种系统各有不同。在 Amoeba 中，一个进程的所有线程工作在同一个 CPU 上，由内核分时，并发操作。Mach 进程有细微控制，由它控制哪个线程运行在哪个 CPU 上（使用处理机集概念）。结果，同一个进程的线程可以并行地运行在不同的 CPU 上。

在 Amoeba 中，让几个单线程进程运行在不同 CPU 上，并且共享一个普通地址空间，也能有相似效果。然而，很显然，Mach 的设计者比 Amoeba 的设计者在多处理机上下了更多的工夫。

Chorus 支持同一进程的多个线程同时运行在不同 CPU 上，但是这整个由操作系统来处理。Chorus 中没有用户可见的管理线程到处理机分配的原语，但将来会有。

然而，Amoeba 的设计者在负载平衡和异构上做了更多工作。当 Amoeba 外壳启动一

个进程时，它要求服务器找到工作负载量最轻的 CPU。除非用户指定了特定的体系结构，否则，进程可在任何一个二进制程序能运行的体系结构上启动，用户甚至不用关心选择了哪一类体系结构。这种机制就是在任何时间里都将工作负载传到尽可能多的机器上。

在 Mach 中，进程一般在用户主机上启动。只有当用户特别要求时，进程才在空闲的工作站上远程运行。即使那样，当工作站的主人按键盘时，它们必须迅速撤回。这种差异关系到处理机模式和工作站模式的根本差别。

Chorus 允许进程在任何机器上启动。UNIX 仿真提供了一种方法来设置缺省站点。

9.7.4 内存模式

Amoeba 的内存模式基于长度可变的段上。一个虚拟地址空间由一些映射到指定地址的段构成。段可以随意地映射进或映射出。每一个段由一个权能控制。拥有其他进程段权能的远程进程（例如调试器）可以从另外任何一台 Amoeba 机器上读写某些段。Amoeba 不支持分页存储管理。在一个进程运行时，它的所有段都在内存中。这种决策的思想就是简化设计提高性能，与现在机器内存越来越大的现实相吻合。

Mach 的内存模式是基于内存对象并且用固定大小的页来实现。内存对象可以随意地映射或不映射。一个内存对象不必整个都放进主存里使用。当用到所缺的页时，会发生页面错误，并给外部的内存管理器发一条消息，使其找到所需页面并映射进来。这种机制与缺省内存管理器一起支持分页虚拟内存。

页面可以以各种方式在多个进程之间共享。一种普通配置就是用于联系子进程和父进程的写拷贝共享。尽管在单节点上，这种机制是一种高效的方式，但是在分布式系统中，它却失去了优势。因为它总是要求进行物理传送（假设接收者需要读数据），在这种环境下，额外的代码和复杂性造成了浪费。这是一个很清楚的例证，说明了 Mach 在单 CPU 和多处理机系统上所做的优化，而不是在分布式系统上。

Chorus 的内存模式主要取自 Mach。它也有可以映射的内存对象（段）。如同 Mach 一样，它们需要在外部分页程序（映像程序）的控制下分页。

Amoeba、Mach 和 Chorus 都支持分布式共享内存，但却以不同的方式实现。Amoeba 支持共享对象在所有使用它的机器上复制一份。对象可以是任意大小，可以支持任何操作。读在本地操作，写通过 Amoeba 可靠广播协议来完成。

相反，Mach 和 Chorus 支持基于分页的分布式共享内存。当一个线程用到一张不在本机上的页面时，该页面就从它当前所在的机器上迁入调用者机器上。如果两台机器大量存取同一个可写页面，则可能发生颠簸（系统失效）。与 Mach 和 Chorus 上潜在的颠簸相比，Amoeba 所作的折衷更昂贵一些（这归咎于可写页面的复制，而 Mach 和 Chorus 中可写页面只有一份拷贝）。

9.7.5 通信

Amoeba 支持 RPC 和组通信作为基本原语。RPC 被送往端口（服务地址）。通过单向函数可以进行加密保护。发送者和接收者可以在任何地方。RPC 的界面非常简单：只有三个系统调用，没有一个带可选的参数。

作为用户手段，组通信提供可靠的广播。消息被可靠地送往任何一个组。此外，所

有的组成员都能以绝对相同的顺序看到所有消息。

低层通信使用 FLIP 协议。这种协议提供进程定位（与机器定位相对）。这种特性允许进程迁移和在网内（或网间）自动重新配置，而软件不必知道。它还支持在分布式系统中很有用的其他特性。

与此相反的是，Mach 的通信是从进程到端口，而不是从进程到进程。而且，发送者和端口必须在同一节点。利用网络消息服务器或诺曼（NORMA）编码作为代理，通信可以扩展到整个网络。但这种间接方式牺牲了性能。Mach 不支持组通信或可靠广播作为基本的内核原语。

通信由使用 *mach_msg* 的系统调用来实现。这个系统调用有七个参数，十种选择和 35 种潜在错误消息。它支持同步和异步消息传送。这种方法与 Amoeba “保持简单性，使其更快” 的策略刚好相反。它的思想是为现在和将来的应用程序提供最大的弹性和最大范围的支持。

Mach 的消息既能是简单的，也能是复杂的。简单的消息只有几位，内核也不作专门处理。复杂的消息可能包含权能。它们能使用写拷贝来传送离线数据，这是 Amoeba 所没有的。另一方面，这种特性在分布式系统中没有什么价值。因为离线数据必须由网络消息服务器来存取，必须与消息头和在线数据合并才能在网上传输。当发送者和接收者在不同的机器上时，这种本地事务优化不会带来任何好处。

在网络上，Mach 使用传统的协议，比如 TCP/IP。这种方法有稳定性和广泛适应性的优点。相反，FLIP 比较新，对典型的 RPC 使用会更快，而且是专门为分布计算的需求而设计的。

Chorus 的通信指导思想与 Mach 相似，但更简单一些。消息直接送到端口，并且既可用异步也可用 RPC 来发送。像 Amoeba 一样，所有的 Chorus 通信都由内核处理。Chorus 中没有与网络消息服务器相似的东西。与 Amoeba 相似，而与 Mach 不同的是，Chorus 消息有一个固定的消息头，内含发送源和发送目的的信息；还有一个无类型的消息体，对系统来说，它只是一个字节数组。如同 Amoeba 一样，消息中的权能不作任何特殊处理。

Chorus 以一种类似 Mach，而不是 Amoeba 的方式来支持内核级的广播。Amoeba 支持内核广播，而 Mach 不支持内核广播。端口可组成端口组，消息送给端口组的所有成员（或者只送给一个）。但并不是像 Amoeba 那样，Chorus 不保证所有的进程以相同的顺序看到所有的消息。

9.7.6 服务器

Amoeba 对专门的功能提供各种服务器，包括文件管理、目录管理、对象复制和负载平衡。所有服务都基于对象和权能。Amoeba 通过含有权能集合的目录来支持对象复制。基于 POSIX，Amoeba 提供了源代码级的 UNIX 仿真，但不是 100% 完成。仿真通过将 UNIX 概念映射到 Amoeba 上，并调用本身的 Amoeba 服务器来完成。例如，在打开一个文件时，就要求有该文件的权能，并将其存于文件描述符表中。

Mach 只有一个服务器作为应用程序运行 BSD UNIX。它提供 100% 的二进制兼容仿真。这对于那些无源代码的文件是很好的。Mach 不支持通用对象复制，其他的服务器都有。

Chorus 提供完全的与 UNIX 系统 V 二进制兼容的能力。仿真用一群进程（像 Amoeba 一样），而不是将 UNIX 当作一个应用程序来完成（像 Mach）。然而，有些进程却像 Mach 一样包含实际的 UNIX 代码。像 Amoeba 一样，Chorus 本身服务器从一开始就以分布式计算的思想来设计。所以，仿真器把 UNIX 结构翻译成 Chorus 结构。例如，和 Amoeba 一样，在打开文件时，要存取该文件权能并将其存在文件描述符表中。

以上讨论的内容在表 9-12 中作了一个小结。

表 9-12 Amoeba、Mach 和 Chorus 的对比

项目	Amoeba	Mach	Chorus
设计目标	分布式系统	单 CPU，多处理器	单 CPU，多处理器
执行模块	池处理器	工作站	工作站
微内核	是	是	是
内核调用数目	30	153	112
自动负载均衡	是	否	否
权能	一般	单一端口	一般
权能位置	用户空间	内核	用户空间
线程管理者	内核	内核	内核
透明多相性？	是	否	否
用户设置属性？	否	是	是
多处理器支持	最小	广泛性	中等
映射对象	段	内存对象	段
请求分页？	否	是	是
写拷贝？	否	是	是
外部分页程序？	否	是	是
分布式共享内存	基于对象	基于页面	基于页面
RPC？	是	是	是
组通信	可靠、有序	无	不可靠
异步通信？	否	是	是
机器间消息	内核	用户空间/内核	内核
信息流向	进程	端口	端口
UNIX 仿真	源码级	二进制级	二进制级
UNIX 兼容性	POSIX（部分）	BSD	UNIX 系统 V
单服务器 UNIX？	否	是	否
多服务器 UNIX？	是	否	是
最优化目标	远程情况	本地情况	本地情况
文件自动复制	是	是	否

9.8 小结

像 Amoeba 和 Mach 一样，Chorus 也是一种用于分布式系统的基于微内核的操作系统。

它提供二进制代码级的 UNIX 系统 V 兼容性，支持实时应用和面向对象的编程。

Chorus 由三个概念层构成：内核层，子系统，用户进程。内核层包括一个合适的微内核，还有一些运行在核心态，共享微内核地址空间的内核进程。中间层包括子系统，用于建造支持顶层用户程序的操作系统。

微内核提供七种关键的抽象概念：进程、线程、区域、消息、端口、端口组和唯一标识符。进程提供一种收集、管理资源的方法。线程是系统中的活动实体，由内核用优先级调度机制来调度。区域是一片虚拟地址空间，段可以映射到区域上。端口是一个缓冲区域，用来保存送来的但还未读取的消息。唯一标识符是用来标识资源的二进制名字。

微内核和子系统一起提供了其他的结构：权能、保护标识符和段。前两个用于命名和保护子系统资源。第三个是内存分配的基础，在运行进程和磁盘上都要用到。

这一章讲述了两种子系统。UNIX 子系统由进程、对象、流和进程间通信等等管理器组成。它们一起提供了 UNIX 仿真的二进制代码兼容。COOL 子系统提供了面向对象编程的支持。

习题

- (1) 为什么 Chorus 中的权能使用了它们的 UIs 时间序数？
- (2) 区域和段之间的区别是什么？
- (3) 请解释为什么 Chorus 监控程序是依赖于机器的，而实时运行却是与机器无关的。
- (4) 为什么 Chorus 除了用户进程和内核进程以外还需要系统进程？
- (5) 线程被挂起和被停止时都不能运行，那么它们之间的区别是什么？
- (6) 简述 Chorus 如何处理异常和中断，并说明它们为什么采用不同处理方法。
- (7) Chorus 既支持信号量又支持互斥，这是必要的吗？如果只支持信号量是否可以满足需要？
- (8) 映像程序的功能是什么？
- (9) 简述 *MpPullIn* 和 *MpPushOut* 的作用。
- (10) Chorus 支持 RPC 和异步发送，这两者之间的区别是什么？
- (11) 试说明 Chorus 中端口迁移的用处。
- (12) Chorus 中，消息可以发送给一个端口组。那么消息发送到端口组所有的端口还是随机地挑选一个？
- (13) Chorus 中有专门创建和撤消端口的调用，但只有一个创建端口组的调用 (*grpAllocate*)。请合理猜测为什么没有 *grpDelete*。
- (14) 为什么要“小型端口”，它能做平常端口不能做的事吗？
- (15) 为什么 Chorus 支持抢先式和非抢先式调度？
- (16) 请说出 Chorus 和 Amoeba 相似的一处，以及 Chorus 和 Mach 相似的一处。
- (17) Chorus 中端口组的使用与 Amoeba 中组通信有何不同？
- (18) 为什么 Chorus 扩展了 UNIX 信号的语义？

第 10 章 实例研究 4: DCE

在前面三章中已经比较详细地介绍了基于微内核的分布式系统。本章将介绍一种截然不同的实现方法, 开放式系统基金会 OSF (Open System Foundation) 的分布式计算环境 (Distributed Computing Environment), 或者简称为 DCE。同完全摒弃现有操作系统, 因而在本质上是一种崭新的操作系统的基于微内核系统不同, DCE 所建立的分布式系统是存在于现有操作系统之上的。在下一节中将介绍 DCE 的基本思想, 而在其后的几节中再比较详细地介绍 DCE 中各主要组成部分。

10.1 关于 DCE 的介绍

在本节中将对 DCE 进行简要介绍。首先要介绍一下它的历史、设计目标、模式与关键组成部分, 然后再说明在 DCE 中占有重要地位的信元 (cell) 概念。

10.1.1 DCE 的历史

OSF (开放式系统基金会) 是由包括 IBM、DEC 和 Hewlett Packard 在内的一些主要计算机供应商建立的, 它的出现是针对 AT&T 与 Sun Microsystems 联合开发和推广 UNIX 操作系统的。其他公司都非常害怕这种安排会使 Sun 公司处于领先地位。OSF 的初始目标是开发和推广他们所控制的 UNIX 版本 (而不是 AT&T/Sun 所控制的 UNIX 版本)。这种目标随着 OSF/1 版本的推出而得到了实现。

从一开始就非常明显, OSF 联盟的用户希望在 OSF/1 和其他 UNIX 系统上建立分布式应用。OSF 通过发布“技术请求”来响应这种要求, 在这种情况下他们要求各个公司提供工具以及其他软件来建立一个分布式系统。许多公司都进行了支持, 而且这些软件都得到了仔细的评估。OSF 然后选择了其中的一部分软件, 通过进一步开发将它们变成统一的可以运行在 OSF/1 以及其他操作系统的集成软件包——DCE。DCE 现在已经是 OSF 的主要产品。OSF 还计划了一个用于分布式系统管理的完整产品, DME (Distributed Management Environment: 分布式管理环境), 但是这个产品从来也没有实现过。

10.1.2 DCE 的目标

DCE 的主要目标是提供一个可以作为分布式应用运行平台的一致、无缝环境。但是与 Amoeba、Mach 和 Chorus 不同的是, 这个环境是建立在现有操作系统之上的, 开始是 UNIX, 但是后来也被移植到 VMS、Windows 和 OS/2。这种解决方案的主导思想是用户可以聚集一些现有的机器, 通过增加 DCE 软件就可以运行分布式应用。所有的这一切都不影响现有的 (非分布式) 操作系统应用。虽然大多数的 DCE 软件包都运行在用户空间, 但是在一些配置情况下小部分 (分布式文件系统的一部分) 必须被添加到内核中。OSF 只出售源代码, 而供应商就可以将它们集成到自己的产品中去。为了简单起见, 本章的介

绍将集中在基于 UNIX 的 DCE。

DCE 所提供的环境由一组工具、服务以及使它们能够有效工作的内部结构组成。通过选择适合的工具与服务可以使它们以一种集成的方式进行协同工作，这样还可以使分布式系统的开发更轻松。例如，DCE 提供了使开发具有高可靠性应用的更为简单的方法。作为另外一个例子，DCE 提供了一种对不同机器的时钟进行同步的机制来产生一个全局时间。

DCE 运行在许多不同种类的机器、操作系统以及网络上。所以应用的开发人员也可以容易地生成可以运行在不同平台上的软件，降低软件开发成本，提高软件的潜在市场价值。

DCE 应用所运行的分布式系统可以是一个由不同供应商提供的计算机所组成的异构系统，系统中的每一台计算机都有它自己的本地操作系统。操作系统上的 DCE 软件层掩盖了这些不同，在需要的情况下自动在不同的数据类型间进行转换。所有的这些对于应用开发程序员来说都是透明的。

前面所讨论的所有内容导致了 DCE 可以轻易提供编写应用程序的途径，所编写的应用程序可以让位于不同站点的不同用户协同工作，通过共享系统的软硬件资源完成某个目标。在这种情况下的安全问题是一个关键因素，所以 DCE 提供了用于认证和保护的扩展工具。

最后，DCE 也被设计为可以在许多领域内同现有的标准协同工作。例如，一组 DCE 机器不仅可以相互通信，还可以使用 TCP/IP 或者 OSI 协议同彼此或外部进行通信，而资源也可以通过使用 DNS 或者 X.500 命名系统来进行定位。DCE 同时也使用了 POSIX 标准。

10.1.3 DCE 部件

所有的 DCE 编程模式都是客户/服务器模式。用户进程作为客户来访问远程服务器所提供的服务。这些服务中的一部分是 DCE 自身所提供的，但是其他都属于应用程序开发人员所提供的应用程序。在本节中将简要介绍 DCE 包自身所提供的分布式服务，主要包括时间、目录、安全以及文件系统服务。

在大多数 DCE 应用中，客户程序是或多或少与特殊库链接过的普通 C 程序。客户端的二进制程序（文件）中包含的库例中有少部分提供与服务器的接口，而将详细的接口过程从程序员面前隐藏起来。服务器则正相反，一般是比较庞大的后台程序。接口一直运行，不断等待需要进行操作的请求到达。当请求到达之后，服务器对它们进行处理，然后返回应答。

除了提供分布式服务之外，DCE 也实现了两个并不以服务形式出现的分布式编程工具：线程和 RPC（Remote Procedure Call：远程过程调用）。线程工具允许在同一进程中同时存在多个控制线程。虽然一些 UNIX 版本自己就提供了线程，但是使用 DCE 则可以获得在整个系统范围内一致的标准线程接口。在可能的情况下，DCE 可以使用本地线程来实现 DCE 线程，但是在没有本地线程存在的情况下，DCE 提供了一整套从头设计的线程包。

另外一个 DCE 工具是 RPC，它是 DCE 中所有通信的基础。为了访问一个服务，客

户进程通过执行一个 RPC 来访问远程服务器进程。服务器进程执行这些要求然后（可以选择的）发送一个应答。DCE 实现了包括服务器定位、绑定以及执行调用在内的整个机制。

图 10-1 给出了 DCE 环境中不同部分合成的大致思路。最底层是计算机硬件，在其上运行着本地操作系统（包括 DCE 附加在内）。为了支持一个全功能的 DCE 服务器，操作系统必须或者是 UNIX，或者是具有 UNIX 所有必须功能的其他系统，这些必须功能包括多道程序（设计）、本地进程间通信、存储管理、时钟（定时器）以及安全机制。如果只支持 DCE 的客户，那么则需要很少的功能，甚至 MS-DOS 也可以。

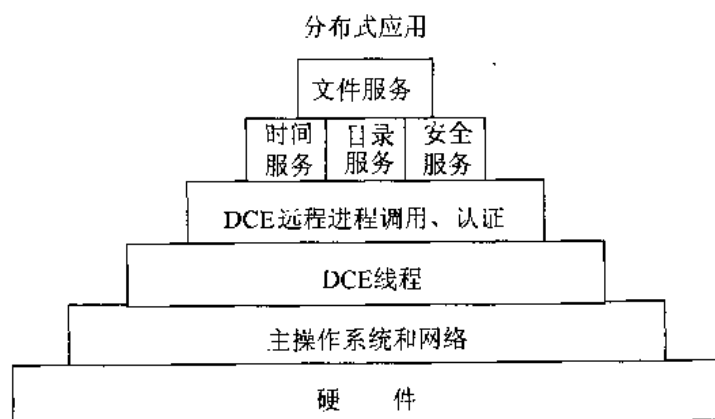


图 10-1 DCE 部分合成的大致思路

在操作系统层之上是 DCE 的线程包。如果操作系统自身就具有合适的线程包，那么 DCE 线程库所起到的作用只是将它的接口转换成为标准的 DCE 接口。如果没有本地的 DCE 线程包，那么 DCE 会提供一个几乎完全运行在用户空间上的线程包，在这个包中只有很少的几百行用于线程堆栈管理的汇编代码运行在内核中。再下来就是 RPC 包，它像线程代码一样是库例程的集合。由于安全部分需要执行授权 RPC，所以在逻辑上它也处于这个层次中。

在 RPC 层之上是不同的 DCE 服务。并不是在所有的机器上都要运行所有服务。系统管理员可以决定在哪里运行哪个服务。标准的服务包括时间、目录以及安全，在它们上面是分布式文件服务，如图所示。在典型的设置中，这些服务只运行在“系统服务器”上，而不运行在客户工作站上。下面是对它们的一些简要介绍。

线程以及 RPC 包的基本功能是清楚的，但是对服务进行一下简单的介绍在目前情况下仍然有助于概念的了解。有关服务与工具的详细介绍将在后面进行。分布式时间服务（distributed time service）存在的原因主要是因为系统中每一台机器都有它自己的时钟，所以在系统中时间概念就比在单一系统复杂。例如在 UNIX 系统中的 *make* 程序会检查源代码以及执行代码的建立时间以决定从上一次编译之后有哪些源代码发生了改变。只有这些改变了代码需要重新进行编译。还应考虑一下源代码与执行代码分别存放在不同的机器上，而且这些机器的时钟不同步的情况。在这里执行文件建立的时间比源文件建立时间迟并不一定意味着这部分代码的重编译工作可以被忽略。这种时间延迟可能主要是因为机器之间的时钟不同步而造成的。DCE 时间服务尝试通过保持时钟同步来获得全局时间以解

决这个问题。

目录服务 (directory service) 主要用来记录所有资源的位置。这些资源包括机器、打印机、服务器、数据等等, 而且它们可能分布于整个世界。除非进程自己关心, 否则目录服务允许一个进程在不关心资源位置的情况下请求一种资源。

安全服务 (security service) 保护各种类型的资源只被经过合法授权的人员使用。例如, 在一个医院信息系统中, 可能存在医生只能访问她自己的病人而不是其他医生的病人的医疗信息的政策。工作在记账部门的人员能够看到任何病人信息, 但只能看到财务方面的信息。如果进行了一个血液检查, 医生可以看到检查结果, 而记账员则只能看到检查费用 (而不是检查结果)。安全服务提供了建立与之类似的应用的辅助工具。

最后, 分布式文件服务 (distributed file service) 是一个提供以相同方式透明访问全系统所有文件的全局文件系统。它既可以被建立在宿主机的本地文件系统之上, 也可以直接取代本地文件系统。

在 DCE 结构的最上层是分布式应用。它们可以使用 (或者跳过) 任何 DCE 工具或服务。简单应用可能只是 RPC 的应用 (通过透明地使用类库), 而更复杂的应用则可能进行许多包括所有服务在内的外部调用。

图 10-1 所给出的结构并不很精确, 但是不同部分之间的依赖关系本身就不容易判断, 这主要是因为它们在很多情况下是递归的。例如, 目录服务使用 RPC 来实现不同服务器之间的通信, 但是 RPC 包又使用目录服务来确定目的地的位置。因而目录服务在 RPC 之上这个说法严格来说并不正确。第二个例子则显示了平行依赖性, 时间服务使用安全服务来判断时间服务的使用者, 但是安全服务也使用时间服务来发放生命周期很短的授权。不管怎样, 图 10-1 给出了系统的一个大致结构, 因此将用它来作为研究的模式。

10.1.4 信元

DCE 系统中的用户、机器及其他资源可以合成起来形成信元 (cell)。DCE 系统的命名、安全、管理等方面都基于这些信元。信元的边界通常是组织单位的映射。例如, 大公司中的一个部门或小公司可能就是一个信元。

在将资源分组为信元的过程中需要考虑以下四个因素:

- (1) 目的 (purpose);
- (2) 安全 (security);
- (3) 开销 (overhead);
- (4) 管理 (administration)。

第一个因素, 目的。意味着信元中的所有机器 (包括用户在内) 都应该为共同的目标或者一个共同的长期项目 (有可能以年来计算) 而努力。用户应该相互知道, 而且之间的联系要比他们同信元之外用户的联系要紧密。公司内的部门通常就是按照这种方式进行组织的。信元也可以按照所提供的服务进行组织。如在线银行, 所有的自动柜员机和中央计算机都属于一个信元。

第二个因素, 安全。同 DCE 的信元内用户之间的相互信任程度要比信元内外用户之间的相互信任程度要高这一事实有关。原因是信元边界的作用非常类似于防火墙——获取内部资源是可以直接进行的, 但是访问另一个信元的任何资源都需要这两个信元进行协商

以获取相互之间的信任。

第三个因素，开销。非常重要，这是因为一些 DCE 函数在信元内操作时可以被优化（如安全）。在这里地理位置有可能会起到一定作用，因为将物理位置分布很远的用户作为一个信元存在通常需要通过广域网进行通信。如果广域网络速度很慢而且不可靠，那么就会需要额外的开销来处理这些问题并在必要的时候进行补偿。

最后，每一个信元都需要管理员。如果一个信元中所有机器和人员都属于一个部门或项目，那么在指定管理员方面就不会存在任何问题。但是如果它们分属于两个不同的部门，而且每个部门的管理都有比较独特的方面，那么在确定信元管理员以及进行信元范围的决策方面就会存在很大困难。

同这些限制条件相反的是应该有尽可能少的信元以尽量减少跨信元的操作。而且如果一个侵入者侵入一个信元而且窃取了安全数据库，那么就必须为所有其他信元都设定新的密码。信元的数量越多，这部分工作也就越多。

为了使信元的概念更清楚，下面来研究两个样例。第一个是一个大型的电子设备生产商，它的产品包括从飞机引擎到面包烘烤机等许多产品。第二个是出版社，它所出版的读物包括从艺术类到动物园类等许多类型。由于电子设备生产商的产品过于繁多，所以它很有可能按照产品来组织信元，如图 10-2 (a) 所示，生产面包烘烤机和生产飞机引擎是由不同信元进行的。每一个信元中都包括设计、生产以及市场人员，当然条件是飞机引擎的市场销售人员同飞机引擎的生产人员的联系要比同面包烘烤机的市场销售人员的联系要紧密。

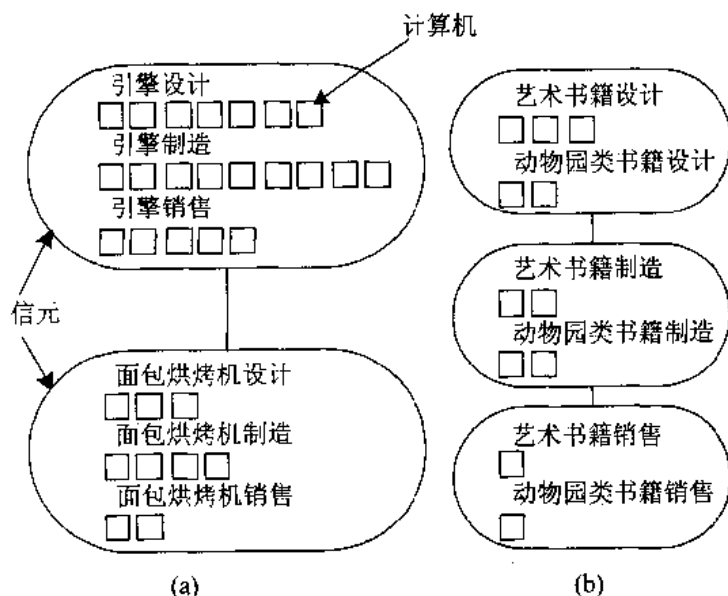


图 10-2 (a) 按产品组织的两个信元
(b) 按功能组织的三个信元

相反的，出版社可能会按照图 10-2 (b) 的方式来组织信元。这主要是因为生产（也就是印刷与装订）一本艺术方面的书同生产一本有关动物园类的书的过程非常类似，所以部门之间的差异要比产品之间的差异要明显得多。另外，如果出版社中有独立自主的儿童

图书、商业图书以及课本的出版分社，那么它们实际上是更为原始的具有自己的管理结构与公司文化的公司，这时按分社划分信元就要比按功能划分信元要好。

以上这些例子的关键之处在于信元边界主要是由商务因素所决定的，而不是由 DCE 的技术特性所决定的。

信元的大小也可以适当进行变化，但是所有信元都应该包含一个时间服务、一个目录服务和一个安全服务。这三个服务可以在同一台机器上运行。除此之外，大多数信元中都或者包含一个或一些应用客户（执行工作）或者包含一个或一些应用服务器（发布服务）。在大型信元中可能存在多个时间、目录和安全服务的实例，同时也有数以百计的应用客户与服务器。

10.2 线程

同 RPC 包一起的线程包是建立 DCE 的基本包之一。在本节中将讨论 DCE 的线程包，而重点就是调度、同步与现有的调用。

10.2.1 DCE 线程介绍

DCE 线程包基于 P1003.4a POSIX 标准。它是一个允许进程建立、删除和操作线程的用户级库过程的集合。然而如果宿主机系统中已经有一个（内核）线程包，那么供应商可以通过设置 DCE 来使用它。基本调用包括线程的建立与删除、等待一个线程以及线程间同步计算。系统还提供了其他一些处理详细信息和其他功能的调用。

一个线程可以处于四个状态，如表 10-1 所示。一个运行（running）线程可以使用 CPU 来进行计算工作。一个就绪（ready）线程等待而且可以运行，但是因为有其他线程在使用 CPU 而无法运行。与此相反，一个等待（waiting）线程在逻辑上是不能运行的，这主要是因为它在等待某个事件（如互斥体的解锁）的发生。最后，一个终止（terminated）线程是已经退出但是还没有被删除，而且内存（也就是堆栈空间）还没有被清除的线程。只有另外一个线程明确的删除这个线程时这个线程才会真正消失。

表 10-1 一个线程可以处于四个状态

状态	说明
运行（Running）	线程正处于使用 CPU 的活动状态
就绪（Ready）	线程等待运行
等待（Waiting）	线程被阻塞以等待某个特定事件的发生
终止（Terminated）	线程已经退出但还没有被清除

在一台只有一个 CPU 的机器上，在任何时间都只有一个线程可以真正处于运行状态。在多处理机中，一个进程中的多个线程可以在不同的 CPU 上同时运行（真正的并行）。

线程包的设计要实现对现有软件系统的冲击最小，所以设计为单线程环境的程序只需要很少量的工作就可以转换成为适合多线程环境的程序。理想的是，一个单线程程序只需要通过将一个参数指定为多线程运行就可以实现全部转换工作。然而问题主要出在三个

方面。

第一个问题同信号有关。无论是忽略还是捕获，信号都可以维持它们的缺省状态。一些信号是同步的，它们由运行线程所引起。它们包括浮点异常，内存保护错误以及本地定时器溢出（go off）。其他是由外部系统所引起的异步信号，如用户通过敲击 DEL 键来中止当前正在运行的进程。

当一个同步信号出现时，它由当前线程进行处理，除非它既不被忽略也不被捕获，这时整个进程将被杀死。当一个异步信号出现时，线程包就会检查是否有线程在等待它。如果有的话，这个消息就会被传送到所有等待处理它的线程中。

第二个问题同标准库有关。标准库中的大多数过程是不可重入的。当一个线程正在进行内存分配时，一个时钟中断可能会引起线程切换。在进行切换时，内存分配器的内部数据结构可能会发生不一致的现象，这样当新调度的线程企图分配一块内存时就会发生问题。

这个问题是通过为一些标准库过程（大部分是 I/O 过程）添加实现调用互斥的外壳来解决。对于其他过程，系统中通过一个全局互斥体来保证任何时候只有一个线程在库中处于活动状态。诸如 *read* 和 *fork* 之类的库过程都是由外壳保护的过程，这个外壳主要处理互斥问题，然后调用另外一个（隐藏）过程真正完成工作。这种解决方法从某些方面来说是一种快速的简单办法，更好的解决办法应该是提供正确的可重入库。

第三个问题同 UNIX 系统调用在全局变量 *errno* 中返回它们的错误结果有关。如果一个线程执行了系统调用，但是在调用刚结束就有另外一个线程被调度执行，而且这个线程也执行了一个系统调用，在这种情况下原来的 *errno* 值就会丢失。一种解决方法是提供一种替换性质（可供选择）的错误处理接口。它由一个允许程序员访问的 *errno* 宏组成，通过该宏可以检查与特定线程有关的 *errno* 版本（version），该 *errno* 版本随进程调度而进行保存或恢复。这种解决办法避免了检查 *errno* 全局版本的必要。除此之外，系统调用也可以通过产生异常来说明错误，因而就完全避免了这个问题。

10.2.2 调度

除了线程调度是应用可见之外，线程调度同进程调度非常相似。调度算法决定了一个线程所能够运行的时间以及下一个运行的线程。通过直接比较进程调度就可以获得许多线程调度算法。

DCE 中的线程具有优先级，而且这些优先级对于调度算法来说是有意义的。高优先级的线程被认为比低优先级线程更重要，因而也会得到更好的处理，这意味着它们不但可以优先运行，而且还可以获得更多的 CPU 时间。

DCE 支持图 10-3 所示的三种调度算法。第一个是 FIFO。它在优先级队列中从高到低进行搜索以找出一个或若干具有最高优先级的线程。这个队列中的第一个线程这时就可以一直运行到阻塞或退出为止。从原则上来说，被选择的线程可以尽它的需要运行。当它结束之后，从可运行队列中将它删除。然后调度程序会再一次从高到低在队列中查找优先级最高的线程运行。

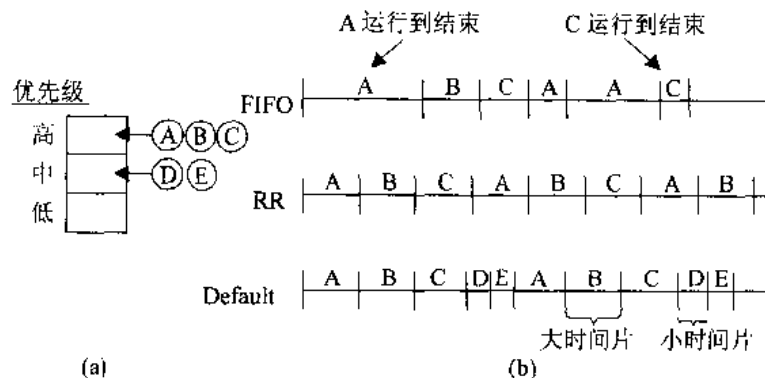


图 10-3 (a) 一个具有五个线程和三种优先级的系统
(b) 三种线程调度算法

第二种调度算法是回转 robin 算法（循环算法）。在本算法中调度程序保存最经常进行操作的队列并使它们依次运行一段固定的时间片。如果一个线程在时间片使用完之前被阻塞或退出，才会（暂时）从队列系统中删除。如果它用完了所有时间片，它将被挂起并放到队列末尾。在图 10-3 (b) 中间的例子中，线程 A、B 和 C 在需要的情况下可以一直交替运行下去。在高优先级线程需要运行的情况下，中等优先级线程 D 和 E 将永远不会得到运行的机会。

第三个算法是缺省算法。它以时间片回转 robin 算法运行所有队列上的线程。但是优先级越高的线程可以得到更多的时间片。在这种方式下所有的线程都可以得到运行机会，因而不会出现第二种算法中的线程饥饿情况（thread starvation）。

实际中还存在具有时间片变长的第四种算法，但是在这种情况下会出现线程饥饿。然而这种方法并没有在 POSIX 中定义，所以它不具有可移植性，因而应该避免使用。

10.2.3 同步

DCE 中提供了两种线程同步方法：互斥体与条件变量。当必须避免多个进程同时访问同一个资源时应该使用互斥。例如，当在链接表中移动元素时，在移动过程中整个链接表将处于不一致状态。为了避免灾难，当一个线程在操作链接表时，所有其他线程都不能再访问这个链接表。通过要求一个线程首先锁定同链接表相关联的互斥体，然后再去访问链接表（访问后对互斥体进行解锁）的方法就可以保证操作的正确性。

实际中存在三种互斥体类型，如表 10-2 所示。它们的主要区别在于对于嵌套锁定的处理方式不同。快速互斥体（fast mutex）类似于数据库系统的锁。如果一个进程企图锁定一个解锁的记录将获得成功。然而如果它企图第二次获得同一个锁时，它就会被阻塞，等待这个锁被释放，而这是永远不会发生的。这就是一个死锁（deadlock）。

表 10-2 DCE 支持的三种互斥类型

互斥类型	说明
Fast	对它的第二次锁定导致一个死锁
Recursive	允许对它进行二次锁定
Nonrecursive	对它的第二次锁定导致一个错误

一个递归互斥 (recursive mutex) 允许一个线程再一次锁定它已经锁定的互斥体。这种互斥的思路在于如果一个线程的主程序锁定了一个互斥体, 那么它所进行的调用也有可能需要锁定这个互斥体。为了避免死锁就必须接受第二个锁定。只要这个互斥体最终的解锁次数同加锁次数相同, 那么加锁的嵌套次数就可以任意。虽然递归互斥具有很高的易使用性, 但是它们的速度很慢, 所以程序员就必须进行选择。DCE 中提供了一种折衷的第三种互斥体, 任何企图第二次锁定这种互斥体的线程都将得到一个错误, 而不是被阻塞。

条件变量 (condition variable) 提供了第二种同步机制。它们可以同互斥体一起使用。典型的, 当一个线程需要某个资源时, 它使用一个互斥体来获得访问一个保存这个资源状态的数据结构的排它权限。如果这个资源没有处于可用状态, 那么这个线程会等待在一个条件变量上, 这将导致线程自动被挂起和互斥体被释放。后来当其他线程给条件变量发送信号时, 这个等待的线程就会重新启动。

10.2.4 线程调用

DCE 的线程包总共有 54 个原语 (库过程)。这些原语中有许多都不是必须的, 系统提供它们只是为了方便起见。这种解决方法有些类似于只具有 +、-、× 和 / 四个功能的便携计算器, 但并不只有 +、-、× 和 / 四个功能键, 它同时也提供了 +1、-1、×2、×10、× π 、/2 和 /10 等功能键, 它们主要是为了节省用户的时间和提高效果。由于调用的数量很多, 在这里只讨论最关键的一些 (大约只有总数的一半)。然而我们的目标应该是对调用的现有功能有一个正确的印象。

表 10-3 部分 DCE 的线程管理原语

调用	说明
Create	建立一个新线程
Exit	线程结束后调用它
Join	类似 UNIX 系统中的 WAIT 系统调用
Detach	当调用者退出后, 父进程没有必要进行等待

注: 本节中所有原语都是以 *pthread_* 为前缀 (也就是 *pthread_create* 而不是 *create*)。在这里我们省略了前缀以节省空间。

为了讨论方便起见, 我们可以将这些调用分为七个大类, 每一个大类的调用都处理线程及其使用的某一个方面。表 10-3 中列出了第一个大类的调用, 作线程管理用。这些调用主要用来处理线程的建立以及结束后的退出。一个父线程可以通过使用 *Join* 来等待它的某个子线程, 这与 UNIX 中的 WAIT 系统调用非常类似。如果父线程不关心它的子线程, 而且也不准备等待它, 那么父线程就可以通过调用 *detach* 来丢弃子线程。在这种情况下, 当子线程退出时, 它的存储被立即回收而不等待父线程调用 *join*。

DCE 包允许用户为线程、互斥以及条件变量建立、删除和维护一个模板。模板可以被设定成合适的初始值。当一个对象被建立时, 建立调用中的一个参数就是指向模板的指针。例如, 可以建立一个线程模板并将它的堆栈大小属性设置为 4K。无论何时, 只要有线程以该模板为参数建立, 那么这个线程就会获得 4K 的堆栈空间。建立模板的关键意义就在于避免为每一个选项独立赋值的弊病。仅就这个包而言, 建立调用可以保持一致。而且新的属性也可以被添加到模板中。表 10-4 中列出了一些模板调用。

表 10-4 部分模板调用

调用	说明
Attr_create	建立线程参数模板
Attr_delete	删除线程模板
Attr_setprio	设定模板中的缺省调度优先级
Attr_getprio	读取模板中的缺省调度优先级
Attr_setstacksize	设定模板中的缺省堆栈尺寸
Attr_getstacksize	读取模板中的缺省堆栈尺寸
Attr_mutexattr_create	建立互斥体参数模板
Attr_mutexattr_delete	删除互斥体参数模板
Attr_mutexattr_setkind_np	设定模板中的缺省互斥体类型
Attr_mutexattr_getkind_np	读取模板中的缺省互斥体类型
Attr_condattr_create	建立条件变量参数模板
Attr_condattr_delete	删除条件变量模板

attr_create 和 *attr_delete* 调用分别建立和删除线程的模板。其他调用允许程序读取或者更新模板的属性，如堆栈大小以及模板中供线程调度使用的调度参数。相似的，系统也为互斥体和条件变量提供了建立和删除模板的调用。由于互斥体和条件变量没有任何属性和操作，所以这些模板的必要性并不十分明显。可能系统的设计者考虑到有一天有人会需要一个属性。

第三组调用处理互斥体，这些互斥体可以动态建立和删除。为互斥体定义的操作主要有三个，如表 10-5 所示。这些操作有互斥体的加锁、解锁以及尝试加锁但是在加锁不可能完成的情况下接受错误。

表 10-5 部分互斥体调用

调用	说明
Mutex_init	建立一个互斥体
Mutex_destroy	删除一个互斥体
Mutex_lock	尝试锁定一个互斥体；如果已经锁定则阻塞
Mutex_trylock	尝试锁定一个互斥体；如果已经锁定则失败
Mutex_unlock	解锁一个互斥体

下一组是有关条件变量的调用，如表 10-6 所示。条件变量也可以动态建立和删除。线程可以等待在挂起某个必须资源的条件变量上。系统提供了两个唤醒操作：signaling，它可以只唤醒一个线程；broadcasting，它可以唤醒所有的等待线程。

表 10-6 部分条件变量调用

调用	说明
Cond_init	建立一个条件变量
Cond_destroy	清除一个条件变量
Cond_wait	在一个条件变量上等待，直到信号或广播到达为止
Cond_signal	最多唤醒一个等待在该条件变量上的线程
Cond_broadcast	唤醒所有等待在该条件变量上的线程

表 10-7 列出了与操作每线程全局变量有关的三个调用。这些变量有可能被建立它们的线程中的任何过程所使用，但是对于其他线程来说却是不可见的。由于所有常用语言中并不支持每线程全局变量的概念，所以它们必须在运行时进行维护。第一个调用建立一个标识符，同时申请存储空间。第二个指定了一个指向每线程全局变量的指针，而第三个则可以让线程读出每线程全局变量的值。许多计算机科学家都认为全局变量与 GOTO 语句属于相同范畴，所以他们将会非常乐于使这些调用的使用非常笨拙。（作者自己曾经尝试设计过一个程序语言，这个语言中具有类似：

IKNOWTHISISASTUPIDTHINGTODOBUTNEVERTHELESSGOTO LABEL;

的说明，但是被他的同事阻止了）。由于大多数程序语言都不允许这个概念在句法上的表现，所以让每线程全局变量使用过程调用，而不是类似全局或局部的语法作用规则，是一种值得讨论的应急的简单规则。

表 10-7 部分每线程全局变量调用

调用	说明
Keycreate	为当前线程建立一个全局变量
Setspecific	设定一个每线程全局变量指针
Getsepcific	从每线程全局变量中读取一个指针值

下一个调用组如表 10-8 所示，主要处理线程的杀死以及线程的抗拒能力。Cancel 调用尝试杀死一个线程，但是有时杀死一个线程会造成毁灭性的影响，如这个线程当前正锁定了互斥体的情况。为了这个原因，线程可以通过许多途径来控制杀死它们的努力，这非常类似于 UNIX 进程可以捕获或者忽略信号而不是为它们所终止。

表 10-8 部分有关杀死进程的调用

调用	说明
Cancel	尝试杀死另外一个线程
Setcancel	设置其他线程是否可以杀死本线程

最后一组调用如表 10-9 所示是有关调度的。包中允许进程中的一个线程根据 FIFO、回转 robin、抢占、非抢占以及其他调度算法进行调度。通过使用这些调用，就可以对调度算法和优先级进行设置。只有线程不建立在冲突调度算法之上，系统才可能获得最佳运行效果。

表 10-9 部分调度调用

调用	说明
Setscheduler	设定调度算法
Getscheduler	读取当前调度算法
Setprio	设定调度优先级
Getprio	读取调度优先级

10.3 远程过程调用

DCE 基于客户/服务器模型。客户通过执行远程过程调用来向远程服务器请求服务。在本节中将说明这种机制在两端的表现和实现。

10.3.1 DCE RPC 的目标

DCE RPC 系统的设计目标相对较为传统。首先也是最重要的，RPC 系统使客户只通过简单调用本地过程来访问远程服务器成为可能。这个接口可以使客户（也就是应用）程序的编制更为简单，对于大多数程序员来说也更熟悉。同时将旧有程序代码向分布式系统的移植工作也大大减轻，几乎可不作任何修改。

RPC 系统可以将所有详细信息从客户面前隐藏起来，而且从一定程度上来说在服务器上也是如此。RPC 系统可以自动定位服务器并绑定到其上，这一切客户都不需要了解。它也可以处理双向消息传递，在需要的情况下对它们进行分割与合成（如其中的一个参数是大数组的情况）。最后，RPC 系统可以自动处理客户和服务器之间的数据类型转换，甚至它们可以分别运行在不同的体系结构之下，而且具有不同的字节次序。

作为 RPC 系统的隐藏细节能力的一个必然结果，客户与服务器彼此之间是高度独立的。一个客户可以使用 C 语言来编制，而服务器则可能用 FORTRAN 编制，情况也有可能正好相反。客户和服务器可以运行在不同的硬件平台之上，它们所使用的操作系统也有可能不同。同时系统还支持许多网络协议与数据表示，所有这些工作都不要客户或服务器的介入。

10.3.2 客户与服务器的编写

DCE RPC 系统由很多部件组成，它们包括语言、库、后台（daemon）和工具程序等等。这些合起来就可以进行客户和服务器的编写工作。在本节中将分别说明这些部分以及它们是如何组合在一起的。RPC 客户和服务器编写与使用的全部过程在图 10-4 中进行了总结。

在客户/服务器系统中，所有部分能够集成起来的关键因素是接口的定义。这是服务器同它的客户之间的协议，这个协议指定了服务器提供给客户的服务。这个协议的实质表示是一个文件，一个接口定义文件，它列出了所有服务器允许客户远程调用的所有过程。每一个过程都有一个列表说明它的参数与结果的名称与类型。理想的情况下，接口的定义中还应该包含有关过程所进行的工作的正式说明，但是这种说明现在还属于能力之外。所以接口定义只定义了调用的语法，而不是它们的语义。编程人员最多可以增添一些说明他希望这个过程所能够完成的工作的说明。

接口定义是通过著名的接口定义语言 IDL（Interface Definition Language）编制的。它允许以一种非常类似于 ANSI C 函数的形式来说明过程。IDL 文件中也包含类型定义、常量声明以及其他用于合成参数与解析结果的信息。

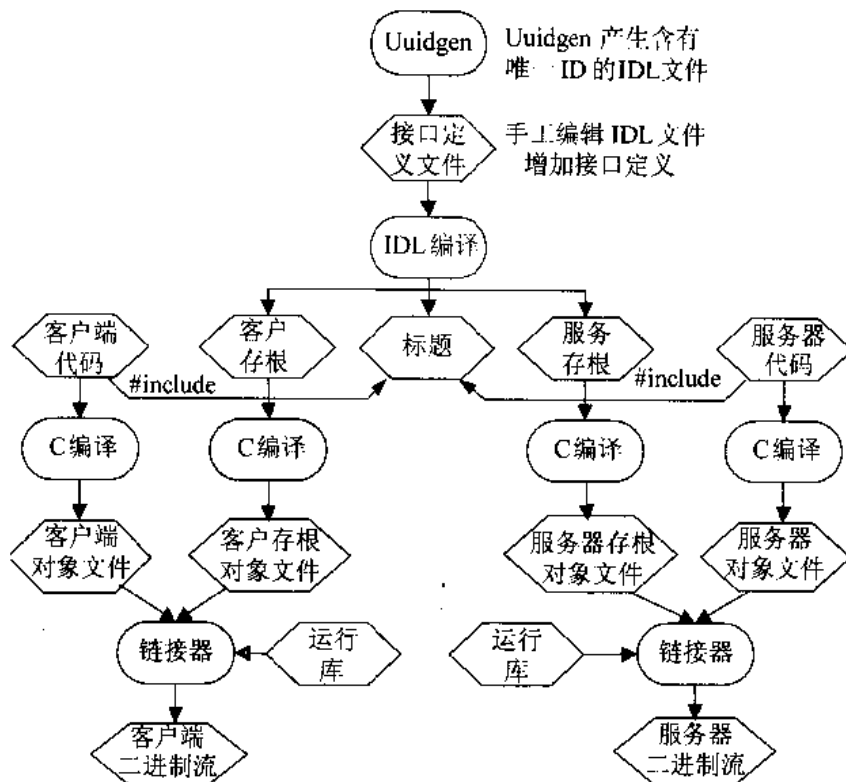


图 10-4 编写客户与服务器的步骤

每一个 IDL 文件中最关键的部分是一个唯一标识接口的标识符。客户在第一个 RPC 消息中发送标识符，而服务器将会认证这个标识符。通过这种方式，如果一个客户不小心申请绑定错误的服务器，甚至是正确服务器的老版本，那么服务器都可以检查出这种错误并拒绝绑定要求。

接口定义和唯一标识符是同 DCE 密切相关的。如图 10-4 所示，编写客户/服务器应用的第一个步骤通常是调用 *uuidgen* 程序来建立一个 IDL 文件原型，这个文件原型中包含一个不会被任何其他 *uuidgen* 重复生成的唯一标识符。这种唯一性是通过生成位置和时间进行编码来保证的。它在 IDL 文件中由一个以十六进制 ASCII 字符串表示的 128 位二进制数组成。

第二步是编辑 IDL 文件，同时填写远程过程的名称和参数。值得注意的是 RPC 并不是完全透明的——例如，客户和服务器不能共享全局变量——但是 IDL 规则使过去不能实现的构想成为可能。

当 IDL 文件完成之后就可以调用 IDL 编译器来处理它。IDL 编译器的输出结果包括三个文件：

- (1) 一个头文件（如：C 语言形式下的 *interface.h*）；
- (2) 客户存根（*client stub*）；
- (3) 服务器存根（*server stub*）。

头文件中包含了唯一标识符、类型定义、常量定义以及函数原型。它应该被客户和服务器代码所包括（通过使用 *#include*）。

客户存根包含了客户程序将要调用的实际过程。这些过程负责将参数收集、打包成输出消息，然后调用运行时系统发送这些消息。客户存根也处理返回应答的解包和将它们返回给客户。

服务器存根中包含了当消息到达后服务器机器上运行时系统所调用的过程。然后又会调用完成实际工作的服务器过程。

接下来程序员的工作就是编写客户和服务器代码，伙同两个存根过程进行编译。得到的客户代码和客户存根文件，然后就可以同运行库进行链接产生客户的运行程序。类似的，服务器代码和存根也经过编译和链接产生服务器运行程序。通过客户和服务器的协同运行就可以使整个应用运行起来。

10.3.3 客户到服务器的绑定

在客户可以调用服务器之前，它首先必须确定服务器的位置，然后再绑定到这个服务器上。虽然普通用户可以忽略绑定而让存根自己去自动处理，但是绑定过程是不能避免的。而高级用户则可以详细控制这个过程，如选定位于远程信元中的一个特殊服务器。在本节中将说明 DCE 中绑定是如何进行的。

在绑定过程中存在的主要问题是客户如何找到正确的服务器。从理论上来说，给每一个信元中的每一个进程发送包含唯一标识符在内的广播消息，通过服务器的回应来进行定位的方法是可行的（安全问题另外考虑），但是由于这种方法的缓慢和浪费时间，导致了它不能实际应用。除此之外，并不是所有的网络都支持广播。

实际上，DCE 服务器的定位主要通过如下两个步骤完成：

- (1) 服务器机器的定位；
- (2) 服务器机器上正确进程的定位。

以上每一个步骤都可以采用许多机制来实现。服务器机器定位的需要是显而易见的，但是当机器定位完成后如何找到具体的服务就比较模糊。客户和服务器能够安全可靠地进行通信的基本需求是网络连接。这种连接需要端点（endpoint），它是服务器机器上可以用来进行网络连接和消息发送的数字地址。让服务器具有固定的地址存在一定风险，这主要是因为同一机器上的其他服务器也有可能会选择相同的地址。由于这个原因，端点是可以被动态指定的，而一个位于服务器机器上的 RPC 后台程序（RPC daemon）保存了登记项为（服务器，端点）的数据库，详情如下所述。

绑定过程可以如图 10-5 所示。在可以接收请求之前，服务器必须向操作系统申请一个端点。然后它将这个端点在 RPC 后台程序中进行注册。RPC 后台程序就会在端点表中记录这些信息（包括服务器所使用的协议）以方便以后使用。服务器也会在一些信元的目录服务器中进行注册，同时将自身所在的主机地址编号传送过去。

现在让我们看一看客户端。以最简单的情况为例，在进行第一个 RPC 调用时，客户存根请求信元的目录服务器查找运行所请求服务器实例的主机。然后客户就可以找到保存端点信息的 RPC 后台程序，请求它在端点表中查找服务器的端点（如 TCP 端口）。拥有这些信息之后就可以进行 RPC 调用。在随后的 RPC 调用中没有必要执行这种查找。DCE 也为客户提供了在必要的情况下执行高级查找以获取合适服务器的能力。认证的 RPC 也是一种可选项。在本章的后面将详细介绍认证和保护。

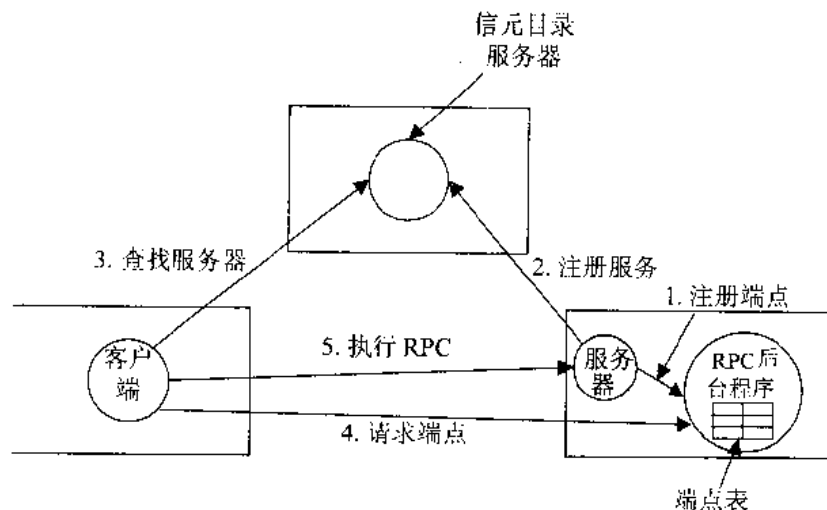


图 10-5 DCE 中客户到服务器的绑定

10.3.4 RPC 的执行

实际的 RPC 一般采用通常的方式透明实现。客户存根可以组装参数并且根据传送需要结果缓冲区（有可能按部分）使用绑定过程中所选择的协议传递给运行时库（runtime library）。当消息到达服务器端时，它会根据消息中所包含的端点信息重新定位到正确的服务器上。运行时库可以将消息传递到服务器存根上，然后它就可以对这些参数进行解析，同时调用服务器来完成所请求的服务。应答的路线则正好相反。

DCE 提供了几种语义选项。缺省值是最多一次（at-most-once）操作，在这种情况下没有任何调用的执行是超过一次的，即使在系统崩溃的情况下也是如此。实际中这意味着如果在 RPC 调用过程中系统发生崩溃而又迅速恢复，那么客户不需要重复执行这个操作，因为它有可能已经执行过一次了。

同样，一个远程过程也可以被标注为幂次（在 IDL 文件中）。在这种情况下，这个调用可以被多次执行而没有损害。例如，读取一个文件的特定块的操作就可以被重复执行，直到成功为止。当一个幂次 RPC 由于服务器崩溃而失败，客户可以等服务器重新启动后再一次进行调用。其他语法在理论上也是存在的（但是很少被使用），它们包括将 RPC 以广播方式发送到局域网络的所有机器。

10.4 时间服务

时间在大多数分布式系统中都是一个重要的概念。可以用射电天文学为例进行说明。分布在世界各地的射电望远镜都同时观察同样的射电天体，同时精确地记录数据以及观察时间。这些数据通过网络传送到一个中央计算机中进行处理。对于一些实验来说（如长程干涉），不同数据流之间的精确同步是必须的。因而实验的成败就取决于是否能够使远程的时钟精确同步。

另外一个例子，在一个计算机股票交易系统中，谁第一个向待售的股票投标可能具

有十分重要的意义。如果投标人分布在纽约、旧金山、伦敦和东京，对投标人加盖时间戳可能是保证公平的一种途径。然而如果所有这些地方的时钟并没有正确同步，那么整个方案就会失败，而且还会引发一系列的以时间为核心的法律纠纷。

为了避免这样的问题，DCE 中有一个称为 DTS (Distributed Time Service: 分布式时间服务) 的服务。DTS 的目标是保证位于不同机器上的时钟的同步。对于它们来说，同步一次是不够的，因为不同机器中晶振的脉动速率也稍有不同，所以不同机器的时间会逐渐漂移。例如，一个时钟的误差可能是百万分之一，这就意味着即使这个时钟设置良好，一个小时之后该时钟也会有 3.6 毫秒的误差。而一天以后，这个误差就会积累到 86 毫秒之多。一个月之后，两个经过精确同步的时钟就有可能相差 5 秒。

DTS 管理着 DCE 中的时钟。它由若干时间服务器组成，这些时间服务器同系统中其他部件一样不停地互相询问“现在是什么时间？”如果 DTS 知道每一个时钟时间漂移（由于它管理着漂移速率）的最大值，它就能够以保证同步所必需的频度进行时间同步。例如，如果时间精度为百万分之一，如果不允许时钟的误差在 10 毫秒以上，那么至少每三小时应该执行一次重新同步。

实际上，DTS 必须分别处理两个问题：

- (1) 保证时钟相互一致；
- (2) 保证时钟与现实一致。

第一点是保证所有时钟在进行时间请求时都会返回相同的值（当然根据时区不同有所调整）。第二点是保证如果所有时钟都返回相同时间，那么这个时间应该同现实世界相一致。如果系统中所有计算机的时间都是 12:04:00.00 而实际时间则为 12:05:00.000 的话意义也不大。下面将详细介绍 DTS 是如何实现这些目标的，但是将首先说明 DTS 的时间模型以及程序的接口。

10.4.1 DTS 时间模型

在大多数系统中，当前时间是一个简单的二进制数，而在 DTS 系统中所有时间都以时间间隔的形式出现。当询问当前时间时，DTS 系统并不回答是 9:52，而有可能回答当前是 9:51 和 9:53 之间的某个时间（经过放大）。使用这种间隔而不是具体的时间可以为用户提供时钟所可能的最大偏移的精确定义。

DTS 在内部以 64 位二进制数记录着从时间开始到现在的时间值。UNIX 系统中这个时间是从 1970 年 1 月 1 日 0 时开始，而 TAI 系统中这个时间是从 1958 年 1 月 1 日 0 时开始，在 DTS 系统中这个日期同它们都不相同，它是从 1582 年 10 月 15 日 0 时开始，在这一天罗马历法被引入了意大利（人们永远也不知道是不是有一个 17 世纪的 FORTRAN 程序突然有一天要运行）。

使用者并不需要去处理时间的二进制表示。它只用于时间的存储与比较。时间的显示形式如图 10-6 所示。这个表现形式是基于国际标准 8601 的，该标准通过既不按照月/日/年（美国日期表现形式）形式也不按照日/月/年（所有其他国家）形式表现日期来解决这个问题。它使用一个 24 小时的时钟，并精确到 0.001 秒。它还使用同格林尼治标准时间的差距来表示时区。最后，也是最重要的，误差是以秒在“J”后面表示的。在本例中的误差是 5.000 秒，这意味着准确时间可能是 3:29:55PM 到 3:30:05PM 之间的任何

值。除了绝对时间之外，DTS 还管理着包括误差在内的时间差异。

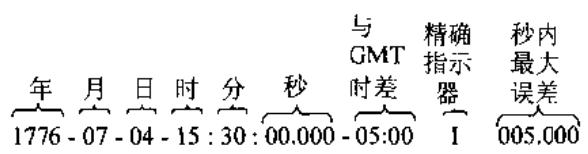


图 10-6 DTS 的时间显示

以间隔形式记录时间的方式引起了其他系统所没有的一个问题：时间的比较并不一定总能够进行，如 UNIX 系统的 *make* 程序。假设一个源文件时间为 10: 35: 10 到 10: 35: 15 而相应的执行文件时间为 10: 35: 14 到 10: 35: 19。执行文件的时间虽然有可能比源文件更新，而实际并不一定如此。解决问题的最好办法就是重新编译这一段程序。

从总体上来说，当一个程序向 DTS 请求比较两个时间时可能会得到三种结果：

- (1) 第一个时间更早；
- (2) 第二个时间更早；
- (3) DTS 不能确定哪个时间更早。

使用 DTS 的软件必须准备接受所有三种结果。为了提供同老程序的向后兼容性，DTS 也提供了传统的单值时间接口，但是盲目使用这种接口有可能会产生错误。

DTS 支持 33 个有关时间的调用（库过程）。这些调用如表 10-10 所示分为六大组。我们现在逐一对它们进行简要介绍。第一组从 DTS 中获取并返回当前时间。这一组两个调用的区别主要在于对时区的处理方式不同。第二组调用主要处理时间二进制值、结构值以及 ASCII 值之间的相互转换。第三组用于找到覆盖作为输入的两个时间的非精确时间间隔。第四组主要用来比较使用或者不使用非精确部分的两个时间。第五组提供了时间的加、减以及给相对时间乘以常数等运算。最后一组用于管理时区。

表 10-10 DTS 与时间有关的组

组	调用个数	说明
时间检索	2	获取时间
时间转换	18	二进制-ASCII 转换
时间操作	3	间隔计算
时间比较	2	比较两个时间
时间计算	5	时间的数学操作
时区使用	3	时区管理

10.4.2 DTS 实现

DTS 服务从概念上有几个部件组成。时间管理员（time clerk）是一个运行在客户机器上的后台进程，它的任务是保证本地时钟同远程时钟的同步。管理员也记录着本地时钟不确定性的增长。例如，一个误差为百万分之一的时钟在一小时内可能会快或者慢 3.6 毫秒。当时间管理员发现可能的误差已经超过允许值时，它就重新进行同步。

一个时间管理员通过同它所在局域网的所有时间服务器（time server）来执行同步操

作。这些后台程序的任务是在一定范围内保证时间的精确与一致。例如在图 10-7 中，一个时间管理员向四个时间服务器提出时间申请并收到应答。其中的每一个都提供了它们所认为的 UTC 所在的时间间隔。管理员按如下步骤计算新时间。首先，管理员会由于不可信而忽略所有与其他时间没有重叠的时间（例如第四个源）。然后会计算剩余时间间隔的最大交集。管理员接下来就会将它的 UTC 值设定为这个间隔的中间值。

在经过同步之后，管理员通常就会有一个同当前值不一样的新 UTC 值。它虽然可以直接将时钟设置为新值，但是通常情况下这样是不明智的。首先，这可能会要求系统将时间回调，这将意味着同步之后的有些文件可能看起来要比同步之前的有些文件还要早。这种情况下的 *make* 程序运行很有可能会不正常。

即使时钟必须要进行回调，也不应该突然执行这个操作，因为一些程序可能会显示一些信息，同时等待用户进行回应。例如在这种情况下（没有时间间隔的话），一个自动考试程序会在屏幕上给考生显示一道考题，然后立即时间到，同时告诉考生他已经用了太多的时间来回答这道题。

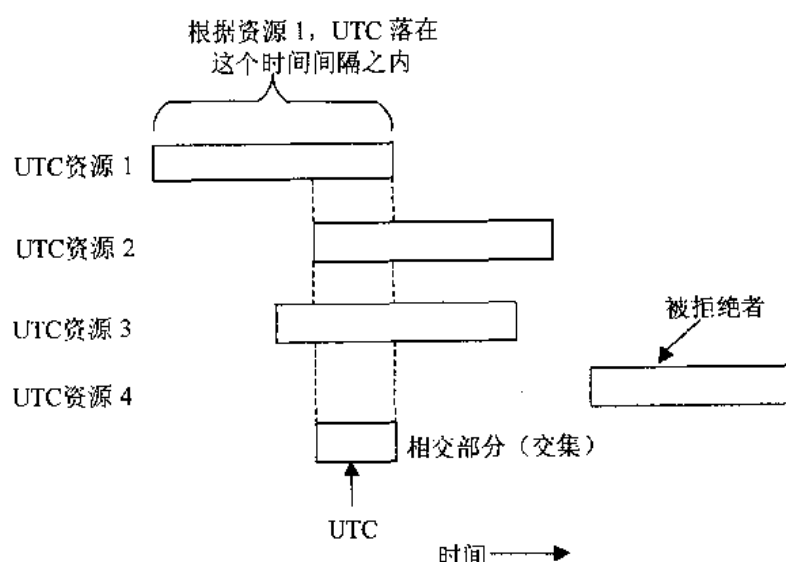


图 10-7 从四个时间源中进行的新 UTC 计算

因而，DTS 可以逐步地完成这种更新。例如，如果时钟慢 5 秒，那么不是在 1 秒钟内连续 100 次增加 10 毫秒，而是在未来的 50 秒内的每一个滴嗒中增加 11 毫秒。

时间服务器主要分两类：本地和全局。本地时间服务器参与它所在信元的时间维护，而全局时间服务器则可以对不同信元内的时间进行同步。这些时间服务器定期相互通信以确保它们的时钟同步。它们也使用图 10-7 中所示的算法来选择新的 UTC。

虽然并不必须，但是如果一个或者多个全局时间服务器通过卫星、无线电或者电话直接同 UTC 源相连接会取得最好的效果。DTS 定义了一个特殊接口，时间提供者接口 (time provider interface) 来定义 DTS 是如何从外部源中获取以及分配 UTC。

10.5 目录服务

DCE 的一个主要目的就是使系统中任何进程都可以访问所有资源，而不用管资源使用者（客户）以及资源提供者（服务器）的相对物理位置。这些资源包括用户、机器、信元、服务器、服务、文件、安全数据等等。为了实现这个目标，DCE 就必须维护一个记录所有资源位置以及提供它们的用户友好名称的目录服务。在本节中将介绍这个服务以及它的运行方式。

DCE 目录服务是按信元组织的。每一个信元中都包含一个信元目录服务 CDS（Cell Directory Service），它记录着信元中资源的名称和属性。这个服务是作为可复制的分布式数据库组织的，它甚至可在系统崩溃的情况下提供良好的性能和健壮性。为了能够操作，每一个信元都必须至少运行一个 CDS 服务器。

每一个资源都有一个唯一的名称，它由信元名和信元内所使用的名字组成。为了定义一个资源，目录服务需要一种定位信元的方法。系统支持两种机制，全局目录服务 GDS（Global Directory Service）和域名系统 DNS（Domain Name System）。GDS 是 DCE 的“亲生”信元定位机制。它使用 X.500 标准。由于许多 DCE 用户都使用 Internet，所以系统也支持 Internet 的标准命名系统 DNS。虽然有一个唯一的资源定位系统（因而有唯一的资源命名语法）是最好的选择，但是出于政治的考虑这种方法就不行了。

图 10-8 中显示了这些部件之间的关系。在图中可以看到目录服务的另外一个部件，全局目录代理 GDA（Global Directory Agent），它主要被 CDS 用来同 GDS 和 DNS 进行交互。当 CDS 需要查找一个远程名称，它会请求 GDA 来完成这项工作。这种设计使 CDS 同 GDS 和 DNS 中所使用的协议无关。同 CDS 和 GDS 一样，GDA 也是一个通过使用 RPC 来进行申请和返回结果的后台进程。

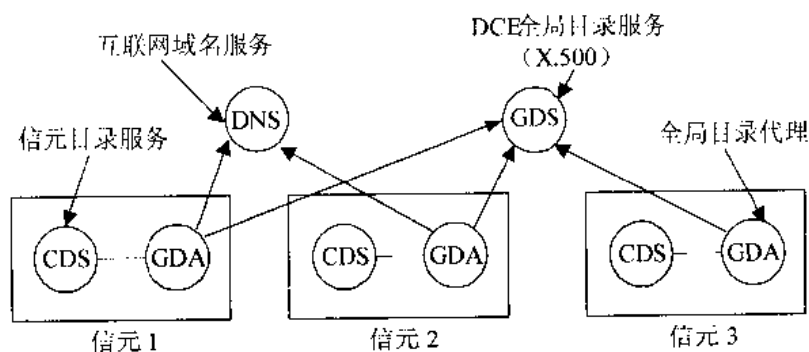


图 10-8 CDS、GDS、GDS 和 DNS 的关系

在下小一节中，将介绍 DCE 中名字的组成。随后还将介绍 CDS 和 GDS。

10.5.1 名字

DCE 中每一个资源都有一个独立的名字。所有名字的集合就构成了 DCE 的名字空间。每一个名字最多可以有五部分，其中的有些部分是可以选择的。图 10-9 中显示了这五个部分。

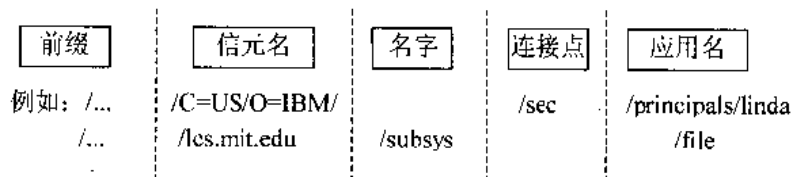


图 10-9 DCE 名字可以有五个部分

第一部分是前缀，它表明这个名字是全局的还是当前信元本地的。前缀/...代表一个全局名字，而前缀/则代表一个本地名字。全局名字中必须要包含信元所需要的名字，而本地名字则一定不是这样。当一个请求到达 CDS 时，它根据请求前缀的不同就可以判断出应该自己处理还是将它发送给 GDA 来让 GDS 进行远程查询。

信元名字既可以使用 X.500 符号也可以使用 DNS 符号进行定义。这两种系统都有十分详细的资料标准，但是对于我们的目标而言只需要对后面的介绍进行说明就足够了。X.500 是国际命名标准。它由电话公司开发，主要目标是为未来客户提供电子电话本（electronic phonebooks）。它可以被用来定义人员、计算机、服务、信元以及其他任何需要唯一命名的实体。

每一个命名实体都有一组用来对它进行说明的属性。这些可能包括它的国家（如 US、GB、DE 等），它的组织（如 IBM、HARVARD、DOD 等）、它的部门（如 CS、SALES、TAX）以及一些更详细的诸如人员编号、管理员、办公室编号、电话号码以及姓名等信息。每一个属性都有一个值。一个 X.500 名字实际上是一个由斜杠分隔的属性值的列表。例如：

/C=US/O=YALE/OU=CS/TITLE=PROF/TELEPHONE=3141/OFFICE=210/SURNAME=LIN/

可能说明的是耶鲁大学计算机系的 LIN 教授。属性 C、O 和 OU 大多用来分别表示名字、国家、组织以及组织信元（部门）。

X.500 的核心思想是只要一个查询提供足够的属性就可以获得一个唯一的目标。在上面的例子中，C、O、OU 和 SURNAME 可以完成工作，但是如果请求者遗忘了名字而记得办公室号码，那么 C、O、OU 和 OFFICE 可能也可以。如果提供了除国家以外的所有属性而希望服务器在整个世界中查找符合条件的结果是不合适的。

DNS 是因特网上主机和其他资源的命名方案。它将整个世界划分为由国家组成的顶级域名，或者用 EDU 代表教育机构，用 COM 代表公司，用 GOV 代表政府，用 MIL 代表军事等。而这些顶级域名又可以不断细分为子域名，如 *harvard.edu*、*princeton.edu* 和 *stanford.edu* 和 *cs.cmu.edu*。X.500 和 DNS 都可以用来指定信元名称。在图 10-9 中的两个例子就是 IBM 的税务部门和 M.I.T 的计算机实验室。

下一个层次的名称经常是一个标准资源的或一个连接点（junction）名称，它非常类似于 UNIX 系统中的一个安装点（mount point），它使得查询在如文件系统或者安全保证系统等不同命名系统之间必须要进行转换。最后才是资源名字自身。

10.5.2 信元目录服务

CDS 管理一个信元的所有名字。它们是分层管理的，虽然在 UNIX 系统中也存在符号链接（称为软链接）。图 10-10 中是一个简单信元命名树的样例。

最高层目录中包含了两个记录 RPC 绑定信息的配置文件 (profile file), 其中一个记录了它们的拓扑非依赖关系, 另一个记录了应用的网络拓扑, 它们对于在客户 LAN 上的服务器选择具有很重要的意义。它也包含了一个指向 CDS 数据库的指针。Hosts 目录记录了信元中所有机器的名字。而在它的子目录中分别记录了主机 RPC 后台程序和缺省配置文件, 同时还提供了 CDS 系统以及其他机器的不同信息。这些数据的共同使用提供了前面所提到的文件系统和安全数据库的连接, 而 *subsys* 子目录则包含了包括 DCE 自己的管理信息在内的所有用户应用。

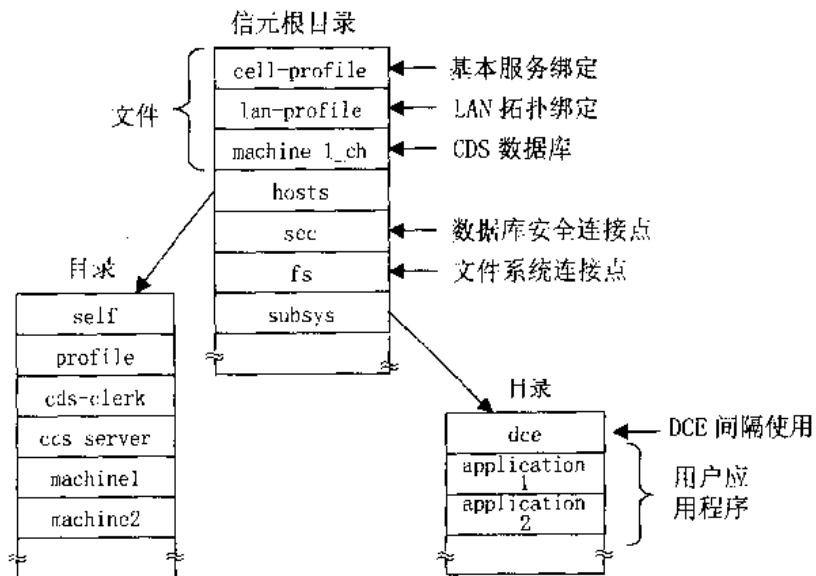


图 10-10 简单 DCE 信元的命名空间

目录系统中大多数原语单元 (unit) 都是 CDS 目录条目 (CDS directory entry), 它由一个名字和一组属性组成。服务的条目包含了服务的名称、它所支持的接口以及服务器的位置。

CDS 中非常重要的一点是它只包含有关资源的信息。它并没有提供对资源自身的访问。例如, 一个 *printer23* 的条目也许说明了它是 20 页/分钟、600 点/英寸的彩色激光打印机, 它位于 Toad 大厅的第二层, 设备的 IP 为 192.30.14.52。任何 RPC 都可以通过使用这种信息来进行绑定, 但是如果要使用打印机, 客户自身还必须要通过 RPC 来进行。

同每一个条目相关联的是一个指出谁可以以什么样的方式 (如谁可以从 CDS 目录中删除条目) 来进行访问的列表。这种保护信息是由 CDS 自身所维护的。具有访问 CDS 条目的权限并不意味着具有可以访问资源的权限。只有管理资源的服务器自身才可以决定谁可以以什么样的方式来访问资源。

相关的条目可以组合起来形成一个 CDS 目录。例如, 为了方便起见, 所有的打印机可以都放在 *printer* 目录中, 其中的每一个条目都用来说明一个或者一组打印机。

CDS 允许对条目进行复制来保证高可靠性和更好的容错性。这种复制的最小单位是目录, 也就是说一个目录中的所有条目要么都被复制, 要么都不被复制。由于这个原因, DCE 中的目录的概念要比 UNIX 中目录的概念重要。CDS 中的目录并不能通过普通的程

序接口建立和删除，而必须通过使用管理程序来完成。

一组目录就形成了票据交换所（clearinghouse），它实际上是一个物理数据库。一个信元有可能会具有多个票据交换所。当一个目录被复制之后，它可能会出现在两个或者更多的票据交换所中。

一个信元的 CDS 可以通过许多服务进行传播，但是系统的设计允许对于任何名称的查找都可以从任何服务开始。从前缀中系统就可以判别所需要的名字是本地的还是全局的。如果它是全局的，那么这个请求就会被传递到 GDA 中以进行进一步的处理。如果这个请求是本地的，那么系统就会在根目录中查找第一个符合条件的部件。因此，每一个 CDS 服务器都有根目录的一个拷贝。根目录所指向的目录无论是在当前服务器的本地还是在其他服务器上都可以继续进行查询以获取这个名字。

在一个信元内使用多目录拷贝造成了一个问题：如何在保证一致性的情况下对它进行更新？DCE 在这里采用了一个比较简单的做法。每一个目录中都有一个拷贝作为主拷贝存在，所有其他目录拷贝都是从拷贝。在主拷贝上可以进行读操作和写操作，而在从拷贝上只能进行读操作。当主拷贝进行更新之后，它会告知从拷贝进行相应的更新。

这种传播可以以两种方式进行。对于所有必须随时保持同步的数据，主拷贝会立即将变化通知所有的从拷贝。而对于不那么关键的数据，对从拷贝的更新则可以稍后进行。这种称为偷懒（skulking）的方案允许许多更新一起以更大、更有效的消息形式进行。

CDS 由两个主要部件实现。第一个，CDS 服务器，是一个运行在服务器机器上的后台进程。它接收所有查询请求，在本地票据交换所中查找，最后返回相应的结果。

第二个，CDS 管理员，是一个运行在客户机器上的后台进程。它的主要功能是截获客户的请求。客户申请查询目录中的数据就是通过 CDS 管理员完成的，它同时会缓存当前结果以供系统今后使用。由于 CDS 服务器定期广播自身所在的位置，所以 CDS 管理员可以知道它们的位置。

图 10-11 (a) 是客户、CDS 管理员以及服务器之间交互的简单样例。为了查找一个名字，客户对它的本地 CDS 管理员执行一个 RPC 调用。然后这个管理员就会查找它的缓存。如果它找到了结果就会立即返回响应。如果没有，那么如图所示，它就会通过网络到 CDS 服务器执行 RPC 调用。在这种情况下，CDS 服务器就会在它的票据交换所中找到所请求的名字，然后再返回给管理员。而管理员在没有把结果返回给客户之前会先将结果进行缓存。

在图 10-11 (b) 中，在信元中有两个 CDS 服务器，但是管理员只知道一个出了错误的服务器的位置。当这个 CDS 服务器发现它没有所需要的目录条目时，它就会查找根目录（请注意所有的 CDS 服务器中都有完整的根目录）以获得正确的 CDS 服务器链接。拥有正确的信息之后，管理员就可以重新进行尝试（消息 4），同时在返回结果之前仍然对它进行缓存。

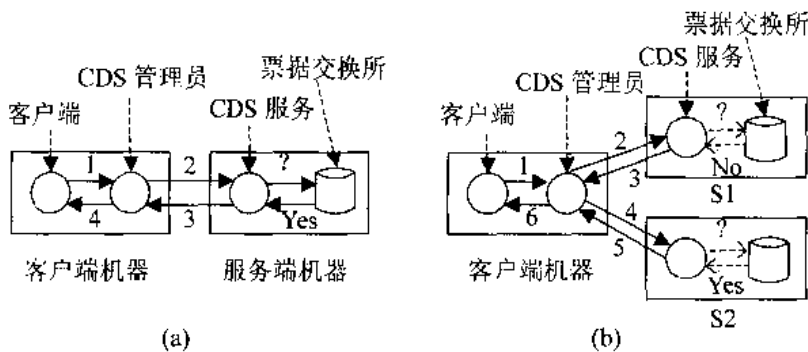


图 10-11 客户查找名字的两个例子

(a) 首先查找的 CDS 服务器包含所需要的名字

(b) 首先查找的 CDS 服务器不包含所需要的名字

10.5.3 全局目录服务

除 CDS 之外，DCE 还有第二种目录服务 GDS——全局目录服务。它可以用来定位远程信元，但是也可以用来存储其他任何目录信息。GDS 的重要性主要在于它是 X.500 的 DCE 实现，而正因为这样，它也可以同任何同样使用 X.500 协议的其他（非 DCE）系统协同工作。X.500 的定义主要根据 ISO 9594 国际标准。

X.500 使用的是面向对象的信息模型。X.500 目录中所存储的每一个条目都是一个对象。一个对象可以是一个国家、一个城市、一个人、一个信元或者一个服务器。

每一个对象都有一个或者多个属性。每一个属性都有一个类型和一个值（有时候是多个值）组成。在表现形式上，类型和值是通过等号分开的，如 C=US 就表示类型是 country，而值为 United States。

对象可以组成类（class），在一个特定类中的所有对象都“参考”同样的对象。一个类可以拥有强制属性，（如一个邮局对象的邮政编码），和可选属性，（如一个公司对象的 FAX 号码）。对象类的属性总是强制的。

X.500 的命名是层次结构的。图 10-12 中给出了一个简单样例。在图中的根目录下只有两个条目，一个是国家（US），另外一个组织（IBM）。X.500 并没有决定将哪个对象存放在哪个位置，而是由与它相关的注册机构决定的。例如，一个新成立的公司，Invisible Graphics，希望将自己注册到世界范围的树的 C=US 下。它就必须到 ANSI 查找这个名字是否已经在使用，如果没有的话它就可以进行申请并且缴纳注册费用。

命名树上的路径是通过一系列用斜杠隔开的属性组成的，就像前面所看到的那样。在我们的例子中，San Francisco Joe Deli 的 Joe 应该是：

/C=US/STATE=CA/LOC=SF/O=JOE'S-DELI/CN=JOE/

其中 LOC 指出一个位置，而 CN 是对象的（普通）名称。用 X.500 行话来说，路径的每一个部分都称为 RDN（Relative Distinguished Name: 相对区分名称），而全路径称为 DN（Distinguished Name: 区分名称）。从任何对象产生的 RDN 都必须是确定的，而从不

同对象产生的 RDN 则有可能是相同的。

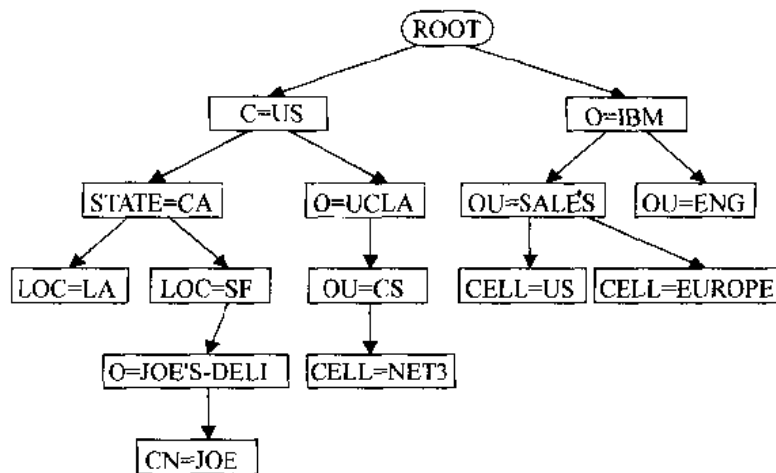


图 10-12 一个简单的 X.500 目录信息树

除了普通对象之外，X.500 还支持可以支持命名其他对象的别名（aliases）。别名非常类似于文件系统的符号链接。

X.500 目录信息树的结构与属性是由它的方案（schema）所决定的，它主要包含在以下三个表中：

- （1）结构规则表——树中每个对象的位置；
- （2）对象类表——类的继承关系；
- （3）属性表——每一个属性的名字和类型的定义。

结构规则表是树的一个表的形式说明，它主要说明了哪个对象是其他哪个对象的孩子。对象类表说明了对象之间的继承层次关系。例如，我们可以想像一下以语音和 FAX 为子类的电话号码类。

图 10-12 中没有包含任何有关对象继承层次关系的信息，这主要是因为组织是在国家、地区以及根之下的。该图中所说明的结构将在结构规则表中得到反映。对象类表可能由组织信元设为组织的一个子类，而信元又作为组织信元的一个子类，但是这种信息也不能从图中得到。除了说明一个类是从哪个类中得到的之外，对象类表还列出了它的唯一对象标识符以及它的强制和可选属性。

属性表说明了每个属性所能够取得的值、它们所占用的内存以及它们的类型（也就是整数、布尔型或实型）。属性在 OSI ANSI.1 符号中进行了说明。DCE 提供了一个从 ANSI.1 到 C（MAVROS）的编译器，它很类似于 RPC 存根的 IDL 编译器。这个编译器真正用于建立 DCE；而在一般情况下用户是不会遇到它的。

每一个属性都可以被标识为公共（PUBLIC）、标准（STANDARD）或敏感（SENSITIVE）。同时也可以为每一个对象建立一个访问控制表来指出哪个用户可以读而哪个用户可以写这些目录中的哪些属性。

系统也支持称为 XOM（X/Open Object Management）的 X.500 标准接口。程序访问 GDS 的通常方法是通过 XDS（X/Open Directory Server）库实现的。当调用 XDS 过程时，由所调用的过程来检查所操作的是一个 CDS 条目还是一个 GDS 条目。如果是 CDS 条目的话，它就

直接进行工作。而如果是 GDS 条目的话，它将调用必要的 XOM 调用来完成工作。

XDS 接口非常小，只有 13 个调用（而 DCE 的 RPC 有 101 个调用，它的操作手册一共有 409 页）。在这些调用中的五个主要用来建立并初始化客户同服务器之间的连接。而八个实际操作目录的调用如表 10-11 所示。

表 10-11 操作目录对象的 XDS 调用

调用	说明
Add_entry	给目录中增加一个对象或别名
Remove_entry	从目录中删除一个对象或别名
List	列出指定对象之下的所有直接对象
Read	读取指定对象的属性
Modify_entry	改变指定对象的基本属性
Compare	在特定属性与指定值之间进行比较
Modify_rdn	改变一个对象的名称
Search	在树的一个部分中查找一个对象

前两个调用分别从目录树中增加和删除对象。每一个调用都通过指定全路径来避免混淆。*list* 调用可以列出指定对象之下的所有直接对象。*Read* 和 *modify_entry* 读取或者填写指定对象的属性，将它们从对象树拷贝给调用者或者正相反。*Compare* 通过把一个特定对象的特定属性与一个给定值比较，来检查它，确定它们是否匹配。*Modify_rdn* 改变对象名称，如将 *a/b/c* 改为 *a/x/c*。最后，*search* 从一个指定对象开始，在其下的对象树中查找符合条件的对象。

所有这些调用的操作都首先判定需要使用 CDS 还是 GDS。X.500 名字由 GDS 进行处理，而 DNS 或混合名字由 CDS 进行处理，如图 10-13 所示。首先让我们看一下使用 X.500 格式的名字查询。XDS 库发现它需要查找一个 X.500 名字，所以它调用 DUA (Directory User Agent: 目录用户代理)，一个链接到用户代码中的库。它主要处理同处理 CDS 缓存的 CDS 管理员非常类似的 GDS 缓存。用户对于 GDS 缓存的控制能力和次数比对 CDS 缓存的控制要多，例如可以指定缓存哪些项目。如果没有必要获取最新数据的话，它们甚至可以旁路 DUA。

与 CDS 服务器很类似，系统使用 DSA (Directory Server Agent: 目录服务代理) 来处理来自本信元以及远程信元及 DUA 的申请。如果一个申请由于信息未准备好而无法完成，DSA 或者将申请提交到正确的信元，或者要求 DUA 来完成申请。

除了 DUA 和 DSA 进程之外，系统中还有独立的存根进程处理在 OSI 协议传输层上使用 OSI ASCE 和 ROSE 协议处理广域通信。

现在看一个 DNS 或者混合名称的查询。XDS 通过向 CDS 管理员执行 RPC 来检查它是否进行过缓存。如果没有进行缓存就请求 CDS 服务器。如果 CDS 服务器认为这个名字为本地的，它就执行查询。如果这个名字属于一个远程信元，那么它就会要求 GDA 来执行工作。GDA 检查远程信元的名字来判别它是否作为一个 DNS 名字或者 X.500 名字来定义。在前一种情况下，它要求 CDS 服务器寻找该信元内一个 CDS 服务器；在后一种情况下它使用 DSA。但是不管怎样，名字查询都是一种复杂的操作。

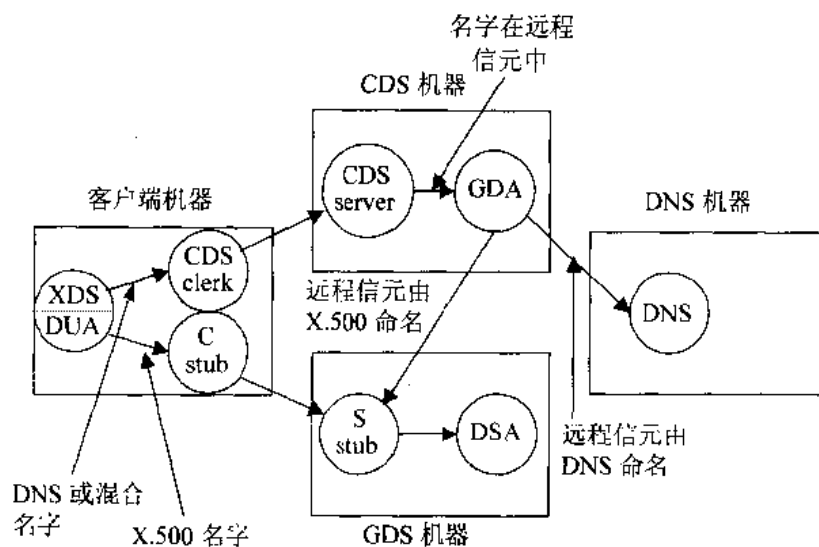


图 10-13 名字查询服务器的调用处理机制

10.6 安全服务

在大多数分布式系统中，安全都是一个关键问题。系统管理员有可能对谁可以使用哪些资源（如大学生以下不能使用彩色激光打印机）有明确的想法，而且许多用户也不想将他们自己的文件和邮箱被别人看到。这些问题在过去的分时系统中也存在，但是这些问题可以通过让内核管理所有资源轻易解决。在由潜在不可信机器以及不安全的网络组成的分布式系统中，这种解决方案就不可行了。然而 DCE 提供了良好的安全性。在本节中将介绍这种安全性是如何实现的。

首先介绍一些主要术语。在 DCE 中，一个主要程序（principal）是一个需要安全通信的用户或进程。管理员、DCE 服务器（如 CDS）以及应用服务器（如银行系统中自动柜员机的软件）都可以认为是主要程序。为了方便起见，具有同样访问权限的主要程序可以合成组。每一个主要程序都有且仅有一个与之相联系的 UUID（Unique User Identifier：唯一用户标识符）。

认证（Authentication）是判定主要程序是否真正与它所声明的身份相符的过程。在分时系统中，一个用户通过输入他的用户名和密码来登录。通过对密码的检查可以确定该用户是否是真正的用户。当一个用户成功登录之后，内核会记录用户的身份，同时根据它来判断是允许还是拒绝对文件或其他资源的访问。

在 DCE 中必须采用不同的认证机制。当一个用户登录时，登录程序会通过使用认证服务器来认证用户的身份。后面将对协议进行介绍，但是目前只需要知道这个过程不牵涉在网络上发送密码的过程。DCE 认证过程使用 M.I.T.(Kohl, 1991; 以及 Steiner 等, 1988) 开发的 Kerberos 系统。Kerberos 是基于 Needham 和 Schroeder (1988) 思想的。其他的认证方法可以参考 (Lampson 等, 1992; Wobber 等, 1994; 以及 Woo 和 Lam, 1992)。

一旦用户经过了认证就产生了该用户可以以什么样的方式访问哪一种资源的问题。

这个问题称为授权(authorization)。在 DCE 中, 授权是通过同每一种资源相关联的 ACL (Access Control List: 访问控制表) 实现的。ACL 可以定义哪个用户、组和组织可以并以何种方式访问资源。资源的粒度可以从一个文件到数据库的一个条目。

DCE 的保护同信元的结构联系紧密。每一个信元都有一个本地主要程序必须信任的安全服务 (security service)。认证服务器是安全服务的一个部分, 而安全服务在称为注册表 (registry) 的数据库中保存了密钥、口令以及其他同安全有关的信息。由于不同的信元可能分别属于不同的公司, 所以一个信元到另外一个信元的通信通常需要复杂的协议进行支持, 同时只有两个信元建立共享密钥之后这种工作才可以进行。为了简单起见, 将下面的讨论限制在一个信元内部。

10.6.1 安全模式

在本节中将简要回顾加密 (cryptography) 的一些基本原理, 它是发送秘密消息的基本技术, 同时也是 DCE 在这方面的必要前提。首先假设客户与服务器双方, 希望通过不安全的网络进行通信。这就意味着即使有一个侵入者 (如一个窃听者) 偷窃到消息, 他也不能理解消息的内容。更进一步来说, 如果侵入者通过模拟客户或者如果侵入者记录从客户发出的合法信息并且将它们返回给服务器, 那么服务器应该能够察觉到并拒绝这些消息。

图 10-14 中显示了传统的加密模式。客户拥有称为明文 (plaintext) 的未加密的消息 P , 这些消息通过以密钥 K 为参数的加密算法进行转换。加密过程可以由客户、操作系统或者由专门硬件来执行。加密的结果是消息 C , 称为密文 (ciphertext), 它对于任何不具有密钥的用户来说都是不可解读的。当密文到达服务器时通过使用 K 进行解密, 其结果就是原来的明文。表示加密的通用表示法为:

$$\text{Ciphertext} = \{ \text{Plaintext} \} \text{Key}$$

也就是说花括号中是明文, 而所使用的密钥则写在它的后面。

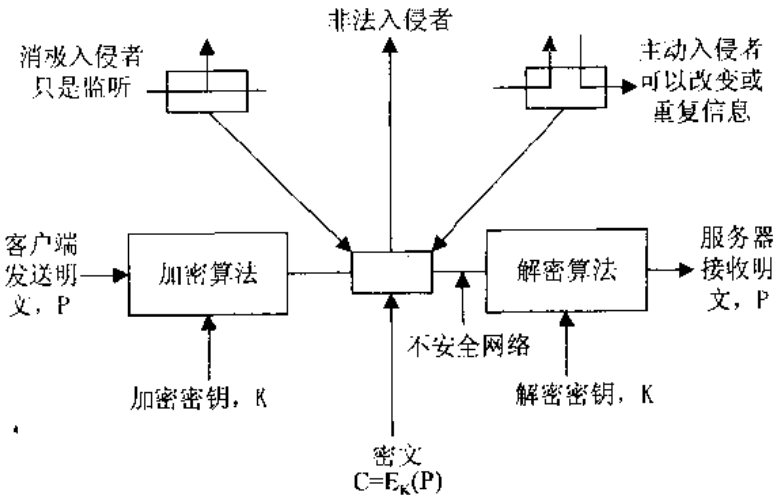


图 10-14 客户给服务器发送加密消息

实际中也存在客户和服务使用不同密钥 (如公共密钥系统) 的加密系统, 但是由

于 DCE 中没有使用这样的系统，所以在这里不进行深入讨论。

为了避免混淆，进行一些假设来保证模式的清晰度。假设网络是完全不安全的，而且一个侵入者可以获取任何网络上传送的消息，同时可能将这些消息删除。侵入者还可能按自己的需要强制注入新消息和重复旧消息。

虽然我们假设大多数服务器是基本安全的，但是在这里明确假设安全服务器（包括它的所有磁盘）都存放在安全的守卫良好的房间中，在这种情况下没有任何侵入者可能尝试去篡改它。同时，系统允许安全服务器保存每一个用户的密码，即使这些密码不能在网络上进行传送。系统也假设用户不会忘记他们的密码，或者很少将他们的密码遗忘在自己的终端前。最后，假设系统的时钟是大致同步的，例如通过使用 DTS。

在这种充满敌意的环境下，从开始设计就设定了一些要求。其中最重要的如下。

第一，在任何时候用户的密码都不能以明文形式在网络上传送或者保存在普通服务器上。这种要求包括向授权服务器传送用户口令来进行认证的情况。

第二，用户口令不应该在用户机器上长期存储(不要超过几个微秒)，这样可防止系统崩溃、核心卸出时可能引起的暴露。

第三，认证必须是双向的。也就是说服务器不仅要认证客户是否真实，而且客户也需要认证服务器是否真实。这种要求可以防止一个侵入者截获客户消息后伪装成文件服务器的情况出现。

最后，系统中必须存在防火墙。如果一个密钥发生了失密（泄漏），那么发生的损失必须是可以控制的。这种要求可以通过为特殊需要建立临时密钥等方法来实现，这种临时密钥的生命周期很短，而且可以在大部分工作中进行使用。即使这种密钥发生了失密，潜在的损失也是可以控制的。

10.6.2 安全部件

DCE 安全系统由几个服务器和程序组成，图 10-15 中列出了其中最重要的几个。

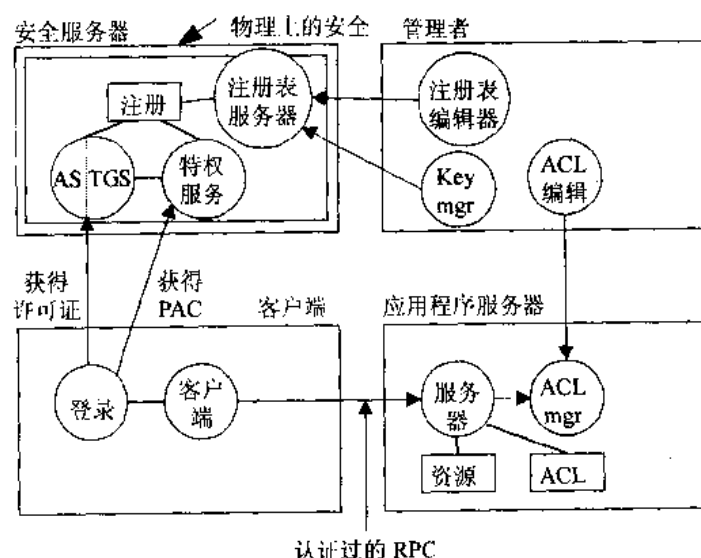


图 10-15 单个 DCE 信元中安全系统的主要部件

注册表服务器 (registry server) 管理安全数据库、注册表(其中存放着所有主要程序、组以及组织的名字)。对于每一个主要程序而言,它都存放了帐户信息、所从属的组和组织、主要程序是服务器还是客户以及其他一些信息。注册表中还包含了每个信元的策略信息(长度、格式、口令的生命周期及其相关信息)。注册表可以被认为是 UNIX 系统的密码文件 (*/etc/passwd*) 的扩展。它可以由系统管理员通过使用注册表编辑程序来进行修改。这些修改包括增加或删除主要程序、更改密钥等等。

认证服务器 (authentication server) 主要在用户登录和服务器引导时使用。它检验主要程序所声称的身份之后发出一种许可证 (ticket) (下面将进行说明) 来允许主要程序继续剩余的认证而无须重复提供口令。当认证服务器主要进行许可证确认而不是用户认证时也可以称作许可证授权服务器 (ticket granting server), 但是这两种功能一般都由同一个服务器完成。

特权服务器 (privilege server) 为通过认证的用户发布称为 PACs (Privilege Attribute Certificates) 的特许属性证书。PACs 是包含了主要程序的身份、组成员关系以及组织成员关系的加密消息, 它的使用可以让服务器立即相信主要程序的身份而不用提供任何其他身份证明信息。这三个服务器都运行在安全服务器机器上, 而这台机器是存放在处于严密保卫的房间中的。

登录程序 (login facility) 是在用户登录过程中询问他们的名称和口令的程序。它使用认证和特权服务器来完成用户的登录以及为他们收集必须的许可证和 PACs。这就是它的主要工作。

一旦一个用户登录到系统中之后, 它就可以使用通过认证的 RPC 来执行一个可以同服务器进程进行安全通信的客户进程。当经过认证的 RPC 请求到达时, 服务器就使用 PAC 来判断用户的身份, 同时检查它自己所保存的访问控制表以判断是否能够执行用户所请求的操作。服务器都有用来保护自己对象的访问控制表管理器。通过使用访问控制表编辑程序不仅可以在列表中增加或删除用户, 而且也可以增加或删除用户所具有的权限。

10.6.3 许可证与鉴别码

在本节中将说明认证以及特权服务器是如何工作的以及它们是如何在不安全的网络上以安全的方式允许用户登录到系统中的。出于篇幅的考虑, 这里仅限于最基本和最关键的部分, 忽略了大多数的选项和变量。

用户都有一个只有他自己和注册表知道的密钥。它是通过将用户的口令使用一个单向 (也就是不可逆) 函数计算得到的。服务器也有密钥。为了加强它们的安全性, 对于这些密钥的使用非常少, 只有当用户登录或者服务器启动时才会用到。在这之后的认证都是通过许可证和 PAC 完成的。

一个许可证是由认证服务器或许可证授权服务器发出的加密数据结构, 它主要用来向一个特定服务器证明持有者是一个具有特定身份的客户。许可证有许多选项, 但是通常所讨论的大多许可证都非常类似于下面的形式:

```
ticket=S, {session-key, client, expiration-time, message-id}Ks
```

其中 *S* 是许可证的目标服务器。花括号中所包含的信息是经过服务器私有密钥 *Ks* 加密的。加密的域包括一个临时会话密钥、用户的身份、许可证的有效期限以及一个消息标

标识符或者用于将请求与应答建立联系的当前 (nonce) 标识。当服务器使用自己的私有密钥解密这些信息之后, 它就可以获得用来同客户进行通信的会话密钥。在下面的说明中, 将忽略除会话密钥和客户名之外的所有加密许可域。

在有些情况下许可证和 PAC 是联合使用的。许可证建立了发送者的身份 (作为 ASCII 字符串), 然而 PAC 则给出了同特定主要程序相联系的 user-id 和 group-id 的数字值。许可证是由认证服务器或许可证授权服务器产生的 (同一个服务器), 而 PAC 则是由特权服务器产生的。

在许多情况下消息的保密并不一定必要, 只要侵入者不能伪造或者修改它们就可以了。为了达到这个目的, 可以通过给明文后面附加鉴别码来防止主动侵入者更改它们。一个鉴别码 (authenticator) 是至少包含以下信息的加密数据结构:

authenticator={sender, MD5-checksum, timestamp}K

其中校验和 (checksum) 算法 MD5 (Message Digest 5) 的特性为: 当使用 128 位校验和时, 从计算角度讲改变消息的内容同时保证校验和 (有加密算法保护) 不变是办不到的。时间戳可以帮助接收者检测到过期的鉴别码。

10.6.4 认证过的 RPC

从登录到第一个认证的 RPC 之间典型的有 5 个步骤。每一个步骤都由从客户到某个服务器的消息以及从服务器到客户的应答组成。图 10-16 中总结了这些步骤。为了简单起见, 大多数的域, 包括信元、消息 ID、生命周期以及标识符都被忽略了。重点是同安全有关的基本原则, 它主要包括密钥、许可证、鉴别码以及 PAC。

主要程序

A: 认证服务器 (处理认证)

C: 客户端 (用户)

P: 特权服务器 (发布 PAC 特权)

S: 应用程序服务器 (处理实际工作)

T: 许可证授权服务器 (颁发许可证)

步骤 1: 客户端使用口令从许可证授权服务器获得许可证

$C \rightarrow A: C$ (只有客户名是明码)

$C \leftarrow A: \{K_1, \{K_1, C\}K_A\}K_C$

步骤 2: 客户端使用原来的许可证为访问特权服务器获取许可证

$C \rightarrow T: \{K_1, C\}K_A, \{C, \text{MD-5 checksum, Timestamp}\}K_1$

$C \leftarrow T: \{K_2, \{C, K_2\}K_P\}K_1$

步骤 3: 客户端请求特权服务器提供初始 PAC

$C \rightarrow P: \{C, K_2\}K_P, \{C, \text{MD-5 checksum, Timestamp}\}K_2$

$C \leftarrow P: \{K_3, \{PAC, K_3\}K_A\}K_2$

步骤 4: 客户端请求许可证授权服务器提供可在 S 上使用的 PAC

$C \rightarrow T: \{K_3, \{PAC, K_3\}K_A, \{C, \text{MD-5 checksum, Timestamp}\}K_3$

$C \leftarrow T: \{K_4, \{PAC, K_4\}K_S\}K_3$

步骤 5: 客户端认可使用应用服务器的一个关键字

$C \rightarrow S: \{PAC, K_4\}K_S, \{C, \text{MD-5 checksum, Timestamp}\}K_4$

$C \leftarrow S: \{K_5, \text{Timestamp}\}K_4$

图 10-16 从登录到 RPC 认证结束的五個简单步骤

当一个用户坐到 DCE 终端面前，登录程序就会询问他的登录名。这个程序然后就会将用户名采用明文的形式发送到认证服务器。认证服务器在注册表中根据这个用户名查找该用户的密钥。它随后产生一个随机数来作为会话密钥并返回根据用户密钥 (K_c) 加密的包括第一个会话密钥 (K_1) 在内的加密消息，同时还要产生一个在后面许可证授权服务器使用的许可证。这些消息在图 10-16 的步骤 1 中进行了说明。需要注意的是本图中显示了 10 个消息，其中每一个消息的发出者和接收者都在消息前面进行了说明，箭头由发出者指向接受者。

当加密的消息到达登录程序之后，系统就会向用户询问口令。这个口令会立即使用从口令产生密钥的单向函数来生成密钥。当密钥 K_c 从密码中生成之后，用户所输入的口令就会立即从内存中清除。这个过程最大程度上减少了由于客户端崩溃而造成口令泄漏的可能性。然后就可以根据这个密钥对认证服务器发来的消息进行解密以获取会话密钥和许可证。当这一切完成之后，用户的密钥也可以从内存中清除。

在这里需要注意的是即使侵入者截获返回消息，也不可能对这个消息进行解密，从而也不能获得其中所包含的会话密钥和许可证。当然在足够长的时间之后，侵入者最终可能会破译这个消息。但即使是这样，由于会话密钥和许可证的生命周期都比较短，所以造成的危害是有限的。

图 10-16 的第二个步骤中，客户将许可证发送到许可证授权服务器（实际上是具有不同名称的认证服务器）以申请同特权服务器进行会话的许可证。除了第一个步骤中的初始认证之外，以认证的方式访问任何服务器都需要经过该服务器私有密钥加密的证明。这些许可证可以如步骤 2 从许可证授权服务器中获得。

当一个许可证授权服务器获得消息之后，它就使用自己的私有密钥 K_s 来解密这个消息。当它发现会话密钥 K_1 之后，它就会在注册表中查找并认证最近为客户 C 所指定的密钥。由于只有 C 知道 K_c ，所以许可证授权服务器认为只有 C 可以解密在步骤 1 中所返回的消息，因而这个请求也一定是从 C 来的。

请求中也包含了一个鉴别码，它是一个基本的包含时间戳的校验和加密，可以用来保护消息的剩余部分，包括信元名称、请求以及图中所没有显示的其他域。这种方案使侵入者不可能在不被人知道的情况下修改消息，因而也就不需要客户对整个消息进行加密，这对于长度较大的消息来说加密代价是非常大的。（在本例中消息并不长，但是为了简单起见总是使用鉴别码。）鉴别码中的时间戳防止了侵入者截获消息，然后再进行回放，因为认证服务器只有在鉴别码保持最新的情况下才会处理这个请求。

在本例中我们在每一个步骤都产生并使用一个新的会话密钥。这种做法虽然在实际中没有必要，但是协议允许这种做法，而且这种做法在系统时钟同步良好时可以允许密钥的生命周期很短。

当拥有访问特权服务器的许可证之后，客户现在就可以申请 PAC。它只包含用户的用户名（ASCII 码）的许可证不同的是，PAC 中包含了用户的身份（以二进制码形式说明 UUID）以及它所从属的所有组的一个表。这些信息的重要性在于一些资源（如特定的打印机）经常可以被一个组（如市场部）的所有成员使用。PAC 就证明了其中所包含的名字从属于其中所列出的组。

步骤 3 中获得的 PAC 是用认证服务器/许可证授权服务器的密钥加密的。这种选择的

重要性在于在整个会话过程中，客户可能会需要不同 PAC 以访问不同应用服务器。为了获得一个应用服务器的 PAC，客户就需要执行步骤 4。在这里 PAC 被送到许可证授权服务器，由它来对 PAC 进行解密。根据解密的结果，许可证授权服务器立即就可以确认这个 PAC 是否合法。然后它就可以根据客户的服务器选择重新进行加密，在这种情况下为 S。

需要注意的是许可证授权服务器并不一定需要知道 PAC 的格式。在步骤 4 中它所需要进行的全部工作就是使用它自己的密钥来对一些密文进行解密，然后再使用它从注册表中获得的另外一个密钥进行加密。在这个过程中，许可证授权服务器可以发布一个新的会话密钥，但是这种方式是可选的。如果客户后来需要其他服务器的 PAC，那么它就可以根据需要使用原始的 PAC 重复步骤 4，每一次都指定所需要访问的服务器。

在步骤 5 中，客户将新的 PAC 发送给应用服务器，应用服务器对 PAC 进行解密以获取加密鉴别器以及用户的 ID 和组的密钥 K_s 。服务器的回应使用最后的密钥，这个密钥只有服务器和客户知道。通过使用这个密钥，客户和服务器现在就可以进行安全通信了。

这个协议是比较复杂的，但是复杂性主要是因为它被设计为可以抗拒各种不同的可能攻击 (Bellovin 和 Merritt, 1991)。它还有许多这里没有讨论的特性和选项。例如，它可以被用来同位于一个远程信元中的服务器建立安全连接，其中的每个 RPC 都可能需要通过几个不受信任的信元，同时也可以向客户进行服务器认证以防止欺骗 (spoofing)，它是一个侵入者伪装成一个受信任的服务器器的情况。

一旦一个受信任的 RPC 建立成功，就需要由客户和服务器来决定采用什么样的保护机制。在一些情况下，客户认证或者相互认证就足够了。而在另一些情况下需要对每一个包进行认证以防止篡改。最后，当需要实现企业级安全时，所有的双向通话都可以进行加密。

10.6.5 访问控制表 (ACL)

DCE 中的资源都可以用一个访问控制表保护，它的模型是根据 POSIX 1003.6 标准建立的。ACL 说明了谁可以以何种方式访问资源。ACL 由 ACL 管理器 (ACL manager) 所管理，它是集成到每一个服务器的库过程。当一个请求到达控制着 ACL 的服务器时，它将客户的 PAC 进行解密以获得它的 ID 和组，根据以上信息以及所需要进行的操作，ACL 就可以调用 ACL 管理器来判断应该接受还是拒绝访问。每一个资源都支持多达 32 种操作。

然而许多服务器都使用标准的，而不倾向于数学的 ACL 管理器。这种服务器将资源分为了两大类：简单资源，如文件和数据库条目；和包含简单资源的容器，如目录和数据库表。它对信元内部用户和外部用户进行了区分，同时还为每一类 (category) 都细分为所有者、组和其他。因而可以指定所有者可以执行任何操作，所有者组的本地成员几乎可以执行任何操作，而其他信元的未知用户则不能进行任何操作等。

系统支持七种标准权限：读 (read)、写 (write)、执行 (execute)、改变 ACL (change-ACL)、容器插入 (container-insert)、容器删除 (container-delete) 和测试 (test)。前三个在 UNIX 中也有。Change-ACL 权限允许用来改变 ACL 本身。而两个容器权限也非常有用，如它们可以控制谁可以增加或删除目录中的文件。最后，测试允许将给定值同资源进行比较而不用暴露资源本身。例如，一个密码文件条目可能会允许用户请求一个给

定的密码是否匹配，但是不会暴露密码文件条目自身。图 10-17 中给出了 ACL 的一个简单例子。

sp_acl_data	ACL 类型
/.../C=NL/O=VU/OU=CS/	缺省信元
user:ast:rwxcdt	缺省信元中的用户
user:bal:rwxidt	
group:staff:rwx	缺省信元中的组
group:students:rt	
other:t	缺省信元中的其他用户
foreign_user:jennifer@/.../ cs.nyu.edu:rt	其他信元中的用户
foreign_group:staff@/.../ cs.yale.edu:rw	

图 10-17 一个 ACL 例子

在这个例子中，就像在所有的 ACL 中一样指定了 ACL 的类型。这个类型将 ACL 有效地分成基于类型的不同类。其后指定了缺省信元。随后是缺省信元中两个特殊用户、两个特殊组以及所有其他用户的权限。最后还可以指定其他信元中用户和组的权限。如果一个不符合表中所列出任何情况的用户申请访问，它将会被拒绝。

系统为用户的增、删和权限的增、删、改提供了 ACL 编辑程序。为了使用这个编辑程序对访问控制表进行修改，该用户必须具有改变访问控制表的权限，由样例图 10-17 中的代码 *c* 所指明。

10.7 分布式文件系统

在 DCE 环境中要研究的最后一个部件是 DFS (Distributed File System: 分布式文件系统) (Kazar 等, 1990)。它是一个世界范围的文件系统，允许 DCE 系统中任何进程访问它们被授权访问的所有文件，即便进程和文件分处于位置很远的不同信元也是如此。

DFS 主要由两个部分组成：局域部分 (local part) 和广域部分 (wide-area part)。局域部分是一个称为 Episode 的单节点文件系统，它非常类似于单一机器上的标准 UNIX 文件系统。一个 DCE 系统的配置可能会要求所有的机器都运行 Episode 而不是 (或者附加在) 标准的 UNIX 文件系统。

DFS 的广域部分的作用是将所有单个文件系统连结在一起形成跨越许多信元的无缝广域文件系统。它是从 CMU AFS 系统中派生出来的，因而也与之有很大的关系。如果需要更进一步了解 AFS 的信息，请参考 (Howard 等, 1988; Morris 等, 1986 以及 Satyanarayanan 等, 1985)。如果需要 Episode 资料，请参考 (Chutani 等, 1992)。

DFS 是一个 DCE 应用，因而就可以使用 DCE 的所有其他功能。特别的，它可以使用线程来实现多文件同时访问、使用 RPC 来进行客户与服务器之间通信、使用 DTS 来同步服务器的时钟、使用目录系统让文件服务器定位以及用使用论证和特权服务器来保护文件。

从 DFS 的角度来看，每一个 DCE 节点既有可能是一个文件客户，也有可能是一个文件服务器，还有可能同时具有客户和服务器的功能。文件客户是使用其他机器上的文件系

统的机器。为了提高性能，文件客户中拥有可以存储最近使用过的文件片的缓存。一个文件服务器是一台具有硬盘、为本机也有可能为其他机器提供文件服务的机器。

DFS 中有许多值得一提的特性。它有同 CDS 相一致的统一命名，所以文件的名称是位置无关的。管理员可以将文件从一个文件服务器移动到本信元内的另外一个文件服务器而不需要对用户程序进行任何修改。也可以通过复制 (replicated) 文件来进行负载均衡以及在文件服务器崩溃的情况下保持高可用性。系统中也有一个工具可以自动向服务器 (包括工作站在内) 分发执行程序以及其他使用率很高的只读文件的最新版本。

Episode 是对 UNIX 的文件系统进行重新编写得到的，它可以在任何具有硬盘的 DCE 机器上替代 UNIX 文件系统。它可以正确支持写文件之后立即对它进行读取情况下的 UNIX 单系统文件语义。与 NFS 不同，读操作能看到刚写入的值。它同 POSIX 1003.1 系统调用标准协调一致，同时也支持 POSIX 兼容的使用 ACL 的访问控制，而这种访问控制正是许多系统得以实现灵活保护机制的关键。虽然在崩溃之后不需要执行 UNIX 的 *fsck* 程序来修复文件系统，但它的设计也支持系统的这种快速恢复。

由于许多站点可能不希望仅仅为了运行 DCE 而修改它们现有的文件系统。DFS 的设计也提供了同 4.3 BSD, System V, NFS, Episode 以及其他文件系统的无缝集成。但是在这种情况下，DFS 的一些特性，如 ACL 的保护机制，在非 Episode 服务器所支持的那些文件系统中就无法实现。

10.7.1 DFS 接口

DFS 的基本接口 (有意地) 同 UNIX 非常类似。文件能够以通常的方式进行打开 (opened)、读取 (read) 和写 (written)。而且大多数现有软件都可以简单地通过使用 DFS 库进行重新编译后则可立即工作。而对于远程文件系统的装载也是可行的。

/目录仍然是本地根目录，而类似于 */bin*、*/lib* 和 */usr* 仍然表示本地执行代码、库和用户文件，就如同没有 DFS 存在时一样。根目录下的一个新条目是 */...*，它代表全局根目录。DFS 系统 (隐含为全局) 中的每一个文件都有一个唯一路径，这个路径从全局根目录开始，包括了它所在的信元名称以及它在信元内部的名称。在图 10-18 (a) 中我们可以看到一个文件，*january*，是如何使用因特网信元名称进行全局定位的。图 10-18 (b) 中我们则可以看到使用 X.500 信元名称的该文件的名称。这些名字在系统中的任何地方都是有效的，而不论使用这个文件的进程所处的信元如何。

(a) 全局文件名字 (因特网格式)

`/.../cs.ucla.edu/fs/usr/ann/exams/january`

(b) 全局文件名字 (X.500 格式)

`/.../C=US/O=UCLA/OU=CS/fs/usr/ann/exams/january`

(c) 全局文件名字 (信元内部)

`/.../fs/usr/ann/exams/january`

(d) 全局文件名字 (文件系统内部)

`/usr/ann/exams/january`

图 10-18 同一文件的四种命名方法

在任何地方都使用全局名字是非常冗长的，所以系统中也提供了一些捷径 (Shortcuts)。一个以 `././fs` 开始的名字意味着这个名字从当前信元的 `fs` 开始 (它实际上是本地文件系统装载到全局 DFS 树的位置)，如图 10-18(c) 所示。这种用法可以通过图 10-18(d) 的方法进一步简化。

与 UNIX 不同的是，DFS 的保护机制采用 ACL 而不是三组 RWX 位机制，至少在 Episode 所管理的文件是这样的。每一个文件都有一个 ACL 来说明谁可以以什么样的方式使用它。除此之外，每一个目录都有三个 ACL。这些 ACL 分别给出了目录自身、目录中的文件以及目录中的目录的访问权限。

DFS 中的 ACL 是由 DFS 自己管理的，因为目录是 DFS 的对象。一个文件或目录的 ACL 是由一组条目组成的。每一个条目的信息或者这些信息可能是所有者、所有者所在组、其他本地用户、外部 (也就是信元之外) 用户或组、或者其他类型，如未经认证用户。对于每一个条目，所有允许的操作都从集合：读 (read)、写 (write)、执行 (execute)、插入 (insert)、删除 (delete) 以及控制 (control) 中得到。前三个同 UNIX 系统中完全一致。插入和删除只有对目录才有用，而控制只有对 I/O 设备对象的 IOCTL 系统调用才有用。

DFS 支持四个层次的聚集 (aggregation)。最底层是单个文件。这些单个文件可以通过普通方式放到目录中。许多目录可以组成文件集 (fileset)。最后，许多文件集可以组成一个磁盘分区 (在 DCE 行话中就是聚集)。

一个文件集通常是文件系统的一个子树。例如，它可能是一个用户的所有文件，或者是一个部门或项目中所有人员的所有文件。一个文件集是 UNIX 文件系统 (也就是用 `mkfs` 建立的单元) 的实例。在 UNIX 中，每一个磁盘分区只存放一个文件系统，而在 DFS 中则有可能存放许多文件集。

从图 10-19 中我们可以看出文件集的价值。在图 10-19(a) 中，可以看到两个分别有三个空目录的磁盘 (或磁盘分区)。随着时间的发展，系统会在这些目录中建立文件。最终如图 10-19(b) 所示，磁盘 1 的使用速度要比磁盘 2 高。如果磁盘 1 被用尽而磁盘 2 上还有充足的空间，我们会遇到问题。

DFS 的解决方案是将每一个目录 *A*、*B* 和 *C* (以及它们的子目录) 都建立成为独立的文件集。DFS 允许对文件集进行移动，所以系统管理员就可以通过将目录 *A* 移动到磁盘 2 上来对磁盘使用空间进行平衡，如图 10-19(c) 所示。只要两个磁盘都在同一信元内部，而且全局名字也没有发生变化，这样所有程序就都可以像没有发生任何变化一样工作。

除了移动文件集之外也可以对它们进行复制 (replicate)。其中一个复制品 (副本) 被设定为主备份，可以对它进行读/写操作，而其他复制品都是只读从备份。文件集，而不是磁盘分区，是移动、复制以及备份等操作的基本单元。对于 UNIX 系统而言，每一个磁盘分区都被认为是只有一个文件集的聚集。DFS 系统为系统管理员提供了许多命令来对文件集进行管理。

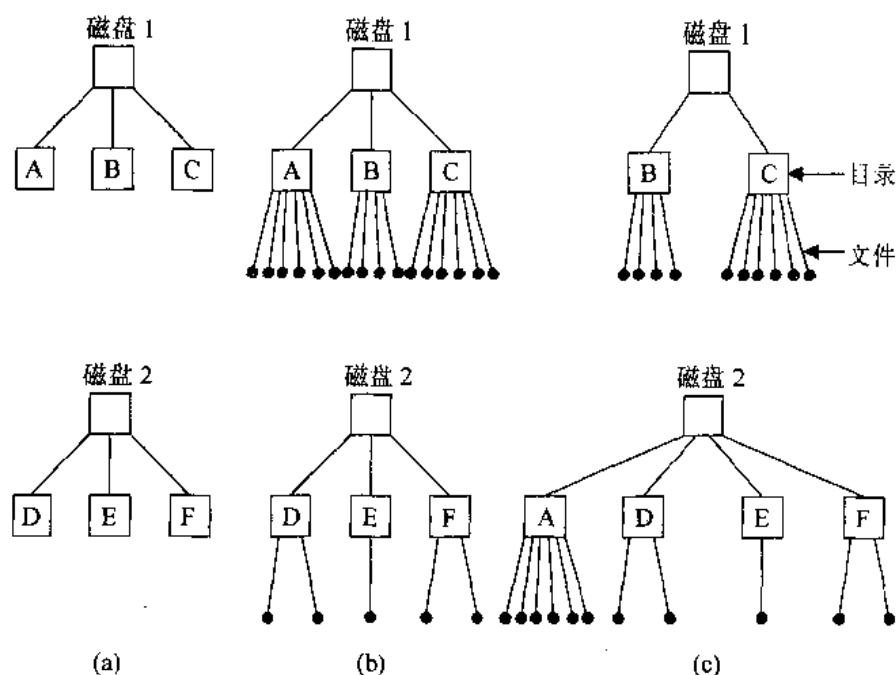


图 10-19 (a) 两个空磁盘
(b) 磁盘 1 的使用速度比磁盘 2 高
(c) 移动一个文件集后的配置

10.7.2 服务器核心中的 DFS 部件

DFS 由附加到客户和服务端内核两端的附件组成，同时也包含了许多用户进程。在本节以及接下来的一节中将介绍这个软件以及它的工作方式。图 10-20 中显示了 DFS 的关键部分。

在服务器端，列出了两个文件系统，本来的 UNIX 文件系统和 DFS 的本地文件系统，Episode。它们上面是权标管理器 (token manager)，它的主要作用是保证一致性。再上面是文件输出器，它主要用于处理同外部世界以及管理本地进程交互的系统调用接口的联系。在客户端新增加的主要是缓存管理器，它可以通过缓存文件段 (fragment) 来增强性能。

现在来看一看 Episode。正如前面所提到的，系统并不一定非要运行 Episode，但是它提供了传统文件系统所没有的一些优势。这些包括基于 ACL 的保护机制、文件集复制、快速恢复以及文件的长度最大可以达到 242 字节。当使用 UNIX 文件系统时，图 10-20 (b) 中标注为“扩展”的软件处理 UNIX 文件系统与 Episode 的接口。例如，将 PAC 和 ACL 转换成为 UNIX 的保护模型。

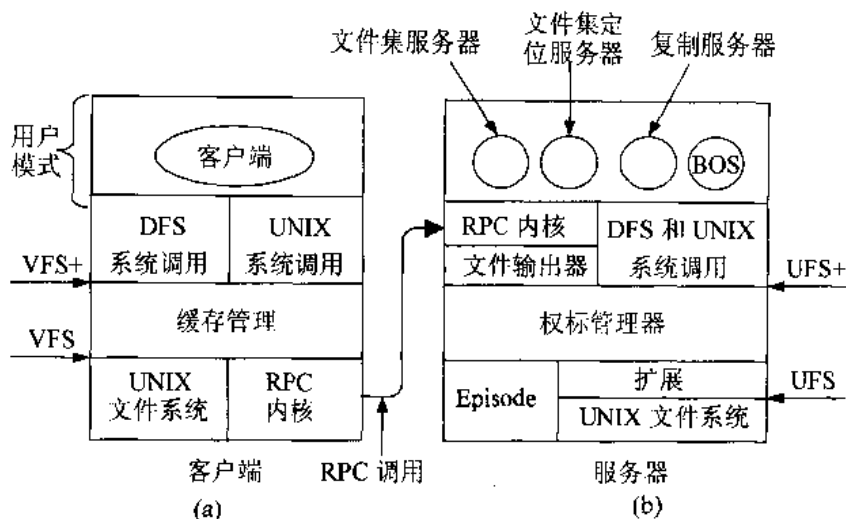


图 10-20 DFS 部件

(a) 文件客户端机器

(b) 文件服务器机器

Episode 一个有趣的特性是它可以克隆一个文件集。当执行这个操作之后，在另一个文件分区中就会有该文件集的一个虚拟拷贝，而原文件集则被标注为“只读”。例如，信元内部可能存在每天早上四点保存整个文件系统快照的策略，这样即使有人不可恢复的删除了一个文件，他总是可以恢复到昨天的版本。

Episode 实现克隆的办法是：将所有的文件系统数据结构（也就是 UNIX 系统中的 i-节点）拷贝到新的分区中去，同时再将旧的文件标注为只读。这两个文件集中的数据结构都指向同样的数据块，而实际的数据块并没有进行拷贝。这样的结果是复制工作可以迅速实现。对于原文件系统的写操作会通过返回错误消息拒绝。而对于新文件系统的写操作则可以成功进行，它会将数据块拷贝后进行修改。

Episode 的设计实现了高度并发访问。它避免了线程在关键数据结构上被长期阻塞情况的发生，这样也就减轻了需要访问同一张表的线程之间的冲突。它也设计为支持异步 I/O，在 I/O 操作结束后提供了事件通知系统。

传统的 UNIX 系统虽然没有对文件的长度进行限制，但是将大多数内部数据结构限制在固定长度的表中。Episode 则正好相反，在内部使用了一种称为 a-节点（a-node）通用存储抽象概念。系统中有专门为文件、文件集、ACL、位图（bit maps）、日志以及其他项目使用的 a-节点。在 a-节点层上面，Episode 就可以不用关心数据的物理存储（如一个很长的 ACL 就像一个长文件一样不存在任何问题）。一个 a-节点是一个 252 字节的数据结构。通过选用这个数字就可以使四个 a-节点和 16 字节的管理数据正好放入 1K 的磁盘块中。

当一个 a-节点用于表示少量数据（最多 204 字节）时，可以直接将数据保存在 a-节点内部。诸如符号链接和许多 ACL 之类的小对象都符合要求。当 a-节点用于表示大量数据结构（如一个文件）时，a-节点中保存了八个充满数据的直接块和四个指向保存更多地址的间接块的地址。

Episode 另外一个值得说明的特性是它对于崩溃恢复的处理。传统的 UNIX 系统倾向于将写入位图、i-节点以及目录的变化迅速写回磁盘以避免在系统崩溃之后出现不一致的现象。而 Episode 则将这些变化通过日志的形式写入磁盘。每一个分区都有自己的日志。每个日志条目都包含了旧值和新值。当发生崩溃之后，通过读日志来判定哪些改变已经执行而哪些还没有执行。还没有执行的改变（也就是那些由于系统崩溃而损失的）现在就可以执行。虽然这时仍然会存在对文件系统的改变丢失的现象（如果它们的日志条目还没有写回磁盘就发生了崩溃），但是文件系统还是可以在恢复之后进行修复。

这种方案的主要优势在于使用它的恢复时间与日志的长度成正比，而传统系统中用来修复磁盘的 UNIX *fsck* 程序的恢复时间则与磁盘的大小成正比。

让我们回到图 10-20，文件系统之上是权标管理器。由于权标同缓存的关系非常紧密，我们将在下一节中讨论缓存的同时讨论权标。在权标层之上是支持 Sun NFS VFS 接口的扩展接口。VFS 支持诸如安装（mount）和卸下（unmount）之类的文件系统操作，同时也支持诸如读、写以及文件改名等单个文件操作。VSP+支持这些操作和其他一些操作。VSP 和 VSP+的最大区别在于权标管理。

在权标管理之上是文件输出器。它由若干主要任务为接收和处理需要进行文件访问的输入 RPC 的线程组成。文件输出器并不仅仅处理 Episode 文件，它也可以处理所有内核中存在的文件系统。它维护着一些记录现有不同文件系统以及文件分区信息的表。它也处理客户认证、PAC 收集以及安全通道的建立。实际上它就是图 10-16 步骤 5 所说明的应用服务器。

10.7.3 客户内核中的 DFS 部件

在 DCE 中为每一个客户机器内核增加的主要部分就是 DFS 的缓存管理器。缓存管理器的主要目的是通过在内存或磁盘中缓存部分文件来增加系统的性能，同时还要维持 UNIX 系统的真正单系统文件语义。为了使大家对这个问题的本质有更进一步的了解，下面首先简要介绍一下 UNIX 语义以及为什么会存在这样的问题。

在 UNIX 系统（对于这个问题而言还包括所有单处理机操作系统）中，一个进程执行写文件操作之后，其他读文件操作将会得到刚才写入的值。而获得其他值则违反了文件系统语义。

这种语义模型是通过在操作系统中使用单一缓冲存储器实现的。当第一个进程写文件之后，修改过的块就进入缓存。如果缓存充满，那么缓存中所存储的所有文件块就会被写回磁盘。当第二个进程请求读取修改过的块时，操作系统首先会在缓存中查找这个块。如果在缓存中找不到的话，它才会在磁盘中进行查找。由于系统中只有一个缓存，所以在任何情况下都可以返回正确的文件块。

现在考虑一下 NFS 的缓存是如何工作的。多台机器可以同时打开同一文件。假设进程 1 读取了文件的一个部分并对它进行了缓存。随后进程 2 对这个文件的这个部分执行了写操作。这个写操作并不会对进程 1 运行的机器有任何影响。如果进程 1 现在希望重新读取文件的这个部分，它就将得到一个过期的值，因而违反了 UNIX 语义。这个问题就是 DFS 设计所要解决的问题之一。

这个问题在实际上比上面所说明的还要糟得多，问题主要出在目录也是可以进行缓

存的。一个进程有可能会读取一个目录并从中删除一个文件。然而在不同机器上运行的另外一个进程现在可能还可以从自己的缓存中读取这个已经被删除的文件。NFS 通过重新检查有效性来最大程度地消除这个问题，实际中仍然可能会发生错误。

DFS 使用权标解决了这个问题。为了执行任何文件操作，一个客户向缓存管理器发出请求，而缓存管理器将首先检查它是否拥有必须的权标或数据。如果它既包含权标也包含数据，那么这个操作就可以立即执行，不用再同文件服务器进行通信。如果没有权标的话，缓存管理器执行 RPC 调用向文件服务器申请权标（和数据）。一旦它获得权标之后就可以执行相应的操作。

打开文件、读写文件、锁定文件以及读写文件状态信息（如文件的所有者）等都分别有不同的权标。文件可以以读、写、读/写、执行或者互斥访问形式打开。打开以及状态权标是对全部文件有效的。读、写以及锁定权标则只对某些特定的字节区间有效。权标是由图 10-20（b）所示的权标管理器发放的。

图 10-21 给出了权标使用的一个样例。在消息 1 中，客户 1 请求一个打开某个文件执行读操作的权标，同时也请求文件第一个部分的权标（和数据）。在消息 2 中，服务器提供了这两个权标和数据。这时，客户 1 可以缓存它刚接收到的数据，同时也可以按照需要对这些数据进行读取。DFS 的普通传送单元为 64K 字节。

随后客户 2 请求打开同样文件进行读写的标志，它同时也申请最初的 64K 字节来进行有选择的覆盖性重写。服务器这时不能简单地发送这些权标，因为它已经不再拥有它们了。它必须通过给客户 1 发送消息 4 来将这些权标要回来。

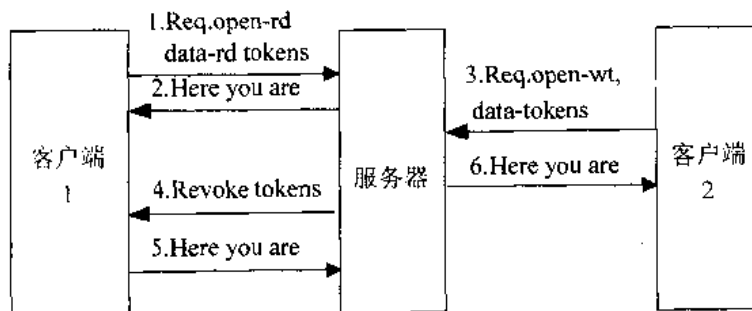


图 10-21 权标循环的例子

只要情况许可，客户 1 必须将撤销的权标返回给服务器。当服务器收到这些权标之后，它就可以将它们发布给客户 2，这时客户 2 就可以按照自己的需要对数据进行读写而不用通知服务器。如果客户 1 需要收回权标，服务器将会要求客户 2 返回权标以及修改过的数据，所以这些数据就可以被发送到客户 1。通过这种方式就可以保证单系统文件语义。

为了最大程度的提高效率，服务器必须知道权标的兼容性。虽然它不会为同样的数据同时发送两个写权标，但是如果两个客户都只需要读相同文件的话，它会为同样的数据同时发出两个读权标。

权标并不总是有效。每一个权标都有失效时间，通常是两分钟。如果一台机器崩溃，并且它不能够（或者不愿意）返回所要求的权标，文件服务器会最多等待两分钟，然后就像接收到权标一样继续执行。

从总体上来说,缓存对于用户进程来说是透明的。然而系统中还存在实现特定存储管理功能的调用,如在文件使用之前的预取、刷新缓存、磁盘限额管理等等。但是普通用户并不需要这些功能。

DFS 和它的前身 AFS 存在两个值得一提的不同点。在 AFS 中传送的是整个文件,而不是 64K 的片。这种策略要求必须有本地磁盘存在,这主要是因为整个文件通常都很大,不能够将它缓存到系统内存中。而通过传输 64K 的文件片就可以避免本地文件的存放问题,磁盘就不一定要是本地的。

另外一个差异是 AFS 文件系统代码一部分运行在内核中,而另一部分运行在用户空间。不幸的是这种方式的性能太差,所以在 DFS 中所有代码都运行在内核中。

10.7.4 用户空间中的 DFS 部件

到目前为止,已经结束了对运行在服务器和客户内核中的 DFS 部件的讨论。现在将简要介绍一下运行在用户空间的部分(参见图 10-20)。文件集服务器(fileset server)管理着所有文件集。每一个文件集都由位于一个分区的一个或多个目录及其所属文件组成。文件集可以以层次结构组织。每一个文件集都有限制它的磁盘限额。

文件集服务器允许系统管理员通过使用单个命令来建立、删除、移动、复制、克隆、备份或者恢复整个文件集。这些操作的每一个都首先锁定文件集、执行操作后释放锁定。一个文件集可以建立来为今后的使用设立管理单元。当它没有必要继续存在的时候就可以被删除。

一个文件集可以从一台机器移动到另外一台机器来进行负载平衡,这种负载平衡不仅仅是对于磁盘存储而言的,而且也是对于每秒钟所需要进行处理请求数量而言的。当对一个文件集进行了复制但并不删除它的原始备份,这样就建立了一个备份。系统支持备份,而且同时提供了负载均衡和容错。但是只有一个拷贝是可写的。

克隆就像前面所说明的那样,它只拷贝 a-节点(a-nodes)的状态信息而不是数据。但是复制则建立了一份新的数据拷贝。克隆必须同原始备份处于同一分区中,而复制的备份则可以位于任何位置(甚至在不同的信元中)。

备份与恢复允许对文件集线性化,同时将它拷贝到备份磁带或者从备份磁带中进行恢复的操作。这些磁带可以在不同的地点进行存储,这样在系统完全崩溃或者发生火灾和洪水等灾害的情况下仍然可以根据这些备份进行恢复。

文件集服务器也可以提供有关文件集自身、操作文件集限额以及其他操作的信息。

文件集定位服务器(file location server)管理着一个信元范围内的备份数据库,这个数据库中的信息可以将文件集名字映射到保存文件集的服务器名字上。如果一个文件集进行了复制,那么就可以在文件集定位服务器中找到保存这个文件集的服务器列表。

文件集定位服务器是缓存管理器用来定位文件集的。当一个用户程序第一次访问一个文件时,它的缓存管理器就会向文件集定位服务器请求这个文件所在的文件集服务器。这些信息被存放在缓存中来供今后使用。

数据库中的每一个条目都包含着文件集的名字、类型(读/写、只读或备份)、保存该文件集的服务器个数、这些服务器的地址、文件集的所有者和组信息、有关克隆、缓存超时信息、权标超时信息以及其他的管理信息。

复制服务器 (replication server) 可以保证文件集的所有备份都处于最新状态。每一个文件集都有一个主 (也就是读/写) 备份, 而其他备份都是从 (也就是只读) 备份。复制服务器定期执行, 它通过检查每一个备份的所有文件来判断这个文件自从上次更新之后是否又进行了更新。它会从主备份中拷贝那些已经又进行了更新的文件来替换相应的文件。当复制服务器运行结束之后, 所有的备份又可以保持最新了。

基本监督服务器 (basic overseer server) 运行在每一台服务器上。它的工作是保证所有其他服务器都处于正常运行状态。如果它发现一些服务器已经崩溃, 它就可以重新运行这个服务器的一个新版本。它也为系统管理员提供了一个手动启动或者中止服务器的接口。

10.8 小结

DCE 是不同于 Amoeba、Mach 和 Chorus 的另一种构造分布式系统的解决方法。DCE 并不是运行在裸机上的从头设计的新操作系统, 而是在本地操作系统之上提供了一个层。这个层将各个机器的不同点都隐藏起来, 同时还提供了公共的服务和功能程序 (facility)。从而将各种机器统一成在许多方面 (并非所有方面) 透明的单一系统。DCE 必须运行于 UNIX 和其他操作系统之上。

无论在 DCE 内部还是通过用户程序, DCE 都支持两种频繁使用的设备——线程和 RPC。线程允许在一个进程里存在多个控制流。每一个线程都有自己的程序计数器、堆栈和寄存器, 但是一个进程里的所有线程共享相同的地址空间、文件说明符和进程的其他资源。

RPC 是 DCE 中基本的通信机制。它允许一个客户进程呼叫一个远程机器上的程序。DCE 对客户选择和绑定服务器提供了很多种选项。

DCE 为客户提供四种主要服务 (和一些小服务) 可以供客户使用。它们是时间、目录、安全和文件服务。时间服务维持 DCE 系统内所有的时钟都同步在已知的限度内。时间服务的一个有趣特点是: 时间不是表示为单个值, 而是表示为间隔。因此当比较两个时间时就可能会无法明确地判断哪一个在先。

目录服务存储各类资源的名称和位置, 并允许客户查找。CDS 掌握着本地 (单元之内) 名称。GDS 掌握着全局 (单元之外) 名称。DNS 和 X.500 两种命名系统目录服务都支持。名称形成了一个层次结构。目录服务实际上就是一个复制的分布式的数据库系统。

安全服务允许客户和服务端彼此认证并执行被认证的 RPC。安全系统的核心就是一种方法, 使得客户被认证并得到 PAC, 而又不让他们的口令出现在网络上, 甚至不能以加密的形式出现。PAC 给客户提供了一种既方便又安全可靠的方法证明他们是谁。

最后, 分布式文件系统对所有的文件提供单个的系统范围内的命名空间。一个全局文件名称由一个信元 (cell) 名称后加一个本地名称组成。DCE 文件系统由 DCE 本地文件系统 Episode 加上文件输出器组成。文件输出器使所有的本地文件系统在全系统内都可见。文件使用权标方式缓存, 这种方式维护传统的单系统文件语义。

尽管 DCE 提供了许多功能程序和工具, 但它还是不完备并且可能永远都不会完备。有些地方还需要做很多工作, 比如: 说明、设计技术和工具、调试帮助、运行时间的管理、

面向对象、原子事务和多媒体支持。

习 题

- (1) 一所大学在校内所有计算机上安装了 DCE 系统。至少用两种方法将机器分成信元组。
- (2) 正如文中所述, DCE 线程能有四种状态。当处于这些状态中的两种状态: 就绪和等待时, 线程不能运行。这两种状态的不同点是什么?
- (3) 在 DCE 系统中, 一些数据结构是属于进程的, 而其他是属于每个线程的。你认为 UNIX 的环境变量是属于进程的还是属于线程的? 对你的观点进行说明。
- (4) DCE 系统中的条件变量总是与互斥联系在一起。为什么?
- (5) 一个程序只是由一段使用私有数据结构的多进程代码写成的。这个数据一次只能被一个进程使用。应该用什么样的互斥来保护它?
- (6) 命名线程模板可能会有两个相似的属性。
- (7) 每个 IDL 文件有一个唯一的号码 (例如, 由 *uuidgen* 程序生成的)。这个号码的值是什么?
- (8) 为什么 IDL 要求程序员指明哪些参数是输入的, 哪些是输出的?
- (9) 为什么 RPC 端点动态地由 RPC 后台程序指派, 而不是静态地? 一个静态指派会更简单。
- (10) 一个 DCE 程序员需要为基准程序记时。它调用本地时间过程 (在它自己的机器上) 并得到时间为 15: 30: 00.0000。然后它立即开始运行基准程序。当基准程序完成时, 它再调用时间程序, 得到时间为 15: 35: 00.0000。因为它是在同一个机器上用同一个时钟。从两个结果看, 可知基准程序用了 5 分钟 ($\pm 0.1\text{msec}$)? 说出你自己的答案。
- (11) 在一个 DCE 系统中, 时钟的不确定性为 $\pm 5\text{sec}$ 。若源文件的时间在 14: 20: 30 到 14: 20: 40 之间, 而它的二进制文件的时间在 14: 20: 32 到 14: 20: 42 之间。如果在 14: 20: 38 调用 *make* 过程, 则 *make* 过程应该做什么? 假设 *make* 过程执行要用 1sec。如果 *make* 过程在几分钟之后再次调用会怎么样?
- (12) 一个时间管理员同时向四个时间服务器发出请求。得到的回应是:
16: 00: 10.075 \pm 0.003,
16: 00: 10.085 \pm 0.003,
16: 00: 10.069 \pm 0.007,
16: 00: 10.070 \pm 0.010。
这时这个管理员应该把它的时钟设置为什么时间?
- (13) DTS 能在没有一个与任一 UTC 台机器相连系的 UTC 资源的情况下运行吗? 这样做的结果是什么?
- (14) 为什么不同的名称必须是唯一的, 但是哪些 RDNs 却不用?
- (15) 给出纽约州 Rochester 市 Kodak 公司市场部的 X.500 名称。
- (16) CDS 管理员的作用是什么? 为什么可以在没有 CDS 管理员的情况下设计 DCE 系统? 其结果是什么?

- (17) CDS 是一个可复制的数据库系统。若两个进程同时在不同的拷贝中把同一个条目改成不同值将会怎么样？
- (18) 若 DCE 系统不支持 Internet 的命名系统，则图 10-13 的各部分应该怎么分配？
- (19) 在认证过程中，一个管理员先得到一个许可证，然后得到一个 PAC。因为它们都有用户的 ID，为什么在已获得必要的许可证的情况下还要去获得 PAC？
- (20) 下面的消息有没有不同，若有则那是什么？ $\{\{message\}K_A\}K_C$ 和 $\{\{message\}K_C\}K_A$ 。
- (21) 文中的认证控制允许一个入侵者通过发送许多消息给认证服务器企图获得一个包括会话密钥的回应。把这个观察导入系统的缺点是什么？（提示：你可以假设所有的消息都包括一个时间戳和其他信息，因此很容易从任意的垃圾中找到有效的消息。）
- (22) 本文只讨论了在单个信元中的安全性问题。当客户在本地信元中而服务器在一个远地信元中时，设计一种方法来为 RPC 做认证工作。
- (23) 在 DFS 中令牌可以到期。这是否会要求同部时钟使用 DFS？
- (24) DFS 相对于 NFS 的一个优点是 DFS 保留了单文件系统的文法。指出一个缺点。
- (25) 为什么一个文件集的一个克隆必须在同一个磁盘区中？

第 11 章 读物列表与参考书目

在前 10 章中讨论了各种各样的主题。本章作为辅助部分，目的在于为那些希望对分布式操作系统作进一步研究的读者能提供一些帮助。11.1 节是推荐读物清单，11.2 节按字母顺序列出了本书中引用的所有书籍和文章的出处。

除了下面给出的参考文献之外，ACM 学会每两年举行一次的操作系统原理专题研讨会（SOSP）论文集《*Proceedings of the n-th ACM Symposium on Operating Systems Principles*》，以及每年举办的分布式计算系统（DCS）的国际会议的论文集《*Proceedings of the n-th International Conf. on Distributed Computing Systems*》都是查阅关于分布式操作系统最新论文的好去处。再有，《*ACM Transactions on Computer Systems*》和《*Operating Systems Review*》这两份期刊中常常会刊登关于分布式操作系统的有趣文章。

11.1 阅读材料建议

11.1.1 介绍性和普通著作

Andrews, *Concurrent Programming—Principles and Practice*

对并发系统程序设计进行了综合介绍。

Byte Magazine, June 1994

这是一期关于分布式计算的专辑，给出了在这个学科上面向用户的观点。其中五篇文章介绍了 DCE，分布式数据管理，安全，可执行信息系统和远程客户。

Champine et al., “Project Athena as a Distributed Computer System”

Athena 是运行在 M.I.T. 的由 1000 多台基于 UNIX 的工作站组成的网络上的网络操作系统。这个工程开发出了几个软件包。它们实际上成了标准，如 X（窗口管理）和 Kerberos（认证）。这篇论文对整个系统进行了综述。

Coulouris et al. *Distributed Systems Concepts and Design*, 2nd ed.

这是一本关于分布式系统的优秀的综合型课本。它描述了网络协议、RPC、分布式操作系统、文件系统、命名服务器、实时、复制、事务、并发控制、容错、安全和 DSM。并研究了 4 个例子：Mach, Chorus, Amoeba 和 Clouds。

Mullender, *Distributed Systems* 2nd ed.

一份暑期学校文集，包括了 21 篇在分布式系统上权威性人士的文章。内容包括：建模、技术要求、容错、实时、通信、命名、文件系统、调度和安全等。

11.1.2 分布式系统通信

Ballart and Ching, "SONET: Now It's the Standard Optical Network"

SONET 对初学者是难以理解的, 但这份辅助材料以相对轻松的方式突出描述了 SONET 的主要特征。它还涉及了标准进程的部分发展历史。

Bershad et al., "Lightweight Remote Procedure Call"

本文描述了在单处理机或多处理机上进行快速 RPC 调用的方法。文中介绍了允许客户机在服务器地址空间上运行一个预选的程序, 以避免上下文相关切换的方法。

Birman and van Renesse, "Reliable Distr.Computing with the ISIS Toolkit"

是一本全方位介绍 ISIS 的论文集。三分之一的文章介绍了 ISIS 素材, 三分之一描述了 ISIS 的理论根据, 三分之一描述了 ISIS 的应用。

Birrell and Nelson, "Implementing Remote Procedure Calls"

分布式系统的进程间通信普遍使用远程过程调用。本文描述了新颖的远程过程调用的设计与实现。

Clark et al., "The Aurora Gigabit Testbed"

Aurora 是为将来的分布式系统设计的几种千兆位实验网络中的一种。本文介绍了网络、主适配器、协议、应用和网络管理。

De Prycker, "Asynchronous Transfer Mode"

关于 ATM 的全书。在这里可以看到完整的 ATM 故事。

Hutchinson et al., "RPC in the x-Kernel: Evaluating New Design Techniques"

x-内核使用一种类似于 UNIX 流的技术, 即允许协议栈控制基于 RPC 的分层协议。采用轻量机制, 在层间使用过程调用。

Le Boudec, "The Asynchronous Transfer Modem"

本文对 ATM 作了简介, 内容包括物理 (结构)、ATM 和适应层, 讨论了将来的基于 ATM 的服务, 并给出了关于 ATM 的一些历史背景。

Mullender, "Interprocess Communication"

在这篇关于进程间通信的指南中对网络、协议和 RPC 进行了详细分析, 文章还涉及了许多的系统问题。

Tay and Ananda, "A Survey of Remote Procedure Calls"

除了某些基本的共同点之外, RPC 系统在许多方面都不同。本文概述了 8 种不同的 RPC 系统 (涉及纯学术研究项目到商业系统), 并从各个方面对它们进行了比较。

11.1.3 分布式系统同步

Fidge, "Logical Time in Distributed Computing Systems"

文章提供了用来处理基于因果关系和局部时间顺序 (而不是全局时间顺序) 的分布式系统中的事件排序的方法。

Ramanathan et al., "Fault-Tolerant Clock Synchronization in Distr.Systems"

文章综述了在分布式系统中所使用的时钟同步算法。涵盖了时钟同步的软、硬件和混合方法。

Raynal, "A Simple Taxonomy for Distributed Mutual Exclusion Algorithms"

是有关分布式互斥算法的分类学和专题书目。主要类别是基于许可权和基于权标的, 在两者的交集上产生了集中算法。

Silberschatz and Galvin, *Operating System Concepts*

本教程的第 18 章讨论了分布式系统的同步问题, 包括事件排序、互斥、一致协议和选择算法。

Singhal, "Deadlock Detection in Distributed Systems"

这是关于分布式系统中死锁检测的一份教程。首先, 它着眼于相关的问题。然后它相继讨论了分布式系统中的集中式的、非集中式的和分层算法。

Weihl, "Transaction-Processing Techniques"

文中介绍了包括嵌套事务在内的原子事务。综合了单点和分布式系统故障的恢复算法。

11.1.4 分布式系统进程和处理机

Anderson et al., "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism"

文中介绍了一种为结合用户层和内核层线程管理的最佳特性而进行的抽象概念。这种抽象概念为每个进程分配了一个虚拟多处理机, 并使用向上调用来通知用户相关的调度事件。

Burns and Wellings, *Real-time Systems and Their Programming Languages*

一本关于如何在 Ada、Modula 1 和 occam2 上进行设计和编制实时系统的介绍性教材。涉及的论题有容错、异常处理、同步、原子操作、资源控制和低层编程。本书还包括了许多代码段以例证其思想。

Cristian, "Understanding Fault-Tolerant Distributed Systems"

介绍分布式系统中的容错技术, 包括错误分类、语义、屏蔽、软、硬件问题都涉及到。硬件问题的例子包括: Tandem、Sequoia、VAX 簇和 IBM XRF。软件问题有组通信和全局一致协议。

Marsh et al., "First-Class, User-Level Threads"

介绍了关于允许在用户空间里管理线程, 但仍能利用内核知识的一系列机制和约定。此思想是基于向上调用的。

Natarajan and Zhao, "Issues in Building Dynamic Real-Time Systems"

着眼于需求, 有效性, 保证和资源管理等问题的一篇关于实时系统的简介。

Nichols, "Using Idle Workstations in a Shared Computing Environment"

描述怎样在 Butler 系统中查找并使用 UNIX 工作站。用注册表来跟踪机器并分配它们。

Shivarati et al., "Load Distributing for Locally Distributed Systems"

在分布式系统中, 因一些机器空闲而另一些过载很容易使工作失去平衡。在这篇教程中讨论了一些平衡负载的算法。

Verissimo, "Real-Time Communication"

容错实时分布式系统具有大多数其他系统所没有的特殊的通信需求。本文讨论了一些通信需求并提供了怎样实现的方法。

11.1.5 分布式文件系统

Levy and Silberschatz, "Distributed File Systems: Concepts and Examples"

本文的前一半讲解了分布式系统的原理, 后半部分是实例: UNIX、United、Locus、NFS、Sprite 和 Andrew。

Satyanarayanan, "A Survey of Distributed File Systems"

在这篇概述中分析了分布式文件系统的一些基本设计问题。对 NFS、Apollo Domain、Andrew、AIX、RFS 和 Sprite 进行了实例研究。

Satyanarayanan, "Distributed File Systems"

这篇介绍性文章讨论了分布式文件系统的原理和实践。涉及了一些现有系统: AFS、Coda 和 NFS 中普遍使用的机制, 如高速缓存, 大容量传送机制和提示机制等。

Svobodova, "File Servers for Network-Based Distributed Systems"

概述了分布式系统中的文件服务器。重点强调提供原子操作和事务的文件服务器。

11.1.6 分布式共享存储器

Li and Hudak, "Memory Coherence in Shared Virtual Memory Systems"

Li 和 Hudak 开创了 DSM 领域。本文描述了使用集中式和非集中式存储器的基于页的 DSM 系统。

Nitzberg and Lo, "Distr.Shared Memory: A Survey of Issues and Algorithms"

这是一份关于 DSM 系统的设计和实现的教程, 重点集中于一致性模式。比较了 9 个不同的例子同时还引证了 8 个例子。

Stumm and Zhou, "Algorithms Implementing Distributed Shared Memory"

一份关于分布式共享存储器的教程。

Tanenbaum et al., "Parallel Progr.Using Shared Objects and Broadcasting."

与上面处理基于页的 DSM 的参考书不同, 本文介绍了怎样在由有本地存储器的机器组成的网络上实现共享对象。

11.1.7 实例研究 1: Amoeba

Douglies et al., "A Comparison of Two Distr.Systems: Amoeba and Sprite."

比较了两个分布式系统: 有微内核和使用处理机池的 Amoeba 和具有单片内核, 使用工作站模式的 Sprite。

Kaashoek and Tanenbaum, "Group Communication in the Amoeba Distributed Operating System"

介绍了 Amoeba 中的组通信, 着重点是可靠的广播式协议的使用和实现。讨论了这种协议的容错能力和可靠的广播式协议怎样自定序器和其他失效中恢复。

Mullender et al., "Amoeba: A Distributed Operating System for the 1990s"

这是对 Amoeba 的概述，强调了通信机制、对象、安全性、文件系统和进程管理。

Tanenbaum et al., “Experiences with the Amoeba Distr.Operating System”

对 Amoeba 进行了介绍。本文强调了对象、RPC、服务器、广域 Amoeba，应用和性能。它以实际经验对设计进行了评价，并指出了系统中什么做对了，什么做错了。

11.1.8 实例研究 2: Mach

Accetta et al., “Mach: A New Kernel Foundation for UNIX Development”

这是有关 Mach 系统最早出版的论文之一。它描述了系统的目标，基本思想(如线程、端口和内存)，以及实现。

Black, “Scheduling Support for Concurrency and Parallelism in the Mach Sys.”

这里介绍了 Mach 的多处理机调度算法。讨论了许多最佳算法，例如传递(handoff)调度，并给出了性能测试。

Boykin et al., *Programming under Mach*

这是一本关于怎样写运行在 Mach 上的程序和怎样使用它的许多工具性程序的全书。重点是怎样使用 Mach 而不是知道它内部是怎样工作的。

Boykin and Langerman, “Mach/4.3BSD: A Conservative Approach to Parallelization,”

很难使 Mach UNIX 仿真器在多处理机上有效地运行，因为设计者从来没打算要它运行在多处理机上。与 I/O 和文件系统有关的问题在这里有所描述，也提供了一些解决办法。

Rashid, “From RIG to Accent to Mach: The Evolution of a Network Op.Sys.”

文章包括设计者们所写的 RIG、Accent、March 系统的发展过程。描述了系统的进展，强调了因新技术和新设计目标的产生而作的改进。

Young et al., “The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System”

介绍了 Mach 存储管理系统的目标、设计和实现以及与通信系统的相互作用。对外部存储器管理器的使用也进行了介绍。

11.1.9 实例研究 3: Chorus

Gien, “Micro-kernel Architecture: Key to Modern Operating Systems Design”

作者是 Chorus 的设计人员之一，本文介绍了 Chorus 的构想和目标。

Gien and Grob, “Micro-kernel Based Operating Systems Design”

本文讨论了 Chorus 怎样才能仿真 UNIX。

Rozier et al., “Chrous Distributed Operating Systems”

本文内容稍显陈旧，但它仍是全面介绍 Chorus 微内核体系结构方面的最好的文章。

11.1.10 实例研究 4: DCE

Bever et al., “Distributed Systems, OSF DCE and Beyond”

本文对 DCE 进行了介绍, 强调了体系结构和 RPC 接口。提出了很多 DCE 系统没有的特性 (例如: 高级工具、面向对象、分布式事务和多媒体支持) 并给出了怎样才能使它们适合 DCE 模型方法。

Kazar et al., "DEcorum File System Architecture Overview"

DEcorum 是 DCE 的分布式文件系统组件, 也是 AFS 的继承者。本文按顺序简要讨论了令牌、高速缓存、复制、死锁等内容。

OSF, *Introduction to OSF DCE*

最容易得到的有关 DCE 的材料, 涉及与本书一样的论题。

Rosenberry et al., *Understanding DCE*

对 DCE 进行了全面介绍, 用了 11 个章节和 4 个附录覆盖了 DCE 的全部基本概念。

Shirley, *Guide to Writing DCE Applications*

是一份讲述如何为 DCE 系统的客户机和服务器编程的教材。并给出了很多代码段作为例子。

11.2 文献目录 (按字母顺序排列)

ABROSSIMOV, A., ARMAND, F., and ORTEGA, M.: "A Distributed Consistency Server for the CHORUS System," *Proc. SEDMS III, Symp. on Experience with Distributed and Multiprocessor Systems*, USENIX, pp.129-148, 1992.

ABROSSIMOV, A., ROZIER, M., and SHAPIRO, M.: "Generic Virtual Memory Management in Operating Systems Kernels," *Proc. 12th Symp. on Operating Systems Principles*, ACM, pp.123-136, 1989.

ACCETTA, M., BARON, R., GOLUB, D., RASHID, R., TEVANI, A., and YOUNG, M.: "MACH: A New Kernel Foundation for UNIX Development," *Proc. Summer 1986 USENIX Conf.*, USENIX, pp.93-112, 1986.

ADVE, S., and HILL, M.: "Weak Ordering: A New Definition," *Proc. 17th Ann. Int'l Symp. on Computer Architecture*, ACM, pp.2-14, 1990.

AGARWAL, A., CHAIKEN, D., D'SOUZA, G., JOHNSON, K., KRANZ, D., KUBIATOWICZ, J., KURIHARA, K., LIM, B., MAA, G., NUSSBAUM, D., PARKIN, M., and YEUNG, D.: "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor," *Proc. Workshop on Scalable Shared Memory Multiprocessors*, Kluwer, 1991.

AGARWAL, A., and CHERIAN, M.: "Adaptive Backoff Synchronization Techniques," *Proc. 16th Ann. Int'l Symp. on Computer Architecture*, ACM, pp. 396-406, 1989.

AGARWAL, A., SIMONI, R., HENNESSY, J., and HOROWITZ, M.: "An Evaluation of Directory Schemes for Cache Coherence," *Proc. 15th Ann. Int'l Symp. on computer Architecture*, ACM, pp.280-289, 1988.

AGRAWAL, D., and EL ABBADI, A.: "An Efficient and Fault-Tolerant Solution of Distributed Mutual Exclusion," *ACM Trans. on Computer Systems*, vol.9, pp.1-20, Feb.1991.

AHAMAD, M., BAZZI, R.A., JOHN, R., KOHLJ, P., and NEIGER, G.: "The Power of Processor Consistency," Tech. Rep.GIT-CC-92/34, College of Computing, Georgia Inst.of Technology, March 1993.

AHMADI, H., and DENZEL, W.: "A Survey of Modern High - Performance Switching Techniques," *IEEE Journal of Selected Areas in Communication*, vol.7, pp.1091-1103, Sept.1989.

ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D., and LEVY, H.M.: "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Proc.13th Symp.on Operating Systems Principles*, ACM, PP.95-109, 1991.

ANDERSON, T.E., OWICKI, S.S., SAXE, J.B., and THACKER, C.P.: "High-Speed Switch Scheduling for Local-Area Networks," *ACM Trans. on Computer Systems*, vol.11, pp.319-352, Nov.1993.

ANDREWS, G.R.: *Concurrent Programming-Principles and Practice*, Redwood City, CA: Benjamin/Cummings, 1991.

ARCHIBALD, J., and BAER, J.-L.: "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. on Computer Systems*, vol.4, pp.273-298, Nov.1986.

ARMAND, F., and DEAN, R.: "Data Movement in Kernelized Systems," *Proc. USENIX Workshop on Microkernels and Other Kernel Architectures*, USENIX, pp.243-261, 1992.

ARTSY, Y., and FINKEL, R.: "Designing a Process Migration Facility," *IEEE Computer*, vol.22, pp.47-56, Sept.1989.

ATTIYA, H., and FRIEDMAN, R.: "A Correctness Condition for High Performance Multiprocessors," *Proc. 24th ACM Symp. on Theory of Computing*, ACM, pp.679-690, 1992.

BAL, H.E.: *Programming Distributed Systems*, Hemel Hempstead, England: Prentice Hall Int'l, 1991.

BAL, H.E., and KAASHOEK, M.F.: "Object Distribution in Orca using Compile-time and Run-time Techniques," *Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications(OOPSLA '93)*, ACM, PP.162-177, Sept.1993.

BAL, H.E., KAASHOEK, M.F., and TANENBAUM, A.S.: "Experience with Distributed Programming in Orca," *Proc. Int'l Conf. on Computer Languages '90*, IEEE, pp.78-89, 1990.

BAL, H.E., KAASHOEK, M.F., and TANENBAUM, A.S.: "Orca : A Language for Parallel Programming of Distributed Systems," *IEEE Trans. on Software Engineering*, vol.18, pp.190-205, March 1992.

BALL, J.E., FELDMAN, J.A., LOW, J.R., RASHID, R.F., and ROVNER, P.D.: "RIG, Rochester's Intelligent Gateway: System Overview," *IEEE Trans. on Software Engineering*, vol. SE-2, pp.321-328, Dec.1976.

BALLART, R., and CHING, Y.-Y.: "SONET: Now It's the Standard Optical Network," *IEEE Communications Magazine*, vol.29, pp.8-15, March 1989.

BARBORAK, M., MALEK, M., and DAHBURA, A.: "The Consensus Problem in Fault-Tolerant Computing," *ACM Computing Surveys*, vol.25, pp.171-220, June 1993.

BARON, R., RASHID, R., SIEGEL, E., TEVANIAN, A., and YOUNG, M.: "Mach-1: An Operating Environment for Large-Scale Multiprocessor Applications," *IEEE Software*, vol.2, pp.65-67, July 1985.

BATLIVALA, N., GLEESON, B., HAMRICK, J., LURN DAL, S., PRICE, D., SODDY, J., and ABROSSIMOV, V.: "Experience with SVR4 Over CHORUS," *Proc. USENIX Workshop on Microkernels and Other Kernel Architectures*, USENIX, pp.223-241, 1992.

BELLOVIN, S.M., and MERRITT, M.: "Limitations of the Kerberos Authentication System," *Proc. Winter 1991 USENIX Conf.*, USENIX, Jan.1991.

BENNETT, J.K., CARTER, J.K., and ZWAENEPOEL, W.: "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. Second ACM Symp. on Principles and Practice of Parallel Programming*, ACM, pp.168-176, 1990.

BERNSTEIN, P.A., and GOODMAN, N.: "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Trans. on Database Systems*, vol.9, pp.596-615, Dec.1984.

BERSHAD, B.N., ANDERSON, T.E., LAZOWSKA, E.D., and LEVY, H.M.: "Lightweight Remote Procedure Call," *ACM Trans. on Computer Systems*, Vol.8, pp.37-55, Feb.1990.

BERSHAD, B.N., and ZEKAUSKAS, M.J.: "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors," CMU Report CMU-CS-91-170, Sept.1991.

BERSHAD, B.N., ZEKAUSKAS, M.J., and SAWDON, W.A.: "The Midway Distributed Shared Memory System," *Proc. IEEE COMPCON Conf.*, IEEE, pp.528-537, 1993.

BEVER, M., GEIHS, K., HEUSER, L., MUHLHAUSER, M., and SCHILL, A.: "Distributed Systems, OSF DCE, and Beyond," in *DCE - The OSF Distributed Computing Environment*, A.Schill(ed.), Berlin: Springer-Verlag, pp.1-20, 1993.

BIRMAN, K.P.: "The Process Group Approach to Reliable Distributed Computing," *Commun. of the ACM*, vol.36, pp.36-53, Dec.1993.

BIRMAN, K.P., and JOSEPH, T.: "Reliable Communication in the Presence of Failures," *ACM Trans. on Computer Systems*, vol.5, pp.47-76, Feb.1987a.

BIRMAN, K.P., and JOSEPH, T.: "Exploiting Virtual Synchrony in Distributed Systems," *Proc. 11th Symp.on Operating Systems Principles*, ACM, pp.123-138, Nov.1987b.

BIRMAN, K.P., SCHIPER, A., and STEPHENSON, P.: "Lightweight Causal and Atomic Group Multicast," *ACM Trans. on Computer Systems*, vol.9, pp.272-314, Aug.1991.

BIRMAN, K.P., and VAN RENESSE, R.: *Reliable Distributed Computing with the ISIS Toolkit*, LosAlamitos, CA: IEEE Computer Society Press, 1994.

BIRRELL, A.D., and NELSON, B.J.: "Implementing Remote Procedure Calls," *ACM Trsns. On Computer Systems*, vol.2, pp.39-59, Feb.1984.

BITAR, P.: "MIMD Synchronization and Coherence," Tech.Rep.90/605, Univ. of Calif. at Berkeley, Nov.1990.

BJORNSON, R.D.: "Linda on Distributed Memory Multiprocessors," Ph.D.Thesis, Yale Univ., 1993.

BLACK, D.: "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *IEEE Computer*, vol.23, pp.35-43, May 1990.

BLACK, D.L., GOLUB, D.B., JULIN, D.P., RASHID, R.F., DRAVES, R.P., DEAN, R.W., FORIN, A., BARRERA, J., TOKUDA, H., MALAN, G., and BOHMAN, D.: "Microkernel Operating System Architecture and Mach," *Proc. USENIX Workshop on Microkernels and Other Kernel Architectures*, USENIX, pp.11-30, 1992.

BOLOSKY, W.J., FITZGERALD, R.P., and SCOTT, M.L.: "Simple but Effective Techniques for NUMA Memory Management," *Proc.12th Symp. on Operating Systems Principles*, ACM, pp.19-31, 1989.

BOYKIN, J., KIRSCHEN, D., LANGERMAN, A., and LoVERSO, S.: *Programming Under Mach*, Reading, MA: Addison-Wesley, 1993.

BOYKIN, J., and LANGERMAN, A.: "Mach/4.3BSD: A Conservative Approach to Parallelization," *Computing Systems*, vol.3., pp.69-99, Winter 1990.

BRERETON, O.P.: "Management of Replicated Files in a UNIX Environment," *Software—Practice and Experience*, vol.16, pp.771-780, Aug.1986.

BRICKER, A., GIEN, M., GULLEMONT, M., LIPKIS, J., ORR, D., and ROZIER, M.: "A New Look at Microkernel-based UNIX operating systems: Lessons in performance and

compatibility, "Proc.EurOpen Spring '91 Conf., EurOpen, pp.13-32, 1991.

BUDHIRAJA, N., MARZULLO, K., SCHNEIDER, F.B., and TOUEG, S.: The Primary—Backup Approach, "in *Distributed Systems*, 2nd ed., S.Mullender(ed.), New York, NY: ACM Press, pp.199-216, 1993.

BURNS, A., and WELLINGS, A.: *Real-Time Systems and Their Programming Languages*, Reading, MA: Addison-Wesley, 1990.

CARRIERO, N., and GELERNTER, D.: "The S/Net's Linda Kernel, " *ACM Trans. on Computer Systems*, vol.4, pp.110-129, May 1986.

CARRIERO, N., and GELERNTER, D.: "Linda in Context, " *Commun. of the ACM*, vol.32, pp.444-458. April 1989.

CARRIERO, N., GELERNTER, D., and LEICHTER, J.: "Distributed Data Structures in Linda, " *Proc. ACM Symposium on Principles of Programming Languages*, ACM, 1986.

CARTER, J.B., BENNETT, J.K., and ZWAENEPOEL, W.: "Implementation and Performance of Munin, " *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp.152-164, 1991.

CARTER, J.B., BENNETT, J.K., and ZWAENEPOEL, W.: "Techniques for Reducing Consistency-Related Communication in Distributed Shared memory Systems, " *ACM Trans. on Computer Systems*, vol.12, 1994.

CATLETT, C.E.: "In Search of Gigabit Applications, " *IEEE Communications Magazine*, vol.30, pp.42-51, April 1992.

CHAMPINE, G.A., GEER, D.E., Jr., and RUH, W.N.: "Project Athena as a Distributed Computer System, " *IEEE Computer*, vol.23, pp.40-51, Sept.1990.

CHANDY, K.M., MISRA, J., and HAAS, L.M.: "Distributed Deadlock Detection, " *ACM Trans. on Computer Systems*, vol.1, pp.144-156, May 1983.

CHANG, J., and MAXEMCHUK, N.F.: "Reliable Broadcast Protocols, " *ACM Trans. on Computer Systems*, vol.2, pp.39-59, Feb.1984.

CHASE, J.S., AMADOR, F.G., LAZOWSKA, E.D., LEVY, H.M., and LITTLEFIELD, R.J.: "The Amber System: Parallel Programming on a Network of Multiprocessors, " *Proc. 12th Symp. on Operating Systems Principles*, ACM, pp.147-158, 1989.

CHEONG, H., and VEIDENBAUM, A.V.: "A Cache Coherence Scheme with Fast Selective

Invalidation, "Proc. 15th Ann. Int'l Symp. on Computer Architecture, ACM, pp.299-307, 1988.

CHEUNG, N.K.: "The Infrastructure of Gigabit Computer Networks, "IEEE Communications Magazine, vol.30, pp.60-68, April 1992.

CHOW, T.C.K., and ABRAHAM, J.A.: "Load Balancing in Distributed Systems, "IEEE Trans. on Software Engineering, vol.SE-8, pp.401-412, July 1982.

CHUTANI, S., ANDERSON, O.T., KAZAR, M.L., LEVERETT, B.W., MASON, W.A., and SIDEBOTHAM, R.N.: "The Episode File System, "Proc.Winter 1992 USENIX Conf., USENIX, pp.43-60, 1992.

CLARK, D.D., DAVIE, B.S., FARBER, D.J., GOPAL, I.S., KADABA, B.K., SINOSKIE, W.D., SMITH, J.M., and TENNENHOUSE, D.L.: "The Aurora Gigabit Testbed, "Computer Networks and ISDN Systems, vol.25, pp.599-621, June 1993.

COHEN, D.: "On Holy Wars and a Plea for Peace, "IEEE Computer, vol.14, pp.48-54, Oct.1981.

COMER, D.E.: *Internetworking with TCP/IP.Vol.1: Principles, Protocols and Architectures*, 2nd ed., Englewood Cliffs, NJ: Prentice Hall , 1991.

COULOURIS, G.F., DOLLIMORE, J., and KINDBERG, T.: *Distributed Systems Concepts and Design*, 2nd ed.Reading, MA: Addison-Wesley, 1994.

COX, A.L., and FOWLER, R.J.: "The Implementation of a Coherent Memory Abstraction of a UNMA Multiprocessor: Experiences with PLATINUM", 12th Symp. on Operating Systems Principles, ACM, pp.32-43, 1989.

CRISTIAN, F.: "Probabilistic Clock Synchronization, "Distributed Computing, vol.3, pp.146-158, 1989.

CRISTIAN, F.: "Understanding Fault-Tolerant Distributed Systems, "Commun. of the ACM, vol.34, pp.56-78, Feb.1991.

DAHLGREN, F., DUBOIS, M., and STENSTROM, P.: "Combined Performance Gains of Simple Cache Protocol Extensions, "Proc. 21st Ann. Int'l Symp. on Computer Architecture, ACM, pp.187-195.

DAY, J.D., and ZIMMERMAN, H.: "The OSI Reference Model, "Proc.IEEE, vol.71, pp.1334 -1340, Dec.1983.

DE PRYCKER, M.: *Asynchronous Transfer Mode*, Chichester, England: Ellis Horwood, 1991.

DELP, G.S.: *The Architecture and Implementation of MemNet.An Experiment on High-Speed Memory Mapped Network Interface*, Ph.D.Thesis, Univ.of Delaware, 1988.

DELP, G.S., FARBER, D.J., MINNICH, R.G., SMITH, J.M., and TAM, M.-C.: "Memory as a Network Abstraction, " *IEEE Network*, vol.5, pp.34-41, July 1991.

DOUGLIS, F., and OUSTERHOUT, J.: "Transparent Process Migration: Design Alternatives and the Sprite Implementation, " *Software—Practice and Experience*, vol.21, pp.757-785, Aug.1991.

DOUGLIS, F., OUSTERHOUT, J.K., KAASHOEK, M.F., and TANENBAUM, A.S.: "A Comparison of Two Distributed Systems: Amoeba and Sprite, " *Computing Systems*, vol.4, pp.353-384, Fall 1991.

DRAVES, R.P.: "The Revised IPC Interface, " *Proc. First USENIX Mach Symp.*, USENIX, pp.101-121, 1990.

DRAVES, R.P., BERSHAD, B.N., RASHID, R.F., and DEAN, R.W.: "Using Continuations to Implement Thread Management and Communication in Operating Systems, " *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp.122-136, 1991.

DRUMMOND, R., and BABAOGLU, O.: "Low-Cost Clock Synchronization, " *Distributed Computing*, vol.6, pp.193-203, 1993.

DUBOIS, M., SCHEURICH, C., and BRIGGS, F.A.: "Memory Access Buffering in Multiprocessors, " *Proc. 13th Ann. Int'l Symp. on Computer Architecture*, ACM, pp.434-442, 1986.

DUBOIS, M., SCHEURICH, C., and BRIGGS, F.A.: "Synchronization, Coherence, and Event Ordering in Multiprocessors, " *IEEE Computer*, vol.21, pp.9-21, Feb.1988.

EAGER, D.L., LAZOWSKA, E.D., and ZAHORJAN, J.: "Adaptive Load Sharing in Homogeneous Distributed Systems, " *IEEE Trans. on Software Engineering*, vol.SE-12, pp.662-675, May 1986.

ECKBERG, A.E.: "B-ISDN/ATM Traffic Control and Congestion, " *IEEE Network*, vol.5, pp.28-37, Sept.1992.

EDLER, J., LIPKIS, J., and SCHONBERG, E.: "Process Management for Highly Parallel UNIX Systems, " *Proc. USENIX Workshop on UNIX and Supercomputers*, USENIX, pp.1-17, Sept.1988.

EGGERS, S.J., and KATZ, R.H.: "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs, " *Proc. Second ASPLOS Conf.*, ACM, pp.257-271, 1989a.

EGGERS, S.J., and KATZ, R.H.: "Evaluating the Performance of Four Snooping Cache Coherency Protocols, " *Proc. 16th Ann. Int'l Symp. on Computer Architecture*, ACM, pp.2-15, 1989b.

ESWARAN, K.P., GRAY, J.N., LORIE, J.N., and TRAIGER, I.L.: "The Notions of Consistency — 430 —

and Predicate Locks in a Database System, "Commun. of the ACM, vol.19, pp.624-633, Nov.1976.

EVANS, A., KANTROWITZ, W., and WEISS, E.: "A User Authentication Scheme Not Requiring Secrecy in the Computer, "Commun. of the ACM, vol 7., pp.437-442, Aug.1974.

FERGUSON, D., YEMINI, Y., and NIKOLAOU, C.: "Microeconomic Algorithms for Load Balancing in Distributed Computer Systems, "Proc. Eighth Int'l Conf. on Distributed Computing Systems, IEEE, pp.491-499, 1988.

FIDGE, C.: "Logical Time in Distributed Computing Systems, "IEEE Computer, vol.24, pp.28-33, Aug.1991.

FISCHER, M., LYNCH, N., and PATERSON, M.: "Impossibility of Distributed Consensus with One Faulty Process, "Journal of the ACM, vol. 32, pp. 374-382, April 1985.

FLEISCH, B., and POPEK, G.: "Mirage: A Coherent Distributed Shared Memory Design, "Proc. 12th Symp. on Operating Systems Principles, ACM, pp.211-223, 1989.

FLYNN, M.J.: "Some Computer Organizations and Their Effectiveness, "IEEE Trans. on Computers, vol.C-21, pp.948-960, Sept.1972.

FORIN, A., BARRERA, J., YOUNG, M., and RASHID, R.: "Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach, "Proc. Winter 1989 USENIX Conf., USENIX, Jan.1989.

FREDRICKSON, N., and LYNCH, N.: "Electing a Leader in a Synchronous Ring, " Journal of the ACM, vol.34, pp.98-115, Jan.1987.

GANTENBEIN, R.E.: "An Annotated Bibliography of Dependable Distributed Computing," Operating Systems Review, vol.20, pp.60-81, April 1992.

GARCIA-MOLINA, H.: "Elections in a Distributed Computing System, "IEEE Trans. on Computers, vol.31, pp.48-59, Jan.1982.

GARCIA-MOLINA, H., and SPAUSTER, A.: "Ordered and Reliable Multicast Communication, "ACM Trans. on Comp. Syst., vol.9, pp.242-271, Aug.1991.

GELERTER, D.: "Generative Communication in Linda, "ACM Trans. on Programming Languages and Systems, vol.7, pp.80-112, Jan.1985.

GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., and HENNESSY, J.: "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, "Proc.

17th Ann. Int'l Symp. on Computer Architecture, ACM, pp.15-26, 1990.

GIEN, M.: "Micro-Kernel Architecture: Key to Modern Operating Systems Design, " *UNIX Review*, pp.10, Nov.1990.

GIEN, M., and GROB, L.: "Microkernel Based Operating Systems: Moving UNIX onto Modern System Architectures, " *Proc. UniForum'92 Conf.*, USENIX, pp.43-55, 1992.

GIFFORD, D.K.: "Weighted Voting for Replicated Data," *Proc. Seventh Symp. on Operating Systems Principles*, ACM, pp.150-162, 1979.

GOLUB, D., DEAM, R., FORIN, A., and RASHID, R.: "UNIX as an Application Program, " *Proc. Summer 1990 USENIX Conf.*, USENIX, pp.87-95, June 1990.

GOODMAN, J.R.: "Using Cache Memory to Reduce Processor Memory Traffic, " *10th Ann. Int'l Symp. on Computer Architecture*, ACM, pp.124-131, 1983.

GOODMAN, J.R.: "Cache Consistency and Sequential Consistency, " *Tech. Rep. 61*, IEEE Scalable Coherent Interface Working Group, IEEE, 1989.

GOPAL, I., GUERIN, R., JANNIELLO, J., and THEOHARAKIS, V.: "ATM Support in a Transparent Network, " *IEEE Network*, vol.6, pp.62-68, Nov.1992.

GRAY, J.: "Notes on Database Operating Systems, " in *Operating Systems: An Advanced Course*, R. Bayer, R.M. Graham, and G. Seegmüller (eds.), Berlin: Springer-Verlag, pp.394-481, 1978.

GRAY, C., and CHERITON, D.: "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File System Consistency, " *Proc. 11th Symp. on Operating Systems Principles*, ACM, p.202-210, 1989.

GARY, J.N., HOMAN, P., KORTH, H.F., and OBERMARCK, R.L.: "A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System, " Report RJ 3066, IBM Research Laboratory, San Jose CA, 1981.

GUSELLA, R., and ZATTI, S.: "The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD, " *IEEE Trans. on Software Engineering*, vol.15, pp.847-853, July 1989.

HARTY, K., and CHERITON, D.: "Application-Controlled Physical Memory Using External Page-Cache Management, " *Proc. Fifth ASPLOS Conf.*, ACM, pp.187-199, 1992.

HOARE, C.A.R.: "Monitors, An Operating Systems Structuring Concept, " *Commun. of the ACM*, vol.17, pp.549-557, Oct.1974; Erratum in *Commun. of the ACM*, vol.18, pp.95, Feb.1975.

HONG, D., and SUDA, T.: "Congestion Control and Prevention in ATM Networks, " *IEEE Network*,

vol.5, pp.10-16, July 1991.

HOWARE, J.H., KAZAR, M.J., MENEES, S.G., NICHOLS, D.A., SATYANARAYANAN, M., SIDEBOTHAM, R.N., and WEST, M.J.: "Scale and Performance in a Distributed File System," *ACM Trans. on Computer Systems*, vol.6, pp.55-81, Feb.1988.

HUTCHINSON, N.C., PETERSON, L.L., ABBOTT, M.B., and O'MALLEY, S.: "RPC in the x-Kernel: Evaluating New Design Techniques," *Proc.12th Symp.on Operating Systems Principles*, ACM, pp.91-101, 1989.

HUTTO, P.W., and AHAMAD, M.: "Slow Memory : Weakening Consistency to Enhance Concurrency in Distributed Shared Memories," *Proc. 10th Int'l Conf. on Distributed Computing Systems*, IEEE, pp.302-311, 1990.

JONES, A.K., CHANSLER, R.J., Jr., DURHAM, I., FELER, P., and SCHWANS, K.: "SoftwareManagement of CM*—A Distributed Multiprocessor," *Proc. NCC, AFIPS*, pp.657-663, 1977.

JUL, E., LEVY, H., HUTCHINSON, N., and BLACK, A.: "Fine-Grained Mobility in the Emerald System," *ACM Trans. on Computer Systems*, vol.6, pp.109-133, Feb.1988.

KAASHOEK, M.F., and TANENBAUM, A.S.: "Group Communication in the Amoeba Distributed Operating System," *Proc. 11th Int'l Conf. on Distributed Computing Systems*, IEEE, pp.222-230, 1991.

KAASHOEK, M.F., TANENBAUM, A.S., HUMMEL, S., and BAL, H.E.: "An Efficient Reliable Broadcast Protocol," *Operating Systems Review*, vol.23, pp.5-19, Oct.1989.

KARLIN, A.R., LI, K., MANASSE, M, S., and OWICKI, S.: "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor," *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp.41-55, 1991.

KAZAR, M.L., LEVERETT, B.W., ANDERSON, O.T., APOSTOLIDES, V., BOTTOS, B.A., CHUTANI, S., EVERHART, C.F., MASON, W.A., TU, S.-T., and ZAYAS, E.R.: "DEcorum File System Architectural Overview," *Proc. Summer 1990 USENIX Conf.*, USENIX, pp.151-163, Summer 1990.

KELEHER, P., COX, A.L., and ZWAENEPOEL, W.: "Lazy Release Consistency," *Proc. 19th Ann. Int'l Symp. on Computer Architecture*, ACM, PP.13-21, 1992.

KIEIN, M.H., LEHOCZKY, J.P., and RAJKUMAR, R.: "Rate-Monotonic Analysis for Real-Time Industrial Computing," *IEEE Computer*, vol.27, pp.24-33, Jan.1994.

KLEINROCK, L.: *Queueing Systems*.vol.1, New York: John Wiley, 1974.

KLEINROCK, L.: "The Latency/Bandwidth Tradeoff in Gigabit Networks," *IEEE Communications*

Magazine, vol.30, pp.36-40, April 1992.

KNAPP, E.: "Deadlock Detection in Distributed Databases," *ACM Computing Surveys*, vol.19, pp.303-328, Dec.1987.

KOHL, J.T.: "The Evolution of the *Kerberos* Authentication Service," *Proc.EurOpen Spring '91 Conf.*, Eur Open, pp.295-313, 1991.

KOPETZ, H., DAMM, A., KOZA, C., MULAZZANI, M., SCHWABL, W., SENFT, C., and ZAINLINGER, R.: "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach," *IEEE Micro*, vol.9, pp.25-40, Feb.1989.

KOPETZ, H., and GRUNSTEIDL, G.: "TTP-A Protocol for Fault-Tolerant Real-Time Systems," *IEEE Computer*, vol.27, pp.14-23, Jan 1994.

KOPETZ, H., and OCHSENREITER, W.: "Clock Synchronization in Distributed Real-Time Systems," *IEEE Trans. on Computers*, vol.C-36, pp.933-940, Aug.1987.

KRANZ, D., JOHNSON, K., AGARWAL, A., KUBIATOWICZ, J.J., and LIM, B.: "Integrating Message Passing and Shared Memory: Early Experiences," *Proc. Fourth Symp. on Principles and Practice of Parallel Programming*, ACM, pp.54-63, May 1993.

KRISHNASWAMY, V.: "A Language Based Architecture for Parallel Computing," Ph.D.Thesis, Yale Univ., 1991.

KUNG, H.T., and ROBINSON, J.T.: "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems*, vol.6, pp.213-226, June 1981.

LAMPORT, L.: "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. of the ACM*, vol.21, pp.558-564, July 1978.

LAMPORT, L.: "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. on Computers*, vol.C-28, pp.690-691, Sept.1979.

LAMPORT, L.: "Concurrent Reading and Writing of Clocks," *ACM Trans. on Computer Systems*, vol.8, pp.305-310, Nov.1990.

LAMPORT, L., SHOSTAK, R., and PEASE, M.: "The Byzantine Generals Problem," *ACM Trans. on Programming Languages and Systems*, vol.4, pp.382-401, July 1982.

LAMPSON, B.W., ABADI, M., BURROWS, M., and WOBBER, E.: "Authentication in Distributed Systems: Theory and Practice," *ACM Trans. on Computer Systems*, vol.10, pp.265-310, Nov.1992.

LAROWE, R.P., and ELLIS, C.S.: "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors, " *ACM Trans. on Computer Systems*, vol.9, pp.319-363, Nov.1991.

LAROWE, R.P., ELLIS, C.S., and KAPLAN, L.S.: "The Robustness of UNMA Memory Management, " *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp.137-151, 1991.

LE BOUDEC, J.-Y.: "The Asynchronous Transfer Mode: A Tutorial" *Computer Networks and ISDN Systems*, vol.24, pp.279-309, April 1992.

LEA, R., AMARAL, P., and JACQUEMOT, C.: "COOL-2 : An Object-Orient Support Platform Built above the Chorus Microkernel, " *Proc. Int'l Workshop on Object-Oriented Systems*, pp.51-55, 1991.

LEA, R., JACQUEMOT, C., and PILLEVESSE, E.: "COOL: System Support for Distributed Programming, " *Commun. of the ACM*, vol.36, pp.37-46, Sept.1993.

LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W.-D., GUPTA, A., HENNESSY, J., HOROWITZ, M., and LAM, M.: "The Stanford Dash Multiprocessor, " *IEEE Computer*, vol.25, pp.63-79, March 1992.

LEVY, E., and SILBERSCHATZ, A.: "Distributed File Systems: Concepts and Examples" *Computing Surveys*, vol.22, pp.321-374, Dec.1990.

LI, K.: "Shared Virtual Memory on Loosely Coupled Multiprocessors, " Ph.D.Thesis, Yale Univ., 1986.

LI, K., and HUDAK, P.: "Memory Coherence in Shared Virtual Memory Systems, " *ACM Trans. on Computer Systems*, vol.7, pp.321-359, Nov.1989.

LILJA, D.J.: "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons, " *ACM Computing Surveys*, vol.25, pp.303-338, Sept.1993.

LIPTON, R.J., and SANDBERG, J.S.: "Pram: A Scalable Shared Memory, " Tech. Rep.CS-TR- 180-88, Princeton Univ., Sept.1988.

LISKOV, B.: "Practical Uses of Synchronized Clocks in Distributed Systems, " *Distributed Computing*, vol.6, pp.211-219, 1993.

LITZKOW, M.J., LIVNY, M., and MUTKA, M.W.: "Condor-A Hunter of Idle Workstations," *Proc. Eighth Int'l Conf. on Distributed Computing Systems*, IEEE, pp.104-111, 1988.

LIU, C. L., and LAYLAND, J.W.: "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, " *Journal of the ACM*, vol.20, pp.46-61, Jan.1973.

LO, V.M.: "Heuristic Algorithms for Task Assignment in Distributed Systems, " *Proc. Fourth Int'l Conf.*

on Distributed Computing Systems, IEEE, pp.30-39, 1984.

LUAN, S.-W., and GLIGOR, V.D.: "A Fault-Tolerant Protocol for Atomic Broadcast, " *IEEE Trans. on Parallel and Distributed Systems*, vol.1, pp.271-285, July 1990.

LUNDELIUS-WELCH, J., and LYNCH, N.: "A New Fault-Tolerant Algorithm for Clock Synchronization, " *Information and Computation*, vol.77, pp.1-36, Jan.1988.

LYLES, J.B., and SWINEHART, D.C.: "The Emerging Gigabit Environment and the Role of Local ATM, " *IEEE Communications Magazine*, vol.30, pp.52-58, April 1992.

MAEKAWA, M., OLDEHOEFT, A.E., and OLDEHOEFT, R.R.: *Operating Systems: Advanced Concepts*, Menlo Park, CA: Benjamin/Cummings, 1987.

MALCOLM, N., and ZHAO, W.: "The Timed-Token Protocol for Real-Time Communication, " *IEEE Computer*, vol.27, pp.35-41, Jan 1994.

MARSH, B.D., SCOTT, M.L., LEBLAMC, T.J., and MARKATOS, E.P.: "First-Class User-Level Threads, " *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp.110-121, 1991.

MELIAR-SMITH, P.M., MOSER, L.E., and AGRAWALA, V.: "Broadcast Protocols for Distributed Systems, " *IEEE Trans. on Parallel and Distributed Systems*, vol.1, pp.17-25, Jan.1990.

MINZER, S.E.: "Broadband ISDN and Asynchronous Transfer Mode (ATM), " *IEEE Communications Magazine*, vol.29, pp.17-24, Sept.1989.

MORRIS, J.H., SATYANARAYANAN, M., CONNER, M.H., HOWARD, J.H., ROSENTHAL, D.S., and SMITH, F.D.: "Andrew: A Distributed Personal Computing Environment, " *Commun. of the ACM*, vol.29, pp.184-201, March 1986.

MOSBERGER, D.: "Memory Consistency Models, " *Tech. Report TR 93/11*, Dept. of Computer Science, Univ. of Arizona, 1993.

MULLENDER, S.J. (ed.): *Distributed Systems*, 2nd ed., New York, NY: ACM Press, 1993.

MULLENDER, S.J.: "Interprocess Communication, " in *Distributed Systems*, 2nd ed., S.Mullender(ed.), New York, NY: ACM Press, pp.217-250, 1993.

MULLENDER, S.J., ROSSUM, G.VAN, TANENBAUM, A.S., RENESSE, R.VAN, and STAVEREN, H.VAN: "Amoeba: A Distributed Operating System for the 1990s, " *IEEE Computer*, vol.23, pp.44-53, May 1990.

MULLENDER, S.J., and TANENBAUM, A.S.: "Immediate Files, " *Software—Practice and*

Experience, vol.14, pp.365-368, April 1984.

MUTKA, M.W., and LIVNY, M.: "Scheduling Remote Processor Capacity in a Workstation-Processor Bank Network, " *Proc. Seventh Int'l Conf. on Distributed Computing Systems*, IEEE, pp.2-9, 1987.

NATARAJAN, S., and ZHAO, W.: "Issues in Building Dynamic Real-Time Systems, " *IEEE Software*, vol.9, pp.16-21, Sept.1992.

NAYFEH, B.A., and OLUKOTUN, K.: "Exploring the Design Space for a SharedCache Multiprocessor, " *Proc. 21st Ann. Int'l Symp. on Computer Architecture*, ACM, pp.166-175, 1994.

NEEDHAM, R.M., and SCHROEDER, M.D.: "Using Encryption for Authentication in Large Networks of Computers, " *Commun. of the ACM*, vol.21, pp.993-999, Dec.1978.

NELSON, B.J.: *Remote Procedure Call*, Ph.D.Thesis, Carnegie-Mellon Univ., 1981.

NELSON, V.P.: "Fault-Tolerant Computing: Fundamental Concepts, " *IEEE Computer*, vol.23, pp.19-25, July 1990.

NEWMAN, P.: "ATM Local Area Networks, " *IEEE Communications Magazine*, vol.32, pp.86-98, March 1994.

NICHOLS, D.A.: "Using Idle Workstations in a Shared Computing Environment, " *Proc. 11th Symp. on Operating Systems Principles*, ACM, pp.5-12, 1987.

NIKOLAIDIS, I., and ONVURAL, R.O.: "A Bibliography on Performance Issues in ATM Networks, " *Computer Communication Review*, vol.22, pp.8-23, Oct.1992.

NITZBERG, B., and LO, V.: "Distributed Shared Memory: A Survey of Issues and Algorithms, " *IEEE Computer*, vol.24, pp.52-60, Aug.1991.

OSF: *Introduction to OSF DCE*, Englewood Cliffs, NJ: Prentice Hall, 1992.

OUSTERHOUT, J.K.: "Scheduling Techniques for Concurrent Systems, " *Proc. Third Int'l Conf. on Distributed Computing Systems*, IEEE, pp.22-30, 1982.

PANZIERI, F., and SHRIVASTAVA, S.K.: "Rajdoot: a remote procedure call mechanism with orphan detection and killing, " *IEEE Trans. on Software Engineering*, vol.14, pp.30-37, Jan.1988.

PARNAS, D.: "On the Criteria to Be Used in Decomposing Systems into Modules, " *Commun. of the ACM*, vol.15, pp.1053-1058, Dec.1972.

PARTRIDGE, C.: "Protocols for High-Speed Networks: Some Questions and a Few Answers, " *Computer*

Networks and ISDN Systems, vol.25, pp.1019-1028, Sept.1993.

PARTRIDGE, C.: *Gigabit Networking*, Reading, MA: Addison-Wesley, 1994.

PATTAVINA, A.: "Nonblocking Architectures for ATM Switching, " *IEEE Communications Magazine*, vol.31, pp.38-48, Feb.1993.

PEASE, M., SHOSTAK, R., and LAMPORT, L.: "Reaching Agreement in the Presence of Faults, " *Journal of the ACM*, vol.27, pp.228-234, April 1980.

PRZYBYLSKI, M., HOROWITZ, J., and HENNESSY, J.: "Performance Tradeoffs in Cache Design, " *Proc. 15th Ann. Int'l Symp. on Computer Architecture*, ACM, pp.290-298, 1988.

PU, C., NOE, J.D., and PROUDFOOT, A.: "Regeneration of Replicated Objects: A Technique and its Eden Implementation, " *Proc. Second Int'l Conf. on Data Engineering*, pp.175-187, Feb 1986.

PURDIN, T.D., SCHLICHTING, R.D., and ANDREWS, G.R.: "A File Replication Facility for Berkeley UNIX, " *Software—Practice and Experience*, vol.17, pp.923-940, Dec.1987.

RAMAMRITHAM, K., STANKOVIC, J.A., and SHIAH, P-F.: "Efficient Scheduling Algorithms and Real-time Multiprocessor Systems, " *IEEE Trans. on Parallel and Distributed Systems*, vol.1, pp.184-194, April 1990.

RAMANATHAN, J., and NI, L.M.: "Critical Factors in NUMA Memory Management, " *Proc. 11th Int'l Conf. on Distributed Computing Systems*, IEEE, pp.500-507, 1991.

RAMANATHAN, P., KANDLUR, D.D., and SHIN, K.G.: "Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems, " *IEEE Trans. on Computers*, vol.C-39, pp.514-524, April 1990a.

RAMANATHAN, P., and SHIN, K.G.: "Delivery of Time-Critical Messages Using a Multiple Copy Approach, " *ACM Trans. on Computer Systems*, vol.10, pp.144-166, May 1992.

RAMANATHAN, P., SHIN, K.G., and BUTLER, R.W.: "Fault-Tolerant Clock Synchronization in Distributed Systems, " *IEEE Computer*, vol.23, pp.33-42, Oct.1990b.

RASHID, R.F.: "Threads of a New System, " *Unix Review*, vol. 4, pp.37-49, Aug.1986a.

RASHID, R.F.: "From RIG to Accent to Mach: The Evolution of a Network Operating System, " *Fall Joint Computer Conf.*, AFIPS, pp.1128-1137, 1986b.

RAYNAL, M.: "A Simple Taxonomy for Distributed Mutual Exclusion Algorithms, " *Operating Systems Review*, vol.25, pp.47-50, April 1991.

REED, D.P.: "Implementing Atomic Actions on Decentralized Data, " *ACM Trans. on Computer Systems*, vol.1, pp.3-23, Feb.1983.

RICART, G., and AGRAWALA, A.K.: "An Optimal Algorithm for Mutual Exclusion in Computer Networks, " *Commun. of the ACM*, vol.24, pp.9-17, Jan. 1981.

ROOHOLAMINI, R., CHERKASSKY, V., and GARVER, M.: "Finding the Right ATM Switch for the Market, " *IEEE Computer*, vol.27, pp.16-28, April 1994.

ROSENBERRY, W., KENNEY, D., and FISHER, G.: *Understanding DCE*, Sebastopol, CA: O'Reilly, 1992.

ROZIER, M., ABROSSIMOV, V., ARMAND, F., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMANN, F., KAISER, C., LEONARD, P., LANGLOIS, S., and NEUHAUSER, W.: "Chorus Distributed Operating Systems, " *Computing Systems*, vol.1, pp.305-379, Oct.1988.

SANDERS, B.A.: "The Information Structure of Distributed Mutual Exclusion, " *ACM Trans. on Computer Systems*, vol.5, pp.284-299, Aug.1987.

SANSOM, R.D., JULIN, D.P., and RASHID, R.F.: "Extending a Capability Based System into a Network Environment, " *Proc. SIGCOMM'86*, ACM, pp.265-274, 1986.

SATYANARAYANAN, M.: "A Study of File Sizes and Functional Lifetimes, " *Proc. Eighth Symp. on Operating Systems Principles*, ACM, pp.96-108, 1981.

SATYANARAYANAN, M.: "A Survey of Distributed File Systems, " *Annual Review of Computer Science*, vol.4, pp.73-104, 1990a.

SATYANARAYANAN, M.: "Scalable, Secure, and Highly Available Distributed File Access, " *IEEE Computer*, vol.23, pp.9-21, May 1990b.

SATYANARAYANAN, M.: "Distributed File Systems, " in *Distributed Systems*, 2nd ed., S.Mullender(ed.), New York, NY: ACM Press, pp.353-383, 1993.

SCHEURICH, C., and DUBOIS, M.: "Correct Memory Operation of Cache-Based Multiprocessors, " *Proc. Fourth Ann. Int'l Symp. on Computer Architecture*, ACM, pp.234-24., 1987.

SCHNEIDER, F.B.: "Implementing Fault-Tolerant Services Using the State Machine Approach, " *ACM Computing Surveys*, vol.22, pp.299-319, Dec.1990.

SCHROEDER, M.D., and BURROWS, M.: "Performance of Firefly RPC, " *ACM Trans. on Computer Systems*, vol.8, pp.1-17, Feb.1990.

SCHWAN, K., and ZHOU, H.: "Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads, " *IEEE Trans. on Software Engineering*, vol.18, pp.736-747, Aug.1992.

SCOTT, M., LEBLANC, T., and MARSH, B.: "Multi-model Parallel Programming in Psyche, " *Proc. Second ACM Symp. on Principles and Practice of Parallel Programming*, ACM, pp.70-78, 1990.

SHIN, K.: "HARTS: A Distributed Real-Time Architecture, " *IEEE Computer*, vol.24, pp.25-35, May1991.

SHIRLEY, J.: *Guide to Writing DCE Applications*, Sebastopol, CA: O'Reilly, 1992.

SHIVARATI, N.G., KRUEGER, P., and SINGHAL, M.: "Load Distributing for Locally Distributed Systems, " *IEEE Computer*, vol.25, pp.33-44, Dec 1992.

SILBERSCHATZ, A., and GALVIN, P.: *Operating System Concepts*, Reading, MA: Addison-Wesley, 1994.

SINGH, S., and KUROSE, J.: "Electing 'Good' Leaders, " *Journal of Parallel and Distributed Computing*, vol.21, pp.184-201, May 1994.

SINGHAL, M.: "Deadlock Detection in Distributed Systems, " *IEEE Computer*, vol.22, pp.37-48, Nov.1989.

SRIKANTH, T.K., and TOUEG, S.: "Optimal Clock Synchronization, " *Journal of the ACM*, vol.34, pp.626-645, July 1987.

STANKOVIC, J.A.: "Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems, " *IEEE Computer*, vol.21, pp.10-19, Oct.1988.

STEINER, J.G., NEUMAN, B.C., and SCHILLER, J.I.: "Kerberos: An Authentication Service for Open Network Systems, " *Proc. Winter 1988 USENIX Conf.*, USENIX, pp.191-202, Feb 1988.

STONE, H.S., and BOKHARI, S.H.: "Control of Distributed Processes, " *IEEE Computer*, vol.11, pp.97-106, July 1978.

STUMM, M., and ZHOU, S.: "Algorithms Implementing Distributed Shared Memory, " *IEEE Computer*, vol.23, pp.54-64, May 1990.

SUBRAMANIAN, I.: "Managing Discardable Pages with an External Pager, " *Proc. Second USENIX Mach Symp.*, USENIX, pp.77-86, 1991.

SUZUKI, T.: "ATM Adaptation Layer Protocol, " *IEEE Communications Magazine*, vol.32, pp.80-83, April 1994.

SVOBODOVA, L.: "File Servers for Network-Based Distributed Systems, " *ACM Computing Surveys*, vol.16, pp.353-398, Dec.1984.

TAM, M.-C., SMITH, J.M., and FARBER, D.J.: "A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems, " *Operating Systems Review*, vol.24, pp.40-67, July 1990.

TANENBAUM, A.S.: *Computer Networks*, 2nd ed., Englewood Cliffs, NJ: Prentice Hall, 1988.

TANENBAUM, A. S., KAASHOEK, M.F., and BAL, H.E.: "Parallel Programming Using Shared Objects and Broadcasting, " *IEEE Computer*, vol.25, 1992.

TANENBAUM, A.S., MULLENDER, S.J., and VAN RENESSE, R.: "Using Sparse Capabilities in a Distributed Operating System, " *Proc. Sixth Int'l Conf. on Distributed Computing Systems*, IEEE, pp.558-563, 1986.

TANENBAUM, A.S., VAN RENESSE, R., STAVEREN, H.VAN, SHARP, G.J., MULLENDER, S.J., JANSEN, J., and ROSSUM, G.VAN: "Experiences with the Amoeba Distributed Operating System, " *Commun. of the ACM*, vol.33, pp.46-63, Dec.1990.

TAY, B.H., and ANANDA, A.L.: "A Survey of Remote Procedure Calls, " *Operating Systems Review*, vol.24, pp.68-79, July 1990.

THEIMER, M.M., LANTZ, K.A., and CHERITON, D.A.: "Preemptable Remote Execution Facilities in the V System, " *Proc. 10th Symp. on Operating Systems Principles*, ACM, pp.2-12, 1985.

THEKKATH, R., and EGGERS, S.J.: "Impact of Sharing-Based Thread Placement on Multithreaded Architectures, " *Proc. 21st Ann. Int'l Symp. on Computer Architecture*, ACM, pp. 176-186, 1994.

TRAJKOVIC, L., and GOLESTANI, S.J.: "Congestion Control for Multimedia Services, " *IEEE Network*, vol.6, pp.20-26, Sept.1992.

TSEUNG, L.N.: "Guaranteed, Reliable, Secure Broadcast Networks, " *IEEE Network*, vol.3, pp.33-37, Nov.1989.

TUREK, J., and SHASHA, D.: "The Many Faces of Consensus in Distributed Systems, " *IEEE Computer*, vol.25, pp.8-17, June 1992.

ULLMAN, J.: "Complexity of Sequence Problems, " in *Computers and Job/Shop Scheduling Theory*, E.G.Coffman(ed.), New York: Wiley, 1976.

VAN RENESSE, R., and TANENBAUM, A.S.: "Voting with Ghosts, " *Proc. Eighth Int'l Conf. on Distributed Computer Systems*, IEEE, 1988.



VAN TILBORG, A.M., and WITTIE, L.D.: "Wave Scheduling: Distributed Allocation of Task Forces in Network Computers," *Proc. Sixth Int'l Conf. on Distributed Computing Systems*, IEEE, pp.337-347, 1981.

VASWANI, R., and ZAHORJAN, J.: "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared Memory Multiprocessors," *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp.26-40, 1991.

VERISSIMO, P.: "Real-Time Communication," in *Distributed Systems*, 2nd ed., S.Mullender (ed.), New York, NY: ACM Press, pp.447-490, 1993.

VERNON, M.K., LAZOWSKA, E.D., and ZAHORJAN, J.: "Snooping Cache-Consistency Protocols," *Proc. 15th Ann. Int'l Symp. on Computer Architecture*, ACM, pp.308-317, 1988.

WEBER, W., and GUPTA, A.: "Analysis of Cache Invalidation patterns in Multiprocessors," *Proc. Third ASPLOS Conf.*, ACM, pp.243-256, 1989.

WEIHL, W.: "Transaction-Processing Techniques," in *Distributed Systems*, 2nd ed., S. Mullender(ed.), New York, NY: ACM Press, pp.329-352, 1993.

WITTIE, L.D., and VAN TILBORG, A.M.: "MICROS, a Distributed Operating Systems for MICRONET, A Reconfigurable Network Computer," *IEEE Trans. on Computers*, vol.C-29, pp.1133-1144, Dec.1980.

WOBBER, E., ABADI, M., BURROWS, M., and LAMPSON, B.: "Authentication in the Taos Operating System," *ACM Trans. on Computer Systems*, vol.12, pp.3-32, Feb.1994.

WOO, T.Y.C., and LAM, S.S.: "Authentication for Distributed Systems," *IEEE Computer*, vol.25, pp.39-52, Jan.1992.

YOUNG, M., TEVANI, A.Jr., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J., BOLOSKY, W., BLACK, D., and BARON, R.: "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proc. 11th Symp. on Operating Systems Principles*, ACM, pp. 63-76, Nov. 1987.

ZAYAS, E.R.: "Attacking the Process Migration Bottleneck," *Proc. 11th Symp. on Operating Systems Principles*, ACM, pp.13-24, 1987.

ZEGURA, E.W.: "Architectures for ATM Switching Systems," *IEEE Communications Magazine*, vol.31, pp.28-37, Feb.1993.